wolfSSL Documentation

2025-06-06

Contents

1 Introduction **17**
1.1 Why Choose wolfSSL? . . . 17

2 Building wolfSSL **18**
2.1 Getting wolfSSL Source Code . . . 18
2.2 Building on Unix-like Systems . . . 18
2.3 Building on Windows . . . 19
2.3.1 VS 2008 . . . 19
2.3.2 VS 2010 . . . 19
2.3.3 VS 2013 (64 bit solution) . . . 19
2.3.4 Cygwin . . . 19
2.4 Building in a non-standard environment . . . 20
2.4.1 Building into Yocto Linux . . . 21
2.4.2 Building with Atollic TrueSTUDIO . . . 21
2.4.3 Building with IAR . . . 22
2.4.4 Building on OS X and iOS . . . 22
2.4.5 Building with GCC ARM . . . 23
2.4.6 Building on Keil MDK-ARM . . . 23
2.5 Features Defined as C Pre-processor Macro . . . 24
2.5.1 Removing Features . . . 24
2.5.2 Enabling Features macros (on by default) . . . 27
2.5.3 Enabling Features Disabled by Default . . . 28
2.5.4 Customizing or Porting wolfSSL . . . 42
2.5.5 Reducing Memory or Code Usage . . . 44
2.5.6 Increasing Performance . . . 46
2.5.7 GCM Performance Tuning . . . 47
2.5.8 wolfSSL's Math Options . . . 47
2.5.9 Stack or Chip Specific Defines . . . 51
2.5.10 OS Specific Defines . . . 54
2.6 Build Options . . . 55
2.6.1 `--enable-debug` . . . 55
2.6.2 `--enable-distro` . . . 55
2.6.3 `--enable-singlethread` . . . 55
2.6.4 `--enable-dtls` . . . 55
2.6.5 `--disable-rng` . . . 55
2.6.6 `--enable-sctp` . . . 55
2.6.7 `--enable-openssh` . . . 55
2.6.8 `--enable-apachehttpd` . . . 55
2.6.9 `--enable-openvpn` . . . 55
2.6.10 `--enable-opensslextra` . . . 56
2.6.11 `--enable-opensslall` . . . 56
2.6.12 `--enable-maxstrength` . . . 56
2.6.13 `--disable-harden` . . . 56
2.6.14 `--enable-ipv6` . . . 56
2.6.15 `--enable-bump` . . . 56
2.6.16 `--enable-leanpsk` . . . 56
2.6.17 `--enable-leantls` . . . 57
2.6.18 `--enable-bigcache` . . . 57
2.6.19 `--enable-hugecache` . . . 57
2.6.20 `--enable-smallcache` . . . 57
2.6.21 `--enable-savesession` . . . 57

COPYRIGHT ©2024 wolfSSL Inc.

2.6.22 --enable-savecert . . . 57
2.6.23 --enable-atomicuser . . . 57
2.6.24 --enable-pkcallbacks . . . 57
2.6.25 --enable-sniffer . . . 58
2.6.26 --enable-aesgcm . . . 58
2.6.27 --enable-aesccm . . . 58
2.6.28 --disable-aescbc . . . 58
2.6.29 --enable-aescfb . . . 58
2.6.30 --enable-aesctr . . . 58
2.6.31 --enable-aesni . . . 58
2.6.32 --enable-intelasm . . . 59
2.6.33 --enable-camellia . . . 59
2.6.34 --enable-md2 . . . 59
2.6.35 --enable-nullcipher . . . 59
2.6.36 --enable-ripemd . . . 59
2.6.37 --enable-blake2 . . . 59
2.6.38 --enable-blake2s . . . 59
2.6.39 --enable-sha3 . . . 59
2.6.40 --enable-sha512 . . . 59
2.6.41 --enable-sessioncerts . . . 59
2.6.42 --enable-keygen . . . 59
2.6.43 --enable-certgen . . . 60
2.6.44 --enable-certext . . . 60
2.6.45 --enable-certreq . . . 60
2.6.46 --enable-sep . . . 60
2.6.47 --enable-hkdf . . . 60
2.6.48 --enable-x963kdf . . . 60
2.6.49 --enable-dsa . . . 60
2.6.50 --enable-eccshamir . . . 60
2.6.51 --enable-ecc . . . 60
2.6.52 --enable-ecccustcurves . . . 60
2.6.53 --enable-compkey . . . 60
2.6.54 --enable-curve25519 . . . 61
2.6.55 --enable-ed25519 . . . 61
2.6.56 --enable-fpecc . . . 61
2.6.57 --enable-eccencrypt . . . 61
2.6.58 --enable-psk . . . 61
2.6.59 --disable-errorstrings . . . 61
2.6.60 --disable-oldtls . . . 61
2.6.61 --enable-sslv3 . . . 61
2.6.62 --enable-stacksize . . . 61
2.6.63 --disable-memory . . . 61
2.6.64 --disable-rsa . . . 61
2.6.65 --enable-rsapss . . . 62
2.6.66 --disable-dh . . . 62
2.6.67 --enable-anon . . . 62
2.6.68 --disable-asn . . . 62
2.6.69 --disable-aes . . . 62
2.6.70 --disable-coding . . . 62
2.6.71 --enable-base64encode . . . 62
2.6.72 --disable-des3 . . . 62
2.6.73 --enable-arc4 . . . 62
2.6.74 --disable-md5 . . . 62
2.6.75 --disable-sha . . . 62

2.6.76 --enable-webserver 62
2.6.77 --enable-fips 63
2.6.78 --enable-sha224 63
2.6.79 --disable-poly1305 63
2.6.80 --disable-chacha 63
2.6.81 --disable-hashdrbg 63
2.6.82 --disable-filesystem 63
2.6.83 --disable-inline 63
2.6.84 --enable-ocsp 63
2.6.85 --enable-ocspstapling 63
2.6.86 --enable-ocspstapling2 63
2.6.87 --enable-crl 63
2.6.88 --enable-crl-monitor 64
2.6.89 --enable-sni 64
2.6.90 --enable-maxfragment 64
2.6.91 --enable-alpn 64
2.6.92 --enable-truncatedhmac 64
2.6.93 --enable-renegotiation-indication 64
2.6.94 --enable-secure-renegotiation 64
2.6.95 --enable-supportedcurves 64
2.6.96 --enable-session-ticket 64
2.6.97 --enable-extended-master 64
2.6.98 --enable-tlsx 65
2.6.99 --enable-pkcs7 65
2.6.100--enable-pkcs11 65
2.6.101--enable-ssh 65
2.6.102--enable-scep 65
2.6.103--enable-srp 65
2.6.104--enable-smallstack 65
2.6.105--enable-valgrind 65
2.6.106--enable-testcert 65
2.6.107--enable-iopool 65
2.6.108--enable-certservice 65
2.6.109--enable-jni 66
2.6.110--enable-lighty 66
2.6.111--enable-stunnel 66
2.6.112--enable-md4 66
2.6.113--enable-pwdbased 66
2.6.114--enable-scrypt 66
2.6.115--enable-cryptonly 66
2.6.116--disable-examples 66
2.6.117--disable-crypttests 66
2.6.118--enable-fast-rsa 66
2.6.119--enable-staticmemory 67
2.6.120--enable-mcapi 67
2.6.121--enable-asynccrypt 67
2.6.122--enable-sessionexport 67
2.6.123--enable-aeskeywrap 67
2.6.124--enable-jobserver 67
2.6.125--enable-shared[=PKGS] 67
2.6.126--enable-static[=PKGS] 67
2.6.127--with-liboqs=PATH 67
2.6.128--with-libz=PATH 67
2.6.129--with-cavium 68

2.6.130 --with-user-crypto . . . 68
2.6.131 --enable-rsavfy . . . 68
2.6.132 --enable-rsapub . . . 68
2.6.133 --enable-armasm . . . 68
2.6.134 --disable-tlsv12 . . . 68
2.6.135 --enable-tls13 . . . 68
2.6.136 --enable-all . . . 68
2.6.137 --enable-xts . . . 68
2.6.138 --enable-asio . . . 68
2.6.139 --enable-qt . . . 69
2.6.140 --enable-qt-test . . . 69
2.6.141 --enable-apache-httpd . . . 69
2.6.142 --enable-afalg . . . 69
2.6.143 --enable-devcrypto . . . 69
2.6.144 --enable-mcast . . . 69
2.6.145 --disable-pkcs12 . . . 69
2.6.146 --enable-fallback-scsv . . . 69
2.6.147 --enable-psk-one-id . . . 69
2.6.148 --enable-cryptocb . . . 70
2.6.149 --enable-reproducible-build . . . 70
2.6.150 --enable-sys-ca-certs . . . 70
2.7 Special Math Optimization Flags . . . 70
2.7.1 --enable-fastmath . . . 70
2.7.2 --enable-fasthugemath . . . 70
2.7.3 --enable-sp-math . . . 70
2.7.4 --enable-sp-math-all . . . 71
2.7.5 --enable-sp-asm . . . 71
2.7.6 --enable-sp=OPT . . . 71
2.8 Cross Compiling . . . 75
2.8.1 Example cross compile configure options for toolchain builds . . . 75
2.9 Building Ports . . . 77
2.10 Building For NXP CAAM . . . 78
2.10.1 i.MX8 (Linux) . . . 78
2.10.2 i.MX8 (QNX) . . . 88
2.10.3 i.MX6 (QNX) . . . 88
2.10.4 IMXRT1170 (FreeRTOS) . . . 88

3 Getting Started **89**
3.1 General Description . . . 89
3.2 Testsuite . . . 89
3.3 Client Example . . . 91
3.4 Server Example . . . 93
3.5 EchoServer Example . . . 94
3.6 EchoClient Example . . . 95
3.7 Benchmark . . . 95
3.7.1 Relative Performance . . . 97
3.7.2 Benchmarking Notes . . . 97
3.7.3 Benchmarking on Embedded Systems . . . 99
3.8 Changing a Client Application to Use wolfSSL . . . 100
3.9 Changing a Server Application to Use wolfSSL . . . 101

4 Features **103**
4.1 Features Overview . . . 103
4.2 Protocol Support . . . 103

4.2.1 Server Functions . . . 103
4.2.2 Client Functions . . . 103
4.2.3 Robust Client and Server Downgrade . . . 104
4.2.4 IPv6 Support . . . 104
4.2.5 DTLS . . . 104
4.2.6 LwIP (Lightweight Internet Protocol) . . . 104
4.2.7 TLS Extensions . . . 105
4.3 Cipher Support . . . 105
4.3.1 Cipher Suite Strength and Choosing Proper Key Sizes . . . 105
4.3.2 Supported Cipher Suites . . . 106
4.3.3 AEAD Suites . . . 109
4.3.4 Block and Stream Ciphers . . . 109
4.3.5 Hashing Functions . . . 110
4.3.6 Public Key Options . . . 110
4.3.7 ECC Support . . . 110
4.3.8 PKCS Support . . . 111
4.3.9 Forcing the Use of a Specific Cipher . . . 112
4.3.10 OpenQuantumSafe's liboqs Integration . . . 112
4.4 Hardware Accelerated Crypto . . . 113
4.4.1 AES-NI . . . 113
4.4.2 STM32F2 . . . 113
4.4.3 Cavium NITROX . . . 113
4.4.4 ESP32-WROOM-32 . . . 114
4.4.5 ESP8266 . . . 114
4.4.6 ERF32 . . . 114
4.4.7 MAX32665/MAX32666 . . . 114
4.5 SSL Inspection (Sniffer) . . . 114
4.6 Static Buffer Allocation Option . . . 116
4.6.1 Basic Operation of Static Buffer Allocation . . . 116
4.6.2 Specifying Static Buffer Use . . . 116
4.6.3 Enabling Static Buffer Allocation . . . 118
4.6.4 Using Static Buffer Allocation . . . 118
4.6.5 Adjustment of Static Buffer Allocation . . . 120
4.7 Compression . . . 122
4.8 Pre-Shared Keys . . . 122
4.9 Client Authentication . . . 123
4.10 Server Name Indication . . . 123
4.11 Handshake Modifications . . . 124
4.11.1 Grouping Handshake Messages . . . 124
4.12 Truncated HMAC . . . 124
4.13 User Crypto Module . . . 124
4.14 Timing-Resistance in wolfSSL . . . 125
4.15 Fixed ABI . . . 125

5 Portability 127
5.1 Abstraction Layers . . . 127
5.1.1 C Standard Library Abstraction Layer . . . 127
5.1.2 Custom Input/Output Abstraction Layer . . . 128
5.1.3 Operating System Abstraction Layer . . . 128
5.2 Supported Operating Systems . . . 128
5.3 Supported Chipmakers . . . 129
5.4 C# Wrapper . . . 129

6 Callbacks 131

COPYRIGHT ©2024 wolfSSL Inc.

6.1 HandShake Callback . . . 131
6.2 Timeout Callback . . . 131
6.3 User Atomic Record Layer Processing . . . 132
6.4 Public Key Callbacks . . . 133
6.4.1 DH Callbacks . . . 133
6.4.2 Ed25519 Callbacks . . . 133
6.4.3 X25519 Callbacks . . . 134
6.4.4 Ed448 Callbacks . . . 134
6.4.5 X448 Callbacks . . . 135
6.4.6 RSA PSS Callbacks . . . 135
6.4.7 ECC Callbacks . . . 136
6.5 Crypto Callbacks (cryptocb) . . . 138
6.5.1 Using Crypto callbacks . . . 138
6.5.2 Writing your crypto callback . . . 139
6.5.3 Handling the request . . . 141
6.5.4 Troubleshooting . . . 141
6.5.5 Examples . . . 141

7 Keys and Certificates 142
7.1 Supported Formats and Sizes . . . 142
7.2 Supported Certificate Extensions . . . 142
7.3 Certificate Loading . . . 144
7.3.1 Loading CA Certificates** . . . 144
7.3.2 Loading Client or Server Certificates . . . 144
7.3.3 Loading Private Keys . . . 144
7.3.4 Loading Trusted Peer Certificates . . . 145
7.4 Certificate Chain Verification . . . 145
7.5 Domain Name Check for Server Certificates . . . 145
7.6 No File System and using Certificates . . . 146
7.6.1 Test Certificate and Key Buffers . . . 146
7.7 Serial Number Retrieval . . . 146
7.8 RSA Key Generation . . . 146
7.8.1 RSA Key Generation Notes . . . 147
7.9 Certificate Generation . . . 148
7.10 Certificate Signing Request (CSR) Generation . . . 150
7.10.1 Limitations . . . 151
7.11 Convert to raw ECC key . . . 151
7.11.1 Example . . . 152

8 Debugging 153
8.1 Debugging and Logging . . . 153
8.2 Error Codes . . . 153

9 Library Design 154
9.1 Library Headers . . . 154
9.2 Startup and Exit . . . 154
9.3 Structure Usage . . . 154
9.4 Thread Safety . . . 154
9.5 Input and Output Buffers . . . 155

10 wolfCrypt Usage Reference 156
10.1 Hash Functions . . . 156
10.1.1 MD4 . . . 156
10.1.2 MD5 . . . 156
10.1.3 SHA / SHA-224 / SHA-256 / SHA-384 / SHA-512 . . . 157

COPYRIGHT ©2024 wolfSSL Inc.

10.1.4 BLAKE2b . . . 157
10.1.5 RIPEMD-160 . . . 158
10.2 Keyed Hash Functions . . . 158
10.2.1 HMAC . . . 158
10.2.2 GMAC . . . 158
10.2.3 Poly1305 . . . 159
10.3 Block Ciphers . . . 159
10.3.1 AES . . . 159
10.4 Stream Ciphers . . . 160
10.4.1 ARC4 . . . 160
10.4.2 ChaCha . . . 161
10.5 Public Key Cryptography . . . 161
10.5.1 RSA . . . 161
10.5.2 DH (Diffie-Hellman) . . . 162
10.5.3 EDH (Ephemeral Diffie-Hellman) . . . 163
10.5.4 DSA (Digital Signature Algorithm) . . . 163

11 SSL Tutorial 165
11.1 Introduction . . . 165
11.1.1 Examples Used in this Tutorial . . . 165
11.2 Quick Summary of SSL/TLS . . . 165
11.3 Getting the Source Code . . . 165
11.4 Base Example Modifications . . . 166
11.4.1 Modifications to the echoserver (tcpserv04.c) . . . 166
11.4.2 Modifications to the echoclient (tcpcli01.c) . . . 166
11.4.3 Modifications to unp.h header . . . 166
11.5 Building and Installing wolfSSL . . . 167
11.6 Initial Compilation . . . 169
11.7 Libraries . . . 169
11.8 Headers . . . 169
11.9 Startup/Shutdown . . . 170
11.10 WOLFSSL_CTX Factory . . . 170
11.11 WOLFSSL Object . . . 172
11.11.1 EchoClient . . . 172
11.11.2 EchoServer . . . 172
11.12 Sending/Receiving Data . . . 173
11.12.1 Sending with EchoClient . . . 173
11.12.2 Receiving with EchoServer . . . 173
11.13 Signal Handling . . . 174
11.13.1 Echoclient / Echoserver . . . 174
11.14 Certificates . . . 175
11.15 Conclusion . . . 175

12 Best Practices for Embedded Devices 177
12.1 Creating Private Keys . . . 177
12.2 Digitally Signing and Authenticating with wolfSSL . . . 177

13 OpenSSL Compatibility 178
13.1 Compatibility with OpenSSL . . . 178
13.2 Differences Between wolfSSL and OpenSSL . . . 178
13.3 Supported OpenSSL Structures . . . 179
13.4 Supported OpenSSL Functions . . . 179
13.5 x509 Certificates . . . 180

14 Licensing 181

14.1 Open Source . . . 181
14.2 Commercial Licensing . . . 181
14.3 FIPS 140-2/3 Validation . . . 181
14.4 Support Packages . . . 181

15 Support and Consulting 182
15.1 How to Get Support . . . 182
15.1.1 Bugs Reports and Support Issues . . . 182
15.2 Consulting . . . 182
15.2.1 Feature Additions and Porting . . . 182
15.2.2 Competitive Upgrade Program . . . 182
15.2.3 Design Consulting . . . 183

16 wolfSSL (formerly CyaSSL) Updates 184
16.1 Product Release Information . . . 184

A wolfSSL API Reference 185
A.1 CertManager API . . . 185
A.1.1 Functions . . . 185
A.1.2 Functions Documentation . . . 187
A.2 Memory Handling . . . 205
A.2.1 Functions . . . 205
A.2.2 Functions Documentation . . . 210
A.3 OpenSSL API . . . 222
A.3.1 Functions . . . 222
A.3.2 Functions Documentation . . . 226
A.4 wolfSSL Certificates and Keys . . . 249
A.4.1 Functions . . . 249
A.4.2 Functions Documentation . . . 259
A.5 wolfSSL Connection, Session, and I/O . . . 325
A.5.1 Functions . . . 325
A.5.2 Functions Documentation . . . 334
A.6 wolfSSL Context and Session Set Up . . . 389
A.6.1 Functions . . . 389
A.6.2 Functions Documentation . . . 406
A.7 wolfSSL Error Handling and Reporting . . . 486
A.7.1 Functions . . . 486
A.7.2 Functions Documentation . . . 487
A.8 wolfSSL Initialization/Shutdown . . . 494
A.8.1 Functions . . . 494
A.8.2 Functions Documentation . . . 496

B wolfCrypt API Reference 508
B.1 ASN.1 . . . 508
B.1.1 Functions . . . 508
B.1.2 Functions Documentation . . . 514
B.2 Base Encoding . . . 564
B.2.1 Functions . . . 564
B.2.2 Functions Documentation . . . 565
B.3 Compression . . . 570
B.3.1 Functions . . . 570
B.3.2 Functions Documentation . . . 571
B.4 Error Reporting . . . 572
B.4.1 Functions . . . 573
B.4.2 Functions Documentation . . . 573

B.5 IoT-Safe Module . . . 574
B.5.1 Functions . . . 574
B.5.2 Detailed Description . . . 575
B.5.3 Functions Documentation . . . 576
B.6 Key and Cert Conversion . . . 587
B.7 Logging . . . 588
B.7.1 Functions . . . 588
B.7.2 Functions Documentation . . . 588
B.8 Math API . . . 589
B.8.1 Functions . . . 589
B.8.2 Functions Documentation . . . 589
B.9 Random Number Generation . . . 590
B.9.1 Functions . . . 590
B.9.2 Functions Documentation . . . 591
B.10 Signature API . . . 598
B.10.1 Functions . . . 598
B.10.2 Functions Documentation . . . 598
B.11 wolfCrypt Init and Cleanup . . . 602
B.11.1 Functions . . . 602
B.11.2 Functions Documentation . . . 602
B.12 Algorithms - 3DES . . . 605
B.12.1 Functions . . . 605
B.12.2 Functions Documentation . . . 607
B.13 Algorithms - AES . . . 618
B.13.1 Functions . . . 618
B.13.2 Functions Documentation . . . 625
B.14 Algorithms - ARC4 . . . 668
B.14.1 Functions . . . 668
B.14.2 Functions Documentation . . . 669
B.15 Algorithms - BLAKE2 . . . 670
B.15.1 Functions . . . 670
B.15.2 Functions Documentation . . . 671
B.16 Algorithms - Camellia . . . 673
B.16.1 Functions . . . 673
B.16.2 Functions Documentation . . . 674
B.17 Algorithms - ChaCha . . . 678
B.17.1 Functions . . . 678
B.17.2 Functions Documentation . . . 678
B.18 Algorithms - ChaCha20_Poly1305 . . . 680
B.18.1 Functions . . . 680
B.18.2 Functions Documentation . . . 681
B.19 Callbacks - CryptoCb . . . 684
B.20 Algorithms - Curve25519 . . . 684
B.20.1 Functions . . . 684
B.20.2 Functions Documentation . . . 686
B.21 Algorithms - Curve448 . . . 702
B.21.1 Functions . . . 702
B.21.2 Functions Documentation . . . 704
B.22 Algorithms - DSA . . . 720
B.22.1 Functions . . . 720
B.22.2 Functions Documentation . . . 721
B.23 Algorithms - Diffie-Hellman . . . 728
B.23.1 Functions . . . 728
B.23.2 Functions Documentation . . . 730

B.24 Algorithms - ECC . . . 739
B.24.1 Functions . . . 739
B.24.2 Functions Documentation . . . 744
B.25 Algorithms - ED25519 . . . 783
B.25.1 Functions . . . 783
B.25.2 Functions Documentation . . . 786
B.26 Algorithms - ED448 . . . 809
B.26.1 Functions . . . 809
B.26.2 Functions Documentation . . . 812
B.27 Platform Security Architecture (PSA) API . . . 832
B.27.1 Functions . . . 832
B.27.2 Functions Documentation . . . 832
B.28 Algorithm - SipHash . . . 835
B.28.1 Functions . . . 835
B.28.2 Functions Documentation . . . 835

C API Header Files 839
C.1 dox_comments/header_files/aes.h . . . 839
C.1.1 Functions . . . 839
C.1.2 Functions Documentation . . . 846
C.1.3 Source code . . . 883
C.2 dox_comments/header_files/arc4.h . . . 886
C.2.1 Functions . . . 886
C.2.2 Functions Documentation . . . 887
C.2.3 Source code . . . 888
C.3 dox_comments/header_files/asn.h . . . 888
C.4 dox_comments/header_files/asn_public.h . . . 888
C.4.1 Functions . . . 888
C.4.2 Functions Documentation . . . 896
C.4.3 Source code . . . 941
C.5 dox_comments/header_files/blake2.h . . . 944
C.5.1 Functions . . . 944
C.5.2 Functions Documentation . . . 945
C.5.3 Source code . . . 947
C.6 dox_comments/header_files/bn.h . . . 947
C.6.1 Functions . . . 947
C.6.2 Functions Documentation . . . 947
C.6.3 Source code . . . 948
C.7 dox_comments/header_files/camellia.h . . . 948
C.7.1 Functions . . . 948
C.7.2 Functions Documentation . . . 949
C.7.3 Source code . . . 952
C.8 dox_comments/header_files/chacha20_poly1305.h . . . 952
C.8.1 Functions . . . 952
C.8.2 Functions Documentation . . . 953
C.8.3 Source code . . . 955
C.9 dox_comments/header_files/chacha.h . . . 956
C.9.1 Functions . . . 956
C.9.2 Functions Documentation . . . 956
C.9.3 Source code . . . 958
C.10 dox_comments/header_files/cmac.h . . . 958
C.10.1 Functions . . . 958
C.10.2 Functions Documentation . . . 959
C.10.3 Source code . . . 964

C.11 dox_comments/header_files/coding.h . . . 964
C.11.1 Functions . . . 964
C.11.2 Functions Documentation . . . 965
C.11.3 Source code . . . 970
C.12 dox_comments/header_files/compress.h . . . 970
C.12.1 Functions . . . 970
C.12.2 Functions Documentation . . . 970
C.12.3 Source code . . . 972
C.13 dox_comments/header_files/cryptocb.h . . . 972
C.13.1 Functions . . . 972
C.13.2 Functions Documentation . . . 972
C.13.3 Source code . . . 974
C.14 dox_comments/header_files/curve25519.h . . . 975
C.14.1 Functions . . . 975
C.14.2 Functions Documentation . . . 976
C.14.3 Source code . . . 990
C.15 dox_comments/header_files/curve448.h . . . 992
C.15.1 Functions . . . 992
C.15.2 Functions Documentation . . . 993
C.15.3 Source code . . . 1007
C.16 dox_comments/header_files/des3.h . . . 1009
C.16.1 Functions . . . 1009
C.16.2 Functions Documentation . . . 1010
C.16.3 Source code . . . 1016
C.17 dox_comments/header_files/dh.h . . . 1017
C.17.1 Functions . . . 1017
C.17.2 Functions Documentation . . . 1019
C.17.3 Source code . . . 1027
C.18 dox_comments/header_files/doxygen_groups.h . . . 1028
C.19 dox_comments/header_files/doxygen_pages.h . . . 1028
C.20 dox_comments/header_files/dsa.h . . . 1028
C.20.1 Functions . . . 1028
C.20.2 Functions Documentation . . . 1029
C.20.3 Source code . . . 1035
C.21 dox_comments/header_files/ecc.h . . . 1036
C.21.1 Functions . . . 1036
C.21.2 Functions Documentation . . . 1040
C.21.3 Source code . . . 1074
C.22 dox_comments/header_files/eccsi.h . . . 1076
C.22.1 Functions . . . 1076
C.22.2 Functions Documentation . . . 1078
C.22.3 Source code . . . 1082
C.23 dox_comments/header_files/ed25519.h . . . 1083
C.23.1 Functions . . . 1083
C.23.2 Functions Documentation . . . 1087
C.23.3 Source code . . . 1107
C.24 dox_comments/header_files/ed448.h . . . 1108
C.24.1 Functions . . . 1108
C.24.2 Functions Documentation . . . 1111
C.24.3 Source code . . . 1128
C.25 dox_comments/header_files/error-crypt.h . . . 1130
C.25.1 Functions . . . 1130
C.25.2 Functions Documentation . . . 1130
C.25.3 Source code . . . 1131

C.26 dox_comments/header_files/evp.h . . . 1131
C.26.1 Functions . . . 1131
C.26.2 Functions Documentation . . . 1133
C.26.3 Source code . . . 1141
C.27 dox_comments/header_files/hash.h . . . 1142
C.27.1 Functions . . . 1142
C.27.2 Functions Documentation . . . 1143
C.27.3 Source code . . . 1151
C.28 dox_comments/header_files/hmac.h . . . 1151
C.28.1 Functions . . . 1151
C.28.2 Functions Documentation . . . 1154
C.28.3 Source code . . . 1165
C.29 dox_comments/header_files/iotsafe.h . . . 1167
C.29.1 Functions . . . 1167
C.29.2 Functions Documentation . . . 1168
C.29.3 Source code . . . 1179
C.30 dox_comments/header_files/logging.h . . . 1180
C.30.1 Functions . . . 1180
C.30.2 Functions Documentation . . . 1180
C.30.3 Source code . . . 1182
C.31 dox_comments/header_files/md2.h . . . 1182
C.31.1 Functions . . . 1182
C.31.2 Functions Documentation . . . 1182
C.31.3 Source code . . . 1184
C.32 dox_comments/header_files/md4.h . . . 1185
C.32.1 Functions . . . 1185
C.32.2 Functions Documentation . . . 1185
C.32.3 Source code . . . 1187
C.33 dox_comments/header_files/md5.h . . . 1187
C.33.1 Functions . . . 1187
C.33.2 Functions Documentation . . . 1187
C.33.3 Source code . . . 1191
C.34 dox_comments/header_files/memory.h . . . 1191
C.34.1 Functions . . . 1191
C.34.2 Functions Documentation . . . 1193
C.34.3 Source code . . . 1201
C.35 dox_comments/header_files/pem.h . . . 1201
C.35.1 Functions . . . 1201
C.35.2 Functions Documentation . . . 1201
C.35.3 Source code . . . 1202
C.36 dox_comments/header_files/pkcs11.h . . . 1202
C.36.1 Functions . . . 1202
C.36.2 Functions Documentation . . . 1203
C.36.3 Source code . . . 1203
C.37 dox_comments/header_files/pkcs7.h . . . 1204
C.37.1 Functions . . . 1204
C.37.2 Functions Documentation . . . 1205
C.37.3 Source code . . . 1215
C.38 dox_comments/header_files/poly1305.h . . . 1216
C.38.1 Functions . . . 1216
C.38.2 Functions Documentation . . . 1216
C.38.3 Source code . . . 1219
C.39 dox_comments/header_files/psa.h . . . 1219
C.39.1 Functions . . . 1219

C.39.2 Functions Documentation . . . 1220
C.39.3 Source code . . . 1222
C.40 dox_comments/header_files/pwdbased.h . . . 1222
C.40.1 Functions . . . 1222
C.40.2 Functions Documentation . . . 1223
C.40.3 Source code . . . 1226
C.41 dox_comments/header_files/quic.h . . . 1226
C.41.1 Functions . . . 1226
C.41.2 Attributes . . . 1229
C.41.3 Functions Documentation . . . 1229
C.41.4 Attributes Documentation . . . 1241
C.41.5 Source code . . . 1242
C.42 dox_comments/header_files/random.h . . . 1244
C.42.1 Functions . . . 1244
C.42.2 Attributes . . . 1244
C.42.3 Functions Documentation . . . 1244
C.42.4 Attributes Documentation . . . 1250
C.42.5 Source code . . . 1251
C.43 dox_comments/header_files/ripemd.h . . . 1251
C.43.1 Functions . . . 1251
C.43.2 Functions Documentation . . . 1251
C.43.3 Source code . . . 1253
C.44 dox_comments/header_files/rsa.h . . . 1254
C.44.1 Functions . . . 1254
C.44.2 Functions Documentation . . . 1258
C.44.3 Source code . . . 1289
C.45 dox_comments/header_files/sakke.h . . . 1291
C.45.1 Functions . . . 1291
C.45.2 Functions Documentation . . . 1292
C.45.3 Source code . . . 1297
C.46 dox_comments/header_files/sha256.h . . . 1298
C.46.1 Functions . . . 1298
C.46.2 Functions Documentation . . . 1299
C.46.3 Source code . . . 1303
C.47 dox_comments/header_files/sha512.h . . . 1304
C.47.1 Functions . . . 1304
C.47.2 Functions Documentation . . . 1304
C.47.3 Source code . . . 1308
C.48 dox_comments/header_files/sha.h . . . 1308
C.48.1 Functions . . . 1308
C.48.2 Functions Documentation . . . 1308
C.48.3 Source code . . . 1311
C.49 dox_comments/header_files/signature.h . . . 1311
C.49.1 Functions . . . 1311
C.49.2 Functions Documentation . . . 1312
C.49.3 Source code . . . 1315
C.50 dox_comments/header_files/siphash.h . . . 1315
C.50.1 Functions . . . 1315
C.50.2 Functions Documentation . . . 1316
C.50.3 Source code . . . 1319
C.51 dox_comments/header_files/srp.h . . . 1319
C.51.1 Functions . . . 1319
C.51.2 Functions Documentation . . . 1320
C.51.3 Source code . . . 1330

C.52 dox_comments/header_files/ssl.h . . . 1330
C.52.1 Functions . . . 1330
C.52.2 Functions Documentation . . . 1387
C.52.3 Source code . . . 1667
C.53 dox_comments/header_files/tfm.h . . . 1688
C.53.1 Functions . . . 1688
C.53.2 Functions Documentation . . . 1689
C.53.3 Source code . . . 1689
C.54 dox_comments/header_files/types.h . . . 1689
C.54.1 Functions . . . 1689
C.54.2 Functions Documentation . . . 1692
C.54.3 Source code . . . 1695
C.55 dox_comments/header_files/wc_encrypt.h . . . 1695
C.55.1 Functions . . . 1695
C.55.2 Functions Documentation . . . 1696
C.55.3 Source code . . . 1701
C.56 dox_comments/header_files/wc_port.h . . . 1701
C.56.1 Functions . . . 1701
C.56.2 Functions Documentation . . . 1701
C.56.3 Source code . . . 1702
C.57 dox_comments/header_files/wolfio.h . . . 1702
C.57.1 Functions . . . 1702
C.57.2 Functions Documentation . . . 1705
C.57.3 Source code . . . 1718

D SSL/TLS Overview 1719
D.1 General Architecture . . . 1719
D.2 SSL Handshake . . . 1719
D.3 Differences between SSL and TLS Protocol Versions . . . 1719
D.3.1 SSL 3.0 . . . 1721
D.3.2 TLS 1.0 . . . 1721
D.3.3 TLS 1.1 . . . 1721
D.3.4 TLS 1.2 . . . 1721
D.3.5 TLS 1.3 . . . 1722

E RFCs, Specifications, and Reference 1723
E.1 Protocols . . . 1723
E.2 Stream Ciphers . . . 1723
E.3 Block Ciphers . . . 1723
E.4 Hashing Functions . . . 1723
E.5 Public Key Cryptography . . . 1723
E.6 Other . . . 1723

F Error Codes 1724
F.1 wolfSSL Error Codes . . . 1724
F.2 wolfCrypt Error Codes . . . 1727
F.3 Common Error Codes and their Solution . . . 1730
F.3.1 `ASN_NO_SIGNER_E` (-188) . . . 1730
F.3.2 `WANT_READ` (-323) . . . 1730

G Experimenting with Post-Quantum Cryptography 1731
G.1 A Gentle Introduction to Post-Quantum Cryptography . . . 1731
G.1.1 Why Post-Quantum Cryptography? . . . 1731
G.1.2 How do we Protect Ourselves? . . . 1731
G.2 Getting Started with Post-Quantum algorithms in wolfSSL . . . 1732

G.2.1 Build Instructions . . . 1732
G.2.2 Making a Quantum Safe TLS Connection . . . 1732
G.3 Post Quantum Algorithm Variant Names . . . 1733
G.4 Cryptographic Artifact Sizes . . . 1733
G.5 Statistics . . . 1734
G.5.1 Runtime Binary Sizes . . . 1734
G.5.2 TLS 1.3 Data Transmission Sizes . . . 1734
G.5.3 Heap and Stack Usage . . . 1735
G.5.4 Benchmarks . . . 1740
G.6 Documentation . . . 1748
G.7 Post-Quantum Stateful Hash-Based Signatures . . . 1748
G.7.1 Motivation . . . 1748
G.7.2 LMS/HSS signatures . . . 1749
G.7.3 XMSS/XMSS^MT signatures . . . 1752
G.7.4 Developer Notes . . . 1756

H wolfSSL Porting Guide 1757
H.1 Purpose . . . 1757
H.2 Audience . . . 1757
H.3 Introduction . . . 1757
H.4 Porting wolfSSL . . . 1757
H.4.1 Data Types . . . 1757
H.4.2 Endianness . . . 1758
H.4.3 writev . . . 1758
H.4.4 Input / Output . . . 1759
H.4.5 Filesystem . . . 1759
H.4.6 Threading . . . 1760
H.4.7 Random Seed . . . 1760
H.4.8 Memory . . . 1761
H.4.9 Time . . . 1761
H.4.10 C Standard Library . . . 1761
H.4.11 Logging . . . 1762
H.4.12 Public Key Operations . . . 1762
H.4.13 Atomic Record Layer Processing . . . 1762
H.4.14 Features . . . 1762
H.5 Next Steps . . . 1763
H.5.1 wolfCrypt Test Application . . . 1763
H.6 Support . . . 1763

I wolfSM (ShangMi) 1764
I.1 Getting and Installing wolfSM . . . 1764
I.1.1 Get wolfSM from GitHub . . . 1764
I.1.2 Get wolfSSL from GitHub . . . 1764
I.1.3 Install SM code into wolfSSL . . . 1764
I.2 Building wolfSM . . . 1764
I.2.1 Optimized SM2 . . . 1765
I.3 Testing wolfSM . . . 1765
I.3.1 Testing TLS . . . 1765

1 Introduction

This manual is written as a technical guide to the wolfSSL embedded SSL/TLS library. It will explain how to build and get started with wolfSSL, provide an overview of build options, features, portability enhancements, support, and much more.

You can find the PDF version of this document here.

1.1 Why Choose wolfSSL?

There are many reasons to choose wolfSSL as your embedded SSL solution. Some of the top reasons include size (typical footprint sizes range from 20-100 kB), support for the newest standards (SSL 3.0, TLS 1.0, TLS 1.1, TLS 1.2, TLS 1.3, DTLS 1.0, DTLS 1.2, and DTLS 1.3), current and progressive cipher support (including stream ciphers), multi- platform, royalty free, and an OpenSSL compatibility API to ease porting into existing applications which have previously used the OpenSSL package. For a complete feature list, see Features Overview.

COPYRIGHT ©2024 wolfSSL Inc.

2 Building wolfSSL

wolfSSL was written with portability in mind and should generally be easy to build on most systems. If you have difficulty building wolfSSL, please don't hesitate to seek support through our support forums (https://www.wolfssl.com/forums) or contact us directly at support@wolfssl.com.

This chapter explains how to build wolfSSL on Unix and Windows, and provides guidance for building wolfSSL in a non-standard environment. You will find the "getting started" guide in Chapter 3 and an SSL tutorial in Chapter 11.

When using the autoconf / automake system to build wolfSSL, wolfSSL uses a single Makefile to build all parts and examples of the library, which is both simpler and faster than using Makefiles recursively.

2.1 Getting wolfSSL Source Code

The most recent version of wolfSSL can be downloaded from the wolfSSL website as a ZIP file:

https://www.wolfssl.com/download

After downloading the ZIP file, unzip the file using the `unzip` command. To use native line endings, enable the `-a` modifier when using unzip. From the unzip man page, the `-a` modifier functionality is described:

> [...] The -a option causes files identified by zip as text files (those with the 't' label in zipinfo listings, rather than 'b') to be automatically extracted as such, converting line endings, end-of-file characters and the character set itself as necessary. [...]

NOTE: Beginning with the release of wolfSSL 2.0.0rc3, the directory structure of wolfSSL was changed as well as the standard install location. These changes were made to make it easier for open source projects to integrate wolfSSL. For more information on header and structure changes, please see Library Headers and Structure Usage.

2.2 Building on Unix-like Systems

When building wolfSSL on Linux, *BSD, OS X, Solaris, or other *nix-like systems, use the autoconf system. To build wolfSSL you only need to run two commands from the wolfSSL root directory, `./configure` and `make`.

The `./configure` script sets up the build environment and you can append any number of build options to `./configure`. For a list of available build options, please see Build Options or run the following command to see a list of possible options to pass to the `./configure` script:

```
./configure --help
```

Once `./configure` has successfully executed, to build wolfSSL, run:

```
make
```

To install wolfSSL run:

```
make install
```

You may need superuser privileges to install, in which case precede the command with sudo:

```
sudo make install
```

To test the build, run the testsuite program from the root wolfSSL directory:

```
./testsuite/testsuite.test
```

Alternatively you can use autoconf to run the testsuite as well as the standard wolfSSL API and crypto tests:

```
make test
```

Further details about expected output of the testsuite program can be found in the Testsuite section. If you want to build only the wolfSSL library and not the additional items (examples, testsuite, benchmark app, etc.), you can run the following command fromthe wolfSSL root directory:

```
make src/libwolfssl.la
```

2.3 Building on Windows

In addition to the instructions below, you can find instructions and tips for building wolfSSL with Visual Studio here.

2.3.1 VS 2008

Solutions are included for Visual Studio 2008 in the root directory of the install. For use with Visual Studio 2010 and later, the existing project files should be able to be converted during the import process.

Note: If importing to a newer version of VS you will be asked: "Do you want to overwrite the project and its imported property sheets?" You can avoid the following by selecting "No". Otherwise if you select "Yes", you will see warnings about `EDITANDCONTINUE` being ignored due to `SAFESEH` specification. You will need to right click on the testsuite, sslSniffer, server, echoserver, echoclient, and client individually and modify their Properties->Configuration Properties->Linker->Advanced (scroll all the way to the bottom in Advanced window). Locate "Image Has Safe Exception Handlers" and click the drop down arrow on the far right. Change this to No (`/SAFESEH:NO`) for each of the aforementioned. The other option is to disable `EDITANDCONTINUE` which, we have found to be useful for debugging purposes and is therefore not recommended for production software.

2.3.2 VS 2010

You will need to download Service Pack 1 to build wolfSSL solution once it has been updated. If VS reports a linker error, clean and rebuild the project; the linker error should be taken care of.

2.3.3 VS 2013 (64 bit solution)

You will need to download Service Pack 4 to build wolfSSL solution once it has been updated. If VS reports a linker error, clean and rebuild the project; the linker error should be taken care of.

To test each build, choose "Build All" from the Visual Studio menu and then run the testsuite program. To edit build options in the Visual Studio project, select your desired project (wolfssl, echoclient, echoserver, etc.) and browse to the "Properties" panel.

Note: After the wolfSSL v3.8.0 release the build preprocessor macros were moved to a centralized file located at `IDE/WIN/user_settings.h`. This file can also be found in the project. To add features such as ECC or ChaCha20/Poly1305, add `#defines` here such as `HAVE_ECC` or `HAVE_CHACHA` / `HAVE_POLY1305`.

2.3.4 Cygwin

If building wolfSSL for Windows on a Windows development machine, we recommend using the included Visual Studio project files to build wolfSSL. However, if Cygwin is required here is a short guide on how our team achieved a successful build:

1. Go to https://www.cygwin.com/install.html and download `setup-x86_64.exe`
2. Run `setup-x86_64.exe` and install however you choose. Click through the installation menus until you reach the "Select Packages" stage.

3. Click on the "+" icon to expand "All"
4. Now go to the "Archive" section and select "unzip" drop down, change "Skip" to 6.0-15 (or some other version).
5. Under "Devel" click "autoconf" drop down and change "Skip" to "10-1" (or some other version)
6. Under "Devel" click "automake" drop down and change "Skip" to "10-1" (or some other version)
7. Under "Devel" click the "gcc-core" drop down and change "Skip" to 7.4.0-1 (NOTE: wolfSSL has not tested GCC 9 or 10 and as they are fairly new does not recommend using them until they have had a bit more time to be fine-tuned for development).
8. Under "Devel" click the "git" drop down and change "Skip" to 2.29.0-1 (or some other version)
9. Under "Devel" click "libtool" drop down and change "Skip" to "2.4.6-5" (or some other version)
10. Under "Devel" click the "make" drop down and change "Skip" to 4.2.1-1 (or some other version)
11. Click "Next" and proceed through the rest of the installation.

The additional packages list should include:

- unzip
- autoconf
- automake
- gcc-core
- git
- libtool
- make

2.3.4.1 Post Install

Open a Cygwin terminal and clone wolfSSL:

```
git clone https://github.com/wolfssl/wolfssl.git
cd wolfssl
./autogen.sh
./configure
make
make check
```

2.4 Building in a non-standard environment

While not officially supported, we try to help users wishing to build wolfSSL in a non-standard environment, particularly with embedded and cross-compilation systems. Below are some notes on getting started with this.

1. The source and header files need to remain in the same directory structure as they are in the wolfSSL download package.
2. Some build systems will want to explicitly know where the wolfSSL header files are located, so you may need to specify that. They are located in the `<wolfssl_root>/wolfssl` directory. Typically, you can add the `<wolfssl_root>` directory to your include path to resolve header problems.
3. wolfSSL defaults to a little endian system unless the configure process detects big endian. Since users building in a non-standard environment aren't using the configure process, `BIG_ENDIAN_ORDER` will need to be defined if using a big endian system.
4. wolfSSL benefits speed-wise from having a 64-bit type available. The configure process determines if long or long long is 64 bits and if so sets up a define. So if `sizeof(long)` is 8 bytes on your system, define `SIZEOF_LONG 8`. If it isn't but `sizeof(long long)` is 8 bytes, then define `SIZEOF_LONG_LONG 8`.
5. Try to build the library, and let us know if you run into any problems. If you need help, contact us at info@wolfssl.com.
6. Some defines that can modify the build are listed in the following sub-sections, below. For more verbose descriptions of many options, please see Build Options.

2.4.1 Building into Yocto Linux

wolfSSL also includes recipes for building wolfSSL on Yocto Linux and OpenEmbedded. These recipes are maintained within the meta-wolfSSL layer as a GitHub repository, here: https://github.com/wolfSSL/meta-wolfssl. Building wolfSSL on Yocto Linux will require Git and bitbake. The following steps list how to get some wolfSSL products (that recipes exist for) built on Yocto Linux.

1. **Cloning wolfSSL meta**

   This can be done through a git-clone command of the following URL: https://github.com/wolfSSL/meta-wolfssl

2. **Insert the "meta-wolfSSL" layer into the build's bblayers.conf**

   Within the BBLAYERS section, add the path to the location where meta-wolfssl was cloned into. Example:

   ```
   BBLAYERS ?= "... \
   /path/to/meta-wolfssl/ \
   ..."
   ```

3. **Build a wolfSSL product recipe**

   bitbake can be used to build one of the three following wolfSSL product recipes: *wolfssl*, *wolfssh*, and *wolfmqtt*. Simply pass one of those recipes into the bitbake command (example: `bitbake wolfssl`). This allows the user to personally confirm compilation succeeds without issues.

4. **Edit local.conf**

   The final step is to edit the build's local.conf file, which allows desired libraries to be included with the image being built. Edit the `IMAGE_INSTALL_append` line to include the name of the desired recipe(s). An example of this is shown below:

   ```
   IMAGE_INSTALL_append = "wolfssl wolfssh wolfmqtt"
   ```

Once the image has been built, wolfSSL's default location (or related products from recipes) will be the `/usr/lib/` directory.

Additionally, wolfSSL can be customized when building into Yocto by using the enable and disable options listed in Build Options. This requires creating a `.bbappend` file and placing it within the wolfSSL application/recipe layer. The contents of this file should include a line specifying content to concatenate onto the `EXTRA_OECONF` variable. An example of this is shown below to enable TLS 1.3 support through the TLS 1.3 enable option:

```
EXTRA_OECONF += "--enable-tls13"
```

Further documentation on building into Yocto can be found in the meta-wolfssl README, located here: https://github.com/wolfSSL/meta-wolfssl/blob/master/README.md

2.4.2 Building with Atollic TrueSTUDIO

Versions of wolfSSL following 3.15.5 include a TrueSTUDIO project file that is used to build wolfSSL on ARM M4-Cortex devices. The TrueSTUDIO project file simplifies the process of building on STM32 devices, is free to download, and is createed by Atollic - a part of ST Microelectronics. To build the wolfSSL static library project file in TrueSTUDIO, it will require the user perform the following steps after opening TrueSTUDIO:

1. Import the project into the workspace (File > Import)
2. Build the project (Project > Build project)

The build then includes the settings located inside of `user_settings.h` at build-time. The default content of the `user_settings.h` file is minimal, and does not contain many features. Users are able to modify this file and add or remove features with options listed in the remainder of this chapter.

2.4.3 Building with IAR

The `<wolfssl_root>/IDE/IAR-EWARM` directory contains the following files:

1. Workspace: `wolfssl.eww` The workspace includes wolfSSL-Lib library and wolfCrypt-test, wolfCrypt-benchmark executable projects.
2. wolfSSL-Lib Project: `lib/wolfSSL-lib.ewp` generates full set library of wolfCrypt and wolfSSL functions.
3. Test suites Project: `test/wolfCrypt-test.ewp` generates test.out test suites executable
4. Benchmark Project: `benchmark/wolfCrypt-benchmark.ewp` generates benchmark.out benchmark executable

These projects have been set up to generic ARM Cortex-M MPUs. In order to generate project for specific target MPU, take following steps.

1. Default Setting: Default Target of the projects are set to Cortex-M3 Simulator. user_settings.h includes default options for the projects. You can build and download the to the simulator. Open Terminal I/O window, by “view”->“Terminal I/O”, and start execution.
2. Project option settings: For each project, choose appropriate “Target” options.
3. For executable projects: Add “SystemInit” and “startup” for your MPU, choose your debug “Driver”.
4. For benchmark project: Choose option for current_time function or write your own “current_time” benchmark timer with WOLFSSL_USER_CURRTIME option.
5. Build and download: Go to “Project->Make” and “Download and Debug” in Menu bar for EWARM build and download.

2.4.4 Building on OS X and iOS

2.4.4.1 XCODE The `<wolfssl_root>/IDE/XCODE` directory contains the following files:

1. `wolfssl.xcworkspace` – workspace with library and testsuite client
2. `wolfssl_testsuite.xcodeproj` – project to run the testsuite.
3. `wolfssl.xcodeproj` – project to build OS/x and iOS libraries for wolfSSL and/or wolfCrypt
4. `wolfssl-FIPS.xcodeproj` – project to build wolfSSL and wolfCrypt-FIPS if available
5. `user_settings.h` – custom library settings, which are shared across projects

The library will output as `libwolfssl_osx.a` or `libwolfssl_ios.a` depending on the target. It will also copy the wolfSSL/wolfCrypt (and the CyaSSL/CtaoCrypt compatibility) headers into an `include` directory located in `Build/Products/Debug` or `Build/Products/Release`.

For the library and testsuite to link properly the build location needs to be configured relative to the workspace.

1. File -> Workspace Settings (or Xcode -> Preferences -> Locations -> Locations)
2. Derived Data -> Advanced
3. Custom -> Relative to Workspace
4. Products -> Build/Products

These Xcode projects define the `WOLFSSL_USER_SETTINGS` preprocessor to enable the `user_settings.h` file for setting macros across multiple projects.

If needed the Xcode preprocessors can be modified with these steps:

1. Click on the Project in “Project Navigator”.
2. Click on the “Build Settings” tab.
3. Scroll down to the “Apple LLVM 6.0 - Preprocessing” section.
4. Open the disclosure for “Preprocessor Macros” and use the “+” and “-” buttons to modify. Remember to do this for both Debug and Release.

This project should build wolfSSL and wolfCrypt using the default settings.

2.4.5 Building with GCC ARM

In the `<wolfssl_root>/IDE/GCC-ARM` directory, you will find an example wolfSSL project for Cortex M series, but it can be adopted for other architectures.

1. Make sure you have `gcc-arm-none-eabi` installed.
2. Modify the `Makefile.common`:
   - Use correct toolchain path `TOOLCHAIN`.
   - Use correct architecture ‘ARCHFLAGS’. See GCC ARM Options `-mcpu=name`.
   - Confirm memory map in linker.ld matches your flash/ram or comment out `SRC_LD = -T./linker.ld` in Makefile.common.
3. Use `make` to build the static library (libwolfssl.a), wolfCrypt test/benchmark and wolfSSL TLS client targets as `.elf` and `.hex` in `/Build`.

2.4.5.1 Building with generic makefile cross-compile Example `Makefile.common` changes for Raspberry Pi with Cortex-A53:

1. In Makefile.common change `ARCHFLAGS` to `-mcpu=cortex-a53 -mthumb`.
2. Comment out `SRC_LD`, since custom memory map is not applicable.
3. Clear `TOOLCHAIN`, so it will use default `gcc`. Set `TOOLCHAIN =`
4. Comment out `LDFLAGS += --specs=nano.specs` and `LDFLAGS += --specs=nosys.specs` to nosys and nano.

2.4.5.2 Building with configure with cross-compile The configure script in the main project directory can perform a cross-compile build with the gcc-arm-none-eabi tools. Assuming the tools are installed in your executable path:

```
./configure \
  --host=arm-non-eabi \
  CC=arm-none-eabi-gcc \
  AR=arm-none-eabi-ar \
  STRIP=arm-none-eabi-strip \
  RANLIB=arm-none-eabi-ranlib \
  --prefix=/path/to/build/wolfssl-arm \
  CFLAGS="-march=armv8-a --specs=nosys.specs \
      -DHAVE_PK_CALLBACKS -DWOLFSSL_USER_IO -DNO_WRITEV" \
  --disable-filesystem --enable-fastmath \
  --disable-shared
make
make install
```

If you are building for a 32-bit architecture, add `-DTIME_T_NOT_64BIT` to the list of CFLAGS.

2.4.6 Building on Keil MDK-ARM

You can find detailed instructions and tips for building wolfSSL on Keil MDK-ARM here.

Note: If MDK-ARM is not installed in the default installation location, you need to change all of the referencing path definitions in the project file to the install location.

2.5 Features Defined as C Pre-processor Macro

2.5.1 Removing Features

The following defines can be used to remove features from wolfSSL. This can be helpful if you are trying to reduce the overall library footprint size. In addition to defining a NO_<feature-name> define, you can also remove the respective source file as well from the build (but not the header file).

2.5.1.1 NO_WOLFSSL_CLIENT Removes calls specific to the client and is for server-only builds. You should only use this if you want to remove a few calls for the sake of size.

2.5.1.2 NO_WOLFSSL_SERVER Likewise removes calls specific to the server side.

2.5.1.3 NO_DES3 Removes the use of DES3 encryptions. DES3 is built-in by default because some older servers still use it and it's required by SSL 3.0. NO_DH and NO_AES are the same as the two above, they are widely used.

2.5.1.4 NO_DSA Removes DSA since it's being phased out of popular use.

2.5.1.5 NO_ERROR_STRINGS Disables error strings. Error strings are located in src/internal.c for wolfSSL or wolfcrypt/src/asn.c for wolfCrypt.

2.5.1.6 NO_HMAC Removes HMAC from the build.

NOTE: SSL/TLS depends on HMAC but if you are only using wolfCrypt IE build option WOLFCRYPT_ONLY then HMAC can be disabled in this case.

2.5.1.7 NO_MD4 Removes MD4 from the build, MD4 is broken and shouldn't be used.

2.5.1.8 NO_MD5 Removes MD5 from the build.

2.5.1.9 NO_SHA Removes SHA-1 from the build.

2.5.1.10 NO_SHA256 Removes SHA-256 from the build.

2.5.1.11 NO_PSK Turns off the use of the pre-shared key extension. It is built-in by default.

2.5.1.12 NO_PWDBASED Disables password-based key derivation functions such as PBKDF1, PBKDF2, and PBKDF from PKCS #12.

2.5.1.13 NO_RC4 Removes the use of the ARC4 stream cipher from the build. ARC4 is built-in by default because it is still popular and widely used.

2.5.1.14 NO_SESSION_CACHE Can be defined when a session cache is not needed. This should reduce memory use by nearly 3 kB.

2.5.1.15 NO_TLS Turns off TLS. We do not recommend turning off TLS.

2.5.1.16 SMALL_SESSION_CACHE Can be defined to limit the size of the SSL session cache used by wolfSSL. This will reduce the default session cache from 33 sessions to 6 sessions and save approximately 2.5 kB.

2.5.1.17 NO_RSA Removes support for the RSA algorithm.

2.5.1.18 WC_NO_RSA_OAEP Removes code for OAEP padding.

2.5.1.19 NO_AES_CBC Turns off AES-CBC algorithm support.

2.5.1.20 NO_AES_DECRYPT Can be set to reduce code size. Set to disable the decrypt and only support encryption.

2.5.1.21 WOLFCRYPT_ONLY Enables wolfCrypt only while disabling TLS.

2.5.1.22 NO_CAMELLIA_CBC Disables Camellia CBC support but only applies to TLS cipher suites only.

2.5.1.23 NO_AES Disables AES algorithm support.

2.5.1.24 NO_AES_128 Used for AES key size selection at compile time.

2.5.1.25 NO_AES_192 Used for AES key size selection at compile time.

2.5.1.26 NO_AES_256 Used for AES key size selection at compile time.

2.5.1.27 NO_AESGCM_AEAD Used for disabliing TLS cipher suites thst use AES GCM. It is used internally when no AES GCM cipher suites are enabled, but can also be used to limit cipher suites.

2.5.1.28 NO_ASN_TIME Disables time checking for ASN. Note: This should be used with caution because all certificate begin/end date checking will be skipped.

2.5.1.29 NO_CHECK_PRIVATE_KEY This macro disables additional private key checking that is on by default. This enables checking to validate the private key is a pair for the public key. It is supported for RSA, ECDSA, ED25519, ED448, Falcon, Dilithium and Sphincs.

2.5.1.30 NO_DH Disables Diffie-Hellman (DH) support.

2.5.1.31 NO_ED25519_CLIENT_AUTH Disables TLS client authentication support for ED25519. It is used to reduce memory usage during TLS if ED25519 is not used, since it requires caching messages.

2.5.1.32 NO_ED448_CLIENT_AUTH Disables client authentification for ED448.

2.5.1.33 NO_FORCE_SCR_SAME_SUITE By default secure renegotiation requires using the same cipher suite. This disables that requirement.

2.5.1.34 NO_MULTIBYTE_PRINT Used for compiling out special characters that embedded devices may have problems with.

2.5.1.35 NO_OLD_SSL_NAMES This disables some of the old OpenSSL compatibility macros for using wolfSSL and OpenSSL together.

2.5.1.36 NO_OLD_WC_NAMES Removes unneeded namespace.

2.5.1.37 NO_OLD_POLY1305 This disables support for the old ChaCha20/Poly1305 TLS 1.2 cipher suite that is typically used for interop.

2.5.1.38 NO_HANDSHAKE_DONE_CB Disables support for the handshake callback set with `wolfSSL_SetHsDoneCb`. This option is useful for reducing code size.

2.5.1.39 NO_STDIO_FILESYSTEM This disables include of stdio.h. Used with portability.

2.5.1.40 NO_TLS_DH Excludes TLS DH. Should not negotiate cipher suites based on ephemeral finite-field Diffie-Hellman key agreement.

2.5.1.41 NO_WOLFSSL_CM_VERIFY Disables the Certificate Manager verify callback. The verify callback allows intercepting errors and overriding them. This option is useful for reducing code size.

2.5.1.42 NO_WOLFSSL_DIR Disable directory support.

2.5.1.43 NO_WOLFSSL_RENESAS_TSIP_TLS_SESSION For disabling only the TSIP TLS-linked Common key encryption method. Note: This is a Renesas RX TSIP specific define.

2.5.1.44 NO_WOLFSSL_SHA256 This applies to TLS 1.3 only. It allows SHA2-256 to be enabled and usable from wolfCrypt, but exclude it from TLS 1.3.

2.5.1.45 WOLFSSL_BLIND_PRIVATE_KEY Used as a mask to blind the private key. The blinding is used to proctect aginst Rowhammer attacks.

2.5.1.46 WOLFSSL_DTLS13_NO_HRR_ON_RESUME If defined, a DTLS server will not do a cookie exchange on successful client resumption: the resumption will be faster (one RTT less) and will consume less bandwidth (one ClientHello and one HelloVerifyRequest/HelloRetryRequest less). On the other hand, if a valid SessionID/ticket/psk is collected, forged clientHello messages will consume resources on the server. For DTLS 1.3, using this option also allows for the server to process Early Data/0-RTT Data. Without this, the Early Data would be dropped since the server doesn't enter stateful processing until receiving a verified ClientHello with the cookie. To allow DTLS 1.3 resumption without the cookie exchange:- Compile wolfSSL with `WOLFSSL_DTLS13_NO_HRR_ON_RESUME` defined - Call wolfSSL_dtls13_no_hrr_on_resume(ssl, 1) on the WOLFSSL object to disable the cookie exchange on resumption - Continue like with a normal connection.

2.5.1.47 WOLFSSL_NO_CLIENT_AUTH Disables the caching code required for using Ed25519 and Ed448.

2.5.1.48 WOLFSSL_NO_CURRDIR Portability macro for platforms that do not support ./ for test paths in wolfssl/test.h. Applies to testing tools only.

2.5.1.49 WOLFSSL_NO_DEF_TICKET_ENC_CB No default ticket encryption callback, server only. The application must set its own callback to use session tickets.

2.5.1.50 WOLFSSL_NO_SOCK Portability macro for disabling built-in socket support. If using TLS without sockets typically `WOLFSSL_USER_IO` would be defined and callbacks used for send/recv.

2.5.1.51 WOLFSSL_NO_TLS12 Define to exlude TLS 1.2.

2.5.1.52 WOLFSSL_PEM_TO_DER Key and cert generation feature support for disabling PEM to DER.

2.5.1.53 NO_DEV_URANDOM Disables the use of `/dev/urandom`

2.5.1.54 WOLFSSL_NO_SIGALG Disables the signature algorithms extension

2.5.1.55 NO_RESUME_SUITE_CHECK Disables the check of cipher suite when resuming a TLS connection

2.5.1.56 NO_ASN Removes support for ASN formatted certificate processing.

2.5.1.57 NO_OLD_TLS Removes support for SSLv3, TLSv1.0 and TLSv1.1

2.5.1.58 WOLFSSL_AEAD_ONLY Removes support for non-AEAD algorithms. AEAD stands for "authenticated encryption with associated data" which means these algorithms (such as AES-GCM) do not just encrypt and decrypt data, they also assure confidentiality and authenticity of that data.

2.5.1.59 WOLFSSL_SP_NO_2048 Removes RSA/DH 2048-bit Single-Precision (SP) optimization.

2.5.1.60 WOLFSSL_SP_NO_3072 Removes RSA/DH 3072-bit Single-Precision (SP) optimization.

2.5.1.61 WOLFSSL_SP_NO_256 Removes ECC Single-Precision (SP) optimization for SECP256R1. Only applies to `WOLFSSL_SP_MATH`.

2.5.2 Enabling Features macros (on by default)

2.5.2.1 HAVE_TLS_EXTENSIONS Enables support for TLS extensions, which are required for most TLS builds. Enabled by default with `./configure`, but needs to be manually defined if building with `WOLFSSL_USER_SETTINGS`.

2.5.2.2 HAVE_SUPPORTED_CURVES Enables the TLS supported curves and key share extensions used with TLS. Required with ECC, Curve25519 and Curve448. Enabled by default with `./configure`, but needs to be manually defined if building with `WOLFSSL_USER_SETTINGS`.

2.5.2.3 HAVE_EXTENDED_MASTER Enables extended master secret PRF for calculation of session keys used with TLS v1.2 and older. The PRF method is on by default and is considered more secure. This is on by default if using `./configure`, but needs to be manually defined if building with `WOLFSSL_USER_SETTINGS`.

2.5.2.4 HAVE_ENCRYPT_THEN_MAC Enables encrypt-then-mac support to perform mac after encryption with block ciphers. This is the default and improves security. This is on by default if using `./configure`, but needs to be manually defined if building with `WOLFSSL_USER_SETTINGS`.

2.5.2.5 HAVE_ONE_TIME_AUTH Required if using Chacha20/Poly1305 with TLS v1.2 for setting up Poly authentication. This is on by default with ChaCha20/Poly1305 if using `./configure`, but needs to be manually defined if building with `WOLFSSL_USER_SETTINGS`.

2.5.3 Enabling Features Disabled by Default

2.5.3.1 WOLFSSL_CERT_GEN Turns on wolfSSL's certificate generation functionality. See Keys and Certificates for more information.

2.5.3.2 WOLFSSL_DER_LOAD Allows loading DER-formatted CA certs into the wolfSSL context (`WOLFSSL_CTX`) using the function `wolfSSL_CTX_der_load_verify_locations()`.

2.5.3.3 WOLFSSL_DTLS Turns on the use of DTLS, or datagram TLS. This isn't widely supported or used.

2.5.3.4 WOLFSSL_KEY_GEN Turns on wolfSSL's RSA key generation functionality. See Keys and Certificates for more information.

2.5.3.5 WOLF_PRIVATE_KEY_ID This is used with PKCS11 to enable support for key ID and label API's. FIPS v5 and older doesn't support `WOLF_PRIVATE_KEY_ID` with Crypto Callbacks.

2.5.3.6 WOLFSSL_WOLFSENTRY_HOOKS This switch adds support in the TLS layer for generic network `accept` and `connect` filter hooks, using `wolfSSL_CTX_set_AcceptFilter()` and `wolfSSL_CTX_set_ConnectFilter()`. It also activates wolfSentry integration in the example client and server applications.

2.5.3.7 WOLFSSL_CERT_EXT Certificate extension, key and cert generation feature.

2.5.3.8 WOLFSSL_CERT_REQ Certificate request, key, and cert generation feature.

2.5.3.9 WOLFSSL_SSLKEYLOGFILE This enables the key logging used by Wireshark. It does produce a compiler warning since the master secret and client random are written to a file. This is useful for testing and not recommended for production.

2.5.3.10 WOLFSSL_SSLKEYLOGFILE_OUTPUT This macro defines the filename for the key logging. It is used with `WOLFSSL_SSLKEYLOGFILE`.

2.5.3.11 WOLFSSL_HAVE_WOLFSCEP Enable feature used by autoconf to see if wolfSCEP is available.

2.5.3.12 WOLFSSL_HAVE_MIN This macro is for portability of the library to indicate if MIN/MAX are already defined by the platform. It prevents duplicate definitions.

2.5.3.13 WOLFSSL_HAVE_TLS_UNIQUE Keeps the "Finished" messages after a TLS handshake for use as the "tls-unique" channel binding. Added in libest port: allow applications to get the 'tls-unique' Channel Binding Type (https://tools.ietf.org/html/rfc5929#section-3). This is used in the EST protocol to bind an enrollment to a TLS session through 'proof-of-possession' (https://tools.ietf.org/html/rfc7030#section-3.4 and https://tools.ietf.org/html/rfc7030#section-3.5).

2.5.3.14 WOLFSSL_ENCRYPTED_KEYS Enable for encrypted keys PKCS8 support. This macro enables PKCS8 password based key encryption. Here is a link to RFC PKCS8 documentation (https://datatracker.ietf.org/doc/html/rfc5208).

2.5.3.15 WOLFSSL_CUSTOM_OID Certificate feature that enables custom OID support for subject and request extensions. This also applies to parsing certificates with custom OID.

2.5.3.16 WOLFSSL_RIPEMD Enables RIPEMD-160 support.

2.5.3.17 WOLFSSL_SHA384 Enables SHA-384 support.

2.5.3.18 WOLFSSL_SHA512 Enables SHA-512 support.

2.5.3.19 WOLFSSL_AES_DIRECT Enables direct AES ECB mode support. On its own ECB mode is not considered secure. This feature is required for PKCS7. Warning: In nearly all use cases ECB mode is considered to be less secure. Please avoid using ECB API's directly whenever possible.

2.5.3.20 DEBUG_WOLFSSL Builds in the ability to debug. For more information regarding debugging wolfSSL, see Debugging.

2.5.3.21 HAVE_AESCCM Enables AES-CCM support.

2.5.3.22 HAVE_AESGCM Enables AES-GCM support.

2.5.3.23 WOLFSSL_AES_XTS Enables AES-XTS support.

2.5.3.24 HAVE_CAMELLIA Enables Camellia support.

2.5.3.25 HAVE_CHACHA Enables ChaCha20 support.

2.5.3.26 HAVE_POLY1305 Enables Poly1305 support.

2.5.3.27 HAVE_CRL Enables Certificate Revocation List (CRL) support.

2.5.3.28 HAVE_CRL_IO Enables blocking inline HTTP request on the CRL URL. It will load the CRL into the `WOLFSSL_CTX` and apply it to all WOLFSSL objects created from it.

2.5.3.29 HAVE_ECC Enables Elliptical Curve Cryptography (ECC) support.

2.5.3.30 HAVE_LIBZ Is an extension that can allow for compression of data over the connection. It normally shouldn't be used, see the note below under configure notes libz.

2.5.3.31 OPENSSL_EXTRA Builds even more OpenSSL compatibility into the library, and enables the wolfSSL OpenSSL compatibility layer to ease porting wolfSSL into existing applications which had been designed to work with OpenSSL. It is off by default.

2.5.3.32 HAVE_EXT_CACHE Enables a feature support use of an external session cache (vs an internal one).

2.5.3.33 WOLFSSL_WPAS_SMALL Enables a smaller subset of the compatibility layer for WPA supplicant support.

2.5.3.34 OPENSSL_ALL Enables support for all compatibility functions for testing integration.

2.5.3.35 OPENSSL_COEXIST OpenSSL compat layer. Needs old names disabled. Mode to allow wolfSSL and OpenSSL to exist together.

2.5.3.36 OPENSSL_VERSION_NUMBER Specifies the version number to implement OpenSSL compatibility.

2.5.3.37 WOLFSSL_NGINX OpenSSL compatibility application specific. Use, nginx `(--enable-nginx) WOLFSSL_NGINX`.

2.5.3.38 WOLFSSL_ERROR_CODE_OPENSSL OpenSSL compatibility API wolfSSL_EVP_PKEY_cmp returns 0 on success and -1 on failure. This behavior is different from OpenSSL. EVP_PKEY_cmp returns: 1: two keys match 0: do not match -1: key types are different -2: the operation is not supported If you want this function to behave the same as openSSL, define WOLFSSL_ERROR_CODE_OPENSSL so that WS_RETURN_CODE translates return codes to match OpenSSL equivalent behavior.

2.5.3.39 WOLFSSL_HARDEN_TLS Implement the recommendations specified in RFC9325. This macro needs to be defined to the desired number of bits of security. The currently implemented values are 112 and 128 bits. The following macros disable certain checks. - WOLFSSL_HARDEN_TLS_ALLOW_TRUNCATED_HMAC - WOLFSSL_HARDEN_TLS_ALLOW_OLD_TLS - WOLFSSL_HARDEN_TLS_NO_SCR_CHECK - WOLFSSL_HARDEN_TLS_NO_PKEY_CHECK - WOLFSSL_HARDEN_TLS_ALLOW_ALL_CIPHERSUITES

2.5.3.40 WOLFSSL_ASIO OpenSSL compatibility specific macro.

2.5.3.41 WOLFSSL_QT OpenSSL compatibility specific. Enable DH Extra for QT, OpenSSL all, OpenSSH, and static ephemeral.

2.5.3.42 WOLFSSL_HAPROXY OpenSSL compatibility specific macro.

2.5.3.43 WOLFSSL_ASN_TEMPLATE Use newer ASN template asn.c code (default). Daul algo certificate features. Dual alg cert support requires the ASN.1 template feature.

2.5.3.44 WOLFSSL_ASYNC_IO Used in async cleanup.

2.5.3.45 WOLFSSL_ATMEL Enables ASF hooks seeding random data using the `atmel_get_random_number` function.

2.5.3.46 WOLFSSL_CMAC Additional CMAC algorithm enable. Note: requires `WOLFSSL_AES_DIRECT`.

2.5.3.47 WOLFSSL_ESPIDF_ERROR_PAUSE Used only in test.c and on test error adds a delay for debugging purposes.

2.5.3.48 TEST_IPV6 Turns on testing of IPv6 in the test applications. wolfSSL proper is IP neutral, but the testing applications use IPv4 by default.

2.5.3.49 TEST_NONBLOCK_CERTS Used only for testing a non-blocking OCSP response. Enabled with `WOLFSSL_NONBLOCK_OCSP` and `OCSP_WANT_READ`.

2.5.3.50 TEST_OPENSSL_COEXIST Use when enabling the build option: `./configure --enable-opensslcoexist`.

2.5.3.51 TEST_PK_PRIVKEY Used for testing PK callbacks only. In wolfssl/test.h it uses the context to pass the actual private key which is loaded and used in the PK callback.

2.5.3.52 TEST_BUFFER_SIZE Allows overriding the TLS benchmarking test buffer size used with the example client/server `-B` option.

2.5.3.53 FORCE_BUFFER_TEST Forces use of the test_certs.h buffers instead of using the file system. Used for internal testing only in wolfssl/test.h.

2.5.3.54 WOLFSSL_FORCE_MALLOC_FAIL_TEST Define for internal testing to induce random malloc failures.

2.5.3.55 WOLFSSL_POST_HANDSHAKE_AUTH TLS extension, Used for post-handshake authentification.

2.5.3.56 WOLFSSL_PSK_MULTI_ID_PER_CS With TLS 1.3 PSK, when `WOLFSSL_PSK_MULTI_ID_PER_CS` is defined, multiple IDs for a cipher suite can be handled.

2.5.3.57 WOLFSSL_PUBLIC_ASN This exposes the ASN.1 API's publicly that are used internally. This is useful for customers who want to use the internal asn.h API's to parse.

2.5.3.58 WOLFSSL_QUIC Enables support for QUIC protocol. See (https://github.com/wolfSSL/wolfssl/blob/master for more information.

2.5.3.59 WOLFSSL_QUIC_MAX_RECORD_CAPACITY Defines max quic capacity as 1024*1024 – 1 MB.

2.5.3.60 WOLFSSL_RENESAS_FSPSM_TLS Not yet supported TLS related capabilities.

2.5.3.61 WOLFSSL_RENESAS_TSIP_TLS This is for disabling only the TSIP TLS-linked common key encryption method.

2.5.3.62 WOLFSSL_SM2 Define to use SM ciphers.

2.5.3.63 WOLFSSL_SM3 Define to use SM ciphers.

2.5.3.64 WOLFSSL_SM4 Define to use SM ciphers.

2.5.3.65 WOLFSSL_SM4_CBC SM setting for SM4 CBC.

2.5.3.66 WOLFSSL_SM4_CCM SM settings for SM4 CCM.

2.5.3.67 WOLFSSL_SM4_GCM SM settings for SM4 GCM.

2.5.3.68 WOLFSSL_SNIFFER_CHAIN_INPUT The Chain Input option allows the sniffer to receive its input as a struct iovec list. Rather than a pointer to a raw packet.

2.5.3.69 XSLEEP_MS Used for testing only. It allows defining a custom delay.

2.5.3.70 XSNPRINTF Allows overriding the snprintf function.

2.5.3.71 DEFAULT_TIMEOUT_SEC Used with `HAVE_IO_TIMEOUT` to specify the wolfio.c socket timeout in seconds. This is used by the internal socket code for OCSP and CRL HTTP.

2.5.3.72 HAVE_IO_TIMEOUT Certificate revocation. IO options enable support for connect timeout, but the default is off.

2.5.3.73 HAVE_OCSP Enables Online Certificate Status Protocol (OCSP) support.

2.5.3.74 HAVE_CSHARP Turns on configuration options needed for C# wrapper.

2.5.3.75 HAVE_CURVE25519 Turns on the use of curve25519 algorithm.

2.5.3.76 HAVE_ED25519 Turns on use of the ed25519 algorithm.

2.5.3.77 WOLFSSL_DH_CONST Turns off use of floating point values when performing Diffie Hellman operations and uses tables for `XPOW()` and `XLOG()`. Removes dependency on external math library.

2.5.3.78 WOLFSSL_TRUST_PEER_CERT Turns on the use of trusted peer certificates. This allows for loading in a peer certificate to match with a connection rather than using a CA. When turned on if a trusted peer certificate is matched than the peer cert chain is not loaded and the peer is considered verified. Using CAs is preferred.

2.5.3.79 WOLFSSL_STATIC_MEMORY Turns on the use of static memory buffers and functions. This allows for using static memory instead of dynamic.

2.5.3.80 WOLFSSL_STATIC_MEMORY_LEAN It requires WOLFSSL_STATIC_MEMORY to be defined. It uses smaller type sizes for structs requiring memory pool sizes of less than 65k and limits features available, like IO buffers, to reduce footprint size.

2.5.3.81 WOLFSSL_SESSION_EXPORT Turns on the use of DTLS session export and import. This allows for serializing and sending/receiving the current state of a DTLS session.

2.5.3.82 WOLFSSL_ARMASM Turns on the use of ARMv8 hardware acceleration.

2.5.3.83 WC_RSA_NONBLOCK Turns on fast math RSA non-blocking support for splitting RSA operations into smaller chunks of work. Feature is enabled by calling wc_RsaSetNonBlock() and checking for FP_WOULDBLOCK return code.

2.5.3.84 WC_RSA_BLINDING Used to enable timing resistance.

2.5.3.85 WC_RSA_PSS Enables RSA PSS padding. The only TLS 1.3 RSA padding scheme supported is PSS (per specification). PSS padding uses random padding.

2.5.3.86 WOLFSSL_RSA_VERIFY_ONLY Turns on small build for RSA verify only use. Should be used with the macros WOLFSSL_RSA_PUBLIC_ONLY, WOLFSSL_RSA_VERIFY_INLINE, NO_SIG_WRAPPER, and WOLFCRYPT_ONLY.

2.5.3.87 WOLFSSL_RSA_PUBLIC_ONLY Turns on small build for RSA public key only use. Should be used with the macro WOLFCRYPT_ONLY.

2.5.3.88 WOLFSSL_SHA3 Turns on build for SHA3 use. This is support for SHA3 Keccak for the sizes SHA3-224, SHA3-256, SHA3-384 and SHA3-512. In addition WOLFSSL_SHA3_SMALL can be used to trade off performance for resource use.

2.5.3.89 USE_ECDSA_KEYSZ_HASH_ALGO Will choose a hash algorithm that matches the ephemeral ECDHE key size or the next highest available. This workaround resolves issues with some peers that do not properly support scenarios such as a P-256 key hashed with SHA512.

2.5.3.90 WOLFSSL_ALT_CERT_CHAINS Allows CA's to be presented by peer, but not part of a valid chain. Default wolfSSL behavior is to require validation of all presented peer certificates. This also allows loading intermediate CA's as trusted and ignoring no signer failures for CA's up the chain to root. The alternate certificate chain mode only requires that the peer certificate validate to a trusted CA.

2.5.3.91 WOLFSSL_SYS_CA_CERTS Allows wolfSSL to use trusted system CA certificates for verification when wolfSSL_CTX_load_system_CA_certs() is called, either by loading them into wolfSSL certificate manager, or by invoking system authentication APIs. See wolfSSL_CTX_load_system_CA_certs() for more details. This preprocessor macro is automatically set by the --enable-sys-ca-certs configure option.

2.5.3.92 WOLFSSL_APPLE_NATIVE_CERT_VERIFICATION Enables the use of Apple's native trust APIs when authenticating TLS peer certificates. Requires WOLFSSL_SYS_CA_CERTS to be defined. This macro does not need to be set by the user if building with configure or CMake on iOS or other apple devices, but should be explicitly set on MacOS if you wish to use the native verification methods.

2.5.3.93 WOLFSSL_CUSTOM_CURVES Allow non-standard curves. Includes the curve "a" variable in calculation. Additional curve types can be enabled using HAVE_ECC_SECPR2, HAVE_ECC_SECPR3, HAVE_ECC_BRAINPOOL and HAVE_ECC_KOBLITZ.

2.5.3.94 HAVE_COMP_KEY Enables ECC compressed key support.

2.5.3.95 WOLFSSL_EXTRA_ALERTS Enables additional alerts to be sent during a TLS connection. This feature is also enabled automatically when --enable-opensslextra is used.

2.5.3.96 WOLFSSL_DEBUG_TLS Enables additional debugging print outs during a TLS connection

2.5.3.97 HAVE_BLAKE2 Enables Blake2s algorithm support

2.5.3.98 HAVE_FALLBACK_SCSV Enables Signaling Cipher Suite Value(SCSV) support on the server side. This handles the cipher suite 0x56 0x00 sent from a client to signal that no downgrade of TLS version should be allowed.

2.5.3.99 HAVE_AEAD Implements the use of AEAD and is required for TLS 1.3.

2.5.3.100 HAVE_AES_CBC Enable option for AES CBC.

2.5.3.101 HAVE_ALPN Crypto enables the option for application-layer protocol negotiation.

2.5.3.102 HAVE_CAVIUM_OCTEON_SYNC This enables the blocking (synchronous) version of the Marvell Cavium/Octeon hardware.

2.5.3.103 HAVE_CERTIFICATE_STATUS_REQUEST Used for Certificate revocation as a cert status request feature.

2.5.3.104 HAVE_CERTIFICATE_STATUS_REQUEST_V2 Used for Certificate revocation as a cert status request feature.

2.5.3.105 HAVE_IO_TIMEOUT Certificate revocation. IO options enable support for connect timeout, but the default is off.

2.5.3.106 HAVE_CURL Used for building a subset of the wolfSSL library when linking with cURL.

2.5.3.107 HAVE_CURVE448 Define for Curve448 support. Additional macro settings can be changed. The default is to enable shared secret, key export, and import.

2.5.3.108 HAVE_DANE This option is only supported with HAVE_RPK (Raw Public Keys) and is a placeholder for when it might be added in the future.

2.5.3.109 HAVE_DILITHIUM Enable to include DILITHIUM post quantum cryptography/signature algo.

2.5.3.110 HAVE_ED25519_KEY_IMPORT ED25519 config. Enables Ed255519 and Curve25519 options for granular control of sign, verify, shared secret, import, and export.

2.5.3.111 HAVE_EX_DATA Enable "extra" EX data APIs for user information in CTX/WOLFSSL.

2.5.3.112 HAVE_EX_DATA_CLEANUP_HOOKS Set the extra data and cleanup callback against the RSA key at an index.

2.5.3.113 HAVE_FALCON Enables post-quantum crypto FALCON from OpenQuantumSafe.

2.5.3.114 HAVE_FIPS Used when implementing different FIPS versions.

2.5.3.115 HAVE_KEYING_MATERIAL Enables exporting keying material based on section 7.5 of RFC 8446.

2.5.3.116 HAVE_OID_DECODING Included in ASN template code. Used to decode in some cases.

2.5.3.117 HAVE_MAX_FRAGMENT Sets maximum fragment size. TLS extension.

2.5.3.118 WOLFSSL_PSK_ONE_ID Enables support for only one PSK ID with TLS 1.3.

2.5.3.119 SHA256_MANY_REGISTERS A SHA256 version that keeps all data in registers and partially unrolls loops.

2.5.3.120 WOLFCRYPT_HAVE_SRP Enables wolfCrypt secure remote password support

2.5.3.121 WOLFSSL_MAX_STRENGTH Enables the strongest security features only and disables any weak or deprecated features. This results in slower performance due to near constant-time execution to protect against timing-based side-channel attacks.

2.5.3.122 MAX_RECORD_SIZE Determine maximum record size. 2^14 is the max size by standard.

2.5.3.123 MAX_CERTIFICATE_SZ Defines the max size of a certificate message payload assumes MAX_CHAIN_DEPTH number of certificates at 2kb per certificate.

2.5.3.124 MAX_CHAIN_DEPTH Defines the max chain depth.

2.5.3.125 MAX_CIPHER_NAME Defines max cipher name.

2.5.3.126 MAX_DATE_SIZE Defines max size of date either used as byte lastdate, or byte nextdate.

2.5.3.127 MAX_EARLY_DATA_SZ Used to define the maximum early data size.

2.5.3.128 WOLFSSL_MAX_SEND_SZ Define to specify max send size.

2.5.3.129 WOLFSSL_MAX_SUITE_SZ Define to specify max suite size. If too small error out.

2.5.3.130 MAX_WOLFSSL_FILE_SIZE 4 MB allocated size limit.

2.5.3.131 WOLFSSL_MAXQ10XX_TLS Lets maxq10xx know what TLS version we are using.

2.5.3.132 WOLFSSL_MAX_SIGALGO Enables the ability to override maximum signature algorithms.

2.5.3.133 WOLFSSL_MEM_GUARD Can assign a specified memory guard.

2.5.3.134 WOLFSSL_STATIC_EPHEMERAL TLS sniffer support.

2.5.3.135 SSL_SNIFFER_EXPORTS WIN32 sniffer export.

2.5.3.136 WOLFSSL_SNIFFER_KEYLOGFILE The SSL Keylog file option enables the sniffer to decrypt TLS traffic using the master secret obtained from a NSS keylog file. This allows the sniffer to decrypt all TLS traffic, even for TLS connections using ephemeral cipher suites. Keylog file sniffing is supported for TLS versions 1.2 and 1.3. WolfSSL can be configured to export a keylog file using the `--enable-keylog-export` configure option, independently from the sniffer feature (NOTE: never do this in a production environment, as it is inherently insecure). To enable sniffer support for keylog files, use the following configure command line and build as before: ./configure –enable-sniffer CPPFLAGS=-DWOLFSSL_SNIFFER_KEYLOGFILE'.

2.5.3.137 WOLFSSL_SNIFFER_STORE_DATA_CB The Store Data Callback option allows the sniffer to take a callback that is called when storing the application data into a custom buffer rather than into the reallocated data pointer. The callback is called in a loop until all data is consumed. To enable this option, use the following configure command line and build as before: `./configure --enable-sniffer CPPFLAGS=-DWOLFSSL_SNIFFER_STORE_DATA_CB`.

2.5.3.138 WOLFSSL_SNIFFER_WATCH The Session Watching option allows the sniffer to watch any packet provided without initial setup. It will start to decode all TLS sessions and when the server's certificate is detected, the certificate is given to a callback function provided by the user which should provide the appropriate private key. To enable this option, use the following configure command line and build as before: `./configure --enable-sniffer CPPFLAGS=-DWOLFSSL_SNIFFER_WATCH`.

2.5.3.139 STATIC_BUFFER_LEN Don't fragment memory from the record header. Expands to: `RECORD_HEADER_SZ`.

2.5.3.140 STATIC_CHUNKS_ONLY The user has the option to turn off the 16K output option if we are using small static buffers (the default) and SSL_write tries to write data larger than the record we have, dynamically getting it unless the user says only write in static buffer chunks.

2.5.3.141 WOLFSSL_DEF_PSK_CIPHER Enables user-defined PSK cipher.

2.5.3.142 WOLFSSL_OLD_PRIME_CHECK Enable feature which uses faster DH and RSA prime checking.

2.5.3.143 WOLFSSL_STATIC_RSA Static ciphers are strongly discouraged and should never be used if avoidable. However there are still legacy systems that ONLY support static cipher suites. To that end if you need to connect to a legacy peer only supporting static RSA cipher suites use this to enable support for static RSA in wolfSSL. (See also `WOLFSSL_STATIC_PSK` and `WOLFSSL_STATIC_DH`)

2.5.3.144 WOLFSSL_STATIC_PSK Feature that enables static PSK cipher suites. Static ciphers are highly discouraged see WOLFSSL_STATIC_RSA

2.5.3.145 WOLFSSL_STATIC_DH Feature that enables static ECDH cipher suites. Static ciphers are highly discouraged see WOLFSSL_STATIC_RSA

2.5.3.146 HAVE_NULL_CIPHER Turns on support for NULL ciphers. This option is highly discouraged from a security standpoint however some systems are too small to perform encrypt/decrypt operations and it is better to at least authenticate messages and peers to prevent message tampering than nothing at all!

2.5.3.147 HAVE_ANON Turns on support for anonymous cipher suites. (Never recommended, some valid use cases involving closed or private networks detached from the web)

2.5.3.148 HAVE_LIBOQS Turn on support for the OpenQuantumSafe team's liboqs integration. Please see the appendix "Experimenting with Quantum-Safe Cryptography" in this document for more details.

2.5.3.149 WOLFSSL_SP_4096 Enable RSA/DH 4096-bit Single-Precision (SP) support.

2.5.3.150 WOLFSSL_SP_384 Enable ECC SECP384R1 Single-Precision (SP) support. Only applies to WOLFSSL_SP_MATH.

2.5.3.151 WOLFSSL_SP_1024 Enable SAKKE pairing based cryptography Single-Precision (SP) support.

2.5.3.152 ATOMIC_USER Enable Atomic Record Layer callbacks.

2.5.3.153 BIG_ENDIAN_ORDER Endianness - defaults to little endian, i.e., makes big-endian.

2.5.3.154 WOLFSSL_32BIT_MILLI_TIME Function TimeNowInMilliseconds() returns an unsigned 32-bit value. The default behavior is to return a signed 64-bit value.

2.5.3.155 WOLFSSL_MAX_DHKEY_BITS DH maximum bit size must be a multiple of 8. DH maximum bit size must not exceed 16384 or greater than WOLFSSL_MIN_DHKEY_BITS.

2.5.3.156 WOLFSSL_MIN_DHKEY_BITS The DH minimum bit size must be a multiple of 8. For 112 bits of security, DH needs at least 2048-bit keys, and the minimum bit size must not be greater than 16000.

2.5.3.157 WOLFSSL_MAX_MTU Max expected MTU. 1500 - 100 bytes to account for UDP and IP headers.

2.5.3.158 IGNORE_NETSCAPE_CERT_TYPE Define to input netscape cert type but holds a place.

2.5.3.159 SESSION_CERTS TLS session cache for certs.

2.5.3.160 WOLFSSL_DUAL_ALG_CERTS Dual algorithm certificate required feature.

2.5.3.161 CRL_MAX_REVOKED_CERTS Specifies the number of buffers to hold RevokedCerts. The default value is set to 4.

2.5.3.162 CRL_STATIC_REVOKED_LIST Enables a fixed static list of RevokedCerts to allow for a binary search.

2.5.3.163 SESSION_INDEX Identifies the session's location in the cache. Specifies index session/row shifts.

2.5.3.164 SESSION_TICKET_HINT_DEFAULT The ticket hint default is used to set the default hint value. The Ticket Key lifetime must be longer than the ticket life hint.

2.5.3.165 WOLFSSL_DTLS13 Enable wolfSSL DTLS 1.3.

2.5.3.166 WOLFSSL_TLS13 Enable TLS 1.3 protocol implementation.

2.5.3.167 WOLFSSL_TLS13_IGNORE_AEAD_LIMITS Limits specified by https://www.rfc-editor.org/rfc/rfc9147.html#
aead-limits. We specify the limit by which we need to do a key update as the halfway point to the hard decryption fail limit.

2.5.3.168 WOLFSSL_TLS13_MIDDLEBOX_COMPAT Enable middlebox compatibility in the TLS 1.3 handshake. This includes sending ChangeCipherSpec before encrypted messages and including a session ID.

2.5.3.169 WOLFSSL_TLS13_SHA512 Allow generation of SHA-512 digests in handshake - no ciphersuite requires SHA-512 at this time. This enables calculation of a SHA2-512 hash for the handshake messages even though its not used by TLS v1.3 yet.

2.5.3.170 WOLFSSL_UIP When CONTIKI is defined, it is an implementation of UIP.

2.5.3.171 TLS13_MAX_TICKET_AGE Specifies Max ticket age. For TLS 1.3, this is 7 days.

2.5.3.172 TLS13_TICKET_NONCE_STATIC_SZ TLS13_TICKET_NONCE_STATIC_SZ is not supported in this `FIPS_VERSION_GE`.

2.5.3.173 TLS13_TICKET_NONCE_MAX_SZ Defines version max size for ticket nonce. Max size is defined as 255 bytes.

2.5.3.174 WOLFSSL_TICKET_ENC_AES128_GCM Use AES128-GCM to encrypt/decrypt session tickets in the default callback. This is server-only. If ChaCha20/Poly1305 is not compiled, this is the default algorithm.

2.5.3.175 WOLFSSL_TICKET_ENC_AES256_GCM Use AES256-GCM to encrypt/decrypt session tickets in default callback. Server only.

2.5.3.176 WOLFSSL_TICKET_ENC_CHACHA20_POLY1305 Use ChaCha20-Poly1305 to encrypt/decrypt session tickets in the default callback. If none are defined, the default algorithm is used, and algorithms are compiled. This is server-only.

2.5.3.177 WOLFSSL_TICKET_EXTRA_PADDING_SZ Defines ticket extra padding size defined as 32.

2.5.3.178 WOLFSSL_TICKET_HAVE_ID Use to make sure the ticket has ID. Only add to the cache when support is built in and when the ticket contains an ID. Otherwise we have no way to actually retrieve a ticket from the cache.

2.5.3.179 WOLFSSL_TICKET_KEY_LIFETIME The default lifetime is 1 hour from the issue of the first ticket with the key. It must be longer than a hint.

2.5.3.180 WOLFSSL_TICKET_NONCE_MALLOC Enable dynamic allocation of ticket nonces. Need to disable the HKDF expand callbacks.

2.5.3.181 SHOW_CERTS Show certs will output certs when defined. Use for embedded debugging.

2.5.3.182 SHOW_SECRETS Used for debugging. It will show applicable secrets.

2.5.3.183 DEBUG_UNIT_TEST_CERTS Used when debugging name constraint tests. Not static to allow use in multiple locations with complex define guards.

2.5.3.184 DEBUG_WOLFSSL_VERBOSE When using the `OPENSSL_EXTRA` or `DEBUG_WOLFSSL_VERBOSE` macro, `WOLFSSL_ERROR` is mapped to the new function `WOLFSSL_ERROR_LINE`, which gets the line number and function name where `WOLFSSL_ERROR` is called.

2.5.3.185 SOCKET_INVALID Used to define an invalid socket and is defined as -1.

2.5.3.186 WOLFSSL_SOCKET_INVALID Used for testing and it only allows overriding the value used to indicate an invalid socket. Typically is -1.

2.5.3.187 WOLFSSL_SOCKET_IS_INVALID Used in socket handling.

2.5.3.188 WOLFSSL_SRTP Used to activate SRTP.

2.5.3.189 WOLFSSL_CIPHER_CHECK_SZ Defined as cipher check size which needs 64-bits to confirm encrypt operation worked.

2.5.3.190 DTLS_CID_MAX_SIZE DTLS 1.3 parsing code copies the record header in a static buffer to decrypt the record. Increasing the CID max size also increases this buffer, impacting the per-session runtime memory footprint. The max size for DTLS CID is 255 bytes.

2.5.3.191 DTLS13_EPOCH_SIZE Portability improvement with DTLS 1.3 epoch. Implements a way to save the key bound to a DTLS epoch and setting the right key/epoch when needed.

COPYRIGHT ©2024 wolfSSL Inc.

2.5.3.192 DTLS13_RETRANS_RN_SIZE Portability improvement with DTLS 1.3. Used in DTLS 1.3 to identify size before retransmission.

2.5.3.193 WOLFSSL_DTLS_FRAG_POOL_SZ Defines the allowed number of fragments per specified time.

2.5.3.194 WOLFSSL_CLIENT_SESSION_DEFINED Declare opaque struct for API to use.

2.5.3.195 WOLFSSL_COND Defined if this system supports signaling COND_TYPE - type that should be passed into the signaling API.

2.5.3.196 WOLFSSL_DTLS_CH_FRAG Allows a server to process a fragmented second/verified (one containing a valid cookie response) ClientHello message. The first/unverified (one without a cookie extension) ClientHello MUST be unfragmented so that the DTLS server can process it statelessly. This is only implemented for DTLS 1.3. The user MUST call `wolfSSL_dtls13_allow_ch_frag()` on the server to explicitly enable this during runtime. Note: Using DTLS 1.3 + pqc without `WOLFSSL_DTLS_CH_FRAG` will probably fail In this case use `--enable-dtls-frag-ch` to enable it.

2.5.3.197 WOLFSSL_DTLS_MTU_ADDITIONAL_READ_BUFFER We need additional bytes to read so that we can work with a peer who has a slightly different MTU than us.

2.5.3.198 WOLFSSL_DTLS_WINDOW_WORDS Used to check storage size or to verify if an index is valid for window.

2.5.3.199 WOLFSSL_EXPORT_SPC_SZ Define to specify the amount of bytes used from CipherSpecs.

2.5.3.200 WOLFSSL_MIN_DOWNGRADE Specifies minimum downgrade version.

2.5.3.201 WOLFSSL_MIN_DTLS_DOWNGRADE Specifies minimum DTLS downgrade version.

2.5.3.202 WOLFSSL_MIN_ECC_BITS Can set minimum ECC key size allowed.

2.5.3.203 WOLFSSL_MIN_RSA_BITS By default, wolfSSL restricts RSA key sizes to 1024-bits minimum. To allow the decoding of smaller, less secure RSA keys like 512-bit keys, you will need to add the compiler flag `-DWOLFSSL_MIN_RSA_BITS=512` to CFLAGS or CPPFLAGS or define it in your user_settings header.

2.5.3.204 WOLFSSL_MODE_AUTO_RETRY_ATTEMPTS Used to limit the possibility of an infinite retry loop.

2.5.3.205 WOLFSSL_MULTICAST DTLS multicast feature.

2.5.3.206 WOLFSSL_MULTICAST_PEERS Multicast feature defined as max allowed 100 peers.

2.5.3.207 WOLFSSL_NAMES_STATIC Uses static ECC structs for Position Independent Code (PIC).

2.5.3.208 WOLFSSL_SEND_HRR_COOKIE TLS extension used by DTLS 1.3.

2.5.3.209 WOLFSSL_SEP Feature certificate policy set extension.

2.5.3.210 WOLFSSL_SESSION_ID_CTX Used to copy over application session context ID.

2.5.3.211 WOLFSSL_SESSION_TIMEOUT Default session resumption cache timeout in seconds is used to define timeout manually.

2.5.3.212 KEEP_OUR_CERT Used to ensure the ability to return SSL certificate.

2.5.3.213 KEEP_PEER_CERT Retains peer certs. Parts of the OpenSSL compatibility layer require peer certs.

2.5.3.214 WOLFSSL_SIGNER_DER_CERT This enables retention of the DER/ASN.1 used for signing. This is used by the compatibility layer an example of this is `wolfSSL_X509_STORE_get1_certs`.

2.5.3.215 CA_TABLE_SIZE Used by the wolfSSL Certificate Manager signer table. The default `CA_TABLE_SIZE` is 11, but this can be adjusted based on actual needs. Each `WOLFSSL_CTX` has its own Certificate Manager (CM).

2.5.3.216 ECDHE_SIZE Define to allow this to be overridden at compile-time. ECDHE server size defaults to 256 bits, which can set a predetermined ECDHE curve size. The default is 32 bytes.

2.5.3.217 CIPHER_NONCE It is used as a cryptographic number, which is implemented in authentication. It is a pseudo-random number, which is an integrity-only cipher suite.

2.5.3.218 WOLFSSL_USE_POPEN_HOST Uses popen for creating socket with host and post for wolfio.c socket open code used with CRL and OCSP.

2.5.3.219 CloseSocket A way to override the function used for closing a socket. Used with CRL, OCSP and BIO.

2.5.3.220 CONFIG_POSIX_API Enables POSIX names for networking systems calls.

2.5.3.221 WOLFSSL_USER_CURRTIME Add in the option to use in test.h without the gettimeofday function using the macro `WOLFSSL_USER_CURRTIME`.

2.5.3.222 WOLFSSL_USER_MUTEX Option for user-defined mutexes with `WOLFSSL_USER_MUTEX`.

2.5.3.223 DEFAULT_MIN_ECCKEY_BITS Identifies the minimum number of bits in ECCkey.

2.5.3.224 DEFAULT_MIN_RSAKEY_BITS Identifies the minimum number of bits in RSA key.

2.5.3.225 EXTERNAL_SERIAL_SIZE A raw serial number byte that writes X509 serial numbers in unsigned binary to a buffer. For all cases, the buffer needs to be at least `EXTERNAL_SERIAL_SIZE` (32). On success, it returns `WOLFSSL_SUCCESS`. Note: this is a internal macro that cannot be user defined.

2.5.3.226 LARGE_STATIC_BUFFERS Embedded callbacks require large static buffers; make sure it gives the option to enable larger buffers to 16K.

2.5.3.227 LIBWOLFSSL_VERSION_STRING This is the wolfSSL version string populated for release bundles or when `./configure` is run. There is also a 32-bit HEX version of this in `LIBWOLFSSL_VERSION_HEX`. These come from wolfssl/version.h.

2.5.4 Customizing or Porting wolfSSL

2.5.4.1 WOLFSSL_USER_SETTINGS If defined, it allows the use of a user-specific settings file. The file must be named `user_settings.h` and exist in the include path. It is included prior to the standard `settings.h` file, so default settings can be overridden.

2.5.4.2 WOLFSSL_CALLBACKS This extension allows debugging callbacks through the use of signals in an environment without a debugger. It is off by default. It can also be used to set up a timer with blocking sockets. Please see Callbacks for more information.

2.5.4.3 WOLF_CRYPTO_CB Enable crypto callback support. This feature is also enabled automatically when `--enable-cryptocb` is used.

2.5.4.4 WOLFSSL_DYN_CERT Allow allocation of subjectCN and publicKey fields when parsing certificates even with WOLFSSL_NO_MALLOC set. If using the WOLFSSL_NO_MALLOC option with RSA certificates the public key needs to be retained for CA's for validate certificates on the peer's certificate. This appears as a ConfirmSignature error -173 BAD_FUNC_ARG, since the ca->publicKey is NULL.

2.5.4.5 WOLFSSL_USER_IO Allows the user to remove automatic setting of the default I/O functions `EmbedSend()` and `EmbedReceive()`. Used for custom I/O abstraction layer (see Abstraction Layers for more details).

2.5.4.6 NO_FILESYSTEM Is used if stdio isn't available to load certificates and key files. This enables the use of buffer extensions to be used instead of the file ones.

2.5.4.7 NO_INLINE Disables the automatic inlining of small, heavily used functions. Turning this on will slow down wolfSSL and actually make it bigger since these are small functions, usually much smaller than function call setup/return. You'll also need to add `wolfcrypt/src/misc.c` to the list of compiled files if you're not using autoconf.

2.5.4.8 NO_DEV_RANDOM Disables the use of the default `/dev/random` random number generator. If defined, the user needs to write an OS-specific `GenerateSeed()` function (found in `wolfcrypt/src/random.c`).

2.5.4.9 NO_MAIN_DRIVER Is used in the normal build environment to determine whether a test application is called on its own or through the testsuite driver application. You'll only need to use it with the test files: `test.c`, `client.c`, `server.c`, `echoclient.c`, `echoserver.c`, and `testsuite.c`

2.5.4.10 NO_WRITEV Disables simulation of `writev()` semantics.

2.5.4.11 SINGLE_THREADED Is a switch that turns off the use of mutexes. wolfSSL currently only uses one for the session cache. If your use of wolfSSL is always single threaded you can turn this on.

2.5.4.12 USER_TICKS Allows the user to define their own clock tick function if time(0) is not wanted. Custom function needs second accuracy, but doesn't have to be correlated to Epoch. See `LowResTimer()` function in `wolfssl_int.c`.

2.5.4.13 USER_TIME Disables the use of time.h structures in the case that the user wants (or needs) to use their own. See `wolfcrypt/src/asn.c` for implementation details. The user will need to define and/or implement `XTIME()`, `XGMTIME()`, and `XVALIDATE_DATE()`.

2.5.4.14 USE_CERT_BUFFERS_256 Enables 256-bit test certificate and key buffers located in `<wolfssl_root>/wolfssl/certs_test.h`. Helpful when testing on and porting to embedded systems with no filesystem.

2.5.4.15 USE_CERT_BUFFERS_1024 Enables 1024-bit test certificate and key buffers located in `<wolfssl_root>/wolfssl/certs_test.h`. Helpful when testing on and porting to embedded systems with no filesystem.

2.5.4.16 USE_CERT_BUFFERS_2048 Enables 2048-bit test certificate and key buffers located in `<wolfssl_root>/wolfssl/certs_test.h`. Helpful when testing on and porting to embedded systems with no filesystem.

2.5.4.17 USE_CERT_BUFFERS_3072 Enables 3072-bit test certificate and key buffers located in `<wolfssl_root>/wolfssl/certs_test.h`. Helpful when testing on and porting to embedded systems with no filesystem.

2.5.4.18 USE_CERT_BUFFERS_4096 Enables 4096-bit test certificate and key buffers located in `<wolfssl_root>/wolfssl/certs_test.h`. Helpful when testing on and porting to embedded systems with no filesystem.

2.5.4.19 USE_CERT_BUFFERS_25519 Enables Ed25519 test certificate and key buffers located in `<wolfssl_root>/wolfssl/certs_test.h`. Helpful when testing on and porting to embedded systems with no filesystem.

2.5.4.20 USE_WOLFSSL_IO This macro enables callbacks to send/recv. You can find an example of it in use here: (https://github.com/wolfSSL/wolfssl-examples/blob/master/tls/client-tls-callback.c#L6)

2.5.4.21 CUSTOM_RAND_GENERATE_SEED Allows user to define custom function equivalent to `wc_GenerateSeed(byte* output, word32 sz)`.

2.5.4.22 CUSTOM_RAND_GENERATE_BLOCK Allows user to define custom random number generation function. Examples of use are as follows.

```
./configure --disable-hashdrbg
CFLAGS="-DCUSTOM_RAND_GENERATE_BLOCK= custom_rand_generate_block"
```

Or

```
/* RNG */
/* #define HAVE_HASHDRBG */
extern int custom_rand_generate_block(unsigned char* output, unsigned int sz);
```

2.5.4.23 NO_PUBLIC_GCM_SET_IV Use this if you have done your own custom hardware port and not provided a public implementation of `wc_AesGcmSetIV()`

2.5.4.24 NO_PUBLIC_CCM_SET_NONCE Use this if you have done your own custom hardware port and not provided a public implementation of `wc_AesGcmSetNonce()`

2.5.4.25 NO_GCM_ENCRYPT_EXTRA Use this if you have done your own custom hardwareport and not provided an implementation of `wc_AesGcmEncrypt_ex()`

2.5.4.26 WOLFSSL_STM32[F1 | F2 | F4 | F7 | L4] Use one of these defines when building for the appropriate STM32 device. Update `wolfssl-root/wolfssl/wolfcrypt/settings.h` section with regards to the wolfSSL porting guide (https://www.wolfssl.com/docs/porting-guide/) as appropriate.

2.5.4.27 WOLFSSL_STM32_CUBEMX When using the CubeMX tool to generate Hardware Abstraction Layer (HAL) API's use this setting to add appropriate support in wolfSSL.

2.5.4.28 WOLFSSL_CUBEMX_USE_LL When using the CubeMX tool to generate APIs there are two options, HAL (Hardware Abstraction Layer) or Low Layer (LL). Use this define to control which headers are include in `wolfssl-root/wolfssl/wolfcrypt/settings.h` in the `WOLFSSL_STM32[F1/F2/F4/F7/L4]` section.

2.5.4.29 NO_STM32_CRYPTO For when an STM32 part does not offer hardware crypto support

2.5.4.30 NO_STM32_HASH For when an STM32 part does not offer hardware hash support

2.5.4.31 NO_STM32_RNG For when an STM32 part does not offer hardware RNG support

2.5.4.32 XTIME_MS Macro to map a function for use to get the time in milliseconds when using TLS 1.3. Example being:

```
extern time_t m2mb_xtime_ms(time_t * timer);
#define XTIME_MS(tl) m2mb_xtime_ms((tl))
```

2.5.4.33 WOLFSSL_CIPHER_TEXT_CHECK Define this to check for possible glitching attack against the AES encrypt operation during a TLS connection.

2.5.4.34 RTTHREAD RT-THREAD macro is used when porting rtthread IoT to wolfSSL.

2.5.4.35 SO_REUSEPORT Allows local address and port reuse.

2.5.4.36 INTIME_RTOS Port setting for INtime RTOS.

2.5.4.37 WOLFSSL_SGX Used when porting to SGX.

2.5.5 Reducing Memory or Code Usage

2.5.5.1 TFM_TIMING_RESISTANT Can be defined when using fast math (`USE_FAST_MATH`) on systems with a small stack size. This will get rid of the large static arrays.

2.5.5.2 ECC_TIMING_RESISTANT This is used as a Timing Resistance feature that enables code in ecc.c to prevent side channel and differential power analysis (DPA) attacks.

2.5.5.3 FUSION_RTOS A Fusion RTOS implementation is used for tickets to represent the difference between when they are first seen and when they are sent. It returns the time in milliseconds as a 32-bit value.

2.5.5.4 WOLFSSL_SMALL_STACK Can be used for devices which have a small stack size. This increases the use of dynamic memory in `wolfcrypt/src/integer.c`, but can lead to slower performance.

2.5.5.5 ALT_ECC_SIZE If using fast math and RSA/DH you can define this to reduce your ECC memory consumption. Instead of using stack for ECC points it will allocate from the heap.

2.5.5.6 ECC_SHAMIR Uses variation of ECC math that is slightly faster, but doubles heap usage.

2.5.5.7 RSA_LOW_MEM When defined CRT is not used which saves on some memory but slows down RSA operations. It is off by default.

2.5.5.8 WOLFSSL_SHA3_SMALL When SHA3 is enabled this macro will reduce build size.

2.5.5.9 WOLFSSL_SMALL_CERT_VERIFY Verify the certificate signature without using DecodedCert. Doubles up on some code but allows smaller peak heap memory usage. Cannot be used with `WOLFSSL_NONBLOCK_OCSP`.

2.5.5.10 GCM_SMALL Option to reduce AES GCM code size by calculating at runtime instead of using tables. Possible options are: `GCM_SMALL`, `GCM_WORD32` or `GCM_TABLE`.

2.5.5.11 CURVED25519_SMALL Defines `CURVE25519_SMALL` and `ED25519_SMALL`.

2.5.5.12 CURVE25519_SMALL Use small memory option for curve25519. This uses less memory, but is slower.

2.5.5.13 ED25519_SMALL Use small memory option for ed25519. This uses less memory, but is slower.

2.5.5.14 USE_SLOW_SHA Reduces code size by not unrolling loops, which reduces performance for SHA.

2.5.5.15 USE_SLOW_SHA256 Reduces code size by not unrolling loops, which reduces performance for SHA. About 2k smaller and about 25% slower.

2.5.5.16 USE_SLOW_SHA512 Reduces code size by not unrolling loops, which reduces performance for SHA. Over twice as small, but 50% slower.

2.5.5.17 ECC_USER_CURVES Allow user to choose ECC curve sizes that are enabled. Only the 256-bit curve is enabled by default. To enable others use `HAVE_ECC192`, `HAVE_ECC224`, etc...

COPYRIGHT ©2024 wolfSSL Inc.

2.5.5.18 WOLFSSL_SP_NO_MALLOC In the SP code, always use stack, no heap XMALLOC()/XREALLOC()/XFREE() calls are made.

2.5.5.19 WOLFSSL_SP_NO_DYN_STACK Disable use of dynamic stack items. Used with small code size and not small stack.

2.5.5.20 WOLFSSL_SP_FAST_MODEXP Compiles in a faster mod_exp implementation at the expense of code size.

2.5.5.21 WC_DISABLE_RADIX_ZERO_PAD Disable printing of leading zero in hexidecimal string output. For example, if this macro is defined, the value 8 will be printed as the string "0x8" but if it is not defined it will be printed as "0x08". Defining this macro can reduce code size.

2.5.5.22 WC_ASN_NAME_MAX This allows overriding the maximum name support for an X.509 certificate field.

2.5.5.23 OPENSSL_EXTRA_X509_SMALL Special small OpenSSL compat layer for certs.

2.5.6 Increasing Performance

2.5.6.1 USE_INTEL_SPEEDUP Enables use of Intel's AVX/AVX2 instructions for accelerating AES, ChaCha20, Poly1305, SHA256, SHA512, ED25519 and Curve25519.

2.5.6.2 WOLFSSL_AESNI Enables use of AES accelerated operations which are built into some Intel and AMD chipsets. When using this define, the `aes_asm.asm` (for Windows with at&t syntax) or `aes_asm.S` file is used to optimize via the Intel AES new instruction set (AESNI).

2.5.6.3 HAVE_INTEL_RDSEED Enable Intel's RDSEED for DRBG seed source.

2.5.6.4 HAVE_INTEL_RDRAND Enable Intel's RDRAND instruction for wolfSSL's random source.

2.5.6.5 FP_ECC Enables ECC Fixed Point Cache, which speeds up repeated operations against same private key. Can also define number of entries and LUT bits using `FP_ENTRIES` and `FP_LUT` to reduce default static memory usage.

2.5.6.6 WOLFSSL_ASYNC_CRYPT This enables support for asynchronous cryptography using hardware based adapters such as the Intel QuickAssist or Marvell (Cavium) Nitrox V. The asynchronous code is not included in the public distribution and is available for evaluation by contacting us via email at facts@wolfssl.com.

2.5.6.7 WOLFSSL_NO_ASYNC_IO This disables the asynchronous I/O networking. Asynchronous I/O is on by default and can take up to around 140 bytes during the handshaking process. If your network interface doesn't return `SOCKET_EWOULDBLOCK` or `SOCKET_EAGAIN` (or `WOLFSSL_CBIO_ERR_WANT_WRITE` for custom I/O callbacks) on writing you can define `WOLFSSL_NO_ASYNC_IO` to have wolfSSL not save the state while building handshake messages.

2.5.7 GCM Performance Tuning

There are 4 variants of GCM performance:

- `GCM_SMALL` - Smallest footprint, slowest (FIPS validated)
- `GCM_WORD32` - Medium (FIPS validated)
- `GCM_TABLE` - Fast (FIPS validated)
- `GCM_TABLE_4BIT` - Fastest (FIPS validated)

2.5.8 wolfSSL's Math Options

There are three math libraries in wolfSSL.

- Big Integer
- Fast Math
- Single Precision Math

When building wolfSSL, only one of these must be used.

Big Integer Library is the most portable option as it is written in C without any assembly. As such it is not optimized for specific architectures. All math variables are instantiated on the heap; minimal stack usage. Unfortunately, Big Integer Library is not timing resistant.

Fast Math Library is a good option. It is implemented using both C and assembly. As such, it has optimizations for specific architectures. All math variables are instantiated on the stack, leading to less heap usage. It can be made timing resistant if the `TFM_TIMING_RESISTANT` macro is defined. We have taken it through FIPS 140-2 and 140-3 certifications.

Single Precision (SP) Math Library is our recommended library. It is implemented using both C and assembly. As such, it has optimizations for specific architectures. All math variables are instantiated on the stack, leading to less heap usage. It is always timing resistant. It is generally optimized for speed at the cost of code size, but is highly configurable to compile out unneeded code. We have taken it through DO-178C certifications.

2.5.8.1 Big Integer Math Library (Deprecation Planned) This library is planned to be deprecated and removed from the wolfSSL/wolfCrypt library by the end of 2023. If desired this can be enabled with `--enable-heapmath` or `CFLAGS=-DUSE_INTEGER_HEAP_MATH`.

Forked from public domain LibTomMath library. For more information about LibTomMath, please see https://www.libtom.net/LibTomMath/. Please note that our fork is considerably more active and secure than the original public domain code.

This library is generally the most portable and easiest to get going with. The negatives to the normal big integer library are that it is slower, it uses a lot of heap memory as all memory is allocated from the heap, requires an `XREALLOC()` implementation, and is not timing resistant. The implementation can be found in `integer.c`.

2.5.8.2 Fast Math

2.5.8.2.1 USE_FAST_MATH Forked from public domain LibTomFastMath library. For more information about LibTomFastMath, please see https://www.libtom.net/TomsFastMath. Please note that our fork is considerably more active and secure than the original public domain code from LibTomFastMath. We have improved performance, security and code quality. Also we have taken the FastMath code through FIPS 140-2 and 140-3 certifications.

The FastMath library uses assembly if possible, and will speed up asymmetric private/public key operations like RSA, DH, and DSA. Assembly inclusion is dependent on compiler and processor combinations.

Some combinations will need additional configure flags and some may not be possible. Help with optimizing FastMath with new assembly routines is available on a consulting basis. See the Architecture-Specific Optimizations.

For FastMath, all memory is allocated on the stack. Because the stack memory usage can be larger when using FastMath, we recommend defining TFM_TIMING_RESISTANT as well when using this option. The FastMath code is timing resistant if TFM_TIMING_RESISTANT is defined. This will reduce some of the large math windows for constant time, which use less memory. This uses less stack because there are no shortcuts and therefore less branching during private key operations. This also makes the implementation more secure as timing attacks are a real threat and can give malicious third parties enough information to reproduce the private key.

On ia32, for example, all of the registers need to be available so high optimization and omitting the frame pointer needs to be taken care of. wolfSSL will add `-O3 -fomit-frame-pointer` to GCC for non debug builds. If you're using a different compiler you may need to add these manually to CFLAGS during configure.

OS X will also need `-mdynamic-no-pic` added to CFLAGS. In addition, if building in shared mode for ia32 on OS X you'll need to pass options to LDFLAGS as well:

```
LDFLAGS="-Wl,-read_only_relocs,warning"
```

This gives warnings for some symbols instead of errors.

FastMath also changes the way dynamic and stack memory are used. The normal math library uses dynamic memory for big integers. FastMath uses fixed size buffers that hold 4096 bit integers by default, allowing for 2048 bit by 2048 bit multiplications. If you need 4096 bit by 4096 bit multiplications then change FP_MAX_BITS in `wolfssl/wolfcrypt/tfm.h`. As FP_MAX_BITS is increased, this will also increase the runtime stack usage since the buffers used in the public key operations will now be larger. FP_MAX_BITS needs to be double the max key size. For example if your biggest key is 2048-bit, FP_MAX_BITS should be 4096 and if it is 4096-bit FP_MAX_BITS should be 8192. If using ECC only this can be reduced to the maximum ECC key size times two. A couple of functions in the library use several temporary big integers, meaning the stack can get relatively large. This should only come into play on embedded systems or in threaded environments where the stack size is set to a low value. If stack corruption occurs with FastMath during public key operations in those environments, increase the stack size to accommodate the stack usage.

If you are enabling FastMath without using the autoconf system, you'll need to define USE_FAST_MATH and add `tfm.c` to the wolfSSL build while removing `integer.c`. Defining ALT_ECC_SIZE will allocate ECC points only from the heap instead of the stack.

2.5.8.2.2 Architecture-Specific Optimizations The following macros can be defined for assembly optimizations with USE_FAST_MATH.

- TFM_ARM
- TFM_SSE2
- TFM_AVR32
- TFM_PPC32
- TFM_PPC64
- TFM_MIPS
- TFM_X86
- TFM_X86_64

If none of these are defined or TFM_NO_ASM is defined, then TFM_ISO will be defined and ISO C portable code will be used.

2.5.8.2.3 Algorithm-Specific Optimizations When enabled, optimized implementations for multiplication and squaring are used for the respective ECC curve.

- TFM_ECC192
- TFM_ECC224
- TFM_ECC256
- TFM_ECC384
- TFM_ECC521

2.5.8.2.4 TFM_SMALL_SET Speed optimization for multiplication of smaller numbers. Includes implementations of 1-16 word Comba multiplication and squaring. Useful for improving the performance of ECC operations.

2.5.8.2.5 TFM_HUGE_SET Speed optimization for multiplication of larger numbers. Includes implementations of 20, 24, 28, 32, 48 and 64 word Comba multiplication and squaring where bit size allows. Useful for improving the performance of RSA/DH/DSA operations.

2.5.8.2.6 TFM_SMALL_MONT_SET Speed optimization for montgomery reduction of smaller numbers on Intel architectures. Includes implementations of 1-16 word Montgomery reduction. Useful for improving the performance of ECC operations.

2.5.8.3 Proprietary Single Precision (SP) Math Support SP math is our recommended default option and has been taken through DO-178C certifications. Use these to speed up public key operations for specific keys sizes and curves that are common. Make sure to include the correct code files such as:

- sp_c32.c
- sp_c64.c
- sp_arm32.c
- sp_arm64.c
- sp_armthumb.c
- sp_cortexm.c
- sp_int.c
- sp_x86_64.c
- sp_x86_64_asm.S
- sp_x86_64_asm.asm

2.5.8.3.1 WOLFSSL_SP Enable Single Precision math support.

2.5.8.4 WOLFSSL_SP_MATH Enable only SP math and algorithms. Eliminates big integer math code such as normal (integer.c) or fast (tfm.c). Restricts key sizes and curves to only ones supported by SP.

2.5.8.5 WOLFSSL_SP_MATH_ALL Enable SP math and algorithms. Implements big integer math code such as normal (integer.c) or fast (tfm.c) for key sizes and curves not supported by SP.

2.5.8.6 WOLFSSL_SP_SMALL If using SP math this will use smaller versions of the code and avoid large stack variables.

2.5.8.6.1 SP_WORD_SIZE Force 32-bit or 64-bit data type for storing one word of a number.

2.5.8.6.2 WOLFSSL_SP_NONBLOCK Enables “non blocking” mode for Single Precision math, which will return FP_WOULDBLOCK for long operations and function must be called again until complete. Currently, this in only supported for ECC and is used in conjunction with WC_ECC_NONBLOCK.

2.5.8.6.3 WOLFSSL_SP_FAST_NCT_EXPTMOD Enables the faster non-constant time modular exponentation implementation. Will only be used for public key operations; not private key operations.

2.5.8.6.4 WOLFSSL_SP_INT_NEGATIVE Allows multi-precision numbers to be negative. (Not required for cryptographic operations.)

2.5.8.6.5 WOLFSSL_SP_INT_DIGIT_ALIGN Enable when unaligned access of sp_int_digit pointer is not allowed.

2.5.8.6.6 WOLFSSL_HAVE_SP_RSA Single Precision RSA for 2048, 3072 and 4096 bit.

2.5.8.6.7 WOLFSSL_HAVE_SP_DH Single Precision DH for 2048, 3072 and 4096 bit.

2.5.8.6.8 WOLFSSL_HAVE_SP_ECC Single Precision ECC for SECP256R1 and SECP384R1.

2.5.8.6.9 WOLFSSL_SP_LARGE_CODE Allows Single-Precision (SP) speedups that come at the cost of larger binary size. Might not be appropriate for some embedded platforms.

2.5.8.6.10 WOLFSSL_SP_DIV_WORD_HALF Indicates that division using a double length word isn't available. For example, on 32-bit CPUs, if you do not want to compile in a 64-bit division from a library, then defining this macro turn on an implementation where the divide is done using half word size portions.

2.5.8.6.11 WOLFSSL_SP_DIV_32 Indicates that 32-bit division isn't available and that wolfSSL should use its own Single-Precision (SP) implementation.

2.5.8.6.12 WOLFSSL_SP_DIV_64 Indicates that 64-bit division isn't available and that wolfSSL should use its own Single-Precision (SP) implementation.

2.5.8.6.13 WOLFSSL_SP_ASM Enables Single-Precision (SP) platform specific assembly code implementation that is faster. Platform is detected.

2.5.8.6.14 WOLFSSL_SP_X86_64_ASM Enable Single-Precision (SP) Intel x64 assembly implementation.

2.5.8.6.15 WOLFSSL_SP_ARM32_ASM Enable Single-Precision (SP) Aarch32 assembly implementation.

2.5.8.6.16 WOLFSSL_SP_ARM64_ASM Enable Single-Precision (SP) Aarch64 assembly implementation.

2.5.8.6.17 WOLFSSL_SP_ARM_CORTEX_M_ASM Enable Single-Precision (SP) Cortex-M family (including Cortex-M4) assembly implementation.

2.5.8.6.18 WOLFSSL_SP_ARM_THUMB_ASM Enable Single-Precision (SP) ARM Thumb assembly implementation (used with -mthumb).

2.5.8.6.19 WOLFSSL_SP_X86_64 Enable Single-Precision (SP) Intel x86 64-bit assembly speedup macros. Only applies if `WOLFSSL_SP_MATH_ALL` is defined. See `sp_int.c`.

2.5.8.6.20 WOLFSSL_SP_X86 Enable Single-Precision (SP) Intel x86 assembly speedup macros. Only applies if `WOLFSSL_SP_MATH_ALL` is defined. See `sp_int.c`.

2.5.8.6.21 WOLFSSL_SP_PPC64 Enable Single-Precision (SP) PPC64 assembly speedup macros. Only applies if `WOLFSSL_SP_MATH_ALL` is defined. See `sp_int.c`.

2.5.8.6.22 WOLFSSL_SP_PPC Enable Single-Precision (SP) PPC assembly speedup macros. Only applies if `WOLFSSL_SP_MATH_ALL` is defined. See `sp_int.c`.

2.5.8.6.23 WOLFSSL_SP_MIPS64 Enable Single-Precision (SP) MIPS64 assembly speedup macros. Only applies if `WOLFSSL_SP_MATH_ALL` is defined. See `sp_int.c`.

2.5.8.6.24 WOLFSSL_SP_MIPS Enable Single-Precision (SP) MIPS assembly speedup macros. Only applies if `WOLFSSL_SP_MATH_ALL` is defined. See `sp_int.c`.

2.5.8.6.25 WOLFSSL_SP_RISCV64 Enable Single-Precision (SP) RISCV64 assembly speedup macros. Only applies if `WOLFSSL_SP_MATH_ALL` is defined. See `sp_int.c`.

2.5.8.6.26 WOLFSSL_SP_RISCV32 Enable Single-Precision (SP) RISCV32 assembly speedup macros. Only applies if `WOLFSSL_SP_MATH_ALL` is defined. See `sp_int.c`.

2.5.8.6.27 WOLFSSL_SP_S390X Enable Single-Precision (SP) S390X assembly speedup macros. Only applies if `WOLFSSL_SP_MATH_ALL` is defined. See `sp_int.c`.

2.5.8.7 SP_INT_BITS The number of bits to have in an sp_int. Which can determine the largest bignum that the library can handle.

2.5.9 Stack or Chip Specific Defines

wolfSSL can be built for a variety of platforms and TCP/IP stacks. Most of the following defines are located in `./wolfssl/wolfcrypt/settings.h` and are commented out by default. Each can be uncommented to enable support for the specific chip or stack referenced below.

2.5.9.1 IPHONE Can be defined if building for use with iOS.

2.5.9.2 THREADX Can be defined when building for use with the ThreadX RTOS (https://www.rtos.com).

2.5.9.3 MICRIUM Can be defined when building wolfSSL to enable support for Micrium's µC/OS-III RTOS (https://www.micrium.com).

2.5.9.4 MBED Can be defined when building for the mbed prototyping platform (https://www.mbed.org).

2.5.9.5 MICROCHIP_PIC32 Can be defined when building for Microchip's PIC32 platform (https://www.microchip.com).

2.5.9.6 MICROCHIP_TCPIP_V5 Can be defined specifically version 5 of microchip tcp/ip stack.

2.5.9.7 MICROCHIP_TCPIP Can be defined for microchip tcp/ip stack version 6 or later.

2.5.9.8 WOLFSSL_MICROCHIP_PIC32MZ Can be defined for PIC32MZ hardware cryptography engine.

2.5.9.9 FREERTOS Can be defined when building for FreeRTOS (https://www.freertos.org). If using LwIP, define `WOLFSSL_LWIP` as well.

2.5.9.10 FREERTOS_WINSIM Can be defined when building for the FreeRTOS windows simulator (https://www.freertos.org).

2.5.9.11 WOLFSSL_CHIBIOS Can be defined when building for ChibiOS RTOS.

2.5.9.12 WOLFSSL_CMSIS_RTOS Can be defined when building for Mbed CMIS-RTOS.

2.5.9.13 WOLFSSL_CMSIS_RTOSv2 Can be defined when building for Mbed CMIS-RTOSv2.

2.5.9.14 WOLFSSL_LWIP_NATIVE Use in platforms for LWIP native.

2.5.9.15 WOLFSSL_DEOS You can enable the wolfSSL support for Deos RTOS available here using this define.

2.5.9.16 WOLFSSL_ESPIDF Can be defined when building for ESP-IDF.

2.5.9.17 WOLFSSL_LINUXKM Use if building for Linux Kernel Module.

2.5.9.18 WORD64_AVAILABLE Portability macro to indicate 64-bit types are supported. Typically its better to use `SIZEOF_LONG_LONG` 8.

2.5.9.19 WOLFSSL_NUCLEUS_1_2 Use if building for Nucleus 1.2.

2.5.9.20 WOLFSSL_PICOTCP use if if building with PicoTCP.

2.5.9.21 WOLFSSL_RENESAS_RA6M3G Use if building with RENESAS RA6M3G.

2.5.9.22 WOLFSSL_RENESAS_RA6M4 use if building with RENESAS RA6M4.

2.5.9.23 WOLFSSL_RIOT_OS Use if building with RIOT-OS.

2.5.9.24 WOLFSSL_uITRON4 Use if building for uITRON4.

2.5.9.25 WOLFSSL_uTKERNEL2 Use if building with uT-Kernel.

2.5.9.26 WOLFSSL_VXWORKS Use if building with VxWorks.

2.5.9.27 DEVKITPRO Use when building for devkitPro.

2.5.9.28 WOLFSSL_VXWORKS_6_x Used only with implementation for VxWorks 6.x only.

2.5.9.29 WOLFSSL_WICED Used if building for WICED Studio.

2.5.9.30 FREESCALE_KSDK_FREERTOS Older name of this is FREESCALE_FREE_RTOS but this is used when building for Freescale KSDK FreeRTOS.

2.5.9.31 FREESCALE_KSDK_MQX Used when building for Freescale KSDK MQX/RTCS/MFS.

2.5.9.32 FREESCALE_MQX_5_0 Used when building for Freescale Classic MQX version 5.0.

2.5.9.33 WOLFSSL_KEIL_TCP_NET Configure the TCP stack (MDK_CONF_NETWORK). By default uses Keil TCP WOLFSSL_KEIL_TCP_NET. Use 0 for none or 2 for user io callbacks.

2.5.9.34 INTEL_GALILEO Used when configuring ARDUINO and wolfSSL. If building for Intel Galileo platform add: `#define INTEL_GALILEO`

2.5.9.35 HAVE_KEIL_RTX WolfSSL for MDK-RTX-TCP-FS Configuration.

2.5.9.36 EBSNET Can be defined when using EBSnet products and RTIP.

2.5.9.37 WOLFSSL_EMBOS Can be defined when building for SEGGER embOS (https://www.segger.com/products/rtos/embos/). If using emNET, define WOLFSSL_EMNET as well.

2.5.9.38 WOLFSSL_EMNET Can be defined when building for SEGGER emNET TCP/IP stack (https://www.segger.com/products/connectivity/emnet/).

2.5.9.39 WOLFSSL_LWIP Can be defined when using wolfSSL with the LwIP TCP/IP stack (https://savannah.nongnu.org/projects/lwip/).

2.5.9.40 WOLFSSL_ISOTP Can be defined when using wolfSSL with the ISO-TP transport protocol, typically used for CAN bus. A usage example can be found in the wolfssl-examples repository.

2.5.9.41 WOLFSSL_GAME_BUILD Can be defined when building wolfSSL for a game console.

2.5.9.42 WOLFSSL_LSR Can be defined if building for LSR.

2.5.9.43 FREESCALE_MQX Can be defined when building for Freescale MQX/RTCS/MFS (https://www.freescale.com). This in turn defines FREESCALE_K70_RNGA to enable support for the Kinetis H/W Random Number Generator Accelerator

2.5.9.44 WOLFSSL_STM32F2 Can be defined when building for STM32F2. This define also enables STM32F2 hardware crypto and hardware RNG support in wolfSSL (https://www.st.com/internet/mcu/subclass/1520.jsp).

2.5.9.45 COMVERGE Can be defined if using Comverge settings.

2.5.9.46 WOLFSSL_QL Can be defined if using QL SEP settings.

2.5.9.47 WOLFSSL_EROAD Can be defined building for EROAD.

2.5.9.48 WOLFSSL_IAR_ARM Can be defined if build for IAR EWARM.

2.5.9.49 WOLFSSL_TIRTOS Can be defined when building for TI-RTOS.

2.5.9.50 WOLFSSL_ROWLEY_ARM Can be defined when building with Rowley CrossWorks.

2.5.9.51 WOLFSSL_NRF51 Can be defined when porting to Nordic nRF51.

2.5.9.52 WOLFSSL_NRF51_AES Can be defined to use built-in AES hardware for AES 128 ECB encrypt when porting to Nordic nRF51.

2.5.9.53 WOLFSSL_CONTIKI Can be defined to enable support for the Contiki operating system.

2.5.9.54 WOLFSSL_APACHE_MYNEWT Can be defined to enable the Apache Mynewt port layer.

2.5.9.55 WOLFSSL_APACHE_HTTPD Can be defined to enable support for the Apache HTTPD web server.

2.5.9.56 ASIO_USE_WOLFSSL Can be defined to make wolfSSL build as an ASIO-compatible version. ASIO then relies on the `BOOST_ASIO_USE_WOLFSSL` preprocessor define.

2.5.9.57 WOLFSSL_CRYPTOCELL Can be defined to enable using ARM CRYPTOCELL.

2.5.9.58 WOLFSSL_SIFIVE_RISC_V Can be defined to enable using RISC-V SiFive/HiFive port.

2.5.9.59 WOLFSSL_MDK_ARM Adds support for MDK ARM

2.5.9.60 WOLFSSL_MDK5 Adds support for MDK5 ARM

2.5.10 OS Specific Defines

2.5.10.1 USE_WINDOWS_API Specify use of windows library APIs' as opposed to Unix/Linux APIs'

2.5.10.2 WIN32_LEAN_AND_MEAN Adds support for the Microsoft win32 lean and mean build.

2.5.10.3 FREERTOS_TCP Adds support for the FREERTOS TCP stack

2.5.10.4 WOLFSSL_SAFERTOS Adds support for SafeRTOS

2.6 Build Options

The following are options which may be appended to the `./configure` script to customize how the wolfSSL library is built.

By default, wolfSSL only builds in shared mode, with static mode being disabled. This speeds up build times by a factor of two. Either mode can be explicitly disabled or enabled if desired.

2.6.1 --enable-debug

Enable wolfSSL debugging support. Enabling debug support allows easier debugging by compiling with debug information and defining the constant `DEBUG_WOLFSSL` which outputs messages to `stderr`. To turn debug on at runtime, call `wolfSSL_Debugging_ON()`. For more information, see Debugging.

2.6.2 --enable-distro

Enable wolfSSL distro build.

2.6.3 --enable-singlethread

Enable single threaded mode, no multi thread protections.

Enabling single threaded mode turns off multi thread protection of the session cache. Only enable single threaded mode if you know your application is single threaded or your application is multithreaded and only one thread at a time will be accessing the library.

2.6.4 --enable-dtls

Enable wolfSSL DTLS support

Enabling DTLS support allows users of the library to also use the DTLS protocol in addition to TLS and SSL. For more information, see the DTLS section.

2.6.5 --disable-rng

Disable compiling and using RNG

2.6.6 --enable-sctp

Enable wolfSSL DTLS-SCTP support

2.6.7 --enable-openssh

Enable OpenSSH compatibility build

2.6.8 --enable-apachehttpd

Enable Apache httpd compatibility build

2.6.9 --enable-openvpn

Enable OpenVPN compatibility build

2.6.10 --enable-opensslextra

Enable extra OpenSSL API compatibility, increases the size

Enabling OpenSSL Extra includes a larger set of OpenSSL compatibility functions. The basic build will enable enough functions for most TLS/SSL needs, but if you're porting an application that uses 10s or 100s of OpenSSL calls, enabling this will allow better support. The wolfSSL OpenSSL compatibility layer is under active development, so if there is a function missing which you need, please contact us and we'll try to help. For more information about the OpenSSL Compatibility Layer, please see OpenSSL Compatibility.

2.6.11 --enable-opensslall

Enable all OpenSSL API.

2.6.12 --enable-maxstrength

Enable Max Strength build, allows TSLv1.2-AEAD-PFS ciphers only. This is disabled by default because it can cause interoperability issues. It also enables glitching detection.

2.6.13 --disable-harden

Disable hardening, timing resistance and RSA blinding. Disabling this feature can give performance improvements.

NOTE Hardening provides mitigations against side channel attacks. Only disable this feature after careful consideration.

To disable via user_settings.h the equivalent settings would be:

- `#define WC_NO_CACHE_RESISTANT`
- `#define WC_NO_HARDEN`
- Remove the setting `WC_RSA_BLINDING` if present or un-define it
- Remove the setting `ECC_TIMING_RESISTANT` if present or un-define it
- Remove the setting `TFM_TIMING_RESISTANT` if present or un-define it

2.6.14 --enable-ipv6

Enable testing of IPv6, wolfSSL proper is IP neutral

Enabling IPV6 changes the test applications to use IPv6 instead of IPv4. wolfSSL proper is IP neutral, either version can be used, but currently the test applications are IP dependent.

2.6.15 --enable-bump

Enable SSL Bump build

2.6.16 --enable-leanpsk

Enable Lean PSK build.

Very small build using PSK, and eliminating many features from the library. Approximate build size for wolfSSL on an embedded system with this enabled is 21kB.

2.6.17 --enable-leantls

Implements a lean TLS 1.2 client only (no client auth), ECC256, AES128 and SHA256 w/o Shamir. Meant to be used by itself at the moment and not in conjunction with other build options.

Enabling produces a small footprint TLS client that supports TLS 1.2 client only (no client auth), ECC256, AES128 and SHA256 w/o Shamir. Meant to be used by itself at the moment and not in conjunction with other build options.

2.6.18 --enable-bigcache

Enable a big session cache.

Enabling the big session cache will increase the session cache from 33 sessions to 20,027 sessions. The default session cache size of 33 is adequate for TLS clients and embedded servers. The big session cache is suitable for servers that aren't under heavy load, basically allowing 200 new sessions per minute or so.

2.6.19 --enable-hugecache

Enable a huge session cache.

Enabling the huge session cache will increase the session cache size to 65,791 sessions. This option is for servers that are under heavy load, over 13,000 new sessions per minute are possible or over 200 new sessions per second.

2.6.20 --enable-smallcache

Enable small session cache.

Enabling the small session cache will cause wolfSSL to only store 6 sessions. This may be useful for embedded clients or systems where the default of nearly 3kB is too much RAM. This define uses less than 500 bytes of RAM.

2.6.21 --enable-savesession

Enable persistent session cache.

Enabling this option will allow an application to persist (save) and restore the wolfSSL session cache to/from memory buffers.

2.6.22 --enable-savecert

Enable persistent cert cache.

Enabling this option will allow an application to persist (save) and restore the wolfSSL certificate cache to/from memory buffers.

2.6.23 --enable-atomicuser

Enable Atomic User Record Layer.

Enabling this option will turn on User Atomic Record Layer Processing callbacks. This will allow the application to register its own MAC/encrypt and decrypt/verify callbacks.

2.6.24 --enable-pkcallbacks

Enable Public Key Callbacks

2.6.25 --enable-sniffer

Enable wolfSSL sniffer support.

Enabling sniffer (SSL inspection) support will allow the collection of SSL traffic packets as well as the ability to decrypt those packets with the correct key file.

Currently the sniffer supports the following RSA ciphers:

CBC ciphers:

- AES-CBC
- Camellia-CBC
- 3DES-CBC

Stream ciphers:

- RC4

2.6.26 --enable-aesgcm

Enable AES-GCM support.

Eabling this option will turn on Public Key callbacks, allowing the application to register its own ECC sign/verify and RSA sign/verify and encrypt/decrypt callbacks.

2.6.27 --enable-aesccm

Enable AES-CCM support

Enabling AES-GCM will add these cipher suites to wolfSSL. wolfSSL offers four different implementations of AES-GCM balancing speed versus memory consumption. If available, wolfSSL will use 64-bit or 32-bit math. For embedded applications, there is a speedy 8-bit version that uses RAM-based lookup tables (8KB per session) which is speed comparable to the 64-bit version and a slower 8-bit version that doesn't take up any additional RAM. The –enable-aesgcm configure option may be modified with the options `=word32`, `=table`, or `=small`, i.e. `--enable-aesgcm=table`.

2.6.28 --disable-aescbc

Used to with `--disable-aescbc` to compile out AES-CBC

AES-GCM will enable Counter with CBC-MAC Mode with 8-byte authentication (CCM-8) for AES.

2.6.29 --enable-aescfb

Turns on AES-CFB mode support

2.6.30 --enable-aesctr

Enable wolfSSL AES-CTR support

Enabling AES-CTR will enable Counter mode.

2.6.31 --enable-aesni

Enable wolfSSL Intel AES-NI support

Enabling AES-NI support will allow AES instructions to be called directly from the chip when using an AES-NI supported chip. This provides speed increases for AES functions. See Features for more details regarding AES-NI.

2.6.32 --enable-intelasm

Enable ASM speedups for Intel and AMD processors.

Enabling the intelasm option for wolfSSL will utilize expanded capabilities of your processor that dramatically enhance AES performance. The instruction sets leveraged when configure option is enabled include AVX1, AVX2, BMI2, RDRAND, RDSEED, AESNI, and ADX. These were first introduced into Intel processors and AMD processors have started adopting them in recent years. When enabled, wolfSSL will check the processor and take advantage of the instruction sets your processor supports.

2.6.33 --enable-camellia

Enable Camellia support

2.6.34 --enable-md2

Enable MD2 support

2.6.35 --enable-nullcipher

Enable wolfSSL NULL cipher support (no encryption)

2.6.36 --enable-ripemd

Enable wolfSSL RIPEMD-160 support

2.6.37 --enable-blake2

Enable wolfSSL BLAKE2 support

2.6.38 --enable-blake2s

Enable wolfSSL BLAKE2s support

2.6.39 --enable-sha3

Enabled by default on x86_64 and Aarch64.

Enables wolfSSL SHA3 support (=small for small build)

2.6.40 --enable-sha512

Enabled by default on x86_64.

Enable wolfSSL SHA-512 support

2.6.41 --enable-sessioncerts

Enable session cert storing

2.6.42 --enable-keygen

Enable key generation (applies to RSA key generation only)

2.6.43 --enable-certgen

Enable cert generation

2.6.44 --enable-certext

Enable cert extensions (see chapter 7 on supported extensions)

2.6.45 --enable-certreq

Enable cert request generation

2.6.46 --enable-sep

Enable SEP extensions

2.6.47 --enable-hkdf

Enable HKDF (HMAC-KDF)

2.6.48 --enable-x963kdf

Enable X9.63 KDF support

2.6.49 --enable-dsa

Enable Digital Signature Algorithm (DSA).

NIST approved digital signature algorithm along with RSA and ECDSA as defined by FIPS 186-4 and are used to generate and verify digital signatures if used in conjunction with an approved hash function as defined by the Secure Hash Standard (FIPS 180-4).

2.6.50 --enable-eccshamir

Enabled by default on x86_64

Enable ECC Shamir

2.6.51 --enable-ecc

Enabled by default on x86_64

Enable ECC.

Enabling this option will build ECC support and cipher suites into wolfSSL.

2.6.52 --enable-ecccustcurves

Enable ECC custom curves (=all to enable all curve types)

2.6.53 --enable-compkey

Enable compressed keys support

2.6.54 --enable-curve25519

Enable Curve25519 (or `--enable-curve25519=small` for CURVE25519_SMALL).

An elliptic curve offering 128 bits of security and to be used with ECDH key agreement (see Cross Compiling). Enabling curve25519 option allows for the use of the curve25519 algorithm. The default curve25519 is set to use more memory but have a faster run time. To have the algorithm use less memory the option `--enable-curve25519=small` can be used. Although using less memory there is a trade off in speed.

2.6.55 --enable-ed25519

Enable ED25519 (or `--enable-ed25519=small` for ED25519_SMALL)

Enabling ed25519 option allows for the use of the ed25519 algorithm. The default ed25519 is set to use more memory but have a faster run time. To have the algorithm use less memory the option `--enable-ed25519=small` can be used. Like with curve25519 using this enable option less is a trade off between speed and memory.

2.6.56 --enable-fpecc

Enable Fixed Point cache ECC

2.6.57 --enable-eccencrypt

Enable ECC encrypt

2.6.58 --enable-psk

Enable PSK (Pre Shared Keys)

2.6.59 --disable-errorstrings

Disable the error strings table

2.6.60 --disable-oldtls

Disable old TLS version < 1.2

2.6.61 --enable-sslv3

Enable SSL version 3.0

2.6.62 --enable-stacksize

Enable stack size info on examples

2.6.63 --disable-memory

Disable memory callbacks

2.6.64 --disable-rsa

Disable RSA

2.6.65 --enable-rsapss

Enable RSA-PSS

2.6.66 --disable-dh

Disable DH

2.6.67 --enable-anon

Enable Anonymous

2.6.68 --disable-asn

Disable ASN

2.6.69 --disable-aes

Disable AES

2.6.70 --disable-coding

Disable Coding base 16/64

2.6.71 --enable-base64encode

Enabled by default on x86_64

Enable Base64 encoding

2.6.72 --disable-des3

Disable DES3

2.6.73 --enable-arc4

Enable ARC4

2.6.74 --disable-md5

Disable MD5

2.6.75 --disable-sha

Disable SHA

2.6.76 --enable-webserver

Enable Web Server.

This turns on functions required over the standard build that will allow full functionality for building with the yaSSL Embedded Web Server.

2.6.77 --enable-fips

Enable FIPS 140-2 (Must have license to implement.)

2.6.78 --enable-sha224

Enabled by default on x86_64

Enable wolfSSL SHA-224 support

2.6.79 --disable-poly1305

Disable wolfSSL POLY1305 support

2.6.80 --disable-chacha

Disable CHACHA

2.6.81 --disable-hashdrbg

Disable Hash DRBG support

2.6.82 --disable-filesystem

Disable Filesystem support.

This makes it easier to disable filesystem use. This option defines `NO_FILESYSTEM`.

2.6.83 --disable-inline

Disable inline functions.

Disabling this option disables function inlining in wolfSSL. Function placeholders that are not linked against but, rather, the code block is inserted into the function call when function inlining is enabled.

2.6.84 --enable-ocsp

Enable Online Certificate Status Protocol (OCSP).

Enabling this option adds OCSP (Online Certificate Status Protocol) support to wolfSSL. It is used to obtain the revocation status of x.509 certificates as described in RFC 6960.

2.6.85 --enable-ocspstapling

Enable OCSP Stapling (=no-multi to disable multiple OCSP Stapling for TLS1.3 Certificate)

2.6.86 --enable-ocspstapling2

Enable OCSP Stapling version 2

2.6.87 --enable-crl

Enable CRL (Certificate Revocation List)

2.6.88 --enable-crl-monitor

Enable CRL Monitor.

Enabling this option adds the ability to have wolfSSL actively monitor a specific CRL (Certificate Revocation List) directory.

2.6.89 --enable-sni

Enable Server Name Indication (SNI).

Enabling this option will turn on the TLS Server Name Indication (SNI) extension.

2.6.90 --enable-maxfragment

Enable Maximum Fragment Length.

Enabling this option will turn on the TLS Maximum Fragment Length extension.

2.6.91 --enable-alpn

Enable Application Layer Protocol Negotiation (ALPN)

2.6.92 --enable-truncatedhmac

Enable Truncated Keyed-hash MAC (HMAC).

Enabling this option will turn on the TLS Truncated HMAC extension.

2.6.93 --enable-renegotiation-indication

Enable Renegotiation Indication.

As described in RFC 5746, this specification prevents an SSL/TLS attack involving renegotiation splicing by tying the renegotiations to the TLS connection they are performed over.

2.6.94 --enable-secure-renegotiation

Enable Secure Renegotiation

2.6.95 --enable-supportedcurves

Enable Supported Elliptic Curves.

Enabling this option will turn on the TLS Supported ECC Curves extension.

2.6.96 --enable-session-ticket

Enable Session Ticket

2.6.97 --enable-extended-master

Enable Extended Master Secret

2.6.98 --enable-tlsx

Enable all TLS extensions.

Enabling this option will turn on all TLS extensions currently supported by wolfSSL.

2.6.99 --enable-pkcs7

Enable PKCS#7 support

2.6.100 --enable-pkcs11

Enable PKCS#11 access

2.6.101 --enable-ssh

Enable wolfSSH options

2.6.102 --enable-scep

Enable wolfSCEP (Simple Certificate Enrollment Protocol)

As defined by the Internet Engineering Task Force, Simple Certificate Enrollment Protocol is a PKI that leverages PKCS#7 and PKCS#10 over HTTP. CERT notes that SCEP does not strongly authenticate certificate requests.

2.6.103 --enable-srp

Enable Secure Remote Password

2.6.104 --enable-smallstack

Enable Small Stack Usage

2.6.105 --enable-valgrind

Enable valgrind for unit tests.

Enabling this option will turn on valgrind when running the wolfSSL unit tests. This can be useful for catching problems early on in the development cycle.

2.6.106 --enable-testcert

Enable Test Cert.

When this option is enabled, it exposes part of the ASN certificate API that is usually not exposed. This can be useful for testing purposes, as seen in the wolfCrypt test application (`wolfcrypt/test/test.c`).

2.6.107 --enable-iopool

Enable I/O Pool example

2.6.108 --enable-certservice

Enable certificate service (Windows Servers)

2.6.109 --enable-jni

Enable wolfSSL JNI

2.6.110 --enable-lighty

Enable lighttpd/lighty

2.6.111 --enable-stunnel

Enable stunnel

2.6.112 --enable-md4

Enable MD4

2.6.113 --enable-pwdbased

Enable PWDBASED

2.6.114 --enable-scrypt

Enable SCRYPT

2.6.115 --enable-cryptonly

Enable wolfCrypt Only build

2.6.116 --disable-examples

Disable building examples.

When enabled, the wolfSSL example applications will be built (client, server, echoclient, echoserver).

2.6.117 --disable-crypttests

Disable Crypt Bench/Test

2.6.118 --enable-fast-rsa

Enable RSA using Intel IPP.

Enabling fast-rsa speeds up RSA operations by using IPP libraries. It has a larger memory consumption then the default RSA set by wolfSSL. If IPP libraries can not be found an error message will be displayed during configuration. The first location that autoconf will look is in the directory `wolfssl_root/IPP` the second is standard location for libraries on the machine such as `/usr/lib/` on linux systems.

The libraries used for RSA operations are in the directory `wolfssl-X.X.X/IPP/` where X.X.X is the current wolfSSL version number. Building from the bundled libraries is dependent on the directory location and name of IPP so the file structure of the subdirectory IPP should not be changed.

When allocating memory the fast-rsa operations have a memory tag of `DYNAMIC_TYPE_USER_CRYPTO`. This allows for viewing the memory consumption of RSA operations during run time with the fast-rsa option.

2.6.119 --enable-staticmemory

Enable static memory use

2.6.120 --enable-mcapi

Enable Microchip API

2.6.121 --enable-asynccrypt

Enable Asynchronous Crypto

2.6.122 --enable-sessionexport

Enable export and import of sessions

2.6.123 --enable-aeskeywrap

Enable AES key wrap support

2.6.124 --enable-jobserver

Values: `yes` (default) / no / #

When using `make` this builds wolfSSL using a multithreaded build, `yes` (default) detects the number of CPU cores and builds using a recommended amount of jobs for that count, # to specify an exact number. This works in a similar way to the `make -j` option.

2.6.125 --enable-shared[=PKGS]

Building shared wolfSSL libraries [default=yes]

Disabling the shared library build will exclude a wolfSSL shared library from being built. By default only a shared library is built in order to save time and space.

2.6.126 --enable-static[=PKGS]

Building static wolfSSL libraries [default=no]

2.6.127 --with-liboqs=PATH

Path to OpenQuantumSafe install (default `/usr/local`).

This turns on the ability for wolfSSL to use the experimental TLS 1.3 quantum-safe KEM groups, hybrid quantum-safe KEM groups and FALCON signature scheme via wolfSSL integration with liboqs. Please see the appendix "Experimenting with Quantum-Safe Cryptography" in this document for more details.

2.6.128 --with-libz=PATH

Optionally include libz for compression.

Enabling libz will allow compression support in wolfSSL from the libz library. Think twice about including this option and using it by calling `wolfSSL_set_compression()`. While compressing data before sending decreases the actual size of the messages being sent and received, the amount of data saved by compression usually takes longer in time to analyze than it does to send it raw on all but the slowest of networks.

2.6.129 --with-cavium

Path to cavium/software directory.

2.6.130 --with-user-crypto

Path to USER_CRYPTO install (default `/usr/local`).

2.6.131 --enable-rsavfy

Enables RSA verify only support (**note** requires `--enable-cryptonly`)

2.6.132 --enable-rsapub

Default value: Enabled RSA public key only support (**note** requires `--enable-cryptonly`)

2.6.133 --enable-armasm

Enables ARMv8 ASM support.

The default configure sets mcpu or mfpu based on 64 vs 32 bit system. It does not overwrite mcpu or mfpu setting passed in by use of CPPFLAGS. On some compilers `-mstrict-align` may be needed due to the constraiants and `-mstrict-align` is now also set by default unless a user passes in mcpu/mfpu flags with CPPFLAGS.

2.6.134 --disable-tlsv12

Disable TLS 1.2 support

2.6.135 --enable-tls13

Enable TLS 1.3 support

This build option can be combined with `--disable-tlsv12` and `--disable-oldtls` to produce a wolfSSL build that is only TLS 1.3.

2.6.136 --enable-all

Enables all wolfSSL features, excluding SSL v3

2.6.137 --enable-xts

Enables AES-XTS mode

2.6.138 --enable-asio

Enables ASIO.

Requires that the options `--enable-opensslextra` and `--enable-opensslall` be enabled when configuring wolfSSL. If these two options are not enabled, then the autoconf tool will automatically enable these options to enable ASIO when configuring wolfSSL.

2.6.139 --enable-qt

Enables Qt 5.12 onwards support.

Enables wolfSSL build settings compatible with the wolfSSL Qt port. Patch file is required from wolfSSL for patching Qt source files.

2.6.140 --enable-qt-test

Enable Qt test compatibility build.

Enables support for building wolfSSL for compatibility with running the built-in Qt tests.

2.6.141 --enable-apache-httpd

Enables Apache httpd support

2.6.142 --enable-afalg

Enables use of Linux module AF_ALG for hardware accleration. Additional Xilinx use with `=xilinx`, `=xilinx-rsa`, `=xilinx-aes`, `=xilinx-sha3`

Is similar to `--enable-devcrypto` in that it leverages a Linux kernel module (AF_ALG) for offloading crypto operations. On some hardware the module has performance accelerations available through the Linux crypto drivers. In the case of Petalinux with Xilinx the flag `--enable-afalg=xilinx` can be used to tell wolfSSL to use the Xilinx interface for AF_ALG.

2.6.143 --enable-devcrypto

Enables use of Linux `/dev/crypto` for hardware acceleration.

Has the ability to receive arguments, being able to receive any combination of `aes` (all aes support), `hash` (all hash algorithms), and `cbc` (aes-cbc only). If no options are given, it will default to using `all`.

2.6.144 --enable-mcast

Enable wolfSSL DTLS multicast support

2.6.145 --disable-pkcs12

Disable PKCS12 code

2.6.146 --enable-fallback-scsv

Enables Signaling Cipher Suite Value(SCSV)

2.6.147 --enable-psk-one-id

Enables support for single PSK ID with TLS 1.3

2.6.148 --enable-cryptocb

Enable crypto callbacks. Register a crypto callback using wc_CryptoCb_RegisterDevice and set the associated devId using wolfSSL_CTX_SetDevId.

The following two defines can be used with `--enable-cryptocb` to complie out RSA or ECC software fallback to optimize for footprint reduction when software RSA/ECC is not required.

- WOLF_CRYPTO_CB_ONLY_RSA - compiles out RSA software crypto fallback
- WOLF_CRYPTO_CB_ONLY_ECC - compiles out ECC software crypto fallback

Use of the WOLF_CRYPTO_CB_ONLY_* options requires disabling the examples. See `--disable-examples`

2.6.149 --enable-reproducible-build

Suppresses the binary jitter (timestamps and other non-functional metadata) to allow generation of bitwise-identical binary packages with identical hashes.

2.6.150 --enable-sys-ca-certs

Allows wolfSSL to use trusted system CA certificates for verification when `wolfSSL_CTX_load_system_CA_certs()` is called, either by loading them into wolfSSL certificate manager, or by invoking system authentication APIs. See `wolfSSL_CTX_load_system_CA_certs()` for more details.

2.7 Special Math Optimization Flags

2.7.1 --enable-fastmath

Enable FastMath implementation. Both FastMath and Big Integer library are disabled if Single-Precision (SP) math is enabled.

See USE_FAST_MATH and Big Integer Math Library sections.

2.7.2 --enable-fasthugemath

Enable fast math + huge code.

Enabling fasthugemath includes support for the FastMath library and greatly increases the code size by unrolling loops for popular key sizes during public key operations. Try using the benchmark utility before and after using fasthugemath to see if the slight speedup is worth the increased code size.

2.7.3 --enable-sp-math

Enable Single-Precision (SP) math implementation with restricted algorithm suite. Unsupported algorithms are disabled. Overrides `--enable-sp`, `--enable-sp-math-all`, `--enable-fastmath` and `--enable-fasthugemath`.

- Replaces the math implementation with that in sp_int.c
- A minimal implementation, turns on portions of sp_int.c but not all.
- MUST combine with –enable-sp to turn on the solutions in sp_x86_64.c or sp_arm.c etc (list of files below depending on the target system) to be able to perform RSA/ECC/DH operations.
- Not to be combined with –enable-sp-math-all (below)

FILE LIST (platform dependent, chosen by configure based on system specs or can be manually controlled when using a Makefile/IDE solution): sp_arm32.c sp_arm64.c sp_armthumb.c sp_cortexm.c sp_dsp32.c sp_x86_64.c sp_x86_64_asm.S sp_x86_64_asm.asm

2.7.4 --enable-sp-math-all

Enabled by default. Enable Single-Precision (SP) math implementation with full algorithm suite. Unsupported algorithms are enabled, but unoptimized. Overrides `--enable-sp`, `--enable-fastmath` and `--enable-fasthugemath`.

- Replaces the math implementation with that in sp_int.c
- A FULL implementation, does not depend on –enable-sp to work
- Can be combined with –enable-sp to allow use of the implementations written in portable c assembly (non-hardware specific assembly) in sp_c32.c for 32-bit or sp_c64.c for 64-bit when possible. The rest of the time (when not possible) the implementations in sp_int.c are used. The portable C assembly gives significant performance gains on targets that do not have hardware optimizations available.
- Not to be combined with –enable-sp-math (above)

NOTE: If you are using asymmetric cryptography with key length in bits [256, 384, 521, 1024, 2048, 3072, 4096], you should consider using –enable-sp-math option to get maximum performance with a larger footprint size.

2.7.5 --enable-sp-asm

Enable Single-Precision (SP) assembly implementation.

Can be used to enable Single-Precision performance improvements through assembly with Intel x86_64 and ARM architectures.

2.7.6 --enable-sp=OPT

Enable Single-Precision (SP) math for RSA, DH, and ECC to improve performance.

There are many possible values for OPT. Below is a list of ways to call enable-sp and the resulting macros that will be defined as a result. All of these can be combined in a coma separated list. For example, `--enable-sp=ec256,ec384`. The meaning of the macros that will be defined are defined above in the [wolfSSL's Proprietary Single Precision (SP) Math Support] section.

NOTE: 1) "–enable-sp=small –enable-sp-math" can be smaller than... 2) "–enable-sp-math-all=small"... as (1) only has implementations of specific key sizes while (2) has implementations to support all key sizes.

NOTE: This is for x86_64 and with no other configuration flags; your results may vary depending on your architectures and other configuration flags that you specify. For example, WOLFSSL_SP_384 and WOLFSSL_SP_4096 will only be enabled for Intel x86_64.

2.7.6.1 --enable-sp=no or --disable-sp

No new macros defined. Equivalent of not using `--enable-sp`.

2.7.6.2 --enable-sp or --enable-sp=yes

- WOLFSSL_HAVE_SP_RSA
- WOLFSSL_HAVE_SP_ECC
- WOLFSSL_HAVE_SP_DH
- WOLFSSL_SP_384
- WOLFSSL_SP_4096
- WOLFSSL_SP_LARGE_CODE

2.7.6.3 --enable-sp=small

- WOLFSSL_HAVE_SP_RSA
- WOLFSSL_HAVE_SP_DH
- WOLFSSL_HAVE_SP_ECC
- WOLFSSL_SP_4096
- WOLFSSL_SP_384
- WOLFSSL_SP_4096
- WOLFSSL_SP_SMALL
- WOLFSSL_SP_LARGE_CODE
- WOLFSSL_SP_384

2.7.6.4 --enable-sp=smallfast

- WOLFSSL_HAVE_SP_RSA
- WOLFSSL_HAVE_SP_DH
- WOLFSSL_HAVE_SP_ECC
- WOLFSSL_SP_4096
- WOLFSSL_SP_384
- WOLFSSL_SP_SMALL
- WOLFSSL_SP_4096
- WOLFSSL_SP_LARGE_CODE
- WOLFSSL_SP_FAST_MODEXP

2.7.6.5 --enable-sp=ec256 or --enable-sp=p256 or --enable-sp=p256

- WOLFSSL_HAVE_SP_ECC

2.7.6.6 --enable-sp=smallec256 or --enable-sp=smallp256 or --enable-sp=small256

- WOLFSSL_HAVE_SP_ECC
- WOLFSSL_SP_SMALL

2.7.6.7 --enable-sp=ec384 or --enable-sp=p384 or --enable-sp=384

- WOLFSSL_HAVE_SP_ECC
- WOLFSSL_SP_384
- WOLFSSL_SP_NO_256

2.7.6.8 --enable-sp=smallec384 or --enable-sp=smallp384 or --enable-sp=small384

- WOLFSSL_HAVE_SP_ECC
- WOLFSSL_SP_384
- WOLFSSL_SP_NO_256
- WOLFSSL_SP_SMALL

2.7.6.9 --enable-sp=ec1024 or --enable-sp=p1024 or --enable-sp=1024

- WOLFSSL_HAVE_SP_ECC
- WOLFSSL_SP_1024
- WOLFSSL_SP_NO_256

2.7.6.10 `--enable-sp=smallec1024` or `--enable-sp=smallp1024` or `--enable-sp=small1024`

- WOLFSSL_HAVE_SP_ECC
- WOLFSSL_SP_1024
- WOLFSSL_SP_NO_256
- WOLFSSL_SP_SMALL

2.7.6.11 `--enable-sp=2048`

- WOLFSSL_HAVE_SP_DH
- WOLFSSL_HAVE_SP_RSA
- WOLFSSL_SP_LARGE_CODE
- WOLFSSL_SP_NO_3072

2.7.6.12 `--enable-sp=small2048`

- WOLFSSL_HAVE_SP_DH
- WOLFSSL_HAVE_SP_RSA
- WOLFSSL_SP_LARGE_CODE
- WOLFSSL_SP_NO_3072
- WOLFSSL_SP_SMALL

2.7.6.13 `--enable-sp=rsa2048`

- WOLFSSL_HAVE_SP_RSA
- WOLFSSL_SP_LARGE_CODE
- WOLFSSL_SP_NO_3072

2.7.6.14 `--enable-sp=smallrsa2048`

- WOLFSSL_HAVE_SP_RSA
- WOLFSSL_SP_LARGE_CODE
- WOLFSSL_SP_NO_3072
- WOLFSSL_SP_SMALL

2.7.6.15 `--enable-sp=3072`

- WOLFSSL_HAVE_SP_DH
- WOLFSSL_HAVE_SP_RSA
- WOLFSSL_SP_LARGE_CODE
- WOLFSSL_SP_NO_2048

2.7.6.16 `--enable-sp=small3072`

- WOLFSSL_HAVE_SP_DH
- WOLFSSL_HAVE_SP_RSA
- WOLFSSL_SP_LARGE_CODE
- WOLFSSL_SP_NO_2048
- WOLFSSL_SP_SMALL

2.7.6.17 `--enable-sp=rsa3072`

- WOLFSSL_HAVE_SP_RSA
- WOLFSSL_SP_LARGE_CODE
- WOLFSSL_SP_NO_2048

2.7.6.18 `--enable-sp=smallrsa3072`

- WOLFSSL_SP_LARGE_CODE
- WOLFSSL_SP_NO_2048
- WOLFSSL_SP_SMALL

2.7.6.19 `--enable-sp=4096` or `--enable-sp=+4096`

- WOLFSSL_HAVE_SP_DH
- WOLFSSL_HAVE_SP_RSA
- WOLFSSL_SP_4096
- WOLFSSL_SP_LARGE_CODE
- WOLFSSL_SP_NO_2048
- WOLFSSL_SP_NO_3072

2.7.6.20 `--enable-sp=small4096`

- WOLFSSL_HAVE_SP_DH
- WOLFSSL_HAVE_SP_RSA
- WOLFSSL_SP_4096
- WOLFSSL_SP_LARGE_CODE
- WOLFSSL_SP_NO_2048
- WOLFSSL_SP_NO_3072
- WOLFSSL_SP_SMALL

2.7.6.21 `--enable-sp=rsa4096`

- WOLFSSL_HAVE_SP_RSA
- WOLFSSL_SP_4096
- WOLFSSL_SP_LARGE_CODE
- WOLFSSL_SP_NO_2048
- WOLFSSL_SP_NO_3072

2.7.6.22 `--enable-sp=smallrsa4096`

- WOLFSSL_HAVE_SP_RSA
- WOLFSSL_SP_4096
- WOLFSSL_SP_LARGE_CODE
- WOLFSSL_SP_NO_2048
- WOLFSSL_SP_NO_3072
- WOLFSSL_SP_SMALL

2.7.6.23 `--enable-sp=nomalloc`

- WOLFSSL_SP_NO_MALLOC

2.7.6.24 `--enable-sp=nonblock`

- WOLFSSL_SP_NO_MALLOC
- WOLFSSL_SP_NONBLOCK
- WOLFSSL_SP_SMALL

2.7.6.25 `asm` Combine with other algorithm options to indicate that assembly code is turned on for those options. For example, `--enable-sp=rsa2048,asm`.

2.8 Cross Compiling

Many users on embedded platforms cross compile wolfSSL for their environment. The easiest way to cross compile the library is to use the `./configure` system. It will generate a Makefile which can then be used to build wolfSSL.

When cross compiling, you'll need to specify the host to `./configure`, such as:

```
./configure --host=arm-linux
```

You may also need to specify the compiler, linker, etc. that you want to use:

```
./configure --host=arm-linux CC=arm-linux-gcc AR=arm-linux-ar RANLIB=arm-linux
```

There is a bug in the configure system which you might see when cross compiling and detecting user overriding malloc. If you get an undefined reference to `rpl_malloc` and/or `rpl_realloc`, please add the following to your `./configure` line:

```
ac_cv_func_malloc_0_nonnull=yes ac_cv_func_realloc_0_nonnull=yes
```

After correctly configuring wolfSSL for cross-compilation, you should be able to follow standard autoconf practices for building and installing the library:

```
make
sudo make install
```

If you have any additional tips or feedback about cross compiling wolfSSL, please let us know at info@wolfssl.com.

2.8.1 Example cross compile configure options for toolchain builds

2.8.1.1 armebv7-eabihf-glibc

```
./configure --host=armeb-linux \
        CC=armeb-linux-gcc LD=armeb-linux-ld \
        AR=armeb-linux-ar \
        RANLIB=armeb-linux-ranlib \
        CFLAGS="-DWOLFSSL_USER_IO -Os" \
        CPPFLAGS="-I./"
```

2.8.1.2 armv5-eabi-glibc

```
./configure --host=arm-linux \
        CC=arm-linux-gcc LD=arm-linux-ld \
        AR=arm-linux-ar \
        RANLIB=arm-linux-ranlib \
        CFLAGS="-DWOLFSSL_USER_IO -Os" \
        CPPFLAGS="-I./"
```

2.8.1.3 armv6-eabihf-glibc

```
./configure --host=arm-linux \
        CC=arm-linux-gcc LD=arm-linux-ld \
        AR=arm-linux-ar \
        RANLIB=arm-linux-ranlib \
        CFLAGS="-DWOLFSSL_USER_IO -Os" \
        CPPFLAGS="-I./"
```

2.8.1.4 armv7-eabihf-glibc

```
./configure --host=arm-linux \
        CC=arm-linux-gcc LD=arm-linux-ld \
        AR=arm-linux-ar \
        RANLIB=arm-linux-ranlib \
        CFLAGS="-DWOLFSSL_USER_IO -Os" \
        CPPFLAGS="-I./"
```

2.8.1.5 armv7m-uclibc

```
./configure --enable-static --disable-shared \--host=arm-linux
 ↪  CC=arm-linux-gcc \
        LD=arm-linux-ld AR=arm-linux-ar \
        RANLIB=arm-linux-ranlib \
        CFLAGS="-DWOLFSSL_USER_IO -Os" \
        CPPFLAGS="-I./"
```

2.8.1.6 arm-none-eabi-gcc

```
./configure --host=arm-none-eabi \
        CC=arm-none-eabi-gcc LD=arm-none-eabi-ld \
        AR=arm-none-eabi-ar RANLIB=arm-none-eabi-ranlib \
        CFLAGS="-DNO_WOLFSSL_DIR \
        -DWOLFSSL_USER_IO -DNO_WRITEV \
        -mcpu=cortex-m4 -mthumb -Os \
        -specs=rdimon.specs" CPPFLAGS="-I./"
```

2.8.1.7 mips32–glibc

```
./configure --host=mips-linux \
        CC=mips-linux-gcc LD=mips-linux-ld \
        AR=mips-linux-ar \
        RANLIB=mips-linux-ranlib \
        CFLAGS="-DWOLFSSL_USER_IO -Os" \
        CPPFLAGS="-I./"
```

2.8.1.8 PowerPc64le-Power8-Glibc

```
./configure --host=powerpc64le-buildroot-linux-gnu \
        CC=powerpc64le-buildroot-linux-gnu-gcc \
        LD=powerpc64le-buildroot-linux-gnu-ld \
        AR=powerpc64le-buildroot-linux-gnu-ar \
        RANLIB=powerpc64le-buildroot-linux-gnu-ranlib \
        CFLAGS="-DWOLFSSL_USER_IO -Os" \
        CPPFLAGS="-I./"
```

2.8.1.9 x86-64-core-i7-glibc

```
./configure --host=x86_64-linux \
        CC=x86_64-linux-gcc LD=x86_64-linux-ld \
        AR=x86_64-linux-ar \
        RANLIB=x86_64-linux-ranlib \
        CFLAGS="-DWOLFSSL_USER_IO -Os" \
        CPPFLAGS="-I./"
```

2.8.1.10 x86-64-core-i7-musl

```
./configure --host=x86_64-linux \
        CC=x86_64-linux-gcc LD=x86_64-linux-ld \
        AR=x86_64-linux-ar \
        RANLIB=x86_64-linux-ranlib \
        CFLAGS="-DWOLFSSL_USER_IO -Os" \CPPFLAGS="-I./"
```

2.8.1.11 x86-64-core-i7-uclibc

```
./configure --host=x86_64-linux \
        CC=x86_64-linux-gcc LD=x86_64-linux-ld \
        AR=x86_64-linux-ar \
        RANLIB=x86_64-linux-ranlib \
        CFLAGS="-DWOLFSSL_USER_IO -Os" \
        CPPFLAGS="-I./"
```

2.9 Building Ports

wolfSSL has been ported to many environments and devices. A portion of these ports and accompanying documentation for them is located in the directory `wolfssl-X.X.X/IDE`, where X.X.X is the current wolfSSL version number. This directory also contains helpful information and code for IDE's used to build wolfSSL for the environments.

PORT Lists:

- Arduino
- LPCXPRESSO
- Wiced Studio
- CSBench
- SGX Windows and Linux
  - These directories (`wolfssl/IDE/WIN-SGX` and `wolfssl/IDE/LINUX-SGX`) contain Makefiles for and Visual Studio solutions for building wolfSSL as a library to be used in an Intel SGX project.
- Hexagon
  - This directory (`wolfssl/IDE/HEXAGON`) contains a Makefile for building with the Hexagon tool chain. It can be used to build wolfSSL for offloading ECC verify operations to a DSP processor. The directory contains a README file to help with the steps required to build.
- Hexiwear
- NetBurner M68K
  - In the directory (`wolfssl/IDE/M68K`) there is a Makefile for building wolfSSL for a MCF5441X device using the Netburner RTOS.
- Renesas
  - This directory (`wolfssl/IDE/Renesas`) contains multiple builds for different Renesas devices. It also has example builds that demonstrate using hardware acceleration.
- XCode
- Eclipse
- Espressif
- IAR-EWARM
- Kinetis Design Studio (KDS)
- Rowley Crossworks ARM
- OpenSTM32
- RISCV
- Zephyr
- Mynewt

- INTIME-RTOS

2.10 Building For NXP CAAM

2.10.1 i.MX8 (Linux)

2.10.1.1 Known Issues

- If exiting an open HSM key store session before closing up the HSM session (wc_SECO_CloseHSM and wolfSSL_CLeanup or wolfCrypt_Cleanup) the next time the NVM is started up it segfaults. A power cycle is needed to work around it.

2.10.1.2 Limitations Found

- AES operations with large inputs (i.e. 1 megabyte) fails with the SECO getting “Not enough space in shared memory”.
- After creating 2 keystores the attempt to create a 3rd will fail. To reset the keystores do rm -rf /etc/seco_hsm and power cycle the device.

2.10.1.3 Intro On i.MX8 devices there is a SECO hardware module available for heightened security. This module handles AES operations and key storage, limited ECC operations and key storage, and provides a RNG. wolfSSL has been expanded to make use of the SECO where possible. For some algorithms the CAAM on i.MX8 supports them but the SECO module does not yet. In these cases wolfSSL will make calls through /dev/crypto down to a Linux CAAM driver that creates jobs for the CAAM directly. There are some algorithms supported by default with the NXP Linux CAAM driver but not all algorithms that the CAAM supports. wolfSSL has expanded the Linux CAAM driver to add support for additional algorithms. To use both avenues of accessing the CAAM from the same application the “devId” associated with the different code paths can be set, either WOLFSSL_CAAM_DEVID or WOLFSSL_SECO_DEVID. These IDs are used when first initializing a structure to set which code path will be used throughout the lifetime of the structure. If using software only then the default INVALID_DEVID should be set. The exception to this is SECO items that do not use the key store; TRNG, and Hashing.

Versions of software used:

- imx-seco-libs branch imx_5.4.24_2.1.0
- NXP “repo” tool and Yocto build. Documentation on Yocto setup is here
- wolfSSL 5.2.0 + (was developed after 5.2.0 release)

2.10.1.4 Supported Algorithms Supported algorithms, modes, and operations include:

- AES (ECB, CBC, CCM, GCM)
- AES CMAC
- SHA256, SHA384
- ECC 256/384 (keygen, sign, verify, ecdh)
- RSA 3072 (keygen, sign, verify)
- HMAC
- Curve25519
- TRNG

2.10.1.5 Building Image

2.10.1.5.1 “repo” Setup Setting up the NXP “repo” command tool was done on Ubuntu 18.04 LTS

```
sudo apt-get install gawk wget git-core diffstat unzip texinfo gcc-multilib
sudo apt-get install build-essential chrpath socat cpio python python3 python3
    -pip
sudo apt-get install python3-pexpect xz-utils debianutils iputils-ping python3
    -git
sudo apt-get install python3-jinja2 libegl1-mesa libsdl1.2-dev pylint3 xterm
    curl
sudo apt-get install ca-certificates

mkdir ~/bin
curl https://storage.googleapis.com/git-repo-downloads/repo > ~/bin/repo
chmod a+x ~/bin/repo
export PATH=~/bin:$PATH

git config --global user.name "Your Git Name"
git config --global user.email "Your Email"
```

As of Jan 11, 2022 GitHub is no longer allowing unauthenticated git connections. NXP's repo tool has not yet taken this into account at the time that this document was created (March, 2022). To work around this redirect git://github.com/ to https://github.com/ with the following command:

```
git config --global url."https://github.com/".insteadOf git://github.com/
```

Make the desired directory to build in, for example:

```
mkdir imx-yocto-bsp
cd imx-yocto-bsp/
```

After setting up NXP's "repo" command tool, initialize and sync a directory with the desired version of Linux. In this case 5.4.24_2.1.0.

```
repo init -u https://source.codeaurora.org/external/imx/imx-manifest -b imx-
    linux-zeus -m imx-5.4.24-2.1.0.xml
repo sync

DISTRO=fsl-imx-wayland MACHINE=imx8qxpc0mek source imx-setup-release.sh -b
    build-xwayland
```

2.10.1.5.2 Additional Yocto CAAM Layers Next download the CAAM driver expansion layers that will apply patches to cryptodev-module, cryptodev-linux, and to files in `linux-imx/drivers/crypto/caam/*`. The base blob and ECDSA (sign/verify/keygen) is located here (https://source.codeaurora.org/external/imxsupport/imx The layer that expands that farther to have RSA black key, ECDH, and Curve25519 support is meta-imx-expand-caam. Place both of these directories into the sources directory next to the other existing meta-* directories.

```
#<assuming in the build-xwayland directory from previous command>

#<either clone the current work or open from a delivered zip>
git clone -b caam_expansion https://github.com/JacobBarthelmeh/imx_sec_apps
cp -r imx_sec_apps/meta-imx-ecdsa-sec ../sources/
cp -r imx_sec_apps/meta-imx-expand-caam ../sources/


# or
```

```
git clone https://source.codeaurora.org/external/imxsupport/imx_sec_apps.git
#(meta-imx-expand-caam comes from wolfSSL)
unzip meta-imx-expand-caam.zip

cp -r imx_sec_apps/meta-imx-ecdsa-sec ../sources/
mv meta-imx-expand-caam ../sources/
```

Add those layers to the build, ecdsa first, then the CAAM expansion.

```
vim conf/bblayers.conf
BBLAYERS += "${BSPDIR}/sources/meta-imx-ecdsa-sec"
BBLAYERS += "${BSPDIR}/sources/meta-imx-expand-caam"
```

Add the desired modules and libraries to the local conf file.

```
vim conf/local.conf
EXTRA_IMAGE_FEATURES_append = " dev-pkgs tools-sdk tools-debug
 ↪  ssh-server-openssh "
IMAGE_INSTALL_append = " cryptodev-module cryptodev-linux eckey "
```

In this build, we had added debugging tools and an SSH server, which are not necessary if looking to trim down on size. The important items to add are “cryptodev-module and cryptodev-linux”. “eckey” is a demo tool from NXP for encapsulating and decapsulating blobs.

[Optional] To add the auto loading of cryptodev module add the following line to conf/local.conf.

```
KERNEL_MODULE_AUTOLOAD += "cryptodev"
```

Otherwise the module will need to be loaded after each power cycle using “modprobe cryptodev”.

2.10.1.5.3 Build and Deploy To kick off the build of the image use the following command. Then if using an sdcard, flash it to the card.

```
bitbake core-image-base

cd tmp/deploy/images/imx8qxpc0mek/
bzcat core-image-base-imx8qxpc0mek.sdcard.bz2 | sudo dd of=/dev/diskX bs=5m
```

Note: 5m is for Mac OS, use 5M for Linux. diskX should be replaced with the location of sdcard, i.e. disk2 on Mac or sdbX on Linux. Check to make sure of the sdcard disk number before executing. For use with building wolfssl/examples later export the install directory using the following:

```
exportCRYPTODEV_DIR=`pwd`/tmp/sysroots-components/aarch64/cryptodev-linux/usr/
   include/
```

To install the toolchain for cross compiling use the following Yocto command

```
bitbake meta-toolchain
sudo ./tmp/deploy/sdk/<version>.sh
```

2.10.1.6 Building NXP HSM

2.10.1.6.1 Build zlib Multiple ways to do this, one is to add it to the Yocto build one way is to build it using bitbake as follows.

```
cd build-xwayland
bitbake zlib
```

This places the results into the directory tmp/sysroots-components/aarch64/zlib/usr/

For use with building wolfssl/examples later export the install directory using the following:

```
export ZLIB_DIR=`pwd`/tmp/sysroots-components/aarch64/zlib/usr/
```

2.10.1.6.2 Build NXP HSM lib Download the NXP HSM library, and adjust the Makefile (or environment variables) in order to find the necessary zlib.

```
git clone https://github.com/NXP/imx-seco-libs.git
cd imx-seco-libs
git checkout imx_5.4.24_2.1.0
```

```
vim Makefile
```

```
CFLAGS = -O1 -Werror -fPIC -I$(ZLIB_DIR)/include -L$(ZLIB_DIR)/lib
```

then

```
make
make install
```

For use with building wolfssl/examples later export the install directory using the following:

```
export HSM_DIR=`pwd`/export/usr/
```

Make install places the results in the subdirectory “export” by default.

2.10.1.7 Building wolfSSL

2.10.1.7.1 Building Using Autoconf If setting up the Yocto image with development tools then wolfSSL can be built directly on the system. For a more minimalistic approach cross compiling can be used. Debug messages can be enabled with –enable-debug. Extra debug messages specific to the SECO work can be enabled by defining the macro DEBUG_SECO and for the /dev/crypto calls DEBUG_DEVCRYPTO. The extra debug messages for both make use of printf, outputting on the stdout pipe. There are a couple key enable options for use with SECO. –enable-caam=seco, –enable-devcrypto=seco, –with-seco=/hsm-lib/export.

Example build with HSM SECO only (no devcrypto support for additional algorithms)

```
source /opt/fsl-imx-wayland/5.4-zeus/environment-setup-aarch64-poky-linux

# Install dependencies for building wolfSSL
sudo apt-get install autoconf automake libtool

./autogen.sh
./configure --host=aarch64-poky-linux --with-libz=$ZLIB_DIR
 ↪  --with-seco=$HSM_DIR \ --enable-caam=seco --enable-cmac --enable-aesgcm
 ↪  --enable-aesccm --enable-keygen \
CPPFLAGS="-DHAVE_AES_ECB"
make
```

Example building with HSM SECO and additional devcrypto support. The include path to crypto/cryptodev.h needs to be set.

```
./configure --host=aarch64-poky-linux --with-libz=$ZLIB_DIR
 ↪  --with-seco=$HSM_DIR \
--enable-caam=seco --enable-cmac --enable-aesgcm --enable-aesccm
 ↪  --enable-keygen \
CPPFLAGS="-DHAVE_AES_ECB -I$CRYPTODEV_DIR" --enable-devcrypto=seco \
--enable-curve25519
make
```

There are fail safes to error out early on wolfCrypt_Init / wolfSSL_Init function calls. One case is if the cryptodev module has not been loaded or does not have the support available for the desired operations. The other case where the init operation would fail is if the NXP HSM was not able to be set up. If the application is failing on initialization, adding –enable-debug to the wolfSSL build and the function call wolfSSL_Debugging_ON() before the initialization of wolfSSL will print out useful debug messages about why it is failing.

Example building with debug options turned on

```
./configure --host=aarch64-poky-linux --with-libz=$ZLIB_DIR
 ↪  --with-seco=$HSM_DIR \
--enable-caam=seco --enable-cmac --enable-aesgcm --enable-aesccm
 ↪  --enable-keygen \
CPPFLAGS="-DHAVE_AES_ECB -I$CRYPTODEV_DIR -DDEBUG_SECO -DDEBUG_DEVCRYPTO" \
--enable-devcrypto=seco --enable-curve25519
```

2.10.1.7.2 Building Using user_settings.h These are the macros that can be enabled for building without autotools:

CAAM

- WOLFSSL_CAAM - Main macro switch to enable CAAM support.
- WOLF_CRYPTO_CB - CAAM support makes use of crypto callbacks
- WOLFSSL_SECO_CAAM - Enable SECO HSM use with CAAM (AES-GCM is required and used as the algo for importing plain keys into the HSM).
- WOLFSSL_HASH_KEEP - When hashing with algos like SHA256 build up the message and send it to be hashed only on a call to Final.
- WOLFSSL_CAAM_ECC - Enable CAAM ECC support.
- WOLFSSL_CAAM_CMAC - Enable CAAM CMAC support.
- WOLFSSL_CAAM_CIPHER - Enable CAAM AES support.
- WOLFSSL_CAAM_HMAC - Enable CAAM HMAC support.
- WOLFSSL_CAAM_HASH - Enable CAAM hashing support such as SHA256.
- WOLFSSL_CAAM_CURVE25519 - Enable CAAM Curve25519 support.

cryptodev-linux

- WOLFSSL_DEVCRYPTO - Main macro switch to enable cryptodev-linux use.
- WOLFSSL_DEVCRYPTO_HMAC - Enable support of HMAC with cryptodev-linux.
- WOLFSSL_DEVCRYPTO_RSA - Enable support of RSA with cryptodev-linux.
- WOLFSSL_DEVCRYPTO_CURVE25519 - Enable support of Curve25519 with cryptodev-linux.
- WOLFSSL_DEVCRYPTO_ECDSA - Enable support of ECDSA with cryptodev-linux.
- WOLFSSL_DEVCRYPTO_HASH_KEEP - Enable support of storing up hashes with cryptodev-linux.

Additional files that need compiled for CAAM support are:

- wolfssl/wolfcrypt/src/port/caam/wolfcaam_aes.c
- wolfssl/wolfcrypt/src/port/caam/wolfcaam_cmac.c
- wolfssl/wolfcrypt/src/port/caam/wolfcaam_rsa.c
- wolfssl/wolfcrypt/src/port/caam/wolfcaam_ecdsa.c
- wolfssl/wolfcrypt/src/port/caam/wolfcaam_x25519.c

- wolfssl/wolfcrypt/src/port/caam/wolfcaam_hash.c
- wolfssl/wolfcrypt/src/port/caam/wolfcaam_hmac.c
- wolfssl/wolfcrypt/src/port/caam/wolfcaam_init.c
- wolfssl/wolfcrypt/src/port/caam/wolfcaam_seco.c
- wolfssl/wolfcrypt/src/port/devcrypto/devcrypto_ecdsa.c
- wolfssl/wolfcrypt/src/port/devcrypto/devcrypto_x25519.c
- wolfssl/wolfcrypt/src/port/devcrypto/devcrypto_rsa.c
- wolfssl/wolfcrypt/src/port/devcrypto/devcrypto_hmac.c
- wolfssl/wolfcrypt/src/port/devcrypto/devcrypto_hash.c
- wolfssl/wolfcrypt/src/port/devcrypto/devcrypto_aes.c
- wolfssl/wolfcrypt/src/port/devcrypto/wc_devcrypto.c
- wolfssl/wolfcrypt/src/cryptocb.c

2.10.1.8 Examples

2.10.1.8.1 Running Testwolfcrypt The unit tests that are bundled with wolfSSL are located in wolfcrypt/test/test.c. An example of building and running the tests on the device would be the following. Note that this uses WOLFSSL_CAAM_DEVID so it is making use of the cryptodev module and not the NXP HSM library.

```
./configure --host=aarch64-poky-linux --with-libz=$ZLIB_DIR \
--with-seco=$HSM_DIR --enable-caam=seco --enable-cmac --enable-aesgcm \
--enable-aesccm --enable-keygen CPPFLAGS="-DHAVE_AES_ECB -I$CRYPTODEV_DIR" \
--enable-devcrypto=seco --enable-curve25519 --enable-sha224 --enable-static \
--disable-shared --disable-filesystem
make

scp wolfcrypt/test/testwolfcrypt root@192.168.0.14:/tmp
ssh root@192.168.0.14
root@imx8qxpc0mek:~# /tmp/testwolfcrypt
------------------------------------------------------------------------------
 wolfSSL version 5.2.0
------------------------------------------------------------------------------
error   test passed!
MEMORY   test passed!
base64   test passed!
asn     test passed!
RANDOM   test passed!
MD5     test passed!
SHA     test passed!
SHA-224  test passed!
SHA-256  test passed!
SHA-384  test passed!
SHA-512  test passed!
SHA-3   test passed!
Hash    test passed!
HMAC-MD5 test passed!
HMAC-SHA test passed!
HMAC-SHA224 test passed!
HMAC-SHA256 test passed!
HMAC-SHA384 test passed!
HMAC-SHA512 test passed!
HMAC-SHA3   test passed!
HMAC-KDF    test passed!
```

```
GMAC    test passed!
Chacha   test passed!
POLY1305 test passed!
ChaCha20-Poly1305 AEAD test passed!
AES     test passed!
AES192   test passed!
AES256   test passed!
AES-GCM  test passed!
AES-CCM  test passed!
RSA     test passed!
DH      test passed!
PWDBASED test passed!
ECC     test passed!
ECC buffer test passed!
CURVE25519 test passed!
CMAC    test passed!
COMPRESS test passed!
logging  test passed!
time test passed!
mutex   test passed!
memcb   test passed!
crypto callback test passed!
Test complete
Exiting main with return code: 0
root@imx8qxpc0mek:~#
```

Additional examples are located in the wolfssl-examples repository under the caam/seco directory.

```
git clone https://github.com/wolfssl/wolfssl-examples
cd wolfssl-examples/caam/seco
make
```

2.10.1.8.2 Compiling Source Code [using user_settings.h] To build a single source file linking to wolfSSL the following commands could be used. Assuming that the env. variables from the previous steps are still set.

```
source /opt/fsl-imx-xwayland/5.4-zeus/environment-setup-aarch64-poky-linux

$CC -DWOLFSSL_USER_SETTINGS -I /path/to/user_settings.h \
-I $CRYPTODEV_DIR -I $HSM_DIR/include -I ./wolfssl server-dtls.c \
libwolfssl.a $HSM_DIR/lib/hsm_lib.a $HSM_DIR/lib/seco_nvm_manager.a \
$ZLIB_DIR/lib/libz.a -lpthread -lm
```

2.10.1.9 API

2.10.1.9.1 API Added *List of additional API added*

- void wc_SECO_AesSetKeyID(Aes* aes, int keyId); This function is used to set a SECO key ID into an Aes structure. It should be called after the Aes structure has been initialized and before the structure gets used for encrypt/decrypt operations.
- int wc_SECO_AesGetKeyID(Aes* aes); A getter function for the SECO key ID set in the Aes structure.
- void wc_SECO_CMACSetKeyID(Cmac* cmac, int keyId); Similar to wc_SECO_AesSetKeyID but for Cmac structures.

- int wc_SECO_CMACGetKeyID(Cmac* cmac); Getter function for the SECO key ID set in the Cmac structure.
- int wc_SECO_OpenHSM(word32 keyStoreId, word32 nonce, word16 maxUpdates, byte flag); This function should be called before doing any operations that require the keystore. Such as ECC or AES operations. The first argument is the keystore ID, "nonce" is the specific 32bit sequence used when creating and unlocking an already existing keystore, "maxUpdates" sets the max number of times the key store can be updated, "flag" is used for optional flags and takes in whether or not the key store is being created. To create the HSM keystore the flag should be CAAM_KEYSTORE_CREATE otherwise to open an existing keystore or update it the flag should be CAAM_KEYSTORE_UPDATE.
- int wc_SECO_CloseHSM(void); This function must be called when done with the keystore and before calling wolfCrypt_Cleanup/wolfSSL_Cleanup. It closes down the currently open keystore.
- int wc_SECO_GenerateKey(int flags, int group, byte* out, int outSz, int keyType, int keyInfo, unsigned int* keyIdOut);
  This function can be used for generating new keys in the SECO. For key generation the flag should be CAAM_GENERATE_KEY. For updating a key the flag should be CAAM_UPDATE_KEY. The keyIdOut argument is an in/out argument that gets populated on key creation and should be set as an input on key update. In the case of updating a key it should be a transient type and the group should be set to 0 for updating. Transient type is set as the keyInfo arg, possible options for keyInfo and KeyTypes are;
  - CAAM_KEY_TRANSIENT (keyInfo)
  - CAAM_KEY_PERSISTENT (keyInfo)
  - CAAM_KEYTYPE_ECDSA_P256 (keyType)
  - CAAM_KEYTYPE_ECDSA_P384 (keyType)
  - CAAM_KEYTYPE_AES128 (keyType)
- int wc_SECO_DeleteKey(unsigned int keyId, int group, int keyTypeIn); Used to delete a key from the keystore.

2.10.1.9.2 Native wolfSSL API With CAAM Support This is a list of native wolfSSL API that now have CAAM support with the SECO build outlined in this documentation.

For generation of any AES encrypt and decrypt operations the key can be generated using the following process. Using wc_SECO_GenerateKey(CAAM_GENERATE_KEY, groupID, pubOut, 0, CAAM_KEYTPE_AES128, CAAM_KEY_PERSISTENT, &keyIdOut); where groupID is a specified group number and pubOut is a 32 byte buffer, and the variable keyIdOut gets set to a the new key ID generated. This new key ID generated can then be set in an Aes structure using wc_SECO_AesSetKeyID(Aes, keyIdOut); . Once the key ID has been set in the structure and the Aes structure has been initialized as a WOLFSSL_SECO_DEVID type it will use that key ID for all encrypt and decrypt operations.

AES (ECB/CBC)

Alternatively to generate AES ECB/CBC keys, if the Aes structure has been initialized with WOLFSSL_SECO_DEVID the function wc_AesSetKey can be called with a plain text key passed to it. The API wc_AesSetKey will then try to encrypt the key using the unique KEK and import it into the SECO HSM. If imported successfully, the value of 0 will be returned and the key ID will be set in the Aes structure.

- CBC encrypt would be done with wc_AesCbcEncrypt, decrypt with wc_AesCbcDecrypt.
- ECB encrypt would be done with wc_AesEcbEncrypt, decrypt with wc_AesEcbDecrypt.

Once finished with the Aes structure it must be free'd using wc_AesFree(Aes); .

AES-GCM

- GCM encrypt would be done with wc_AesGcmEncrypt, decrypt with wc_AesGcmDecrypt.

The AES-GCM encrypt function takes in the Aes structure, output buffer, input buffer, input buffer size, nonce, nonce size (required to be 12 bytes), MAC or known as tag, tag size (required to be 16 bytes),

additional data, additional data size (4 bytes). On encryption the input buffer is encrypted and the tag buffer is filled with a created MAC.

For AES-GCM decrypt the function takes in the Aes structure, plain text output buffer, cipher text input buffer, input buffer size, nonce, nonce size (12 bytes), previously created tag from encryption call, tag buffer size, additional data, additional data size. On decryption the tag buffer is checked to verify the message's integrity.

Once finished with the Aes structure it must be free'd using wc_AesFree(Aes); .

AES-CCM

- CCM encrypt would be done with wc_AesCcmEncrypt, decrypt with wc_AesCcmDecrypt.

The AES-CCM encrypt function takes in the Aes structure, output buffer, input buffer, input buffer size, nonce, nonce size (required to be 12 bytes), MAC or known as tag, tag size (required to be 16 bytes), additional data, additional data size (0 bytes). The additional data buffer should be NULL and a size of 0 is required with the NXP HSM library. On encryption the input buffer is encrypted and the tag buffer is filled with a created MAC.
For AES-CCM decrypt the function takes in the Aes structure, plain text output buffer, cipher text input buffer, input buffer size, nonce, nonce size (12 bytes), previously created tag from encryption call, tag buffer size, additional data, additional data size. Similar to the encrypt function the additional data buffer should be NULL. On decryption the tag buffer is checked to verify the message's integrity.
Once finished with the Aes structure it must be free'd using wc_AesFree(Aes); .

AES CMAC

For AES CMAC operations the AES key can be generated using wc_SECO_GenerateKey(CAAM_GENERATE_KEY, groupID, pubOut, 0, CAAM_KEYTPE_AES128, CAAM_KEY_PERSISTENT, &keyIdOut); where groupID is a specified group number and pubOut is a 32 byte buffer, and the variable keyIdOut gets set to a the new key ID generated. This new key ID generated can then be set in an Aes structure using wc_SECO_CMACSetKeyID(Cmac, keyIdOut); . Once the key ID has been set in the structure and the Aes structure has been initialized as a WOLFSSL_SECO_DEVID type it will use that key ID for all encrypt and decrypt operations.

Since the HSM library is a single shot type, each call to wc_CmacUpdate stores up the input into an internal buffer. Then once wc_CmacFinal is called the whole buffer is passed on to the hardware for creating the MAC.

RSA

RSA operations make use of the cryptodev-linux module. This includes support for AES-ECB encrypted black private keys, which is the default when initialized with WOLFSSL_CAAM_DEVID.

Example of native wolfSSL API that would be used with the cryptodev-linux module is as follows:

```
wc_InitRsaKey_ex(key, heap-hint (can be NULL), WOLFSSL_CAAM_DEVID);
wc_MakeRsaKey(key, 3072, WC_RSA_EXPONENT, &rng);
wc_RsaSSL_Sign or wc_RsaPublicEncrypt
wc_RsaSSL_Verify or wc_RsaPrivateDecrypt
wc_FreeRsaKey(key)
```

ECC

ECC sign and verify operations can use either the cryptodev-linux module or the NXP HSM library. ECDH operations for creating a shared secret can only be done with the cryptodev-linux module.

For use with SECO (using the NXP HSM library) the dev ID flag of WOLFSSL_SECO_DEVID should be used when initializing the ecc_key structure. For use with the cryptodev-linux module the dev ID flag

WOLFSSL_CAAM_DEVID should be used. After initialization with the function wc_ecc_init_ec(key, heap-hint (can be NULL), dev ID); then both use cases follow the same function native wolfSSL function calls for sign and verify.

Example function calls after initialization of the ecc_key structure would be:

```
wc_ecc_make_key(&rng, ECC_P256_KEYSIZE, key);
wc_ecc_sign_hash(hash, hashSz, sigOut, sigOutSz, &rng, key);
wc_ecc_verify_hash(sig, sigSz, hash, hashSz, &result, key);
```

And with the cryptodev-linux module (WOLFSSL_CAAM_DEVID) the ECDH function can be used:

```
wc_ecc_shared_secret(keyA, keyB, sharedSecret, sharedSecretSz);
```

Hash (Sha256, Sha384, HMAC)

SHA256 and SHA384 operations use the NXP HSM library. HMAC operations make use of the cryptodev-linux module.

By default SHA operations try to make use of the NXP HSM library, but explicitly set them to the dev ID WOLFSSL_SECO_DEVID can be used.

```
wc_InitSha256_ex(sha256, heap-hint, WOLFSSL_SECO_DEVID);
wc_InitSha384_ex(sha384, heap-hint, WOLFSSL_SECO_DEVID);
```

Because the NXP HSM library supports a single shot operation for hashing, each call to “update” will store the buffer until a “final” function is called and then pass the whole buffer on to the hardware for creating the hash digest.

Where HMAC makes use of the cryptodev-linux the Hmac structure should be initialized using the dev ID WOLFSSL_CAAM_DEVID.

```
wc_HmacInit(hmac, heap-hint, WOLFSSL_CAAM_DEVID);
```

It then can be used like it typically would be with native wolfSSL API:

```
wc_HmacSetKey(hmac, hash-type, key, keySz);
wc_HmacUpdate(hmac, input, inputSz);
wc_HmacFinal(hmac, digestOut);
```

Curve25519

Curve25519 point multiplication is done using the cryptodev-linux module and should be initialized with the dev ID WOLFSSL_CAAM_DEVID for use with the hardware.

Example API calls would be as follows:

```
wc_curve25519_init_ex(key, heap-hint, WOLFSSL_CAAM_DEVID);
wc_curve25519_make_key(&rng, CURVE25519_KEYSIZE,  key);
wc_curve25519_shared_secret(key, keyB, sharedSecretOut, sharedSecretOutSz);
```

RNG

TRNG for seeding the wolfSSL HASH-DRBG makes use of the NXP HSM library. This is compiled in to the wolfcrypt/src/random.c file when wolfSSL is built with –enable-caam=seco. All RNG initializations in wolfSSL will make use of the TRNG for seeding. Standard RNG API calls would be as follows:

```
wc_InitRng(rng);
wc_RNG_GenerateBlock(rng, output, outputSz);
wc_FreeRng(rng);
```

2.10.2 i.MX8 (QNX)

<Documentation available, not inserted here yet @TODO>

2.10.3 i.MX6 (QNX)

<Documentation available, not inserted here yet @TODO>

2.10.4 IMXRT1170 (FreeRTOS)

Example IDE Setup for use with IMXRT1170 can be found in the directory IDE/MCUEXPRESSO/RT1170

2.10.4.1 Build Steps

- Open MCUEXPRESSO and set the workspace to wolfssl/IDE/MCUEXPRESSO/RT1170
- File -> Open Projects From File System... -> Directory : and set the browse to wolfssl/IDE/MCUEXPRESSO/RT1170 directory then click “select directory”
- Select wolfssl_cm7, wolfcrypt_test_cm7, CSR_example, PKCS7_example
- Right click the projects -> SDK Management -> Refresh SDK Components and click “yes”
- increase the size of configTOTAL_HEAP_SIZE in FreeRTOSConfig.h to be 60240 for CSR and PKCS7 example and around 100000 for wolfcrypt_test_cm7
- (note board files need to be recreated .... this can be done by creating a new project that has the same settings and copying over the generated board/* files)
- Build the projects

2.10.4.2 Expanding RT1170 CAAM Driver The files RT1170/fsl_caam_h.patch and RT1170/fsl_caam_c.patch include changes to the existing NXP CAAM driver for use with creating/opening Blobs and generating and using ECC black keys.

To apply the patches first create a project that has the caam driver. This will generate the base fsl_caam.c and fsl_caam.h in the drivers directory. (i.e PKCS7_example_cm7/drivers/fls_caam.{c,h}) . Once the base files are generated then ‘cd’ to the drivers directory and apply the patch (cd PKCS7_example_cm7/drivers/ && patch -p1 < ../../fsl_caam_c.patch && patch -p1 < ../../fsl_caam_h.patch)

In the patch for fsl_caam.h there are macros defined for both the ECC and Blob expansion (CAAM_ECC_EXPANSION and CAAM_BLOB_EXPANSION). When wolfSSL code finds that these macros are defined (the patch has been applied) then it tries to compile in use of the expanded driver.

3 Getting Started

3.1 General Description

wolfSSL, formerly CyaSSL, is about 10 times smaller than yaSSL, and up to 20 times smaller than OpenSSL when using the compile options described in Chapter 2. User benchmarking and feedback also reports dramatically better performance from wolfSSL vs. OpenSSL in the vast majority of standard SSL operations.

For instructions on the build process please see Chapter 2.

3.2 Testsuite

The testsuite program is designed to test the ability of wolfSSL and its cryptography library, wolfCrypt, to run on the system.

wolfSSL needs all examples and tests to be run from the wolfSSL home directory. This is because it finds certs and keys from ./certs. To run testsuite, execute:

```
./testsuite/testsuite.test
```

Or when using autoconf:

```
make test
```

On *nix or Windows the examples and testsuite will check to see if the current directory is the source directory and if so, attempt to change to the wolfSSL home directory. This should work in most setup cases, if not, just use the first method above and specify the full path.

On a successful run you should see output like this, with additional output for unit tests and cipher suite tests:

```
------------------------------------------------------------------------------
 wolfSSL version 4.8.1
------------------------------------------------------------------------------
error    test passed!
MEMORY   test passed!
base64   test passed!
base16   test passed!
asn      test passed!
RANDOM   test passed!
MD5      test passed!
SHA      test passed!
SHA-224  test passed!
SHA-256  test passed!
SHA-384  test passed!
SHA-512  test passed!
SHA-3    test passed!
Hash     test passed!
HMAC-MD5 test passed!
HMAC-SHA test passed!
HMAC-SHA224 test passed!
HMAC-SHA256 test passed!
HMAC-SHA384 test passed!
HMAC-SHA512 test passed!
HMAC-SHA3   test passed!
HMAC-KDF    test passed!
GMAC     test passed!
```

```
Chacha   test passed!
POLY1305 test passed!
ChaCha20-Poly1305 AEAD test passed!
AES      test passed!
AES192   test passed!
AES256   test passed!
AES-GCM  test passed!
RSA      test passed!
DH       test passed!
PWDBASED test passed!
OPENSSL  test passed!
OPENSSL (EVP MD) passed!
OPENSSL (PKEY0) passed!
OPENSSL (PKEY1) passed!
OPENSSL (EVP Sign/Verify) passed!
ECC      test passed!
logging  test passed!
mutex    test passed!
memcb    test passed!
Test complete
Alternate cert chain used
 issuer : /C=US/ST=Montana/L=Bozeman/O=Sawtooth/OU=Consulting/CN=www.wolfssl.
    com/emailAddress=info@wolfssl.com
 subject: /C=US/ST=Montana/L=Bozeman/O=wolfSSL/OU=Support/CN=www.wolfssl.com/
    emailAddress=info@wolfssl.com
 altname = example.com
Alternate cert chain used
 issuer : /C=US/ST=Montana/L=Bozeman/O=wolfSSL_2048/OU=Programming-2048/CN=www
    .wolfssl.com/emailAddress=info@wolfssl.com
 subject: /C=US/ST=Montana/L=Bozeman/O=wolfSSL_2048/OU=Programming-2048/CN=www
    .wolfssl.com/emailAddress=info@wolfssl.com
 altname = example.com
 serial number:01
SSL version is TLSv1.2
SSL cipher suite is TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
SSL signature algorithm is RSA-SHA256
SSL curve name is SECP256R1
Session timeout set to 500 seconds
Client Random :  serial number:f1:5c:99:43:66:3d:96:04
SSL version is TLSv1.2
SSL cipher suite is TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
SSL signature algorithm is RSA-SHA256
SSL curve name is SECP256R1
1DC16A2C0D3AC49FC221DD5B8346B7B38CB9899B7A402341482183Server Random : 1679
   E50DBBBB3DB88C90F600C4C578F4F5D3CEAEC9B16BCCA215C276B448
765A1385611D6A
Client message: hello wolfssl!
I hear you fa shizzle!
sending server shutdown command: quit!
client sent quit command: shutting down!
ciphers = TLS13-AES128-GCM-SHA256:TLS13-AES256-GCM-SHA384:TLS13-CHACHA20-
   POLY1305-SHA256:DHE-RSA-AES128-SHA:DHE-RSA-AES256-SHA:ECDHE-RSA-AES128-SHA:
   ECDHE-RSA-AES256-SHA:ECDHE-ECDSA-AES128-SHA:ECDHE-ECDSA-AES256-SHA:DHE-RSA-
   AES128-SHA256:DHE-RSA-AES256-SHA256:DHE-RSA-AES128-GCM-SHA256:DHE-RSA-
```

```
    AES256-GCM-SHA384:ECDHE-RSA-AES128-GCM-SHA256:ECDHE-RSA-AES256-GCM-SHA384:
    ECDHE-ECDSA-AES128-GCM-SHA256:ECDHE-ECDSA-AES256-GCM-SHA384:ECDHE-RSA-
    AES128-SHA256:ECDHE-ECDSA-AES128-SHA256:ECDHE-RSA-AES256-SHA384:ECDHE-ECDSA
    -AES256-SHA384:ECDHE-RSA-CHACHA20-POLY1305:ECDHE-ECDSA-CHACHA20-POLY1305:
    DHE-RSA-CHACHA20-POLY1305:ECDHE-RSA-CHACHA20-POLY1305-OLD:ECDHE-ECDSA-
    CHACHA20-POLY1305-OLD:DHE-RSA-CHACHA20-POLY1305-OLD
33bc1a4570f4f1abccd5c48aace529b01a42ab51293954a297796e90d20970f0  input
33bc1a4570f4f1abccd5c48aace529b01a42ab51293954a297796e90d20970f0  /tmp/output-
    gNQWZL

All tests passed!
```

This indicates that everything is configured and built correctly. If any of the tests fail, make sure the build system was set up correctly. Likely culprits include having the wrong endianness or not properly setting the 64-bit type. If you've set anything to the non-default settings try removing those, rebuilding wolfSSL, and then re-testing.

3.3 Client Example

You can use the client example found in examples/client to test wolfSSL against any SSL server. To see a list of available command line runtime options, run the client with the `--help` argument:

```
./examples/client/client --help
```

Which returns:

```
wolfSSL client 4.8.1 NOTE: All files relative to wolfSSL home dir
Max RSA key size in bits for build is set at : 4096
-? <num>    Help, print this usage
            0: English, 1: Japanese
--help      Help, in English
-h <host>   Host to connect to, default 127.0.0.1
-p <num>    Port to connect on, not 0, default 11111
-v <num>    SSL version [0-4], SSLv3(0) - TLS1.3(4)), default 3
-V          Prints valid ssl version numbers, SSLv3(0) - TLS1.3(4)
-l <str>    Cipher suite list (: delimited)
-c <file>   Certificate file,           default ./certs/client-cert.pem
-k <file>   Key file,                   default ./certs/client-key.pem
-A <file>   Certificate Authority file, default ./certs/ca-cert.pem
-Z <num>    Minimum DH key bits,        default 1024
-b <num>    Benchmark <num> connections and print stats
-B <num>    Benchmark throughput using <num> bytes and print stats
-d          Disable peer checks
-D          Override Date Errors example
-e          List Every cipher suite available,
-g          Send server HTTP GET
-u          Use UDP DTLS, add -v 2 for DTLSv1, -v 3 for DTLSv1.2 (default)
-m          Match domain name in cert
-N          Use Non-blocking sockets
-r          Resume session
-w          Wait for bidirectional shutdown
-M <prot>   Use STARTTLS, using <prot> protocol (smtp)
-f          Fewer packet/group messages
-x          Disable client cert/key loading
-X          Driven by eXternal test case
-j          Use verify callback override
```

```
-n          Disable Extended Master Secret
-H <arg>    Internal tests [defCipherList, exitWithRet, verifyFail,
   useSupCurve,
                           loadSSL, disallowETM]
-J          Use HelloRetryRequest to choose group for KE
-K          Key Exchange for PSK not using (EC)DHE
-I          Update keys and IVs before sending data
-y          Key Share with FFDHE named groups only
-Y          Key Share with ECC named groups only
-1 <num>    Display a result by specified language.
            0: English, 1: Japanese
-2          Disable DH Prime check
-6          Simulate WANT_WRITE errors on every other IO send
-7          Set minimum downgrade protocol version [0-4]  SSLv3(0) - TLS1.3(4)
```

To test against example.com:443 try the following. This is using wolfSSL compiled with the `--enable-opensslextra` and `--enable-supportedcurves` build options:

```
./examples/client/client -h example.com -p 443 -d -g
```

Which returns:

```
Alternate cert chain used
 issuer : /C=US/O=DigiCert Inc/CN=DigiCert TLS RSA SHA256 2020 CA1
 subject: /C=US/ST=California/L=Los Angeles/O=Internet Corporation for
    Assigned Names and Numbers/CN=www.example.org
 altname = www.example.net
 altname = www.example.edu
 altname = www.example.com
 altname = example.org
 altname = example.net
 altname = example.edu
 altname = example.com
 altname = www.example.org
 serial number:0f:be:08:b0:85:4d:05:73:8a:b0:cc:e1:c9:af:ee:c9
SSL version is TLSv1.2
SSL cipher suite is TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
SSL curve name is SECP256R1
Session timeout set to 500 seconds
Client Random : 20640
   B8131D8E542646D395B362354F9308057B1624C2442C0B5FCDD064BFE29
SSL connect ok, sending GET...
HTTP/1.0 200 OK
Accept-Ranges: bytes
Content-Type: text/html
Date: Thu, 14 Oct 2021 16:50:28 GMT
Last-Modified: Thu, 14 Oct 2021 16:45:10 GMT
Server: ECS (nyb/1D10)
Content-Length: 94
Connection: close
```

This tells the client to connect to (`-h`) example.com on the HTTPS port (`-p`) of 443 and sends a generic (`-g`) GET request. The (`-d`) option tells the client not to verify the server. The rest is the initial output from the server that fits into the read buffer.

If no command line arguments are given, then the client attempts to connect to the localhost on the

wolfSSL default port of 11111. It also loads the client certificate in case the server wants to perform client authentication.

The client is able to benchmark a connection when using the `-b <num>` argument. When used, the client attempts to connect to the specified server/port the argument number of times and gives the average time in milliseconds that it took to perform `SSL_connect()`. For example:

```
./examples/client/client -b 100 -h example.com -p 443 -d
```

Returns:

```
wolfSSL_connect avg took:  296.417 milliseconds
```

If you'd like to change the default host from localhost, or the default port from 11111, you can change these settings in `/wolfssl/test.h`. The variables wolfSSLIP and wolfSSLPort control these settings. Re-build all of the examples including testsuite when changing these settings otherwise the test programs won't be able to connect to each other.

By default, the wolfSSL example client tries to connect to the specified server using TLS 1.2. The user is able to change the SSL/TLS version which the client uses by passing the `-v` command line option. The following values are available for this option:

- `-v 0` - SSL 3.0 (disabled by default)
- `-v 1` - TLS 1.0
- `-v 2` - TLS 1.1
- `-v 3` - TLS 1.2 (selected by default)
- `-v 4` - TLS 1.3

A common error users see when using the example client is -188:

```
wolfSSL_connect error -188, ASN no signer error to confirm failure
wolfSSL error: wolfSSL_connect failed
```

This is typically caused by the wolfSSL client not being able to verify the certificate of the server it is connecting to. By default, the wolfSSL client loads the yaSSL test CA certificate as a trusted root certificate. This test CA certificate will not be able to verify an external server certificate which was signed by a different CA. As such, to solve this problem, users either need to turn off verification of the peer (server), using the `-d` option:

```
./examples/client/client -h myhost.com -p 443 -d
```

Or load the correct CA certificate into the wolfSSL client using the `-A` command line option:

```
./examples/client/client -h myhost.com -p 443 -A serverCA.pem
```

3.4 Server Example

The server example demonstrates a simple SSL server that optionally performs client authentication. Only one client connection is accepted and then the server quits. The client example in normal mode (no command line arguments) will work just fine against the example server, but if you specify command line arguments for the client example, then a client certificate isn't loaded and the `wolfSSL_connect()` will fail (unless client cert check is disabled using the `-d` option). The server will report an error "-245, peer didn't send cert". Like the example client, the server can be used with several command line arguments as well:

```
./examples/server/server --help
```

Which returns:

```
server 4.8.1 NOTE: All files relative to wolfSSL home dir
-? <num>    Help, print this usage
```

```
            0: English, 1: Japanese
--help      Help, in English
-p <num>    Port to listen on, not 0, default 11111
-v <num>    SSL version [0-4], SSLv3(0) - TLS1.3(4)), default 3
-l <str>    Cipher suite list (: delimited)
-c <file>   Certificate file,           default ./certs/server-cert.pem
-k <file>   Key file,                   default ./certs/server-key.pem
-A <file>   Certificate Authority file, default ./certs/client-cert.pem
-R <file>   Create Ready file for external monitor default none
-D <file>   Diffie-Hellman Params file, default ./certs/dh2048.pem
-Z <num>    Minimum DH key bits,        default 1024
-d          Disable client cert check
-b          Bind to any interface instead of localhost only
-s          Use pre Shared keys
-u          Use UDP DTLS, add -v 2 for DTLSv1, -v 3 for DTLSv1.2 (default)
-f          Fewer packet/group messages
-r          Allow one client Resumption
-N          Use Non-blocking sockets
-S <str>    Use Host Name Indication
-w          Wait for bidirectional shutdown
-x          Print server errors but do not close connection
-i          Loop indefinitely (allow repeated connections)
-e          Echo data mode (return raw bytes received)
-B <num>    Benchmark throughput using <num> bytes and print stats
-g          Return basic HTML web page
-C <num>    The number of connections to accept, default: 1
-H <arg>    Internal tests [defCipherList, exitWithRet, verifyFail,
   useSupCurve,
                           loadSSL, disallowETM]
-U          Update keys and IVs before sending
-K          Key Exchange for PSK not using (EC)DHE
-y          Pre-generate Key Share using FFDHE_2048 only
-Y          Pre-generate Key Share using P-256 only
-F          Send alert if no mutual authentication
-2          Disable DH Prime check
-1 <num>    Display a result by specified language.
            0: English, 1: Japanese
-6          Simulate WANT_WRITE errors on every other IO send
-7          Set minimum downgrade protocol version [0-4]  SSLv3(0) - TLS1.3(4)
```

3.5 EchoServer Example

The echoserver example sits in an endless loop waiting for an unlimited number of client connections. Whatever the client sends the echoserver echoes back. Client authentication isn't performed so the example client can be used against the echoserver in all 3 modes. Four special commands aren't echoed back and instruct the echoserver to take a different action.

1. `quit` - If the echoserver receives the string "quit" it will shutdown.
2. `break` If the echoserver receives the string "break" it will stop the current session but continue handling requests. This is particularly useful for DTLS testing.
3. `printstats` - If the echoserver receives the string "printstats" it will print out statistics for the session cache.
4. GET - If the echoserver receives the string "GET" it will handle it as an http get and send back a simple page with the message "greeting from wolfSSL". This allows testing of various TLS/SSL

clients like Safari, IE, Firefox, gnutls, and the like against the echoserver example.

The output of the echoserver is echoed to stdout unless NO_MAIN_DRIVER is defined. You can redirect output through the shell or through the first command line argument. To create a file named output.txt with the output from the echoserver run:

```
./examples/echoserver/echoserver output.txt
```

3.6 EchoClient Example

The echoclient example can be run in interactive mode or batch mode with files. To run in interactive mode and write 3 strings "hello", "wolfssl", and "quit" results in:

```
./examples/echoclient/echoclient
hello
hello
wolfssl
wolfssl
quit
sending server shutdown command: quit!
```

To use an input file, specify the filename on the command line as the first argument. To echo the contents of the file input.txt issue:

```
./examples/echoclient/echoclient input.txt
```

If you want the result to be written out to a file, you can specify the output file name as an additional command line argument. The following command will echo the contents of file input.txt and write the result from the server to output.txt:

```
./examples/echoclient/echoclient input.txt output.txt
```

The testsuite program does just that, but hashes the input and output files to make sure that the client and server were getting/sending the correct and expected results.

3.7 Benchmark

Many users are curious about how the wolfSSL embedded SSL library will perform on a specific hardware device or in a specific environment. Because of the wide variety of different platforms and compilers used today in embedded, enterprise, and cloud-based environments, it is hard to give generic performance calculations across the board.

To help wolfSSL users and customers in determining SSL performance for wolfSSL / wolfCrypt, a benchmark application is provided which is bundled with wolfSSL. wolfSSL uses the wolfCrypt cryptography library for all crypto operations by default. Because the underlying crypto is a very performance-critical aspect of SSL/TLS, our benchmark application runs performance tests on wolfCrypt's algorithms.

The benchmark utility located in wolfcrypt/benchmark (./wolfcrypt/benchmark/benchmark) may be used to benchmark the cryptographic functionality of wolfCrypt. Typical output may look like the following (in this output, several optional algorithms/ciphers were enabled including ECC, SHA-256, SHA-512, AES-GCM, AES-CCM, and Camellia):

```
./wolfcrypt/benchmark/benchmark

------------------------------------------------------------------------------
 wolfSSL version 4.8.1
------------------------------------------------------------------------------
wolfCrypt Benchmark (block bytes 1048576, min 1.0 sec each)
```

```
RNG                105 MB took 1.004 seconds,  104.576 MB/s Cycles per byte =
     20.94
AES-128-CBC-enc    310 MB took 1.008 seconds,  307.434 MB/s Cycles per byte =
      7.12
AES-128-CBC-dec    290 MB took 1.002 seconds,  289.461 MB/s Cycles per byte =
      7.56
AES-192-CBC-enc    265 MB took 1.010 seconds,  262.272 MB/s Cycles per byte =
      8.35
AES-192-CBC-dec    240 MB took 1.013 seconds,  236.844 MB/s Cycles per byte =
      9.24
AES-256-CBC-enc    240 MB took 1.011 seconds,  237.340 MB/s Cycles per byte =
      9.22
AES-256-CBC-dec    235 MB took 1.018 seconds,  230.864 MB/s Cycles per byte =
      9.48
AES-128-GCM-enc    160 MB took 1.011 seconds,  158.253 MB/s Cycles per byte =
     13.83
AES-128-GCM-dec    160 MB took 1.016 seconds,  157.508 MB/s Cycles per byte =
     13.90
AES-192-GCM-enc    150 MB took 1.022 seconds,  146.815 MB/s Cycles per byte =
     14.91
AES-192-GCM-dec    150 MB took 1.039 seconds,  144.419 MB/s Cycles per byte =
     15.16
AES-256-GCM-enc    130 MB took 1.017 seconds,  127.889 MB/s Cycles per byte =
     17.12
AES-256-GCM-dec    140 MB took 1.030 seconds,  135.943 MB/s Cycles per byte =
     16.10
GMAC Table 4-bit   321 MB took 1.002 seconds,  320.457 MB/s Cycles per byte =
      6.83
CHACHA             420 MB took 1.002 seconds,  419.252 MB/s Cycles per byte =
      5.22
CHA-POLY           330 MB took 1.013 seconds,  325.735 MB/s Cycles per byte =
      6.72
MD5                655 MB took 1.007 seconds,  650.701 MB/s Cycles per byte =
      3.36
POLY1305          1490 MB took 1.002 seconds, 1486.840 MB/s Cycles per byte =
      1.47
SHA                560 MB took 1.004 seconds,  557.620 MB/s Cycles per byte =
      3.93
SHA-224            240 MB took 1.011 seconds,  237.474 MB/s Cycles per byte =
      9.22
SHA-256            250 MB took 1.020 seconds,  245.081 MB/s Cycles per byte =
      8.93
SHA-384            380 MB took 1.005 seconds,  377.963 MB/s Cycles per byte =
      5.79
SHA-512            380 MB took 1.007 seconds,  377.260 MB/s Cycles per byte =
      5.80
SHA3-224           385 MB took 1.009 seconds,  381.679 MB/s Cycles per byte =
      5.74
SHA3-256           360 MB took 1.004 seconds,  358.583 MB/s Cycles per byte =
      6.11
SHA3-384           270 MB took 1.020 seconds,  264.606 MB/s Cycles per byte =
      8.27
SHA3-512           185 MB took 1.019 seconds,  181.573 MB/s Cycles per byte =
     12.06
```

```
HMAC-MD5           665 MB took 1.004 seconds,  662.154 MB/s Cycles per byte =
     3.31
HMAC-SHA           590 MB took 1.004 seconds,  587.535 MB/s Cycles per byte =
     3.73
HMAC-SHA224        240 MB took 1.018 seconds,  235.850 MB/s Cycles per byte =
     9.28
HMAC-SHA256        245 MB took 1.013 seconds,  241.805 MB/s Cycles per byte =
     9.05
HMAC-SHA384        365 MB took 1.006 seconds,  362.678 MB/s Cycles per byte =
     6.04
HMAC-SHA512        365 MB took 1.009 seconds,  361.674 MB/s Cycles per byte =
     6.05
PBKDF2              30 KB took 1.000 seconds,   29.956 KB/s Cycles per byte =
    74838.56
RSA     2048 public      18400 ops took 1.004 sec, avg 0.055 ms, 18335.019 ops
    /sec
RSA     2048 private       300 ops took 1.215 sec, avg 4.050 ms, 246.891 ops/
    sec
DH      2048 key gen      1746 ops took 1.000 sec, avg 0.573 ms, 1745.991 ops/
    sec
DH      2048 agree         900 ops took 1.060 sec, avg 1.178 ms, 849.210 ops/
    sec
ECC   [      SECP256R1]   256 key gen       901 ops took 1.000 sec, avg 1.110
    ms, 900.779 ops/sec
ECDHE [      SECP256R1]   256 agree        1000 ops took 1.105 sec, avg 1.105
    ms, 904.767 ops/sec
ECDSA [      SECP256R1]   256 sign          900 ops took 1.022 sec, avg 1.135
    ms, 880.674 ops/sec
ECDSA [      SECP256R1]   256 verify       1300 ops took 1.012 sec, avg 0.779
    ms, 1284.509 ops/sec
Benchmark complete
```

This is especially useful for comparing the public key speed before and after changing the math library. You can test the results using the normal math library (`./configure`), the fastmath library (`./configure --enable-fastmath`), and the fasthugemath library (`./configure --enable-fasthugemath`).

For more details and benchmark results, please refer to the wolfSSL Benchmarks page: https://www.wolfssl.com/docs/benchmarks

3.7.1 Relative Performance

Although the performance of individual ciphers and algorithms will depend on the host platform, the following graph shows relative performance between wolfCrypt's ciphers. These tests were conducted on a Macbook Pro (OS X 10.6.8) running a 2.2 GHz Intel Core i7.

If you want to use only a subset of ciphers, you can customize which specific cipher suites and/or ciphers wolfSSL uses when making an SSL/TLS connection. For example, to force 128-bit AES, add the following line after the call to `wolfSSL_CTX_new(SSL_CTX_new)`:

```
wolfSSL_CTX_set_cipher_list(ctx, "AES128-SHA");
```

3.7.2 Benchmarking Notes

1. The processors native register size (32 vs 64-bit) can make a big difference when doing 1000+ bit public key operations.

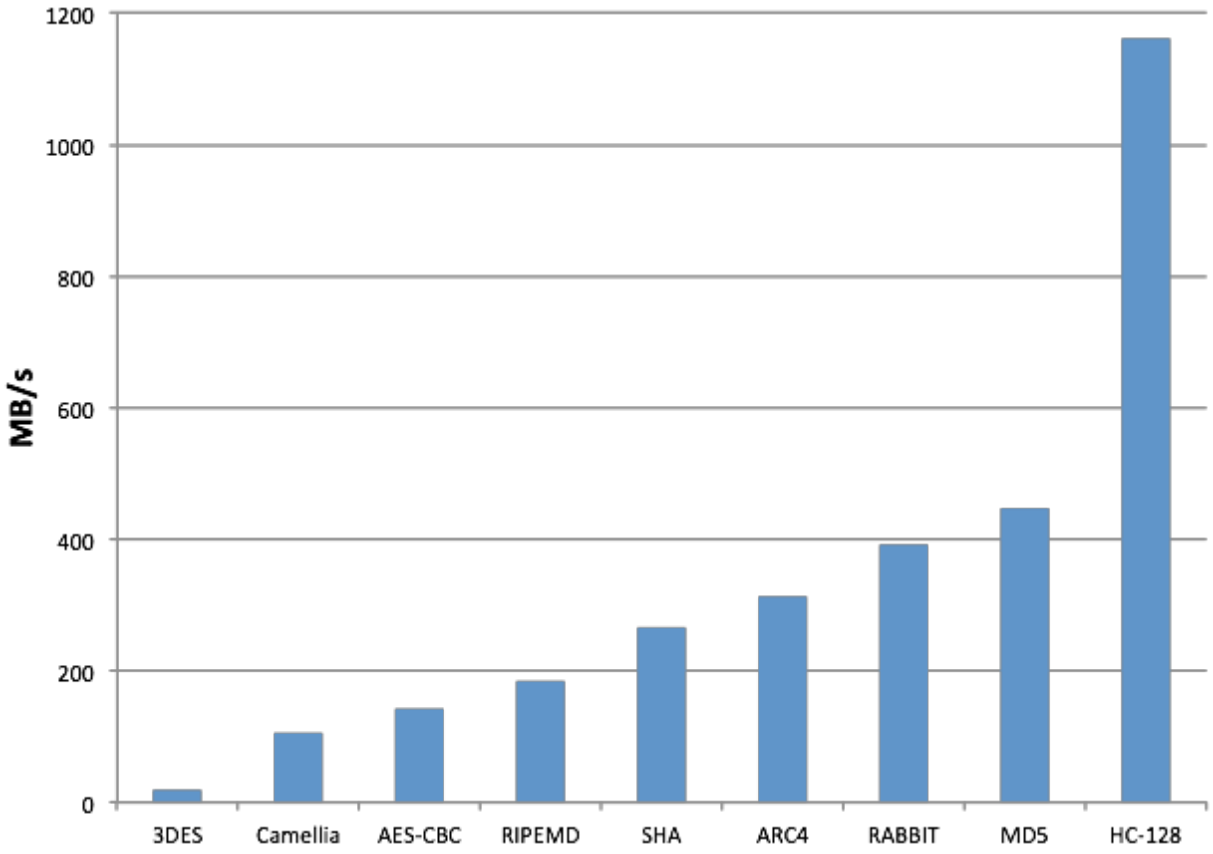

Figure 1: Benchmark

2. **keygen** (`--enable-keygen`) will allow you to also benchmark key generation speeds when running the benchmark utility.
3. **fastmath** (`--enable-fastmath`) reduces dynamic memory usage and speeds up public key operations. If you are having trouble building on 32-bit platform with fastmath, disable shared libraries so that PIC isn't hogging a register (also see notes in the README):

   ```
   ./configure --enable-fastmath --disable-shared
   make clean
   make
   ```

   Note: doing a `make clean` is good practice with wolfSSL when switching configure options.
4. By default, fastmath tries to use assembly optimizations if possible. If assembly optimizations don't work, you can still use fastmath without them by adding `TFM_NO_ASM` to `CFLAGS` when building wolfSSL:

   ```
   ./configure --enable-fastmath C_EXTRA_FLAGS="-DTFM_NO_ASM"
   ```
5. Using fasthugemath can try to push fastmath even more for users who are not running on embedded platforms:

   ```
   ./configure --enable-fasthugemath
   ```
6. With the default wolfSSL build, we have tried to find a good balance between memory usage and performance. If you are more concerned about one of the two, please refer back to Build Options for additional wolfSSL configuration options.
7. **Bulk Transfers**: wolfSSL by default uses 128 byte I/O buffers since about 80% of SSL traffic falls within this size and to limit dynamic memory use. It can be configured to use 16K buffers (the maximum SSL size) if bulk transfers are required.

3.7.3 Benchmarking on Embedded Systems

There are several build options available to make building the benchmark application on an embedded system easier. These include:

3.7.3.1 BENCH_EMBEDDED Enabling this define will switch the benchmark application from using Megabytes to using Kilobytes, therefore reducing the memory usage. By default, when using this define, ciphers and algorithms will be benchmarked with 25kB. Public key algorithms will only be benchmarked over 1 iteration (as public key operations on some embedded processors can be fairly slow). These can be adjusted in `benchmark.c` by altering the variables `numBlocks` and `times` located inside the `BENCH_EMBEDDED` define.

3.7.3.2 USE_CERT_BUFFERS_1024 Enabling this define will switch the benchmark application from loading test keys and certificates from the file system and instead use 1024-bit key and certificate buffers located in `<wolfssl_root>/wolfssl/certs_test.h`. It is useful to use this define when an embedded platform has no filesystem (used with `NO_FILESYSTEM`) and a slow processor where 2048-bit public key operations may not be reasonable.

3.7.3.3 USE_CERT_BUFFERS_2048 Enabling this define is similar to `USE_CERT_BUFFERS_1024` except that 2048-bit key and certificate buffers are used instead of 1024-bit ones. This define is useful when the processor is fast enough to do 2048-bit public key operations but when there is no filesystem available to load keys and certificates from files.

3.8 Changing a Client Application to Use wolfSSL

This section will explain the basic steps needed to add wolfSSL to a client application, using the wolfSSL native API. For a server explanation, please see Changing a Server Application to Use wolfSSL. A more complete walk-through with example code is located in the SSL Tutorial in Chapter 11. If you want more information about the OpenSSL compatibility layer, please see OpenSSL Compatibility.

1. Include the wolfSSL header:

   ```
   #include <wolfssl/ssl.h>
   ```

2. Initialize wolfSSL and the WOLFSSL_CTX. You can use one WOLFSSL_CTX no matter how many WOLFSSL objects you end up creating. You'll just need to load CA certificates to verify the server you are connecting to. Basic initialization looks like:

   ```
   wolfSSL_Init();
   WOLFSSL_CTX* ctx;
   if ((ctx = wolfSSL_CTX_new(wolfTLSv1_client_method())) == NULL)
   {
       fprintf(stderr, "wolfSSL_CTX_new error.\n");
       exit(EXIT_FAILURE);
   }
   if (wolfSSL_CTX_load_verify_locations(ctx,"./ca-cert.pem",0) !=
    ↪  SSL_SUCCESS) {
       fprintf(stderr, "Error loading ./ca-cert.pem,"
                       " please check the file.\n");
       exit(EXIT_FAILURE);
   }
   ```

3. Create the WOLFSSL object after each TCP connect and associate the file descriptor with the session:

   ```
   /*after connecting to socket fd*/
   WOLF SSL* ssl;
   if ((ssl = wolfSSL_new(ctx)) == NULL) {
       fprintf(stderr, "wolfSSL_new error.\n");
       exit(EXIT_FAILURE);
   }
   wolfSSL_set_fd(ssl, fd);
   ```

4. Change all calls from read() (or recv()) to wolfSSL_read() so:

   ```
   result = read(fd, buffer, bytes);
   ```

   becomes:

   ```
   result = wolfSSL_read(ssl, buffer, bytes);
   ```

5. Change all calls from write() (or send()) to wolfSSL_write() so:

   ```
   result = write(fd, buffer, bytes);
   ```

   becomes

   ```
   result = wolfSSL_write(ssl, buffer, bytes);
   ```

6. You can manually call wolfSSL_connect() if it hasn't taken place yet.

7. Error checking. Each wolfSSL_read() call will return the number of bytes written upon success, 0 upon connection closure, and -1 for an error, just like read() and write(). In the event of an error you can use two calls to get more information about the error:

```
char errorString[80];
int err = wolfSSL_get_error(ssl, 0);
wolfSSL_ERR_error_string(err, errorString);
```

If you are using non-blocking sockets, you can test for errno EAGAIN/EWOULDBLOCK or more correctly you can test the specific error code returned by `wolfSSL_get_error()` for SSL_ERROR_WANT_READ orSSL_ERROR_WANT_WRITE.

8. Cleanup. After each WOLFSSL object is done being used you can free it up by calling:

   ```
   wolfSSL_free(ssl);
   ```

   When you are completely done using SSL/TLS altogether you can free the WOLFSSL_CTX object by calling:

   ```
   wolfSSL_CTX_free(ctx);
   wolfSSL_Cleanup();
   ```

   For an example of a client application using wolfSSL, see the client example located in the <wolfssl_root>/examples/client.c file.

3.9 Changing a Server Application to Use wolfSSL

This section will explain the basic steps needed to add wolfSSL to a server application using the wolfSSL native API. For a client explanation, please see Changing a Client Application to Use wolfSSL. A more complete walk-through, with example code, is located in the SSL Tutorial chapter.

1. Follow the instructions above for a client, except change the client method call in step 5 to a server one, so:

   ```
   wolfSSL_CTX_new(wolfTLSv1_client_method());
   ```

   becomes:

   ```
   wolfSSL_CTX_new(wolfTLSv1_server_method());
   ```

   or even:

   ```
   wolfSSL_CTX_new(wolfSSLv23_server_method());
   ```

   To allow SSLv3 and TLSv1+ clients to connect to the server.

2. Add the server's certificate and key file to the initialization in step 5 above:

   ```
   if (wolfSSL_CTX_use_certificate_file(ctx,"./server-cert.pem",
    ↪  SSL_FILETYPE_PEM) != SSL_SUCCESS) {
       fprintf(stderr, "Error loading ./server-cert.pem,"
                       " please check the file.\n");
       exit(EXIT_FAILURE);
   }
   if (wolfSSL_CTX_use_PrivateKey_file(ctx,"./server-key.pem",
    ↪  SSL_FILETYPE_PEM) != SSL_SUCCESS) {
       fprintf(stderr, "Error loading ./server-key.pem,"
                       " please check the file.\n");
       exit(EXIT_FAILURE);
   }
   ```

It is possible to load certificates and keys from buffers as well if there is no filesystem available. In this case, see the `wolfSSL_CTX_use_certificate_buffer()` API documentation, linked here, for more information.

For an example of a server application using wolfSSL, see the server example located in the <wolfssl_root>/examples/server.c file.

4 Features

wolfSSL (formerly CyaSSL) supports the C programming language as a primary interface, but also supports several other host languages, including Java, PHP, Perl, and Python (through a SWIG interface). If you have interest in hosting wolfSSL in another programming language that is not currently supported, please contact us.

This chapter covers some of the features of wolfSSL in more depth, including Stream Ciphers, AES-NI, IPv6 support, SSL Inspection (Sniffer) support, and more.

4.1 Features Overview

For an overview of wolfSSL features, please reference the wolfSSL product webpage: https://www.wolfssl.com/products/wolfssl

4.2 Protocol Support

wolfSSL supports **SSL 3.0**, **TLS** (**1.0**, **1.1**, **1.2, 1.3**), and **DTLS** (**1.0**, **1.2**, **1.3**). You can easily select a protocol to use by using one of the following functions (as shown for either the client or server). wolfSSL does not support SSL 2.0, as it has been insecure for several years. The client and server functions below change slightly when using the OpenSSL compatibility layer. For the OpenSSL-compatible functions, please see OpenSSL Compatibility.

4.2.1 Server Functions

- `wolfDTLSv1_server_method()` - DTLS 1.0
- `wolfDTLSv1_2_server_method()` - DTLS 1.2
- `wolfSSLv3_server_method()` - SSL 3.0
- `wolfTLSv1_server_method()` - TLS 1.0
- `wolfTLSv1_1_server_method()` - TLS 1.1
- `wolfTLSv1_2_server_method()` - TLS 1.2
- `wolfTLSv1_3_server_method()` - TLS 1.3
- `wolfSSLv23_server_method()` - Use highest possible version from SSLv3 - TLS 1.2

wolfSSL supports robust server downgrade with the `wolfSSLv23_server_method()` function. See Robust Client and Server Downgrade for details.

4.2.2 Client Functions

- `wolfDTLSv1_client_method()` - DTLS 1.0
- `wolfDTLSv1_2_client_method_ex()` - DTLS 1.2
- `wolfSSLv3_client_method()` - SSL 3.0
- `wolfTLSv1_client_method()` - TLS 1.0
- `wolfTLSv1_1_client_method()` - TLS 1.1
- `wolfTLSv1_2_client_method()` - TLS 1.2
- `wolfTLSv1_3_client_method()` - TLS 1.3
- `wolfSSLv23_client_method()` - Use highest possible version from SSLv3 - TLS 1.2

wolfSSL supports robust client downgrade with the `wolfSSLv23_client_method()` function. See Robust Client and Server Downgrade for details.

For details on how to use these functions, please see the Getting Started chapter. For a comparison between SSL, TLS, and DTLS, please see Appendix A.

COPYRIGHT ©2024 wolfSSL Inc.

4.2.3 Robust Client and Server Downgrade

Both wolfSSL clients and servers have robust version downgrade capability. If a specific protocol version method is used on either side, then only that protocol version will be negotiated or an error will be returned. For example, if a client that uses TLS 1.0 tries to connect to an SSL 3.0 only server, then the connection will fail; likewise, connecting to a TLS 1.1 server will fail as well.

To resolve this issue, a client that uses the `wolfSSLv23_client_method()` function will support the highest protocol version supported by the server by downgrading if necessary. In this case, the client will be able to connect to a server running TLS 1.0 - TLS 1.3 (or a subset or superset that includes SSL 3.0 depending on which protocol versions are configured in wolfSSL). The only versions it can't connect to are SSL 2.0 which has been insecure for years, and SSL 3.0 which has been disabled by default.

Similarly, a server using the `wolfSSLv23_server_method()` function can handle clients supporting protocol versions from TLS 1.0 - TLS 1.3. A wolfSSL server can't accept a connection from SSLv2 because no security is provided.

4.2.4 IPv6 Support

If you are an adopter of IPv6 and want to use an embedded SSL implementation then you may have been wondering if wolfSSL supports IPv6. The answer is yes, we do support wolfSSL running on top of IPv6.

wolfSSL was designed as IP neutral, and will work with both IPv4 and IPv6, but the current test applications default to IPv4 (so as to apply to a broader range of systems). To change the test applications to IPv6, use the **–enable-ipv6** option while building wolfSSL.

Further information on IPv6 can be found here:

https://en.wikipedia.org/wiki/IPv6.

4.2.5 DTLS

wolfSSL has support for DTLS ("Datagram" TLS) for both client and server. The current supported versions are DTLS 1.0, 1.2, and 1.3.

The TLS protocol was designed to provide a secure transport channel across a **reliable** medium (such as TCP). As application layer protocols began to be developed using UDP transport (such as SIP and various electronic gaming protocols), a need arose for a way to provide communications security for applications which are delay sensitive. This need lead to the creation of the DTLS protocol.

Many people believe the difference between TLS and DTLS is the same as TCP vs. UDP. This is incorrect. UDP has the benefit of having no handshake, no tear-down, and no delay in the middle if something gets lost (compared with TCP). DTLS on the other hand, has an extended SSL handshake and tear-down and must implement TCP-like behavior for the handshake. In essence, DTLS reverses the benefits that are offered by UDP in exchange for a secure connection.

DTLS can be enabled when building wolfSSL by using the `--enable-dtls` build option.

4.2.6 LwIP (Lightweight Internet Protocol)

wolfSSL supports the lightweight internet protocol implementation out of the box. To use this protocol all you need to do is define `WOLFSSL_LWIP` or navigate to the `settings.h` file and uncomment the line:

```
/*#define WOLFSSL_LWIP*/
```

The focus of lwIP is to reduce RAM usage while still providing a full TCP stack. That focus makes lwIP great for use in embedded systems, an area where wolfSSL is an ideal match for SSL/TLS needs.

4.2.7 TLS Extensions

A list of TLS extensions supported by wolfSSL and the RFC corresponding to each extension:

RFC	Extension	wolfSSL Type
6066	Server Name Indication	TLSX_SERVER_NAME
6066	Maximum Fragment Length Negotiation	TLSX_MAX_FRAGMENT_LENGTH
6066	Truncated HMAC	TLSX_TRUNCATED_HMAC
6066	Status Request	TLSX_STATUS_REQUEST
7919	Supported Groups	TLSX_SUPPORTED_GROUPS
5246	Signature Algorithm	TLSX_SIGNATURE_ALGORITHMS
7301	Application Layer Protocol Negotiation	TLSX_APPLICATION_LAYER_PROTOCOL
6961	Multiple Certificate Status Request	TLSX_STATUS_REQUEST_V2
5077	Session Ticket	TLSX_SESSION_TICKET
5746	Renegotiation Indication	TLSX_RENEGOTIATION_INFO
8446	Key Share	TLSX_KEY_SHARE
8446	Pre Shared Key	TLSX_PRE_SHARED_KEY
8446	PSK Key Exchange Modes	TLSX_PSK_KEY_EXCHANGE_MODES
8446	Early Data	TLSX_EARLY_DATA
8446	Cookie	TLSX_COOKIE
8446	Supported Versions	TLSX_SUPPORTED_VERSIONS
8446	Post Handshake Authorization	TLSX_POST_HANDSHAKE_AUTH

4.3 Cipher Support

4.3.1 Cipher Suite Strength and Choosing Proper Key Sizes

To see what ciphers are currently being used you can call the method: `wolfSSL_get_ciphers()`.

This function will return the currently enabled cipher suites.

Cipher suites come in a variety of strengths. Because they are made up of several different types of algorithms (authentication, encryption, and message authentication code (MAC)), the strength of each varies with the chosen key sizes.

There can be many methods of grading the strength of a cipher suite - the specific method used seems to vary between different projects and companies and can include things such as symmetric and public key algorithm key sizes, type of algorithm, performance, and known weaknesses.

NIST (National Institute of Standards and Technology) makes recommendations on choosing an acceptable cipher suite by providing comparable algorithm strengths for varying key sizes of each. The NIST Special Publication, SP800-57, states how two algorithms can be compared:

> Two algorithms are considered to be of comparable strength for the given key sizes (X and Y) if the amount of work needed to "break the algorithms" or determine the keys (with the given key sizes) is approximately the same using a given resource. The security strength of an algorithm for a given key size is traditionally described in terms of the amount of work it takes to try all keys for a symmetric algorithm with a key size of "X" that has no shortcut attacks (i.e., the most efficient attack is to try all possible keys).

The following two tables are based off of both Table 2 (pg. 56) and Table 4 (pg. 59) from NIST SP800-57, and shows comparable security strength between algorithms as well as a strength measurement (based off of NIST's suggested algorithm security lifetimes using bits of security).

Note: In the following table "L" is the size of the public key for finite field cryptography (FFC), "N" is the size of the private key for FFC, "k" is considered the key size for integer factorization cryptography (IFC), and "f" is considered the key size for elliptic curve cryptography.

Bits of Security	Symmetric Key Algorithms	**FFC** Key Size (DSA, DH, etc.)	**IFC** Key Size (RSA, etc.)	**ECC** Key Size (ECDSA, etc.)	Description
80	2TDEA, etc.	L = 1024, N = 160	k = 1024	f = 160-223	Security good through 2010
128	AES-128, etc.	L = 3072, N = 256	k = 3072	f = 256-383	Security good through 2030
192	AES-192, etc.	L = 7680, N = 384	k = 7680	f = 384-511	Long Term Protection
256	AES-256, etc.	L = 15360, N = 512	k = 15360	f = 512+	Secure for the fore-see-able future

Using this table as a guide, to begin to classify a cipher suite, we categorize it based on the strength of the symmetric encryption algorithm. In doing this, a rough grade classification can be devised to classify each cipher suite based on bits of security (only taking into account symmetric key size):

- **LOW** - bits of security smaller than 128 bits
- **MEDIUM** - bits of security equal to 128 bits
- **HIGH** - bits of security larger than 128 bits

Outside of the symmetric encryption algorithm strength, the strength of a cipher suite will depend greatly on the key sizes of the key exchange and authentication algorithm keys. The strength is only as good as the cipher suite's weakest link.

Following the above grading methodology (and only basing it on symmetric encryption algorithm strength), wolfSSL 2.0.0 currently supports a total of 0 LOW strength cipher suites, 12 MEDIUM strength cipher suites, and 8 HIGH strength cipher suites – as listed below. The following strength classification could change depending on the chosen key sizes of the other algorithms involved. For a reference on hash function security strength, see Table 3 (pg. 56) of NIST SP800-57.

In some cases, you will see ciphers referenced as "**EXPORT**" ciphers. These ciphers originated from the time period in US history (as late as 1992) when it was illegal to export software with strong encryption from the United States. Strong encryption was classified as "Munitions" by the US Government (under the same category as Nuclear Weapons, Tanks, and Ballistic Missiles). Because of this restriction, software being exported included "weakened" ciphers (mostly in smaller key sizes). In the current day, this restriction has been lifted, and as such, EXPORT ciphers are no longer a mandated necessity.

4.3.2 Supported Cipher Suites

The following cipher suites are supported by wolfSSL. A cipher suite is a combination of authentication, encryption, and message authentication code (MAC) algorithms which are used during the TLS or SSL

handshake to negotiate security settings for a connection.

Each cipher suite defines a key exchange algorithm, a bulk encryption algorithm, and a message authentication code algorithm (MAC). The **key exchange algorithm** (RSA, DSS, DH, EDH) determines how the client and server will authenticate during the handshake process. The **bulk encryption algorithm** (DES, 3DES, AES, ARC4), including block ciphers and stream ciphers, is used to encrypt the message stream. The **message authentication code (MAC) algorithm** (MD2, MD5, SHA-1, SHA-256, SHA-512, RIPEMD) is a hash function used to create the message digest.

The table below matches up to the cipher suites (and categories) found in `<wolfssl_root>/wolfssl/internal.h` (starting at about line 706). If you are looking for a cipher suite which is not in the following list, please contact us to discuss getting it added to wolfSSL.

ECC cipher suites:

- TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA
- TLS_DHE_RSA_WITH_AES_256_CBC_SHA
- TLS_DHE_RSA_WITH_AES_128_CBC_SHA
- TLS_DH_anon_WITH_AES_128_CBC_SHA
- TLS_RSA_WITH_AES_256_CBC_SHA
- TLS_RSA_WITH_AES_128_CBC_SHA
- TLS_RSA_WITH_NULL_SHA
- TLS_PSK_WITH_AES_256_CBC_SHA
- TLS_PSK_WITH_AES_128_CBC_SHA256
- TLS_PSK_WITH_AES_256_CBC_SHA384
- TLS_PSK_WITH_AES_128_CBC_SHA
- TLS_PSK_WITH_NULL_SHA256
- TLS_PSK_WITH_NULL_SHA384
- TLS_PSK_WITH_NULL_SHA
- SSL_RSA_WITH_RC4_128_SHA
- SSL_RSA_WITH_RC4_128_MD5
- SSL_RSA_WITH_3DES_EDE_CBC_SHA
- TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA
- TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA
- TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA
- TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA
- TLS_ECDHE_RSA_WITH_RC4_128_SHA
- TLS_ECDHE_ECDSA_WITH_RC4_128_SHA
- TLS_ECDHE_RSA_WITH_3DES_EDE_CBC_SHA
- TLS_ECDHE_ECDSA_WITH_3DES_EDE_CBC_SHA
- TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256
- TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256
- TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384
- TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384
- TLS_ECDHE_PSK_WITH_NULL_SHA256
- TLS_ECDHE_PSK_WITH_AES_128_CBC_SHA256
- TLS_ECDHE_ECDSA_WITH_NULL_SHA

Static ECC cipher suites:

- TLS_ECDH_RSA_WITH_AES_256_CBC_SHA
- TLS_ECDH_RSA_WITH_AES_128_CBC_SHA
- TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA
- TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA
- TLS_ECDH_RSA_WITH_RC4_128_SHA
- TLS_ECDH_ECDSA_WITH_RC4_128_SHA
- TLS_ECDH_RSA_WITH_3DES_EDE_CBC_SHA

- TLS_ECDH_ECDSA_WITH_3DES_EDE_CBC_SHA
- TLS_ECDH_RSA_WITH_AES_128_CBC_SHA256
- TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA256
- TLS_ECDH_RSA_WITH_AES_256_CBC_SHA384
- TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA384

Blake2b cipher suites:

- TLS_RSA_WITH_AES_128_CBC_B2B256
- TLS_RSA_WITH_AES_256_CBC_B2B256

SHA-256 cipher suites:

- TLS_DHE_RSA_WITH_AES_256_CBC_SHA256
- TLS_DHE_RSA_WITH_AES_128_CBC_SHA256
- TLS_RSA_WITH_AES_256_CBC_SHA256
- TLS_RSA_WITH_AES_128_CBC_SHA256
- TLS_RSA_WITH_NULL_SHA256
- TLS_DHE_PSK_WITH_AES_128_CBC_SHA256
- TLS_DHE_PSK_WITH_NULL_SHA256

SHA-384 cipher suites:

- TLS_DHE_PSK_WITH_AES_256_CBC_SHA384
- TLS_DHE_PSK_WITH_NULL_SHA384

AES-GCM cipher suites:

- TLS_RSA_WITH_AES_128_GCM_SHA256
- TLS_RSA_WITH_AES_256_GCM_SHA384
- TLS_DHE_RSA_WITH_AES_128_GCM_SHA256
- TLS_DHE_RSA_WITH_AES_256_GCM_SHA384
- TLS_PSK_WITH_AES_128_GCM_SHA256
- TLS_PSK_WITH_AES_256_GCM_SHA384
- TLS_DHE_PSK_WITH_AES_128_GCM_SHA256
- TLS_DHE_PSK_WITH_AES_256_GCM_SHA384

ECC AES-GCM cipher suites:

- TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256
- TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384
- TLS_ECDH_ECDSA_WITH_AES_128_GCM_SHA256
- TLS_ECDH_ECDSA_WITH_AES_256_GCM_SHA384
- TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
- TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
- TLS_ECDH_RSA_WITH_AES_128_GCM_SHA256
- TLS_ECDH_RSA_WITH_AES_256_GCM_SHA384

AES-CCM cipher suites:

- TLS_RSA_WITH_AES_128_CCM_8
- TLS_RSA_WITH_AES_256_CCM_8
- TLS_ECDHE_ECDSA_WITH_AES_128_CCM
- TLS_ECDHE_ECDSA_WITH_AES_128_CCM_8
- TLS_ECDHE_ECDSA_WITH_AES_256_CCM_8
- TLS_PSK_WITH_AES_128_CCM
- TLS_PSK_WITH_AES_256_CCM
- TLS_PSK_WITH_AES_128_CCM_8
- TLS_PSK_WITH_AES_256_CCM_8
- TLS_DHE_PSK_WITH_AES_128_CCM

- TLS_DHE_PSK_WITH_AES_256_CCM

Camellia cipher suites:

- TLS_RSA_WITH_CAMELLIA_128_CBC_SHA
- TLS_RSA_WITH_CAMELLIA_256_CBC_SHA
- TLS_RSA_WITH_CAMELLIA_128_CBC_SHA256
- TLS_RSA_WITH_CAMELLIA_256_CBC_SHA256
- TLS_DHE_RSA_WITH_CAMELLIA_128_CBC_SHA
- TLS_DHE_RSA_WITH_CAMELLIA_256_CBC_SHA
- TLS_DHE_RSA_WITH_CAMELLIA_128_CBC_SHA256
- TLS_DHE_RSA_WITH_CAMELLIA_256_CBC_SHA256

ChaCha cipher suites:

- TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256
- TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305_SHA256
- TLS_DHE_RSA_WITH_CHACHA20_POLY1305_SHA256
- TLS_ECDHE_PSK_WITH_CHACHA20_POLY1305_SHA256
- TLS_PSK_WITH_CHACHA20_POLY1305_SHA256
- TLS_DHE_PSK_WITH_CHACHA20_POLY1305_SHA256
- TLS_ECDHE_RSA_WITH_CHACHA20_OLD_POLY1305_SHA256
- TLS_ECDHE_ECDSA_WITH_CHACHA20_OLD_POLY1305_SHA256
- TLS_DHE_RSA_WITH_CHACHA20_OLD_POLY1305_SHA256

Renegotiation Indication Extension Special Suite:

- TLS_EMPTY_RENEGOTIATION_INFO_SCSV

4.3.3 AEAD Suites

wolfSSL supports AEAD suites, including AES-GCM, AES-CCM, and CHACHA-POLY1305. The big difference between these AEAD suites and others is that they authenticate the encrypted data with any additional cleartext data. This helps with mitigating man in the middle attacks that result in having data tampered with. AEAD suites use a combination of a block cipher (or more recently also a stream cipher) algorithm combined with a tag produced by a keyed hash algorithm. Combining these two algorithms is handled by the wolfSSL encrypt and decrypt process which makes it easier for users. All that is needed for using a specific AEAD suite is simply enabling the algorithms that are used in a supported suite.

4.3.4 Block and Stream Ciphers

wolfSSL supports the **AES**, **DES**, **3DES**, and **Camellia** block ciphers and the **RC4**, and **CHACHA20** stream ciphers. AES, DES, 3DES, RC4, and ChaCha20 are enabled by default. Camellia can be enabled when building wolfSSL with the `--enable-camellia` build option. The default mode of AES is CBC mode. To enable GCM or CCM mode with AES, use the `--enable-aesgcm` and `--enable-aesccm` build options. Please see the examples for usage and the wolfCrypt Usage Reference for specific usage information.

While SSL uses RC4 as the default stream cipher, it has been obsoleted due to compromise. Recently wolfSSL added ChaCha20. While RC4 is about 11% more performant than ChaCha, RC4 is generally considered less secure than ChaCha. ChaCha can put up very nice times of it's own with added security as a tradeoff.

To see a comparison of cipher performance, visit the wolfSSL Benchmark web page, located here: https://www.wolfssl.com/docs/benchmarks.

4.3.4.1 What's the Difference? A block cipher has to be encrypted in chunks that are the block size for the cipher. For example, AES has a block size of 16 bytes. So if you're encrypting a bunch of small, 2 or 3 byte chunks back and forth, over 80% of the data is useless padding, decreasing the speed of the encryption/decryption process and needlessly wasting network bandwidth to boot. Basically, block ciphers are designed for large chunks of data, have block sizes requiring padding, and use a fixed, unvarying transformation.

Stream ciphers work well for large or small chunks of data. They are suitable for smaller data sizes because no block size is required. If speed is a concern, stream ciphers are your answer, because they use a simpler transformation that typically involves an xor'd keystream. So if you need to stream media, encrypt various data sizes including small ones, or have a need for a fast cipher then stream ciphers are your best bet.

4.3.5 Hashing Functions

wolfSSL supports several different hashing functions, including **MD2**, **MD4**, **MD5**, **SHA-1**, **SHA-2** (SHA-224, SHA-256, SHA-384, SHA-512), **SHA-3** (BLAKE2), and **RIPEMD-160**. Detailed usage of these functions can be found in the wolfCrypt Usage Reference, Hash Functions.

4.3.6 Public Key Options

wolfSSL supports the **RSA**, **ECC**, **DSA/DSS** and **DH** public key options, with support for **EDH** (Ephemeral Diffie-Hellman) on the wolfSSL server. Detailed usage of these functions can be found in the wolfCrypt Usage Reference, Public Key Cryptography.

4.3.7 ECC Support

wolfSSL has support for Elliptic Curve Cryptography (ECC) including but not limited to: ECDH-ECDSA, ECDHE-ECDSA, ECDH-RSA, ECDHE-PSK and ECDHE-RSA.

wolfSSL's ECC implementation can be found in the `<wolfssl_root>/wolfssl/wolfcrypt/ecc.h` header file and the `<wolfssl_root>/wolfcrypt/src/ecc.c` source file.

Supported cipher suites are shown in the table above. ECC is disabled by default on non x86_64 builds, but can be turned on when building wolfSSL with the HAVE_ECC define or by using the autoconf system:

```
./configure --enable-ecc
make
make check
```

When `make check` runs, note the numerous cipher suites that wolfSSL checks (if make check doesn't produce a list of cipher suites run `./testsuite/testsuite.test` on its own). Any of these cipher suites can be tested individually, e.g., to try ECDH-ECDSA with AES256-SHA, the example wolfSSL server can be started like this:

```
./examples/server/server -d -l ECDHE-ECDSA-AES256-SHA -c
 ↪  ./certs/server-ecc.pem -k ./certs/ecc-key.pem
```

(`-d`) disables client cert check while (`-l`) specifies the cipher suite list. (`-c`) is the certificate to use and (`-k`) is the corresponding private key to use. To have the client connect try:

```
./examples/client/client -A ./certs/server-ecc.pem
```

where (`-A`) is the CA certificate to use to verify the server.

4.3.8 PKCS Support

PKCS (Public Key Cryptography Standards) refers to a group of standards created and published by RSA Security, Inc. wolfSSL has support for **PKCS #1, PKCS #3, PKCS #5, PKCS #7, PKCS #8, PKCS #9, PKCS #10, PKCS #11,** and **PKCS #12.**

Additionally, wolfSSL also provides support for RSA-Probabilistic Signature Scheme (PSS), which is standardized as part of PKCS #1.

4.3.8.1 PKCS #5, PBKDF1, PBKDF2, PKCS #12 PKCS #5 is a password-based key derivation method which combines a password, a salt, and an iteration count to generate a password-based key. wolfSSL supports both PBKDF1 and PBKDF2 key derivation functions. A key derivation function produces a derived key from a base key and other parameters (such as the salt and iteration count as explained above). PBKDF1 applies a hash function (MD5, SHA1, etc) to derive keys, where the derived key length is bounded by the length of the hash function output. With PBKDF2, a pseudorandom function is applied (such as HMAC-SHA-1) to derive the keys. In the case of PBKDF2, the derived key length is unbounded.

wolfSSL also supports the PBKDF function from PKCS #12 in addition to PBKDF1 and PBKDF2. The function prototypes look like this:

```
int PBKDF2(byte* output, const byte* passwd, int pLen,
           const byte* salt,int sLen, int iterations,
           int kLen, int hashType);

int PKCS12_PBKDF(byte* output, const byte* passwd, int pLen,
                 const byte* salt, int sLen, int iterations,
                 int kLen, int hashType, int purpose);
```

`output` contains the derived key, `passwd` holds the user password of length `pLen`, `salt` holds the salt input of length `sLen`, `iterations` is the number of iterations to perform, `kLen` is the desired derived key length, and `hashType` is the hash to use (which can be MD5, SHA1, or SHA2).

If you are using `./configure` to build wolfssl, the way to enable this functionality is to use the option `--enable-pwdbased`

A full example can be found in `<wolfSSL Root>/wolfcrypt/test.c`. More information can be found on PKCS #5, PBKDF1, and PBKDF2 from the following specifications:

PKCS#5, PBKDF1, PBKDF2: https://tools.ietf.org/html/rfc2898

4.3.8.2 PKCS #8 PKCS #8 is designed as the Private-Key Information Syntax Standard, which is used to store private key information - including a private key for some public-key algorithm and set of attributes.

The PKCS #8 standard has two versions which describe the syntax to store both encrypted private keys and non-encrypted keys. wolfSSL supports both unencrypted and encrypted PKCS #8. Supported formats include PKCS #5 version 1 - version 2, and PKCS#12. Types of encryption available include DES, 3DES, RC4, and AES.

PKCS#8: https://tools.ietf.org/html/rfc5208

4.3.8.3 PKCS #7 PKCS #7 is designed to transfer bundles of data whether is an enveloped certificate or unencrypted but signed string of data. The functionality is turned on by using the enable option (`--enable-pkcs7`) or by using the macro HAVE_PKCS7. Note that degenerate cases are allowed by default as per the RFC having an empty set of signers. To toggle allowing degenerate cases on and off the function `wc_PKCS7_AllowDegenerate()` can be called.

Supported features include:

- Degenerate bundles
- KARI, KEKRI, PWRI, ORI, KTRI bundles
- Detached signatures
- Compressed and Firmware package bundles
- Custom callback support
- Limited streaming capability

4.3.8.3.1 PKCS #7 Callbacks Additional callbacks and supporting functions were added to allow for a user to choose their keys after the PKCS7 bundle has been parsed. For unwrapping the CEK the function `wc_PKCS7_SetWrapCEKCb()` can be called. The callback set by this function gets called in the case of KARI and KEKRI bundles. The keyID or SKID gets passed from wolfSSL to the user along with the originator key in the case of KARI. After the user unwraps the CEK with their KEK the decrypted key to be used should then be passed back to wolfSSL. An example of this can be found in the wolfssl-examples repository in the file `signedData-EncryptionFirmwareCB.c`.

An additional callback was added for decryption of PKCS7 bundles. For setting a decryption callback function the API `wc_PKCS7_SetDecodeEncryptedCb()` can be used. To set a user defined context the API `wc_PKCS7_SetDecodeEncryptedCtx()` should be used. This callback will get executed on calls to `wc_PKCS7_DecodeEncryptedData()`.

4.3.8.3.2 PKCS #7 Streaming Stream oriented API for PKCS7 decoding gives the option of passing inputs in smaller chunks instead of all at once. By default the streaming functionality with PKCS7 is on. To turn off support for streaming PKCS7 API the macro `NO_PKCS7_STREAM` can be defined. An example of doing this with autotools would be `./configure --enable-pkcs7 CFLAGS=-DNO_PKCS7_STREAM`.

For streaming when decoding/verifying bundles the following functions are supported:

1. `wc_PKCS7_DecodeEncryptedData()`
2. `wc_PKCS7_VerifySignedData()`
3. `wc_PKCS7_VerifySignedData_ex()`
4. `wc_PKCS7_DecodeEnvelopedData()`
5. `wc_PKCS7_DecodeAuthEnvelopedData()`

Note: that when calling `wc_PKCS7_VerifySignedData_ex` it is expected that the argument pkiMsgFoot is the full buffer. The internal structure only supports streaming of one buffer, which in this case would be `pkiMsgHead`.

4.3.9 Forcing the Use of a Specific Cipher

By default, wolfSSL will pick the "best" (highest security) cipher suite that both sides of the connection can support. To force a specific cipher, such as 128 bit AES, add something similar to:

```
wolfSSL_CTX_set_cipher_list(ctx, "AES128-SHA");
```

after the call to `wolfSSL_CTX_new()` so that you have:

```
ctx = wolfSSL_CTX_new(method);
wolfSSL_CTX_set_cipher_list(ctx, "AES128-SHA");
```

4.3.10 OpenQuantumSafe's liboqs Integration

Please see the appendix "Experimenting with Quantum-Safe Cryptography" in this document for more details.

4.4 Hardware Accelerated Crypto

wolfSSL is able to take advantage of several hardware accelerated (or "assisted") crypto functionalities in various processors and chips. The following sections explain which technologies wolfSSL supports out-of-the-box.

4.4.1 AES-NI

AES is a key encryption standard used by governments worldwide, which wolfSSL has always supported. Intel has released a new set of instructions that is a faster way to implement AES. wolfSSL is the first SSL library to fully support the new instruction set for production environments.

Essentially, Intel and AMD have added AES instructions at the chip level that perform the computationally-intensive parts of the AES algorithm, boosting performance. For a list of Intel's chips that currently have support for AES-NI, you can look here:

https://ark.intel.com/search/advanced/?s=t&AESTech=true

We have added the functionality to wolfSSL to allow it to call the instructions directly from the chip, instead of running the algorithm in software. This means that when you're running wolfSSL on a chipset that supports AES-NI, you can run your AES crypto 5-10 times faster!

If you are running on an AES-NI supported chipset, enable AES-NI with the `--enable-aesni` build option. To build wolfSSL with AES-NI, GCC 4.4.3 or later is required to make use of the assembly code. wolfSSL supports the ASM instructions on AMD processors using the same build options.

References and further reading on AES-NI, ordered from general to specific, are listed below. For information about performance gains with AES-NI, please see the third link to the Intel Software Network page.

- AES (Wikipedia)
- AES-NI (Wikipedia)
- AES-NI (Intel Software Network page)

AES-NI will accelerate the following AES cipher modes: AES-CBC, AES-GCM, AES-CCM-8, AES-CCM, and AES-CTR. AES-GCM is further accelerated with the use of the 128-bit multiply function added to the Intel chips for the GHASH authentication.

4.4.2 STM32F2

wolfSSL is able to use the STM32F2 hardware-based cryptography and random number generator through the STM32F2 Standard Peripheral Library.

For necessary defines, see the `WOLFSSL_STM32F2` define in `settings.h`. The `WOLFSSL_STM32F2` define enables STM32F2 hardware crypto and RNG support by default. The defines for enabling these individually are `STM32F2_CRYPTO` (for hardware crypto support) and `STM32F2_RNG` (for hardware RNG support).

Documentation for the STM32F2 Standard Peripheral Library can be found in the following document: https://www.st.com/internet/com/TECHNICAL_RESOURCES/TECHNICAL_LITERATURE/USER_MANUAL/DM00023896.pdf

4.4.3 Cavium NITROX

wolfSSL has support for Marvell (previously Cavium) NITROX (https://www.marvell.com/products/security-solutions.html). To enable Marvell NITROX support when building wolfSSL use the following configure option:

```
./configure --with-cavium=/home/user/cavium/software
```

Where the --with-cavium=** option is pointing to your licensed cavium/software directory. Since Cavium doesn't build a library wolfSSL pulls in the cavium_common.o file which gives a libtool warning about the portability of this. Also, if you're using the github source tree you'll need to remove the -Wredundant-decls warning from the generated Makefile because the cavium headers don't conform to this warning.

Currently wolfSSL supports Cavium RNG, AES, 3DES, RC4, HMAC, and RSA directly at the crypto layer. Support at the SSL level is partial and currently just does AES, 3DES, and RC4. RSA and HMAC are slower until the Cavium calls can be utilized in non-blocking mode. The example client turns on cavium support as does the crypto test and benchmark. Please see the HAVE_CAVIUM define.

4.4.4 ESP32-WROOM-32

wolfSSL is able to use the ESP32-WROOM-32 hardware-based cryptography.

For necessary defines, see the WOLFSSL_ESPWROOM32 define in settings.h. The WOLFSSL_ESPWROOM32 define enables ESP32-WROOM-32 hardware crypto and RNG support by default. Currently wolfSSL supports RNG, AES, SHA and RSA primitive at the crypt layer. The example projects including TLS server/client, wolfCrypt test and benchmark can be found at /examples/protocols directory in ESP-IDF after deploying files.

4.4.5 ESP8266

Unlike the ESP32, there's no hardware-based cryptography available for the ESP8266. See the WOLFSSL_ESP8266 define in user_settings.h or use ./configure CFLAGS="-DWOLFSSL_ESP8266" to compile for the embedded ESP8266 target.

4.4.6 ERF32

wolfSSL is able to use the ERF32 family of devices for hardware-based cryptography.

To enable support define WOLFSSL_SILABS_SE_ACCEL in user_settings.h. wolfSSL currently supports the hardware acceleration of RNG, AES-CBC, AES-GCM, AES-CCM, SHA-1, SHA-2, ECDHE, and ECDSA on the EFR32 platform.

More details and benchmarks are available in the README.md in wolfcrypt/src/port/silabs of the wolfSSL repository tree.

4.4.7 MAX32665/MAX32666

wolfSSL supports using the Trust Protection Unit (TPU), Modular Arithmetic Accelerator (MAA) and TRNG found on supported models of the MAX32666/MAX32665 microcontrollers from Analog Devices.

To enable support define WOLFSSL_MAX3266X and WOLFSSL_SP_MATH_ALL. wolfSSL currently supports the hardware acceleration of RNG, AES-CBC, AES-GCM, AES-ECB, SHA-1, SHA-2, RSA 2048, and ECDSA.

This HW also supports the use of wolfSSL's crypto callback feature to allow the usage of both HW and SW implentations.

More details of the support can be found in the README.md at wolfcrypt/src/port/maxim of the wolfSSL repository tree.

4.5 SSL Inspection (Sniffer)

Beginning with the wolfSSL 1.5.0 release, wolfSSL has included a build option allowing it to be built with SSL Sniffer (SSL Inspection) functionality. This means that you can collect SSL traffic packets and

with the correct key file, are able to decrypt them as well. The ability to "inspect" SSL traffic can be useful for several reasons, some of which include:

- Analyzing Network Problems
- Detecting network misuse by internal and external users
- Monitoring network usage and data in motion
- Debugging client/server communications

To enable sniffer support, build wolfSSL with the `--enable-sniffer` option on *nix or use the **vcproj** files on Windows. You will need to have **pcap** installed on *nix or **WinPcap** on Windows. The main sniffer functions which can be found in `sniffer.h` are listed below with a short description of each:

- `ssl_SetPrivateKey` - Sets the private key for a specific server and port.
- `ssl_SetNamedPrivateKey` - Sets the private key for a specific server, port and domain name.
- `ssl_DecodePacket` - Passes in a TCP/IP packet for decoding.
- `ssl_Trace` - Enables / Disables debug tracing to the traceFile.
- `ssl_InitSniffer` - Initialize the overall sniffer.
- `ssl_FreeSniffer` - Free the overall sniffer.
- `ssl_EnableRecovery` - Enables option to attempt to pick up decoding of SSL traffic in the case of lost packets.
- `ssl_GetSessionStats` - Obtains memory usage for the sniffer sessions.

To look at wolfSSL's sniffer support and see a complete example, please see the `snifftest` app in the `sslSniffer/sslSnifferTest` folder from the wolfSSL download.

Keep in mind that because the encryption keys are setup in the SSL Handshake, the handshake needs to be decoded by the sniffer in order for future application data to be decoded. For example, if you are using "snifftest" with the wolfSSL example echoserver and echoclient, the snifftest application must be started before the handshake begins between the server and client.

The sniffer can only decode streams encrypted with the following algorithms: AES-CBC, DES3-CBC, ARC4 and Camellia-CBC. If ECDHE or DHE key agreement is used the stream cannot be sniffed; only RSA or ECDH key-exchange is supported.

Watch callbacks with wolfSSL sniffer can be turned on with `WOLFSSL_SNIFFER_WATCH`. With the sniffer watch feature compiled in, the function `ssl_SetWatchKeyCallback()` can be used to set a custom callback. The callback is then used to inspect the certificate chain, error value, and digest of the certificate sent from the peer. If a non 0 value is returned from the callback then an error state is set when processing the peer's certificate. Additional supporting functions for the watch callbacks are:

- `ssl_SetWatchKeyCtx`: Sets a custom user context that gets passed to the watch callback.
- `ssl_SetWatchKey_buffer`: Loads a new DER format key into server session.
- `ssl_SetWatchKey_file`: File version of `ssl_SetWatchKey_buffer`.

Statistics collecting with the sniffer can be compiled in with defining the macro `WOLFSSL_SNIFFER_STATS`. The statistics are kept in a SSLStats structure and are copied to an applications SSLStats structure by a call to `ssl_ReadStatistics`. Additional API to use with sniffer statistics is `ssl_ResetStatistics` (resets the collection of statistics) and `ssl_ReadResetStatistics` (reads the current statistic values and then resets the internal state). The following is the current statistics kept when turned on:

- `sslStandardConns`
- `sslClientAuthConns`
- `sslResumedConns`
- `sslEphemeralMisses`
- `sslResumeMisses`
- `sslCiphersUnsupported`
- `sslKeysUnmatched`
- `sslKeyFails`
- `sslDecodeFails`

- sslAlerts
- sslDecryptedBytes
- sslEncryptedBytes
- sslEncryptedPackets
- sslDecryptedPackets
- sslKeyMatches
- sslEncryptedConns

4.6 Static Buffer Allocation Option

By default, wolfSSL assumes that the execution environment provides dynamic memory allocation, i.e., buffers can be allocated/freed with the malloc/free functions. The wolfCrypt cryptography library, which wolfSSL uses internally for underlying cryptography operations, can alternatively be configured to not use dynamic memory. This can be helpful for environments without dynamic memory support, or safety-critical applications where dynamic memory use is disallowed.

4.6.1 Basic Operation of Static Buffer Allocation

"Dynamic memory allocation" is a management method that dynamically finds/allocates and provides a buffer of a "specified size (variable length)". Buffer usage efficiency is high, but processing is relatively complicated. On the other hand, the "static buffer allocation" provided by wolfSSL is a memory management model that searches for a buffer close to the requested size from among several types of buffers prepared in advance (statically) and provides it back to the caller. A memory block larger than the requested size may be allocated and returned to the requester of the buffer (thus reducing the efficiency of use). Although not as precise in memory management, it is simple and simulates dynamic memory allocation that dynamically allocates a memory block of any size.

Using static-buffer-allocation is equivalent in API to using dynamic memory with wolfSSL. This functional equivalency is achieved in wolfSSL by abstracting memory allocation/free into XMALLOC/XFREE function calls. Once static-buffer-allocation is set, wolfSSL will use it from then on to allocate buffers and other structures used internally. Since this feature is set for WOLFSSL_CTX, it will continue to work for the lifetime of the context object.

The static-buffer-allocation set in a WOLFSSL_CTX is thread-safe. Even if the same WOLFSSL_CTX is shared by different threads, buffer allocation/free is used under exclusive control inside wolfSSL. The following is a visual representation of the CTX structure, the arrows indicate passing a pointer to the heap and "..." references all structs and not to the ones listed.

In comparison to a memory pool functionality offered by an RTOS implementation, memory functionality in an RTOS will commonly suspend a thread (task) if an unused memory block cannot be found when requested until a free block becomes available. wolfSSL's static memory functionality has no such synchronization capability.

4.6.2 Specifying Static Buffer Use

With static-buffer-allocation in wolfSSL, it is possible to divide memory between two purposes. It's possible to allocate and free buffers separately between general purposes and I/O use cases. The buffer used for I/O is set relatively large (about 17KB) to accommodate the maximum TLS fragment size of up to 2^16 bytes. This is different from the buffer sizes for other general uses. Additionally, when setting the buffer configuration the user can limit the maximum number of WOLFSSL objects that can be created simultaneously. If the maximum number of WOLFSSL sessions is limited, each use of the wolfSSL_new() function will check the number of WOLFSSL objects that can be created and will error out if the limit is exceeded.

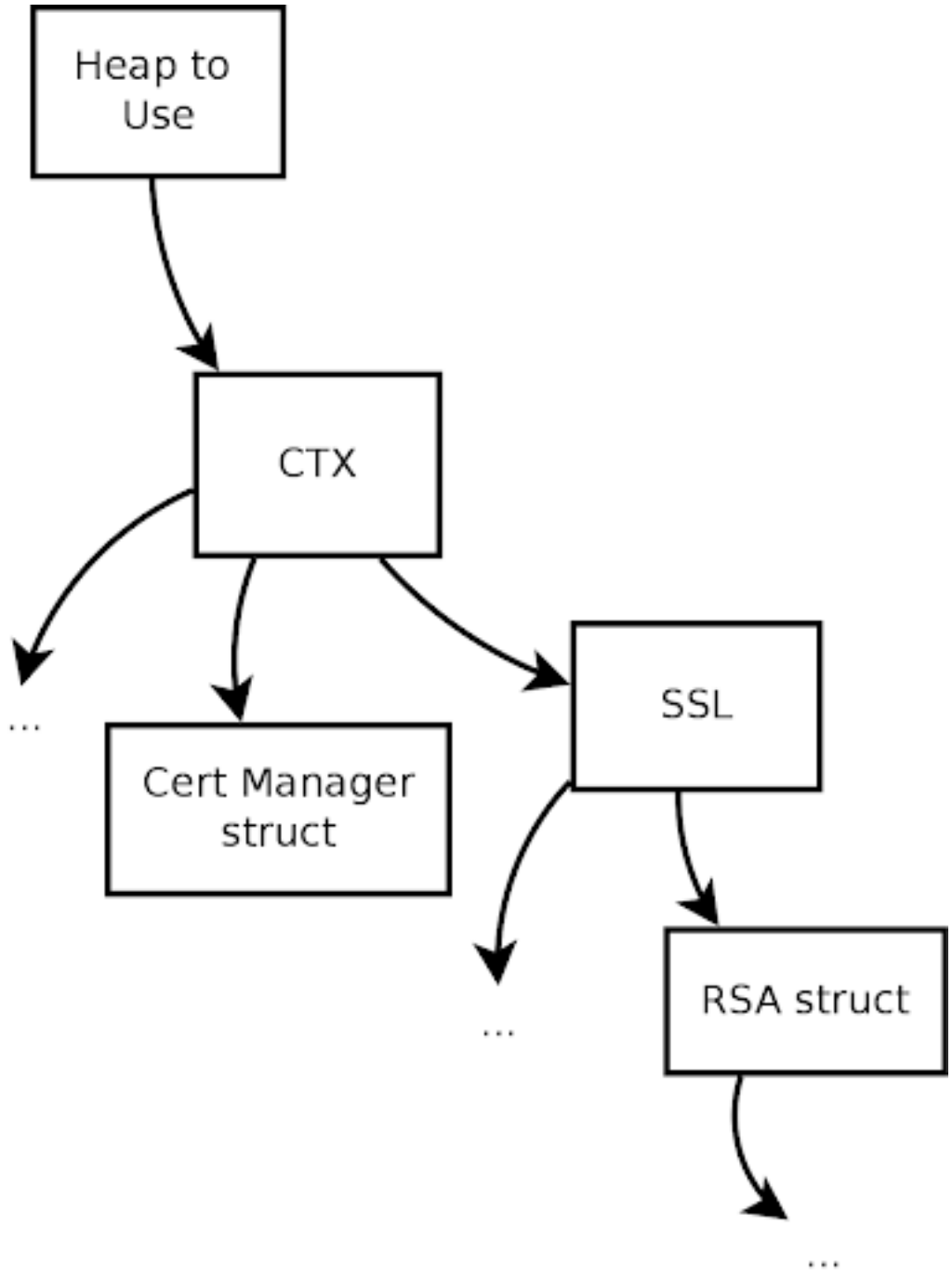

Figure 2: Alt text

4.6.3 Enabling Static Buffer Allocation

Enable the static-buffer-allocation option when building wolfSSL. For systems built using Autoconf, specify "–enable-staticmemory" as follows:

```
$ ./configure --enable-staticmemory
```

Or if you are using a user_settings.h header, add the following macro definition:

user_settings.h

```
#define WOLFSSL_STATIC_MEMORY
```

The static-buffer-allocation option is implemented by default to fall back to the standard malloc() function when a NULL heap hint is passed in. If a heap hint is passed in and the memory associated with it is exhausted, an error will occur. If the environment does not provide dynamic memory management functionality, a link error will occur. Therefore, also define the **WOLFSSL_NO_MALLOC** macro to disable this feature if needed:

user_settings.h

```
#define WOLFSSL_STATIC_MEMORY
    #define WOLFSSL_NO_MALLOC
```

In addition there are two build configurations. `--enable-staticmemory=small` which is a smaller version that has smaller struct sizes and less supporting API's available. The other build configuration is `--enable-staticmemory=debug` that enables the ability to set a callback function. This is useful in cases where printf() is not available for determining what is being allocated and what bucket sizes are being used. Here is what the example client output looks like with example callback:

```
./examples/client/client
...
...
...
Free'ing : 16128
OUT BUFFER: Alloc'd 6 bytes using bucket size 16992
Alloc'd 848 bytes using bucket size 1024
OUT BUFFER: Alloc'd 150 bytes using bucket size 16992
Alloc'd 13 bytes using bucket size 64
Alloc'd 12 bytes using bucket size 64
Alloc'd 848 bytes using bucket size 1024
IN BUFFER: Alloc'd 40 bytes using bucket size 16992
Alloc'd 13 bytes using bucket size 64
Alloc'd 12 bytes using bucket size 64
Free'ing : 1024
Free'ing : 512

...
...
...
```

4.6.4 Using Static Buffer Allocation

This can be helpful for environments without dynamic memory support, or safety-critical applications where dynamic memory use is disallowed.

4.6.4.1 Static buffer setup function and its arguments This can be helpful for environments without dynamic memory support, or safety-critical applications where dynamic memory use is disallowed.

```
int wolfSSL_CTX_load_static_memory(
      WOLFSSL_CTX** ctx, /* address of the variable to hold WOLFSSL_CTX */
      wolfSSL_method_func method,/* method pointer */
      unsigned char* buf,   /* pointer to the buffer to use as heap */
      unsigned int sz,      /* buffer size */
      int flag,              /* heap usage */
      int max);              /* maximum number of objects allowed */
```

- parameter **ctx** specifies the address of a variable that receives a pointer to the generated WOLFSSL_CTX structure.
- parameter **method** specifies a function pointer with "_ex", such as wolfSSLv23_client_method_ex(). The functions that can be used are listed in a later chapter.
- parameter **buf** and **sz** specify the address and size of the buffer used for the heap, respectively. For information on determining the buffer size to be set, see "**Obtaining the Required Buffer Size**".
- parameter **flag** is a flag that specifies the usage of the buffer. You can also specify whether to track the allocation status. When specifying for general use, specify "**0**" or **WOLFMEM_GENERAL**. For I/O use, specify **WOLFMEM_IO_POOL** or **WOLFMEM_IO_POOL_FIXED**. When tracking the allocation status of static buffers, **OR** the value specifying the usage with **WOLFMEM_TRACK_STATS**.
- parameter **max** is related to the use of the buffer specified by the argument flag. If the buffer is for general use, you may want to set the maximum number of WOLFSSL objects that can be generated simultaneously (the number of objects that can exist at the same time). Specify **0** if there is **no need to limit**. If you specify a limit value other than 0, subsequent calls to wolfSSL_new() will fail if the number of concurrent WOLFSSL objects created exceeds the set value.

4.6.4.2 How to call the static buffer setup function When using the static-buffer-allocation option, call the wolfSSL_CTX_load_static_memory() function **twice**. The first sets up a buffer for general use, and then uses that buffer to allocate a WOLFSSL_CTX structure. The second call sets up the I/O buffer:

```
WOLFSSL_CTX* ctx = NULL; /* pass NULL to generate WOLFSSL_CTX */
int ret;

#define MAX_CONCURRENT_TLS  0
#define MAX_CONCURRENT_IO   0

unsigned char GEN_MEM[GEN_MEM_SIZE];
unsigned char IO_MEM[IO_MEM_SIZE  ];

  /* set up a general-purpose buffer and generate WOLFSSL_CTX from it on the
     first call. */
   ret = wolfSSL_CTX_load_static_memory(
          &ctx,                              /* set NULL to ctx */
          wolfSSLv23_client_method_ex(),  /* use function with "_ex" */
          GEN_MEM, GEN_MEM_SIZE,            /* buffer and its size */
          WOLFMEM_GENERAL,                  /* general purpose */
          MAX_CONCURRENT_TLS);              /* max concurrent objects */

   /* set up a I/O-purpose buffer on the second call. */
```

```
ret = wolfSSL_CTX_load_static_memory(
        &ctx,                 /* make sure ctx is holding the object */
        NULL,                             /* pass it to NULL this time */
        IO_MEM, IO_MEM_SIZE,                /* buffer and its size */
        WOLFMEM_IO_FIXED,                             /* I/O purpose */
        MAX_CONCURRENT_IO);               /* max concurrent objects */
```

After this, when you are done using the WOLFSSL_CTX structure, free it with the usual wolfSSL_CTX_free().

4.6.4.3 Setting a Global Heap Hint A global heap hint can be set using the API `void* wolfSSL_SetGlobalHeapHint(void* heap)`. When a global heap hint is set all calls to XMALLOC and XFREE using a NULL pointer as the heap hint will be redirected to use the global heap hint set. This is useful in cases where no system malloc is available to fall back to and NULL heap hint pointers are being used. The function wolfSSL_SetGlobalHeapHint returns the current global heap hint set, and is NOT considered to be thread safe.

The getter function `void* wolfSSL_GetGlobalHeapHint(void)` can be used to get the current global heap hint set.

This functionality was added in versions of wolfSSL after version 5.7.0.

4.6.5 Adjustment of Static Buffer Allocation

The static-buffer-allocation option provided by wolfSSL manages the specified buffer by dividing it into multiple areas called “buckets” as shown in the following diagram. Multiple memory blocks of the same size are linked within a bucket. The figure below omits the structure that manages the memory block, but a buffer with a size that includes the omitted structure is required.

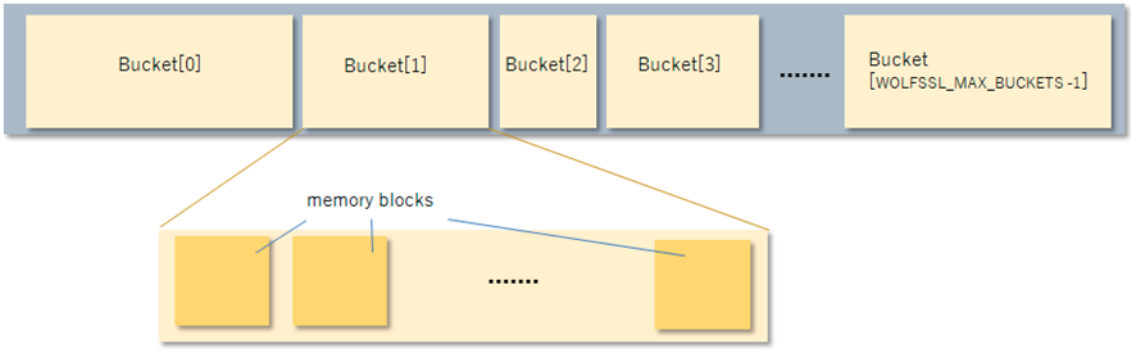

Figure 3: Alt text

4.6.5.1 Macros for General Use Buffers Each bucket varies in size depending on the number of memory blocks it contains and their size.

The memory block size and number of blocks for each area to be used are defined in /wolfssl/wolfcrypt/memory.h with the following macros:

`/wolfssl/wolfcrypt/memory.h`

```
  #define WOLFSSL_STATIC_ALIGN 16  /* alignment 16 bytes by default*/
  #define WOLFMEM_MAX_BUCKETS     9/* number of buckets */
#define WOLFMEM_IO_SZ         16992   /* buffer size for I/O  */
  #define LARGEST_MEM_BUCKET   16128   /* the max block size */
```

```
#define WOLFMEM_BUCKETS      64,128,256,512,1024,2432,3456,
                                                  4544,LARGEST_MEM_BUCKET
#define WOLFMEM_DIST         49,10,6,14,5,6,9,1,1
```

- **WOLFSSL_STATIC_ALIGN** specifies the buffer alignment size. 16 bytes by default. You need to change it according to the alignment size of your MCU.
- **WOLFMEM_MAX_BUCKETS** shows the number of buckets. This means using 9 different bucket sizes.
- **WOLFMEM_BUCKETS** specifies the number of bytes in blocks in each bucket, separated by commas, from smallest to largest. This definition applies to general purpose buffers.
- **WOLFMEM_DIST** specifies the number of same-sized blocks in each bucket, separated by commas, corresponding to each block in WOLFMEM_BUCKETS. This definition applies to general purpose buffers.

In the example above, a bucket with a block size of 64 bytes is the minimum size, and that bucket would have 49 memory blocks. The next larger bucket means 10 memory blocks with a block size of 128 bytes. The above defined values can be used as default values, but the size of each bucket and the number of memory blocks it contains may need to be adjusted when used in an actual environment.

4.6.5.2 Macros for I/O Use Buffers For TLS client

- `wolfTLSv1_3_client_method_ex`
- `wolfTLSv1_2_client_method_ex`
- `wolfTLSv1_1_client_method_ex`
- `wolfSSLv23_client_method_ex`

For TLS server - `wolfTLSv1_3_server_method_ex` - `wolfTLSv1_2_server_method_ex` - `wolfTLSv1_1_server_method_ex` - `wolfSSLv23_server_method_ex`

For DTLS client

- `wolfDTLSv1_3_client_method_ex`
- `wolfTLSv1_2_client_method_ex`
- `wolfTLSv1_1_client_method_ex`
- `wolfSSLv23_client_method_ex`

For DTLS server

- `wolfDTLSv1_3_server_method_ex`
- `wolfTLSv1_2_server_method_ex`
- `wolfTLSv1_1_server_method_ex`
- `wolfSSLv23_server_method_ex`

4.6.5.3 APIs for Static Buffer Allocation

API	description
wolfSSL_CTX_load_static_memory	Set buffer for WOLFSSL_CTX as a heap memory.
wolfSSL_CTX_is_static_memory	Returns whether “Static buffer Allocation” is used. If it is the case, gets usage report.
wolfSSL_is_static_memory	Returns whether “Static buffer Allocation” is used. If it is the case, gets usage report.
wc_LoadStaticMemory	Used to set aside static memory for wolfCrypt use.

API	description
wolfSSL_StaticBufferSz	Calculate required buffer size for "Static buffer Allocation" based on the macros defined in /wolfssl/wolfcrypt/memory.h.

4.7 Compression

wolfSSL supports data compression with the **zlib** library. The ./configure build system detects the presence of this library, but if you're building in some other way define the constant HAVE_LIBZ and include the path to zlib.h for your includes.

Compression is off by default for a given cipher. To turn it on, use the function wolfSSL_set_compression() before SSL connecting or accepting. Both the client and server must have compression turned on in order for compression to be used.

Keep in mind that while compressing data before sending decreases the actual size of the messages being sent and received, the amount of data saved by compression usually takes longer time to analyze than it does to send it raw on all but the slowest of networks.

4.8 Pre-Shared Keys

wolfSSL has support for these ciphers with static pre-shared keys:

- TLS_PSK_WITH_AES_256_CBC_SHA
- TLS_PSK_WITH_AES_128_CBC_SHA256
- TLS_PSK_WITH_AES_256_CBC_SHA384
- TLS_PSK_WITH_AES_128_CBC_SHA
- TLS_PSK_WITH_NULL_SHA256
- TLS_PSK_WITH_NULL_SHA384
- TLS_PSK_WITH_NULL_SHA
- TLS_PSK_WITH_AES_128_GCM_SHA256
- TLS_PSK_WITH_AES_256_GCM_SHA384
- TLS_PSK_WITH_AES_128_CCM
- TLS_PSK_WITH_AES_256_CCM
- TLS_PSK_WITH_AES_128_CCM_8
- TLS_PSK_WITH_AES_256_CCM_8
- TLS_PSK_WITH_CHACHA20_POLY1305

These suites are built into wolfSSL with WOLFSSL_STATIC_PSK on, all PSK suites can be turned off at build time with the constant NO_PSK. To only use these ciphers at runtime use the function wolfSSL_CTX_set_cipher_list() with the desired ciphersuite.

wolfSSL has support for ephemeral key PSK suites:

- ECDHE-PSK-AES128-CBC-SHA256
- ECDHE-PSK-NULL-SHA256
- ECDHE-PSK-CHACHA20-POLY1305
- DHE-PSK-CHACHA20-POLY1305
- DHE-PSK-AES256-GCM-SHA384
- DHE-PSK-AES128-GCM-SHA256
- DHE-PSK-AES256-CBC-SHA384
- DHE-PSK-AES128-CBC-SHA256
- DHE-PSK-AES128-CBC-SHA256

On the client, use the function wolfSSL_CTX_set_psk_client_callback() to setup the callback. The client example in <wolfSSL_Home>/examples/client/client.c gives example usage for setting up the client identity and key, though the actual callback is implemented in wolfssl/test.h.

On the server side two additional calls are required:

- wolfSSL_CTX_set_psk_server_callback()
- wolfSSL_CTX_use_psk_identity_hint()

The server stores its identity hint to help the client with the 2nd call, in our server example that's "wolfssl server". An example server psk callback can also be found in my_psk_server_cb() in wolfssl/test.h.

wolfSSL supports identities and hints up to 128 octets and pre-shared keys up to 64 octets.

4.9 Client Authentication

Client authentication is a feature which enables the server to authenticate clients by requesting that the clients send a certificate to the server for authentication when they connect. Client authentication requires an X.509 client certificate from a CA (or self-signed if generated by you or someone other than a CA).

By default, wolfSSL validates all certificates that it receives - this includes both client and server. To set up client authentication, the server must load the list of trusted CA certificates to be used to verify the client certificate against:

```
wolfSSL_CTX_load_verify_locations(ctx, caCert, 0);
```

To turn on client verification and control its behavior, the wolfSSL_CTX_set_verify() include SSL_VERIFY_NONE and SSL_VERIFY_CLIENT_ONCE.

```
wolfSSL_CTX_set_verify(ctx,SSL_VERIFY_PEER | ((usePskPlus)?
                       SSL_VERIFY_FAIL_EXCEPT_PSK :
                       SSL_VERIFY_FAIL_IF_NO_PEER_CERT),0);
```

An example of client authentication can be found in the example server (server.c) included in the wolfSSL download (/examples/server/server.c).

4.10 Server Name Indication

SNI is useful when a server hosts multiple 'virtual' servers at a single underlying network address. It may be desirable for clients to provide the name of the server which it is contacting. To enable SNI with wolfSSL you can simply do:

```
./configure --enable-sni
```

Using SNI on the client side requires an additional function call, which should be one of the following functions:

- wolfSSL_CTX_UseSNI()
- wolfSSL_UseSNI()

wolfSSL_CTX_UseSNI() is most recommended when the client contacts the same server multiple times. Setting the SNI extension at the context level will enable the SNI usage in all SSL objects created from that same context from the moment of the call forward.

wolfSSL_UseSNI() will enable SNI usage for one SSL object only, so it is recommended to use this function when the server name changes between sessions.

On the server side one of the same function calls is required. Since the wolfSSL server doesn't host multiple 'virtual' servers, the SNI usage is useful when the termination of the connection is desired

in the case of SNI mismatch. In this scenario, wolfSSL_CTX_UseSNI() will be more efficient, as the server will set it only once per context creating all subsequent SSL objects with SNI from that same context.

4.11 Handshake Modifications

4.11.1 Grouping Handshake Messages

wolfSSL has the ability to group handshake messages if the user desires. This can be done at the context level with wolfSSL_CTX_set_group_messages(ctx);.

4.12 Truncated HMAC

Currently defined TLS cipher suites use the HMAC to authenticate record-layer communications. In TLS, the entire output of the hash function is used as the MAC tag. However, it may be desirable in constrained environments to save bandwidth by truncating the output of the hash function to 80 bits when forming MAC tags. To enable the usage of Truncated HMAC at wolfSSL you can simply do:

```
./configure --enable-truncatedhmac
```

Using Truncated HMAC on the client side requires an additional function call, which should be one of the following functions:

- wolfSSL_CTX_UseTruncatedHMAC()
- wolfSSL_UseTruncatedHMAC()

wolfSSL_CTX_UseTruncatedHMAC() is most recommended when the client would like to enable Truncated HMAC for all sessions. Setting the Truncated HMAC extension at context level will enable it in all SSL objects created from that same context from the moment of the call forward.

wolfSSL_UseTruncatedHMAC() will enable it for one SSL object only, so it's recommended to use this function when there is no need for Truncated HMAC on all sessions.

On the server side no call is required. The server will automatically attend to the client's request for Truncated HMAC.

All TLS extensions can also be enabled with:

```
./configure --enable-tlsx
```

4.13 User Crypto Module

User Crypto Module allows for a user to plug in custom crypto that they want used during supported operations (Currently RSA operations are supported). An example of a module is located in the directory root_wolfssl/wolfcrypt/user-crypto/ using IPP libraries. Examples of the configure option when building wolfSSL to use a crypto module is as follows:

```
./configure --with-user-crypto
```

or

```
./configure --with-user-crypto=/dir/to
```

When creating a user crypto module that performs RSA operations, it is mandatory that there is a header file for RSA called user_rsa.h. For all user crypto operations it is mandatory that the users library be called libusercrypto. These are the names that wolfSSL autoconf tools will be looking for when linking and using a user crypto module. In the example provided with wolfSSL, the header file user_rsa.h can be found in the directory wolfcrypt/user-crypto/include/ and the library once created is located in the directory wolfcrypt/user-crypto/lib/. For a list of required API look at the header file provided.

To build the example, after having installed IPP libraries, the following commands from the root wolfSSL directory should be ran.

```
cd wolfcrypt/user-crypto/
./autogen.sh
./configure
make
sudo make install
```

The included example in wolfSSL requires the use of IPP, which will need to be installed before the project can be built. Though even if not having IPP libraries to build the example it is intended to provide users with an example of file name choice and API interface. Once having made and installed both the library libusercrypto and header files, making wolfSSL use the crypto module does not require any extra steps. Simply using the configure flag `--with-user-crypto` will map all function calls from the typical wolfSSL crypto to the user crypto module.

Memory allocations, if using wolfSSL's XMALLOC, should be tagged with `DYNAMIC_TYPE_USER_CRYPTO`. Allowing for analyzing memory allocations used by the module.

User crypto modules **cannot** be used in conjunction with the wolfSSL configure options fast-rsa and/or fips. Fips requires that specific, certified code be used and fast-rsa makes use of the example user crypto module to perform RSA operations.

4.14 Timing-Resistance in wolfSSL

wolfSSL provides the function "ConstantCompare" which guarantees constant time when doing comparison operations that could potentially leak timing information. This API is used at both the TLS and crypto level in wolfSSL to deter against timing based, side-channel attacks.

The wolfSSL ECC implementation has the define `ECC_TIMING_RESISTANT` to enable timing-resistance in the ECC algorithm. Similarly the define `TFM_TIMING_RESISTANT` is provided in the fast math libraries for RSA algorithm timing-resistance. The function exptmod uses the timing resistant Montgomery ladder.

See also: `--disable-harden`

Timing resistance and cache resistance defines enabled with `--enable-harden`:

- DEPRECATED: `WOLFSSL_SP_CACHE_RESISTANT`: Always on by default, see `--disable-harden` to disable default cache resistance.
- `WC_RSA_BLINDING`: Enables blinding mode, to prevent timing attacks.
- `ECC_TIMING_RESISTANT`: ECC specific timing resistance.
- `TFM_TIMING_RESISTANT`: Fast math specific timing resistance.

4.15 Fixed ABI

wolfSSL provides a fixed Application Binary Interface (ABI) for a subset of the Application Programming Interface (API). Starting with wolfSSL v4.3.0, the following functions will be compatible across all future releases of wolfSSL:

- `wolfSSL_Init()`
- `wolfTLSv1_2_client_method()`
- `wolfTLSv1_3_client_method()`
- `wolfSSL_CTX_new()`
- `wolfSSL_CTX_load_verify_locations()`
- `wolfSSL_new()`
- `wolfSSL_set_fd()`
- `wolfSSL_connect()`

- wolfSSL_read()
- wolfSSL_write()
- wolfSSL_get_error()
- wolfSSL_shutdown()
- wolfSSL_free()
- wolfSSL_CTX_free()
- wolfSSL_check_domain_name()
- wolfSSL_UseALPN()
- wolfSSL_CTX_SetMinVersion()
- wolfSSL_pending()
- wolfSSL_set_timeout()
- wolfSSL_CTX_set_timeout()
- wolfSSL_get_session()
- wolfSSL_set_session()
- wolfSSL_flush_sessions()
- wolfSSL_CTX_set_session_cache_mode()
- wolfSSL_get_sessionID()
- wolfSSL_UseSNI()
- wolfSSL_CTX_UseSNI()
- wc_ecc_init_ex()
- wc_ecc_make_key_ex()
- wc_ecc_sign_hash()
- wc_ecc_free()
- wolfSSL_SetDevId()
- wolfSSL_CTX_SetDevId()
- wolfSSL_CTX_SetEccSignCb()
- wolfSSL_CTX_use_certificate_chain_file()
- wolfSSL_CTX_use_certificate_file()
- wolfSSL_use_certificate_chain_file()
- wolfSSL_use_certificate_file()
- wolfSSL_CTX_use_PrivateKey_file()
- wolfSSL_use_PrivateKey_file()
- wolfSSL_X509_load_certificate_file()
- wolfSSL_get_peer_certificate()
- wolfSSL_X509_NAME_oneline()
- wolfSSL_X509_get_issuer_name()
- wolfSSL_X509_get_subject_name()
- wolfSSL_X509_get_next_altname()
- wolfSSL_X509_notBefore()
- wolfSSL_X509_notAfter()
- wc_ecc_key_new()
- wc_ecc_key_free()

5 Portability

5.1 Abstraction Layers

5.1.1 C Standard Library Abstraction Layer

wolfSSL (formerly CyaSSL) can be built without the C standard library to provide a higher level of portability and flexibility to developers. The user will have to map the functions they wish to use instead of the C standard ones.

5.1.1.1 Memory Use Most C programs use `malloc()` and `free()` for dynamic memory allocation. wolfSSL uses `XMALLOC()` and `XFREE()` instead. By default, these point to the C runtime versions. By defining `XMALLOC_USER`, the user can provide their own hooks. Each memory function takes two additional arguments over the standard ones, a heap hint, and an allocation type. The user is free to ignore these or use them in any way they like. You can find the wolfSSL memory functions in `wolfssl/wolfcrypt/types.h`.

wolfSSL also provides the ability to register memory override functions at runtime instead of compile time. `wolfssl/wolfcrypt/memory.h` is the header for this functionality and the user can call the following function to set up the memory functions:

```
int wolfSSL_SetAllocators(wolfSSL_Malloc_cb  malloc_function,
                          wolfSSL_Free_cb    free_function,
                          wolfSSL_Realloc_cb realloc_function);
```

See the header `wolfssl/wolfcrypt/memory.h` for the callback prototypes and `memory.c` for the implementation.

5.1.1.2 string.h wolfSSL uses several functions that behave like `string.h`'s `memcpy()`, `memset()`, and `memcmp()` amongst others. They are abstracted to `XMEMCPY()`, `XMEMSET()`, and `XMEMCMP()` respectively. And by default, they point to the C standard library versions. Defining `STRING_USER` allows the user to provide their own hooks in `types.h`. For example, by default `XMEMCPY()` is:

```
#define XMEMCPY(d,s,l)    memcpy((d),(s),(l))
```

After defining `STRING_USER` you could do:

```
#define XMEMCPY(d,s,l)    my_memcpy((d),(s),(l))
```

Or if you prefer to avoid macros:

```
external void* my_memcpy(void* d, const void* s, size_t n);
```

to set wolfSSL's abstraction layer to point to your version `my_memcpy()`.

5.1.1.3 math.h wolfSSL uses two functions that behave like `math.h`'s `pow()` `log()`. They are only required by Diffie-Hellman, so if you exclude DH from the build, then you don't have to provide your own. They are abstracted to `XPOW()` and `XLOG()` and found in `wolfcrypt/src/dh.c`.

5.1.1.4 File System Use By default, wolfSSL uses the system's file system for the purpose of loading keys and certificates. This can be turned off by defining `NO_FILESYSTEM`, see item V. If instead, you'd like to use a file system but not the system one, you can use the `XFILE()` layer in `ssl.c` to point the file system calls to the ones you'd like to use. See the example provided by the MICRIUM define.

5.1.2 Custom Input/Output Abstraction Layer

wolfSSL provides a custom I/O abstraction layer for those who wish to have higher control over I/O of their SSL connection or run SSL on top of a different transport medium other than TCP/IP.

The user will need to define two functions:

1. The network Send function
2. The network Receive function

These two functions are prototyped by `CallbackIOSend` and `CallbackIORecv` in `ssl.h`:

```
typedef int (*CallbackIORecv)(WOLFSSL *ssl, char *buf, int sz, void *ctx);
typedef int (*CallbackIOSend)(WOLFSSL *ssl, char *buf, int sz, void *ctx);
```

The user needs to register these functions per `WOLFSSL_CTX` with `wolfSSL_SetIOSend()` and `wolfSSL_SetIORecv()`. For example, in the default case, `CBIORecv()` and `CBIOSend()` are registered at the bottom of `io.c`:

```
void wolfSSL_SetIORecv(WOLFSSL_CTX *ctx, CallbackIORecv CBIORecv)
{
     ctx->CBIORecv = CBIORecv;
}

void wolfSSL_SetIOSend(WOLFSSL_CTX *ctx, CallbackIOSend CBIOSend)
{
    ctx->CBIOSend = CBIOSend;
}
```

The user can set a context per WOLFSSL object (session) with `wolfSSL_SetIOWriteCtx()` and `wolfSSL_SetIOReadCtx()`, as demonstrated at the bottom of `io.c`. For example, if the user is using memory buffers, the context may be a pointer to a structure describing where and how to access the memory buffers. The default case, with no user overrides, registers the socket as the context.

The `CBIORecv` and `CBIOSend` function pointers can be pointed to your custom I/O functions. The default `Send()` and `Receive()` functions, `EmbedSend()` and `EmbedReceive()`, located in `io.c`, can be used as templates and guides.

`WOLFSSL_USER_IO` can be defined to remove the automatic setting of the default I/O functions `EmbedSend()` and `EmbedReceive()`.

5.1.3 Operating System Abstraction Layer

The wolfSSL OS abstraction layer helps facilitate easier porting of wolfSSL to a user's operating system. The `wolfssl/wolfcrypt/settings.h` file contains settings which end up triggering the OS layer.

OS-specific defines are located in `wolfssl/wolfcrypt/types.h` for wolfCrypt and `wolfssl/internal.h` for wolfSSL.

5.2 Supported Operating Systems

One factor which defines wolfSSL is its ability to be easily ported to new platforms. As such, wolfSSL has support for a long list of operating systems out-of-the-box. Currently-supported operating systems include:

- Win32/64
- Linux
- Mac OS X
- Solaris
- ThreadX

- VxWorks
- FreeBSD
- NetBSD
- OpenBSD
- embedded Linux
- Yocto Linux
- OpenEmbedded
- WinCE
- Haiku
- OpenWRT
- iPhone (iOS)
- Android
- Nintendo Wii and Gamecube through DevKitPro
- QNX
- MontaVista
- NonStop
- TRON/ITRON/µITRON
- Micrium's µC/OS-III
- FreeRTOS
- SafeRTOS
- NXP/Freescale MQX
- Nucleus
- TinyOS
- HP/UX
- AIX
- ARC MQX
- TI-RTOS
- uTasker
- embOS
- INtime
- Mbed
- µT-Kernel
- RIOT
- CMSIS-RTOS
- FROSTED
- Green Hills INTEGRITY
- Keil RTX
- TOPPERS
- PetaLinux
- Apache Mynewt

5.3 Supported Chipmakers

wolfSSL has support for chipsets including ARM, Intel, Motorola, mbed, Freescale, Microchip (PIC32), STMicro (STM32F2/F4), NXP, Analog Devices, Texas Instruments, AMD and more.

5.4 C# Wrapper

wolfSSL has limited support for use in C#. A Visual Studio project containing the port can be found in the directory `root_wolfSSL/wrapper/CSharp/`. After opening the Visual Studio project set the "Active solution configuration" and "Active solution platform" by clicking on BUILD->Configuration Manager... The supported "Active solution configuration"s are DLL Debug and DLL Release. The supported platforms are Win32 and x64.

Once having set the solution and platform the preprocessor flag HAVE_CSHARP will need to be added. This turns on the options used by the C# wrapper and used by the examples included.

To then build simply select build solution. This creates the `wolfssl.dll`, `wolfSSL_CSharp.dll` and examples. Examples can be ran by targeting them as an entry point and then running debug in Visual Studio.

Adding the created C# wrapper to C# projects can be done a couple of ways. One way is to install the created `wolfssl.dll` and `wolfSSL_CSharp.dll` into the directory `C:/Windows/System/`. This will allow projects that have:

```
using wolfSSL.CSharp

public some_class {

    public static main(){
    wolfssl.Init()
    ...
}
...
```

to make calls to the wolfSSL C# wrapper. Another way is to create a Visual Studio project and have it reference the bundled C# wrapper solution in wolfSSL.

6 Callbacks

6.1 HandShake Callback

wolfSSL (formerly CyaSSL) has an extension that allows a HandShake Callback to be set for connect or accept. This can be useful in embedded systems for debugging support when another debugger isn't available and sniffing is impractical. To use wolfSSL HandShake Callbacks, use the extended functions, `wolfSSL_connect_ex()` and `wolfSSL_accept_ex()`:

```
int wolfSSL_connect_ex(WOLFSSL*, HandShakeCallBack, TimeoutCallBack,
                       Timeval)
int wolfSSL_accept_ex(WOLFSSL*, HandShakeCallBack, TimeoutCallBack,
                      Timeval)
```

HandShakeCallBack is defined as:

```
typedef int (*HandShakeCallBack)(HandShakeInfo*);
```

HandShakeInfo is defined in `wolfssl/callbacks.h` (which should be added to a non-standard build):

```
typedef struct handShakeInfo_st {
    char   cipherName[MAX_CIPHERNAME_SZ + 1];  /*negotiated name */
    char   packetNames[MAX_PACKETS_HANDSHAKE][MAX_PACKETNAME_SZ+1];
                                          /* SSL packet names */
    int    numberPackets;                 /*actual # of packets  */
    int    negotiationError;              /*cipher/parameter err */
} HandShakeInfo;
```

No dynamic memory is used since the maximum number of SSL packets in a handshake exchange is known. Packet names can be accessed through `packetNames[idx]` up to `numberPackets`. The callback will be called whether or not a handshake error occurred. Example usage is also in the client example.

6.2 Timeout Callback

The same extensions used with wolfSSL Handshake Callbacks can be used for wolfSSL Timeout Callbacks as well. These extensions can be called with either, both, or neither callbacks (Handshake and/or Timeout). `TimeoutCallback` is defined as:

```
typedef int (*TimeoutCallBack)(TimeoutInfo*);
```

Where `TimeoutInfo` looks like:

```
typedef struct timeoutInfo_st {
    char       timeoutName[MAX_TIMEOUT_NAME_SZ +1]; /*timeout Name*/
    int        flags;                               /* for future use*/
    int        numberPackets;                /*actual # of packets */
    PacketInfo packets[MAX_PACKETS_HANDSHAKE];  /*list of packets */
    Timeval    timeoutValue;                /*timer that caused it */
} TimeoutInfo;
```

Again, no dynamic memory is used for this structure since a maximum number of SSL packets is known for a handshake. `Timeval` is just a typedef for struct timeval.

`PacketInfo` is defined like this:

```
typedef struct packetInfo_st {
    char           packetName[MAX_PACKETNAME_SZ + 1]; /* SSL name */
    Timeval        timestamp;                  /* when it occurred */
```

```
    unsigned char  value[MAX_VALUE_SZ];     /* if fits, it's here */
    unsigned char* bufferValue;         /* otherwise here (non 0) */
    int            valueSz;              /* sz of value or buffer */
} PacketInfo;
```

Here, dynamic memory may be used. If the SSL packet can fit in `value` then that's where it's placed. `valueSz` holds the length and `bufferValue` is 0. If the packet is too big for `value`, only **Certificate** packets should cause this, then the packet is placed in `bufferValue`. `valueSz` still holds the size.

If memory is allocated for a **Certificate** packet then it is reclaimed after the callback returns. The timeout is implemented using signals, specifically `SIGALRM`, and is thread safe. If a previous alarm is set of type `ITIMER_REAL` then it is reset, along with the correct handler, afterwards. The old timer will be time adjusted for any time wolfSSL spends processing. If an existing timer is shorter than the passed timer, the existing timer value is used. It is still reset afterwards. An existing timer that expires will be reset if has an interval associated with it. The callback will only be issued if a timeout occurs.

See the client example for usage.

6.3 User Atomic Record Layer Processing

wolfSSL provides Atomic Record Processing callbacks for users who wish to have more control over MAC/encrypt and decrypt/verify functionality during the SSL/TLS connection.

The user will need to define 2 functions:

1. MAC/encrypt callback function
2. Decrypt/verify callback function

These two functions are prototyped by `CallbackMacEncrypt` and `CallbackDecryptVerify` in `ssl.h`:

```
typedef int (*CallbackMacEncrypt)(WOLFSSL* ssl,
    unsigned char* macOut,const unsigned char* macIn,
    unsigned int macInSz,int macContent, int macVerify,
    unsigned char* encOut, const unsigned char* encIn,
    unsigned int encSz,void* ctx);

typedef int (*CallbackDecryptVerify)(WOLFSSL* ssl,
    unsigned char* decOut, const unsigned char* decIn,
    unsigned int decSz, int content, int verify,
    unsigned int* padSz, void* ctx);
```

The user needs to write and register these functions per wolfSSL context (`WOLFSSL_CTX`) with `wolfSSL_CTX_SetMacEncryptCb()` and [wolfSSL_CTX_SetDecryptVerifyCb()](#function-wolfssl_ctx_setdecryptverifycb().

The user can set a context per WOLFSSL object (session) with `wolfSSL_SetMacEncryptCtx()` and `wolfSSL_SetDecryptVerifyCtx()`. This context may be a pointer to any user-specified context, which will then in turn be passed back to the MAC/encrypt and decrypt/verify callbacks through the `void* ctx` parameter.

1. Example callbacks can be found in `wolfssl/test.h`, under `myMacEncryptCb()` and `myDecryptVerifyCb()`. Usage can be seen in the wolfSSL example client (`examples/client/client.c`), when using the `-U` command line option.

To use Atomic Record Layer callbacks, wolfSSL needs to be compiled using the `--enable-atomicuser` configure option, or by defining the `ATOMIC_USER` preprocessor flag.

6.4 Public Key Callbacks

wolfSSL provides Public Key (PK) callbacks for users who wish to have more control over DH, ECC, Ed25519, X25519, Ed448, X448, and RSA operations during the SSL/TLS connection.

To use Public Key callbacks, wolfSSL needs to be compiled with HAVE_PK_CALLBACKS defined. This can be done using the configure option:

```
./configure --enable-pkcallbacks
```

Additionally, each callback type requires its specific feature to be enabled (e.g., HAVE_DH, HAVE_ED25519, etc.).

6.4.1 DH Callbacks

wolfSSL provides DH (Diffie-Hellman) callbacks for users who wish to have more control over DH key generation and key agreement operations during the SSL/TLS connection.

The user can optionally define 2 functions: 1. DH key generation callback 2. DH key agreement callback

These functions are prototyped by CallbackDhGenerateKeyPair and CallbackDhAgree in ssl.h:

```
typedef int (*CallbackDhGenerateKeyPair)(DhKey* key, WC_RNG* rng,
                                         byte* priv, word32* privSz,
                                         byte* pub, word32* pubSz);

typedef int (*CallbackDhAgree)(WOLFSSL* ssl, struct DhKey* key,
        const unsigned char* priv, unsigned int privSz,
        const unsigned char* otherPubKeyDer, unsigned int otherPubKeySz,
        unsigned char* out, word32* outlen,
        void* ctx);
```

The user needs to write and register these functions per wolfSSL context (WOLFSSL_CTX) with: * wolfSSL_CTX_SetDhGenerateKeyPair() * wolfSSL_CTX_SetDhAgreeCb()

The user can set a context per WOLFSSL object (session) with wolfSSL_SetDhAgreeCtx().

Example callbacks can be found in wolfssl/test.h, under myDhCallback(). Usage can be seen in the wolfSSL example client.

To use DH callbacks, wolfSSL needs to be compiled with HAVE_DH defined.

6.4.2 Ed25519 Callbacks

wolfSSL provides Ed25519 callbacks for users who wish to have more control over Ed25519 sign/verify operations during the SSL/TLS connection.

The user can optionally define 2 functions: 1. Ed25519 sign callback 2. Ed25519 verify callback

These functions are prototyped by CallbackEd25519Sign and CallbackEd25519Verify in ssl.h:

```
typedef int (*CallbackEd25519Sign)(WOLFSSL* ssl,
       const unsigned char* in, unsigned int inSz,
       unsigned char* out, unsigned int* outSz,
       const unsigned char* keyDer, unsigned int keySz,
       void* ctx);

typedef int (*CallbackEd25519Verify)(WOLFSSL* ssl,
       const unsigned char* sig, unsigned int sigSz,
       const unsigned char* msg, unsigned int msgSz,
```

```
        const unsigned char* keyDer, unsigned int keySz,
        int* result, void* ctx);
```

The user needs to write and register these functions per wolfSSL context (WOLFSSL_CTX) with: * wolfSSL_CTX_SetEd25519SignCb() * wolfSSL_CTX_SetEd25519VerifyCb()

The user can set a context per WOLFSSL object (session) with: * wolfSSL_SetEd25519SignCtx() * wolfSSL_SetEd25519VerifyCtx()

Example callbacks can be found in wolfssl/test.h, under myEd25519Sign() and myEd25519Verify(). Usage can be seen in the wolfSSL example client.

To use Ed25519 callbacks, wolfSSL needs to be compiled with HAVE_PK_CALLBACKS and HAVE_ED25519 defined.

6.4.3 X25519 Callbacks

wolfSSL provides X25519 callbacks for users who wish to have more control over X25519 key generation and shared secret computation during the SSL/TLS connection.

The user can optionally define 2 functions: 1. X25519 key generation callback 2. X25519 shared secret callback

These functions are prototyped by CallbackX25519KeyGen and CallbackX25519SharedSecret in ssl.h:

```
typedef int (*CallbackX25519KeyGen)(WOLFSSL* ssl, struct curve25519_key* key,
    unsigned int keySz, void* ctx);

typedef int (*CallbackX25519SharedSecret)(WOLFSSL* ssl,
        struct curve25519_key* otherKey,
        unsigned char* pubKeyDer, unsigned int* pubKeySz,
        unsigned char* out, unsigned int* outlen,
        int side, void* ctx);
```

The user needs to write and register these functions per wolfSSL context (WOLFSSL_CTX) with: * wolfSSL_CTX_SetX25519KeyGenCb() * wolfSSL_CTX_SetX25519SharedSecretCb()

The user can set a context per WOLFSSL object (session) with: * wolfSSL_SetX25519KeyGenCtx() * wolfSSL_SetX25519SharedSecretCtx()

Example callbacks can be found in wolfssl/test.h, under myX25519KeyGen() and myX25519SharedSecret(). Usage can be seen in the wolfSSL example client.

To use X25519 callbacks, wolfSSL needs to be compiled with HAVE_PK_CALLBACKS and HAVE_CURVE25519 defined.

6.4.4 Ed448 Callbacks

wolfSSL provides Ed448 callbacks for users who wish to have more control over Ed448 sign/verify operations during the SSL/TLS connection.

The user can optionally define 2 functions: 1. Ed448 sign callback 2. Ed448 verify callback

These functions are prototyped by CallbackEd448Sign and CallbackEd448Verify in ssl.h:

```
typedef int (*CallbackEd448Sign)(WOLFSSL* ssl,
        const unsigned char* in, unsigned int inSz,
        unsigned char* out, unsigned int* outSz,
        const unsigned char* keyDer, unsigned int keySz,
        void* ctx);
```

```
typedef int (*CallbackEd448Verify)(WOLFSSL* ssl,
       const unsigned char* sig, unsigned int sigSz,
       const unsigned char* msg, unsigned int msgSz,
       const unsigned char* keyDer, unsigned int keySz,
       int* result, void* ctx);
```

The user needs to write and register these functions per wolfSSL context (`WOLFSSL_CTX`) with: * `wolfSSL_CTX_SetEd448SignCb()` * `wolfSSL_CTX_SetEd448VerifyCb()`

The user can set a context per WOLFSSL object (session) with: * `wolfSSL_SetEd448SignCtx()` * `wolfSSL_SetEd448VerifyCtx()`

Example callbacks can be found in `wolfssl/test.h`, under `myEd448Sign()` and `myEd448Verify()`. Usage can be seen in the wolfSSL example client.

To use Ed448 callbacks, wolfSSL needs to be compiled with `HAVE_PK_CALLBACKS` and `HAVE_ED448` defined.

6.4.5 X448 Callbacks

wolfSSL provides X448 callbacks for users who wish to have more control over X448 key generation and shared secret operations during the SSL/TLS connection.

The user can optionally define 2 functions: 1. X448 key generation callback 2. X448 shared secret callback

These functions are prototyped by `CallbackX448KeyGen` and `CallbackX448SharedSecret` in `ssl.h`:

```
typedef int (*CallbackX448KeyGen)(WOLFSSL* ssl, struct curve448_key* key,
    unsigned int keySz, void* ctx);

typedef int (*CallbackX448SharedSecret)(WOLFSSL* ssl,
        struct curve448_key* otherKey,
        unsigned char* pubKeyDer, unsigned int* pubKeySz,
        unsigned char* out, unsigned int* outlen,
        int side, void* ctx);
```

The user needs to write and register these functions per wolfSSL context (`WOLFSSL_CTX`) with: * `wolfSSL_CTX_SetX448KeyGenCb()` * `wolfSSL_CTX_SetX448SharedSecretCb()`

The user can set a context per WOLFSSL object (session) with: * `wolfSSL_SetX448KeyGenCtx()` * `wolfSSL_SetX448SharedSecretCtx()`

Example callbacks can be found in `wolfssl/test.h`, under `myX448KeyGen()` and `myX448SharedSecret()`. Usage can be seen in the wolfSSL example client.

To use X448 callbacks, wolfSSL needs to be compiled with `HAVE_PK_CALLBACKS` and `HAVE_CURVE448` defined.

6.4.6 RSA PSS Callbacks

wolfSSL provides RSA PSS callbacks for users who wish to have more control over RSA PSS sign/verify operations during the SSL/TLS connection.

The user can optionally define 4 functions: 1. RSA PSS sign callback 2. RSA PSS verify callback 3. RSA PSS sign check callback 4. RSA PSS verify check callback

These functions are prototyped by CallbackRsaPssSign, CallbackRsaPssVerify, and CallbackRsaPssSignCheck in ssl.h:

```
typedef int (*CallbackRsaPssSign)(WOLFSSL* ssl,
        const unsigned char* in, unsigned int inSz,
        unsigned char* out, unsigned int* outSz,
        int hash, int mgf,
        const unsigned char* keyDer, unsigned int keySz,
        void* ctx);

typedef int (*CallbackRsaPssVerify)(WOLFSSL* ssl,
        unsigned char* sig, unsigned int sigSz,
        unsigned char** out,
        int hash, int mgf,
        const unsigned char* keyDer, unsigned int keySz,
        void* ctx);

typedef int (*CallbackRsaPssSignCheck)(WOLFSSL* ssl,
        unsigned char* sig, unsigned int sigSz,
        unsigned char** out,
        const unsigned char* keyDer, unsigned int keySz,
        void* ctx);
```

The user needs to write and register these functions per wolfSSL context (WOLFSSL_CTX) with: * wolfSSL_CTX_SetRsaPssSignCb() * wolfSSL_CTX_SetRsaPssVerifyCb() * wolfSSL_CTX_SetRsaSignCheckCb() * wolfSSL_CTX_SetRsaPssSignCheckCb()

The user can set a context per WOLFSSL object (session) with: * wolfSSL_SetRsaPssSignCtx() * wolfSSL_SetRsaPssVerifyCtx() * wolfSSL_SetRsaSignCheckCtx() * wolfSSL_SetRsaPssSignCheckCtx()

Example callbacks can be found in wolfssl/test.h, under myRsaPssSign(), myRsaPssVerify(), and myRsaPssSignCheck(). Usage can be seen in the wolfSSL example client.

To use RSA PSS callbacks, wolfSSL needs to be compiled with HAVE_PK_CALLBACKS and WC_RSA_PSS defined.

6.4.7 ECC Callbacks

The user can optionally define 7 functions:

1. ECC sign callback
2. ECC verify callback
3. ECC shared secret callback
4. RSA sign callback
5. RSA verify callback
6. RSA encrypt callback
7. RSA decrypt callback

These two functions are prototyped by CallbackEccSign, CallbackEccVerify, CallbackEccSharedSecret, CallbackRsaSign, CallbackRsaVerify, CallbackRsaEnc, and CallbackRsaDec in ssl.h:

```
typedef int (*CallbackEccSign)(WOLFSSL* ssl, const unsigned
        char* in, unsigned int inSz, unsigned char* out,
        unsigned int* outSz, const unsigned char* keyDer,
        unsigned int keySz, void* ctx);

typedef int (*CallbackEccVerify)(WOLFSSL* ssl,
```

```
    const unsigned char* sig, unsigned int sigSz,
    const unsigned char* hash, unsigned int hashSz,
    const unsigned char* keyDer, unsigned int keySz,
    int* result, void* ctx);

typedef int (*CallbackEccSharedSecret)(WOLFSSL* ssl,
        struct ecc_key* otherKey,
    unsigned char* pubKeyDer, unsigned int* pubKeySz,
    unsigned char* out, unsigned int* outlen,
    int side, void* ctx);

typedef int (*CallbackRsaSign)(WOLFSSL* ssl,
    const unsigned char* in, unsigned int inSz,
    unsigned char* out, unsigned int* outSz,
    const unsigned char* keyDer, unsigned int keySz,
    void* ctx);

typedef int (*CallbackRsaVerify)(WOLFSSL* ssl,
    unsigned char* sig, unsigned int sigSz,
    unsigned char** out, const unsigned char* keyDer,
    unsigned int keySz, void* ctx);

typedef int (*CallbackRsaEnc)(WOLFSSL* ssl,
    const unsigned char* in, unsigned int inSz,
    Unsigned char* out, unsigned int* outSz,
    const unsigned char* keyDer,
    unsigned int keySz, void* ctx);

typedef int (*CallbackRsaDec)(WOLFSSL* ssl, unsigned char* in,
    unsigned int inSz, unsigned char** out,
    const unsigned char* keyDer, unsigned int keySz,
    void* ctx);
```

The user needs to write and register these functions per wolfSSL context (WOLFSSL_CTX) with:

- wolfSSL_CTX_SetEccSignCb()
- wolfSSL_CTX_SetEccVerifyCb()
- wolfSSL_CTX_SetEccSharedSecretCb()
- wolfSSL_CTX_SetRsaSignCb()
- wolfSSL_CTX_SetRsaVerifyCb()
- wolfSSL_CTX_SetRsaEncCb()
- wolfSSL_CTX_SetRsaDecCb()

The user can set a context per WOLFSSL object (session) with:

- wolfSSL_SetEccSignCtx()
- wolfSSL_SetEccVerifyCtx()
- wolfSSL_SetEccSharedSecretCtx()
- wolfSSL_SetRsaSignCtx()
- wolfSSL_SetRsaVerifyCtx()
- wolfSSL_SetRsaEncCtx()
- wolfSSL_SetRsaDecCtx()

These contexts may be pointers to any user-specified context, which will then in turn be passed back to the respective public key callback through the void* ctx parameter.

Example callbacks can be found in wolfssl/test.h, under myEccSign(), myEccVerify(), myEcc-

SharedSecret(), myRsaSign(), myRsaVerify(), myRsaEnc(), and myRsaDec(). Usage can be seen in the wolfSSL example client (examples/client/client.c), when using the -P command line option.

To use Atomic Record Layer callbacks, wolfSSL needs to be compiled using the --enable-pkcallbacks configure option, or by defining the HAVE_PK_CALLBACKS preprocessor flag.

6.5 Crypto Callbacks (cryptocb)

The Crypto callback framework in wolfSSL/wolfCrypt enables users to override the default implementation of select cryptographic algorithms and provide their own custom implementations at runtime. The most common use case for crypto callbacks is to offload an algorithm to custom hardware acceleration, however it could also be used to add additional logging/introspection, retrieve keys from secure storage, to call crypto from another library for standards compliance reasons, or even to perform a remote procedure call.

6.5.1 Using Crypto callbacks

The high level steps use crypto callbacks are:

1. Compile wolfSSL with crypto callback support
2. Register a callback and unique device ID (devId) with wolfCrypt
3. Pass the devId as an argument to a wolfCrypt API function
4. (TLS only): associate the devId with a wolfSSL context

6.5.1.1 1. Compile wolfSSL with crypto callback support Support for crypto callbacks is enabled in wolfSSL via the –enable-cryptocb configure option, or #define WOLF_CRYPTO_CB.

6.5.1.2 2. Register your callback and unique devID with wolfCrypt To use crypto callbacks, the user must register a top-level cryptographic callback function with wolfCrypt and provide a device ID (devId), which is an integer ID that uniquely identifies this particular callback function to wolfCrypt. This allows multiple crypto callbacks to be registered with wolfCrypt at once. The appropriate callback function will then be invoked internally by every wolfCrypt function that takes a devId parameter, or by wolfSSL when associated with a WOLFSSL_CTX or WOLFSSL structure.

To register a cryptographic callback function with wolfCrypt, use the wc_CryptoCb_RegisterDevice API. This takes a (devId), the callback function, and an optional user context that will be passed through to the callback when it is invoked.

```
typedef int (*CryptoDevCallbackFunc)(int devId, wc_CryptoInfo* info, void*
 ↪  ctx);

WOLFSSL_API int wc_CryptoCb_RegisterDevice(int devId,
                                           CryptoDevCallbackFunc cb,
                                           void* ctx);
```

Note: A registered devId can be any value other than -2, which is reserved for the special INVALID_DEVID enum value. Passing devId == INVALID_DEVID as an argument into a wolfCrypt API indicates that no callback should be invoked and to use the default internal implementation instead.

6.5.1.3 3. Pass the devId as an argument to a wolfCrypt API function For wolfCrypt API's, use the init functions that accept devId such as:

```
wc_InitRsaKey_ex
wc_ecc_init_ex
```

```
wc_AesInit
wc_InitSha256_ex
wc_InitSha_ex
wc_HmacInit
wc_InitCmac_ex
```

This will cause wolfCrypt to invoke the crypto callback in place of the default implementation. This is not an exhaustive API list. Please refer to the wolfCrypt API documentation to see if a particular algorithm supports crypto callbacks.

6.5.1.4 4. (TLS only): associate the devId with a wolfSSL context To enable use of a crypto callback when using TLS, you must supply the `devId` arguments on initialization of a `WOLFSSL_CTX` or `WOLFSSL` struct.

```
wolfSSL_CTX_SetDevId(ctx, devId);
wolfSSL_SetDevId(ssl, devId);
```

All relevant crypto for TLS connections associated with those structures will now invoke the crypto callback.

6.5.2 Writing your crypto callback

When a crypto callback is invoked, it will need to know what cryptographic operation has been requested, as well as the location of the input and output data and how to communicate success/failure back to wolfCrypt. This information is relayed to the callback through the `wc_CryptoInfo* info` argument.

At a high level, a crypto callback should determine if it supports the requested operation, act on the input data, update the `wc_CryptoInfo *info` argument with any relevant output data, and return a valid wolfCrypt error code.

6.5.2.1 Determining the Type of Request The `wc_CryptoInfo` structure contains a union of different structures (some of which are unions themselves), each corresponding to a specific type of cryptographic operation. The type of operation to be performed is determined by `info->algo_type`, which should be a variant of `enum wc_AlgoType` defined in `wolfssl/wolfcrypt/types.h`. Based on this `algo_type`, the appropriate union element within `wc_CryptoInfo` can be accessed to get or set parameters specific to the operation. We can refer to this top-level union as the "algo type union".

To determine the type of cryptographic request and process it accordingly, one would typically use a switch-case statement on the `algo_type` field of the `wc_CryptoInfo` structure. Each case within the switch would correspond to a different cryptographic operation, such as a symmetric cipher, hashing, public key operations, etc. There is a one-to-one mapping between `wc_AlgoType` and the corresponding algo type union, which are commented in the `wc_CryptoInfo` struct definition in `wolfssl/wolfcrypt/cryptocb.h`.

The crypto callback should return zero on success, or a valid wolfCrypt error code on failure. For unsupported algorithms, the callback should return `CRYPTOCB_UNAVAILABLE`, which will cause wolfCrypt to fall back to the internal implementation.

Here's a simplified example to illustrate this:

```
int myCryptoCallback(int devId, wc_CryptoInfo* info, void* ctx)
{
    int ret = CRYPTOCB_UNAVAILABLE;

    switch (info->algo_type) {
        case WC_ALGO_TYPE_HASH:
```

```
            /* Handle hashing, using algo type union: info->hash */
            ret = 0; /* or wolfCrypt error code */
            break;

        case WC_ALGO_TYPE_PK:
            /* Handle public key operations, using algo type union: info->pk */
            ret = 0; /* or wolfCrypt error code */
            break;

        case WC_ALGO_TYPE_CIPHER:
            /* Handle cipher operations, using algo type union: info->cipher */
            ret = 0; /* or wolfCrypt error code */
            break;

        /* and so on for other algo types... */
    }

    return ret; // Return success or an appropriate error code
}
```

Some of the simpler algo type unions, such as the RNG and seed unions, require no further processing and can be immediately acted on. However, more complicated operations like cipher or public key types have multiple union variants unto themselves that must be conditionally interpreted in the same manner. The variants and levels of hierarchy of each union are specific to each category of algorithm type, the full definitions of which can be found in the definition of the `wc_CryptoInfo` structure. As a general rule, each level that is a union will contain some sort of type indicator informing how the rest of the union should be dereferenced.

Here is a simplified example for a complex algo type union with multiple levels of union decoding. The callback contains support for random number generation, as well as ECC key agreement, sign, and verify.

```
int myCryptoCallback(int devId, wc_CryptoInfo* info, void* ctx)
{
    int ret = CRYPTOCB_UNAVAILABLE;

    switch (info->algo_type) {
        case WC_ALGO_TYPE_PK:
            /* Handle public key operations, using algo type union: info->pk */
            switch (info->pk.type) { /* pk type is of type wc_PkType */
                case WC_PK_TYPE_ECDH:
                    /* use info->pk.ecdh */
                    ret = 0; /* or wolfCrypt error code */
                    break;

                case WC_PK_TYPE_ECDSA_SIGN:
                    /* use info->pk.eccsign */
                    ret = 0; /* or wolfCrypt error code */
                    break;

                case WC_PK_TYPE_ECDSA_VERIFY:
                    /* use info->pk.eccverify */
                    ret = 0; /* or wolfCrypt error code */
                    break;
            }
            break;
```

```
        case WC_ALGO_TYPE_RNG:
            /* use info->rng */
            ret = 0; /* or wolfCrypt error code */
            break;
    }

    return ret;
}
```

6.5.3 Handling the request

The data structure for each type of request generally contains a pointer and associated size for input and output data. Depending on the request it may include additional data including cryptographic keys, nonces, or further configuration. The crypto callback should operate on the input data and write relevant output data back to the appropriate variant of the algo type union. Writing data to a union variant that does not correspond to the algorithm type in question (e.g. using `info->cipher` when `info->algo_type == WC_ALGO_TYPE_RNG` is a memory error and can result in undefined behavior.

6.5.4 Troubleshooting

Some older compilers don't allow "anonymous inline aggregates", which wolfCrypt uses when defining the nested `wcCryptoInfo` anonymous unions in order to save space. To disable the use of anonymous inline aggregates, define the `HAVE_ANONYMOUS_INLINE_AGGREGATES` preprocessor macro as `0`. This will allow crypto callbacks to be used, but will dramatically increase the size of the `wcCryptoInfo` structure.

6.5.5 Examples

Full examples of crypto callbacks can be found in the following locations

- VaultIC Crypto Callbacks
- STSAFE-A100 ECC Crypto Callbacks
- TPM 2.0 wolfTPM Crypto Callbacks
- Generic wolfCrypt tests

7 Keys and Certificates

For an introduction to X.509 certificates, as well as how they are used in SSL and TLS, please see Appendix A.

7.1 Supported Formats and Sizes

wolfSSL (formerly CyaSSL) has support for **PEM**, and **DER** formats for certificates and keys, as well as PKCS#8 private keys (with PKCS#5 or PKCS#12 encryption).

PEM, or "Privacy Enhanced Mail" is the most common format that certificates are issued in by certificate authorities. PEM files are Base64 encoded ASCII files which can include multiple server certificates, intermediate certificates, and private keys, and usually have a `.pem`, `.crt`, `.cer`, or `.key` file extension. Certificates inside PEM files are wrapped in the "`-----BEGIN CERTIFICATE-----`" and "`-----END CERTIFICATE-----`" statements.

DER, or "Distinguished Encoding Rules", is a binary format of a certificate. DER file extensions can include `.der` and `.cer`, and cannot be viewed with a text editor.

An X.509 certificate is encoded using ASN.1 format. The DER format is the ASN.1 encoding. The PEM format is Base64 encoded and wrapped with a human readable header and footer. TLS sends certificates in DER format.

7.2 Supported Certificate Extensions

If an unsupported or unknown extension that is marked as critical is found, then an error message is returned, otherwise unsupported or unknown extensions found are ignored. Certificate extension parsing expects that at the very least `--enable-certext` (macro WOLFSSL_CERT_EXT) has been used when compiling the wolfSSL library. This is a high level list of certificate extensions that can be **parsed** and at least part, if not all, of the extensions be used.

Extension From RFC 5280	Supported
Authority Key Identifier	Yes
Subject Key Identifier	Yes
Key Usage	Yes
Certificate Policies	Yes
Policy Mappings	No
Subject Alternative Name	Yes
Issuer Alternative Name	No
Subject Directory Attributes	No
Basic Constraints	Yes
Name Constraints	Yes
Policy Constraints	Yes
Extended Key Usage	Yes
CRL Distribution Points	Yes
Inihibit anyPolicy	Yes
Freshest CRL	No

Additional Extension	Supported
Netscape	Yes
Custom OID	Yes

The next couple of sections delve deeper into the support of the individual certificate extensions.

7.2.0.1 Auth/Subject Key ID By default key ID's are a SHA1 hash of the key. SHA256 hashes of the key are also supported.

7.2.0.2 Subject Alternative Names

Alternative Name Types	Supported
email	Yes
DNS name	Yes
IP address	Yes
URI	Yes

7.2.0.3 Key Usage Key usage can be parsed and retrieved after parsing of the certificate is complete.

Key Usage	Supported
digitalSignature	Yes
nonRepudiation	Yes
keyEncipherment	Yes
dataEncipherment	Yes
keyAgreement	Yes
keyCertSign	Yes
cRLSign	Yes
encipherOnly	Yes
decipherOnly	Yes

7.2.0.4 Extended Key Usage Starting at wolfSSL version 3.15.5 if an extended key usage is unknown/unsupported then it is ignored. For versions before 3.15.5 an unknown extended key usage OID will cause a parsing error.

Extended Key Usage	Supported
anyExtendedKeyUsage	Yes
id-kp-serverAuth	Yes
id-kp-clientAuth	Yes
id-kp-codeSigning	Yes
id-kp-emailProtection	Yes
id-kp-timeStamping	Yes
id-kp-OCSPSigning	Yes

7.2.0.5 Custom OID Extensions Custom OID X.509 extension injection and parsing was introduced in wolfSSL version 5.3.0. The enable options `--enable-certgen` must be used along with the macros `WOLFSSL_ASN_TEMPLATE`, `WOLFSSL_CUSTOM_OID` and `HAVE_OID_ENCODING` to enable custom extensions. `WOLFSSL_ASN_TEMPLATE` is defined by default when using `./configure` but needs to be manually defined if building with `WOLFSSL_USER_SETTINGS`.

After building wolfSSL with these settings the function `wc_SetCustomExtension()` can be used to set a custom extension in a `Cert` struct and `wc_SetUnknownExtCallback()` can be used to register a callback for handling unknown extension OIDs in a `DecodedCert` struct.

7.3 Certificate Loading

Certificates are normally loaded using the file system (although loading from memory buffers is supported as well - see No File System and using Certificates).

7.3.1 Loading CA Certificates**

CA certificate files can be loaded using the wolfSSL_CTX_load_verify_locations() function:

```
int wolfSSL_CTX_load_verify_locations(WOLFSSL_CTX *ctx,
                                      const char *CAfile,
                                      const char *CApath);
```

CA loading can also parse multiple CA certificates per file using the above function by passing in a CAfile in PEM format with as many certs as possible. This makes initialization easier, and is useful when a client needs to load several root CAs at startup. This makes wolfSSL easier to port into tools that expect to be able to use a single file for CAs.

NOTE: If you have to load a chain of Roots and Intermediate certificates you must load them in the order of trust. Load ROOT CA first followed by Intermediate 1 followed by Intermediate 2 and so on. You may call wolfSSL_CTX_load_verify_locations() for each cert to be loaded or just once with a file containing the certs in order (Root at the top of the file and certs ordered by the chain of trust)

7.3.2 Loading Client or Server Certificates

Loading single client or server certificates can be done with the wolfSSL_CTX_use_certificate_file() function. If this function is used with a certificate chain, only the actual, or "bottom" certificate will be sent.

```
int wolfSSL_CTX_use_certificate_file(WOLFSSL_CTX *ctx,
                                     const char *CAfile,
                                     int type);
```

CAfile is the CA certificate file, and type is the format of the certificate - such as SSL_FILETYPE_PEM.

The server and client can send certificate chains using the wolfSSL_CTX_use_certificate_chain_file() function. The certificate chain file must be in PEM format and must be sorted starting with the subject's certificate (the actual client or server cert), followed by any intermediate certificates and ending (optionally) at the root "top" CA. The example server (/examples/server/server.c) uses this functionality.

NOTE: This is the exact reverse of the order necessary when loading a certificate chain for verification! Your file contents in this scenario would be Entity cert at the top of the file followed by the next cert up the chain and so on with Root CA at the bottom of the file.

```
int wolfSSL_CTX_use_certificate_chain_file(WOLFSSL_CTX *ctx,
                                   const char *file);
```

7.3.3 Loading Private Keys

Server private keys can be loaded using the wolfSSL_CTX_use_PrivateKey_file() function.

```
int wolfSSL_CTX_use_PrivateKey_file(WOLFSSL_CTX *ctx,
                       const char *keyFile, int type);
```

keyFile is the private key file, and type is the format of the private key (e.g. SSL_FILETYPE_PEM).

7.3.4 Loading Trusted Peer Certificates

Loading a trusted peer certificate to use can be done with wolfSSL_CTX_trust_peer_cert().

```
int wolfSSL_CTX_trust_peer_cert(WOLFSSL_CTX *ctx,
                const char *trustCert, int type);
```

trustCert is the certificate file to load, and type is the format of the private key (i.e. SSL_FILETYPE_PEM).

7.4 Certificate Chain Verification

wolfSSL requires that only the top or "root" certificate in a chain to be loaded as a trusted certificate in order to verify a certificate chain. This means that if you have a certificate chain (A -> B -> C), where C is signed by B, and B is signed by A, wolfSSL only requires that certificate A be loaded as a trusted certificate in order to verify the entire chain (A->B->C).

For example, if a server certificate chain looks like:

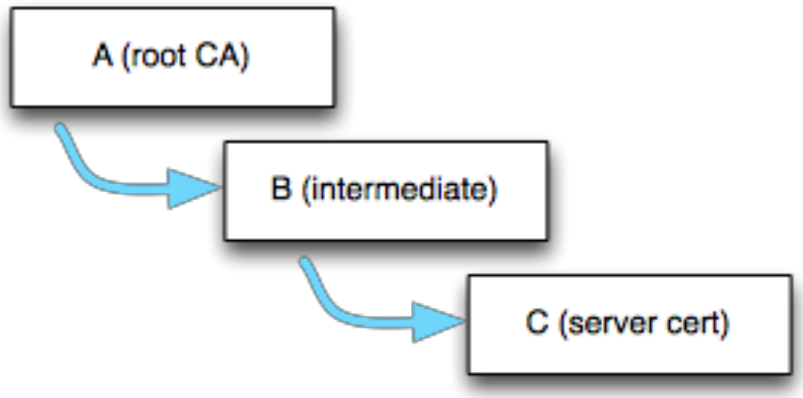

Figure 4: Certificate Chain

The wolfSSL client should already have at least the root cert (A) loaded as a trusted root (with wolfSSL_CTX_load_verify_locations()). When the client receives the server cert chain, it uses the signature of A to verify B, and if B has not been previously loaded into wolfSSL as a trusted root, B gets stored in wolfSSL's internal cert chain (wolfSSL just stores what is necessary to verify a certificate: common name hash, public key and key type, etc.). If B is valid, then it is used to verify C.

Following this model, as long as root cert "A" has been loaded as a trusted root into the wolfSSL server, the server certificate chain will still be able to be verified if the server sends (A->B->C), or (B->C). If the server just sends (C), and not the intermediate certificate, the chain will not be able to be verified unless the wolfSSL client has already loaded B as a trusted root.

7.5 Domain Name Check for Server Certificates

wolfSSL has an extension on the client that automatically checks the domain of the server certificate. In OpenSSL mode nearly a dozen function calls are needed to perform this. wolfSSL checks that the date of the certificate is in range, verifies the signature, and additionally verifies the domain if you call wolfSSL_check_domain_name(WOLFSSL* ssl, const char* dn) will return DOMAIN_NAME_MISMATCH.

Checking the domain name of the certificate is an important step that verifies the server is actually who it claims to be. This extension is intended to ease the burden of performing the check.

7.6 No File System and using Certificates

Normally a file system is used to load private keys, certificates, and CAs. Since wolfSSL is sometimes used in environments without a full file system an extension to use memory buffers instead is provided. To use the extension define the constant NO_FILESYSTEM and the following functions will be made available:

- `int wolfSSL_CTX_load_verify_buffer(WOLFSSL_CTX* ctx, const unsigned char* in,long sz, int format);`
- `int wolfSSL_CTX_use_certificate_buffer(WOLFSSL_CTX* ctx, const unsigned char* in, long sz, int format);`
- `int wolfSSL_CTX_use_PrivateKey_buffer(WOLFSSL_CTX* ctx, const unsigned char* in, long sz, int format);`
- `int wolfSSL_CTX_use_certificate_chain_buffer(WOLFSSL_CTX* ctx, const unsigned char* in,long sz);`
- `int wolfSSL_CTX_trust_peer_buffer(WOLFSSL_CTX* ctx, const unsigned char* in, Long sz, int format);`

Use these functions exactly like their counterparts that are named *_file instead of *_buffer. And instead of providing a filename provide a memory buffer. See API documentation for usage details.

7.6.1 Test Certificate and Key Buffers

wolfSSL has come bundled with test certificate and key files in the past. Now it also comes bundled with test certificate and key buffers for use in environments with no filesystem available. These buffers are available in certs_test.h when defining one or more of USE_CERT_BUFFERS_1024, USE_CERT_BUFFERS_2048, or USE_CERT_BUFFERS_256.

7.7 Serial Number Retrieval

The serial number of an X.509 certificate can be extracted from wolfSSL using wolfSSL_X509_get_serial_number(). The serial number can be of any length.

```
int wolfSSL_X509_get_serial_number(WOLFSSL_X509* x509,
            unsigned char* buffer, int* inOutSz)
```

buffer will be written to with at most *inOutSz bytes on input. After the call, if successful (return of 0), *inOutSz will hold the actual number of bytes written to buffer. A full example is included wolfssl/test.h.

7.8 RSA Key Generation

wolfSSL supports RSA key generation of varying lengths up to 4096 bits. Key generation is off by default but can be turned on during the ./configure process with --enable-keygen or by defining WOLFSSL_KEY_GEN in Windows or non-standard environments. Creating a key is easy, only requiring one function from rsa.h:

```
int MakeRsaKey(RsaKey* key, int size, long e, RNG* rng);
```

Where size is the length in bits and e is the public exponent, using 65537 is usually a good choice for e. The following from wolfcrypt/test/test.c gives an example creating an RSA key of 1024 bits:

```
RsaKey genKey;
RNG    rng;
int    ret;

InitRng(&rng);
```

```
InitRsaKey(&genKey, 0);

ret = MakeRsaKey(&genKey, 1024, 65537, &rng);
if (ret != 0)
    /* ret contains error */;
```

The RsaKey genKey can now be used like any other RsaKey. If you need to export the key, wolfSSL provides both DER and PEM formatting in asn.h. Always convert the key to DER format first, and then if you need PEM use the generic DerToPem() function like this:

```
byte der[4096];
int  derSz = RsaKeyToDer(&genKey, der, sizeof(der));
if (derSz < 0)
    /* derSz contains error */;
```

The buffer der now holds a DER format of the key. To convert the DER buffer to PEM use the conversion function:

```
byte pem[4096];
int  pemSz = DerToPem(der, derSz, pem, sizeof(pem),
                      PRIVATEKEY_TYPE);
if (pemSz < 0)
    /* pemSz contains error */;
```

The last argument of *DerToPem()* takes a type parameter, usually either PRIVATEKEY_TYPE or CERT_TYPE. Now the buffer pem holds the PEM format of the key. Supported types are:

- CA_TYPE
- TRUSTED_PEER_TYPE
- CERT_TYPE
- CRL_TYPE
- DH_PARAM_TYPE
- DSA_PARAM_TYPE
- CERTREQ_TYPE
- DSA_TYPE
- DSA_PRIVATEKEY_TYPE
- ECC_TYPE
- ECC_PRIVATEKEY_TYPE
- RSA_TYPE
- PRIVATEKEY_TYPE
- ED25519_TYPE
- EDDSA_PRIVATEKEY_TYPE
- PUBLICKEY_TYPE
- ECC_PUBLICKEY_TYPE
- PKCS8_PRIVATEKEY_TYPE
- PKCS8_ENC_PRIVATEKEY_TYPE

7.8.1 RSA Key Generation Notes

The RSA private key contains the public key as well. The private key can be used as both a private and public key by wolfSSL as used in test.c. The private key and the public key (in the form of a certificate) is all that is typically needed for SSL.

A separate public key can be loaded into wolfSSL manually using the RsaPublicKeyDecode() function if need be. Additionally, the wc_RsaKeyToPublicDer() function can be used to export the public RSA key.

7.9 Certificate Generation

wolfSSL supports X.509 v3 certificate generation. Certificate generation is off by default but can be turned on during the `./configure` process with `--enable-certgen` or by defining `WOLFSSL_CERT_GEN` in Windows or non-standard environments.

Before a certificate can be generated the user needs to provide information about the subject of the certificate. This information is contained in a structure from `wolfssl/wolfcrypt/asn_public.h` named `Cert`:

```
/* for user to fill for certificate generation */
typedef struct Cert {
    int      version;                   /* x509 version  */
    byte     serial[CTC_SERIAL_SIZE];   /* serial number */
    int      sigType;                   /*signature algo type */
    CertName issuer;                    /* issuer info */
    int      daysValid;                 /* validity days */
    int      selfSigned;                /* self signed flag */
    CertName subject;                   /* subject info */
    int      isCA;                      /*is this going to be a CA*/
    ...
} Cert;
```

Where CertName looks like:

```
typedef struct CertName {
char country[CTC_NAME_SIZE];
    char countryEnc;
    char state[CTC_NAME_SIZE];
    char stateEnc;
    char locality[CTC_NAME_SIZE];
    char localityEnc;
    char sur[CTC_NAME_SIZE];
    char surEnc;
    char org[CTC_NAME_SIZE];
    char orgEnc;
    char unit[CTC_NAME_SIZE];
    char unitEnc;
    char commonName[CTC_NAME_SIZE];
    char commonNameEnc;
    char email[CTC_NAME_SIZE];  /* !!!! email has to be last!!!! */
} CertName;
```

Before filling in the subject information an initialization function needs to be called like this:

```
Cert myCert;
InitCert(&myCert);
```

`InitCert()` sets defaults for some of the variables including setting the version to **3** (0x02), the serial number to **0** (randomly generated), the sigType to `CTC_SHAwRSA`, the daysValid to **500**, and selfSigned to **1** (TRUE). Supported signature types include:

- `CTC_SHAwDSA`
- `CTC_MD2wRSA`
- `CTC_MD5wRSA`
- `CTC_SHAwRSA`
- `CTC_SHAwECDSA`
- `CTC_SHA256wRSA`

- CTC_SHA256wECDSA
- CTC_SHA384wRSA
- CTC_SHA384wECDSA
- CTC_SHA512wRSA
- CTC_SHA512wECDSA

Now the user can initialize the subject information like this example from wolfcrypt/test/test.c:

```
strncpy(myCert.subject.country, "US", CTC_NAME_SIZE);
strncpy(myCert.subject.state, "OR", CTC_NAME_SIZE);
strncpy(myCert.subject.locality, "Portland", CTC_NAME_SIZE);
strncpy(myCert.subject.org, "yaSSL", CTC_NAME_SIZE);
strncpy(myCert.subject.unit, "Development", CTC_NAME_SIZE);
strncpy(myCert.subject.commonName, "www.wolfssl.com", CTC_NAME_SIZE);
strncpy(myCert.subject.email, "info@wolfssl.com", CTC_NAME_SIZE);
```

Then, a self-signed certificate can be generated using the variables genKey and rng from the above key generation example (of course any valid RsaKey or RNG can be used):

```
byte derCert[4096];

int certSz = MakeSelfCert(&myCert, derCert, sizeof(derCert), &key, &rng);
if (certSz < 0)
  /* certSz contains the error */;
```

The buffer derCert now contains a DER format of the certificate. If you need a PEM format of the certificate you can use the generic DerToPem() function and specify the type to be CERT_TYPE like this:

```
byte* pem;

int pemSz = DerToPem(derCert, certSz, pem, sizeof(pemCert), CERT_TYPE);
if (pemCertSz < 0)
  /* pemCertSz contains error */;
```

Supported types are:

- CA_TYPE
- TRUSTED_PEER_TYPE
- CERT_TYPE
- CRL_TYPE
- DH_PARAM_TYPE
- DSA_PARAM_TYPE
- CERTREQ_TYPE
- DSA_TYPE
- DSA_PRIVATEKEY_TYPE
- ECC_TYPE
- ECC_PRIVATEKEY_TYPE
- RSA_TYPE
- PRIVATEKEY_TYPE
- ED25519_TYPE
- EDDSA_PRIVATEKEY_TYPE
- PUBLICKEY_TYPE
- ECC_PUBLICKEY_TYPE
- PKCS8_PRIVATEKEY_TYPE
- PKCS8_ENC_PRIVATEKEY_TYPE

Now the buffer pemCert holds the PEM format of the certificate.

If you wish to create a CA signed certificate then a couple of steps are required. After filling in the subject information as before, you'll need to set the issuer information from the CA certificate. This can be done with SetIssuer() like this:

```
ret = SetIssuer(&myCert, "ca-cert.pem");
if (ret < 0)
    /* ret contains error */;
```

Then you'll need to perform the two-step process of creating the certificate and then signing it (MakeSelfCert() does these both in one step). You'll need the private keys from both the issuer (caKey) and the subject (key). Please see the example in test.c for complete usage.

```
byte derCert[4096];

int certSz = MakeCert(&myCert, derCert, sizeof(derCert), &key, NULL, &rng);
if (certSz < 0);
   /*certSz contains the error*/;

certSz = SignCert(myCert.bodySz, myCert.sigType, derCert,
           sizeof(derCert), &caKey, NULL, &rng);
if (certSz < 0);
   /*certSz contains the error*/;
```

The buffer derCert now contains a DER format of the CA signed certificate. If you need a PEM format of the certificate please see the self signed example above. Note that MakeCert() and SignCert() provide function parameters for either an RSA or ECC key to be used. The above example uses an RSA key and passes NULL for the ECC key parameter.

7.10 Certificate Signing Request (CSR) Generation

wolfSSL supports X.509 v3 certificate signing request (CSR) generation. CSR generation is off by default but can be turned on during the ./configure process with --enable-certreq --enable-certgen or by defining WOLFSSL_CERT_GEN **and** WOLFSSL_CERT_REQ in Windows or non-standard environments.

Before a CSR can be generated the user needs to provide information about the subject of the certificate. This information is contained in a structure from wolfssl/wolfcrypt/asn_public.h named Cert:

For details on the Cert and CertName structures please reference Certificate Generation above.

Before filling in the subject information an initialization function needs to be called like this:

```
Cert request;
InitCert(&request);
```

InitCert() sets defaults for some of the variables including setting the version to **3** (0x02), the serial number to **0** (randomly generated), the sigType to CTC_SHAwRSA, the daysValid to **500**, and selfSigned to **1** (TRUE). Supported signature types include:

- CTC_SHAwDSA
- CTC_MD2wRSA
- CTC_MD5wRSA
- CTC_SHAwRSA
- CTC_SHAwECDSA
- CTC_SHA256wRSA
- CTC_SHA256wECDSA
- CTC_SHA384wRSA
- CTC_SHA384wECDSA

- CTC_SHA512wRSA
- CTC_SHA512wECDSA

Now the user can initialize the subject information like this example from https://github.com/wolfSSL/wolfssl-examples/blob/master/certgen/csr_example.c:

```
strncpy(req.subject.country, "US", CTC_NAME_SIZE);
strncpy(req.subject.state, "OR", CTC_NAME_SIZE);
strncpy(req.subject.locality, "Portland", CTC_NAME_SIZE);
strncpy(req.subject.org, "wolfSSL", CTC_NAME_SIZE);
strncpy(req.subject.unit, "Development", CTC_NAME_SIZE);
strncpy(req.subject.commonName, "www.wolfssl.com", CTC_NAME_SIZE);
strncpy(req.subject.email, "info@wolfssl.com", CTC_NAME_SIZE);
```

Then, a valid signed CSR can be generated using the variable key from the above key generation example (of course any valid ECC/RSA key or RNG can be used):

```
byte der[4096]; /* Store request in der format once made */

ret = wc_MakeCertReq(&request, der, sizeof(der), NULL, &key);
/* check ret value for error handling, <= 0 indicates a failure */
```

Next you will want to sign your request making it valid, use the rng variable from the above key generation example. (of course any valid ECC/RSA key or RNG can be used)

```
derSz = ret;

req.sigType = CTC_SHA256wECDSA;
ret = wc_SignCert(request.bodySz, request.sigType, der, sizeof(der), NULL,
 ↪  &key, &rng);
/* check ret value for error handling, <= 0 indicates a failure */
```

Lastly it is time to convert the CSR to PEM format for sending to a CA authority to use in issueing a certificate:

```
ret = wc_DerToPem(der, derSz, pem, sizeof(pem), CERTREQ_TYPE);
/* check ret value for error handling, <= 0 indicates a failure */
printf("%s", pem); /* or write to a file */
```

7.10.1 Limitations

There are fields that are mandatory in a certificate that are excluded in a CSR. There are other fields in a CSR that are also deemed “optional” that are otherwise mandatory when in a certificate. Because of this the wolfSSL certificate parsing engine, which strictly checks all certificate fields AND considers all fields mandatory, does not support consuming a CSR at this time. Therefore while CSR generation AND certificate generation from scratch are supported, wolfSSL does not support certificate generation FROM a CSR. Passing in a CSR to the wolfSSL parsing engine will return a failure at this time. Check back for updates once we support consuming a CSR for use in certificate generation!

See also: Certificate Generation

7.11 Convert to raw ECC key

With our recently added support for raw ECC key import comes the ability to convert an ECC key from PEM to DER. Use the following with the specified arguments to accomplish this:

```
EccKeyToDer(ecc_key*, byte* output, word32 inLen);
```

7.11.1 Example

```
#define FOURK_BUF 4096
byte  der[FOURK_BUF];
ecc_key userB;

EccKeyToDer(&userB, der, FOURK_BUF);
```

8 Debugging

8.1 Debugging and Logging

wolfSSL (formerly CyaSSL) has support for debugging through log messages in environments where debugging is limited. To turn logging on use the function `wolfSSL_Debugging_ON()`. In a normal build (release mode) these functions will have no effect. In a debug build, define `DEBUG_WOLFSSL` to ensure these functions are turned on.

As of wolfSSL 2.0, logging callback functions may be registered at runtime to provide more flexibility with how logging is done. The logging callback can be registered with the function `wolfSSL_SetLoggingCb()`:

```
int wolfSSL_SetLoggingCb(wolfSSL_Logging_cb log_function);

typedef void (*wolfSSL_Logging_cb)(const int logLevel,
                  const char *const logMessage);
```

The log levels can be found in `wolfssl/wolfcrypt/logging.h`, and the implementation is located in `logging.c`. By default, wolfSSL logs to `stderr` with `fprintf`.

8.2 Error Codes

wolfSSL tries to provide informative error messages in order to help with debugging.

Each `wolfSSL_read()` and `wolfSSL_write()` call will return the number of bytes written upon success, 0 upon connection closure, and -1 for an error, just like `read()` and `write()`. In the event of an error you can use two calls to get more information about the error.

The function `wolfSSL_get_error()` will return the current error code. It takes the current `WOLFSSL` object, and `wolfSSL_read()` or `wolfSSL_write()` result value as arguments and returns the current error code.

```
int err = wolfSSL_get_error(ssl, result);
```

To get a more human-readable error code description, the `wolfSSL_ERR_error_string()` function can be used. It takes the return code from `wolfSSL_get_error` and a storage buffer as arguments, and places the corresponding error description into the storage buffer (`errorString` in the example below).

```
char errorString[80];
wolfSSL_ERR_error_string(err, errorString);
```

If you are using non blocking sockets, you can test for errno EAGAIN/EWOULDBLOCK or more correctly you can test the specific error code for `SSL_ERROR_WANT_READ` or `SSL_ERROR_WANT_WRITE`.

For a list of wolfSSL and wolfCrypt error codes, please see Appendix C (Error Codes).

9 Library Design

9.1 Library Headers

With the release of wolfSSL 2.0.0 RC3, library header files are now located in the following locations:

- wolfSSL: `wolfssl/`
- wolfCrypt: `wolfssl/wolfcrypt/`
- wolfSSL OpenSSL Compatibility Layer: `wolfssl/openssl/`

When using the OpenSSL Compatibility layer (see OpenSSL Compatibility), the `/wolfssl/openssl/ssl.h` header is required to be included:

```
#include <wolfssl/openssl/ssl.h>
```

When using only the wolfSSL native API, only the `/wolfssl/ssl.h` header is required to be included:

```
#include <wolfssl/ssl.h>
```

9.2 Startup and Exit

All applications should call `wolfSSL_Init()` at program termination. Currently these functions only initialize and free the shared mutex for the session cache in multi-user mode but in the future they may do more so it's always a good idea to use them.

9.3 Structure Usage

In addition to header file location changes, the release of wolfSSL 2.0.0 RC3 created a more visible distinction between the native wolfSSL API and the wolfSSL OpenSSL Compatibility Layer. With this distinction, the main SSL/TLS structures used by the native wolfSSL API have changed names. The new structures are as follows. The previous names are still used when using the OpenSSL Compatibility Layer (see OpenSSL Compatibility).

- `WOLFSSL` (previously SSL)
- `WOLFSSL_CTX` (previously SSL_CTX)
- `WOLFSSL_METHOD` (previously SSL_METHOD)
- `WOLFSSL_SESSION` (previously SSL_SESSION)
- `WOLFSSL_X509` (previously X509)
- `WOLFSSL_X509_NAME` (previously X509_NAME)
- `WOLFSSL_X509_CHAIN` (previously X509_CHAIN)

9.4 Thread Safety

wolfSSL (formerly CyaSSL) is thread safe by design. Multiple threads can enter the library simultaneously without creating conflicts because wolfSSL avoids global data, static data, and the sharing of objects. The user must still take care to avoid potential problems in some areas.

1. A client may share an WOLFSSL object across multiple threads but access must be synchronized, i.e., trying to read/write at the same time from two different threads with the same SSL pointer is not supported.

   wolfSSL could take a more aggressive (constrictive) stance and lock out other users when a function is entered that cannot be shared but this level of granularity seems counter-intuitive. All users (even single threaded ones) will pay for the locking and multi-thread ones won't be able to re-enter the library even if they aren't sharing objects across threads. This penalty seems much too high and wolfSSL leaves the responsibility of synchronizing shared objects in the hands of the user.

2. Besides sharing WOLFSSL pointers, users must also take care to completely initialize an WOLFSSL_CTX before passing the structure to wolfSSL_new() creation and any future (or simultaneous changes) to the WOLFSSL_CTX will not be reflected once the WOLFSSL object is created.

   Again, multiple threads should synchronize writing access to a WOLFSSL_CTX and it is advised that a single thread initialize the WOLFSSL_CTX to avoid the synchronization and update problem described above.

3. Some optimizations allocate memory on a per thread basis. If fixed point ECC cache is enabled (FP_ECC), then threads should release the cached buffers using wc_ecc_fp_free() before the thread exits.

9.5 Input and Output Buffers

wolfSSL now uses dynamic buffers for input and output. They default to 0 bytes and are controlled by the RECORD_SIZE define in wolfssl/internal.h. If an input record is received that is greater in size than the static buffer, then a dynamic buffer is temporarily used to handle the request and then freed. You can set the static buffer size up to the MAX_RECORD_SIZE which is 2^16 or 16,384.

If you prefer the previous way that wolfSSL operated, with 16Kb static buffers that will never need dynamic memory, you can still get that option by defining LARGE_STATIC_BUFFERS.

If dynamic buffers are used and the user requests a wolfSSL_write() that is bigger than the buffer size, then a dynamic block up to MAX_RECORD_SIZE is used to send the data. Users wishing to only send the data in chunks of at most RECORD_SIZE size can do this by defining STATIC_CHUNKS_ONLY. This will cause wolfSSL to use I/O buffers which grow up to RECORD_SIZE, which is 128 bytes by default.

COPYRIGHT ©2024 wolfSSL Inc.

10 wolfCrypt Usage Reference

wolfCrypt is the cryptography library primarily used by wolfSSL. It is optimized for speed, small footprint, and portability. wolfSSL interchanges with other cryptography libraries as required.

Types used in the examples:

```
typedef unsigned char byte;
typedef unsigned int  word32;
```

10.1 Hash Functions

10.1.1 MD4

NOTE: MD4 is outdated and considered insecure. Please consider using a different hashing function if possible.

To use MD4 include the MD4 header `wolfssl/wolfcrypt/md4.h`. The structure to use is `Md4`, which is a typedef. Before using, the hash initialization must be done with the `wc_InitMd4()` to retrieve the final hash.

```
byte md4sum[MD4_DIGEST_SIZE];
byte buffer[1024];
/* fill buffer with data to hash*/

Md4 md4;
wc_InitMd4(&md4);

wc_Md4Update(&md4, buffer, sizeof(buffer));  /*can be called again
                                    and again*/
wc_Md4Final(&md4, md4sum);
```

`md4sum` now contains the digest of the hashed data in buffer.

10.1.2 MD5

NOTE: MD5 is outdated and considered insecure. Please consider using a different hashing function if possible.

To use MD5 include the MD5 header `wolfssl/wolfcrypt/md5.h`. The structure to use is `Md5`, which is a typedef. Before using, the hash initialization must be done with the `wc_InitMd5()` to retrieve the final hash

```
byte md5sum[MD5_DIGEST_SIZE];
byte buffer[1024];
/*fill buffer with data to hash*/

Md5 md5;
wc_InitMd5(&md5);

wc_Md5Update(&md5, buffer, sizeof(buffer));  /*can be called again
                                    and again*/
wc_Md5Final(&md5, md5sum);
```

`md5sum` now contains the digest of the hashed data in buffer.

10.1.3 SHA / SHA-224 / SHA-256 / SHA-384 / SHA-512

To use SHA include the SHA header `wolfssl/wolfcrypt/sha.h`. The structure to use is Sha, which is a typedef. Before using, the hash initialization must be done with the `wc_InitSha()` to retrieve the final hash:

```
byte shaSum[SHA_DIGEST_SIZE];
byte buffer[1024];
/*fill buffer with data to hash*/

Sha sha;
wc_InitSha(&sha);

wc_ShaUpdate(&sha, buffer, sizeof(buffer));  /*can be called again
                                               and again*/
wc_ShaFinal(&sha, shaSum);
```

`shaSum` now contains the digest of the hashed data in buffer.

To use either SHA-224, SHA-256, SHA-384, or SHA-512, follow the same steps as shown above, but use either the `wolfssl/wolfcrypt/sha256.h` or `wolfssl/wolfcrypt/sha512.h` (for both SHA-384 and SHA-512). The SHA-256, SHA-384, and SHA-512 functions are named similarly to the SHA functions.

For **SHA-224**, the functions `wc_InitSha224()` will be used with the structure Sha224.

For **SHA-256**, the functions `wc_InitSha256()` will be used with the structure Sha256.

For **SHA-384**, the functions `wc_InitSha384()` will be used with the structure Sha384.

For **SHA-512**, the functions `wc_InitSha512()` will be used with the structure Sha512.

SHA interleaving (typically only of interest to hardware-acceleration that supports it) is enabled by default, define `NO_WOLFSSL_SHA256_INTERLEAVE` to disable it. Software SHA has always supported interleaving.

10.1.4 BLAKE2b

To use BLAKE2b (a SHA-3 finalist) include the BLAKE2b header `wolfssl/wolfcrypt/blake2.h`. The structure to use is `Blake2b`, which is a typedef. Before using, the hash initialization must be done with the `wc_InitBlake2b()` to retrieve the final hash:

```
byte digest[64];
byte input[64];     /*fill input with data to hash*/

Blake2b b2b;
wc_InitBlake2b(&b2b, 64);

wc_Blake2bUpdate(&b2b, input, sizeof(input));
wc_Blake2bFinal(&b2b, digest, 64);
```

The second parameter to `wc_InitBlake2b()` should be the final digest size. `digest` now contains the digest of the hashed data in buffer.

Example usage can be found in the wolfCrypt test application (`wolfcrypt/test/test.c`), inside the `blake2b_test()` function.

10.1.5 RIPEMD-160

To use RIPEMD-160, include the header wolfssl/wolfcrypt/ripemd.h. The structure to use is RipeMd, which is a typedef. Before using, the hash initialization must be done with the wc_InitRipeMd() to retrieve the final hash

```
byte ripeMdSum[RIPEMD_DIGEST_SIZE];
byte buffer[1024];
/*fill buffer with data to hash*/

RipeMd ripemd;
wc_InitRipeMd(&ripemd);

wc_RipeMdUpdate(&ripemd, buffer, sizeof(buffer));  /*can be called
                                          again and again*/
wc_RipeMdFinal(&ripemd, ripeMdSum);
```

ripeMdSum now contains the digest of the hashed data in buffer.

10.2 Keyed Hash Functions

10.2.1 HMAC

wolfCrypt currently provides HMAC for message digest needs. The structure Hmac is found in the header wolfssl/wolfcrypt/hmac.h. HMAC initialization is done with wc_HmacSetKey(). 5 different types are supported with HMAC: MD5, SHA, SHA-256, SHA-384, and SHA-512. Here's an example with SHA-256.

```
Hmac    hmac;
byte    key[24];        /*fill key with keying material*/
byte    buffer[2048];   /*fill buffer with data to digest*/
byte    hmacDigest[SHA256_DIGEST_SIZE];

wc_HmacSetKey(&hmac, SHA256, key, sizeof(key));
wc_HmacUpdate(&hmac, buffer, sizeof(buffer));
wc_HmacFinal(&hmac, hmacDigest);
```

hmacDigest now contains the digest of the hashed data in buffer.

10.2.2 GMAC

wolfCrypt also provides GMAC for message digest needs. The structure Gmac is found in the header wolfssl/wolfcrypt/aes.h, as it is an application AES-GCM. GMAC initialization is done with wc_GmacSetKey().

```
Gmac gmac;
byte    key[16];         /*fill key with keying material*/
byte iv[12];         /*fill iv with an initialization vector*/
byte    buffer[2048];   /*fill buffer with data to digest*/
byte    gmacDigest[16];

wc_GmacSetKey(&gmac, key, sizeof(key));
wc_GmacUpdate(&gmac, iv, sizeof(iv), buffer, sizeof(buffer),
gmacDigest, sizeof(gmacDigest));
```

gmacDigest now contains the digest of the hashed data in buffer.

10.2.3 Poly1305

wolfCrypt also provides Poly1305 for message digest needs. The structure Poly1305 is found in the header wolfssl/wolfcrypt/poly1305.h. Poly1305 initialization is done with wc_Poly1305SetKey() has been called.

```
Poly1305    pmac;
byte           key[32];        /*fill key with keying material*/
byte           buffer[2048];   /*fill buffer with data to digest*/
byte           pmacDigest[16];

wc_Poly1305SetKey(&pmac, key, sizeof(key));
wc_Poly1305Update(&pmac, buffer, sizeof(buffer));
wc_Poly1305Final(&pmac, pmacDigest);
```

pmacDigest now contains the digest of the hashed data in buffer.

10.3 Block Ciphers

10.3.1 AES

wolfCrypt provides support for AES with key sizes of 16 bytes (128 bits), 24 bytes (192 bits), or 32 bytes (256 bits). Supported AES modes include CBC, CTR, GCM (GCM-Streaming), OFB, CFB(1, 8, 128), SIV, XTS (XTS-Streaming), GMAC, CMAC, ECB, KW (KeyWrap), and CCM-8.

NOTE: wc_AesInit().

CBC mode is supported for both encryption and decryption and is provided through the wc_AesSetKey() functions. Please include the header wolfssl/wolfcrypt/aes.h to use AES. AES has a block size of 16 bytes and the IV should also be 16 bytes. Function usage is usually as follows:

```
Aes enc;
Aes dec;

const byte key[] = {  /*some 24 byte key*/ };
const byte iv[] = { /*some 16 byte iv*/ };

byte plain[32];   /*an increment of 16, fill with data*/
byte cipher[32];

wc_AesInit(&enc, HEAP_HINT, INVALID_DEVID);
wc_AesInit(&dec, HEAP_HINT, INVALID_DEVID);

/*encrypt*/
wc_AesSetKey(&enc, key, sizeof(key), iv, AES_ENCRYPTION);
wc_AesCbcEncrypt(&enc, cipher, plain, sizeof(plain));
```

cipher now contains the ciphertext from the plain text.

```
/*decrypt*/
wc_AesSetKey(&dec, key, sizeof(key), iv, AES_DECRYPTION);
wc_AesCbcDecrypt(&dec, plain, cipher, sizeof(cipher));
```

plain now contains the original plaintext from the ciphertext.

wolfCrypt also supports CTR (Counter), GCM (Galois/Counter), and CCM-8 (Counter with CBC-MAC) modes of operation for AES. When using these modes, like CBC, include the wolfssl/wolfcrypt/aes.h header.

GCM mode is available for both encryption and decryption through the wc_AesGcmSetKey() functions. For a usage example, see the aesgcm_test() function in <wolfssl_root>/wolfcrypt/test/test.c.

CCM-8 mode is supported for both encryption and decryption through the wc_AesCcmSetKey() functions. For a usage example, see the aesccm_test() function in <wolfssl_root>/wolfcrypt/test/test.c.

CTR mode is available for both encryption and decryption through the wc_AesCtrEncrypt() function. The encrypt and decrypt actions are identical so the same function is used for both. For a usage example, see the function aes_test() in file wolfcrypt/test/test.c.

10.3.1.1 DES and 3DES wolfCrypt provides support for DES and 3DES (Des3 since 3 is an invalid leading C identifier). To use these include the header wolfssl/wolfcrypt/des.h. The structures you can use are Des and Des3. Initialization is done through wc_Des_SetKey(). Des has a key size of 8 bytes (24 for 3DES) and the block size is 8 bytes, so only pass increments of 8 bytes to encrypt/decrypt functions. If your data isn't in a block size increment you'll need to add padding to make sure it is. Each SetKey() also takes an IV (an initialization vector that is the same size as the key size). Usage is usually like the following:

```
Des3 enc;
Des3 dec;

const byte key[] = {  /*some 24 byte key*/ };
const byte iv[]  = {  /*some 24 byte iv*/  };

byte plain[24];   /*an increment of 8, fill with data*/
byte cipher[24];

/*encrypt*/
wc_Des3_SetKey(&enc, key, iv, DES_ENCRYPTION);
wc_Des3_CbcEncrypt(&enc, cipher, plain, sizeof(plain));
```

cipher now contains the ciphertext from the plain text.

```
/*decrypt*/
wc_Des3_SetKey(&dec, key, iv, DES_DECRYPTION);
wc_Des3_CbcDecrypt(&dec, plain, cipher, sizeof(cipher));
```

plain now contains the original plaintext from the ciphertext.

10.3.1.2 Camellia wolfCrypt provides support for the Camellia block cipher. To use Camellia include the header wolfssl/wolfcrypt/camellia.h. The structure you can use is called Camellia. Initialization is done through wc_CamelliaSetKey().

For usage examples please see the camellia_test() function in <wolfssl_root>/wolfcrypt/test/test.c.

10.4 Stream Ciphers

10.4.1 ARC4

NOTE: ARC4 is outdated and considered insecure. Please consider using a different stream cipher.

The most common stream cipher used on the Internet is ARC4. wolfCrypt supports it through the header wolfssl/wolfcrypt/arc4.h. Usage is simpler than block ciphers because there is no block size and the key length can be any length. The following is a typical usage of ARC4.

```
Arc4 enc;
Arc4 dec;
```

```
const byte key[] = {  /*some key any length*/};

byte plain[27];   /*no size restriction, fill with data*/
byte cipher[27];

/*encrypt*/
wc_Arc4SetKey(&enc, key, sizeof(key));
wc_Arc4Process(&enc, cipher, plain, sizeof(plain));
```

cipher now contains the ciphertext from the plain text.

```
/*decrypt*/
wc_Arc4SetKey(&dec, key, sizeof(key));
wc_Arc4Process(&dec, plain, cipher, sizeof(cipher));
```

plain now contains the original plaintext from the ciphertext.

10.4.2 ChaCha

ChaCha with 20 rounds is slightly faster than ARC4 while maintaining a high level of security. To use it with wolfCrypt, please include the header wolfssl/wolfcrypt/chacha.h. ChaCha typically uses 32 byte keys (256 bit) but can also use 16 byte keys (128 bits).

```
CHACHA enc;
CHACHA dec;

const byte key[] = {  /*some key 32 bytes*/};
const byte iv[]  = {  /*some iv 12 bytes*/ };

byte plain[37];   /*no size restriction, fill with data*/
byte cipher[37];

/*encrypt*/
wc_Chacha_SetKey(&enc, key, keySz);
wc_Chacha_SetIV(&enc, iv, counter); /*counter is the start block
                        counter is usually set as 0*/
wc_Chacha_Process(&enc, cipher, plain, sizeof(plain));
```

cipher now contains the ciphertext from the plain text.

```
/*decrypt*/
wc_Chacha_SetKey(&enc, key, keySz);
wc_Chacha_SetIV(&enc, iv, counter);
wc_Chacha_Process(&enc, plain, cipher, sizeof(cipher));
```

plain now contains the original plaintext from the ciphertext.

wc_Chacha_SetKey must be called with a new iv (nonce). Counter is set as an argument to allow for partially decrypting/encrypting information by starting at a different block when performing the encrypt/decrypt process, but in most cases is set to 0. **ChaCha should not be used without a mac algorithm (e.g. Poly1305 , hmac).**

10.5 Public Key Cryptography

10.5.1 RSA

wolfCrypt provides support for RSA through the header wolfssl/wolfcrypt/rsa.h. There are two types of RSA keys, public and private. A public key allows anyone to encrypt something that only the

holder of the private key can decrypt. It also allows the private key holder to sign something and anyone with a public key can verify that only the private key holder actually signed it. Usage is usually like the following:

```
RsaKey rsaPublicKey;

byte publicKeyBuffer[]  = { /*holds the raw data from the key, maybe
                        from a file like RsaPublicKey.der*/ };
word32 idx = 0;            /*where to start reading into the buffer*/

RsaPublicKeyDecode(publicKeyBuffer, &idx, &rsaPublicKey,
 ↪  sizeof(publicKeyBuffer));

byte in[] = { /*plain text to encrypt*/ };
byte out[128];
RNG rng;

wc_InitRng(&rng);

word32 outLen = RsaPublicEncrypt(in, sizeof(in), out, sizeof(out),
 ↪  &rsaPublicKey, &rng);
```

Now out holds the ciphertext from the plain text in. `wc_RsaPublicEncrypt()` will write.

In the event of an error, a negative return from `wc_RsaPublicEnrypt()` to get a string describing the error that occurred.

```
void wc_ErrorString(int error, char* buffer);
```

Make sure that buffer is at least MAX_ERROR_SZ bytes (80).

Now to decrypt out:

```
RsaKey rsaPrivateKey;

byte privateKeyBuffer[] = { /*hold the raw data from the key, maybe
                            from a file like RsaPrivateKey.der*/ };
word32 idx = 0;           /*where to start reading into the buffer*/

wc_RsaPrivateKeyDecode(privateKeyBuffer, &idx, &rsaPrivateKey,
                       sizeof(privateKeyBuffer));

byte plain[128];
word32 plainSz = wc_RsaPrivateDecrypt(out, outLen, plain,
                       sizeof(plain), &rsaPrivateKey);
```

Now plain will hold plainSz bytes or an error code. For complete examples of each type in wolfCrypt please see the file `wolfcrypt/test/test.c`. Note that the `wc_RsaPrivateKeyDecode` function only accepts keys in raw DER format.

10.5.2 DH (Diffie-Hellman)

wolfCrypt provides support for Diffie-Hellman through the header `wolfssl/wolfrypt/dh.h`. The Diffie-Hellman key exchange algorithm allows two parties to establish a shared secret key. Usage is usually similar to the following example, where **sideA** and **sideB** designate the two parties.

In the following example, dhPublicKey contains the Diffie-Hellman public parameters signed by a Certificate Authority (or self-signed). privA holds the generated private key for sideA, pubA holds the generated public key for sideA, and agreeA holds the mutual key that both sides have agreed on.

```
DhKey   dhPublicKey;
word32 idx = 0;  /*where to start reading into the
                   publicKeyBuffer*/
word32 pubASz, pubBSz, agreeASz;
byte   tmp[1024];
RNG    rng;

byte privA[128];
byte pubA[128];
byte agreeA[128];

wc_InitDhKey(&dhPublicKey);

byte publicKeyBuffer[] = { /*holds the raw data from the public key
                             parameters, maybe from a file like
                             dh1024.der*/ }
wc_DhKeyDecode(tmp, &idx, &dhPublicKey, publicKeyBuffer);
wc_InitRng(&rng);  /*Initialize random number generator*/
```

`wc_DhGenerateKeyPair()` will generate a public and private DH key based on the initial public parameters in dhPublicKey.

```
wc_DhGenerateKeyPair(&dhPublicKey, &rng, privA, &privASz,
            pubA, &pubASz);
```

After sideB sends their public key (pubB) to sideA, sideA can then generate the mutually-agreed key(agreeA) using the `wc_DhAgree()` function.

```
wc_DhAgree(&dhPublicKey, agreeA, &agreeASz, privA, privASz,
           pubB, pubBSz);
```

Now, agreeA holds sideA's mutually-generated key (of size `agreeASz` bytes). The same process will have been done on sideB.

For a complete example of Diffie-Hellman in wolfCrypt, see the file `wolfcrypt/test/test.c`.

10.5.3 EDH (Ephemeral Diffie-Hellman)

A wolfSSL server can do Ephemeral Diffie-Hellman. No build changes are needed to add this feature, though an application will have to register the ephemeral group parameters on the server side to enable the EDH cipher suites. A new API can be used to do this:

```
int wolfSSL_SetTmpDH(WOLFSSL* ssl, unsigned char* p,
                     int pSz,unsigned char* g,int gSz);
```

The example server and echoserver use this function from `SetDH()`.

10.5.4 DSA (Digital Signature Algorithm)

wolfCrypt provides support for DSA and DSS through the header `wolfssl/wolfcrypt/dsa.h`. DSA allows for the creation of a digital signature based on a given data hash. DSA uses the SHA hash algorithm to generate a hash of a block of data, then signs that hash using the signer's private key. Standard usage is similar to the following.

We first declare our DSA key structure (key), initialize our initial message (`message`) to be signed, and initialize our DSA key buffer (`dsaKeyBuffer`).

```
DsaKey key;
Byte   message[]      = { /*message data to sign*/ }
```

```
byte    dsaKeyBuffer[] = { /*holds the raw data from the DSA key,
                              maybe from a file like dsa512.der*/ }
```

We then declare our SHA structure (sha), random number generator (rng), array to store our SHA hash (hash), array to store our signature (signature), idx (to mark where to start reading in our dsaKeyBuffer), and an int (answer) to hold our return value after verification.

```
Sha     sha;
RNG     rng;
byte    hash[SHA_DIGEST_SIZE];
byte    signature[40];
word32 idx = 0;
int     answer;
```

Set up and create the SHA hash. For more information on wolfCrypt's SHA algorithm, see SHA / SHA-224 / SHA-256 / SHA-384 / SHA-512. The SHA hash of message is stored in the variable hash.

```
wc_InitSha(&sha);
wc_ShaUpdate(&sha, message, sizeof(message));
wc_ShaFinal(&sha, hash);
```

Initialize the DSA key structure, populate the structure key value, and initialize the random number generator (rng).

```
wc_InitDsaKey(&key);
wc_DsaPrivateKeyDecode(dsaKeyBuffer, &idx, &key,
                       sizeof(dsaKeyBuffer));
wc_InitRng(&rng);
```

The wc_DsaSign() function creates a signature (signature) using the DSA private key, hash value, and random number generator.

```
wc_DsaSign(hash, signature, &key, &rng);
```

To verify the signature, use wc_DsaVerify().

```
wc_DsaVerify(hash, signature, &key, &answer);
wc_FreeDsaKey(&key);
```

11 SSL Tutorial

11.1 Introduction

The wolfSSL embedded SSL/TLS library can easily be integrated into your existing application or device to provide enhanced communication security through the addition of SSL and TLS. wolfSSL has been targeted at embedded and RTOS environments, and as such, offers a minimal footprint while maintaining excellent performance. Minimum build sizes for wolfSSL range between 20-100kB depending on the selected build options and platform being used.

The goal of this tutorial is to walk through the integration of SSL and TLS into a simple application. Hopefully the process of going through this tutorial will also lead to a better understanding of SSL in general. This tutorial uses wolfSSL in conjunction with simple echoserver and echoclient examples to keep things as simple as possible while still demonstrating the general procedure of adding SSL support to an application. The echoserver and echoclient examples have been taken from the popular book titled Unix Network Programming, Volume 1, 3rd Edition by Richard Stevens, Bill Fenner, and Andrew Rudoff.

This tutorial assumes that the reader is comfortable with editing and compiling C code using the GNU GCC compiler as well as familiar with the concepts of public key encryption. Please note that access to the Unix Network Programming book is not required for this tutorial.

11.1.1 Examples Used in this Tutorial

- echoclient - Figure 5.4, Page 124
- echoserver - Figure 5.12, Page 139

11.2 Quick Summary of SSL/TLS

TLS (Transport Layer Security) and **SSL** (Secure Sockets Layer) are cryptographic protocols that allow for secure communication across a number of different transport protocols. The primary transport protocol used is TCP/IP. The most recent version of SSL/TLS is TLS 1.3. wolfSSL supports SSL 3.0, TLS 1.0, 1.1, 1.2, 1.3 in addition to DTLS 1.0, 1.2, and 1.3

SSL and TLS sit between the Transport and Application layers of the OSI model, where any number of protocols (including TCP/IP, Bluetooth, etc.) may act as the underlying transport medium. Application protocols are layered on top of SSL and can include protocols such as HTTP, FTP, and SMTP. A diagram of how SSL fits into the OSI model, as well as a simple diagram of the SSL handshake process can be found in Appendix A.

11.3 Getting the Source Code

All of the source code used in this tutorial can be downloaded from the wolfSSL website, specifically from the following location. The download contains both the original and completed source code for both the echoserver and echoclient used in this tutorial. Specific contents are listed below the link.

https://www.wolfssl.com/documentation/ssl-tutorial-2.5.zip

The downloaded ZIP file has the following structure:

```
/finished_src
    /certs  (Certificate files)
    /echoclient (Completed echoclient code)
    /echoserver (Completed echoserver code)
    /include    (Modified unp.h)
    /lib    (Library functions)
```

```
/original_src
    /echoclient (Starting echoclient code)
    /echoserver (Starting echoserver code)
    /include    (Modified unp.h)
    /lib    (Library functions)
README
```

11.4 Base Example Modifications

This tutorial, and the source code that accompanies it, have been designed to be as portable as possible across platforms. Because of this, and because we want to focus on how to add SSL and TLS into an application, the base examples have been kept as simple as possible. Several modifications have been made to the examples taken from Unix Network Programming in order to either remove unnecessary complexity or increase the range of platforms supported. If you believe there is something we could do to increase the portability of this tutorial, please let us know at support@wolfssl.com.

The following is a list of modifications that were made to the original echoserver and echoclient examples found in the above listed book.

11.4.1 Modifications to the echoserver (tcpserv04.c)

- Removed call to the `fork()` function because `fork()` is not supported by Windows. The result of this is an echoserver which only accepts one client simultaneously. Along with this removal, Signal handling was removed.
- Moved `str_echo()` function from `str_echo.c` file into `tcpserv04.c` file
- Added a printf statement to view the client address and the port we have connected through:

```
printf("Connection from %s, port %d\n",
        inet_ntop(AF_INET, &cliaddr.sin_addr, buff, sizeof(buff)),
        ntohs(cliaddr.sin_port));
```

- Added a call to `setsockopt()` after creating the listening socket to eliminate the “Address already in use” bind error.
- Minor adjustments to clean up newer compiler warnings

11.4.2 Modifications to the echoclient (tcpcli01.c)

- Moved `str_cli()` function from `str_cli.c` file into `tcpcli01.c` file
- Minor adjustments to clean up newer compiler warnings

11.4.3 Modifications to unp.h header

- This header was simplified to contain only what is needed for this example.

Please note that in these source code examples, certain functions will be capitalized. For example, `Fputs()` and `Writen()`. The authors of Unix Network Programming have written custom wrapper functions for normal functions in order to cleanly handle error checking. For a more thorough explanation of this, please see **Section 1.4** (page 11) in the *Unix Network Programming* book.

11.5 Building and Installing wolfSSL

Before we begin, download the example code (`echoserver` and `echoclient`) from the Getting the Source Code section, above. This section will explain how to download, configure, and install the wolfSSL embedded SSL/TLS library on your system.

You will need to download and install the most recent version of wolfSSL from the wolfSSL download page.

For a full list of available build options, see the Building wolfSSL guide. wolfSSL was written with portability in mind, and should generally be easy to build on most systems. If you have difficulty building wolfSSL, please feel free to ask for support on the wolfSSL product support forums.

When building wolfSSL on Linux, *BSD, OS X, Solaris, or other* nix like systems, you can use the autoconf system. For Windows-specific instructions, please refer to the Building wolfSSL section of the wolfSSL Manual. To configure and build wolfSSL, run the following two commands from the terminal. Any desired build options may be appended to `./configure` (ex: `./configure --enable-opensslextra`):

```
./configure
make
```

To install wolfSSL, run:

```
sudo make install
```

This will install wolfSSL headers into `/usr/local/include/wolfssl` and the wolfSSL libraries into `/usr/local/lib` on your system. To test the build, run the testsuite application from the wolfSSL root directory:

```
./testsuite/testsuite.test
```

A set of tests will be run on wolfCrypt and wolfSSL to verify it has been installed correctly. After a successful run of the testsuite application, you should see output similar to the following:

```
------------------------------------------------------------------------------
 wolfSSL version 5.7.0
------------------------------------------------------------------------------
error    test passed!
MEMORY   test passed!
base64   test passed!
asn      test passed!
RANDOM   test passed!
MD5      test passed!
SHA      test passed!
SHA-224  test passed!
SHA-256  test passed!
SHA-384  test passed!
SHA-512  test passed!
SHA-512/224  test passed!
SHA-512/256  test passed!
SHA-3    test passed!
Hash     test passed!
HMAC-MD5 test passed!
HMAC-SHA test passed!
HMAC-SHA224 test passed!
HMAC-SHA256 test passed!
HMAC-SHA384 test passed!
HMAC-SHA512 test passed!
```

```
HMAC-SHA3   test passed!
HMAC-KDF    test passed!
PRF         test passed!
TLSv1.3 KDF test passed!
GMAC     test passed!
Chacha   test passed!
POLY1305 test passed!
ChaCha20-Poly1305 AEAD test passed!
AES      test passed!
AES192   test passed!
AES256   test passed!
AES-GCM  test passed!
RSA      test passed!
DH       test passed!
PWDBASED test passed!
ECC      test passed!
logging  test passed!
time     test passed!
mutex    test passed!
memcb    test passed!
Test complete

Running simple test
SSL version is TLSv1.2
SSL version is TLSv1.2
SSL cipher suite is TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
SSL curve name is SECP256R1
SSL cipher suite is TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
SSL curve name is SECP256R1
Client message: hello wolfssl!
I hear you fa shizzle!

Running TLS test
sending server shutdown command: quit!
client sent quit command: shutting down!
ciphers = TLS13-AES128-GCM-SHA256:TLS13-AES256-GCM-SHA384:TLS13-CHACHA20-
    POLY1305-SHA256:DHE-RSA-AES128-SHA:DHE-RSA-AES256-SHA:ECDHE-RSA-AES128-SHA:
    ECDHE-RSA-AES256-SHA:ECDHE-ECDSA-AES128-SHA:ECDHE-ECDSA-AES256-SHA:DHE-RSA-
    AES128-SHA256:DHE-RSA-AES256-SHA256:DHE-RSA-AES128-GCM-SHA256:DHE-RSA-
    AES256-GCM-SHA384:ECDHE-RSA-AES128-GCM-SHA256:ECDHE-RSA-AES256-GCM-SHA384:
    ECDHE-ECDSA-AES128-GCM-SHA256:ECDHE-ECDSA-AES256-GCM-SHA384:ECDHE-RSA-
    AES128-SHA256:ECDHE-ECDSA-AES128-SHA256:ECDHE-RSA-AES256-SHA384:ECDHE-ECDSA
    -AES256-SHA384:ECDHE-RSA-CHACHA20-POLY1305:ECDHE-ECDSA-CHACHA20-POLY1305:
    DHE-RSA-CHACHA20-POLY1305:ECDHE-RSA-CHACHA20-POLY1305-OLD:ECDHE-ECDSA-
    CHACHA20-POLY1305-OLD:DHE-RSA-CHACHA20-POLY1305-OLD
33bc1a4570f4f1abccd5c48aace529b01a42ab51293954a297796e90d20970f0  input
33bc1a4570f4f1abccd5c48aace529b01a42ab51293954a297796e90d20970f0  /var/folders
    /dy/888x7r7d6dgcqw4840l32tpw0000gp/T//testsuite-output-9Ymbuv

All tests passed!
```

Now that wolfSSL has been installed, we can begin modifying the example code to add SSL functionality. We will first begin by adding SSL to the echoclient and subsequently move on to the echoserver.

11.6 Initial Compilation

To compile and run the example echoclient and echoserver code from the SSL Tutorial source bundle, you can use the included Makefiles. Change directory (cd) to either the echoclient or echoserver directory and run:

```
make
```

This will compile the example code and produce an executable named either echoserver or echoclient depending on which one is being built. The GCC command which is used in the Makefile can be seen below. If you want to build one of the examples without using the supplied Makefile, change directory to the example directory and replace `tcpcli01.c` (echoclient) or `tcpserv04.c` (echoserver) in the following command with correct source file for the example:

```
gcc -o echoserver ../lib/*.c tcpserv04.c -I ../include
```

This will compile the current example into an executable, creating either an "echoserver" or "echoclient" application. To run one of the examples after it has been compiled, change your current directory to the desired example directory and start the application. For example, to start the echoserver use:

```
./echoserver
```

You may open a second terminal window to test the echoclient on your local host and you will need to supply the IP address of the server when starting the application, which in our case will be 127.0.0.1. Change your current directory to the "echoclient" directory and run the following command. Note that the echoserver must already be running:

```
./echoclient 127.0.0.1
```

Once you have both the echoserver and echoclient running, the echoserver should echo back any input that it receives from the echoclient. To exit either the echoserver or echoclient, use *Ctrl + C* to quit the application. Currently, the data being echoed back and forth between these two examples is being sent in the clear - easily allowing anyone with a little bit of skill to inject themselves in between the client and server and listen to your communication.

11.7 Libraries

The wolfSSL library, once compiled, is named `libwolfssl`, and unless otherwise configured the wolfSSL build and install process creates only a shared library under the following directory. Both shared and static libraries may be enabled or disabled by using the appropriate build options:

```
/usr/local/lib
```

The first step we need to do is link the wolfSSL library to our example applications. Modifying the GCC command (using the echoserver as an example), gives us the following new command. Since wolfSSL installs header files and libraries in standard locations, the compiler should be able to find them without explicit instructions (using `-l` or `-L`). Note that by using `-lwolfssl` the compiler will automatically choose the correct type of library (static or shared):

```
gcc -o echoserver ../lib/*.c tcpserv04.c -I ../include -lm -lwolfssl
```

11.8 Headers

The first thing we will need to do is include the wolfSSL native API header in both the client and the server. In the `tcpcli01.c` file for the client and the `tcpserv04.c` file for the server add the following line near the top:

```
#include <wolfssl/ssl.h>
```

11.9 Startup/Shutdown

Before we can use wolfSSL in our code, we need to initialize the library and the WOLFSSL_CTX. wolfSSL is initialized by calling wolfSSL_Init(). This must be done first before anything else can be done with the library.

11.10 WOLFSSL_CTX Factory

The WOLFSSL_CTX structure (wolfSSL Context) contains global values for each SSL connection, including certificate information. A single WOLFSSL_CTX can be used with any number of WOLFSSL objects created. This allows us to load certain information, such as a list of trusted CA certificates only once.

To create a new WOLFSSL_CTX, use wolfSSL_CTX_new(). The possible client and server protocol options are shown below. SSL 2.0 is not supported by wolfSSL because it has been insecure for several years.

EchoClient:

- wolfSSLv3_client_method(); - SSL 3.0
- wolfTLSv1_client_method(); - TLS 1.0
- wolfTLSv1_1_client_method(); - TLS 1.1
- wolfTLSv1_2_client_method(); - TLS 1.2
- wolfTLSv1_3_client_method(); - TLS 1.3
- wolfSSLv23_client_method(); - Use highest version possible from SSL 3.0 - TLS 1.3
- wolfDTLSv1_client_method(); - DTLS 1.0
- wolfDTLSv1_2_client_method_ex(); - DTLS 1.2
- wolfDTLSv1_3_client_method_ex(); - DTLS 1.3

EchoServer:

- wolfSSLv3_server_method(); - SSL 3.0
- wolfTLSv1_server_method(); - TLS 1.0
- wolfTLSv1_1_server_method(); - TLS 1.1
- wolfTLSv1_2_server_method(); - TLS 1.2
- wolfTLSv1_3_server_method(); - TLS 1.3
- wolfSSLv23_server_method(); - Allow clients to connect with TLS 1.0+
- wolfDTLSv1_server_method(); - DTLS 1.0
- wolfDTLSv1_2_server_method(); - DTLS 1.2
- wolfDTLSv1_3_server_method(); - DTLS 1.3

We need to load our CA (Certificate Authority) certificate into the WOLFSSL_CTX so that the when the echoclient connects to the echoserver, it is able to verify the server's identity. To load the CA certificates into the WOLFSSL_CTX, use wolfSSL_CTX_load_verify_locations() function returns either SSL_SUCCESS or SSL_FAILURE:

```
wolfSSL_CTX_load_verify_locations(WOLFSSL_CTX* ctx, const char* file, const
↪  char* path)
```

Putting these things together (library initialization, protocol selection, and CA certificate), we have the following. Here, we choose to use TLS 1.2:

EchoClient:

```
    WOLFSSL_CTX* ctx;

    wolfSSL_Init();/* Initialize wolfSSL */

    /* Create the WOLFSSL_CTX */
    if ( (ctx = wolfSSL_CTX_new(wolfTLSv1_2_client_method())) == NULL) {
```

```
        fprintf(stderr, "wolfSSL_CTX_new error.\n");
        exit(EXIT_FAILURE);
    }

    /* Load CA certificates into WOLFSSL_CTX */
    if (wolfSSL_CTX_load_verify_locations(ctx,"../certs/ca-cert.pem",0) !=
        SSL_SUCCESS) {
        fprintf(stderr, "Error loading ../certs/ca-cert.pem, please check"
            "the file.\n");
        exit(EXIT_FAILURE);
    }
```

Add the above code to `tcpcli01.c` in `main()` after the variable definitions and the check that the user has started the client with an IP address.

EchoServer:

When loading certificates into the `WOLFSSL_CTX`, the server certificate and key file should be loaded in addition to the CA certificate. This will allow the server to send the client its certificate for identification verification:

```
WOLFSSL_CTX* ctx;

wolfSSL_Init();  /* Initialize wolfSSL */

/* Create the WOLFSSL_CTX */
if ( (ctx = wolfSSL_CTX_new(wolfTLSv1_2_server_method())) == NULL) {
    fprintf(stderr, "wolfSSL_CTX_new error.\n");
    exit(EXIT_FAILURE);
}

/* Load CA certificates into WOLFSSL_CTX */
if (wolfSSL_CTX_load_verify_locations(ctx, "../certs/ca-cert.pem", 0) !=
         SSL_SUCCESS) {
    fprintf(stderr, "Error loading ../certs/ca-cert.pem, "
        "please check the file.\n");
    exit(EXIT_FAILURE);
}

/* Load server certificates into WOLFSSL_CTX */
if (wolfSSL_CTX_use_certificate_file(ctx,"../certs/server-cert.pem",
        SSL_FILETYPE_PEM) != SSL_SUCCESS) {
    fprintf(stderr, "Error loading ../certs/server-cert.pem, please"
        "check the file.\n");
    exit(EXIT_FAILURE);
}

/* Load keys */
if (wolfSSL_CTX_use_PrivateKey_file(ctx,"../certs/server-key.pem",
        SSL_FILETYPE_PEM) != SSL_SUCCESS) {
    fprintf(stderr, "Error loading ../certs/server-key.pem, please check"
        "the file.\n");
    exit(EXIT_FAILURE);
}
```

The code shown above should be added to the beginning of `tcpserv04.c` after the variable definitions in `main()`. A version of the finished code is included in the SSL tutorial ZIP file for reference.

Now that wolfSSL and the WOLFSSL_CTX have been initialized, make sure that the WOLFSSL_CTX object and the wolfSSL library are freed when the application is completely done using SSL/TLS. In both the client and the server, the following two lines should be placed at the end of the main() function (in the client right before the call to exit()):

```
wolfSSL_CTX_free(ctx);
wolfSSL_Cleanup();
```

11.11 WOLFSSL Object

11.11.1 EchoClient

A WOLFSSL object needs to be created after each TCP Connect and the socket file descriptor needs to be associated with the session. In the echoclient example, we will do this after the call to Connect(), shown below:

```
/* Connect to socket file descriptor */
Connect(sockfd, (SA *) &servaddr, sizeof(servaddr));
```

Directly after connecting, create a new WOLFSSL object using the wolfSSL_new() function. This function returns a pointer to the WOLFSSL object if successful or NULL in the case of failure. We can then associate the socket file descriptor (sockfd) with the new WOLFSSL object (ssl):

```
/* Create WOLFSSL object */
WOLFSSL* ssl;

if( (ssl = wolfSSL_new(ctx)) == NULL) {
    fprintf(stderr, "wolfSSL_new error.\n");
    exit(EXIT_FAILURE);
}

wolfSSL_set_fd(ssl, sockfd);
```

One thing to notice here is that we haven't made a call to the wolfSSL_connect() do it for us.

11.11.2 EchoServer

At the end of the for loop in the main method, insert the WOLFSSL object and associate the socket file descriptor (connfd) with the WOLFSSL object (ssl), just as with the client:

```
/* Create WOLFSSL object */
WOLFSSL* ssl;

if ( (ssl = wolfSSL_new(ctx)) == NULL) {
            fprintf(stderr, "wolfSSL_new error.\n");
            exit(EXIT_FAILURE);
}

wolfSSL_set_fd(ssl, connfd);
```

Again, a WOLFSSL object needs to be created after each TCP Connect and the socket file descriptor needs to be associated with the session.

11.12 Sending/Receiving Data

11.12.1 Sending with EchoClient

The next step is to begin sending data securely. Take note that in the echoclient example, the `main()` function hands off the sending and receiving work to `str_cli()`. The `str_cli()` function is where our function replacements will be made. First we need access to our `WOLFSSL` object in the `str_cli()` function, so we add another argument and pass the ssl variable to `str_cli()`. Because the `WOLFSSL` object is now going to be used inside of the `str_cli()` function, we remove the `sockfd` parameter. The new `str_cli()` function signature after this modification is shown below:

```
void str_cli(FILE *fp, WOLFSSL* ssl)
```

In the `main()` function, the new argument (`ssl`) is passed to `str_cli()`:

```
str_cli(stdin, ssl);
```

Inside the `str_cli()` function, `Writen()` and `Readline()` are replaced with calls to `wolfSSL_write()` were successful.

The authors of the Unix Programming book wrote error checking into their `Writen()` function which we must make up for after it has been replaced. We add a new int variable, n, to monitor the return value of `wolfSSL_read` and before printing out the contents of the buffer, `recvline`, the end of our read data is marked with a `\0`:

```
void
str_cli(FILE *fp, WOLFSSL* ssl)
{
    char  sendline[MAXLINE], recvline[MAXLINE];
    int   n = 0;

    while (Fgets(sendline, MAXLINE, fp) != NULL) {

        if(wolfSSL_write(ssl, sendline, strlen(sendline)) !=
                    strlen(sendline)){
            err_sys("wolfSSL_write failed");
        }

        if ((n = wolfSSL_read(ssl, recvline, MAXLINE)) <= 0)
            err_quit("wolfSSL_read error");

        recvline[n] = '\0';
        Fputs(recvline, stdout);
    }
}
```

The last thing to do is free the `WOLFSSL` object when we are completely done with it. In the `main()` function, right before the line to free the `WOLFSSL_CTX`, call to `wolfSSL_free()`:

```
str_cli(stdin, ssl);

wolfSSL_free(ssl);      /* Free WOLFSSL object */
wolfSSL_CTX_free(ctx);  /* Free WOLFSSL_CTX object */
wolfSSL_Cleanup();      /* Free wolfSSL */
```

11.12.2 Receiving with EchoServer

The echo server makes a call to `str_echo()` to handle reading and writing (whereas the client made a call to `str_cli()`). As with the client, modify `str_echo()` by replacing the sockfd parameter with

a WOLFSSL object (ssl) parameter in the function signature:

```
void str_echo(WOLFSSL* ssl)
```

Replace the calls to Read() and Writen() with calls to the wolfSSL_read():

```
void
str_echo(WOLFSSL* ssl)
{
    int n;
    char buf[MAXLINE];

    while ( (n = wolfSSL_read(ssl, buf, MAXLINE)) > 0) {
        if(wolfSSL_write(ssl, buf, n) != n) {
            err_sys("wolfSSL_write failed");
        }
    }

    if( n < 0 )
        printf("wolfSSL_read error = %d\n", wolfSSL_get_error(ssl,n));
    else if( n == 0 )
        printf("The peer has closed the connection.\n");
}
```

In main() call the str_echo() function at the end of the for loop (soon to be changed to a while loop). After this function, inside the loop, make calls to free the WOLFSSL object and close the connfd socket:

```
str_echo(ssl);                  /* process the request */

wolfSSL_free(ssl);              /* Free WOLFSSL object */
Close(connfd);
```

We will free the ctx and cleanup before the call to exit.

11.13 Signal Handling

11.13.1 Echoclient / Echoserver

In the echoclient and echoserver, we will need to add a signal handler for when the user closes the app by using "Ctrl+C". The echo server is continually running in a loop. Because of this, we need to provide a way to break that loop when the user presses "Ctrl+C". To do this, the first thing we need to do is change our loop to a while loop which terminates when an exit variable (cleanup) is set to true.

First, define a new static int variable called cleanup at the top of tcpserv04.c right after the #include statements:

```
static int cleanup;  /* To handle shutdown */
```

Modify the echoserver loop by changing it from a for loop to a while loop:

```
while(cleanup != 1)
{
    /* echo server code here */
}
```

For the echoserver we need to disable the operating system from restarting calls which were being executed before the signal was handled after our handler has finished. By disabling these, the operating system will not restart calls to accept() after the signal has been handled. If we didn't do this, we would have to wait for another client to connect and disconnect before the echoserver would clean

up resources and exit. To define the signal handler and turn off SA_RESTART, first define act and oact structures in the echoserver's main() function:

```
struct sigaction     act, oact;
```

Insert the following code after variable declarations, before the call to wolfSSL_Init() in the main function:

```
/* Signal handling code */
struct sigaction act, oact;         /* Declare the sigaction structs    */
act.sa_handler = sig_handler;       /* Tell act to use sig_handler      */
sigemptyset(&act.sa_mask);          /* Tells act to exclude all sa_mask *
                                     * signals during execution of      *
                                     * sig_handler.                     */
act.sa_flags = 0;                   /* States that act has a special    *
                                     * flag of 0                        */
sigaction(SIGINT, &act, &oact);     /* Tells the program to use (o)act  *
                                     * on a signal or interrupt         */
```

The echoserver's sig_handler function is shown below:

```
void sig_handler(const int sig)
{
    printf("\nSIGINT handled.\n");
    cleanup = 1;
    return;
}
```

That's it - the echoclient and echoserver are now enabled with TLSv1.2!!

What we did:

- Included the wolfSSL headers
- Initialized wolfSSL
- Created a WOLFSSL_CTX structure in which we chose what protocol we wanted to use
- Created a WOLFSSL object to use for sending and receiving data
- Replaced calls to Writen() and Readline() with wolfSSL_write()
- Freed WOLFSSL, WOLFSSL_CTX
- Made sure we handled client and server shutdown with signal handler

There are many more aspects and methods to configure and control the behavior of your SSL connections. For more detailed information, please see additional wolfSSL documentation and resources.

Once again, the completed source code can be found in the downloaded ZIP file at the top of this section.

11.14 Certificates

For testing purposes, you may use the certificates provided by wolfSSL. These can be found in the wolfSSL download, and specifically for this tutorial, they can be found in the finished_src folder.

For production applications, you should obtain correct and legitimate certificates from a trusted certificate authority.

11.15 Conclusion

This tutorial walked through the process of integrating the wolfSSL embedded SSL/TLS library into a simple client and server application. Although this example is simple, the same principles may be applied for adding SSL or TLS into your own application. The wolfSSL embedded SSL/TLS library provides

all the features you would need in a compact and efficient package that has been optimized for both size and speed.

Being dual licensed under GPLv2 and standard commercial licensing, you are free to download the wolfSSL source code directly from our website. Feel free to post to our support forums (https://www.wolfssl.com/forums) with any questions or comments you might have. If you would like more information about our products, please contact info@wolfssl.com.

We welcome any feedback you have on this SSL tutorial. If you believe it could be improved or enhanced in order to make it either more useful, easier to understand, or more portable, please let us know at support@wolfssl.com.

COPYRIGHT ©2024 wolfSSL Inc.

12 Best Practices for Embedded Devices

12.1 Creating Private Keys

Embedding a private key into firmware allows anyone to extract the key and turns an otherwise secure connection into something nothing more secure than TCP.

We have a few ideas about creating private keys for SSL enabled devices.

1. Each device acting as a server should have a unique private key, just like in the non-embedded world.
2. If the key can't be placed onto the device before delivery, have it generated during setup.
3. If the device lacks the power to generate its own key during setup, have the client setting up the device generate the key and send it to the device.
4. If the client lacks the ability to generate a private key, have the client retrieve a unique private key over an SSL/TLS connection from the devices known website (for example).

wolfSSL (formerly CyaSSL) can be used in all of these steps to help ensure an embedded device has a secure unique private key. Taking these steps will go a long way towards securing the SSL connection itself.

12.2 Digitally Signing and Authenticating with wolfSSL

wolfSSL is a popular tool for digitally signing applications, libraries, or files prior to loading them on embedded devices. Most desktop and server operating systems allow creation of this type of functionality through system libraries, but stripped down embedded operating systems do not. The reason that embedded RTOS environments do not include digital signature functionality is because it has historically not been a requirement for most embedded applications. In today's world of connected devices and heightened security concerns, digitally signing what is loaded onto your embedded or mobile device has become a top priority.

Examples of embedded connected devices where this requirement was not found in years past include set top boxes, DVR's, POS systems, both VoIP and mobile phones, connected home, and even automobile-based computing systems. Because wolfSSL supports the key embedded and real time operating systems, encryption standards, and authentication functionality, it is a natural choice for embedded systems developers to use when adding digital signature functionality.

Generally, the process for setting up code and file signing on an embedded device are as follows:

1. The embedded systems developer will generate an RSA key pair.
2. A server-side script-based tool is developed
    1. The server side tool will create a hash of the code to be loaded on the device (with SHA-256 for example).
    2. The hash is then digitally signed, also called RSA private encrypt.
    3. A package is created that contains the code along with the digital signature.
3. The package is loaded on the device along with a way to get the RSA public key. The hash is recreated on the device then digitally verified (also called RSA public decrypt) against the existing digital signature.

Benefits to enabling digital signatures on your device include:

1. Easily enable a secure method for allowing third parties to load files to your device.
2. Ensure against malicious files finding their way onto your device.
3. Digitally secure firmware updates
4. Ensure against firmware updates from unauthorized parties

General information on code signing:

https://en.wikipedia.org/wiki/Code_signing

COPYRIGHT ©2024 wolfSSL Inc.

13 OpenSSL Compatibility

13.1 Compatibility with OpenSSL

wolfSSL (formerly CyaSSL) provides an OpenSSL compatibility header, `wolfssl/openssl/ssl.h`, in addition to the wolfSSL native API, to ease the transition into using wolfSSL or to aid in porting an existing OpenSSL application over to wolfSSL. For an overview of the OpenSSL Compatibility Layer, please continue reading below. To view the complete set of OpenSSL functions supported by wolfSSL, please see the `wolfssl/openssl/ssl.h` file.

The OpenSSL Compatibility Layer maps a subset of the most commonly-used OpenSSL commands to wolfSSL's native API functions. This should allow for an easy replacement of OpenSSL by wolfSSL in your application or project without changing much code.

Our test beds for OpenSSL compatibility are stunnel and Lighttpd, which means that we build both of them with wolfSSL as a way to test our OpenSSL compatibility API.

Building wolfSSL With Compatibility Layer:

1. Enable with (`--enable-opensslextra`) or by defining the macro `OPENSSL_EXTRA`.

   ```
   ./configure --enable-opensslextra
   ```

2. Include <`wolfssl/options.h`> as first wolfSSL header
3. Header files for migration are located under:
   - `./wolfssl/openssl/*.h`
   - Ex: <`wolfssl/openssl/ssl.h`>

13.2 Differences Between wolfSSL and OpenSSL

Many people are curious how wolfSSL compares to OpenSSL and what benefits there are to using an SSL/TLS library that has been optimized to run on embedded platforms. Obviously, OpenSSL is free and presents no initial costs to begin using, but we believe that wolfSSL will provide you with more flexibility, an easier integration of SSL/TLS into your existing platform, current standards support, and much more – all provided under a very easy-to-use license model.

The points below outline several of the main differences between wolfSSL and OpenSSL.

1. With a 20-100 kB build size, wolfSSL is up to 20 times smaller than OpenSSL. wolfSSL is a better choice for resource constrained environments – where every byte matters.
2. wolfSSL is up to date with the most current standards of TLS 1.3 with DTLS. The wolfSSL team is dedicated to continually keeping wolfSSL up-to-date with current standards.
3. wolfSSL offers the best current ciphers and standards available today, including ciphers for streaming media support. In addition, the recently-introduced liboqs integration allows for you to start experimenting with post-quantum cryptography.
4. wolfSSL is dual licensed under both the GPLv2 as well as a commercial license, where OpenSSL is available only under their unique license from multiple sources.
5. wolfSSL is backed by an outstanding company who cares about its users and about their security, and is always willing to help. The team actively works to improve and expand wolfSSL. The wolfSSL team is based primarily out of Bozeman, MT, Portland, OR, and Seattle, WA, along with other team members located around the globe.
6. wolfSSL is the leading SSL/TLS library for real time, mobile, and embedded systems by virtue of its breadth of platform support and successful implementations on embedded environments. Chances are we've already been ported to your environment. If not, let us know and we'll be glad to help.

COPYRIGHT ©2024 wolfSSL Inc.

7. wolfSSL offers several abstraction layers to make integrating SSL into your environment and platform as easy as possible. With an OS layer, a custom I/O layer, and a C Standard Library abstraction layer, integration has never been so easy.
8. wolfSSL offers several support packages for wolfSSL. Available directly through phone, email or the wolfSSL product support forums, your questions are answered quickly and accurately to help you make progress on your project as quickly as possible.

13.3 Supported OpenSSL Structures

- SSL_METHOD holds SSL version information and is either a client or server method. (Same as WOLFSSL_METHOD in the native wolfSSL API).
- SSL_CTX holds context information including certificates. (Same as WOLFSSL_CTX in the native wolfSSL API).
- SSL holds session information for a secure connection. (Same as WOLFSSL in the native wolfSSL API).

13.4 Supported OpenSSL Functions

The three structures shown above are usually initialized in the following way:

```
SSL_METHOD* method = SSLv3_client_method();
SSL_CTX* ctx = SSL_CTX_new(method);
SSL* ssl = SSL_new(ctx);
```

This establishes a client side SSL version 3 method, creates a context based on the method, and initializes the SSL session with the context. A server side program is no different except that the SSL_METHOD is created using SSLv3_server_method(), or one of the available functions. For a list of supported functions, please see the Protocol Support section. When using the OpenSSL Compatibility layer, the functions in this section should be modified by removing the "wolf" prefix. For example, the native wolfSSL API function:

```
wolfTLSv1_client_method()
```

Becomes:

```
TLSv1_client_method()
```

When an SSL connection is no longer needed the following calls free the structures created during initialization.

```
SSL_CTX_free(ctx);
SSL_free(ssl);
```

SSL_CTX_free() has the additional responsibility of freeing the associated SSL_METHOD. Failing to use the XXX_free() functions will result in a resource leak. Using the system's free() instead of the SSL ones results in undefined behavior.

Once an application has a valid SSL pointer from SSL_new(), the SSL handshake process can begin. From the client's view, SSL_connect() will attempt to establish a secure connection.

```
SSL_set_fd(ssl, sockfd);
SSL_connect(ssl);
```

Before the SSL_connect() can be issued, the user must supply wolfSSL with a valid socket file descriptor, sockfd in the example above. sockfd is typically the result of the TCP function socket() which is later established using TCP connect(). The following creates a valid client side socket descriptor for use with a local wolfSSL server on port 11111, error handling is omitted for simplicity.

```
int sockfd = socket(AF_INET, SOCK_STREAM, 0);
sockaddr_in servaddr;
```

```
memset(&servaddr, 0, sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_port = htons(11111);
servaddr.sin_addr.s_addr = inet_addr("127.0.0.1");
connect(sockfd, (const sockaddr*)&servaddr, sizeof(servaddr));
```

Once a connection is established, the client may read and write to the server. Instead of using the TCP functions `send()` and `receive()`, wolfSSL and yaSSL use the SSL functions `SSL_write()` and `SSL_read()`. Here is a simple example from the client demo:

```
char msg[] = "hello wolfssl!";
int wrote = SSL_write(ssl, msg, sizeof(msg));
char reply[1024];
int read = SSL_read(ssl, reply, sizeof(reply));
reply[read] = 0;
printf("Server response: %s\n", reply);
```

The server connects in the same way, except that it uses `SSL_accept()` instead of `SSL_connect()`, analogous to the TCP API. See the server example for a complete server demo program.

13.5 x509 Certificates

Both the server and client can provide wolfSSL with certificates in either **PEM** or **DER**.

Typical usage is like this:

```
SSL_CTX_use_certificate_file(ctx, "certs/cert.pem",
SSL_FILETYPE_PEM);
SSL_CTX_use_PrivateKey_file(ctx, "certs/key.der",
SSL_FILETYPE_ASN1);
```

A key file can also be presented to the Context in either format. `SSL_FILETYPE_PEM` signifies the file is PEM formatted while `SSL_FILETYPE_ASN1` declares the file to be in DER format. To verify that the key file is appropriate for use with the certificate the following function can be used:

```
SSL_CTX_check_private_key(ctx);
```

14 Licensing

14.1 Open Source

wolfSSL, wolfCrypt, wolfMQTT, wolfTPM, wolfBoot, and wolfSentry are free software downloads and may be modified to the needs of the user as long as the user adheres to version two of the GPL License. The GPLv2 license can be found on the gnu.org website https://www.gnu.org/licenses/old-licenses/gpl-2.0.html.

wolfSSH software is a free software download and may be modified to the needs of the user as long as the user adheres to version three of the GPL license. The GPLv3 license can be found on the gnu.org website (https://www.gnu.org/licenses/gpl.html).

14.2 Commercial Licensing

Businesses and enterprises who wish to incorporate wolfSSL products into proprietary appliances or other commercial software products for re-distribution must license commercial versions. Commercial licenses for wolfSSL and wolfCrypt are available per end product or SKU. Licenses are generally issued for one product and include unlimited royalty-free distribution. Custom licensing terms are also available.

Commercial licenses are also available for wolfMQTT, wolfSSH, wolfTPM, wolfBoot, and wolfSentry. Please contact licensing@wolfssl.com with inquiries.

14.3 FIPS 140-2/3 Validation

wolfSSL is currently the leader in embedded FIPS certificates. For details on currently-active FIPS certificates and validation options, please see the wolfCrypt FIPS FAQ or contact fips@wolfssl.com.

14.4 Support Packages

Support packages for wolfSSL products are available on an annual basis directly from wolfSSL. With four different package options, you can compare them side-by-side and choose the package that best fits your specific needs. Please see our Support Packages page (https://www.wolfssl.com/products/support-and-maintenance) for more details.

COPYRIGHT ©2024 wolfSSL Inc.

15 Support and Consulting

15.1 How to Get Support

For general product support, wolfSSL (formerly CyaSSL) maintains an online forum for the wolfSSL product family. Please post to the forums or contact wolfSSL directly with any questions.

- wolfSSL (yaSSL) Forums: https://www.wolfssl.com/forums
- Email Support: support@wolfssl.com

For information regarding wolfSSL products, questions regarding licensing, or general comments, please contact wolfSSL by emailing info@wolfssl.com. For support packages, please see Licensing.

15.1.1 Bugs Reports and Support Issues

If you are submitting a bug report or asking about a problem, please include the following information with your submission:

1. wolfSSL version number
2. Operating System version
3. Compiler version
4. The exact error you are seeing
5. A description of how we can reproduce or try to replicate this problem

With the above information, we will do our best to resolve your problems. Without this information, it is very hard to pinpoint the source of the problem. wolfSSL values your feedback and makes it a top priority to get back to you as soon as possible.

15.2 Consulting

wolfSSL offers both on and off site consulting - providing feature additions, porting, a Competitive Upgrade Program (see Competitive Upgrade Program), and design consulting.

15.2.1 Feature Additions and Porting

We can add additional features that you may need which are not currently offered in our products on a contract or co-development basis. We also offer porting services on our products to new host languages or new operating environments.

15.2.2 Competitive Upgrade Program

We will help you move from an outdated or expensive SSL/TLS library to wolfSSL with low cost and minimal disturbance to your code base.

Program Outline:

1. You need to currently be using a commercial competitor to wolfSSL.
2. You will receive up to one week of on-site consulting to switch out your old SSL library for wolfSSL. Travel expenses are not included.
3. Normally, up to one week is the right amount of time for us to make the replacement in your code and do initial testing. Additional consulting on a replacement is available as needed.
4. You will receive the standard wolfSSL royalty free license to ship with your product.

The purpose of this program is to enable users who are currently spending too much on their embedded SSL implementation to move to wolfSSL with ease. If you are interested in learning more, then please contact us at info@wolfssl.com.

COPYRIGHT ©2024 wolfSSL Inc.

15.2.3 Design Consulting

If your application or framework needs to be secured with SSL/TLS but you are uncertain about how the optimal design of a secured system would be structured, we can help!

We offer design consulting for building SSL/TLS security into devices using wolfSSL. Our consultants can provide you with the following services:

1. *Assessment*: An evaluation of your current SSL/TLS implementation. We can give you advice on your current setup and how we think you could improve upon this by using wolfSSL.
2. *Design*: Looking at your system requirements and parameters, we'll work closely with you to make recommendations on how to implement wolfSSL into your application such that it provides you with optimal security.

If you would like to learn more about design consulting for building SSL into your application or device, please contact info@wolfssl.com for more information.

16 wolfSSL (formerly CyaSSL) Updates

16.1 Product Release Information

We regularly post update information on Twitter. For additional release information, you can keep track of our projects on GitHub, follow us on Facebook, or follow our daily blog.

- wolfSSL on GitHub - https://www.github.com/wolfssl/wolfssl
- wolfSSL on Twitter - https://twitter.com/wolfSSL
- wolfSSL on Facebook - https://www.facebook.com/wolfSSL
- wolfSSL on Reddit - https://www.reddit.com/r/wolfssl/
- Daily Blog - https://www.wolfssl.com/blog

COPYRIGHT ©2024 wolfSSL Inc.

A wolfSSL API Reference

A.1 CertManager API

A.1.1 Functions

	Name
WOLFSSL_CERT_MANAGER *	**wolfSSL_CertManagerNew_ex**(void * heap)Allocates and initializes a new Certificate Manager context. This context may be used independent of SSL needs. It may be used to load certificates, verify certificates, and check the revocation status.
WOLFSSL_CERT_MANAGER *	**wolfSSL_CertManagerNew**(void)Allocates and initializes a new Certificate Manager context. This context may be used independent of SSL needs. It may be used to load certificates, verify certificates, and check the revocation status.
void	**wolfSSL_CertManagerFree**(WOLFSSL_CERT_MANAGER *)Frees all resources associated with the Certificate Manager context. Call this when you no longer need to use the Certificate Manager.
int	**wolfSSL_CertManagerLoadCA**(WOLFSSL_CERT_MANAGER * cm, const char * f, const char * d)Specifies the locations for CA certificate loading into the manager context. The PEM certificate CAfile may contain several trusted CA certificates. If CApath is not NULL it specifies a directory containing CA certificates in PEM format.
int	**wolfSSL_CertManagerLoadCABuffer**(WOLFSSL_CERT_MANAGER * cm, const unsigned char * in, long sz, int format)Loads the CA Buffer by calling wolfSSL_CTX_load_verify_buffer and returning that result using a temporary cm so as not to lose the information in the cm passed into the function.
int	**wolfSSL_CertManagerUnloadCAs**(WOLFSSL_CERT_MANAGER * cm)This function unloads the CA signer list.
int	**wolfSSL_CertManagerUnloadIntermediateCerts**(WOLFSSL_CE * cm)This function unloads intermediate certificates add to the CA signer list.
int	**wolfSSL_CertManagerUnload_trust_peers**(WOLFSSL_CERT_MA * cm)The function will free the Trusted Peer linked list and unlocks the trusted peer list.
int	**wolfSSL_CertManagerVerify**(WOLFSSL_CERT_MANAGER * cm, const char * f, int format)Specifies the certificate to verify with the Certificate Manager context. The format can be SSL_FILETYPE_PEM or SSL_FILETYPE_ASN1.

	Name
int	**wolfSSL_CertManagerVerifyBuffer**(WOLFSSL_CERT_MANAGER * cm, const unsigned char * buff, long sz, int format)Specifies the certificate buffer to verify with the Certificate Manager context. The format can be SSL_FILETYPE_PEM or SSL_FILETYPE_ASN1.
void	**wolfSSL_CertManagerSetVerify**(WOLFSSL_CERT_MANAGER * cm, VerifyCallback vc)The function sets the verifyCallback function in the Certificate Manager. If present, it will be called for each cert loaded. If there is a verification error, the verify callback can be used to over-ride the error.
int	**wolfSSL_CertManagerEnableCRL**(WOLFSSL_CERT_MANAGER * cm, int options)Turns on Certificate Revocation List checking when verifying certificates with the Certificate Manager. By default, CRL checking is off. options include WOLFSSL_CRL_CHECKALL which performs CRL checking on each certificate in the chain versus the Leaf certificate only which is the default.
int	**wolfSSL_CertManagerDisableCRL**(WOLFSSL_CERT_MANAGER *)Turns off Certificate Revocation List checking when verifying certificates with the Certificate Manager. By default, CRL checking is off. You can use this function to temporarily or permanently disable CRL checking with this Certificate Manager context that previously had CRL checking enabled.
int	**wolfSSL_CertManagerLoadCRL**(WOLFSSL_CERT_MANAGER * cm, const char * path, int type, int monitor)Error checks and passes through to LoadCRL() in order to load the cert into the CRL for revocation checking. An updated CRL can be loaded by first calling wolfSSL_CertManagerFreeCRL, then loading the new CRL.
int	**wolfSSL_CertManagerLoadCRLBuffer**(WOLFSSL_CERT_MANAGE * cm, const unsigned char * buff, long sz, int type)The function loads the CRL file by calling BufferLoadCRL.
int	**wolfSSL_CertManagerSetCRL_Cb**(WOLFSSL_CERT_MANAGER * cm, CbMissingCRL cb)This function sets the CRL Certificate Manager callback. If HAVE_CRL is defined and a matching CRL record is not found then the cbMissingCRL is called (set via wolfSSL_CertManagerSetCRL_Cb). This allows you to externally retrieve the CRL and load it.

	Name
int	**wolfSSL_CertManagerSetCRLUpdate_Cb**(WOLFSSL_CERT_MANA * cm, CbUpdateCRL cb)This function sets the CRL Update callback. If HAVE_CRL and HAVE_CRL_UPDATE_CB is defined , and an entry with the same issuer and a lower CRL number exists when a CRL is added, then the CbUpdateCRL is called with the details of the existing entry and the new one replacing it.
int	**wolfSSL_CertManagerGetCRLInfo**(WOLFSSL_CERT_MANAGER * cm, CrlInfo * info, const byte * buff, long sz, int type)This function yields a structure with parsed CRL information from an encoded CRL buffer.
int	**wolfSSL_CertManagerCheckOCSP**(WOLFSSL_CERT_MANAGER * cm, unsigned char * der, int sz)The function enables the WOLFSSL_CERT_MANAGER's member, ocspEnabled to signify that the OCSP check option is enabled.
int	**wolfSSL_CertManagerEnableOCSP**(WOLFSSL_CERT_MANAGER * cm, int options)Turns on OCSP if it's turned off and if compiled with the set option available.
int	**wolfSSL_CertManagerDisableOCSP**(WOLFSSL_CERT_MANAGER *)Disables OCSP certificate revocation.
int	**wolfSSL_CertManagerSetOCSPOverrideURL**(WOLFSSL_CERT_M * cm, const char * url)The function copies the url to the ocspOverrideURL member of the WOLFSSL_CERT_MANAGER structure.
int	**wolfSSL_CertManagerSetOCSP_Cb**(WOLFSSL_CERT_MANAGER * cm, CbOCSPIO ioCb, CbOCSPRespFree respFreeCb, void * ioCbCtx)The function sets the OCSP callback in the WOLFSSL_CERT_MANAGER.
int	**wolfSSL_CertManagerEnableOCSPStapling**(WOLFSSL_CERT_MA * cm)This function turns on OCSP stapling if it is not turned on as well as set the options.

A.1.2 Functions Documentation

```
WOLFSSL_CERT_MANAGER * wolfSSL_CertManagerNew_ex(
    void * heap
)
```

Allocates and initializes a new Certificate Manager context. This context may be used independent of SSL needs. It may be used to load certificates, verify certificates, and check the revocation status.

Parameters:

- **none** No parameters.

See: wolfSSL_CertManagerFree

Return:

- WOLFSSL_CERT_MANAGER If successful the call will return a valid WOLFSSL_CERT_MANAGER pointer.
- NULL will be returned for an error state.

```
WOLFSSL_CERT_MANAGER * wolfSSL_CertManagerNew(
    void
)
```

Allocates and initializes a new Certificate Manager context. This context may be used independent of SSL needs. It may be used to load certificates, verify certificates, and check the revocation status.

Parameters:

- **none** No parameters.

See: wolfSSL_CertManagerFree

Return:

- WOLFSSL_CERT_MANAGER If successful the call will return a valid WOLFSSL_CERT_MANAGER pointer.
- NULL will be returned for an error state.

Example

```
#import <wolfssl/ssl.h>

WOLFSSL_CERT_MANAGER* cm;
cm = wolfSSL_CertManagerNew();
if (cm == NULL) {
// error creating new cert manager
}
```

```
void wolfSSL_CertManagerFree(
    WOLFSSL_CERT_MANAGER *
)
```

Frees all resources associated with the Certificate Manager context. Call this when you no longer need to use the Certificate Manager.

Parameters:

- **cm** a pointer to a WOLFSSL_CERT_MANAGER structure, created using wolfSSL_CertManagerNew().

See: wolfSSL_CertManagerNew

Return: none

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_CERT_MANAGER* cm;
...
wolfSSL_CertManagerFree(cm);
```

```
int wolfSSL_CertManagerLoadCA(
    WOLFSSL_CERT_MANAGER * cm,
    const char * f,
    const char * d
)
```

Specifies the locations for CA certificate loading into the manager context. The PEM certificate CAfile may contain several trusted CA certificates. If CApath is not NULL it specifies a directory containing CA certificates in PEM format.

Parameters:

- **cm** a pointer to a WOLFSSL_CERT_MANAGER structure, created using wolfSSL_CertManagerNew().
- **file** pointer to the name of the file containing CA certificates to load.
- **path** pointer to the name of a directory path containing CA c ertificates to load. The NULL pointer may be used if no certificate directory is desired.

See: wolfSSL_CertManagerVerify

Return:

- SSL_SUCCESS If successful the call will return.
- SSL_BAD_FILETYPE will be returned if the file is the wrong format.
- SSL_BAD_FILE will be returned if the file doesn't exist, can't be read, or is corrupted.
- MEMORY_E will be returned if an out of memory condition occurs.
- ASN_INPUT_E will be returned if Base16 decoding fails on the file.
- BAD_FUNC_ARG is the error that will be returned if a pointer is not provided.
- SSL_FATAL_ERROR - will be returned upon failure.

Example

```
#include <wolfssl/ssl.h>

int ret = 0;
WOLFSSL_CERT_MANAGER* cm;
...
ret = wolfSSL_CertManagerLoadCA(cm, “path/to/cert-file.pem”, 0);
if (ret != SSL_SUCCESS) {
// error loading CA certs into cert manager
}
```

```
int wolfSSL_CertManagerLoadCABuffer(
    WOLFSSL_CERT_MANAGER * cm,
    const unsigned char * in,
    long sz,
    int format
)
```

Loads the CA Buffer by calling wolfSSL_CTX_load_verify_buffer and returning that result using a temporary cm so as not to lose the information in the cm passed into the function.

Parameters:

- **cm** a pointer to a WOLFSSL_CERT_MANAGER structure, created using wolfSSL_CertManagerNew().
- **in** buffer for cert information.
- **sz** length of the buffer.
- **format** certificate format, either PEM or DER.

See:

- wolfSSL_CTX_load_verify_buffer
- ProcessChainBuffer
- ProcessBuffer
- cm_pick_method

Return:

- SSL_FATAL_ERROR is returned if the WOLFSSL_CERT_MANAGER struct is NULL or if wolfSSL_CTX_new() returns NULL.
- SSL_SUCCESS is returned for a successful execution.

Example

```
WOLFSSL_CERT_MANAGER* cm = (WOLFSSL_CERT_MANAGER*)vp;
…
const unsigned char* in;
long sz;
int format;
…
if(wolfSSL_CertManagerLoadCABuffer(vp, sz, format) != SSL_SUCCESS){
    Error returned. Failure case code block.
}
```

```
int wolfSSL_CertManagerUnloadCAs(
    WOLFSSL_CERT_MANAGER * cm
)
```

This function unloads the CA signer list.

Parameters:

- **cm** a pointer to a WOLFSSL_CERT_MANAGER structure, created using wolfSSL_CertManagerNew().

See: UnlockMutex

Return:

- SSL_SUCCESS returned on successful execution of the function.
- BAD_FUNC_ARG returned if the WOLFSSL_CERT_MANAGER is NULL.
- BAD_MUTEX_E returned if there was a mutex error.

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_CTX* ctx = wolfSSL_CTX_new(protocol method);
WOLFSSL_CERT_MANAGER* cm = wolfSSL_CTX_GetCertManager(ctx);
...
if(wolfSSL_CertManagerUnloadCAs(cm) != SSL_SUCCESS){
    Failure case.
}
```

```
int wolfSSL_CertManagerUnloadIntermediateCerts(
    WOLFSSL_CERT_MANAGER * cm
)
```

This function unloads intermediate certificates add to the CA signer list.

Parameters:

- **cm** a pointer to a WOLFSSL_CERT_MANAGER structure, created using wolfSSL_CertManagerNew().

See: UnlockMutex

Return:

- SSL_SUCCESS returned on successful execution of the function.
- BAD_FUNC_ARG returned if the WOLFSSL_CERT_MANAGER is NULL.
- BAD_MUTEX_E returned if there was a mutex error.

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_CTX* ctx = wolfSSL_CTX_new(protocol method);
WOLFSSL_CERT_MANAGER* cm = wolfSSL_CTX_GetCertManager(ctx);
...
if(wolfSSL_CertManagerUnloadIntermediateCerts(cm) != SSL_SUCCESS){
    Failure case.
}
```

```
int wolfSSL_CertManagerUnload_trust_peers(
    WOLFSSL_CERT_MANAGER * cm
)
```

The function will free the Trusted Peer linked list and unlocks the trusted peer list.

Parameters:

- **cm** a pointer to a WOLFSSL_CERT_MANAGER structure, created using wolfSSL_CertManagerNew().

See: UnLockMutex

Return:

- SSL_SUCCESS if the function completed normally.
- BAD_FUNC_ARG if the WOLFSSL_CERT_MANAGER is NULL.
- BAD_MUTEX_E mutex error if tpLock, a member of the WOLFSSL_CERT_MANAGER struct, is 0 (nill).

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_CTX* ctx = WOLFSSL_CTX_new(Protocol define);
WOLFSSL_CERT_MANAGER* cm = wolfSSL_CertManagerNew();
...
if(wolfSSL_CertManagerUnload_trust_peers(cm) != SSL_SUCCESS){
    The function did not execute successfully.
}
```

```
int wolfSSL_CertManagerVerify(
    WOLFSSL_CERT_MANAGER * cm,
    const char * f,
    int format
)
```

Specifies the certificate to verify with the Certificate Manager context. The format can be SSL_FILETYPE_PEM or SSL_FILETYPE_ASN1.

Parameters:

- **cm** a pointer to a WOLFSSL_CERT_MANAGER structure, created using wolfSSL_CertManagerNew().
- **fname** pointer to the name of the file containing the certificates to verify.
- **format** format of the certificate to verify - either SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM.

See:

- wolfSSL_CertManagerLoadCA
- wolfSSL_CertManagerVerifyBuffer

Return:

- SSL_SUCCESS If successful.
- ASN_SIG_CONFIRM_E will be returned if the signature could not be verified.
- ASN_SIG_OID_E will be returned if the signature type is not supported.
- CRL_CERT_REVOKED is an error that is returned if this certificate has been revoked.
- CRL_MISSING is an error that is returned if a current issuer CRL is not available.
- ASN_BEFORE_DATE_E will be returned if the current date is before the before date.
- ASN_AFTER_DATE_E will be returned if the current date is after the after date.
- SSL_BAD_FILETYPE will be returned if the file is the wrong format.
- SSL_BAD_FILE will be returned if the file doesn't exist, can't be read, or is corrupted.
- MEMORY_E will be returned if an out of memory condition occurs.
- ASN_INPUT_E will be returned if Base16 decoding fails on the file.
- BAD_FUNC_ARG is the error that will be returned if a pointer is not provided.

Example

```
int ret = 0;
WOLFSSL_CERT_MANAGER* cm;
...

ret = wolfSSL_CertManagerVerify(cm, “path/to/cert-file.pem”,
SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
    error verifying certificate
}
```

```
int wolfSSL_CertManagerVerifyBuffer(
    WOLFSSL_CERT_MANAGER * cm,
    const unsigned char * buff,
    long sz,
    int format
)
```

Specifies the certificate buffer to verify with the Certificate Manager context. The format can be SSL_FILETYPE_PEM or SSL_FILETYPE_ASN1.

Parameters:

- **cm** a pointer to a WOLFSSL_CERT_MANAGER structure, created using wolfSSL_CertManagerNew().
- **buff** buffer containing the certificates to verify.
- **sz** size of the buffer, buf.
- **format** format of the certificate to verify, located in buf - either SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM.

See:

- wolfSSL_CertManagerLoadCA
- wolfSSL_CertManagerVerify

Return:

- SSL_SUCCESS If successful.
- ASN_SIG_CONFIRM_E will be returned if the signature could not be verified.
- ASN_SIG_OID_E will be returned if the signature type is not supported.
- CRL_CERT_REVOKED is an error that is returned if this certificate has been revoked.
- CRL_MISSING is an error that is returned if a current issuer CRL is not available.
- ASN_BEFORE_DATE_E will be returned if the current date is before the before date.
- ASN_AFTER_DATE_E will be returned if the current date is after the after date.
- SSL_BAD_FILETYPE will be returned if the file is the wrong format.
- SSL_BAD_FILE will be returned if the file doesn't exist, can't be read, or is corrupted.
- MEMORY_E will be returned if an out of memory condition occurs.
- ASN_INPUT_E will be returned if Base16 decoding fails on the file.
- BAD_FUNC_ARG is the error that will be returned if a pointer is not provided.

Example

```
#include <wolfssl/ssl.h>

int ret = 0;
int sz = 0;
WOLFSSL_CERT_MANAGER* cm;
byte certBuff[...];
...

ret = wolfSSL_CertManagerVerifyBuffer(cm, certBuff, sz, SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
    error verifying certificate
}
```

```
void wolfSSL_CertManagerSetVerify(
    WOLFSSL_CERT_MANAGER * cm,
    VerifyCallback vc
)
```

The function sets the verifyCallback function in the Certificate Manager. If present, it will be called for each cert loaded. If there is a verification error, the verify callback can be used to over-ride the error.

Parameters:

- **cm** a pointer to a WOLFSSL_CERT_MANAGER structure, created using wolfSSL_CertManagerNew().
- **vc** a VerifyCallback function pointer to the callback routine

See: wolfSSL_CertManagerVerify

Return: none No return.

Example

```
#include <wolfssl/ssl.h>

int myVerify(int preverify, WOLFSSL_X509_STORE_CTX* store)
{ // do custom verification of certificate }
```

```
WOLFSSL_CTX* ctx = WOLFSSL_CTX_new(Protocol define);
WOLFSSL_CERT_MANAGER* cm = wolfSSL_CertManagerNew();
...
wolfSSL_CertManagerSetVerify(cm, myVerify);
```

```
int wolfSSL_CertManagerEnableCRL(
    WOLFSSL_CERT_MANAGER * cm,
    int options
)
```

Turns on Certificate Revocation List checking when verifying certificates with the Certificate Manager. By default, CRL checking is off. options include WOLFSSL_CRL_CHECKALL which performs CRL checking on each certificate in the chain versus the Leaf certificate only which is the default.

Parameters:

- **cm** a pointer to a WOLFSSL_CERT_MANAGER structure, created using wolfSSL_CertManagerNew().
- **options** options to use when enabling the Certification Manager, cm.

See: wolfSSL_CertManagerDisableCRL

Return:

- SSL_SUCCESS If successful the call will return.
- NOT_COMPILED_IN will be returned if wolfSSL was not built with CRL enabled.
- MEMORY_E will be returned if an out of memory condition occurs.
- BAD_FUNC_ARG is the error that will be returned if a pointer is not provided.
- SSL_FAILURE will be returned if the CRL context cannot be initialized properly.

Example

```
#include <wolfssl/ssl.h>

int ret = 0;
WOLFSSL_CERT_MANAGER* cm;
...

ret = wolfSSL_CertManagerEnableCRL(cm, 0);
if (ret != SSL_SUCCESS) {
    error enabling cert manager
}

...
```

```
int wolfSSL_CertManagerDisableCRL(
    WOLFSSL_CERT_MANAGER *
)
```

Turns off Certificate Revocation List checking when verifying certificates with the Certificate Manager. By default, CRL checking is off. You can use this function to temporarily or permanently disable CRL checking with this Certificate Manager context that previously had CRL checking enabled.

Parameters:

- **cm** a pointer to a WOLFSSL_CERT_MANAGER structure, created using wolfSSL_CertManagerNew().

See: wolfSSL_CertManagerEnableCRL

Return:

- SSL_SUCCESS If successful the call will return.
- BAD_FUNC_ARG is the error that will be returned if a function pointer is not provided.

Example

```
#include <wolfssl/ssl.h>

int ret = 0;
WOLFSSL_CERT_MANAGER* cm;
...
ret = wolfSSL_CertManagerDisableCRL(cm);
if (ret != SSL_SUCCESS) {
    error disabling cert manager
}
...
```

```
int wolfSSL_CertManagerLoadCRL(
    WOLFSSL_CERT_MANAGER * cm,
    const char * path,
    int type,
    int monitor
)
```

Error checks and passes through to LoadCRL() in order to load the cert into the CRL for revocation checking. An updated CRL can be loaded by first calling wolfSSL_CertManagerFreeCRL, then loading the new CRL.

Parameters:

- **cm** a pointer to a WOLFSSL_CERT_MANAGER structure, created using wolfSSL_CertManagerNew().
- **path** a constant char pointer holding the CRL path.
- **type** type of certificate to be loaded.
- **monitor** requests monitoring in LoadCRL().

See:

- wolfSSL_CertManagerEnableCRL
- wolfSSL_LoadCRL

- wolfSSL_CertManagerFreeCRL

Return:

- SSL_SUCCESS if there is no error in wolfSSL_CertManagerLoadCRL and if LoadCRL returns successfully.
- BAD_FUNC_ARG if the WOLFSSL_CERT_MANAGER struct is NULL.
- SSL_FATAL_ERROR if wolfSSL_CertManagerEnableCRL returns anything other than SSL_SUCCESS.
- BAD_PATH_ERROR if the path is NULL.
- MEMORY_E if LoadCRL fails to allocate heap memory.

Example

```
#include <wolfssl/ssl.h>

int wolfSSL_LoadCRL(WOLFSSL* ssl, const char* path, int type,
int monitor);
…
wolfSSL_CertManagerLoadCRL(SSL_CM(ssl), path, type, monitor);
```

```
int wolfSSL_CertManagerLoadCRLBuffer(
    WOLFSSL_CERT_MANAGER * cm,
    const unsigned char * buff,
    long sz,
    int type
)
```

The function loads the CRL file by calling BufferLoadCRL.

Parameters:

- **cm** a pointer to a WOLFSSL_CERT_MANAGER structure.
- **buff** a constant byte type and is the buffer.
- **sz** a long int representing the size of the buffer.
- **type** a long integer that holds the certificate type.

See:

- BufferLoadCRL
- wolfSSL_CertManagerEnableCRL

Return:

- SSL_SUCCESS returned if the function completed without errors.
- BAD_FUNC_ARG returned if the WOLFSSL_CERT_MANAGER is NULL.
- SSL_FATAL_ERROR returned if there is an error associated with the WOLFSSL_CERT_MANAGER.

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_CERT_MANAGER* cm;
const unsigned char* buff;
long sz; size of buffer
int type;  cert type
...
int ret = wolfSSL_CertManagerLoadCRLBuffer(cm, buff, sz, type);
if(ret == SSL_SUCCESS){
return ret;
} else {
    Failure case.
}
```

```
int wolfSSL_CertManagerSetCRL_Cb(
    WOLFSSL_CERT_MANAGER * cm,
    CbMissingCRL cb
)
```

This function sets the CRL Certificate Manager callback. If HAVE_CRL is defined and a matching CRL record is not found then the cbMissingCRL is called (set via wolfSSL_CertManagerSetCRL_Cb). This allows you to externally retrieve the CRL and load it.

Parameters:

- **cm** the WOLFSSL_CERT_MANAGER structure holding the information for the certificate.
- **cb** a function pointer to (*CbMissingCRL) that is set to the cbMissingCRL member of the WOLFSSL_CERT_MANAGER.

See:

- CbMissingCRL
- wolfSSL_SetCRL_Cb

Return:

- SSL_SUCCESS returned upon successful execution of the function and subroutines.
- BAD_FUNC_ARG returned if the WOLFSSL_CERT_MANAGER structure is NULL.

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_CTX* ctx = wolfSSL_CTX_new(protocol method);
WOLFSSL* ssl = wolfSSL_new(ctx);
…
void cb(const char* url){
    Function body.
}
…
```

```
CbMissingCRL cb = CbMissingCRL;
…
if(ctx){
    return wolfSSL_CertManagerSetCRL_Cb(SSL_CM(ssl), cb);
}
```

```
int wolfSSL_CertManagerSetCRLUpdate_Cb(
    WOLFSSL_CERT_MANAGER * cm,
    CbUpdateCRL cb
)
```

This function sets the CRL Update callback. If HAVE_CRL and HAVE_CRL_UPDATE_CB is defined , and an entry with the same issuer and a lower CRL number exists when a CRL is added, then the CbUpdateCRL is called with the details of the existing entry and the new one replacing it.

Parameters:

- **cm** the WOLFSSL_CERT_MANAGER structure holding the information for the certificate.
- **cb** a function pointer to (*CbUpdateCRL) that is set to the cbUpdateCRL member of the WOLFSSL_CERT_MANAGER. Signature requirement: void (*CbUpdateCRL)(CrlInfo *old, CrlInfo* new);

See: CbUpdateCRL

Return:

- SSL_SUCCESS returned upon successful execution of the function and subroutines.
- BAD_FUNC_ARG returned if the WOLFSSL_CERT_MANAGER structure is NULL.

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_CTX* ctx = wolfSSL_CTX_new(protocol method);
WOLFSSL* ssl = wolfSSL_new(ctx);
…
void cb(CrlInfo *old, CrlInfo *new){
    Function body.
}
…
CbUpdateCRL cb = CbUpdateCRL;
…
if(ctx){
    return wolfSSL_CertManagerSetCRLUpdate_Cb(SSL_CM(ssl), cb);
}
```

```
int wolfSSL_CertManagerGetCRLInfo(
    WOLFSSL_CERT_MANAGER * cm,
    CrlInfo * info,
    const byte * buff,
```

```
    long sz,
    int type
)
```

This function yields a structure with parsed CRL information from an encoded CRL buffer.

Parameters:

- **cm** the WOLFSSL_CERT_MANAGER structure..
- **info** pointer to caller managed CrlInfo structure that will receive the CRL information.
- **buff** input buffer containing encoded CRL.
- **sz** the length in bytes of the input CRL data in buff.
- **type** WOLFSSL_FILETYPE_PEM or WOLFSSL_FILETYPE_DER
- **cm** a pointer to a WOLFSSL_CERT_MANAGER structure, created using wolfSSL_CertManagerNew().

See:

- CbUpdateCRL
- wolfSSL_SetCRL_Cb
- wolfSSL_CertManagerLoadCRL

Return:

- SSL_SUCCESS returned upon successful execution of the function and subroutines.
- BAD_FUNC_ARG returned if the WOLFSSL_CERT_MANAGER structure is NULL.
- SSL_SUCCESS returned upon successful execution of the function and subroutines.
- BAD_FUNC_ARG returned if the WOLFSSL_CERT_MANAGER structure is NULL.

Example

```
#include <wolfssl/ssl.h>

CrlInfo info;
WOLFSSL_CERT_MANAGER* cm = NULL;

cm = wolfSSL_CertManagerNew();

// Read crl data from file into buffer

wolfSSL_CertManagerGetCRLInfo(cm, &info, crlData, crlDataLen,
                              WOLFSSL_FILETYPE_PEM);
```

This function frees the CRL stored in the Cert Manager. An application can update the CRL by calling wolfSSL_CertManagerFreeCRL and then loading the new CRL.

Example

```
#include <wolfssl/ssl.h>

const char* crl1     = "./certs/crl/crl.pem";
WOLFSSL_CERT_MANAGER* cm = NULL;
```

```
cm = wolfSSL_CertManagerNew();
wolfSSL_CertManagerLoadCRL(cm, crl1, WOLFSSL_FILETYPE_PEM, 0);
…
wolfSSL_CertManagerFreeCRL(cm);
```

```
int wolfSSL_CertManagerCheckOCSP(
    WOLFSSL_CERT_MANAGER * cm,
    unsigned char * der,
    int sz
)
```

The function enables the WOLFSSL_CERT_MANAGER's member, ocspEnabled to signify that the OCSP check option is enabled.

Parameters:

- **cm** a pointer to a WOLFSSL_CERT_MANAGER structure, created using wolfSSL_CertManagerNew().
- **der** a byte pointer to the certificate.
- **sz** an int type representing the size of the DER cert.

See:

- ParseCertRelative
- CheckCertOCSP

Return:

- SSL_SUCCESS returned on successful execution of the function. The ocspEnabled member of the WOLFSSL_CERT_MANAGER is enabled.
- BAD_FUNC_ARG returned if the WOLFSSL_CERT_MANAGER structure is NULL or if an argument value that is not allowed is passed to a subroutine.
- MEMORY_E returned if there is an error allocating memory within this function or a subroutine.

Example

```
#import <wolfssl/ssl.h>

WOLFSSL* ssl = wolfSSL_new(ctx);
byte* der;
int sz; size of der
...
if(wolfSSL_CertManagerCheckOCSP(cm, der, sz) != SSL_SUCCESS){
 Failure case.
}
```

```
int wolfSSL_CertManagerEnableOCSP(
    WOLFSSL_CERT_MANAGER * cm,
    int options
)
```

Turns on OCSP if it's turned off and if compiled with the set option available.

Parameters:

- **cm** a pointer to a WOLFSSL_CERT_MANAGER structure, created using wolfSSL_CertManagerNew().
- **options** used to set values in WOLFSSL_CERT_MANAGER struct.

See: wolfSSL_CertManagerNew

Return:

- SSL_SUCCESS returned if the function call is successful.
- BAD_FUNC_ARG if cm struct is NULL.
- MEMORY_E if WOLFSSL_OCSP struct value is NULL.
- SSL_FAILURE initialization of WOLFSSL_OCSP struct fails to initialize.
- NOT_COMPILED_IN build not compiled with correct feature enabled.

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_CTX* ctx = wolfSSL_CTX_new(protocol method);
WOLFSSL* ssl = wolfSSL_new(ctx);
WOLFSSL_CERT_MANAGER* cm = wolfSSL_CertManagerNew();
int options;
…
if(wolfSSL_CertManagerEnableOCSP(SSL_CM(ssl), options) != SSL_SUCCESS){
    Failure case.
}
```

```
int wolfSSL_CertManagerDisableOCSP(
    WOLFSSL_CERT_MANAGER *
)
```

Disables OCSP certificate revocation.

Parameters:

- **ssl** - a pointer to a WOLFSSL structure, created using wolfSSL_new().

See: wolfSSL_DisableCRL

Return:

- SSL_SUCCESS wolfSSL_CertMangerDisableCRL successfully disabled the crlEnabled member of the WOLFSSL_CERT_MANAGER structure.
- BAD_FUNC_ARG the WOLFSSL structure was NULL.

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_CTX* ctx = wolfSSL_CTX_new(method);
WOLFSSL* ssl = wolfSSL_new(ctx);
...
if(wolfSSL_CertManagerDisableOCSP(ssl) != SSL_SUCCESS){
    Fail case.
}
```

```
int wolfSSL_CertManagerSetOCSPOverrideURL(
    WOLFSSL_CERT_MANAGER * cm,
    const char * url
)
```

The function copies the url to the ocspOverrideURL member of the WOLFSSL_CERT_MANAGER structure.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- ocspOverrideURL
- wolfSSL_SetOCSP_OverrideURL

Return:

- SSL_SUCCESS the function was able to execute as expected.
- BAD_FUNC_ARG the WOLFSSL_CERT_MANAGER struct is NULL.
- MEMEORY_E Memory was not able to be allocated for the ocspOverrideURL member of the certificate manager.

Example

```
#include <wolfssl/ssl.h>
WOLFSSL_CERT_MANAGER* cm = wolfSSL_CertManagerNew();
const char* url;
…
int wolfSSL_SetOCSP_OverrideURL(WOLFSSL* ssl, const char* url)
…
if(wolfSSL_CertManagerSetOCSPOverrideURL(SSL_CM(ssl), url) != SSL_SUCCESS){
    Failure case.
}
```

```
int wolfSSL_CertManagerSetOCSP_Cb(
    WOLFSSL_CERT_MANAGER * cm,
    CbOCSPIO ioCb,
```

```
    CbOCSPRespFree respFreeCb,
    void * ioCbCtx
)
```

The function sets the OCSP callback in the WOLFSSL_CERT_MANAGER.

Parameters:

- **cm** a pointer to a WOLFSSL_CERT_MANAGER structure.
- **ioCb** a function pointer of type CbOCSPIO.
- **respFreeCb** - a function pointer of type CbOCSPRespFree.
- **ioCbCtx** - a void pointer variable to the I/O callback user registered context.

See:

- wolfSSL_CertManagerSetOCSPOverrideURL
- wolfSSL_CertManagerCheckOCSP
- wolfSSL_CertManagerEnableOCSPStapling
- wolfSSL_EnableOCSP
- wolfSSL_DisableOCSP
- wolfSSL_SetOCSP_Cb

Return:

- SSL_SUCCESS returned on successful execution. The arguments are saved in the WOLFSSL_CERT_MANAGER structure.
- BAD_FUNC_ARG returned if the WOLFSSL_CERT_MANAGER is NULL.

Example

```
#include <wolfssl/ssl.h>

wolfSSL_SetOCSP_Cb(WOLFSSL* ssl, CbOCSPIO ioCb,
CbOCSPRespFree respFreeCb, void* ioCbCtx){
…
return wolfSSL_CertManagerSetOCSP_Cb(SSL_CM(ssl), ioCb, respFreeCb, ioCbCtx);
```

```
int wolfSSL_CertManagerEnableOCSPStapling(
    WOLFSSL_CERT_MANAGER * cm
)
```

This function turns on OCSP stapling if it is not turned on as well as set the options.

Parameters:

- **cm** a pointer to a WOLFSSL_CERT_MANAGER structure, a member of the WOLFSSL_CTX structure.

See: wolfSSL_CTX_EnableOCSPStapling

Return:

- SSL_SUCCESS returned if there were no errors and the function executed successfully.
- BAD_FUNC_ARG returned if the WOLFSSL_CERT_MANAGER structure is NULL or otherwise if there was a unpermitted argument value passed to a subroutine.
- MEMORY_E returned if there was an issue allocating memory.
- SSL_FAILURE returned if the initialization of the OCSP structure failed.
- NOT_COMPILED_IN returned if wolfSSL was not compiled with HAVE_CERTIFICATE_STATUS_REQUEST option.

Example

```
int wolfSSL_CTX_EnableOCSPStapling(WOLFSSL_CTX* ctx){
…
return wolfSSL_CertManagerEnableOCSPStapling(ctx->cm);
```

A.2 Memory Handling

A.1.2.24 function wolfSSL_CertManagerEnableOCSPStapling

A.2.1 Functions

	Name
void *	**wolfSSL_Malloc**(size_t size, void * heap, int type)This function is similar to malloc(), but calls the memory allocation function which wolfSSL has been configured to use. By default, wolfSSL uses malloc(). This can be changed using the wolfSSL memory abstraction layer _ see wolfSSL_SetAllocators(). Note wolfSSL_Malloc is not called directly by wolfSSL, but instead called by macro XMALLOC. For the default build only the size argument exists. If using WOLFSSL_STATIC_MEMORY build then heap and type arguments are included.
void	**wolfSSL_Free**(void * ptr, void * heap, int type)This function is similar to free(), but calls the memory free function which wolfSSL has been configured to use. By default, wolfSSL uses free(). This can be changed using the wolfSSL memory abstraction layer _ see wolfSSL_SetAllocators(). Note wolfSSL_Free is not called directly by wolfSSL, but instead called by macro XFREE. For the default build only the ptr argument exists. If using WOLFSSL_STATIC_MEMORY build then heap and type arguments are included.

	Name
void *	**wolfSSL_Realloc**(void * ptr, size_t size, void * heap, int type)This function is similar to realloc(), but calls the memory re_allocation function which wolfSSL has been configured to use. By default, wolfSSL uses realloc(). This can be changed using the wolfSSL memory abstraction layer _ see wolfSSL_SetAllocators(). Note wolfSSL_Realloc is not called directly by wolfSSL, but instead called by macro XREALLOC. For the default build only the size argument exists. If using WOLFSSL_STATIC_MEMORY build then heap and type arguments are included.
int	**wolfSSL_SetAllocators**(wolfSSL_Malloc_cb , wolfSSL_Free_cb , wolfSSL_Realloc_cb)This function registers the allocation functions used by wolfSSL. By default, if the system supports it, malloc/free and realloc are used. Using this function allows the user at runtime to install their own memory handlers.
int	**wolfSSL_StaticBufferSz**(byte * buffer, word32 sz, int flag)This function is available when static memory feature is used (–enable_staticmemory). It gives the optimum buffer size for memory “buckets”. This allows for a way to compute buffer size so that no extra unused memory is left at the end after it has been partitioned. The returned value, if positive, is the computed buffer size to use.
int	**wolfSSL_MemoryPaddingSz**(void)This function is available when static memory feature is used (–enable_staticmemory). It gives the size of padding needed for each partition of memory. This padding size will be the size needed to contain a memory management structure along with any extra for memory alignment.

	Name
int	**wolfSSL_CTX_load_static_memory**(WOLFSSL_CTX ** ctx, wolfSSL_method_func method, unsigned char * buf, unsigned int sz, int flag, int max)This function is used to set aside static memory for a CTX. Memory set aside is then used for the CTX's lifetime and for any SSL objects created from the CTX. By passing in a NULL ctx pointer and a wolfSSL_method_func function the creation of the CTX itself will also use static memory. wolfSSL_method_func has the function signature of WOLFSSL_METHOD* (*wolfSSL_method_func)(void* heap);. Passing in 0 for max makes it behave as if not set and no max concurrent use restrictions is in place. The flag value passed in determines how the memory is used and behavior while operating. Available flags are the following.
int	**wolfSSL_CTX_is_static_memory**(WOLFSSL_CTX * ctx, WOLFSSL_MEM_STATS * mem_stats)This function does not change any of the connections behavior and is used only for gathering information about the static memory usage.
int	**wolfSSL_is_static_memory**(WOLFSSL * ssl, WOLFSSL_MEM_CONN_STATS * mem_stats)wolfSSL_is_static_memory is used to gather information about a SSL's static memory usage. The return value indicates if static memory is being used and WOLFSSL_MEM_CONN_STATS will be filled out if and only if the flag WOLFMEM_TRACK_STATS was passed to the parent CTX when loading in static memory.
int	**wc_LoadStaticMemory**(WOLFSSL_HEAP_HINT * hint, unsigned char * buf, unsigned int sz, int flag, int max)This function is used to set aside static memory for wolfCrypt use. Memory can be used by passing the created heap hint into functions. An example of this is when calling wc_InitRng_ex. The flag value passed in determines how the memory is used and behavior while operating, in general wolfCrypt operations will use memory from a WOLFMEM_GENERAL pool. Available flags are the following.

	Name
void *	**XMALLOC**(size_t n, void * heap, int type)This is not actually a function, but rather a preprocessor macro, which allows the user to substitute in their own malloc, realloc, and free functions in place of the standard C memory functions. To use external memory functions, define XMALLOC_USER. This will cause the memory functions to be replaced by external functions of the form: extern void *XMALLOC(size_t n, void* heap, int type); extern void _XREALLOC(void *p, size_t n, void_ heap, int type); extern void XFREE(void* p, void *heap, int type); To use the basic C memory functions in place of wolfSSL_Malloc, wolfSSL_Realloc, wolfSSL_Free, define NO_WOLFSSL_MEMORY. This will replace the memory functions with: #define XMALLOC(s, h, t) ((void)h, (void)t, malloc((s))) #define XFREE(p, h, t) {void* xp = (p); if((xp)) free((xp));} #define XREALLOC(p, n, h, t) realloc((p), (n)) If none of these options are selected, the system will default to use the wolfSSL memory functions. A user can set custom memory functions through callback hooks, (see wolfSSL_Malloc, wolfSSL_Realloc, wolfSSL_Free). This option will replace the memory functions with: #define XMALLOC(s, h, t) ((void)h, (void)t, wolfSSL_Malloc((s))) #define XFREE(p, h, t) {void* xp = (p); if((xp)) wolfSSL_Free((xp));} #define XREALLOC(p, n, h, t) wolfSSL_Realloc((p), (n))

	Name
void *	**XREALLOC**(void * p, size_t n, void * heap, int type)This is not actually a function, but rather a preprocessor macro, which allows the user to substitute in their own malloc, realloc, and free functions in place of the standard C memory functions. To use external memory functions, define XMALLOC_USER. This will cause the memory functions to be replaced by external functions of the form: extern void *XMALLOC(size_t n, void* heap, int type); extern void _XREALLOC(void *p, size_t n, void_ heap, int type); extern void XFREE(void* p, void *heap, int type); To use the basic C memory functions in place of wolfSSL_Malloc, wolfSSL_Realloc, wolfSSL_Free, define NO_WOLFSSL_MEMORY. This will replace the memory functions with: #define XMALLOC(s, h, t) ((void)h, (void)t, malloc((s))) #define XFREE(p, h, t) {void* xp = (p); if((xp)) free((xp));} #define XREALLOC(p, n, h, t) realloc((p), (n)) If none of these options are selected, the system will default to use the wolfSSL memory functions. A user can set custom memory functions through callback hooks, (see wolfSSL_Malloc, wolfSSL_Realloc, wolfSSL_Free). This option will replace the memory functions with: #define XMALLOC(s, h, t) ((void)h, (void)t, wolfSSL_Malloc((s))) #define XFREE(p, h, t) {void* xp = (p); if((xp)) wolfSSL_Free((xp));} #define XREALLOC(p, n, h, t) wolfSSL_Realloc((p), (n))

	Name
void	**XFREE**(void * p, void * heap, int type)This is not actually a function, but rather a preprocessor macro, which allows the user to substitute in their own malloc, realloc, and free functions in place of the standard C memory functions. To use external memory functions, define XMALLOC_USER. This will cause the memory functions to be replaced by external functions of the form: extern void *XMALLOC(size_t n, void* heap, int type); extern void _XREALLOC(void *p, size_t n, void_ heap, int type); extern void XFREE(void* p, void *heap, int type); To use the basic C memory functions in place of wolfSSL_Malloc, wolfSSL_Realloc, wolfSSL_Free, define NO_WOLFSSL_MEMORY. This will replace the memory functions with: #define XMALLOC(s, h, t) ((void)h, (void)t, malloc((s))) #define XFREE(p, h, t) {void* xp = (p); if((xp)) free((xp));} #define XREALLOC(p, n, h, t) realloc((p), (n)) If none of these options are selected, the system will default to use the wolfSSL memory functions. A user can set custom memory functions through callback hooks, (see wolfSSL_Malloc, wolfSSL_Realloc, wolfSSL_Free). This option will replace the memory functions with: #define XMALLOC(s, h, t) ((void)h, (void)t, wolfSSL_Malloc((s))) #define XFREE(p, h, t) {void* xp = (p); if((xp)) wolfSSL_Free((xp));} #define XREALLOC(p, n, h, t) wolfSSL_Realloc((p), (n))

A.2.2 Functions Documentation

```
void * wolfSSL_Malloc(
    size_t size,
    void * heap,
    int type
)
```

This function is similar to malloc(), but calls the memory allocation function which wolfSSL has been configured to use. By default, wolfSSL uses malloc(). This can be changed using the wolfSSL memory abstraction layer - see wolfSSL_SetAllocators(). Note wolfSSL_Malloc is not called directly by wolfSSL, but instead called by macro XMALLOC. For the default build only the size argument exists. If using WOLFSSL_STATIC_MEMORY build then heap and type arguments are included.

Parameters:

- **size** size, in bytes, of the memory to allocate
- **heap** heap hint to use for memory. Can be NULL
- **type** dynamic type (see DYNAMIC_TYPE_ list in types.h)

See:

- wolfSSL_Free
- wolfSSL_Realloc
- wolfSSL_SetAllocators
- XMALLOC
- XFREE
- XREALLOC

Return:

- pointer If successful, this function returns a pointer to allocated memory.
- error If there is an error, NULL will be returned.

Example

```
int* tenInts = (int*)wolfSSL_Malloc(sizeof(int)*10);
```

```
void wolfSSL_Free(
    void * ptr,
    void * heap,
    int type
)
```

This function is similar to free(), but calls the memory free function which wolfSSL has been configured to use. By default, wolfSSL uses free(). This can be changed using the wolfSSL memory abstraction layer - see wolfSSL_SetAllocators(). Note wolfSSL_Free is not called directly by wolfSSL, but instead called by macro XFREE. For the default build only the ptr argument exists. If using WOLFSSL_STATIC_MEMORY build then heap and type arguments are included.

Parameters:

- **ptr** pointer to the memory to be freed.
- **heap** heap hint to use for memory. Can be NULL
- **type** dynamic type (see DYNAMIC_TYPE_ list in types.h)

See:

- wolfSSL_Alloc
- wolfSSL_Realloc
- wolfSSL_SetAllocators
- XMALLOC
- XFREE
- XREALLOC

Return: none No returns.

Example

```
int* tenInts = (int*)wolfSSL_Malloc(sizeof(int)*10);
// process data as desired
...
if(tenInts) {
    wolfSSL_Free(tenInts);
}
```

```
void * wolfSSL_Realloc(
    void * ptr,
    size_t size,
    void * heap,
    int type
)
```

This function is similar to realloc(), but calls the memory re-allocation function which wolfSSL has been configured to use. By default, wolfSSL uses realloc(). This can be changed using the wolfSSL memory abstraction layer - see wolfSSL_SetAllocators(). Note wolfSSL_Realloc is not called directly by wolfSSL, but instead called by macro XREALLOC. For the default build only the size argument exists. If using WOLFSSL_STATIC_MEMORY build then heap and type arguments are included.

Parameters:

- **ptr** pointer to the previously-allocated memory, to be reallocated.
- **size** number of bytes to allocate.
- **heap** heap hint to use for memory. Can be NULL
- **type** dynamic type (see DYNAMIC_TYPE_ list in types.h)

See:

- wolfSSL_Free
- wolfSSL_Malloc
- wolfSSL_SetAllocators
- XMALLOC
- XFREE
- XREALLOC

Return:

- pointer If successful, this function returns a pointer to re-allocated memory. This may be the same pointer as ptr, or a new pointer location.
- Null If there is an error, NULL will be returned.

Example

```
int* tenInts = (int*)wolfSSL_Malloc(sizeof(int)*10);
int* twentyInts = (int*)wolfSSL_Realloc(tenInts, sizeof(int)*20);
```

```
int wolfSSL_SetAllocators(
    wolfSSL_Malloc_cb ,
    wolfSSL_Free_cb ,
    wolfSSL_Realloc_cb
)
```

This function registers the allocation functions used by wolfSSL. By default, if the system supports it, malloc/free and realloc are used. Using this function allows the user at runtime to install their own memory handlers.

Parameters:

- **malloc_function** memory allocation function for wolfSSL to use. Function signature must match wolfSSL_Malloc_cb prototype, above.
- **free_function** memory free function for wolfSSL to use. Function signature must match wolfSSL_Free_cb prototype, above.
- **realloc_function** memory re-allocation function for wolfSSL to use. Function signature must match wolfSSL_Realloc_cb prototype, above.

See: none

Return:

- Success If successful this function will return 0.
- BAD_FUNC_ARG is the error that will be returned if a function pointer is not provided.

Example

```
static void* MyMalloc(size_t size)
{
    // custom malloc function
}

static void MyFree(void* ptr)
{
    // custom free function
}

static void* MyRealloc(void* ptr, size_t size)
{
    // custom realloc function
}

// Register custom memory functions with wolfSSL
int ret = wolfSSL_SetAllocators(MyMalloc, MyFree, MyRealloc);
if (ret != 0) {
    // failed to set memory functions
}
```

```
int wolfSSL_StaticBufferSz(
    byte * buffer,
```

```
    word32 sz,
    int flag
)
```

This function is available when static memory feature is used (–enable-staticmemory). It gives the optimum buffer size for memory “buckets”. This allows for a way to compute buffer size so that no extra unused memory is left at the end after it has been partitioned. The returned value, if positive, is the computed buffer size to use.

Parameters:

- **buffer** pointer to buffer
- **size** size of buffer
- **type** desired type of memory ie WOLFMEM_GENERAL or WOLFMEM_IO_POOL

See:

- wolfSSL_Malloc
- wolfSSL_Free

Return:

- Success On successfully completing buffer size calculations a positive value is returned. This returned value is for optimum buffer size.
- Failure All negative values are considered to be error cases.

Example

```
byte buffer[1000];
word32 size = sizeof(buffer);
int optimum;
optimum = wolfSSL_StaticBufferSz(buffer, size, WOLFMEM_GENERAL);
if (optimum < 0) { //handle error case }
printf(“The optimum buffer size to make use of all memory is %d\n”,
optimum);
...
```

```
int wolfSSL_MemoryPaddingSz(
    void
)
```

This function is available when static memory feature is used (–enable-staticmemory). It gives the size of padding needed for each partition of memory. This padding size will be the size needed to contain a memory management structure along with any extra for memory alignment.

Parameters:

- **none** No parameters.

See:

- wolfSSL_Malloc
- wolfSSL_Free

Return:

- On successfully memory padding calculation the return value will be a positive value
- All negative values are considered error cases.

Example

```
int padding;
padding = wolfSSL_MemoryPaddingSz();
if (padding < 0) { //handle error case }
printf("The padding size needed for each \"bucket\" of memory is %d\n",
padding);
// calculation of buffer for IO POOL size is number of buckets
// times (padding + WOLFMEM_IO_SZ)
...
```

```
int wolfSSL_CTX_load_static_memory(
    WOLFSSL_CTX ** ctx,
    wolfSSL_method_func method,
    unsigned char * buf,
    unsigned int sz,
    int flag,
    int max
)
```

This function is used to set aside static memory for a CTX. Memory set aside is then used for the CTX's lifetime and for any SSL objects created from the CTX. By passing in a NULL ctx pointer and a wolfSSL_method_func function the creation of the CTX itself will also use static memory. wolfSSL_method_func has the function signature of WOLFSSL_METHOD* (*wolfSSL_method_func)(void* heap);. Passing in 0 for max makes it behave as if not set and no max concurrent use restrictions is in place. The flag value passed in determines how the memory is used and behavior while operating. Available flags are the following.

Parameters:

- **ctx** address of pointer to a WOLFSSL_CTX structure.
- **method** function to create protocol. (should be NULL if ctx is not also NULL)
- **buf** memory to use for all operations.
- **sz** size of memory buffer being passed in.
- **flag** type of memory.
- **max** max concurrent operations.

See:

- wolfSSL_CTX_new
- wolfSSL_CTX_is_static_memory
- wolfSSL_is_static_memory

Return:

- If successful, SSL_SUCCESS will be returned.
- All unsuccessful return values will be less than 0 or equal to SSL_FAILURE.

0 - default general memory

WOLFMEM_IO_POOL - used for input/output buffer when sending receiving messages. Overrides general memory, so all memory in buffer passed in is used for IO. WOLFMEM_IO_FIXED - same as WOLFMEM_IO_POOL but each SSL now keeps two buffers to themselves for their lifetime. WOLFMEM_TRACK_STATS - each SSL keeps track of memory stats while running.

Example

```
WOLFSSL_CTX* ctx;
WOLFSSL* ssl;
int ret;
unsigned char memory[MAX];
int memorySz = MAX;
unsigned char IO[MAX];
int IOSz = MAX;
int flag = WOLFMEM_IO_FIXED | WOLFMEM_TRACK_STATS;
...
// create ctx also using static memory, start with general memory to use
ctx = NULL:
ret = wolfSSL_CTX_load_static_memory(&ctx, wolfSSLv23_server_method_ex,
 ↪  memory, memorySz, 0,
MAX_CONCURRENT_HANDSHAKES);
if (ret != SSL_SUCCESS) {
// handle error case
}
// load in memory for use with IO
ret = wolfSSL_CTX_load_static_memory(&ctx, NULL, IO, IOSz, flag,
 ↪  MAX_CONCURRENT_IO);
if (ret != SSL_SUCCESS) {
// handle error case
}
...
```

```
int wolfSSL_CTX_is_static_memory(
    WOLFSSL_CTX * ctx,
    WOLFSSL_MEM_STATS * mem_stats
)
```

This function does not change any of the connections behavior and is used only for gathering information about the static memory usage.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().
- **mem_stats** structure to hold information about staic memory usage.

See:

- wolfSSL_CTX_new
- wolfSSL_CTX_load_static_memory
- wolfSSL_is_static_memory

Return:

- A value of 1 is returned if using static memory for the CTX is true.
- 0 is returned if not using static memory.

Example

```
WOLFSSL_CTX* ctx;
int ret;
WOLFSSL_MEM_STATS mem_stats;
...
//get information about static memory with CTX

ret = wolfSSL_CTX_is_static_memory(ctx, &mem_stats);

if (ret == 1) {
    // handle case of is using static memory
    // print out or inspect elements of mem_stats
}

if (ret == 0) {
    //handle case of ctx not using static memory
}
...
```

```
int wolfSSL_is_static_memory(
    WOLFSSL * ssl,
    WOLFSSL_MEM_CONN_STATS * mem_stats
)
```

wolfSSL_is_static_memory is used to gather information about a SSL's static memory usage. The return value indicates if static memory is being used and WOLFSSL_MEM_CONN_STATS will be filled out if and only if the flag WOLFMEM_TRACK_STATS was passed to the parent CTX when loading in static memory.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **mem_stats** structure to contain static memory usage

See:

- wolfSSL_new
- wolfSSL_CTX_is_static_memory

Return:

- A value of 1 is returned if using static memory for the CTX is true.
- 0 is returned if not using static memory.

Example

```
WOLFSSL* ssl;
int ret;
WOLFSSL_MEM_CONN_STATS mem_stats;

...

ret = wolfSSL_is_static_memory(ssl, mem_stats);

if (ret == 1) {
    // handle case when is static memory
    // investigate elements in mem_stats if WOLFMEM_TRACK_STATS flag
}
...
```

```
int wc_LoadStaticMemory(
    WOLFSSL_HEAP_HINT * hint,
    unsigned char * buf,
    unsigned int sz,
    int flag,
    int max
)
```

This function is used to set aside static memory for wolfCrypt use. Memory can be used by passing the created heap hint into functions. An example of this is when calling wc_InitRng_ex. The flag value passed in determines how the memory is used and behavior while operating, in general wolfCrypt operations will use memory from a WOLFMEM_GENERAL pool. Available flags are the following.

Parameters:

- **hint** WOLFSSL_HEAP_HINT structure to use
- **buf** memory to use for all operations.
- **sz** size of memory buffer being passed in.
- **flag** type of memory.
- **max** max concurrent operations (handshakes, IO).

See: none

Return:

- If successful, 0 will be returned.
- All unsuccessful return values will be less than 0.

WOLFMEM_GENERAL - default general memory

WOLFMEM_IO_POOL - used for input/output buffer when sending receiving messages. Overrides general memory, so all memory in buffer passed in is used for IO. WOLFMEM_IO_FIXED - same as WOLFMEM_IO_POOL but each SSL now keeps two buffers to themselves for their lifetime. WOLFMEM_TRACK_STATS - each SSL keeps track of memory stats while running

Example

```
WOLFSSL_HEAP_HINT hint;
int ret;
unsigned char memory[MAX];
int memorySz = MAX;
int flag = WOLFMEM_GENERAL | WOLFMEM_TRACK_STATS;
...

// load in memory for use

ret = wc_LoadStaticMemory(&hint, memory, memorySz, flag, 0);
if (ret != SSL_SUCCESS) {
    // handle error case
}
...

ret = wc_InitRng_ex(&rng, hint, 0);

// check ret value
```

```
void * XMALLOC(
    size_t n,
    void * heap,
    int type
)
```

This is not actually a function, but rather a preprocessor macro, which allows the user to substitute in their own malloc, realloc, and free functions in place of the standard C memory functions. To use external memory functions, define XMALLOC_USER. This will cause the memory functions to be replaced by external functions of the form: extern void *XMALLOC(size_t n, void* heap, int type); extern void _XREALLOC(void *p, size_t n, void_ heap, int type); extern void XFREE(void* p, void *heap, int type); To use the basic C memory functions in place of wolfSSL_Malloc, wolfSSL_Realloc, wolfSSL_Free, define NO_WOLFSSL_MEMORY. This will replace the memory functions with: #define XMALLOC(s, h, t) ((void)h, (void)t, malloc((s))) #define XFREE(p, h, t) {void* xp = (p); if((xp)) free((xp));} #define XREALLOC(p, n, h, t) realloc((p), (n)) If none of these options are selected, the system will default to use the wolfSSL memory functions. A user can set custom memory functions through callback hooks, (see wolfSSL_Malloc, wolfSSL_Realloc, wolfSSL_Free). This option will replace the memory functions with: #define XMALLOC(s, h, t) ((void)h, (void)t, wolfSSL_Malloc((s))) #define XFREE(p, h, t) {void* xp = (p); if((xp)) wolfSSL_Free((xp));} #define XREALLOC(p, n, h, t) wolfSSL_Realloc((p), (n))

Parameters:

- **s** size of memory to allocate
- **h** (used by custom XMALLOC function) pointer to the heap to use
- **t** memory allocation types for user hints. See enum in types.h

See:

- wolfSSL_Malloc
- wolfSSL_Realloc
- wolfSSL_Free
- wolfSSL_SetAllocators

Return:

- pointer Return a pointer to allocated memory on success
- NULL on failure

Example

```
int* tenInts = XMALLOC(sizeof(int)*10, NULL, DYNAMIC_TYPE_TMP_BUFFER);
if (tenInts == NULL) {
    // error allocating space
    return MEMORY_E;
}
```

```
void * XREALLOC(
    void * p,
    size_t n,
    void * heap,
    int type
)
```

This is not actually a function, but rather a preprocessor macro, which allows the user to substitute in their own malloc, realloc, and free functions in place of the standard C memory functions. To use external memory functions, define XMALLOC_USER. This will cause the memory functions to be replaced by external functions of the form: extern void *XMALLOC(size_t n, void* heap, int type); extern void _XREALLOC(void *p, size_t n, void_ heap, int type); extern void XFREE(void* p, void *heap, int type); To use the basic C memory functions in place of wolfSSL_Malloc, wolfSSL_Realloc, wolfSSL_Free, define NO_WOLFSSL_MEMORY. This will replace the memory functions with: #define XMALLOC(s, h, t) ((void)h, (void)t, malloc((s))) #define XFREE(p, h, t) {void* xp = (p); if((xp)) free((xp));} #define XREALLOC(p, n, h, t) realloc((p), (n)) If none of these options are selected, the system will default to use the wolfSSL memory functions. A user can set custom memory functions through callback hooks, (see wolfSSL_Malloc, wolfSSL_Realloc, wolfSSL_Free). This option will replace the memory functions with: #define XMALLOC(s, h, t) ((void)h, (void)t, wolfSSL_Malloc((s))) #define XFREE(p, h, t) {void* xp = (p); if((xp)) wolfSSL_Free((xp));} #define XREALLOC(p, n, h, t) wolfSSL_Realloc((p), (n))

Parameters:

- **p** pointer to the address to reallocate
- **n** size of memory to allocate

- **h** (used by custom XREALLOC function) pointer to the heap to use
- **t** memory allocation types for user hints. See enum in types.h

See:

- wolfSSL_Malloc
- wolfSSL_Realloc
- wolfSSL_Free
- wolfSSL_SetAllocators

Return:

- Return a pointer to allocated memory on success
- NULL on failure

Example

```
int* tenInts = (int*)XMALLOC(sizeof(int)*10, NULL, DYNAMIC_TYPE_TMP_BUFFER);
int* twentyInts = (int*)XREALLOC(tenInts, sizeof(int)*20, NULL,
    DYNAMIC_TYPE_TMP_BUFFER);
```

```
void XFREE(
    void * p,
    void * heap,
    int type
)
```

This is not actually a function, but rather a preprocessor macro, which allows the user to substitute in their own malloc, realloc, and free functions in place of the standard C memory functions. To use external memory functions, define XMALLOC_USER. This will cause the memory functions to be replaced by external functions of the form: extern void *XMALLOC(size_t n, void* heap, int type); extern void _XREALLOC(void *p, size_t n, void_ heap, int type); extern void XFREE(void* p, void *heap, int type); To use the basic C memory functions in place of wolfSSL_Malloc, wolfSSL_Realloc, wolfSSL_Free, define NO_WOLFSSL_MEMORY. This will replace the memory functions with: #define XMALLOC(s, h, t) ((void)h, (void)t, malloc((s))) #define XFREE(p, h, t) {void* xp = (p); if((xp)) free((xp));} #define XREALLOC(p, n, h, t) realloc((p), (n)) If none of these options are selected, the system will default to use the wolfSSL memory functions. A user can set custom memory functions through callback hooks, (see wolfSSL_Malloc, wolfSSL_Realloc, wolfSSL_Free). This option will replace the memory functions with: #define XMALLOC(s, h, t) ((void)h, (void)t, wolfSSL_Malloc((s))) #define XFREE(p, h, t) {void* xp = (p); if((xp)) wolfSSL_Free((xp));} #define XREALLOC(p, n, h, t) wolfSSL_Realloc((p), (n))

Parameters:

- **p** pointer to the address to free
- **h** (used by custom XFREE function) pointer to the heap to use
- **t** memory allocation types for user hints. See enum in types.h

See:

- wolfSSL_Malloc

- wolfSSL_Realloc
- wolfSSL_Free
- wolfSSL_SetAllocators

Return: none No returns.

Example

```
int* tenInts = XMALLOC(sizeof(int) * 10, NULL, DYNAMIC_TYPE_TMP_BUFFER);
if (tenInts == NULL) {
    // error allocating space
    return MEMORY_E;
}
```

A.3 OpenSSL API

A.2.2.13 function XFREE

A.3.1 Functions

	Name
int	**wolfSSL_BN_mod_exp**(WOLFSSL_BIGNUM * r, const WOLFSSL_BIGNUM * a, const WOLFSSL_BIGNUM * p, const WOLFSSL_BIGNUM * m, WOLFSSL_BN_CTX * ctx)This function performs the following math "r = (a^p) % m".
const WOLFSSL_EVP_CIPHER *	**wolfSSL_EVP_des_ede3_ecb**(void)Getter functions for the respective WOLFSSL_EVP_CIPHER pointers. wolfSSL_EVP_init() must be called once in the program first to populate these cipher strings. WOLFSSL_DES_ECB macro must be defined for wolfSSL_EVP_des_ede3_ecb().
const WOLFSSL_EVP_CIPHER *	**wolfSSL_EVP_des_cbc**(void)Getter functions for the respective WOLFSSL_EVP_CIPHER pointers. wolfSSL_EVP_init() must be called once in the program first to populate these cipher strings. WOLFSSL_DES_ECB macro must be defined for wolfSSL_EVP_des_ecb().
int	**wolfSSL_EVP_DigestInit_ex**(WOLFSSL_EVP_MD_CTX * ctx, const WOLFSSL_EVP_MD * type, WOLFSSL_ENGINE * impl)Function for initializing WOLFSSL_EVP_MD_CTX. This function is a wrapper for wolfSSL_EVP_DigestInit() because wolfSSL does not use WOLFSSL_ENGINE.

	Name
int	**wolfSSL_EVP_CipherInit_ex**(WOLFSSL_EVP_CIPHER_CTX * ctx, const WOLFSSL_EVP_CIPHER * type, WOLFSSL_ENGINE * impl, const unsigned char * key, const unsigned char * iv, int enc)Function for initializing WOLFSSL_EVP_CIPHER_CTX. This function is a wrapper for wolfSSL_CipherInit() because wolfSSL does not use WOLFSSL_ENGINE.
int	**wolfSSL_EVP_EncryptInit_ex**(WOLFSSL_EVP_CIPHER_CTX * ctx, const WOLFSSL_EVP_CIPHER * type, WOLFSSL_ENGINE * impl, const unsigned char * key, const unsigned char * iv)Function for initializing WOLFSSL_EVP_CIPHER_CTX. This function is a wrapper for wolfSSL_EVP_CipherInit() because wolfSSL does not use WOLFSSL_ENGINE. Sets encrypt flag to be encrypt.
int	**wolfSSL_EVP_DecryptInit_ex**(WOLFSSL_EVP_CIPHER_CTX * ctx, const WOLFSSL_EVP_CIPHER * type, WOLFSSL_ENGINE * impl, const unsigned char * key, const unsigned char * iv)Function for initializing WOLFSSL_EVP_CIPHER_CTX. This function is a wrapper for wolfSSL_EVP_CipherInit() because wolfSSL does not use WOLFSSL_ENGINE. Sets encrypt flag to be decrypt.
int	**wolfSSL_EVP_CipherUpdate**(WOLFSSL_EVP_CIPHER_CTX * ctx, unsigned char * out, int * outl, const unsigned char * in, int inl)Function for encrypting/decrypting data. In buffer is added to be encrypted or decrypted and out buffer holds the results. outl will be the length of encrypted/decrypted information.
int	**wolfSSL_EVP_CipherFinal**(WOLFSSL_EVP_CIPHER_CTX * ctx, unsigned char * out, int * outl)This function performs the final cipher operations adding in padding. If WOLFSSL_EVP_CIPH_NO_PADDING flag is set in WOLFSSL_EVP_CIPHER_CTX structure then 1 is returned and no encryption/decryption is done. If padding flag is seti padding is added and encrypted when ctx is set to encrypt, padding values are checked when set to decrypt.
int	**wolfSSL_EVP_CIPHER_CTX_set_key_length**(WOLFSSL_EVP_CIPHE * ctx, int keylen)Setter function for WOLFSSL_EVP_CIPHER_CTX structure key length.
int	**wolfSSL_EVP_CIPHER_CTX_block_size**(const WOLFSSL_EVP_CIPHER_CTX * ctx)This is a getter function for the ctx block size.

	Name
int	**wolfSSL_EVP_CIPHER_block_size**(const WOLFSSL_EVP_CIPHER * cipher)This is a getter function for the block size of cipher.
void	**wolfSSL_EVP_CIPHER_CTX_set_flags**(WOLFSSL_EVP_CIPHER_CTX * ctx, int flags)Setter function for WOLFSSL_EVP_CIPHER_CTX structure.
void	**wolfSSL_EVP_CIPHER_CTX_clear_flags**(WOLFSSL_EVP_CIPHER_C * ctx, int flags)Clearing function for WOLFSSL_EVP_CIPHER_CTX structure.
int	**wolfSSL_EVP_CIPHER_CTX_set_padding**(WOLFSSL_EVP_CIPHER_ * c, int pad)Setter function for WOLFSSL_EVP_CIPHER_CTX structure to use padding.
unsigned long	**wolfSSL_EVP_CIPHER_CTX_flags**(const WOLFSSL_EVP_CIPHER_CTX * ctx)Getter function for WOLFSSL_EVP_CIPHER_CTX structure. Deprecated v1.1.0.
int	**wolfSSL_PEM_write_bio_PrivateKey**(WOLFSSL_BIO * bio, WOLFSSL_EVP_PKEY * key, const WOLFSSL_EVP_CIPHER * cipher, unsigned char * passwd, int len, wc_pem_password_cb * cb, void * arg)This function writes a key into a WOLFSSL_BIO structure in PEM format.
int	**wolfSSL_CTX_use_RSAPrivateKey_file function. The file argument contains a pointer to the RSA private key file, in the format specified by format.
int	**wolfSSL_use_certificate_file**(WOLFSSL * ssl, const char * file, int format)This function loads a certificate file into the SSL session (WOLFSSL structure). The certificate file is provided by the file argument. The format argument specifies the format type of the file - either SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM.
int	**wolfSSL_use_PrivateKey_file**(WOLFSSL * ssl, const char * file, int format)This function loads a private key file into the SSL session (WOLFSSL structure). The key file is provided by the file argument. The format argument specifies the format type of the file - SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM.
int	**wolfSSL_use_certificate_chain_file**(WOLFSSL * ssl, const char * file)This function loads a chain of certificates into the SSL session (WOLFSSL structure). The file containing the certificate chain is provided by the file argument, and must contain PEM_formatted certificates. This function will process up to MAX_CHAIN_DEPTH (default = 9, defined in internal.h) certificates, plus the subject certificate.

	Name
int	**wolfSSL_use_RSAPrivateKey_file function. The file argument contains a pointer to the RSA private key file, in the format specified by format.
long	**wolfSSL_set_tlsext_status_type**(WOLFSSL * s, int type)This function is called when the client application request that a server send back an OCSP status response (also known as OCSP stapling).Currently, the only supported type is TLSEXT_STATUSTYPE_ocsp.
WOLFSSL_X509_CHAIN *	**wolfSSL_get_peer_chain**(WOLFSSL * ssl)Retrieves the peer's certificate chain.
int	**wolfSSL_get_chain_count**(WOLFSSL_X509_CHAIN * chain)Retrieve's the peers certificate chain count.
int	**wolfSSL_get_chain_length**(WOLFSSL_X509_CHAIN * chain, int idx)Retrieves the peer's ASN1.DER certificate length in bytes at index (idx).
unsigned char *	**wolfSSL_get_chain_cert**(WOLFSSL_X509_CHAIN * chain, int idx)Retrieves the peer's ASN1.DER certificate at index (idx).
int	**wolfSSL_get_chain_cert_pem**(WOLFSSL_X509_CHAIN * chain, int idx, unsigned char * buf, int inLen, int * outLen)Retrieves the peer's PEM certificate at index (idx).
const unsigned char *	**wolfSSL_get_sessionID**(const WOLFSSL_SESSION * s)Retrieves the session's ID. The session ID is always 32 bytes long.
int	**wolfSSL_X509_get_serial_number**(WOLFSSL_X509 * x509, unsigned char * in, int * inOutSz)Retrieves the peer's certificate serial number. The serial number buffer (in) should be at least 32 bytes long and be provided as the *inOutSz argument as input. After calling the function* inOutSz will hold the actual length in bytes written to the in buffer.
WC_PKCS12 *	**wolfSSL_d2i_PKCS12_bio**(WOLFSSL_BIO * bio, WC_PKCS12 ** pkcs12)wolfSSL_d2i_PKCS12_bio (d2i_PKCS12_bio) copies in the PKCS12 information from WOLFSSL_BIO to the structure WC_PKCS12. The information is divided up in the structure as a list of Content Infos along with a structure to hold optional MAC information. After the information has been divided into chunks (but not decrypted) in the structure WC_PKCS12, it can then be parsed and decrypted by calling.
WC_PKCS12 *	**wolfSSL_i2d_PKCS12_bio**(WOLFSSL_BIO * bio, WC_PKCS12 * pkcs12)wolfSSL_i2d_PKCS12_bio (i2d_PKCS12_bio) copies in the cert information from the structure WC_PKCS12 to WOLFSSL_BIO.

	Name
int	**wolfSSL_PKCS12_parse(WOLFSSL_X509)** ca)PKCS12 can be enabled with adding –enable_opensslextra to the configure command. It can use triple DES and RC4 for decryption so would recommend also enabling these features when enabling opensslextra (–enable_des3 –enable_arc4). wolfSSL does not currently support RC2 so decryption with RC2 is currently not available. This may be noticeable with default encryption schemes used by OpenSSL command line to create .p12 files. wolfSSL_PKCS12_parse (PKCS12_parse). The first thing this function does is check the MAC is correct if present. If the MAC fails then the function returns and does not try to decrypt any of the stored Content Infos. This function then parses through each Content Info looking for a bag type, if the bag type is known it is decrypted as needed and either stored in the list of certificates being built or as a key found. After parsing through all bags the key found is then compared with the certificate list until a matching pair is found. This matching pair is then returned as the key and certificate, optionally the certificate list found is returned as a STACK_OF certificates. At the moment a CRL, Secret or SafeContents bag will be skipped over and not parsed. It can be seen if these or other “Unknown” bags are skipped over by viewing the debug print out. Additional attributes such as friendly name are skipped over when parsing a PKCS12 file.

A.3.2 Functions Documentation

```
int wolfSSL_BN_mod_exp(
    WOLFSSL_BIGNUM * r,
    const WOLFSSL_BIGNUM * a,
    const WOLFSSL_BIGNUM * p,
    const WOLFSSL_BIGNUM * m,
    WOLFSSL_BN_CTX * ctx
)
```

This function performs the following math “r = (a^p) % m”.

Parameters:

- **r** structure to hold result.
- **a** value to be raised by a power.
- **p** power to raise a by.
- **m** modulus to use.

- **ctx** currently not used with wolfSSL can be NULL.

See:

- wolfSSL_BN_new
- wolfSSL_BN_free

Return:

- SSL_SUCCESS On successfully performing math operation.
- SSL_FAILURE If an error case was encountered.

Example

```
WOLFSSL_BIGNUM r,a,p,m;
int ret;
// set big number values
ret  = wolfSSL_BN_mod_exp(r, a, p, m, NULL);
// check ret value
```

```
const WOLFSSL_EVP_CIPHER * wolfSSL_EVP_des_ede3_ecb(
    void
)
```

Getter functions for the respective WOLFSSL_EVP_CIPHER pointers. wolfSSL_EVP_init() must be called once in the program first to populate these cipher strings. WOLFSSL_DES_ECB macro must be defined for wolfSSL_EVP_des_ede3_ecb().

Parameters:

- **none** No parameters.

See: wolfSSL_EVP_CIPHER_CTX_init

Return: pointer Returns a WOLFSSL_EVP_CIPHER pointer for DES EDE3 operations.

Example

```
printf("block size des ede3 cbc = %d\n",
wolfSSL_EVP_CIPHER_block_size(wolfSSL_EVP_des_ede3_cbc()));
printf("block size des ede3 ecb = %d\n",
wolfSSL_EVP_CIPHER_block_size(wolfSSL_EVP_des_ede3_ecb()));
```

```
const WOLFSSL_EVP_CIPHER * wolfSSL_EVP_des_cbc(
    void
)
```

Getter functions for the respective WOLFSSL_EVP_CIPHER pointers. wolfSSL_EVP_init() must be called once in the program first to populate these cipher strings. WOLFSSL_DES_ECB macro must be defined for wolfSSL_EVP_des_ecb().

Parameters:

- **none** No parameters.

See: wolfSSL_EVP_CIPHER_CTX_init

Return: pointer Returns a WOLFSSL_EVP_CIPHER pointer for DES operations.

Example

```
WOLFSSL_EVP_CIPHER* cipher;
cipher = wolfSSL_EVP_des_cbc();
…
```

```
int wolfSSL_EVP_DigestInit_ex(
    WOLFSSL_EVP_MD_CTX * ctx,
    const WOLFSSL_EVP_MD * type,
    WOLFSSL_ENGINE * impl
)
```

Function for initializing WOLFSSL_EVP_MD_CTX. This function is a wrapper for wolfSSL_EVP_DigestInit() because wolfSSL does not use WOLFSSL_ENGINE.

Parameters:

- **ctx** structure to initialize.
- **type** type of hash to do, for example SHA.
- **impl** engine to use. N/A for wolfSSL, can be NULL.

See:

- wolfSSL_EVP_MD_CTX_new
- wolfCrypt_Init
- wolfSSL_EVP_MD_CTX_free

Return:

- SSL_SUCCESS If successfully set.
- SSL_FAILURE If not successful.

Example

```
WOLFSSL_EVP_MD_CTX* md = NULL;
wolfCrypt_Init();
md = wolfSSL_EVP_MD_CTX_new();
if (md == NULL) {
```

```
    printf("error setting md\n");
    return -1;
}
printf("cipher md init ret = %d\n", wolfSSL_EVP_DigestInit_ex(md,
wolfSSL_EVP_sha1(), e));
//free resources
```

```
int wolfSSL_EVP_CipherInit_ex(
    WOLFSSL_EVP_CIPHER_CTX * ctx,
    const WOLFSSL_EVP_CIPHER * type,
    WOLFSSL_ENGINE * impl,
    const unsigned char * key,
    const unsigned char * iv,
    int enc
)
```

Function for initializing WOLFSSL_EVP_CIPHER_CTX. This function is a wrapper for wolfSSL_CipherInit() because wolfSSL does not use WOLFSSL_ENGINE.

Parameters:

- **ctx** structure to initialize.
- **type** type of encryption/decryption to do, for example AES.
- **impl** engine to use. N/A for wolfSSL, can be NULL.
- **key** key to set .
- **iv** iv if needed by algorithm.
- **enc** encryption (1) or decryption (0) flag.

See:

- wolfSSL_EVP_CIPHER_CTX_new
- wolfCrypt_Init
- wolfSSL_EVP_CIPHER_CTX_free

Return:

- SSL_SUCCESS If successfully set.
- SSL_FAILURE If not successful.

Example

```
WOLFSSL_EVP_CIPHER_CTX* ctx = NULL;
WOLFSSL_ENGINE* e = NULL;
unsigned char key[16];
unsigned char iv[12];
wolfCrypt_Init();
ctx = wolfSSL_EVP_CIPHER_CTX_new();
if (ctx == NULL) {
    printf("issue creating ctx\n");
    return -1;
```

```
}

printf("cipher init ex error ret = %d\n", wolfSSL_EVP_CipherInit_ex(NULL,
EVP_aes_128_    cbc(), e, key, iv, 1));
printf("cipher init ex success ret = %d\n", wolfSSL_EVP_CipherInit_ex(ctx,
EVP_aes_128_c    bc(), e, key, iv, 1));
// free resources
```

```
int wolfSSL_EVP_EncryptInit_ex(
    WOLFSSL_EVP_CIPHER_CTX * ctx,
    const WOLFSSL_EVP_CIPHER * type,
    WOLFSSL_ENGINE * impl,
    const unsigned char * key,
    const unsigned char * iv
)
```

Function for initializing WOLFSSL_EVP_CIPHER_CTX. This function is a wrapper for wolfSSL_EVP_CipherInit() because wolfSSL does not use WOLFSSL_ENGINE. Sets encrypt flag to be encrypt.

Parameters:

- **ctx** structure to initialize.
- **type** type of encryption to do, for example AES.
- **impl** engine to use. N/A for wolfSSL, can be NULL.
- **key** key to use.
- **iv** iv to use.

See:

- wolfSSL_EVP_CIPHER_CTX_new
- wolfCrypt_Init
- wolfSSL_EVP_CIPHER_CTX_free

Return:

- SSL_SUCCESS If successfully set.
- SSL_FAILURE If not successful.

Example

```
WOLFSSL_EVP_CIPHER_CTX* ctx = NULL;
wolfCrypt_Init();
ctx = wolfSSL_EVP_CIPHER_CTX_new();
if (ctx == NULL) {
    printf("error setting ctx\n");
    return -1;
}
printf("cipher ctx init ret = %d\n", wolfSSL_EVP_EncryptInit_ex(ctx,
wolfSSL_EVP_aes_128_cbc(), e, key, iv));
//free resources
```

```
int wolfSSL_EVP_DecryptInit_ex(
    WOLFSSL_EVP_CIPHER_CTX * ctx,
    const WOLFSSL_EVP_CIPHER * type,
    WOLFSSL_ENGINE * impl,
    const unsigned char * key,
    const unsigned char * iv
)
```

Function for initializing WOLFSSL_EVP_CIPHER_CTX. This function is a wrapper for wolfSSL_EVP_CipherInit() because wolfSSL does not use WOLFSSL_ENGINE. Sets encrypt flag to be decrypt.

Parameters:

- **ctx** structure to initialize.
- **type** type of encryption/decryption to do, for example AES.
- **impl** engine to use. N/A for wolfSSL, can be NULL.
- **key** key to set .
- **iv** iv if needed by algorithm.
- **enc** encryption (1) or decryption (0) flag.

See:

- wolfSSL_EVP_CIPHER_CTX_new
- wolfCrypt_Init
- wolfSSL_EVP_CIPHER_CTX_free

Return:

- SSL_SUCCESS If successfully set.
- SSL_FAILURE If not successful.

Example

```
WOLFSSL_EVP_CIPHER_CTX* ctx = NULL;
WOLFSSL_ENGINE* e = NULL;
unsigned char key[16];
unsigned char iv[12];

wolfCrypt_Init();

ctx = wolfSSL_EVP_CIPHER_CTX_new();
if (ctx == NULL) {
    printf("issue creating ctx\n");
    return -1;
}

printf("cipher init ex error ret = %d\n", wolfSSL_EVP_DecryptInit_ex(NULL,
EVP_aes_128_    cbc(), e, key, iv, 1));
printf("cipher init ex success ret = %d\n", wolfSSL_EVP_DecryptInit_ex(ctx,
```

```
EVP_aes_128_c    bc(), e, key, iv, 1));
// free resources



int wolfSSL_EVP_CipherUpdate(
    WOLFSSL_EVP_CIPHER_CTX * ctx,
    unsigned char * out,
    int * outl,
    const unsigned char * in,
    int inl
)
```

Function for encrypting/decrypting data. In buffer is added to be encrypted or decrypted and out buffer holds the results. outl will be the length of encrypted/decrypted information.

Parameters:

- **ctx** structure to get cipher type from.
- **out** buffer to hold output.
- **outl** adjusted to be size of output.
- **in** buffer to perform operation on.
- **inl** length of input buffer.

See:

- wolfSSL_EVP_CIPHER_CTX_new
- wolfCrypt_Init
- wolfSSL_EVP_CIPHER_CTX_free

Return:

- SSL_SUCCESS If successful.
- SSL_FAILURE If not successful.

Example

```
WOLFSSL_EVP_CIPHER_CTX* ctx = NULL;
unsigned char out[100];
int outl;
unsigned char in[100];
int inl = 100;

ctx = wolfSSL_EVP_CIPHER_CTX_new();
// set up ctx
ret = wolfSSL_EVP_CipherUpdate(ctx, out, outl, in, inl);
// check ret value
// buffer out holds outl bytes of data
// free resources
```

```
int wolfSSL_EVP_CipherFinal(
    WOLFSSL_EVP_CIPHER_CTX * ctx,
    unsigned char * out,
    int * outl
)
```

This function performs the final cipher operations adding in padding. If WOLFSSL_EVP_CIPH_NO_PADDING flag is set in WOLFSSL_EVP_CIPHER_CTX structure then 1 is returned and no encryption/decryption is done. If padding flag is seti padding is added and encrypted when ctx is set to encrypt, padding values are checked when set to decrypt.

Parameters:

- **ctx** structure to decrypt/encrypt with.
- **out** buffer for final decrypt/encrypt.
- **out1** size of out buffer when data has been added by function.

See: wolfSSL_EVP_CIPHER_CTX_new

Return:

- 1 Returned on success.
- 0 If encountering a failure.

Example

```
WOLFSSL_EVP_CIPHER_CTX* ctx;
int out1;
unsigned char out[64];
// create ctx
wolfSSL_EVP_CipherFinal(ctx, out, &out1);
```

```
int wolfSSL_EVP_CIPHER_CTX_set_key_length(
    WOLFSSL_EVP_CIPHER_CTX * ctx,
    int keylen
)
```

Setter function for WOLFSSL_EVP_CIPHER_CTX structure key length.

Parameters:

- **ctx** structure to set key length.
- **keylen** key length.

See: wolfSSL_EVP_CIPHER_flags

Return:

- SSL_SUCCESS If successfully set.
- SSL_FAILURE If failed to set key length.

Example

```
WOLFSSL_EVP_CIPHER_CTX* ctx;
int keylen;
// create ctx
wolfSSL_EVP_CIPHER_CTX_set_key_length(ctx, keylen);
```

```
int wolfSSL_EVP_CIPHER_CTX_block_size(
    const WOLFSSL_EVP_CIPHER_CTX * ctx
)
```

This is a getter function for the ctx block size.

Parameters:

- **ctx** the cipher ctx to get block size of.

See: wolfSSL_EVP_CIPHER_block_size

Return: size Returns ctx->block_size.

Example

```
const WOLFSSL_CVP_CIPHER_CTX* ctx;
//set up ctx
printf("block size = %d\n", wolfSSL_EVP_CIPHER_CTX_block_size(ctx));
```

```
int wolfSSL_EVP_CIPHER_block_size(
    const WOLFSSL_EVP_CIPHER * cipher
)
```

This is a getter function for the block size of cipher.

Parameters:

- **cipher** cipher to get block size of.

See: wolfSSL_EVP_aes_256_ctr

Return: size returns the block size.

Example

```
printf("block size = %d\n",
wolfSSL_EVP_CIPHER_block_size(wolfSSL_EVP_aes_256_ecb()));
```

```
void wolfSSL_EVP_CIPHER_CTX_set_flags(
    WOLFSSL_EVP_CIPHER_CTX * ctx,
    int flags
)
```

Setter function for WOLFSSL_EVP_CIPHER_CTX structure.

Parameters:

- **ctx** structure to set flag.
- **flag** flag to set in structure.

See:

- wolfSSL_EVP_CIPHER_flags
- wolfSSL_EVP_CIPHER_CTX_flags

Return: none No returns.

Example

```
WOLFSSL_EVP_CIPHER_CTX* ctx;
int flag;
// create ctx
wolfSSL_EVP_CIPHER_CTX_set_flags(ctx, flag);
```

```
void wolfSSL_EVP_CIPHER_CTX_clear_flags(
    WOLFSSL_EVP_CIPHER_CTX * ctx,
    int flags
)
```

Clearing function for WOLFSSL_EVP_CIPHER_CTX structure.

Parameters:

- **ctx** structure to clear flag.
- **flag** flag value to clear in structure.

See:

- wolfSSL_EVP_CIPHER_flags
- wolfSSL_EVP_CIPHER_CTX_flags

Return: none No returns.

Example

```
WOLFSSL_EVP_CIPHER_CTX* ctx;
int flag;
// create ctx
wolfSSL_EVP_CIPHER_CTX_clear_flags(ctx, flag);
```

```
int wolfSSL_EVP_CIPHER_CTX_set_padding(
    WOLFSSL_EVP_CIPHER_CTX * c,
    int pad
)
```

Setter function for WOLFSSL_EVP_CIPHER_CTX structure to use padding.

Parameters:

- **ctx** structure to set padding flag.
- **padding** 0 for not setting padding, 1 for setting padding.

See: wolfSSL_EVP_CIPHER_CTX_new

Return:

- SSL_SUCCESS If successfully set.
- BAD_FUNC_ARG If null argument passed in.

Example

```
WOLFSSL_EVP_CIPHER_CTX* ctx;
// create ctx
wolfSSL_EVP_CIPHER_CTX_set_padding(ctx, 1);
```

```
unsigned long wolfSSL_EVP_CIPHER_CTX_flags(
    const WOLFSSL_EVP_CIPHER_CTX * ctx
)
```

Getter function for WOLFSSL_EVP_CIPHER_CTX structure. Deprecated v1.1.0.

Parameters:

- **ctx** structure to get flag.

See:

- wolfSSL_EVP_CIPHER_CTX_new
- wolfSSL_EVP_CIPHER_flags

Return: unsigned long of flags/mode.

Example

```
WOLFSSL_EVP_CIPHER_CTX* ctx;
unsigned long flags;
ctx = wolfSSL_EVP_CIPHER_CTX_new()
flags = wolfSSL_EVP_CIPHER_CTX_flags(ctx);
```

```
int wolfSSL_PEM_write_bio_PrivateKey(
    WOLFSSL_BIO * bio,
    WOLFSSL_EVP_PKEY * key,
    const WOLFSSL_EVP_CIPHER * cipher,
    unsigned char * passwd,
    int len,
    wc_pem_password_cb * cb,
    void * arg
)
```

This function writes a key into a WOLFSSL_BIO structure in PEM format.

Parameters:

- **bio** WOLFSSL_BIO structure to get PEM buffer from.
- **key** key to convert to PEM format.
- **cipher** EVP cipher structure.
- **passwd** password.
- **len** length of password.
- **cb** password callback.
- **arg** optional argument.

See: wolfSSL_PEM_read_bio_X509_AUX

Return:

- SSL_SUCCESS upon success.
- SSL_FAILURE upon failure.

Example

```
WOLFSSL_BIO* bio;
WOLFSSL_EVP_PKEY* key;
int ret;
// create bio and setup key
ret = wolfSSL_PEM_write_bio_PrivateKey(bio, key, NULL, NULL, 0, NULL, NULL);
//check ret value
```

```
int wolfSSL_CTX_use_RSAPrivateKey_file(
    WOLFSSL_CTX * ctx,
    const char * file,
    int format
)
```

This function loads the private RSA key used in the SSL connection into the SSL context (WOLFSSL_CTX). This function is only available when wolfSSL has been compiled with the OpenSSL compatibility layer enabled (–enable-opensslExtra, #define OPENSSL_EXTRA), and is identical to the more-typically used wolfSSL_CTX_use_PrivateKey_file() function. The file argument contains a pointer to the RSA private key file, in the format specified by format.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new()
- **file** a pointer to the name of the file containing the RSA private key to be loaded into the wolfSSL SSL context, with format as specified by format.
- **format** the encoding type of the RSA private key specified by file. Possible values include SSL_FILETYPE_PEM and SSL_FILETYPE_ASN1.

See:

- wolfSSL_CTX_use_PrivateKey_buffer
- wolfSSL_CTX_use_PrivateKey_file
- wolfSSL_use_RSAPrivateKey_file
- wolfSSL_use_PrivateKey_buffer
- wolfSSL_use_PrivateKey_file

Return:

- SSL_SUCCESS upon success.
- SSL_FAILURE If the function call fails, possible causes might include: The input key file is in the wrong format, or the wrong format has been given using the “format” argument, file doesn’t exist, can’t be read, or is corrupted, an out of memory condition occurs.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx;
...
ret = wolfSSL_CTX_use_RSAPrivateKey_file(ctx, “./server-key.pem”,
                                         SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
    // error loading private key file
}
...
```

```
int wolfSSL_use_certificate_file(
    WOLFSSL * ssl,
    const char * file,
    int format
)
```

This function loads a certificate file into the SSL session (WOLFSSL structure). The certificate file is provided by the file argument. The format argument specifies the format type of the file - either SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created with wolfSSL_new().

- **file** a pointer to the name of the file containing the certificate to be loaded into the wolfSSL SSL session, with format as specified by format.
- **format** the encoding type of the certificate specified by file. Possible values include SSL_FILETYPE_PEM and SSL_FILETYPE_ASN1.

See:

- wolfSSL_CTX_use_certificate_buffer
- wolfSSL_CTX_use_certificate_file
- wolfSSL_use_certificate_buffer

Return:

- SSL_SUCCESS upon success
- SSL_FAILURE If the function call fails, possible causes might include: The file is in the wrong format, or the wrong format has been given using the "format" argument, file doesn't exist, can't be read, or is corrupted, an out of memory condition occurs, Base16 decoding fails on the file

Example

```
int ret = 0;
WOLFSSL* ssl;
...
ret = wolfSSL_use_certificate_file(ssl, "./client-cert.pem",
                                   SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
    // error loading cert file
}
...
```

```
int wolfSSL_use_PrivateKey_file(
    WOLFSSL * ssl,
    const char * file,
    int format
)
```

This function loads a private key file into the SSL session (WOLFSSL structure). The key file is provided by the file argument. The format argument specifies the format type of the file - SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created with wolfSSL_new().
- **file** a pointer to the name of the file containing the key file to be loaded into the wolfSSL SSL session, with format as specified by format.
- **format** the encoding type of the key specified by file. Possible values include SSL_FILETYPE_PEM and SSL_FILETYPE_ASN1.

See:

- wolfSSL_CTX_use_PrivateKey_buffer
- wolfSSL_CTX_use_PrivateKey_file
- wolfSSL_use_PrivateKey_buffer
- wc_CryptoCb_RegisterDevice
- wolfSSL_SetDevId

Return:

- SSL_SUCCESS upon success.
- SSL_FAILURE If the function call fails, possible causes might include: The file is in the wrong format, or the wrong format has been given using the “format” argument, The file doesn’t exist, can’t be read, or is corrupted, An out of memory condition occurs, Base16 decoding fails on the file, The key file is encrypted but no password is provided

If using an external key store and do not have the private key you can instead provide the public key and register the crypro callback to handle the signing. For this you can build with either build with crypto callbacks or PK callbacks. To enable crypto callbacks use –enable-cryptocb or WOLF_CRYPTO_CB and register a crypto callback using wc_CryptoCb_RegisterDevice and set the associated devId using wolfSSL_SetDevId.

Example

```
int ret = 0;
WOLFSSL* ssl;
...
ret = wolfSSL_use_PrivateKey_file(ssl, “./server-key.pem”,
                                  SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
    // error loading key file
}
...
```

```
int wolfSSL_use_certificate_chain_file(
    WOLFSSL * ssl,
    const char * file
)
```

This function loads a chain of certificates into the SSL session (WOLFSSL structure). The file containing the certificate chain is provided by the file argument, and must contain PEM-formatted certificates. This function will process up to MAX_CHAIN_DEPTH (default = 9, defined in internal.h) certificates, plus the subject certificate.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new()
- **file** a pointer to the name of the file containing the chain of certificates to be loaded into the wolfSSL SSL session. Certificates must be in PEM format.

See:

- wolfSSL_CTX_use_certificate_chain_file

- wolfSSL_CTX_use_certificate_chain_buffer
- wolfSSL_use_certificate_chain_buffer

Return:

- SSL_SUCCESS upon success.
- SSL_FAILURE If the function call fails, possible causes might include: The file is in the wrong format, or the wrong format has been given using the “format” argument, file doesn’t exist, can’t be read, or is corrupted, an out of memory condition occurs

Example

```
int ret = 0;
WOLFSSL* ctx;
...
ret = wolfSSL_use_certificate_chain_file(ssl, “./cert-chain.pem”);
if (ret != SSL_SUCCESS) {
    // error loading cert file
}
...
```

```
int wolfSSL_use_RSAPrivateKey_file(
    WOLFSSL * ssl,
    const char * file,
    int format
)
```

This function loads the private RSA key used in the SSL connection into the SSL session (WOLFSSL structure). This function is only available when wolfSSL has been compiled with the OpenSSL compatibility layer enabled (–enable-opensslExtra, #define OPENSSL_EXTRA), and is identical to the more-typically used wolfSSL_use_PrivateKey_file() function. The file argument contains a pointer to the RSA private key file, in the format specified by format.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new()
- **file** a pointer to the name of the file containing the RSA private key to be loaded into the wolfSSL SSL session, with format as specified by format.
- **format** the encoding type of the RSA private key specified by file. Possible values include SSL_FILETYPE_PEM and SSL_FILETYPE_ASN1.

See:

- wolfSSL_CTX_use_RSAPrivateKey_file
- wolfSSL_CTX_use_PrivateKey_buffer
- wolfSSL_CTX_use_PrivateKey_file
- wolfSSL_use_PrivateKey_buffer
- wolfSSL_use_PrivateKey_file

Return:

- SSL_SUCCESS upon success
- SSL_FAILURE If the function call fails, possible causes might include: The input key file is in the wrong format, or the wrong format has been given using the “format” argument, file doesn’t exist, can’t be read, or is corrupted, an out of memory condition occurs

Example

```
int ret = 0;
WOLFSSL* ssl;
...
ret = wolfSSL_use_RSAPrivateKey_file(ssl, “./server-key.pem”,
                                     SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
    // error loading private key file
}
...
```

```
long wolfSSL_set_tlsext_status_type(
    WOLFSSL * s,
    int type
)
```

This function is called when the client application request that a server send back an OCSP status response (also known as OCSP stapling).Currently, the only supported type is TLSEXT_STATUSTYPE_ocsp.

Parameters:

- **s** pointer to WOLFSSL struct which is created by SSL_new() function
- **type** ssl extension type which TLSEXT_STATUSTYPE_ocsp is only supported.

See:

- wolfSSL_new
- wolfSSL_CTX_new
- wolfSSL_free
- wolfSSL_CTX_free

Return:

- 1 upon success.
- 0 upon error.

Example

```
WOLFSSL *ssl;
WOLFSSL_CTX *ctx;
int ret;
ctx = wolfSSL_CTX_new(wolfSSLv23_server_method());
```

```
ssl = wolfSSL_new(ctx);
ret = WolfSSL_set_tlsext_status_type(ssl,TLSEXT_STATUSTYPE_ocsp);
wolfSSL_free(ssl);
wolfSSL_CTX_free(ctx);
```

```
WOLFSSL_X509_CHAIN * wolfSSL_get_peer_chain(
    WOLFSSL * ssl
)
```

Retrieves the peer's certificate chain.

Parameters:

- **ssl** pointer to a valid WOLFSSL structure.

See:

- wolfSSL_get_chain_count
- wolfSSL_get_chain_length
- wolfSSL_get_chain_cert
- wolfSSL_get_chain_cert_pem

Return:

- chain If successful the call will return the peer's certificate chain.
- 0 will be returned if an invalid WOLFSSL pointer is passed to the function.

Example

none

```
int wolfSSL_get_chain_count(
    WOLFSSL_X509_CHAIN * chain
)
```

Retrieve's the peers certificate chain count.

Parameters:

- **chain** pointer to a valid WOLFSSL_X509_CHAIN structure.

See:

- wolfSSL_get_peer_chain
- wolfSSL_get_chain_length
- wolfSSL_get_chain_cert
- wolfSSL_get_chain_cert_pem

Return:

- Success If successful the call will return the peer's certificate chain count.
- 0 will be returned if an invalid chain pointer is passed to the function.

Example

none

```
int wolfSSL_get_chain_length(
    WOLFSSL_X509_CHAIN * chain,
    int idx
)
```

Retrieves the peer's ASN1.DER certificate length in bytes at index (idx).

Parameters:

- **chain** pointer to a valid WOLFSSL_X509_CHAIN structure.
- **idx** index to start of chain.

See:

- wolfSSL_get_peer_chain
- wolfSSL_get_chain_count
- wolfSSL_get_chain_cert
- wolfSSL_get_chain_cert_pem

Return:

- Success If successful the call will return the peer's certificate length in bytes by index.
- 0 will be returned if an invalid chain pointer is passed to the function.

Example

none

```
unsigned char * wolfSSL_get_chain_cert(
    WOLFSSL_X509_CHAIN * chain,
    int idx
)
```

Retrieves the peer's ASN1.DER certificate at index (idx).

Parameters:

- **chain** pointer to a valid WOLFSSL_X509_CHAIN structure.
- **idx** index to start of chain.

See:

- wolfSSL_get_peer_chain
- wolfSSL_get_chain_count
- wolfSSL_get_chain_length
- wolfSSL_get_chain_cert_pem

Return:

- Success If successful the call will return the peer's certificate by index.
- 0 will be returned if an invalid chain pointer is passed to the function.

Example

none

```
int wolfSSL_get_chain_cert_pem(
    WOLFSSL_X509_CHAIN * chain,
    int idx,
    unsigned char * buf,
    int inLen,
    int * outLen
)
```

Retrieves the peer's PEM certificate at index (idx).

Parameters:

- **chain** pointer to a valid WOLFSSL_X509_CHAIN structure.
- **idx** indexto start of chain.

See:

- wolfSSL_get_peer_chain
- wolfSSL_get_chain_count
- wolfSSL_get_chain_length
- wolfSSL_get_chain_cert

Return:

- Success If successful the call will return the peer's certificate by index.
- 0 will be returned if an invalid chain pointer is passed to the function.

Example

none

```
const unsigned char * wolfSSL_get_sessionID(
    const WOLFSSL_SESSION * s
)
```

Retrieves the session's ID. The session ID is always 32 bytes long.

Parameters:

- **session** pointer to a valid wolfssl session.

See: SSL_get_session

Return: id The session ID.

Example

none

```
int wolfSSL_X509_get_serial_number(
    WOLFSSL_X509 * x509,
    unsigned char * in,
    int * inOutSz
)
```

Retrieves the peer's certificate serial number. The serial number buffer (in) should be at least 32 bytes long and be provided as the *inOutSz argument as input. After calling the function* inOutSz will hold the actual length in bytes written to the in buffer.

Parameters:

- **in** The serial number buffer and should be at least 32 bytes long
- **inOutSz** will hold the actual length in bytes written to the in buffer.

See: SSL_get_peer_certificate

Return:

- SSL_SUCCESS upon success.
- BAD_FUNC_ARG will be returned if a bad function argument was encountered.

Example

none

```
WC_PKCS12 * wolfSSL_d2i_PKCS12_bio(
    WOLFSSL_BIO * bio,
    WC_PKCS12 ** pkcs12
)
```

wolfSSL_d2i_PKCS12_bio (d2i_PKCS12_bio) copies in the PKCS12 information from WOLFSSL_BIO to the structure WC_PKCS12. The information is divided up in the structure as a list of Content Infos along with a structure to hold optional MAC information. After the information has been divided into chunks (but not decrypted) in the structure WC_PKCS12, it can then be parsed and decrypted by calling.

Parameters:

- **bio** WOLFSSL_BIO structure to read PKCS12 buffer from.
- **pkcs12** WC_PKCS12 structure pointer for new PKCS12 structure created. Can be NULL

See:

- wolfSSL_PKCS12_parse
- wc_PKCS12_free

Return:

- WC_PKCS12 pointer to a WC_PKCS12 structure.
- Failure If function failed it will return NULL.

Example

```
WC_PKCS12* pkcs;
WOLFSSL_BIO* bio;
WOLFSSL_X509* cert;
WOLFSSL_EVP_PKEY* pkey;
STACK_OF(X509) certs;
//bio loads in PKCS12 file
wolfSSL_d2i_PKCS12_bio(bio, &pkcs);
wolfSSL_PKCS12_parse(pkcs, “a password”, &pkey, &cert, &certs)
wc_PKCS12_free(pkcs)
//use cert, pkey, and optionally certs stack
```

```
WC_PKCS12 * wolfSSL_i2d_PKCS12_bio(
    WOLFSSL_BIO * bio,
    WC_PKCS12 * pkcs12
)
```

wolfSSL_i2d_PKCS12_bio (i2d_PKCS12_bio) copies in the cert information from the structure WC_PKCS12 to WOLFSSL_BIO.

Parameters:

- **bio** WOLFSSL_BIO structure to write PKCS12 buffer to.
- **pkcs12** WC_PKCS12 structure for PKCS12 structure as input.

See:

- wolfSSL_PKCS12_parse
- wc_PKCS12_free

Return:

- 1 for success.
- Failure 0.

Example

```
WC_PKCS12 pkcs12;
FILE *f;
byte buffer[5300];
char file[] = "./test.p12";
int bytes;
WOLFSSL_BIO* bio;
pkcs12 = wc_PKCS12_new();
f = fopen(file, "rb");
bytes = (int)fread(buffer, 1, sizeof(buffer), f);
fclose(f);
//convert the DER file into an internal structure
wc_d2i_PKCS12(buffer, bytes, pkcs12);
bio = wolfSSL_BIO_new(wolfSSL_BIO_s_mem());
//convert PKCS12 structure into bio
wolfSSL_i2d_PKCS12_bio(bio, pkcs12);
wc_PKCS12_free(pkcs)
//use bio
```

```
int wolfSSL_PKCS12_parse(
    WC_PKCS12 * pkcs12,
    const char * psw,
    WOLFSSL_EVP_PKEY ** pkey,
    WOLFSSL_X509 ** cert,
    WOLF_STACK_OF(WOLFSSL_X509) ** ca
)
```

PKCS12 can be enabled with adding –enable-opensslextra to the configure command. It can use triple DES and RC4 for decryption so would recommend also enabling these features when enabling opensslextra (–enable-des3 –enable-arc4). wolfSSL does not currently support RC2 so decryption with RC2 is currently not available. This may be noticeable with default encryption schemes used by OpenSSL command line to create .p12 files. wolfSSL_PKCS12_parse (PKCS12_parse). The first thing this function does is check the MAC is correct if present. If the MAC fails then the function returns and does not try to decrypt any of the stored Content Infos. This function then parses through each Content Info looking for a bag type, if the bag type is known it is decrypted as needed and either stored in the list of certificates being built or as a key found. After parsing through all bags the key found is then compared with the certificate list until a matching pair is found. This matching pair is then returned as the key and certificate, optionally the certificate list found is returned as a STACK_OF certificates. At the moment a CRL, Secret or SafeContents bag will be skipped over and not parsed.

It can be seen if these or other “Unknown” bags are skipped over by viewing the debug print out. Additional attributes such as friendly name are skipped over when parsing a PKCS12 file.

Parameters:

- **pkcs12** WC_PKCS12 structure to parse.
- **paswd** password for decrypting PKCS12.
- **pkey** structure to hold private key decoded from PKCS12.
- **cert** structure to hold certificate decoded from PKCS12.
- **stack** optional stack of extra certificates.

See:

- wolfSSL_d2i_PKCS12_bio
- wc_PKCS12_free

Return:

- SSL_SUCCESS On successfully parsing PKCS12.
- SSL_FAILURE If an error case was encountered.

Example

```
WC_PKCS12* pkcs;
WOLFSSL_BIO* bio;
WOLFSSL_X509* cert;
WOLFSSL_EVP_PKEY* pkey;
STACK_OF(X509) certs;
//bio loads in PKCS12 file
wolfSSL_d2i_PKCS12_bio(bio, &pkcs);
wolfSSL_PKCS12_parse(pkcs, “a password”, &pkey, &cert, &certs)
wc_PKCS12_free(pkcs)
//use cert, pkey, and optionally certs stack
```

A.4 wolfSSL Certificates and Keys

A.3.2.33 function wolfSSL_PKCS12_parse

A.4.1 Functions

	Name
int	**wc_KeyPemToDer**(const unsigned char * pem, int pemSz, unsigned char * buff, int buffSz, const char * pass)Converts a key in PEM format to DER format.

	Name
int	**wc_CertPemToDer**(const unsigned char * pem, int pemSz, unsigned char * buff, int buffSz, int type)This function converts a PEM formatted certificate to DER format. Calls OpenSSL function PemToDer.
int	**wc_GetPubKeyDerFromCert before calling wc_InitDecodedCert().
int	**wolfSSL_CTX_use_certificate_file**(WOLFSSL_CTX * ctx, const char * file, int format)This function loads a certificate file into the SSL context (WOLFSSL_CTX). The file is provided by the file argument. The format argument specifies the format type of the file, either SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM. Please see the examples for proper usage.
int	**wolfSSL_CTX_use_PrivateKey_file**(WOLFSSL_CTX * ctx, const char * file, int format)This function loads a private key file into the SSL context (WOLFSSL_CTX). The file is provided by the file argument. The format argument specifies the format type of the file - SSL_FILETYPE_ASN1or SSL_FILETYPE_PEM. Please see the examples for proper usage.
int	**wolfSSL_CTX_load_verify_locations**(WOLFSSL_CTX * ctx, const char * file, const char * path)This function loads PEM_formatted CA certificate files into the SSL context (WOLFSSL_CTX). These certificates will be treated as trusted root certificates and used to verify certs received from peers during the SSL handshake. The root certificate file, provided by the file argument, may be a single certificate or a file containing multiple certificates. If multiple CA certs are included in the same file, wolfSSL will load them in the same order they are presented in the file. The path argument is a pointer to the name of a directory that contains certificates of trusted root CAs. If the value of file is not NULL, path may be specified as NULL if not needed. If path is specified and NO_WOLFSSL_DIR was not defined when building the library, wolfSSL will load all CA certificates located in the given directory. This function will attempt to load all files in the directory. This function expects PEM formatted CERT_TYPE file with header “—–BEGIN CERTIFICATE—–”.

	Name
int	**wolfSSL_CTX_load_verify_locations_ex**(WOLFSSL_CTX * ctx, const char * file, const char * path, unsigned int flags)This function loads PEM_formatted CA certificate files into the SSL context (WOLFSSL_CTX). These certificates will be treated as trusted root certificates and used to verify certs received from peers during the SSL handshake. The root certificate file, provided by the file argument, may be a single certificate or a file containing multiple certificates. If multiple CA certs are included in the same file, wolfSSL will load them in the same order they are presented in the file. The path argument is a pointer to the name of a directory that contains certificates of trusted root CAs. If the value of file is not NULL, path may be specified as NULL if not needed. If path is specified and NO_WOLFSSL_DIR was not defined when building the library, wolfSSL will load all CA certificates located in the given directory. This function will attempt to load all files in the directory based on flags specified. This function expects PEM formatted CERT_TYPE files with header "—–BEGIN CERTIFICATE—–".
const char **	**wolfSSL_get_system_CA_dirs**(word32 * num)This function returns a pointer to an array of strings representing directories wolfSSL will search for system CA certs when wolfSSL_CTX_load_system_CA_certs is called. On systems that don't store certificates in an accessible system directory (such as Apple platforms), this function will always return NULL.
int	**wolfSSL_CTX_load_system_CA_certs**(WOLFSSL_CTX * ctx)On most platforms (including Linux and Windows), this function attempts to load CA certificates into a WOLFSSL_CTX from an OS-dependent CA certificate store. Loaded certificates will be trusted.
int	**wolfSSL_CTX_use_certificate_chain_file**(WOLFSSL_CTX * ctx, const char * file)This function loads a chain of certificates into the SSL context (WOLFSSL_CTX). The file containing the certificate chain is provided by the file argument, and must contain PEM_formatted certificates. This function will process up to MAX_CHAIN_DEPTH (default = 9, defined in internal.h) certificates, plus the subject cert.

	Name
int	**wolfSSL_CTX_der_load_verify_locations**(WOLFSSL_CTX * ctx, const char * file, int format)This function is similar to wolfSSL_CTX_load_verify_locations, but allows the loading of DER_formatted CA files into the SSL context (WOLFSSL_CTX). It may still be used to load PEM_formatted CA files as well. These certificates will be treated as trusted root certificates and used to verify certs received from peers during the SSL handshake. The root certificate file, provided by the file argument, may be a single certificate or a file containing multiple certificates. If multiple CA certs are included in the same file, wolfSSL will load them in the same order they are presented in the file. The format argument specifies the format which the certificates are in either, SSL_FILETYPE_PEM or SSL_FILETYPE_ASN1 (DER). Unlike wolfSSL_CTX_load_verify_locations, this function does not allow the loading of CA certificates from a given directory path. Note that this function is only available when the wolfSSL library was compiled with WOLFSSL_DER_LOAD defined.
void	**wolfSSL_SetCertCbCtx**(WOLFSSL * ssl, void * ctx)This function stores user CTX object information for verify callback.
void	**wolfSSL_CTX_SetCertCbCtx**(WOLFSSL_CTX * ctx, void * userCtx)This function stores user CTX object information for verify callback.
int	**wolfSSL_CTX_save_cert_cache**(WOLFSSL_CTX * ctx, const char * fname)This function writes the cert cache from memory to file.
int	**wolfSSL_CTX_restore_cert_cache**(WOLFSSL_CTX * ctx, const char * fname)This function persistes certificate cache from a file.
int	**wolfSSL_CTX_memsave_cert_cache**(WOLFSSL_CTX * ctx, void * mem, int sz, int * used)This function persists the certificate cache to memory.
int	**wolfSSL_CTX_get_cert_cache_memsize**(WOLFSSL_CTX * ctx)Returns the size the certificate cache save buffer needs to be.
char *	**wolfSSL_X509_NAME_oneline**(WOLFSSL_X509_NAME * name, char * in, int sz)This function copies the name of the x509 into a buffer.
WOLFSSL_X509_NAME *	**wolfSSL_X509_get_issuer_name**(WOLFSSL_X509 * cert)This function returns the name of the certificate issuer.
WOLFSSL_X509_NAME *	**wolfSSL_X509_get_subject_name**(WOLFSSL_X509 * cert)This function returns the subject member of the WOLFSSL_X509 structure.

	Name
int	**wolfSSL_X509_get_isCA**(WOLFSSL_X509 * cert)Checks the isCa member of the WOLFSSL_X509 structure and returns the value.
int	**wolfSSL_X509_NAME_get_text_by_NID**(WOLFSSL_X509_NAME * name, int nid, char * buf, int len)This function gets the text related to the passed in NID value.
int	**wolfSSL_X509_get_signature_type**(WOLFSSL_X509 * cert)This function returns the value stored in the sigOID member of the WOLFSSL_X509 structure.
int	**wolfSSL_X509_get_signature**(WOLFSSL_X509 * x509, unsigned char * buf, int * bufSz)Gets the X509 signature and stores it in the buffer.
int	**wolfSSL_X509_STORE_add_cert**(WOLFSSL_X509_STORE * store, WOLFSSL_X509 * x509)This function adds a certificate to the WOLFSSL_X509_STRE structure.
WOLFSSL_STACK *	**wolfSSL_X509_STORE_CTX_get_chain**(WOLFSSL_X509_STORE_CT * ctx)This function is a getter function for chain variable in WOLFSSL_X509_STORE_CTX structure. Currently chain is not populated.
int	**wolfSSL_X509_STORE_set_flags**(WOLFSSL_X509_STORE * store, unsigned long flag)This function takes in a flag to change the behavior of the WOLFSSL_X509_STORE structure passed in. An example of a flag used is WOLFSSL_CRL_CHECK.
const byte *	**wolfSSL_X509_notBefore**(WOLFSSL_X509 * x509)This function the certificate “not before” validity encoded as a byte array.
const byte *	**wolfSSL_X509_notAfter**(WOLFSSL_X509 * x509)This function the certificate “not after” validity encoded as a byte array.
const char *	**wolfSSL_get_psk_identity_hint**(const WOLFSSL *)This function returns the psk identity hint.
const char *	**wolfSSL_get_psk_identity**(const WOLFSSL *)The function returns a constant pointer to the client_identity member of the Arrays structure.
int	**wolfSSL_CTX_use_psk_identity_hint**(WOLFSSL_CTX * ctx, const char * hint)This function stores the hint argument in the server_hint member of the WOLFSSL_CTX structure.
int	**wolfSSL_use_psk_identity_hint**(WOLFSSL * ssl, const char * hint)This function stores the hint argument in the server_hint member of the Arrays structure within the WOLFSSL structure.
WOLFSSL_X509 *	**wolfSSL_get_peer_certificate**(WOLFSSL * ssl)This function gets the peer’s certificate.
WOLFSSL_X509 *	**wolfSSL_get_chain_X509**(WOLFSSL_X509_CHAIN * chain, int idx)This function gets the peer’s wolfSSL_X509_certificate at index (idx) from the chain of certificates.

	Name
char *	**wolfSSL_X509_get_subjectCN**(WOLFSSL_X509 *)Returns the common name of the subject from the certificate.
const unsigned char *	**wolfSSL_X509_get_der**(WOLFSSL_X509 * x509, int * outSz)This function gets the DER encoded certificate in the WOLFSSL_X509 struct.
WOLFSSL_ASN1_TIME *	**wolfSSL_X509_get_notAfter**(WOLFSSL_X509 *)This function checks to see if x509 is NULL and if it's not, it returns the notAfter member of the x509 struct.
int	**wolfSSL_X509_version**(WOLFSSL_X509 *)This function retrieves the version of the X509 certificate.
WOLFSSL_X509 *	**wolfSSL_X509_d2i_fp**(WOLFSSL_X509 ** x509, FILE * file)If NO_STDIO_FILESYSTEM is defined this function will allocate heap memory, initialize a WOLFSSL_X509 structure and return a pointer to it.
WOLFSSL_X509 *	**wolfSSL_X509_load_certificate_file**(const char * fname, int format)The function loads the x509 certificate into memory.
unsigned char *	**wolfSSL_X509_get_device_type**(WOLFSSL_X509 * x509, unsigned char * in, int * inOutSz)This function copies the device type from the x509 structure to the buffer.
unsigned char *	**wolfSSL_X509_get_hw_type**(WOLFSSL_X509 * x509, unsigned char * in, int * inOutSz)The function copies the hwType member of the WOLFSSL_X509 structure to the buffer.
unsigned char *	**wolfSSL_X509_get_hw_serial_number**(WOLFSSL_X509 * x509, unsigned char * in, int * inOutSz)This function returns the hwSerialNum member of the x509 object.
int	**wolfSSL_SetTmpDH**(WOLFSSL * ssl, const unsigned char * p, int pSz, const unsigned char * g, int gSz)Server Diffie-Hellman Ephemeral parameters setting. This function sets up the group parameters to be used if the server negotiates a cipher suite that uses DHE.
int	**wolfSSL_SetTmpDH_buffer**(WOLFSSL * ssl, const unsigned char * b, long sz, int format)The function calls the wolfSSL_SetTMpDH_buffer_wrapper, which is a wrapper for Diffie-Hellman parameters.
int	**wolfSSL_SetTmpDH_file**(WOLFSSL * ssl, const char * f, int format)This function calls wolfSSL_SetTmpDH_file_wrapper to set server Diffie-Hellman parameters.
int	**wolfSSL_CTX_SetTmpDH**(WOLFSSL_CTX * ctx, const unsigned char * p, int pSz, const unsigned char * g, int gSz)Sets the parameters for the server CTX Diffie-Hellman.

	Name
int	**wolfSSL_CTX_SetTmpDH_buffer**(WOLFSSL_CTX * ctx, const unsigned char * b, long sz, int format)A wrapper function that calls wolfSSL_SetTmpDH_buffer_wrapper.
int	**wolfSSL_CTX_SetTmpDH_file**(WOLFSSL_CTX * ctx, const char * f, int format)The function calls wolfSSL_SetTmpDH_file_wrapper to set the server Diffie-Hellman parameters.
int	**wolfSSL_CTX_SetMinDhKey_Sz**(WOLFSSL_CTX * ctx, word16)This function sets the minimum size (in bits) of the Diffie Hellman key size by accessing the minDhKeySz member in the WOLFSSL_CTX structure.
int	**wolfSSL_SetMinDhKey_Sz**(WOLFSSL * ssl, word16 keySz_bits)Sets the minimum size (in bits) for a Diffie-Hellman key in the WOLFSSL structure.
int	**wolfSSL_CTX_SetMaxDhKey_Sz**(WOLFSSL_CTX * ctx, word16 keySz_bits)This function sets the maximum size (in bits) of the Diffie Hellman key size by accessing the maxDhKeySz member in the WOLFSSL_CTX structure.
int	**wolfSSL_SetMaxDhKey_Sz**(WOLFSSL * ssl, word16 keySz_bits)Sets the maximum size (in bits) for a Diffie-Hellman key in the WOLFSSL structure.
int	**wolfSSL_GetDhKey_Sz**(WOLFSSL *)Returns the value of dhKeySz (in bits) that is a member of the options structure. This value represents the Diffie-Hellman key size in bytes.
int	**wolfSSL_CTX_SetMinRsaKey_Sz**(WOLFSSL_CTX * ctx, short keySz)Sets the minimum RSA key size in both the WOLFSSL_CTX structure and the WOLFSSL_CERT_MANAGER structure.
int	**wolfSSL_SetMinRsaKey_Sz**(WOLFSSL * ssl, short keySz)Sets the minimum allowable key size in bits for RSA located in the WOLFSSL structure.
int	**wolfSSL_CTX_SetMinEccKey_Sz**(WOLFSSL_CTX * ssl, short keySz)Sets the minimum size in bits for the ECC key in the WOLF_CTX structure and the WOLFSSL_CERT_MANAGER structure.
int	**wolfSSL_SetMinEccKey_Sz**(WOLFSSL * ssl, short keySz)Sets the value of the minEccKeySz member of the options structure. The options struct is a member of the WOLFSSL structure and is accessed through the ssl parameter.
int	**wolfSSL_make_eap_keys**(WOLFSSL * ssl, void * key, unsigned int len, const char * label)This function is used by EAP_TLS and EAP-TTLS to derive keying material from the master secret.

	Name
int	**wolfSSL_CTX_load_verify_buffer**(WOLFSSL_CTX * ctx, const unsigned char * in, long sz, int format)This function loads a CA certificate buffer into the WOLFSSL Context. It behaves like the non-buffered version, only differing in its ability to be called with a buffer as input instead of a file. The buffer is provided by the in argument of size sz. format specifies the format type of the buffer; SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM. More than one CA certificate may be loaded per buffer as long as the format is in PEM. Please see the examples for proper usage.
int	**wolfSSL_CTX_load_verify_buffer_ex**(WOLFSSL_CTX * ctx, const unsigned char * in, long sz, int format, int userChain, word32 flags)This function loads a CA certificate buffer into the WOLFSSL Context. It behaves like the non-buffered version, only differing in its ability to be called with a buffer as input instead of a file. The buffer is provided by the in argument of size sz. format specifies the format type of the buffer; SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM. More than one CA certificate may be loaded per buffer as long as the format is in PEM. The _ex version was added in PR 2413 and supports additional arguments for userChain and flags.
int	**wolfSSL_CTX_load_verify_chain_buffer_format**(WOLFSSL_CTX * ctx, const unsigned char * in, long sz, int format)This function loads a CA certificate chain buffer into the WOLFSSL Context. It behaves like the non-buffered version, only differing in its ability to be called with a buffer as input instead of a file. The buffer is provided by the in argument of size sz. format specifies the format type of the buffer; SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM. More than one CA certificate may be loaded per buffer as long as the format is in PEM. Please see the examples for proper usage.
int	**wolfSSL_CTX_use_certificate_buffer**(WOLFSSL_CTX * ctx, const unsigned char * in, long sz, int format)This function loads a certificate buffer into the WOLFSSL Context. It behaves like the non-buffered version, only differing in its ability to be called with a buffer as input instead of a file. The buffer is provided by the in argument of size sz. format specifies the format type of the buffer; SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM. Please see the examples for proper usage.

	Name
int	**wolfSSL_CTX_use_PrivateKey_buffer**(WOLFSSL_CTX * ctx, const unsigned char * in, long sz, int format)This function loads a private key buffer into the SSL Context. It behaves like the non-buffered version, only differing in its ability to be called with a buffer as input instead of a file. The buffer is provided by the in argument of size sz. format specifies the format type of the buffer; SSL_FILETYPE_ASN1or SSL_FILETYPE_PEM. Please see the examples for proper usage.
int	**wolfSSL_CTX_use_certificate_chain_buffer**(WOLFSSL_CTX * ctx, const unsigned char * in, long sz)This function loads a certificate chain buffer into the WOLFSSL Context. It behaves like the non-buffered version, only differing in its ability to be called with a buffer as input instead of a file. The buffer is provided by the in argument of size sz. The buffer must be in PEM format and start with the subject's certificate, ending with the root certificate. Please see the examples for proper usage.
int	**wolfSSL_use_certificate_buffer**(WOLFSSL * ssl, const unsigned char * in, long sz, int format)This function loads a certificate buffer into the WOLFSSL object. It behaves like the non-buffered version, only differing in its ability to be called with a buffer as input instead of a file. The buffer is provided by the in argument of size sz. format specifies the format type of the buffer; SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM. Please see the examples for proper usage.
int	**wolfSSL_use_PrivateKey_buffer**(WOLFSSL * ssl, const unsigned char * in, long sz, int format)This function loads a private key buffer into the WOLFSSL object. It behaves like the non-buffered version, only differing in its ability to be called with a buffer as input instead of a file. The buffer is provided by the in argument of size sz. format specifies the format type of the buffer; SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM. Please see the examples for proper usage.

	Name
int	**wolfSSL_use_certificate_chain_buffer**(WOLFSSL * ssl, const unsigned char * in, long sz)This function loads a certificate chain buffer into the WOLFSSL object. It behaves like the non-buffered version, only differing in its ability to be called with a buffer as input instead of a file. The buffer is provided by the in argument of size sz. The buffer must be in PEM format and start with the subject's certificate, ending with the root certificate. Please see the examples for proper usage.
int	**wolfSSL_UnloadCertsKeys**(WOLFSSL *)This function unloads any certificates or keys that SSL owns.
int	**wolfSSL_GetIVSize**(WOLFSSL *)Returns the iv_size member of the specs structure held in the WOLFSSL struct.
void	**wolfSSL_KeepArrays**(WOLFSSL *)Normally, at the end of the SSL handshake, wolfSSL frees temporary arrays. Calling this function before the handshake begins will prevent wolfSSL from freeing temporary arrays. Temporary arrays may be needed for things such as wolfSSL_get_keys() or PSK hints. When the user is done with temporary arrays, either wolfSSL_FreeArrays() may be called to free the resources immediately, or alternatively the resources will be freed when the associated SSL object is freed.
void	**wolfSSL_FreeArrays has been called before the handshake, wolfSSL will not free temporary arrays. This function explicitly frees temporary arrays and should be called when the user is done with temporary arrays and does not want to wait for the SSL object to be freed to free these resources.
int	**wolfSSL_DeriveTlsKeys**(unsigned char * key_data, word32 keyLen, const unsigned char * ms, word32 msLen, const unsigned char * sr, const unsigned char * cr, int tls1_2, int hash_type)An external facing wrapper to derive TLS Keys.
int	**wolfSSL_X509_get_ext_by_NID**(const WOLFSSL_X509 * x509, int nid, int lastPos)This function looks for and returns the extension index matching the passed in NID value.
void *	**wolfSSL_X509_get_ext_d2i**(const WOLFSSL_X509 * x509, int nid, int * c, int * idx)This function looks for and returns the extension matching the passed in NID value.

	Name
int	**wolfSSL_X509_digest**(const WOLFSSL_X509 * x509, const WOLFSSL_EVP_MD * digest, unsigned char * buf, unsigned int * len)This function returns the hash of the DER certificate.
int	**wolfSSL_use_PrivateKey**(WOLFSSL * ssl, WOLFSSL_EVP_PKEY * pkey)This is used to set the private key for the WOLFSSL structure.
int	**wolfSSL_use_PrivateKey_ASN1**(int pri, WOLFSSL * ssl, unsigned char * der, long derSz)This is used to set the private key for the WOLFSSL structure. A DER formatted key buffer is expected.
int	**wolfSSL_use_RSAPrivateKey_ASN1**(WOLFSSL * ssl, unsigned char * der, long derSz)This is used to set the private key for the WOLFSSL structure. A DER formatted RSA key buffer is expected.
WOLFSSL_DH *	**wolfSSL_DSA_dup_DH**(const WOLFSSL_DSA * r)This function duplicates the parameters in dsa to a newly created WOLFSSL_DH structure.
WOLFSSL_X509 *	**wolfSSL_d2i_X509_bio**(WOLFSSL_BIO * bio, WOLFSSL_X509 ** x509)This function get the DER buffer from bio and converts it to a WOLFSSL_X509 structure.
WOLFSSL_X509 *	**wolfSSL_PEM_read_bio_X509_AUX**(WOLFSSL_BIO * bp, WOLFSSL_X509 ** x, wc_pem_password_cb * cb, void * u)This function behaves the same as wolfSSL_PEM_read_bio_X509. AUX signifies containing extra information such as trusted/rejected use cases and friendly name for human readability.
long	**wolfSSL_CTX_set_tmp_dh**(WOLFSSL_CTX * ctx, WOLFSSL_DH * dh)Initializes the WOLFSSL_CTX structure's dh member with the Diffie-Hellman parameters.
WOLFSSL_DSA *	**wolfSSL_PEM_read_bio_DSAparams**(WOLFSSL_BIO * bp, WOLFSSL_DSA ** x, wc_pem_password_cb * cb, void * u)This function get the DSA parameters from a PEM buffer in bio.
	WOLF_STACK_OF(WOLFSSL_X509) constThis function gets the peer's certificate chain.
char *	**wolfSSL_X509_get_next_altname**(WOLFSSL_X509 *)This function returns the next, if any, altname from the peer certificate.
WOLFSSL_ASN1_TIME *	**wolfSSL_X509_get_notBefore**(WOLFSSL_X509 *)The function checks to see if x509 is NULL and if it's not, it returns the notBefore member of the x509 struct.

A.4.2 Functions Documentation

```
int wc_KeyPemToDer(
    const unsigned char * pem,
    int pemSz,
    unsigned char * buff,
    int buffSz,
    const char * pass
)
```

Converts a key in PEM format to DER format.

Parameters:

- **pem** a pointer to the PEM encoded certificate.
- **pemSz** the size of the PEM buffer (pem)
- **buff** a pointer to the copy of the buffer member of the DerBuffer struct.
- **buffSz** size of the buffer space allocated in the DerBuffer struct.
- **pass** password passed into the function.

See: wc_PemToDer

Return:

- int the function returns the number of bytes written to the buffer on successful execution.
- int negative int returned indicating an error.

Example

```
byte* loadBuf;
long fileSz = 0;
byte* bufSz;
static int LoadKeyFile(byte** keyBuf, word32* keyBufSz,
const char* keyFile,
                int typeKey, const char* password);
…
bufSz = wc_KeyPemToDer(loadBuf, (int)fileSz, saveBuf,
(int)fileSz, password);

if(saveBufSz > 0){
    // Bytes were written to the buffer.
}
```

```
int wc_CertPemToDer(
    const unsigned char * pem,
    int pemSz,
    unsigned char * buff,
    int buffSz,
    int type
)
```

This function converts a PEM formatted certificate to DER format. Calls OpenSSL function PemToDer.

Parameters:

- **pem** pointer PEM formatted certificate.
- **pemSz** size of the certificate.
- **buff** buffer to be copied to DER format.
- **buffSz** size of the buffer.
- **type** Certificate file type found in asn_public.h enum CertType.

See: wc_PemToDer

Return: buffer returns the bytes written to the buffer.

Example

```
const unsigned char* pem;
int pemSz;
unsigned char buff[BUFSIZE];
int buffSz = sizeof(buff)/sizeof(char);
int type;
...
if(wc_CertPemToDer(pem, pemSz, buff, buffSz, type) <= 0) {
    // There were bytes written to buffer
}
```

```
int wc_GetPubKeyDerFromCert(
    struct DecodedCert * cert,
    byte * derKey,
    word32 * derKeySz
)
```

This function gets the public key in DER format from a populated DecodedCert struct. Users must call wc_InitDecodedCert() and wc_ParseCert() before calling this API. wc_InitDecodedCert() accepts a DER/ASN.1 encoded certificate. To convert a PEM cert to DER, first use wc_CertPemToDer() before calling wc_InitDecodedCert().

Parameters:

- **cert** populated DecodedCert struct holding X.509 certificate
- **derKey** output buffer to place DER encoded public key
- **derKeySz** [IN/OUT] size of derKey buffer on input, size of public key on return. If derKey is passed in as NULL, derKeySz will be set to required buffer size for public key and LENGTH_ONLY_E will be returned from function.

See: wc_GetPubKeyDerFromCert

Return: 0 on success, negative on error. LENGTH_ONLY_E if derKey is NULL and returning length only.

```
int wolfSSL_CTX_use_certificate_file(
    WOLFSSL_CTX * ctx,
    const char * file,
    int format
)
```

This function loads a certificate file into the SSL context (WOLFSSL_CTX). The file is provided by the file argument. The format argument specifies the format type of the file, either SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM. Please see the examples for proper usage.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new()
- **file** a pointer to the name of the file containing the certificate to be loaded into the wolfSSL SSL context.
- **format** - format of the certificates pointed to by file. Possible options are SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM.

See:

- wolfSSL_CTX_use_certificate_buffer
- wolfSSL_use_certificate_file
- wolfSSL_use_certificate_buffer

Return:

- SSL_SUCCESS upon success.
- SSL_FAILURE If the function call fails, possible causes might include the file is in the wrong format, or the wrong format has been given using the “format” argument, file doesn’t exist, can’t be read, or is corrupted, an out of memory condition occurs, Base16 decoding fails on the file.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx;
...
ret = wolfSSL_CTX_use_certificate_file(ctx, “./client-cert.pem”,
                                       SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
    // error loading cert file
}
...
```

```
int wolfSSL_CTX_use_PrivateKey_file(
    WOLFSSL_CTX * ctx,
    const char * file,
    int format
)
```

This function loads a private key file into the SSL context (WOLFSSL_CTX). The file is provided by the file argument. The format argument specifies the format type of the file - SSL_FILETYPE_ASN1or SSL_FILETYPE_PEM. Please see the examples for proper usage.

Parameters:

- **none** No parameters.

See:

- wolfSSL_CTX_use_PrivateKey_buffer
- wolfSSL_use_PrivateKey_file
- wolfSSL_use_PrivateKey_buffer
- wc_CryptoCb_RegisterDevice
- wolfSSL_CTX_SetDevId

Return:

- SSL_SUCCESS upon success.
- SSL_FAILURE The file is in the wrong format, or the wrong format has been given using the “format” argument. The file doesn’t exist, can’t be read, or is corrupted. An out of memory condition occurs. Base16 decoding fails on the file. The key file is encrypted but no password is provided.

If using an external key store and do not have the private key you can instead provide the public key and register the crypro callback to handle the signing. For this you can build with either build with crypto callbacks or PK callbacks. To enable crypto callbacks use –enable-cryptocb or WOLF_CRYPTO_CB and register a crypto callback using wc_CryptoCb_RegisterDevice and set the associated devId using wolfSSL_CTX_SetDevId.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx;
...
ret = wolfSSL_CTX_use_PrivateKey_file(ctx, “./server-key.pem”,
                                      SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
    // error loading key file
}
...
```

```
int wolfSSL_CTX_load_verify_locations(
    WOLFSSL_CTX * ctx,
    const char * file,
    const char * path
)
```

This function loads PEM-formatted CA certificate files into the SSL context (WOLFSSL_CTX). These certificates will be treated as trusted root certificates and used to verify certs received from peers during the SSL handshake. The root certificate file, provided by the file argument, may be a single certificate or a file containing multiple certificates. If multiple CA certs are included in the same file, wolfSSL will load them in the same order they are presented in the file. The path argument is a pointer to the name of a directory that contains certificates of trusted root CAs. If the value of file is not NULL, path may be specified as NULL if not needed. If path is specified and NO_WOLFSSL_DIR was not defined when building the library, wolfSSL will load all CA certificates located in the given directory. This function will attempt to load all files in the directory. This function expects PEM formatted CERT_TYPE file with header “—–BEGIN CERTIFICATE—–”.

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().
- **file** pointer to name of the file containing PEM-formatted CA certificates.
- **path** pointer to the name of a directory to load PEM-formatted certificates from.

See:

- wolfSSL_CTX_load_verify_locations_ex
- wolfSSL_CTX_load_verify_buffer
- wolfSSL_CTX_use_certificate_file
- wolfSSL_CTX_use_PrivateKey_file
- wolfSSL_CTX_use_certificate_chain_file
- wolfSSL_use_certificate_file
- wolfSSL_use_PrivateKey_file
- wolfSSL_use_certificate_chain_file

Return:

- SSL_SUCCESS up success.
- SSL_FAILURE will be returned if ctx is NULL, or if both file and path are NULL.
- SSL_BAD_FILETYPE will be returned if the file is the wrong format.
- SSL_BAD_FILE will be returned if the file doesn't exist, can't be read, or is corrupted.
- MEMORY_E will be returned if an out of memory condition occurs.
- ASN_INPUT_E will be returned if Base16 decoding fails on the file.
- ASN_BEFORE_DATE_E will be returned if the current date is before the before date.
- ASN_AFTER_DATE_E will be returned if the current date is after the after date.
- BUFFER_E will be returned if a chain buffer is bigger than the receiving buffer.
- BAD_PATH_ERROR will be returned if opendir() fails when trying to open path.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx;
...
ret = wolfSSL_CTX_load_verify_locations(ctx, "./ca-cert.pem", NULL);
if (ret != WOLFSSL_SUCCESS) {
    // error loading CA certs
}
...
```

```
int wolfSSL_CTX_load_verify_locations_ex(
    WOLFSSL_CTX * ctx,
    const char * file,
    const char * path,
    unsigned int flags
)
```

This function loads PEM-formatted CA certificate files into the SSL context (WOLFSSL_CTX). These certificates will be treated as trusted root certificates and used to verify certs received from peers during the SSL handshake. The root certificate file, provided by the file argument, may be a single certificate

or a file containing multiple certificates. If multiple CA certs are included in the same file, wolfSSL will load them in the same order they are presented in the file. The path argument is a pointer to the name of a directory that contains certificates of trusted root CAs. If the value of file is not NULL, path may be specified as NULL if not needed. If path is specified and NO_WOLFSSL_DIR was not defined when building the library, wolfSSL will load all CA certificates located in the given directory. This function will attempt to load all files in the directory based on flags specified. This function expects PEM formatted CERT_TYPE files with header "—–BEGIN CERTIFICATE—–".

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().
- **file** pointer to name of the file containing PEM-formatted CA certificates.
- **path** pointer to the name of a directory to load PEM-formatted certificates from.
- **flags** possible mask values are: WOLFSSL_LOAD_FLAG_IGNORE_ERR, WOLFSSL_LOAD_FLAG_DATE_ERR_OKAY and WOLFSSL_LOAD_FLAG_PEM_CA_ONLY

See:

- wolfSSL_CTX_load_verify_locations
- wolfSSL_CTX_load_verify_buffer
- wolfSSL_CTX_use_certificate_file
- wolfSSL_CTX_use_PrivateKey_file
- wolfSSL_CTX_use_certificate_chain_file
- wolfSSL_use_certificate_file
- wolfSSL_use_PrivateKey_file
- wolfSSL_use_certificate_chain_file

Return:

- SSL_SUCCESS up success.
- SSL_FAILURE will be returned if ctx is NULL, or if both file and path are NULL. This will also be returned if at least one cert is loaded successfully but there is one or more that failed. Check error stack for reason.
- SSL_BAD_FILETYPE will be returned if the file is the wrong format.
- SSL_BAD_FILE will be returned if the file doesn't exist, can't be read, or is corrupted.
- MEMORY_E will be returned if an out of memory condition occurs.
- ASN_INPUT_E will be returned if Base16 decoding fails on the file.
- BUFFER_E will be returned if a chain buffer is bigger than the receiving buffer.
- BAD_PATH_ERROR will be returned if opendir() fails when trying to open path.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx;
...
ret = wolfSSL_CTX_load_verify_locations_ex(ctx, NULL, "./certs/external",
    WOLFSSL_LOAD_FLAG_PEM_CA_ONLY);
if (ret != WOLFSSL_SUCCESS) {
    // error loading CA certs
}
...
```

```
const char ** wolfSSL_get_system_CA_dirs(
    word32 * num
)
```

This function returns a pointer to an array of strings representing directories wolfSSL will search for system CA certs when wolfSSL_CTX_load_system_CA_certs is called. On systems that don't store certificates in an accessible system directory (such as Apple platforms), this function will always return NULL.

Parameters:

- **num** pointer to a word32 that will be populated with the length of the array of strings.

See:

- wolfSSL_CTX_load_system_CA_certs
- wolfSSL_CTX_load_verify_locations
- wolfSSL_CTX_load_verify_locations_ex

Return:

- Valid pointer on success.
- NULL pointer on failure.

Example

```
WOLFSSL_CTX* ctx;
const char** dirs;
word32 numDirs;

dirs = wolfSSL_get_system_CA_dirs(&numDirs);
for (int i = 0; i < numDirs; ++i) {
    printf("Potential system CA dir: %s\n", dirs[i]);
}
...
```

```
int wolfSSL_CTX_load_system_CA_certs(
    WOLFSSL_CTX * ctx
)
```

On most platforms (including Linux and Windows), this function attempts to load CA certificates into a WOLFSSL_CTX from an OS-dependent CA certificate store. Loaded certificates will be trusted.

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().

See:

- wolfSSL_get_system_CA_dirs
- wolfSSL_CTX_load_verify_locations
- wolfSSL_CTX_load_verify_locations_ex

Return:

- WOLFSSL_SUCCESS on success.
- WOLFSSL_BAD_PATH if no system CA certs were loaded.
- WOLFSSL_FAILURE for other failure types (e.g. Windows cert store wasn't properly closed).

On Apple platforms (excluding macOS), certificates can't be obtained from the system, and therefore cannot be loaded into the wolfSSL certificate manager. For these platforms, this function enables TLS connections bound to the WOLFSSL_CTX to use the native system trust APIs to verify authenticity of the peer certificate chain if the authenticity of the peer cannot first be authenticated against certificates loaded by the user.

The platforms supported and tested are: Linux (Debian, Ubuntu, Gentoo, Fedora, RHEL), Windows 10/11, Android, macOS, and iOS.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx;
...
ret = wolfSSL_CTX_load_system_CA_certs(ctx,);
if (ret != WOLFSSL_SUCCESS) {
    // error loading system CA certs
}
...
```

```
int wolfSSL_CTX_use_certificate_chain_file(
    WOLFSSL_CTX * ctx,
    const char * file
)
```

This function loads a chain of certificates into the SSL context (WOLFSSL_CTX). The file containing the certificate chain is provided by the file argument, and must contain PEM-formatted certificates. This function will process up to MAX_CHAIN_DEPTH (default = 9, defined in internal.h) certificates, plus the subject cert.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new()
- **file** a pointer to the name of the file containing the chain of certificates to be loaded into the wolfSSL SSL context. Certificates must be in PEM format.

See:

- wolfSSL_CTX_use_certificate_file
- wolfSSL_CTX_use_certificate_buffer
- wolfSSL_use_certificate_file

- wolfSSL_use_certificate_buffer

Return:

- SSL_SUCCESS upon success
- SSL_FAILURE If the function call fails, possible causes might include the file is in the wrong format, or the wrong format has been given using the "format" argument, file doesn't exist, can't be read, or is corrupted, an out of memory condition occurs.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx;
...
ret = wolfSSL_CTX_use_certificate_chain_file(ctx, "./cert-chain.pem");
if (ret != SSL_SUCCESS) {
    // error loading cert file
}
...
```

```
int wolfSSL_CTX_der_load_verify_locations(
    WOLFSSL_CTX * ctx,
    const char * file,
    int format
)
```

This function is similar to wolfSSL_CTX_load_verify_locations, but allows the loading of DER-formatted CA files into the SSL context (WOLFSSL_CTX). It may still be used to load PEM-formatted CA files as well. These certificates will be treated as trusted root certificates and used to verify certs received from peers during the SSL handshake. The root certificate file, provided by the file argument, may be a single certificate or a file containing multiple certificates. If multiple CA certs are included in the same file, wolfSSL will load them in the same order they are presented in the file. The format argument specifies the format which the certificates are in either, SSL_FILETYPE_PEM or SSL_FILETYPE_ASN1 (DER). Unlike wolfSSL_CTX_load_verify_locations, this function does not allow the loading of CA certificates from a given directory path. Note that this function is only available when the wolfSSL library was compiled with WOLFSSL_DER_LOAD defined.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new()
- **file** a pointer to the name of the file containing the CA certificates to be loaded into the wolfSSL SSL context, with format as specified by format.
- **format** the encoding type of the certificates specified by file. Possible values include SSL_FILETYPE_PEM and SSL_FILETYPE_ASN1.

See:

- wolfSSL_CTX_load_verify_locations
- wolfSSL_CTX_load_verify_buffer

Return:

- SSL_SUCCESS upon success.
- SSL_FAILURE upon failure.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx;
...
ret = wolfSSL_CTX_der_load_verify_locations(ctx, "./ca-cert.der",
                                            SSL_FILETYPE_ASN1);
if (ret != SSL_SUCCESS) {
    // error loading CA certs
}
...
```

```
void wolfSSL_SetCertCbCtx(
    WOLFSSL * ssl,
    void * ctx
)
```

This function stores user CTX object information for verify callback.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **ctx** a void pointer that is set to WOLFSSL structure's verifyCbCtx member's value.

See:

- wolfSSL_CTX_save_cert_cache
- wolfSSL_CTX_restore_cert_cache
- wolfSSL_CTX_set_verify

Return: none No return.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
(void*)ctx;
...
if(ssl != NULL){
wolfSSL_SetCertCbCtx(ssl, ctx);
} else {
    // Error case, the SSL is not initialized properly.
}
```

```
void wolfSSL_CTX_SetCertCbCtx(
    WOLFSSL_CTX * ctx,
    void * userCtx
)
```

This function stores user CTX object information for verify callback.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure.
- **userCtx** a void pointer that is used to set WOLFSSL_CTX structure's verifyCbCtx member's value.

See:

- wolfSSL_CTX_save_cert_cache
- wolfSSL_CTX_restore_cert_cache
- wolfSSL_CTX_set_verify

Return: none No return.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
void* userCtx = NULL; // Assign some user defined context
...
if(ctx != NULL){
    wolfSSL_SetCertCbCtx(ctx, userCtx);
} else {
    // Error case, the SSL is not initialized properly.
}
```

```
int wolfSSL_CTX_save_cert_cache(
    WOLFSSL_CTX * ctx,
    const char * fname
)
```

This function writes the cert cache from memory to file.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, holding the certificate information.
- **fname** a constant char pointer that points to a file for writing.

See:

- CM_SaveCertCache
- DoMemSaveCertCache

Return:

- SSL_SUCCESS if CM_SaveCertCache exits normally.
- BAD_FUNC_ARG is returned if either of the arguments are NULL.
- SSL_BAD_FILE if the cert cache save file could not be opened.
- BAD_MUTEX_E if the lock mutex failed.
- MEMORY_E the allocation of memory failed.
- FWRITE_ERROR Certificate cache file write failed.

Example

```
WOLFSSL_CTX* ctx = WOLFSSL_CTX_new( protocol def );
const char* fname;
...
if(wolfSSL_CTX_save_cert_cache(ctx, fname)){
    // file was written.
}
```

```
int wolfSSL_CTX_restore_cert_cache(
    WOLFSSL_CTX * ctx,
    const char * fname
)
```

This function persistes certificate cache from a file.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, holding the certificate information.
- **fname** a constant char pointer that points to a file for reading.

See:

- CM_RestoreCertCache
- XFOPEN

Return:

- SSL_SUCCESS returned if the function, CM_RestoreCertCache, executes normally.
- SSL_BAD_FILE returned if XFOPEN returns XBADFILE. The file is corrupted.
- MEMORY_E returned if the allocated memory for the temp buffer fails.
- BAD_FUNC_ARG returned if fname or ctx have a NULL value.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
WOLFSSL* ssl = wolfSSL_new(ctx);
const char* fname = "path to file";
...
if(wolfSSL_CTX_restore_cert_cache(ctx, fname)){
    // check to see if the execution was successful
}
```

```
int wolfSSL_CTX_memsave_cert_cache(
    WOLFSSL_CTX * ctx,
    void * mem,
    int sz,
    int * used
)
```

This function persists the certificate cache to memory.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().
- **mem** a void pointer to the destination (output buffer).
- **sz** the size of the output buffer.
- **used** a pointer to size of the cert cache header.

See:

- DoMemSaveCertCache
- GetCertCacheMemSize
- CM_MemRestoreCertCache
- CM_GetCertCacheMemSize

Return:

- SSL_SUCCESS returned on successful execution of the function. No errors were thrown.
- BAD_MUTEX_E mutex error where the WOLFSSL_CERT_MANAGER member caLock was not 0 (zero).
- BAD_FUNC_ARG returned if ctx, mem, or used is NULL or if sz is less than or equal to 0 (zero).
- BUFFER_E output buffer mem was too small.

Example

```
WOLFSSL_CTX* ctx = WOLFSSL_CTX_new( protocol );
void* mem;
int sz;
int* used;
...
if(wolfSSL_CTX_memsave_cert_cache(ctx, mem, sz, used) != SSL_SUCCESS){
    // The function returned with an error
}
```

```
int wolfSSL_CTX_get_cert_cache_memsize(
    WOLFSSL_CTX * ctx
)
```

Returns the size the certificate cache save buffer needs to be.

Parameters:

- **ctx** a pointer to a wolfSSL_CTX structure, created using wolfSSL_CTX_new().

See: CM_GetCertCacheMemSize

Return:

- int integer value returned representing the memory size upon success.
- BAD_FUNC_ARG is returned if the WOLFSSL_CTX struct is NULL.
- BAD_MUTEX_E - returned if there was a mutex lock error.

Example

```
WOLFSSL_CTX* ctx = WOLFSSL_CTX_new(protocol);
...
int certCacheSize = wolfSSL_CTX_get_cert_cache_memsize(ctx);

if(certCacheSize != BAD_FUNC_ARG || certCacheSize != BAD_MUTEX_E){
// Successfully retrieved the memory size.
}
```

```
char * wolfSSL_X509_NAME_oneline(
    WOLFSSL_X509_NAME * name,
    char * in,
    int sz
)
```

This function copies the name of the x509 into a buffer.

Parameters:

- **name** a pointer to a WOLFSSL_X509 structure.
- **in** a buffer to hold the name copied from the WOLFSSL_X509_NAME structure.
- **sz** the maximum size of the buffer.

See:

- wolfSSL_X509_get_subject_name
- wolfSSL_X509_get_issuer_name
- wolfSSL_X509_get_isCA
- wolfSSL_get_peer_certificate
- wolfSSL_X509_version

Return: A char pointer to the buffer with the WOLFSSL_X509_NAME structures name member's data is returned if the function executed normally.

Example

```
WOLFSSL_X509 x509;
char* name;
...
name = wolfSSL_X509_NAME_oneline(wolfSSL_X509_get_issuer_name(x509), 0, 0);
```

```
if(name <= 0){
    // There's nothing in the buffer.
}
```

```
WOLFSSL_X509_NAME * wolfSSL_X509_get_issuer_name(
    WOLFSSL_X509 * cert
)
```

This function returns the name of the certificate issuer.

Parameters:

- **cert** a pointer to a WOLFSSL_X509 structure.

See:

- wolfSSL_X509_get_subject_name
- wolfSSL_X509_get_isCA
- wolfSSL_get_peer_certificate
- wolfSSL_X509_NAME_oneline

Return:

- point a pointer to the WOLFSSL_X509 struct's issuer member is returned.
- NULL if the cert passed in is NULL.

Example

```
WOLFSSL_X509* x509;
WOLFSSL_X509_NAME issuer;
...
issuer = wolfSSL_X509_NAME_oneline(wolfSSL_X509_get_issuer_name(x509), 0, 0);

if(!issuer){
    // NULL was returned
} else {
    // issuer hods the name of the certificate issuer.
}
```

```
WOLFSSL_X509_NAME * wolfSSL_X509_get_subject_name(
    WOLFSSL_X509 * cert
)
```

This function returns the subject member of the WOLFSSL_X509 structure.

Parameters:

- **cert** a pointer to a WOLFSSL_X509 structure.

See:

- wolfSSL_X509_get_issuer_name
- wolfSSL_X509_get_isCA
- wolfSSL_get_peer_certificate

Return: pointer a pointer to the WOLFSSL_X509_NAME structure. The pointer may be NULL if the WOLFSSL_X509 struct is NULL or if the subject member of the structure is NULL.

Example

```
WOLFSSL_X509* cert;
WOLFSSL_X509_NAME name;
…
name = wolfSSL_X509_get_subject_name(cert);
if(name == NULL){
    // Deal with the NULL cacse
}
```

```
int wolfSSL_X509_get_isCA(
    WOLFSSL_X509 * cert
)
```

Checks the isCa member of the WOLFSSL_X509 structure and returns the value.

Parameters:

- **cert** a pointer to a WOLFSSL_X509 structure.

See:

- wolfSSL_X509_get_issuer_name
- wolfSSL_X509_get_isCA

Return:

- isCA returns the value in the isCA member of the WOLFSSL_X509 structure is returned.
- 0 returned if there is not a valid x509 structure passed in.

Example

```
WOLFSSL* ssl;
...
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
...
if(wolfSSL_X509_get_isCA(ssl)){
    // This is the CA
```

```
}else {
    // Failure case
}
```

```
int wolfSSL_X509_NAME_get_text_by_NID(
    WOLFSSL_X509_NAME * name,
    int nid,
    char * buf,
    int len
)
```

This function gets the text related to the passed in NID value.

Parameters:

- **name** WOLFSSL_X509_NAME to search for text.
- **nid** NID to search for.
- **buf** buffer to hold text when found.
- **len** length of buffer.

See: none

Return: int returns the size of the text buffer.

Example

```
WOLFSSL_X509_NAME* name;
char buffer[100];
int bufferSz;
int ret;
// get WOLFSSL_X509_NAME
ret = wolfSSL_X509_NAME_get_text_by_NID(name, NID_commonName,
buffer, bufferSz);

//check ret value
```

```
int wolfSSL_X509_get_signature_type(
    WOLFSSL_X509 * cert
)
```

This function returns the value stored in the sigOID member of the WOLFSSL_X509 structure.

Parameters:

- **cert** a pointer to a WOLFSSL_X509 structure.

See:

- wolfSSL_X509_get_signature

- wolfSSL_X509_version
- wolfSSL_X509_get_der
- wolfSSL_X509_get_serial_number
- wolfSSL_X509_notBefore
- wolfSSL_X509_notAfter
- wolfSSL_X509_free

Return:

- 0 returned if the WOLFSSL_X509 structure is NULL.
- int an integer value is returned which was retrieved from the x509 object.

Example

```
WOLFSSL_X509 x509 = (WOLFSSL_X509*)XMALLOC(sizeof(WOLFSSL_X509), NULL,
                        DYNAMIC_TYPE_X509);
...
int x509SigType = wolfSSL_X509_get_signature_type(x509);

if(x509SigType != EXPECTED){
// Deal with an unexpected value
}
```

```
int wolfSSL_X509_get_signature(
    WOLFSSL_X509 * x509,
    unsigned char * buf,
    int * bufSz
)
```

Gets the X509 signature and stores it in the buffer.

Parameters:

- **x509** pointer to a WOLFSSL_X509 structure.
- **buf** a char pointer to the buffer.
- **bufSz** an integer pointer to the size of the buffer.

See:

- wolfSSL_X509_get_serial_number
- wolfSSL_X509_get_signature_type
- wolfSSL_X509_get_device_type

Return:

- SSL_SUCCESS returned if the function successfully executes. The signature is loaded into the buffer.
- SSL_FATAL_ERRROR returns if the x509 struct or the bufSz member is NULL. There is also a check for the length member of the sig structure (sig is a member of x509).

Example

```
WOLFSSL_X509* x509 = (WOLFSSL_X509)XMALOC(sizeof(WOLFSSL_X509), NULL,
DYNAMIC_TYPE_X509);
unsigned char* buf; // Initialize
int* bufSz = sizeof(buf)/sizeof(unsigned char);
...
if(wolfSSL_X509_get_signature(x509, buf, bufSz) != SSL_SUCCESS){
    // The function did not execute successfully.
} else{
    // The buffer was written to correctly.
}
```

```
int wolfSSL_X509_STORE_add_cert(
    WOLFSSL_X509_STORE * store,
    WOLFSSL_X509 * x509
)
```

This function adds a certificate to the WOLFSSL_X509_STRE structure.

Parameters:

- **str** certificate store to add the certificate to.
- **x509** certificate to add.

See: wolfSSL_X509_free

Return:

- SSL_SUCCESS If certificate is added successfully.
- SSL_FATAL_ERROR: If certificate is not added successfully.

Example

```
WOLFSSL_X509_STORE* str;
WOLFSSL_X509* x509;
int ret;
ret = wolfSSL_X509_STORE_add_cert(str, x509);
//check ret value
```

```
WOLFSSL_STACK * wolfSSL_X509_STORE_CTX_get_chain(
    WOLFSSL_X509_STORE_CTX * ctx
)
```

This function is a getter function for chain variable in WOLFSSL_X509_STORE_CTX structure. Currently chain is not populated.

Parameters:

- **ctx** certificate store ctx to get parse chain from.

See: wolfSSL_sk_X509_free

Return:

- pointer if successful returns WOLFSSL_STACK (same as STACK_OF(WOLFSSL_X509)) pointer
- Null upon failure

Example

```
WOLFSSL_STACK* sk;
WOLFSSL_X509_STORE_CTX* ctx;
sk = wolfSSL_X509_STORE_CTX_get_chain(ctx);
//check sk for NULL and then use it. sk needs freed after done.
```

```
int wolfSSL_X509_STORE_set_flags(
    WOLFSSL_X509_STORE * store,
    unsigned long flag
)
```

This function takes in a flag to change the behavior of the WOLFSSL_X509_STORE structure passed in. An example of a flag used is WOLFSSL_CRL_CHECK.

Parameters:

- **str** certificate store to set flag in.
- **flag** flag for behavior.

See:

- wolfSSL_X509_STORE_new
- wolfSSL_X509_STORE_free

Return:

- SSL_SUCCESS If no errors were encountered when setting the flag.
- <0 a negative value will be returned upon failure.

Example

```
WOLFSSL_X509_STORE* str;
int ret;
// create and set up str
ret = wolfSSL_X509_STORE_set_flags(str, WOLFSSL_CRL_CHECKALL);
If (ret != SSL_SUCCESS) {
    //check ret value and handle error case
}
```

```
const byte * wolfSSL_X509_notBefore(
    WOLFSSL_X509 * x509
)
```

This function the certificate “not before” validity encoded as a byte array.

Parameters:

- **x509** pointer to a WOLFSSL_X509 structure.

See:

- wolfSSL_X509_get_signature
- wolfSSL_X509_version
- wolfSSL_X509_get_der
- wolfSSL_X509_get_serial_number
- wolfSSL_X509_notAfter
- wolfSSL_X509_free

Return:

- NULL returned if the WOLFSSL_X509 structure is NULL.
- byte is returned that contains the notBeforeData.

Example

```
WOLFSSL_X509* x509 = (WOLFSSL_X509*)XMALLOC(sizeof(WOLFSSL_X509), NULL,
                        DYNAMIC_TYPE_X509);
...
byte* notBeforeData = wolfSSL_X509_notBefore(x509);
```

```
const byte * wolfSSL_X509_notAfter(
    WOLFSSL_X509 * x509
)
```

This function the certificate “not after” validity encoded as a byte array.

Parameters:

- **x509** pointer to a WOLFSSL_X509 structure.

See:

- wolfSSL_X509_get_signature
- wolfSSL_X509_version
- wolfSSL_X509_get_der
- wolfSSL_X509_get_serial_number
- wolfSSL_X509_notBefore
- wolfSSL_X509_free

Return:

- NULL returned if the WOLFSSL_X509 structure is NULL.
- byte is returned that contains the notAfterData.

Example

```
WOLFSSL_X509* x509 = (WOLFSSL_X509*)XMALLOC(sizeof(WOLFSSL_X509), NULL,
                        DYNAMIC_TYPE_X509);
...
byte* notAfterData = wolfSSL_X509_notAfter(x509);
```

```
const char * wolfSSL_get_psk_identity_hint(
    const WOLFSSL *
)
```

This function returns the psk identity hint.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See: wolfSSL_get_psk_identity

Return:

- pointer a const char pointer to the value that was stored in the arrays member of the WOLFSSL structure is returned.
- NULL returned if the WOLFSSL or Arrays structures are NULL.

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
char* idHint;
...
idHint = wolfSSL_get_psk_identity_hint(ssl);
if(idHint){
    // The hint was retrieved
    return idHint;
} else {
    // Hint wasn't successfully retrieved
}
```

```
const char * wolfSSL_get_psk_identity(
    const WOLFSSL *
)
```

The function returns a constant pointer to the client_identity member of the Arrays structure.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_get_psk_identity_hint
- wolfSSL_use_psk_identity_hint

Return:

- string the string value of the client_identity member of the Arrays structure.
- NULL if the WOLFSSL structure is NULL or if the Arrays member of the WOLFSSL structure is NULL.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
const char* pskID;
...
pskID = wolfSSL_get_psk_identity(ssl);

if(pskID == NULL){
    // There is not a value in pskID
}
```

```
int wolfSSL_CTX_use_psk_identity_hint(
    WOLFSSL_CTX * ctx,
    const char * hint
)
```

This function stores the hint argument in the server_hint member of the WOLFSSL_CTX structure.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().
- **hint** a constant char pointer that will be copied to the WOLFSSL_CTX structure.

See: wolfSSL_use_psk_identity_hint

Return: SSL_SUCCESS returned for successful execution of the function.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
const char* hint;
int ret;
…
```

```
ret = wolfSSL_CTX_use_psk_identity_hint(ctx, hint);
if(ret == SSL_SUCCESS){
    // Function was successful.
return ret;
} else {
    // Failure case.
}
```

```
int wolfSSL_use_psk_identity_hint(
    WOLFSSL * ssl,
    const char * hint
)
```

This function stores the hint argument in the server_hint member of the Arrays structure within the WOLFSSL structure.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **hint** a constant character pointer that holds the hint to be saved in memory.

See: wolfSSL_CTX_use_psk_identity_hint

Return:

- SSL_SUCCESS returned if the hint was successfully stored in the WOLFSSL structure.
- SSL_FAILURE returned if the WOLFSSL or Arrays structures are NULL.

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
const char* hint;
...
if(wolfSSL_use_psk_identity_hint(ssl, hint) != SSL_SUCCESS){
    // Handle failure case.
}
```

```
WOLFSSL_X509 * wolfSSL_get_peer_certificate(
    WOLFSSL * ssl
)
```

This function gets the peer's certificate.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_X509_get_issuer_name
- wolfSSL_X509_get_subject_name
- wolfSSL_X509_get_isCA

Return:

- pointer a pointer to the peerCert member of the WOLFSSL_X509 structure if it exists.
- 0 returned if the peer certificate issuer size is not defined.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
...
WOLFSSL_X509* peerCert = wolfSSL_get_peer_certificate(ssl);

if(peerCert){
    // You have a pointer peerCert to the peer certification
}
```

```
WOLFSSL_X509 * wolfSSL_get_chain_X509(
    WOLFSSL_X509_CHAIN * chain,
    int idx
)
```

This function gets the peer's wolfSSL_X509_certificate at index (idx) from the chain of certificates.

Parameters:

- **chain** a pointer to the WOLFSSL_X509_CHAIN used for no dynamic memory SESSION_CACHE.
- **idx** the index of the WOLFSSL_X509 certificate.

See:

- InitDecodedCert
- ParseCertRelative
- CopyDecodedToX509

Return: pointer returns a pointer to a WOLFSSL_X509 structure.

Note that it is the user's responsibility to free the returned memory by calling wolfSSL_FreeX509().

Example

```
WOLFSSL_X509_CHAIN* chain = &session->chain;
int idx = 999; // set idx
...
WOLFSSL_X509_CHAIN ptr;
prt = wolfSSL_get_chain_X509(chain, idx);

if(ptr != NULL){
```

```
    // ptr contains the cert at the index specified
    wolfSSL_FreeX509(ptr);
} else {
    // ptr is NULL
}
```

```
char * wolfSSL_X509_get_subjectCN(
    WOLFSSL_X509 *
)
```

Returns the common name of the subject from the certificate.

Parameters:

- **x509** a pointer to a WOLFSSL_X509 structure containing certificate information.

See:

- wolfSSL_X509_Name_get_entry
- wolfSSL_X509_get_next_altname
- wolfSSL_X509_get_issuer_name
- wolfSSL_X509_get_subject_name

Return:

- NULL returned if the x509 structure is null
- string a string representation of the subject's common name is returned upon success

Example

```
WOLFSSL_X509 x509 = (WOLFSSL_X509*)XMALLOC(sizeof(WOLFSSL_X509), NULL,
                        DYNAMIC_TYPE_X509);
...
int x509Cn = wolfSSL_X509_get_subjectCN(x509);
if(x509Cn == NULL){
    // Deal with NULL case
} else {
    // x509Cn contains the common name
}
```

```
const unsigned char * wolfSSL_X509_get_der(
    WOLFSSL_X509 * x509,
    int * outSz
)
```

This function gets the DER encoded certificate in the WOLFSSL_X509 struct.

Parameters:

- **x509** a pointer to a WOLFSSL_X509 structure containing certificate information.
- **outSz** length of the derBuffer member of the WOLFSSL_X509 struct.

See:

- wolfSSL_X509_version
- wolfSSL_X509_Name_get_entry
- wolfSSL_X509_get_next_altname
- wolfSSL_X509_get_issuer_name
- wolfSSL_X509_get_subject_name

Return:

- buffer This function returns the DerBuffer structure's buffer member, which is of type byte.
- NULL returned if the x509 or outSz parameter is NULL.

Example

```
WOLFSSL_X509 x509 = (WOLFSSL_X509*)XMALLOC(sizeof(WOLFSSL_X509), NULL,
                        DYNAMIC_TYPE_X509);
int* outSz; // initialize
...
byte* x509Der = wolfSSL_X509_get_der(x509, outSz);
if(x509Der == NULL){
    // Failure case one of the parameters was NULL
}
```

```
WOLFSSL_ASN1_TIME * wolfSSL_X509_get_notAfter(
    WOLFSSL_X509 *
)
```

This function checks to see if x509 is NULL and if it's not, it returns the notAfter member of the x509 struct.

Parameters:

- **x509** a pointer to the WOLFSSL_X509 struct.

See: wolfSSL_X509_get_notBefore

Return:

- pointer to struct with ASN1_TIME to the notAfter member of the x509 struct.
- NULL returned if the x509 object is NULL.

Example

```
WOLFSSL_X509* x509 = (WOLFSSL_X509)XMALOC(sizeof(WOLFSSL_X509), NULL,
DYNAMIC_TYPE_X509) ;
...
```

```
const WOLFSSL_ASN1_TIME* notAfter = wolfSSL_X509_get_notAfter(x509);
if(notAfter == NULL){
    // Failure case, the x509 object is null.
}
```

```
int wolfSSL_X509_version(
    WOLFSSL_X509 *
)
```

This function retrieves the version of the X509 certificate.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_X509_get_subject_name
- wolfSSL_X509_get_issuer_name
- wolfSSL_X509_get_isCA
- wolfSSL_get_peer_certificate

Return:

- 0 returned if the x509 structure is NULL.
- version the version stored in the x509 structure will be returned.

Example

```
WOLFSSL_X509* x509;
int version;
...
version = wolfSSL_X509_version(x509);
if(!version){
    // The function returned 0, failure case.
}
```

```
WOLFSSL_X509 * wolfSSL_X509_d2i_fp(
    WOLFSSL_X509 ** x509,
    FILE * file
)
```

If NO_STDIO_FILESYSTEM is defined this function will allocate heap memory, initialize a WOLFSSL_X509 structure and return a pointer to it.

Parameters:

- **x509** a pointer to a WOLFSSL_X509 pointer.

- **file** a defined type that is a pointer to a FILE.

See:

- wolfSSL_X509_d2i
- XFTELL
- XREWIND
- XFSEEK

Return:

- *WOLFSSL_X509 WOLFSSL_X509 structure pointer is returned if the function executes successfully.
- NULL if the call to XFTELL macro returns a negative value.

Example

```
WOLFSSL_X509* x509a = (WOLFSSL_X509*)XMALLOC(sizeof(WOLFSSL_X509), NULL,
DYNAMIC_TYPE_X509);
WOLFSSL_X509** x509 = x509a;
XFILE file;  (mapped to struct fs_file*)
...
WOLFSSL_X509* newX509 = wolfSSL_X509_d2i_fp(x509, file);
if(newX509 == NULL){
    // The function returned NULL
}
```

```
WOLFSSL_X509 * wolfSSL_X509_load_certificate_file(
    const char * fname,
    int format
)
```

The function loads the x509 certificate into memory.

Parameters:

- **fname** the certificate file to be loaded.
- **format** the format of the certificate.

See:

- InitDecodedCert
- PemToDer
- wolfSSL_get_certificate
- AssertNotNull

Return:

- pointer a successful execution returns pointer to a WOLFSSL_X509 structure.

- NULL returned if the certificate was not able to be written.

Example

```
#define cliCert    “certs/client-cert.pem”
…
X509* x509;
…
x509 = wolfSSL_X509_load_certificate_file(cliCert, SSL_FILETYPE_PEM);
AssertNotNull(x509);
```

```
unsigned char * wolfSSL_X509_get_device_type(
    WOLFSSL_X509 * x509,
    unsigned char * in,
    int * inOutSz
)
```

This function copies the device type from the x509 structure to the buffer.

Parameters:

- **x509** pointer to a WOLFSSL_X509 structure, created with WOLFSSL_X509_new().
- **in** a pointer to a byte type that will hold the device type (the buffer).
- **inOutSz** the minimum of either the parameter inOutSz or the deviceTypeSz member of the x509 structure.

See:

- wolfSSL_X509_get_hw_type
- wolfSSL_X509_get_hw_serial_number
- wolfSSL_X509_d2i

Return:

- pointer returns a byte pointer holding the device type from the x509 structure.
- NULL returned if the buffer size is NULL.

Example

```
WOLFSSL_X509* x509 = (WOLFSSL_X509)XMALOC(sizeof(WOLFSSL_X509), NULL,
DYNAMIC_TYPE_X509);
byte* in;
int* inOutSz;
...
byte* deviceType = wolfSSL_X509_get_device_type(x509, in, inOutSz);

if(!deviceType){
    // Failure case, NULL was returned.
}
```

```
unsigned char * wolfSSL_X509_get_hw_type(
    WOLFSSL_X509 * x509,
    unsigned char * in,
    int * inOutSz
)
```

The function copies the hwType member of the WOLFSSL_X509 structure to the buffer.

Parameters:

- **x509** a pointer to a WOLFSSL_X509 structure containing certificate information.
- **in** pointer to type byte that represents the buffer.
- **inOutSz** pointer to type int that represents the size of the buffer.

See:

- wolfSSL_X509_get_hw_serial_number
- wolfSSL_X509_get_device_type

Return:

- byte The function returns a byte type of the data previously held in the hwType member of the WOLFSSL_X509 structure.
- NULL returned if inOutSz is NULL.

Example

```
WOLFSSL_X509* x509;  // X509 certificate
byte* in;  // initialize the buffer
int* inOutSz;  // holds the size of the buffer
...
byte* hwType = wolfSSL_X509_get_hw_type(x509, in, inOutSz);

if(hwType == NULL){
    // Failure case function returned NULL.
}
```

```
unsigned char * wolfSSL_X509_get_hw_serial_number(
    WOLFSSL_X509 * x509,
    unsigned char * in,
    int * inOutSz
)
```

This function returns the hwSerialNum member of the x509 object.

Parameters:

- **x509** pointer to a WOLFSSL_X509 structure containing certificate information.
- **in** a pointer to the buffer that will be copied to.
- **inOutSz** a pointer to the size of the buffer.

See:

- wolfSSL_X509_get_subject_name
- wolfSSL_X509_get_issuer_name
- wolfSSL_X509_get_isCA
- wolfSSL_get_peer_certificate
- wolfSSL_X509_version

Return: pointer the function returns a byte pointer to the in buffer that will contain the serial number loaded from the x509 object.

Example

```
char* serial;
byte* in;
int* inOutSz;
WOLFSSL_X509 x509;
...
serial = wolfSSL_X509_get_hw_serial_number(x509, in, inOutSz);

if(serial == NULL || serial <= 0){
    // Failure case
}
```

```
int wolfSSL_SetTmpDH(
    WOLFSSL * ssl,
    const unsigned char * p,
    int pSz,
    const unsigned char * g,
    int gSz
)
```

Server Diffie-Hellman Ephemeral parameters setting. This function sets up the group parameters to be used if the server negotiates a cipher suite that uses DHE.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **p** Diffie-Hellman prime number parameter.
- **pSz** size of p.
- **g** Diffie-Hellman “generator” parameter.
- **gSz** size of g.

See: SSL_accept

Return:

- SSL_SUCCESS upon success.
- MEMORY_ERROR will be returned if a memory error was encountered.
- SIDE_ERROR will be returned if this function is called on an SSL client instead of an SSL server.

Example

```
WOLFSSL* ssl;
static unsigned char p[] = {...};
static unsigned char g[] = {...};
...
wolfSSL_SetTmpDH(ssl, p, sizeof(p), g, sizeof(g));
```

```
int wolfSSL_SetTmpDH_buffer(
    WOLFSSL * ssl,
    const unsigned char * b,
    long sz,
    int format
)
```

The function calls the wolfSSL_SetTMpDH_buffer_wrapper, which is a wrapper for Diffie-Hellman parameters.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **buf** allocated buffer passed in from wolfSSL_SetTMpDH_file_wrapper.
- **sz** a long int that holds the size of the file (fname within wolfSSL_SetTmpDH_file_wrapper).
- **format** an integer type passed through from wolfSSL_SetTmpDH_file_wrapper() that is a representation of the certificate format.

See:

- wolfSSL_SetTmpDH_buffer_wrapper
- wc_DhParamsLoad
- wolfSSL_SetTmpDH
- PemToDer
- wolfSSL_CTX_SetTmpDH
- wolfSSL_CTX_SetTmpDH_file

Return:

- SSL_SUCCESS on successful execution.
- SSL_BAD_FILETYPE if the file type is not PEM and is not ASN.1. It will also be returned if the wc_DhParamsLoad does not return normally.
- SSL_NO_PEM_HEADER returns from PemToDer if there is not a PEM header.
- SSL_BAD_FILE returned if there is a file error in PemToDer.
- SSL_FATAL_ERROR returned from PemToDer if there was a copy error.
- MEMORY_E - if there was a memory allocation error.
- BAD_FUNC_ARG returned if the WOLFSSL struct is NULL or if there was otherwise a NULL argument passed to a subroutine.
- DH_KEY_SIZE_E is returned if their is a key size error in wolfSSL_SetTmpDH().
- SIDE_ERROR returned if it is not the server side in wolfSSL_SetTmpDH.

Example

```
Static int wolfSSL_SetTmpDH_file_wrapper(WOLFSSL_CTX* ctx, WOLFSSL* ssl,
Const char* fname, int format);
long sz = 0;
byte* myBuffer = staticBuffer[FILE_BUFFER_SIZE];
…
if(ssl)
ret = wolfSSL_SetTmpDH_buffer(ssl, myBuffer, sz, format);
```

```
int wolfSSL_SetTmpDH_file(
    WOLFSSL * ssl,
    const char * f,
    int format
)
```

This function calls wolfSSL_SetTmpDH_file_wrapper to set server Diffie-Hellman parameters.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **fname** a constant char pointer holding the certificate.
- **format** an integer type that holds the format of the certification.

See:

- wolfSSL_CTX_SetTmpDH_file
- wolfSSL_SetTmpDH_file_wrapper
- wolfSSL_SetTmpDH_buffer
- wolfSSL_CTX_SetTmpDH_buffer
- wolfSSL_SetTmpDH_buffer_wrapper
- wolfSSL_SetTmpDH
- wolfSSL_CTX_SetTmpDH

Return:

- SSL_SUCCESS returned on successful completion of this function and its subroutines.
- MEMORY_E returned if a memory allocation failed in this function or a subroutine.
- SIDE_ERROR if the side member of the Options structure found in the WOLFSSL struct is not the server side.
- SSL_BAD_FILETYPE returns if the certificate fails a set of checks.
- DH_KEY_SIZE_E returned if the DH parameter's key size is less than the value of the minDhKeySz member in the WOLFSSL struct.
- DH_KEY_SIZE_E returned if the DH parameter's key size is greater than the value of the maxDhKeySz member in the WOLFSSL struct.
- BAD_FUNC_ARG returns if an argument value is NULL that is not permitted such as, the WOLFSSL structure.

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
const char* dhParam;
…
AssertIntNE(SSL_SUCCESS,
wolfSSL_SetTmpDH_file(ssl, dhParam, SSL_FILETYPE_PEM));
```

```
int wolfSSL_CTX_SetTmpDH(
    WOLFSSL_CTX * ctx,
    const unsigned char * p,
    int pSz,
    const unsigned char * g,
    int gSz
)
```

Sets the parameters for the server CTX Diffie-Hellman.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().
- **p** a constant unsigned char pointer loaded into the buffer member of the serverDH_P struct.
- **pSz** an int type representing the size of p, initialized to MAX_DH_SIZE.
- **g** a constant unsigned char pointer loaded into the buffer member of the serverDH_G struct.
- **gSz** an int type representing the size of g, initialized to MAX_DH_SIZE.

See:

- wolfSSL_SetTmpDH
- wc_DhParamsLoad

Return:

- SSL_SUCCESS returned if the function and all subroutines return without error.
- BAD_FUNC_ARG returned if the CTX, p or g parameters are NULL.
- DH_KEY_SIZE_E returned if the DH parameter's key size is less than the value of the minDhKeySz member of the WOLFSSL_CTX struct.
- DH_KEY_SIZE_E returned if the DH parameter's key size is greater than the value of the maxDhKeySz member of the WOLFSSL_CTX struct.
- MEMORY_E returned if the allocation of memory failed in this function or a subroutine.

Exmaple

```
WOLFSSL_CTX* ctx =  WOLFSSL_CTX_new( protocol );
byte* p;
byte* g;
word32 pSz = (word32)sizeof(p)/sizeof(byte);
word32 gSz = (word32)sizeof(g)/sizeof(byte);
…
int ret =  wolfSSL_CTX_SetTmpDH(ctx, p, pSz, g, gSz);

if(ret != SSL_SUCCESS){
```

```
    // Failure case
}



int wolfSSL_CTX_SetTmpDH_buffer(
    WOLFSSL_CTX * ctx,
    const unsigned char * b,
    long sz,
    int format
)
```

A wrapper function that calls wolfSSL_SetTmpDH_buffer_wrapper.

Parameters:

- **ctx** a pointer to a WOLFSSL structure, created using wolfSSL_CTX_new().
- **buf** a pointer to a constant unsigned char type that is allocated as the buffer and passed through to wolfSSL_SetTmpDH_buffer_wrapper.
- **sz** a long integer type that is derived from the fname parameter in wolfSSL_SetTmpDH_file_wrapper().
- **format** an integer type passed through from wolfSSL_SetTmpDH_file_wrapper().

See:

- wolfSSL_SetTmpDH_buffer_wrapper
- wolfSSL_SetTMpDH_buffer
- wolfSSL_SetTmpDH_file_wrapper
- wolfSSL_CTX_SetTmpDH_file

Return:

- 0 returned for a successful execution.
- BAD_FUNC_ARG returned if the ctx or buf parameters are NULL.
- MEMORY_E if there is a memory allocation error.
- SSL_BAD_FILETYPE returned if format is not correct.

Example

```
static int wolfSSL_SetTmpDH_file_wrapper(WOLFSSL_CTX* ctx, WOLFSSL* ssl,
Const char* fname, int format);
#ifdef WOLFSSL_SMALL_STACK
byte staticBuffer[1]; // force heap usage
#else
byte* staticBuffer;
long sz = 0;
…
if(ssl){
    ret = wolfSSL_SetTmpDH_buffer(ssl, myBuffer, sz, format);
} else {
ret = wolfSSL_CTX_SetTmpDH_buffer(ctx, myBuffer, sz, format);
}
```

```
int wolfSSL_CTX_SetTmpDH_file(
    WOLFSSL_CTX * ctx,
    const char * f,
    int format
)
```

The function calls wolfSSL_SetTmpDH_file_wrapper to set the server Diffie-Hellman parameters.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().
- **fname** a constant character pointer to a certificate file.
- **format** an integer type passed through from wolfSSL_SetTmpDH_file_wrapper() that is a representation of the certificate format.

See:

- wolfSSL_SetTmpDH_buffer_wrapper
- wolfSSL_SetTmpDH
- wolfSSL_CTX_SetTmpDH
- wolfSSL_SetTmpDH_buffer
- wolfSSL_CTX_SetTmpDH_buffer
- wolfSSL_SetTmpDH_file_wrapper
- AllocDer
- PemToDer

Return:

- SSL_SUCCESS returned if the wolfSSL_SetTmpDH_file_wrapper or any of its subroutines return successfully.
- MEMORY_E returned if an allocation of dynamic memory fails in a subroutine.
- BAD_FUNC_ARG returned if the ctx or fname parameters are NULL or if a subroutine is passed a NULL argument.
- SSL_BAD_FILE returned if the certificate file is unable to open or if the a set of checks on the file fail from wolfSSL_SetTmpDH_file_wrapper.
- SSL_BAD_FILETYPE returned if the format is not PEM or ASN.1 from wolfSSL_SetTmpDH_buffer_wrapper().
- DH_KEY_SIZE_E returned if the DH parameter's key size is less than the value of the minDhKeySz member of the WOLFSSL_CTX struct.
- DH_KEY_SIZE_E returned if the DH parameter's key size is greater than the value of the maxDhKeySz member of the WOLFSSL_CTX struct.
- SIDE_ERROR returned in wolfSSL_SetTmpDH() if the side is not the server end.
- SSL_NO_PEM_HEADER returned from PemToDer if there is no PEM header.
- SSL_FATAL_ERROR returned from PemToDer if there is a memory copy failure.

Example

```
#define dhParam     "certs/dh2048.pem"
#DEFINE aSSERTiNTne(x, y)     AssertInt(x, y, !=, ==)
WOLFSSL_CTX* ctx;
…
AssertNotNull(ctx = wolfSSL_CTX_new(wolfSSLv23_client_method()))
…
```

```
AssertIntNE(SSL_SUCCESS, wolfSSL_CTX_SetTmpDH_file(NULL, dhParam,
SSL_FILETYPE_PEM));
```

```
int wolfSSL_CTX_SetMinDhKey_Sz(
    WOLFSSL_CTX * ctx,
    word16
)
```

This function sets the minimum size (in bits) of the Diffie Hellman key size by accessing the minDhKeySz member in the WOLFSSL_CTX structure.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **keySz_bits** a word16 type used to set the minimum DH key size in bits. The WOLFSSL_CTX struct holds this information in the minDhKeySz member.

See:

- wolfSSL_SetMinDhKey_Sz
- wolfSSL_CTX_SetMaxDhKey_Sz
- wolfSSL_SetMaxDhKey_Sz
- wolfSSL_GetDhKey_Sz
- wolfSSL_CTX_SetTMpDH_file

Return:

- SSL_SUCCESS returned if the function completes successfully.
- BAD_FUNC_ARG returned if the WOLFSSL_CTX struct is NULL or if the keySz_bits is greater than 16,000 or not divisible by 8.

Example

```
public static int CTX_SetMinDhKey_Sz(IntPtr ctx, short minDhKey){
…
return wolfSSL_CTX_SetMinDhKey_Sz(local_ctx, minDhKeyBits);
```

```
int wolfSSL_SetMinDhKey_Sz(
    WOLFSSL * ssl,
    word16 keySz_bits
)
```

Sets the minimum size (in bits) for a Diffie-Hellman key in the WOLFSSL structure.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

- **keySz_bits** a word16 type used to set the minimum DH key size in bits. The WOLFSSL_CTX struct holds this information in the minDhKeySz member.

See:

- wolfSSL_CTX_SetMinDhKey_Sz
- wolfSSL_GetDhKey_Sz

Return:

- SSL_SUCCESS the minimum size was successfully set.
- BAD_FUNC_ARG the WOLFSSL structure was NULL or if the keySz_bits is greater than 16,000 or not divisible by 8.

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
word16 keySz_bits;
...
if(wolfSSL_SetMinDhKey_Sz(ssl, keySz_bits) != SSL_SUCCESS){
    // Failed to set.
}
```

```
int wolfSSL_CTX_SetMaxDhKey_Sz(
    WOLFSSL_CTX * ctx,
    word16 keySz_bits
)
```

This function sets the maximum size (in bits) of the Diffie Hellman key size by accessing the maxDhKeySz member in the WOLFSSL_CTX structure.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **keySz_bits** a word16 type used to set the maximum DH key size in bits. The WOLFSSL_CTX struct holds this information in the maxDhKeySz member.

See:

- wolfSSL_SetMinDhKey_Sz
- wolfSSL_CTX_SetMinDhKey_Sz
- wolfSSL_SetMaxDhKey_Sz
- wolfSSL_GetDhKey_Sz
- wolfSSL_CTX_SetTMpDH_file

Return:

- SSL_SUCCESS returned if the function completes successfully.

- BAD_FUNC_ARG returned if the WOLFSSL_CTX struct is NULL or if the keySz_bits is greater than 16,000 or not divisible by 8.

Example

```
public static int CTX_SetMaxDhKey_Sz(IntPtr ctx, short maxDhKey){
…
return wolfSSL_CTX_SetMaxDhKey_Sz(local_ctx, keySz_bits);
```

```
int wolfSSL_SetMaxDhKey_Sz(
    WOLFSSL * ssl,
    word16 keySz_bits
)
```

Sets the maximum size (in bits) for a Diffie-Hellman key in the WOLFSSL structure.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **keySz** a word16 type representing the bit size of the maximum DH key.

See:

- wolfSSL_CTX_SetMaxDhKey_Sz
- wolfSSL_GetDhKey_Sz

Return:

- SSL_SUCCESS the maximum size was successfully set.
- BAD_FUNC_ARG the WOLFSSL structure was NULL or the keySz parameter was greater than the allowable size or not divisible by 8.

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
word16 keySz;
...
if(wolfSSL_SetMaxDhKey(ssl, keySz) != SSL_SUCCESS){
    // Failed to set.
}
```

```
int wolfSSL_GetDhKey_Sz(
    WOLFSSL *
)
```

Returns the value of dhKeySz (in bits) that is a member of the options structure. This value represents the Diffie-Hellman key size in bytes.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_SetMinDhKey_sz
- wolfSSL_CTX_SetMinDhKey_Sz
- wolfSSL_CTX_SetTmpDH
- wolfSSL_SetTmpDH
- wolfSSL_CTX_SetTmpDH_file

Return:

- dhKeySz returns the value held in ssl->options.dhKeySz which is an integer value representing a size in bits.
- BAD_FUNC_ARG returns if the WOLFSSL struct is NULL.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
WOLFSSL* ssl = wolfSSL_new(ctx);
int dhKeySz;
...
dhKeySz = wolfSSL_GetDhKey_Sz(ssl);

if(dhKeySz == BAD_FUNC_ARG || dhKeySz <= 0){
    // Failure case
} else {
    // dhKeySz holds the size of the key.
}
```

```
int wolfSSL_CTX_SetMinRsaKey_Sz(
    WOLFSSL_CTX * ctx,
    short keySz
)
```

Sets the minimum RSA key size in both the WOLFSSL_CTX structure and the WOLFSSL_CERT_MANAGER structure.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().
- **keySz** a short integer type stored in minRsaKeySz in the ctx structure and the cm structure converted to bytes.

See: wolfSSL_SetMinRsaKey_Sz

Return:

- SSL_SUCCESS returned on successful execution of the function.
- BAD_FUNC_ARG returned if the ctx structure is NULL or the keySz is less than zero or not divisible by 8.

Example

```
WOLFSSL_CTX* ctx = SSL_CTX_new(method);
(void)minDhKeyBits;
ourCert = myoptarg;
…
minDhKeyBits = atoi(myoptarg);
…
if(wolfSSL_CTX_SetMinRsaKey_Sz(ctx, minRsaKeyBits) != SSL_SUCCESS){
…
```

```
int wolfSSL_SetMinRsaKey_Sz(
    WOLFSSL * ssl,
    short keySz
)
```

Sets the minimum allowable key size in bits for RSA located in the WOLFSSL structure.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **keySz** a short integer value representing the the minimum key in bits.

See: wolfSSL_CTX_SetMinRsaKey_Sz

Return:

- SSL_SUCCESS the minimum was set successfully.
- BAD_FUNC_ARG returned if the ssl structure is NULL or if the ksySz is less than zero or not divisible by 8.

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
short keySz;
…

int isSet =  wolfSSL_SetMinRsaKey_Sz(ssl, keySz);
if(isSet != SSL_SUCCESS){
    Failed to set.
}
```

```
int wolfSSL_CTX_SetMinEccKey_Sz(
    WOLFSSL_CTX * ssl,
    short keySz
)
```

Sets the minimum size in bits for the ECC key in the WOLF_CTX structure and the WOLFSSL_CERT_MANAGER structure.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().
- **keySz** a short integer type that represents the minimum ECC key size in bits.

See: wolfSSL_SetMinEccKey_Sz

Return:

- SSL_SUCCESS returned for a successful execution and the minEccKeySz member is set.
- BAD_FUNC_ARG returned if the WOLFSSL_CTX struct is NULL or if the keySz is negative or not divisible by 8.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
short keySz; // minimum key size
…
if(wolfSSL_CTX_SetMinEccKey(ctx, keySz) != SSL_SUCCESS){
    // Failed to set min key size
}
```

```
int wolfSSL_SetMinEccKey_Sz(
    WOLFSSL * ssl,
    short keySz
)
```

Sets the value of the minEccKeySz member of the options structure. The options struct is a member of the WOLFSSL structure and is accessed through the ssl parameter.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **keySz** value used to set the minimum ECC key size. Sets value in the options structure.

See:

- wolfSSL_CTX_SetMinEccKey_Sz
- wolfSSL_CTX_SetMinRsaKey_Sz
- wolfSSL_SetMinRsaKey_Sz

Return:

- SSL_SUCCESS if the function successfully set the minEccKeySz member of the options structure.
- BAD_FUNC_ARG if the WOLFSSL_CTX structure is NULL or if the key size (keySz) is less than 0 (zero) or not divisible by 8.

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx); // New session
short keySz = 999; // should be set to min key size allowable
...
if(wolfSSL_SetMinEccKey_Sz(ssl, keySz) != SSL_SUCCESS){
    // Failure case.
}
```

```
int wolfSSL_make_eap_keys(
    WOLFSSL * ssl,
    void * key,
    unsigned int len,
    const char * label
)
```

This function is used by EAP_TLS and EAP-TTLS to derive keying material from the master secret.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **msk** a void pointer variable that will hold the result of the p_hash function.
- **len** an unsigned integer that represents the length of the msk variable.
- **label** a constant char pointer that is copied from in wc_PRF().

See:

- wc_PRF
- wc_HmacFinal
- wc_HmacUpdate

Return:

- BUFFER_E returned if the actual size of the buffer exceeds the maximum size allowable.
- MEMORY_E returned if there is an error with memory allocation.

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);;
void* msk;
unsigned int len;
const char* label;
…
return wolfSSL_make_eap_keys(ssl, msk, len, label);
```

```
int wolfSSL_CTX_load_verify_buffer(
    WOLFSSL_CTX * ctx,
    const unsigned char * in,
    long sz,
    int format
)
```

This function loads a CA certificate buffer into the WOLFSSL Context. It behaves like the non-buffered version, only differing in its ability to be called with a buffer as input instead of a file. The buffer is provided by the in argument of size sz. format specifies the format type of the buffer; SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM. More than one CA certificate may be loaded per buffer as long as the format is in PEM. Please see the examples for proper usage.

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().
- **in** pointer to the CA certificate buffer.
- **sz** size of the input CA certificate buffer, in.
- **format** format of the buffer certificate, either SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM.

See:

- wolfSSL_CTX_load_verify_locations
- wolfSSL_CTX_use_certificate_buffer
- wolfSSL_CTX_use_PrivateKey_buffer
- wolfSSL_CTX_use_certificate_chain_buffer
- wolfSSL_use_certificate_buffer
- wolfSSL_use_PrivateKey_buffer
- wolfSSL_use_certificate_chain_buffer

Return:

- SSL_SUCCESS upon success
- SSL_BAD_FILETYPE will be returned if the file is the wrong format.
- SSL_BAD_FILE will be returned if the file doesn't exist, can't be read, or is corrupted.
- MEMORY_E will be returned if an out of memory condition occurs.
- ASN_INPUT_E will be returned if Base16 decoding fails on the file.
- BUFFER_E will be returned if a chain buffer is bigger than the receiving buffer.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx;
byte certBuff[...];
long sz = sizeof(certBuff);
...

ret = wolfSSL_CTX_load_verify_buffer(ctx, certBuff, sz, SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
    // error loading CA certs from buffer
}
...
```

```
int wolfSSL_CTX_load_verify_buffer_ex(
    WOLFSSL_CTX * ctx,
    const unsigned char * in,
    long sz,
    int format,
    int userChain,
    word32 flags
)
```

This function loads a CA certificate buffer into the WOLFSSL Context. It behaves like the non-buffered version, only differing in its ability to be called with a buffer as input instead of a file. The buffer is provided by the in argument of size sz. format specifies the format type of the buffer; SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM. More than one CA certificate may be loaded per buffer as long as the format is in PEM. The _ex version was added in PR 2413 and supports additional arguments for userChain and flags.

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().
- **in** pointer to the CA certificate buffer.
- **sz** size of the input CA certificate buffer, in.
- **format** format of the buffer certificate, either SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM.
- **userChain** If using format WOLFSSL_FILETYPE_ASN1 this set to non-zero indicates a chain of DER's is being presented.
- **flags** See ssl.h around WOLFSSL_LOAD_VERIFY_DEFAULT_FLAGS.

See:

- wolfSSL_CTX_load_verify_buffer
- wolfSSL_CTX_load_verify_locations
- wolfSSL_CTX_use_certificate_buffer
- wolfSSL_CTX_use_PrivateKey_buffer
- wolfSSL_CTX_use_certificate_chain_buffer
- wolfSSL_use_certificate_buffer
- wolfSSL_use_PrivateKey_buffer
- wolfSSL_use_certificate_chain_buffer

Return:

- SSL_SUCCESS upon success
- SSL_BAD_FILETYPE will be returned if the file is the wrong format.
- SSL_BAD_FILE will be returned if the file doesn't exist, can't be read, or is corrupted.
- MEMORY_E will be returned if an out of memory condition occurs.
- ASN_INPUT_E will be returned if Base16 decoding fails on the file.
- BUFFER_E will be returned if a chain buffer is bigger than the receiving buffer.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx;
```

```
byte certBuff[...];
long sz = sizeof(certBuff);
...

// Example for force loading an expired certificate
ret = wolfSSL_CTX_load_verify_buffer_ex(ctx, certBuff, sz, SSL_FILETYPE_PEM,
    0, (WOLFSSL_LOAD_FLAG_DATE_ERR_OKAY));
if (ret != SSL_SUCCESS) {
    // error loading CA certs from buffer
}
...
```

```
int wolfSSL_CTX_load_verify_chain_buffer_format(
    WOLFSSL_CTX * ctx,
    const unsigned char * in,
    long sz,
    int format
)
```

This function loads a CA certificate chain buffer into the WOLFSSL Context. It behaves like the non-buffered version, only differing in its ability to be called with a buffer as input instead of a file. The buffer is provided by the in argument of size sz. format specifies the format type of the buffer; SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM. More than one CA certificate may be loaded per buffer as long as the format is in PEM. Please see the examples for proper usage.

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().
- **in** pointer to the CA certificate buffer.
- **sz** size of the input CA certificate buffer, in.
- **format** format of the buffer certificate, either SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM.

See:

- wolfSSL_CTX_load_verify_locations
- wolfSSL_CTX_use_certificate_buffer
- wolfSSL_CTX_use_PrivateKey_buffer
- wolfSSL_CTX_use_certificate_chain_buffer
- wolfSSL_use_certificate_buffer
- wolfSSL_use_PrivateKey_buffer
- wolfSSL_use_certificate_chain_buffer

Return:

- SSL_SUCCESS upon success
- SSL_BAD_FILETYPE will be returned if the file is the wrong format.
- SSL_BAD_FILE will be returned if the file doesn't exist, can't be read, or is corrupted.
- MEMORY_E will be returned if an out of memory condition occurs.
- ASN_INPUT_E will be returned if Base16 decoding fails on the file.
- BUFFER_E will be returned if a chain buffer is bigger than the receiving buffer.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx;
byte certBuff[...];
long sz = sizeof(certBuff);
...

ret = wolfSSL_CTX_load_verify_chain_buffer_format(ctx,
                      certBuff, sz, WOLFSSL_FILETYPE_ASN1);
if (ret != SSL_SUCCESS) {
    // error loading CA certs from buffer
}
...
```

```
int wolfSSL_CTX_use_certificate_buffer(
    WOLFSSL_CTX * ctx,
    const unsigned char * in,
    long sz,
    int format
)
```

This function loads a certificate buffer into the WOLFSSL Context. It behaves like the non-buffered version, only differing in its ability to be called with a buffer as input instead of a file. The buffer is provided by the in argument of size sz. format specifies the format type of the buffer; SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM. Please see the examples for proper usage.

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().
- **in** the input buffer containing the certificate to be loaded.
- **sz** the size of the input buffer.
- **format** the format of the certificate located in the input buffer (in). Possible values are SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM.

See:

- wolfSSL_CTX_load_verify_buffer
- wolfSSL_CTX_use_PrivateKey_buffer
- wolfSSL_CTX_use_certificate_chain_buffer
- wolfSSL_use_certificate_buffer
- wolfSSL_use_PrivateKey_buffer
- wolfSSL_use_certificate_chain_buffer

Return:

- SSL_SUCCESS upon success
- SSL_BAD_FILETYPE will be returned if the file is the wrong format.
- SSL_BAD_FILE will be returned if the file doesn't exist, can't be read, or is corrupted.
- MEMORY_E will be returned if an out of memory condition occurs.

- ASN_INPUT_E will be returned if Base16 decoding fails on the file.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx;
byte certBuff[...];
long sz = sizeof(certBuff);
...
ret = wolfSSL_CTX_use_certificate_buffer(ctx, certBuff, sz, SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
    // error loading certificate from buffer
}
...
```

```
int wolfSSL_CTX_use_PrivateKey_buffer(
    WOLFSSL_CTX * ctx,
    const unsigned char * in,
    long sz,
    int format
)
```

This function loads a private key buffer into the SSL Context. It behaves like the non-buffered version, only differing in its ability to be called with a buffer as input instead of a file. The buffer is provided by the in argument of size sz. format specifies the format type of the buffer; SSL_FILETYPE_ASN1or SSL_FILETYPE_PEM. Please see the examples for proper usage.

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().
- **in** the input buffer containing the private key to be loaded.
- **sz** the size of the input buffer.
- **format** the format of the private key located in the input buffer (in). Possible values are SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM.

See:

- wolfSSL_CTX_load_verify_buffer
- wolfSSL_CTX_use_certificate_buffer
- wolfSSL_CTX_use_certificate_chain_buffer
- wolfSSL_use_certificate_buffer
- wolfSSL_use_PrivateKey_buffer
- wolfSSL_use_certificate_chain_buffer

Return:

- SSL_SUCCESS upon success
- SSL_BAD_FILETYPE will be returned if the file is the wrong format.
- SSL_BAD_FILE will be returned if the file doesn't exist, can't be read, or is corrupted.
- MEMORY_E will be returned if an out of memory condition occurs.

- ASN_INPUT_E will be returned if Base16 decoding fails on the file.
- NO_PASSWORD will be returned if the key file is encrypted but no password is provided.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx;
byte keyBuff[...];
long sz = sizeof(certBuff);
...
ret = wolfSSL_CTX_use_PrivateKey_buffer(ctx, keyBuff, sz, SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
    // error loading private key from buffer
}
...
```

```
int wolfSSL_CTX_use_certificate_chain_buffer(
    WOLFSSL_CTX * ctx,
    const unsigned char * in,
    long sz
)
```

This function loads a certificate chain buffer into the WOLFSSL Context. It behaves like the non-buffered version, only differing in its ability to be called with a buffer as input instead of a file. The buffer is provided by the in argument of size sz. The buffer must be in PEM format and start with the subject's certificate, ending with the root certificate. Please see the examples for proper usage.

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().
- **in** the input buffer containing the PEM-formatted certificate chain to be loaded.
- **sz** the size of the input buffer.

See:

- wolfSSL_CTX_load_verify_buffer
- wolfSSL_CTX_use_certificate_buffer
- wolfSSL_CTX_use_PrivateKey_buffer
- wolfSSL_use_certificate_buffer
- wolfSSL_use_PrivateKey_buffer
- wolfSSL_use_certificate_chain_buffer

Return:

- SSL_SUCCESS upon success
- SSL_BAD_FILETYPE will be returned if the file is the wrong format.
- SSL_BAD_FILE will be returned if the file doesn't exist, can't be read, or is corrupted.
- MEMORY_E will be returned if an out of memory condition occurs.
- ASN_INPUT_E will be returned if Base16 decoding fails on the file.
- BUFFER_E will be returned if a chain buffer is bigger than the receiving buffer.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx;
byte certChainBuff[...];
long sz = sizeof(certBuff);
...
ret = wolfSSL_CTX_use_certificate_chain_buffer(ctx, certChainBuff, sz);
if (ret != SSL_SUCCESS) {
    // error loading certificate chain from buffer
}
...
```

```
int wolfSSL_use_certificate_buffer(
    WOLFSSL * ssl,
    const unsigned char * in,
    long sz,
    int format
)
```

This function loads a certificate buffer into the WOLFSSL object. It behaves like the non-buffered version, only differing in its ability to be called with a buffer as input instead of a file. The buffer is provided by the in argument of size sz. format specifies the format type of the buffer; SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM. Please see the examples for proper usage.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().
- **in** buffer containing certificate to load.
- **sz** size of the certificate located in buffer.
- **format** format of the certificate to be loaded. Possible values are SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM.

See:

- wolfSSL_CTX_load_verify_buffer
- wolfSSL_CTX_use_certificate_buffer
- wolfSSL_CTX_use_PrivateKey_buffer
- wolfSSL_CTX_use_certificate_chain_buffer
- wolfSSL_use_PrivateKey_buffer
- wolfSSL_use_certificate_chain_buffer

Return:

- SSL_SUCCESS upon success.
- SSL_BAD_FILETYPE will be returned if the file is the wrong format.
- SSL_BAD_FILE will be returned if the file doesn't exist, can't be read, or is corrupted.
- MEMORY_E will be returned if an out of memory condition occurs.
- ASN_INPUT_E will be returned if Base16 decoding fails on the file.

Example

```
int ret;
byte certBuff[...];
WOLFSSL* ssl = 0;
long buffSz = sizeof(certBuff);
...

ret = wolfSSL_use_certificate_buffer(ssl, certBuff, buffSz, SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
    // failed to load certificate from buffer
}
```

```
int wolfSSL_use_PrivateKey_buffer(
    WOLFSSL * ssl,
    const unsigned char * in,
    long sz,
    int format
)
```

This function loads a private key buffer into the WOLFSSL object. It behaves like the non-buffered version, only differing in its ability to be called with a buffer as input instead of a file. The buffer is provided by the in argument of size sz. format specifies the format type of the buffer; SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM. Please see the examples for proper usage.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().
- **in** buffer containing private key to load.
- **sz** size of the private key located in buffer.
- **format** format of the private key to be loaded. Possible values are SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM.

See:

- wolfSSL_use_PrivateKey
- wolfSSL_CTX_load_verify_buffer
- wolfSSL_CTX_use_certificate_buffer
- wolfSSL_CTX_use_PrivateKey_buffer
- wolfSSL_CTX_use_certificate_chain_buffer
- wolfSSL_use_certificate_buffer
- wolfSSL_use_certificate_chain_buffer

Return:

- SSL_SUCCESS upon success.
- SSL_BAD_FILETYPE will be returned if the file is the wrong format.
- SSL_BAD_FILE will be returned if the file doesn't exist, can't be read, or is corrupted.
- MEMORY_E will be returned if an out of memory condition occurs.
- ASN_INPUT_E will be returned if Base16 decoding fails on the file.

- NO_PASSWORD will be returned if the key file is encrypted but no password is provided.

Example

```
int ret;
byte keyBuff[...];
WOLFSSL* ssl = 0;
long buffSz = sizeof(certBuff);
...
ret = wolfSSL_use_PrivateKey_buffer(ssl, keyBuff, buffSz, SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
    // failed to load private key from buffer
}
```

```
int wolfSSL_use_certificate_chain_buffer(
    WOLFSSL * ssl,
    const unsigned char * in,
    long sz
)
```

This function loads a certificate chain buffer into the WOLFSSL object. It behaves like the non-buffered version, only differing in its ability to be called with a buffer as input instead of a file. The buffer is provided by the in argument of size sz. The buffer must be in PEM format and start with the subject's certificate, ending with the root certificate. Please see the examples for proper usage.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().
- **in** buffer containing certificate to load.
- **sz** size of the certificate located in buffer.

See:

- wolfSSL_CTX_load_verify_buffer
- wolfSSL_CTX_use_certificate_buffer
- wolfSSL_CTX_use_PrivateKey_buffer
- wolfSSL_CTX_use_certificate_chain_buffer
- wolfSSL_use_certificate_buffer
- wolfSSL_use_PrivateKey_buffer

Return:

- SSL_SUCCES upon success.
- SSL_BAD_FILETYPE will be returned if the file is the wrong format.
- SSL_BAD_FILE will be returned if the file doesn't exist, can't be read, or is corrupted.
- MEMORY_E will be returned if an out of memory condition occurs.
- ASN_INPUT_E will be returned if Base16 decoding fails on the file.
- BUFFER_E will be returned if a chain buffer is bigger than the receiving buffer.

Example

```
int ret;
byte certChainBuff[...];
WOLFSSL* ssl = 0;
long buffSz = sizeof(certBuff);
...
ret = wolfSSL_use_certificate_chain_buffer(ssl, certChainBuff, buffSz);
if (ret != SSL_SUCCESS) {
    // failed to load certificate chain from buffer
}
```

```
int wolfSSL_UnloadCertsKeys(
    WOLFSSL *
)
```

This function unloads any certificates or keys that SSL owns.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See: wolfSSL_CTX_UnloadCAs

Return:

- SSL_SUCCESS - returned if the function executed successfully.
- BAD_FUNC_ARG - returned if the WOLFSSL object is NULL.

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
...
int unloadKeys = wolfSSL_UnloadCertsKeys(ssl);
if(unloadKeys != SSL_SUCCESS){
    // Failure case.
}
```

```
int wolfSSL_GetIVSize(
    WOLFSSL *
)
```

Returns the iv_size member of the specs structure held in the WOLFSSL struct.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_GetKeySize
- wolfSSL_GetClientWriteIV
- wolfSSL_GetServerWriteIV

Return:

- iv_size returns the value held in ssl->specs.iv_size.
- BAD_FUNC_ARG returned if the WOLFSSL structure is NULL.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
int ivSize;
...
ivSize = wolfSSL_GetIVSize(ssl);

if(ivSize > 0){
    // ivSize holds the specs.iv_size value.
}
```

```
void wolfSSL_KeepArrays(
    WOLFSSL *
)
```

Normally, at the end of the SSL handshake, wolfSSL frees temporary arrays. Calling this function before the handshake begins will prevent wolfSSL from freeing temporary arrays. Temporary arrays may be needed for things such as wolfSSL_get_keys() or PSK hints. When the user is done with temporary arrays, either wolfSSL_FreeArrays() may be called to free the resources immediately, or alternatively the resources will be freed when the associated SSL object is freed.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See: wolfSSL_FreeArrays

Return: none No return.

Example

```
WOLFSSL* ssl;
...
wolfSSL_KeepArrays(ssl);
```

```
void wolfSSL_FreeArrays(
    WOLFSSL *
)
```

Normally, at the end of the SSL handshake, wolfSSL frees temporary arrays. If wolfSSL_KeepArrays() has been called before the handshake, wolfSSL will not free temporary arrays. This function explicitly frees temporary arrays and should be called when the user is done with temporary arrays and does not want to wait for the SSL object to be freed to free these resources.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See: wolfSSL_KeepArrays

Return: none No return.

Example

```
WOLFSSL* ssl;
...
wolfSSL_FreeArrays(ssl);
```

```
int wolfSSL_DeriveTlsKeys(
    unsigned char * key_data,
    word32 keyLen,
    const unsigned char * ms,
    word32 msLen,
    const unsigned char * sr,
    const unsigned char * cr,
    int tls1_2,
    int hash_type
)
```

An external facing wrapper to derive TLS Keys.

Parameters:

- **key_data** a byte pointer that is allocateded in DeriveTlsKeys and passed through to wc_PRF to hold the final hash.
- **keyLen** a word32 type that is derived in DeriveTlsKeys from the WOLFSSL structure's specs member.
- **ms** a constant pointer type holding the master secret held in the arrays structure within the WOLFSSL structure.
- **msLen** a word32 type that holds the length of the master secret in an enumerated define, SECRET_LEN.
- **sr** a constant byte pointer to the serverRandom member of the arrays structure within the WOLFSSL structure.
- **cr** a constant byte pointer to the clientRandom member of the arrays structure within the WOLFSSL structure.
- **tls1_2** an integer type returned from IsAtLeastTLSv1_2().
- **hash_type** an integer type held in the WOLFSSL structure.

See:

- wc_PRF
- DeriveTlsKeys
- IsAtLeastTLSv1_2

Return:

- 0 returned on success.
- BUFFER_E returned if the sum of labLen and seedLen (computes total size) exceeds the maximum size.
- MEMORY_E returned if the allocation of memory failed.

Example

```
int DeriveTlsKeys(WOLFSSL* ssl){
int ret;
…
ret = wolfSSL_DeriveTlsKeys(key_data, length, ssl->arrays->masterSecret,
SECRET_LEN, ssl->arrays->clientRandom,
IsAtLeastTLSv1_2(ssl), ssl->specs.mac_algorithm);
…
}
```

```
int wolfSSL_X509_get_ext_by_NID(
    const WOLFSSL_X509 * x509,
    int nid,
    int lastPos
)
```

This function looks for and returns the extension index matching the passed in NID value.

Parameters:

- **x509** certificate to get parse through for extension.
- **nid** extension OID to be found.
- **lastPos** start search from extension after lastPos. Set to -1 initially.

Return:

- = 0 If successful the extension index is returned.
- -1 If extension is not found or error is encountered.

Example

```
const WOLFSSL_X509* x509;
int lastPos = -1;
int idx;

idx = wolfSSL_X509_get_ext_by_NID(x509, NID_basic_constraints, lastPos);
```

```
void * wolfSSL_X509_get_ext_d2i(
    const WOLFSSL_X509 * x509,
    int nid,
    int * c,
    int * idx
)
```

This function looks for and returns the extension matching the passed in NID value.

Parameters:

- **x509** certificate to get parse through for extension.
- **nid** extension OID to be found.
- **c** if not NULL is set to -2 for multiple extensions found -1 if not found, 0 if found and not critical and 1 if found and critical.
- **idx** if NULL return first extension matched otherwise if not stored in x509 start at idx.

See: wolfSSL_sk_ASN1_OBJECT_free

Return:

- pointer If successful a STACK_OF(WOLFSSL_ASN1_OBJECT) pointer is returned.
- NULL If extension is not found or error is encountered.

Example

```
const WOLFSSL_X509* x509;
int c;
int idx = 0;
STACK_OF(WOLFSSL_ASN1_OBJECT)* sk;

sk = wolfSSL_X509_get_ext_d2i(x509, NID_basic_constraints, &c, &idx);
//check sk for NULL and then use it. sk needs freed after done.
```

```
int wolfSSL_X509_digest(
    const WOLFSSL_X509 * x509,
    const WOLFSSL_EVP_MD * digest,
    unsigned char * buf,
    unsigned int * len
)
```

This function returns the hash of the DER certificate.

Parameters:

- **x509** certificate to get the hash of.
- **digest** the hash algorithm to use.
- **buf** buffer to hold hash.

- **len** length of buffer.

See: none

Return:

- SSL_SUCCESS On successfully creating a hash.
- SSL_FAILURE Returned on bad input or unsuccessful hash.

Example

```
WOLFSSL_X509* x509;
unsigned char buffer[64];
unsigned int bufferSz;
int ret;

ret = wolfSSL_X509_digest(x509, wolfSSL_EVP_sha256(), buffer, &bufferSz);
//check ret value
```

```
int wolfSSL_use_PrivateKey(
    WOLFSSL * ssl,
    WOLFSSL_EVP_PKEY * pkey
)
```

This is used to set the private key for the WOLFSSL structure.

Parameters:

- **ssl** WOLFSSL structure to set argument in.
- **pkey** private key to use.

See:

- wolfSSL_new
- wolfSSL_free

Return:

- SSL_SUCCESS On successful setting argument.
- SSL_FAILURE If a NULL ssl passed in. All error cases will be negative values.

Example

```
WOLFSSL* ssl;
WOLFSSL_EVP_PKEY* pkey;
int ret;
// create ssl object and set up private key
ret  = wolfSSL_use_PrivateKey(ssl, pkey);
// check ret value
```

```
int wolfSSL_use_PrivateKey_ASN1(
    int pri,
    WOLFSSL * ssl,
    unsigned char * der,
    long derSz
)
```

This is used to set the private key for the WOLFSSL structure. A DER formatted key buffer is expected.

Parameters:

- **pri** type of private key.
- **ssl** WOLFSSL structure to set argument in.
- **der** buffer holding DER key.
- **derSz** size of der buffer.

See:

- wolfSSL_new
- wolfSSL_free
- wolfSSL_use_PrivateKey

Return:

- SSL_SUCCESS On successful setting parsing and setting the private key.
- SSL_FAILURE If an NULL ssl passed in. All error cases will be negative values.

Example

```
WOLFSSL* ssl;
unsigned char* pkey;
long pkeySz;
int ret;
// create ssl object and set up private key
ret  = wolfSSL_use_PrivateKey_ASN1(1, ssl, pkey, pkeySz);
// check ret value
```

```
int wolfSSL_use_RSAPrivateKey_ASN1(
    WOLFSSL * ssl,
    unsigned char * der,
    long derSz
)
```

This is used to set the private key for the WOLFSSL structure. A DER formatted RSA key buffer is expected.

Parameters:

- **ssl** WOLFSSL structure to set argument in.
- **der** buffer holding DER key.

- **derSz** size of der buffer.

See:

- wolfSSL_new
- wolfSSL_free
- wolfSSL_use_PrivateKey

Return:

- SSL_SUCCESS On successful setting parsing and setting the private key.
- SSL_FAILURE If an NULL ssl passed in. All error cases will be negative values.

Example

```
WOLFSSL* ssl;
unsigned char* pkey;
long pkeySz;
int ret;
// create ssl object and set up RSA private key
ret  = wolfSSL_use_RSAPrivateKey_ASN1(ssl, pkey, pkeySz);
// check ret value
```

```
WOLFSSL_DH * wolfSSL_DSA_dup_DH(
    const WOLFSSL_DSA * r
)
```

This function duplicates the parameters in dsa to a newly created WOLFSSL_DH structure.

Parameters:

- **dsa** WOLFSSL_DSA structure to duplicate.

See: none

Return:

- WOLFSSL_DH If duplicated returns WOLFSSL_DH structure
- NULL upon failure

Example

```
WOLFSSL_DH* dh;
WOLFSSL_DSA* dsa;
// set up dsa
dh = wolfSSL_DSA_dup_DH(dsa);

// check dh is not null
```

```
WOLFSSL_X509 * wolfSSL_d2i_X509_bio(
    WOLFSSL_BIO * bio,
    WOLFSSL_X509 ** x509
)
```

This function get the DER buffer from bio and converts it to a WOLFSSL_X509 structure.

Parameters:

- **bio** pointer to the WOLFSSL_BIO structure that has the DER certificate buffer.
- **x509** pointer that get set to new WOLFSSL_X509 structure created.

See: none

Return:

- pointer returns a WOLFSSL_X509 structure pointer on success.
- Null returns NULL on failure

Example

```
WOLFSSL_BIO* bio;
WOLFSSL_X509* x509;
// load DER into bio
x509 = wolfSSL_d2i_X509_bio(bio, NULL);
Or
wolfSSL_d2i_X509_bio(bio, &x509);
// use x509 returned (check for NULL)
```

```
WOLFSSL_X509 * wolfSSL_PEM_read_bio_X509_AUX(
    WOLFSSL_BIO * bp,
    WOLFSSL_X509 ** x,
    wc_pem_password_cb * cb,
    void * u
)
```

This function behaves the same as wolfSSL_PEM_read_bio_X509. AUX signifies containing extra information such as trusted/rejected use cases and friendly name for human readability.

Parameters:

- **bp** WOLFSSL_BIO structure to get PEM buffer from.
- **x** if setting WOLFSSL_X509 by function side effect.
- **cb** password callback.
- **u** NULL terminated user password.

See: wolfSSL_PEM_read_bio_X509

Return:

- WOLFSSL_X509 on successfully parsing the PEM buffer a WOLFSSL_X509 structure is returned.

- Null if failed to parse PEM buffer.

Example

```
WOLFSSL_BIO* bio;
WOLFSSL_X509* x509;
// setup bio
X509 = wolfSSL_PEM_read_bio_X509_AUX(bio, NULL, NULL, NULL);
//check x509 is not null and then use it
```

```
long wolfSSL_CTX_set_tmp_dh(
    WOLFSSL_CTX * ctx,
    WOLFSSL_DH * dh
)
```

Initializes the WOLFSSL_CTX structure's dh member with the Diffie-Hellman parameters.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().
- **dh** a pointer to a WOLFSSL_DH structure.

See: wolfSSL_BN_bn2bin

Return:

- SSL_SUCCESS returned if the function executed successfully.
- BAD_FUNC_ARG returned if the ctx or dh structures are NULL.
- SSL_FATAL_ERROR returned if there was an error setting a structure value.
- MEMORY_E returned if their was a failure to allocate memory.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
WOLFSSL_DH* dh;
…
return wolfSSL_CTX_set_tmp_dh(ctx, dh);
```

```
WOLFSSL_DSA * wolfSSL_PEM_read_bio_DSAparams(
    WOLFSSL_BIO * bp,
    WOLFSSL_DSA ** x,
    wc_pem_password_cb * cb,
    void * u
)
```

This function get the DSA parameters from a PEM buffer in bio.

Parameters:

- **bio** pointer to the WOLFSSL_BIO structure for getting PEM memory pointer.
- **x** pointer to be set to new WOLFSSL_DSA structure.
- **cb** password callback function.
- **u** null terminated password string.

See: none

Return:

- WOLFSSL_DSA on successfully parsing the PEM buffer a WOLFSSL_DSA structure is created and returned.
- Null if failed to parse PEM buffer.

Example

```
WOLFSSL_BIO* bio;
WOLFSSL_DSA* dsa;
// setup bio
dsa = wolfSSL_PEM_read_bio_DSAparams(bio, NULL, NULL, NULL);

// check dsa is not NULL and then use dsa
```

```
WOLF_STACK_OF(
    WOLFSSL_X509
) const
```

This function gets the peer's certificate chain.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_X509_get_issuer_name
- wolfSSL_X509_get_subject_name
- wolfSSL_X509_get_isCA

Return:

- pointer returns a pointer to the peer's Certificate stack.
- NULL returned if no peer certificate.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
...
wolfSSL_connect(ssl);
STACK_OF(WOLFSSL_X509)* chain = wolfSSL_get_peer_cert_chain(ssl);
```

```
ifchain){
    // You have a pointer to the peer certificate chain
}
```

```
char * wolfSSL_X509_get_next_altname(
    WOLFSSL_X509 *
)
```

This function returns the next, if any, altname from the peer certificate.

Parameters:

- **cert** a pointer to the wolfSSL_X509 structure.

See:

- wolfSSL_X509_get_issuer_name
- wolfSSL_X509_get_subject_name

Return:

- NULL if there is not a next altname.
- cert->altNamesNext->name from the WOLFSSL_X509 structure that is a string value from the altName list is returned if it exists.

Example

```
WOLFSSL_X509 x509 = (WOLFSSL_X509*)XMALLOC(sizeof(WOLFSSL_X509), NULL,
                                                    DYNAMIC_TYPE_X509);
…
int x509NextAltName = wolfSSL_X509_get_next_altname(x509);
if(x509NextAltName == NULL){
        //There isn't another alt name
}
```

```
WOLFSSL_ASN1_TIME * wolfSSL_X509_get_notBefore(
    WOLFSSL_X509 *
)
```

The function checks to see if x509 is NULL and if it's not, it returns the notBefore member of the x509 struct.

Parameters:

- **x509** a pointer to the WOLFSSL_X509 struct.

See: wolfSSL_X509_get_notAfter

Return:

- pointer to struct with ASN1_TIME to the notBefore member of the x509 struct.
- NULL the function returns NULL if the x509 structure is NULL.

Example

```
WOLFSSL_X509* x509 = (WOLFSSL_X509)XMALLOC(sizeof(WOLFSSL_X509), NULL,
DYNAMIC_TYPE_X509) ;
…
const WOLFSSL_ASN1_TIME* notAfter = wolfSSL_X509_get_notBefore(x509);
if(notAfter == NULL){
        //The x509 object was NULL
}
```

A.5 wolfSSL Connection, Session, and I/O

A.4.2.88 function wolfSSL_X509_get_notBefore

A.5.1 Functions

	Name
long	**wolfSSL_get_verify_depth**(WOLFSSL * ssl)This function returns the maximum chain depth allowed, which is 9 by default, for a valid session i.e. there is a non_null session object (ssl).
char *	**wolfSSL_get_cipher_list**(int priority)Get the name of cipher at priority level passed in.
int	**wolfSSL_get_ciphers**(char * buf, int len)This function gets the ciphers enabled in wolfSSL.
const char *	**wolfSSL_get_cipher_name**(WOLFSSL * ssl)This function gets the cipher name in the format DHE-RSA by passing through argument to wolfSSL_get_cipher_name_internal.
int	**wolfSSL_get_fd**(const WOLFSSL *)This function returns the read file descriptor (fd) used as the input facility for the SSL connection. Typically this will be a socket file descriptor.
int	**wolfSSL_get_wfd**(const WOLFSSL *)This function returns the write file descriptor (fd) used as the output facility for the SSL connection. Typically this will be a socket file descriptor.

	Name
int	**wolfSSL_get_using_nonblock**(WOLFSSL *)This function allows the application to determine if wolfSSL is using non_blocking I/O. If wolfSSL is using non_blocking I/O, this function will return 1, otherwise 0. After an application creates a WOLFSSL object, if it will be used with a non_blocking socket, call wolfSSL_set_using_nonblock() on it. This lets the WOLFSSL object know that receiving EWOULDBLOCK means that the recvfrom call would block rather than that it timed out.
int	**wolfSSL_write will yield either SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE. The calling process must then repeat the call to wolfSSL_write() when the underlying I/O is ready. If the underlying I/O is blocking, wolfSSL_write() will only return once the buffer data of size sz has been completely written or an error occurred.
int	**wolfSSL_read**(WOLFSSL * ssl, void * data, int sz)This function reads sz bytes from the SSL session (ssl) internal read buffer into the buffer data. The bytes read are removed from the internal receive buffer. If necessary wolfSSL_read() will negotiate an SSL/TLS session if the handshake has not already been performed yet by wolfSSL_connect() or wolfSSL_accept(). The SSL/TLS protocol uses SSL records which have a maximum size of 16kB (the max record size can be controlled by the MAX_RECORD_SIZE define in /wolfssl/internal.h). As such, wolfSSL needs to read an entire SSL record internally before it is able to process and decrypt the record. Because of this, a call to wolfSSL_read() will only be able to return the maximum buffer size which has been decrypted at the time of calling. There may be additional not_yet_decrypted data waiting in the internal wolfSSL receive buffer which will be retrieved and decrypted with the next call to wolfSSL_read(). If sz is larger than the number of bytes in the internal read buffer, SSL_read() will return the bytes available in the internal read buffer. If no bytes are buffered in the internal read buffer yet, a call to wolfSSL_read() will trigger processing of the next record.

	Name
int	**wolfSSL_peek. If sz is larger than the number of bytes in the internal read buffer, SSL_peek() will return the bytes available in the internal read buffer. If no bytes are buffered in the internal read buffer yet, a call to wolfSSL_peek() will trigger processing of the next record.
int	**wolfSSL_accept will yield either SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE. The calling process must then repeat the call to wolfSSL_accept when data is available to read and wolfSSL will pick up where it left off. When using a non_blocking socket, nothing needs to be done, but select() can be used to check for the required condition. If the underlying I/O is blocking, wolfSSL_accept() will only return once the handshake has been finished or an error occurred.
int	**wolfDTLS_accept_stateless**(WOLFSSL * ssl)This function is called on the server side and statelessly listens for an SSL client to initiate the DTLS handshake.
int	**wolfSSL_send will yield either SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE. The calling process must then repeat the call to wolfSSL_send() when the underlying I/O is ready. If the underlying I/O is blocking, wolfSSL_send() will only return once the buffer data of size sz has been completely written or an error occurred.
int	**wolfSSL_recv. The SSL/TLS protocol uses SSL records which have a maximum size of 16kB (the max record size can be controlled by the MAX_RECORD_SIZE define in /wolfssl/internal.h). As such, wolfSSL needs to read an entire SSL record internally before it is able to process and decrypt the record. Because of this, a call to wolfSSL_recv() will only be able to return the maximum buffer size which has been decrypted at the time of calling. There may be additional not_yet_decrypted data waiting in the internal wolfSSL receive buffer which will be retrieved and decrypted with the next call to wolfSSL_recv(). If sz is larger than the number of bytes in the internal read buffer, SSL_recv() will return the bytes available in the internal read buffer. If no bytes are buffered in the internal read buffer yet, a call to wolfSSL_recv() will trigger processing of the next record.

	Name
int	**wolfSSL_get_alert_history**(WOLFSSL * ssl, WOLFSSL_ALERT_HISTORY * h)This function gets the alert history.
WOLFSSL_SESSION *	**wolfSSL_get_session**(WOLFSSL * ssl)When NO_SESSION_CACHE_REF is defined this function returns a pointer to the current session (WOLFSSL_SESSION) used in ssl. This function returns a non_persistent pointer to the WOLFSSL_SESSION object. The pointer returned will be freed when wolfSSL_free is called. This call should only be used to inspect or modify the current session. For session resumption it is recommended to use wolfSSL_get1_session(). For backwards compatibility when NO_SESSION_CACHE_REF is not defined this function returns a persistent session object pointer that is stored in the local cache. The cache size is finite and there is a risk that the session object will be overwritten by another ssl connection by the time the application calls wolfSSL_set_session() on it. It is recommended to define NO_SESSION_CACHE_REF in your application and to use wolfSSL_get1_session() for session resumption.
void	**wolfSSL_flush_sessions**(WOLFSSL_CTX * ctx, long tm)This function flushes session from the session cache which have expired. The time, tm, is used for the time comparison. Note that wolfSSL currently uses a static table for sessions, so no flushing is needed. As such, this function is currently just a stub. This function provides OpenSSL compatibility (SSL_flush_sessions) when wolfSSL is compiled with the OpenSSL compatibility layer.
int	**wolfSSL_GetSessionIndex**(WOLFSSL * ssl)This function gets the session index of the WOLFSSL structure.
int	**wolfSSL_GetSessionAtIndex**(int idx, WOLFSSL_SESSION * session)This function gets the session at specified index of the session cache and copies it into memory. The WOLFSSL_SESSION structure holds the session information.
WOLFSSL_X509_CHAIN *	**wolfSSL_SESSION_get_peer_chain**(WOLFSSL_SESSION * session)Returns the peer certificate chain from the WOLFSSL_SESSION struct.
int	**wolfSSL_pending.
int	**wolfSSL_save_session_cache**(const char * fname)This function persists the session cache to file. It doesn't use memsave because of additional memory use.

	Name
int	**wolfSSL_restore_session_cache**(const char * fname)This function restores the persistent session cache from file. It does not use memstore because of additional memory use.
int	**wolfSSL_memsave_session_cache**(void * mem, int sz)This function persists session cache to memory.
int	**wolfSSL_memrestore_session_cache**(const void * mem, int sz)This function restores the persistent session cache from memory.
int	**wolfSSL_get_session_cache_memsize**(void)This function returns how large the session cache save buffer should be.
int	**wolfSSL_session_reused**(WOLFSSL * ssl)This function returns the resuming member of the options struct. The flag indicates whether or not to reuse a session. If not, a new session must be established.
const char *	**wolfSSL_get_version**(WOLFSSL * ssl)Returns the SSL version being used as a string.
int	**wolfSSL_get_current_cipher_suite**(WOLFSSL * ssl)Returns the current cipher suit an ssl session is using.
WOLFSSL_CIPHER *	**wolfSSL_get_current_cipher**(WOLFSSL * ssl)This function returns a pointer to the current cipher in the ssl session.
const char *	**wolfSSL_CIPHER_get_name**(const WOLFSSL_CIPHER * cipher)This function matches the cipher suite in the SSL object with the available suites and returns the string representation.
const char *	**wolfSSL_get_cipher**(WOLFSSL *)This function matches the cipher suite in the SSL object with the available suites.
int	**wolfSSL_BIO_get_mem_data**(WOLFSSL_BIO * bio, void * p)This is used to set a byte pointer to the start of the internal memory buffer.
long	**wolfSSL_BIO_set_fd**(WOLFSSL_BIO * b, int fd, int flag)Sets the file descriptor for bio to use.
int	**wolfSSL_BIO_set_close**(WOLFSSL_BIO * b, long flag)Sets the close flag, used to indicate that the i/o stream should be closed when the BIO is freed.
WOLFSSL_BIO_METHOD *	**wolfSSL_BIO_s_socket**(void)This is used to get a BIO_SOCKET type WOLFSSL_BIO_METHOD.
int	**wolfSSL_BIO_set_write_buf_size**(WOLFSSL_BIO * b, long size)This is used to set the size of write buffer for a WOLFSSL_BIO. If write buffer has been previously set this function will free it when resetting the size. It is similar to wolfSSL_BIO_reset in that it resets read and write indexes to 0.

	Name
int	**wolfSSL_BIO_make_bio_pair**(WOLFSSL_BIO * b1, WOLFSSL_BIO * b2)This is used to pair two bios together. A pair of bios acts similar to a two way pipe writing to one can be read by the other and vice versa. It is expected that both bios be in the same thread, this function is not thread safe. Freeing one of the two bios removes both from being paired. If a write buffer size was not previously set for either of the bios it is set to a default size of 17000 (WOLFSSL_BIO_SIZE) before being paired.
int	**wolfSSL_BIO_ctrl_reset_read_request**(WOLFSSL_BIO * bio)This is used to set the read request flag back to 0.
int	**wolfSSL_BIO_nread0**(WOLFSSL_BIO * bio, char ** buf)This is used to get a buffer pointer for reading from. Unlike wolfSSL_BIO_nread the internal read index is not advanced by the number returned from the function call. Reading past the value returned can result in reading out of array bounds.
int	**wolfSSL_BIO_nread**(WOLFSSL_BIO * bio, char ** buf, int num)This is used to get a buffer pointer for reading from. The internal read index is advanced by the number returned from the function call with buf being pointed to the beginning of the buffer to read from. In the case that less bytes are in the read buffer than the value requested with num the lesser value is returned. Reading past the value returned can result in reading out of array bounds.
int	**wolfSSL_BIO_nwrite**(WOLFSSL_BIO * bio, char ** buf, int num)Gets a pointer to the buffer for writing as many bytes as returned by the function. Writing more bytes to the pointer returned then the value returned can result in writing out of bounds.
int	**wolfSSL_BIO_reset**(WOLFSSL_BIO * bio)Resets bio to an initial state. As an example for type BIO_BIO this resets the read and write index.
int	**wolfSSL_BIO_seek**(WOLFSSL_BIO * bio, int ofs)This function adjusts the file pointer to the offset given. This is the offset from the head of the file.
int	**wolfSSL_BIO_write_filename**(WOLFSSL_BIO * bio, char * name)This is used to set and write to a file. WIll overwrite any data currently in the file and is set to close the file when the bio is freed.

	Name
long	**wolfSSL_BIO_set_mem_eof_return**(WOLFSSL_BIO * bio, int v)This is used to set the end of file value. Common value is -1 so as not to get confused with expected positive values.
long	**wolfSSL_BIO_get_mem_ptr**(WOLFSSL_BIO * bio, WOLFSSL_BUF_MEM ** m)This is a getter function for WOLFSSL_BIO memory pointer.
const char *	**wolfSSL_lib_version**(void)This function returns the current library version.
word32	**wolfSSL_lib_version_hex**(void)This function returns the current library version in hexadecimal notation.
int	**wolfSSL_negotiate is performed if called from the server side.
int	**wolfSSL_connect_cert will yield either SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE. The calling process must then repeat the call to wolfSSL_connect_cert() when the underlying I/O is ready and wolfSSL will pick up where it left off. When using a non_blocking socket, nothing needs to be done, but select() can be used to check for the required condition. If the underlying I/O is blocking, wolfSSL_connect_cert() will only return once the peer's certificate chain has been received.
int	**wolfSSL_writev**(WOLFSSL * ssl, const struct iovec * iov, int iovcnt)Simulates writev semantics but doesn't actually do block at a time because of SSL_write() behavior and because front adds may be small. Makes porting into software that uses writev easier.
unsigned char	**wolfSSL_SNI_Status**(WOLFSSL * ssl, unsigned char type)This function gets the status of an SNI object.
int	**wolfSSL_UseSecureRenegotiation**(WOLFSSL * ssl)This function forces secure renegotiation for the supplied WOLFSSL structure. This is not recommended.
int	**wolfSSL_Rehandshake**(WOLFSSL * ssl)This function executes a secure renegotiation handshake; this is user forced as wolfSSL discourages this functionality.
int	**wolfSSL_UseSessionTicket**(WOLFSSL * ssl)Force provided WOLFSSL structure to use session ticket. The constant HAVE_SESSION_TICKET should be defined and the constant NO_WOLFSSL_CLIENT should not be defined to use this function.

	Name
int	**wolfSSL_get_SessionTicket**(WOLFSSL * ssl, unsigned char * buf, word32 * bufSz)This function copies the ticket member of the Session structure to the buffer.
int	**wolfSSL_set_SessionTicket**(WOLFSSL * ssl, const unsigned char * buf, word32 bufSz)This function sets the ticket member of the WOLFSSL_SESSION structure within the WOLFSSL struct. The buffer passed into the function is copied to memory.
int	**wolfSSL_PrintSessionStats**(void)This function prints the statistics from the session.
int	**wolfSSL_get_session_stats**(unsigned int * active, unsigned int * total, unsigned int * peak, unsigned int * maxSessions)This function gets the statistics for the session.
long	**wolfSSL_BIO_set_fp**(WOLFSSL_BIO * bio, XFILE fp, int c)This is used to set the internal file pointer for a BIO.
long	**wolfSSL_BIO_get_fp**(WOLFSSL_BIO * bio, XFILE * fp)This is used to get the internal file pointer for a BIO.
size_t	**wolfSSL_BIO_ctrl_pending**(WOLFSSL_BIO * b)Gets the number of pending bytes to read. If BIO type is BIO_BIO then is the number to read from pair. If BIO contains an SSL object then is pending data from SSL object (wolfSSL_pending(ssl)). If is BIO_MEMORY type then returns the size of memory buffer.
int	**wolfSSL_set_jobject**(WOLFSSL * ssl, void * objPtr)This function sets the jObjectRef member of the WOLFSSL structure.
void *	**wolfSSL_get_jobject**(WOLFSSL * ssl)This function returns the jObjectRef member of the WOLFSSL structure.

	Name
int	**wolfSSL_connect will yield either SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE. The calling process must then repeat the call to wolfSSL_connect() when the underlying I/O is ready and wolfSSL will pick up where it left off. When using a non_blocking socket, nothing needs to be done, but select() can be used to check for the required condition. If the underlying I/O is blocking, wolfSSL_connect() will only return once the handshake has been finished or an error occurred. wolfSSL takes a different approach to certificate verification than OpenSSL does. The default policy for the client is to verify the server, this means that if you don't load CAs to verify the server you'll get a connect error, unable to verify (_155). It you want to mimic OpenSSL behavior of having SSL_connect succeed even if verifying the server fails and reducing security you can do this by calling: SSL_CTX_set_verify(ctx, SSL_VERIFY_NONE, 0); before calling SSL_new(); Though it's not recommended.
int	**wolfSSL_update_keys**(WOLFSSL * ssl)This function is called on a TLS v1.3 client or server wolfSSL to force the rollover of keys. A KeyUpdate message is sent to the peer and new keys are calculated for encryption. The peer will send back a KeyUpdate message and the new decryption keys will then be calculated. This function can only be called after a handshake has been completed.
int	**wolfSSL_key_update_response is called, a KeyUpdate message is sent and the encryption key is updated. The decryption key is updated when the response is received.
int	**wolfSSL_request_certificate**(WOLFSSL * ssl)This function requests a client certificate from the TLS v1.3 client. This is useful when a web server is serving some pages that require client authentication and others that don't. A maximum of 256 requests can be sent on a connection.

	Name
int	**wolfSSL_connect_TLSv13 will only return once the handshake has been finished or an error occurred. wolfSSL takes a different approach to certificate verification than OpenSSL does. The default policy for the client is to verify the server, this means that if you don't load CAs to verify the server you'll get a connect error, unable to verify (_155). It you want to mimic OpenSSL behavior of having SSL_connect succeed even if verifying the server fails and reducing security you can do this by calling: SSL_CTX_set_verify(ctx, SSL_VERIFY_NONE, 0); before calling SSL_new(); Though it's not recommended. **wolfSSL_accept_TLSv13 will only return once the handshake has been finished or an error occurred. Call this function when expecting a TLS v1.3 connection though older version ClientHello messages are supported.
int	**wolfSSL_write_early_data to connect to the server and send the data in the handshake. This function is only used with clients.
int	**wolfSSL_read_early_data to accept a client and read any early data in the handshake. The function should be invoked until wolfSSL_is_init_finished() returns true. Early data may be sent by the client in multiple messages. If there is no early data then the handshake will be processed as normal. This function is only used with servers.
int	**wolfSSL_inject to extract the plaintext data from the WOLFSSL object.
void *	**wolfSSL_GetIOReadCtx**(WOLFSSL * ssl)This function returns the IOCB_ReadCtx member of the WOLFSSL struct.
void *	**wolfSSL_GetIOWriteCtx**(WOLFSSL * ssl)This function returns the IOCB_WriteCtx member of the WOLFSSL structure.
void	**wolfSSL_SetIO_NetX**(WOLFSSL * ssl, NX_TCP_SOCKET * nxsocket, ULONG waitoption)This function sets the nxSocket and nxWait members of the nxCtx struct within the WOLFSSL structure.

A.5.2 Functions Documentation

```
long wolfSSL_get_verify_depth(
    WOLFSSL * ssl
)
```

This function returns the maximum chain depth allowed, which is 9 by default, for a valid session i.e. there is a non-null session object (ssl).

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See: wolfSSL_CTX_get_verify_depth

Return:

- MAX_CHAIN_DEPTH returned if the WOLFSSL structure is not NULL. By default the value is 9.
- BAD_FUNC_ARG returned if the WOLFSSL structure is NULL.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
WOLFSSL* ssl = wolfSSL_new(ctx);
...
long sslDep = wolfSSL_get_verify_depth(ssl);

if(sslDep > EXPECTED){
    // The verified depth is greater than what was expected
} else {
    // The verified depth is smaller or equal to the expected value
}
```

```
char * wolfSSL_get_cipher_list(
    int priority
)
```

Get the name of cipher at priority level passed in.

Parameters:

- **priority** Integer representing the priority level of a cipher.

See:

- wolfSSL_CIPHER_get_name
- wolfSSL_get_current_cipher

Return:

- string Success
- 0 Priority is either out of bounds or not valid.

Example

```
printf("The cipher at 1 is %s", wolfSSL_get_cipher_list(1));
```

```
int wolfSSL_get_ciphers(
    char * buf,
    int len
)
```

This function gets the ciphers enabled in wolfSSL.

Parameters:

- **buf** a char pointer representing the buffer.
- **len** the length of the buffer.

See:

- GetCipherNames
- wolfSSL_get_cipher_list
- ShowCiphers

Return:

- SSL_SUCCESS returned if the function executed without error.
- BAD_FUNC_ARG returned if the buf parameter was NULL or if the len argument was less than or equal to zero.
- BUFFER_E returned if the buffer is not large enough and will overflow.

Example

```
static void ShowCiphers(void){
char* ciphers;
int ret = wolfSSL_get_ciphers(ciphers, (int)sizeof(ciphers));

if(ret == SSL_SUCCES){
        printf("%s\n", ciphers);
    }
}
```

```
const char * wolfSSL_get_cipher_name(
    WOLFSSL * ssl
)
```

This function gets the cipher name in the format DHE-RSA by passing through argument to wolfSSL_get_cipher_name_internal.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_CIPHER_get_name
- wolfSSL_get_current_cipher
- wolfSSL_get_cipher_name_internal

Return:

- string This function returns the string representation of the cipher suite that was matched.
- NULL error or cipher not found.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
WOLFSSL* ssl = wolfSSL_new(ctx);
…
char* cipherS = wolfSSL_get_cipher_name(ssl);

if(cipher == NULL){
    // There was not a cipher suite matched
} else {
    // There was a cipher suite matched
    printf("%s\n", cipherS);
}
```

```
int wolfSSL_get_fd(
    const WOLFSSL *
)
```

This function returns the read file descriptor (fd) used as the input facility for the SSL connection. Typically this will be a socket file descriptor.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().

See:

- wolfSSL_set_fd
- wolfSSL_set_read_fd
- wolfSSL_set_write_fd

Return: fd If successful the call will return the SSL session file descriptor.

Example

```
int sockfd;
WOLFSSL* ssl = 0;
...
sockfd = wolfSSL_get_fd(ssl);
...
```

```
int wolfSSL_get_wfd(
    const WOLFSSL *
)
```

This function returns the write file descriptor (fd) used as the output facility for the SSL connection. Typically this will be a socket file descriptor.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().

See:

- wolfSSL_set_fd
- wolfSSL_set_read_fd
- wolfSSL_set_write_fd

Return: fd If successful the call will return the SSL session file descriptor.

Example

```
int sockfd;
WOLFSSL* ssl = 0;
...
sockfd = wolfSSL_get_wfd(ssl);
...
```

```
int wolfSSL_get_using_nonblock(
    WOLFSSL *
)
```

This function allows the application to determine if wolfSSL is using non-blocking I/O. If wolfSSL is using non-blocking I/O, this function will return 1, otherwise 0. After an application creates a WOLFSSL object, if it will be used with a non-blocking socket, call wolfSSL_set_using_nonblock() on it. This lets the WOLFSSL object know that receiving EWOULDBLOCK means that the recvfrom call would block rather than that it timed out.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().

See: wolfSSL_set_session

Return:

- 0 underlying I/O is blocking.
- 1 underlying I/O is non-blocking.

Example

```
int ret = 0;
WOLFSSL* ssl = 0;
...
ret = wolfSSL_get_using_nonblock(ssl);
if (ret == 1) {
    // underlying I/O is non-blocking
}
...
```

```
int wolfSSL_write(
    WOLFSSL * ssl,
    const void * data,
    int sz
)
```

This function writes sz bytes from the buffer, data, to the SSL connection, ssl. If necessary, wolfSSL_write() will negotiate an SSL/TLS session if the handshake has not already been performed yet by wolfSSL_connect() or wolfSSL_accept(). When using (D)TLSv1.3 and early data feature is compiled in, this function progresses the handshake only up to the point when it is possible to send data. Next invocations of wolfSSL_Connect()/wolfSSL_Accept()/wolfSSL_read() will complete the handshake. wolfSSL_write() works with both blocking and non-blocking I/O. When the underlying I/O is non-blocking, wolfSSL_write() will return when the underlying I/O could not satisfy the needs of wolfSSL_write() to continue. In this case, a call to wolfSSL_get_error() will yield either SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE. The calling process must then repeat the call to wolfSSL_write() when the underlying I/O is ready. If the underlying I/O is blocking, wolfSSL_write() will only return once the buffer data of size sz has been completely written or an error occurred.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().
- **data** data buffer which will be sent to peer.
- **sz** size, in bytes, of data to send to the peer (data).

See:

- wolfSSL_send
- wolfSSL_read
- wolfSSL_recv

Return:

- 0 the number of bytes written upon success.
- 0 will be returned upon failure. Call wolfSSL_get_error() for the specific error code.
- SSL_FATAL_ERROR will be returned upon failure when either an error occurred or, when using non-blocking sockets, the SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE error was received and and the application needs to call wolfSSL_write() to get a specific error code.

Example

```
WOLFSSL* ssl = 0;
char msg[64] = "hello wolfssl!";
int msgSz = (int)strlen(msg);
int flags;
int ret;
...

ret = wolfSSL_write(ssl, msg, msgSz);
if (ret <= 0) {
    // wolfSSL_write() failed, call wolfSSL_get_error()
}
```

```
int wolfSSL_read(
    WOLFSSL * ssl,
    void * data,
    int sz
)
```

This function reads sz bytes from the SSL session (ssl) internal read buffer into the buffer data. The bytes read are removed from the internal receive buffer. If necessary wolfSSL_read() will negotiate an SSL/TLS session if the handshake has not already been performed yet by wolfSSL_connect() or wolfSSL_accept(). The SSL/TLS protocol uses SSL records which have a maximum size of 16kB (the max record size can be controlled by the MAX_RECORD_SIZE define in /wolfssl/internal.h). As such, wolfSSL needs to read an entire SSL record internally before it is able to process and decrypt the record. Because of this, a call to wolfSSL_read() will only be able to return the maximum buffer size which has been decrypted at the time of calling. There may be additional not-yet-decrypted data waiting in the internal wolfSSL receive buffer which will be retrieved and decrypted with the next call to wolfSSL_read(). If sz is larger than the number of bytes in the internal read buffer, SSL_read() will return the bytes available in the internal read buffer. If no bytes are buffered in the internal read buffer yet, a call to wolfSSL_read() will trigger processing of the next record.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().
- **data** buffer where wolfSSL_read() will place data read.
- **sz** number of bytes to read into data.

See:

- wolfSSL_recv
- wolfSSL_write
- wolfSSL_peek
- wolfSSL_pending

Return:

- 0 the number of bytes read upon success.

- 0 will be returned upon failure. This may be caused by a either a clean (close notify alert) shutdown or just that the peer closed the connection. Call wolfSSL_get_error() for the specific error code.
- SSL_FATAL_ERROR will be returned upon failure when either an error occurred or, when using nonblocking sockets, the SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE error was received and and the application needs to call wolfSSL_read() to get a specific error code.

Example

```
WOLFSSL* ssl = 0;
char reply[1024];
...

input = wolfSSL_read(ssl, reply, sizeof(reply));
if (input > 0) {
    // "input" number of bytes returned into buffer "reply"
}

See wolfSSL examples (client, server, echoclient, echoserver) for more
complete examples of wolfSSL_read().
```

```
int wolfSSL_peek(
    WOLFSSL * ssl,
    void * data,
    int sz
)
```

This function copies sz bytes from the SSL session (ssl) internal read buffer into the buffer data. This function is identical to wolfSSL_read(). If sz is larger than the number of bytes in the internal read buffer, SSL_peek() will return the bytes available in the internal read buffer. If no bytes are buffered in the internal read buffer yet, a call to wolfSSL_peek() will trigger processing of the next record.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().
- **data** buffer where wolfSSL_peek() will place data read.
- **sz** number of bytes to read into data.

See: wolfSSL_read

Return:

- 0 the number of bytes read upon success.
- 0 will be returned upon failure. This may be caused by a either a clean (close notify alert) shutdown or just that the peer closed the connection. Call wolfSSL_get_error() for the specific error code.
- SSL_FATAL_ERROR will be returned upon failure when either an error occurred or, when using nonblocking sockets, the SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE error was received and and the application needs to call wolfSSL_peek() to get a specific error code.

Example

```
WOLFSSL* ssl = 0;
char reply[1024];
...

input = wolfSSL_peek(ssl, reply, sizeof(reply));
if (input > 0) {
    // “input” number of bytes returned into buffer “reply”
}
```

```
int wolfSSL_accept(
    WOLFSSL *
)
```

This function is called on the server side and waits for an SSL client to initiate the SSL/TLS handshake. When this function is called, the underlying communication channel has already been set up. wolfSSL_accept() works with both blocking and non-blocking I/O. When the underlying I/O is non-blocking, wolfSSL_accept() will return when the underlying I/O could not satisfy the needs of wolfSSL_accept to continue the handshake. In this case, a call to wolfSSL_get_error() will yield either SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE. The calling process must then repeat the call to wolfSSL_accept when data is available to read and wolfSSL will pick up where it left off. When using a non_blocking socket, nothing needs to be done, but select() can be used to check for the required condition. If the underlying I/O is blocking, wolfSSL_accept() will only return once the handshake has been finished or an error occurred.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_get_error
- wolfSSL_connect

Return:

- SSL_SUCCESS upon success.
- SSL_FATAL_ERROR will be returned if an error occurred. To get a more detailed error code, call wolfSSL_get_error().

Example

```
int ret = 0;
int err = 0;
WOLFSSL* ssl;
char buffer[80];
...
```

```
ret = wolfSSL_accept(ssl);
if (ret != SSL_SUCCESS) {
    err = wolfSSL_get_error(ssl, ret);
    printf("error = %d, %s\n", err, wolfSSL_ERR_error_string(err, buffer));
}
```

```
int wolfDTLS_accept_stateless(
    WOLFSSL * ssl
)
```

This function is called on the server side and statelessly listens for an SSL client to initiate the DTLS handshake.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_accept
- wolfSSL_get_error
- wolfSSL_connect

Return:

- WOLFSSL_SUCCESS ClientHello containing a valid cookie was received. The connection can be continued with wolfSSL_accept().
- WOLFSSL_FAILURE The I/O layer returned WANT_READ. This is either because there is no data to read and we are using non-blocking sockets or we sent a cookie request and we are waiting for a reply. The user should call wolfDTLS_accept_stateless again after data becomes available in the I/O layer.
- WOLFSSL_FATAL_ERROR A fatal error occurred. The ssl object should be free'd and allocated again to continue.

Example

```
int ret = 0;
int err = 0;
WOLFSSL* ssl;
...
do {
    ret = wolfDTLS_accept_stateless(ssl);
    if (ret == WOLFSSL_FATAL_ERROR)
        // re-allocate the ssl object with wolfSSL_free() and wolfSSL_new()
} while (ret != WOLFSSL_SUCCESS);
ret = wolfSSL_accept(ssl);
if (ret != SSL_SUCCESS) {
    err = wolfSSL_get_error(ssl, ret);
    printf("error = %d, %s\n", err, wolfSSL_ERR_error_string(err, buffer));
}
```

```
int wolfSSL_send(
    WOLFSSL * ssl,
    const void * data,
    int sz,
    int flags
)
```

This function writes sz bytes from the buffer, data, to the SSL connection, ssl, using the specified flags for the underlying write operation. If necessary wolfSSL_send() will negotiate an SSL/TLS session if the handshake has not already been performed yet by wolfSSL_connect() or wolfSSL_accept() will yield either SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE. The calling process must then repeat the call to wolfSSL_send() when the underlying I/O is ready. If the underlying I/O is blocking, wolfSSL_send() will only return once the buffer data of size sz has been completely written or an error occurred.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().
- **data** data buffer to send to peer.
- **sz** size, in bytes, of data to be sent to peer.
- **flags** the send flags to use for the underlying send operation.

See:

- wolfSSL_write
- wolfSSL_read
- wolfSSL_recv

Return:

- 0 the number of bytes written upon success.
- 0 will be returned upon failure. Call wolfSSL_get_error() for the specific error code.
- SSL_FATAL_ERROR will be returned upon failure when either an error occurred or, when using non-blocking sockets, the SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE error was received and and the application needs to call wolfSSL_send() to get a specific error code.

Example

```
WOLFSSL* ssl = 0;
char msg[64] = “hello wolfssl!”;
int msgSz = (int)strlen(msg);
int flags = ... ;
...

input = wolfSSL_send(ssl, msg, msgSz, flags);
if (input != msgSz) {
```

```
    // wolfSSL_send() failed
}



int wolfSSL_recv(
    WOLFSSL * ssl,
    void * data,
    int sz,
    int flags
)
```

This function reads sz bytes from the SSL session (ssl) internal read buffer into the buffer data using the specified flags for the underlying recv operation. The bytes read are removed from the internal receive buffer. This function is identical to wolfSSL_read(). The SSL/TLS protocol uses SSL records which have a maximum size of 16kB (the max record size can be controlled by the MAX_RECORD_SIZE define in /wolfssl/internal.h). As such, wolfSSL needs to read an entire SSL record internally before it is able to process and decrypt the record. Because of this, a call to wolfSSL_recv() will only be able to return the maximum buffer size which has been decrypted at the time of calling. There may be additional not_yet_decrypted data waiting in the internal wolfSSL receive buffer which will be retrieved and decrypted with the next call to wolfSSL_recv(). If sz is larger than the number of bytes in the internal read buffer, SSL_recv() will return the bytes available in the internal read buffer. If no bytes are buffered in the internal read buffer yet, a call to wolfSSL_recv() will trigger processing of the next record.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().
- **data** buffer where wolfSSL_recv() will place data read.
- **sz** number of bytes to read into data.
- **flags** the recv flags to use for the underlying recv operation.

See:

- wolfSSL_read
- wolfSSL_write
- wolfSSL_peek
- wolfSSL_pending

Return:

-
    0 the number of bytes read upon success.
- 0 will be returned upon failure. This may be caused by a either a clean (close notify alert) shutdown or just that the peer closed the connection. Call wolfSSL_get_error() for the specific error code.
- SSL_FATAL_ERROR will be returned upon failure when either an error occurred or, when using nonblocking sockets, the SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE error was received and and the application needs to call wolfSSL_recv() to get a specific error code.

Example

```
WOLFSSL* ssl = 0;
char reply[1024];
int flags = ... ;
...

input = wolfSSL_recv(ssl, reply, sizeof(reply), flags);
if (input > 0) {
    // “input” number of bytes returned into buffer “reply”
}
```

```
int wolfSSL_get_alert_history(
    WOLFSSL * ssl,
    WOLFSSL_ALERT_HISTORY * h
)
```

This function gets the alert history.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **h** a pointer to a WOLFSSL_ALERT_HISTORY structure that will hold the WOLFSSL struct's alert_history member's value.

See: wolfSSL_get_error

Return: SSL_SUCCESS returned when the function completed successfully. Either there was alert history or there wasn't, either way, the return value is SSL_SUCCESS.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(protocol method);
WOLFSSL* ssl = wolfSSL_new(ctx);
WOLFSSL_ALERT_HISTORY* h;
...
wolfSSL_get_alert_history(ssl, h);
// h now has a copy of the ssl->alert_history  contents
```

```
WOLFSSL_SESSION * wolfSSL_get_session(
    WOLFSSL * ssl
)
```

When NO_SESSION_CACHE_REF is defined this function returns a pointer to the current session (WOLFSSL_SESSION) used in ssl. This function returns a non-persistent pointer to the WOLFSSL_SESSION object. The pointer returned will be freed when wolfSSL_free is called. This call should only be used to inspect or modify the current session. For session resumption it is recommended to use wolfSSL_get1_session(). For backwards compatibility when NO_SESSION_CACHE_REF is not defined this function returns a persistent session object pointer that is stored in the local cache. The cache size is finite and there is a risk that the session object will be overwritten by another ssl connection by the time the application calls wolfSSL_set_session() on it. It is recommended to define NO_SESSION_CACHE_REF in your application and to use wolfSSL_get1_session() for session resumption.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().

See:

- wolfSSL_get1_session
- wolfSSL_set_session

Return:

- pointer If successful the call will return a pointer to the the current SSL session object.
- NULL will be returned if ssl is NULL, the SSL session cache is disabled, wolfSSL doesn't have the Session ID available, or mutex functions fail.

Example

```
WOLFSSL* ssl;
WOLFSSL_SESSION* session;
...
session = wolfSSL_get_session(ssl);
if (session == NULL) {
    // failed to get session pointer
}
...
```

```
void wolfSSL_flush_sessions(
    WOLFSSL_CTX * ctx,
    long tm
)
```

This function flushes session from the session cache which have expired. The time, tm, is used for the time comparison. Note that wolfSSL currently uses a static table for sessions, so no flushing is needed. As such, this function is currently just a stub. This function provides OpenSSL compatibility (SSL_flush_sessions) when wolfSSL is compiled with the OpenSSL compatibility layer.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().
- **tm** time used in session expiration comparison.

See:

- wolfSSL_get1_session
- wolfSSL_set_session

Return: none No returns.

Example

```
WOLFSSL_CTX* ssl;
...
wolfSSL_flush_sessions(ctx, time(0));
```

```
int wolfSSL_GetSessionIndex(
    WOLFSSL * ssl
)
```

This function gets the session index of the WOLFSSL structure.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See: wolfSSL_GetSessionAtIndex

Return: int The function returns an int type representing the sessionIndex within the WOLFSSL struct.

Example

```
WOLFSSL_CTX_new( protocol method );
WOLFSSL* ssl = WOLFSSL_new(ctx);
...
int sesIdx = wolfSSL_GetSessionIndex(ssl);

if(sesIdx < 0 || sesIdx > sizeof(ssl->sessionIndex)/sizeof(int)){
    // You have an out of bounds index number and something is not right.
}
```

```
int wolfSSL_GetSessionAtIndex(
    int idx,
    WOLFSSL_SESSION * session
)
```

This function gets the session at specified index of the session cache and copies it into memory. The WOLFSSL_SESSION structure holds the session information.

Parameters:

- **idx** an int type representing the session index.
- **session** a pointer to the WOLFSSL_SESSION structure.

See:

- UnLockMutex
- LockMutex
- wolfSSL_GetSessionIndex

Return:

- SSL_SUCCESS returned if the function executed successfully and no errors were thrown.
- BAD_MUTEX_E returned if there was an unlock or lock mutex error.
- SSL_FAILURE returned if the function did not execute successfully.

Example

```
int idx; // The index to locate the session.
WOLFSSL_SESSION* session;  // Buffer to copy to.
...
if(wolfSSL_GetSessionAtIndex(idx, session) != SSL_SUCCESS){
    // Failure case.
}
```

```
WOLFSSL_X509_CHAIN * wolfSSL_SESSION_get_peer_chain(
    WOLFSSL_SESSION * session
)
```

Returns the peer certificate chain from the WOLFSSL_SESSION struct.

Parameters:

- **session** a pointer to a WOLFSSL_SESSION structure.

See:

- wolfSSL_GetSessionAtIndex
- wolfSSL_GetSessionIndex
- AddSession

Return: pointer A pointer to a WOLFSSL_X509_CHAIN structure that contains the peer certification chain.

Example

```
WOLFSSL_SESSION* session;
WOLFSSL_X509_CHAIN* chain;
...
chain = wolfSSL_SESSION_get_peer_chain(session);
if(!chain){
    // There was no chain. Failure case.
}
```

```
int wolfSSL_pending(
    WOLFSSL *
)
```

This function returns the number of bytes which are buffered and available in the SSL object to be read by wolfSSL_read().

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().

See:

- wolfSSL_recv
- wolfSSL_read
- wolfSSL_peek

Return: int This function returns the number of bytes pending.

Example

```
int pending = 0;
WOLFSSL* ssl = 0;
...

pending = wolfSSL_pending(ssl);
printf("There are %d bytes buffered and available for reading", pending);
```

```
int wolfSSL_save_session_cache(
    const char * fname
)
```

This function persists the session cache to file. It doesn't use memsave because of additional memory use.

Parameters:

- **fname** is a constant char pointer that points to a file for writing.

See:

- XFWRITE
- wolfSSL_restore_session_cache
- wolfSSL_memrestore_session_cache

Return:

- SSL_SUCCESS returned if the function executed without error. The session cache has been written to a file.
- SSL_BAD_FILE returned if fname cannot be opened or is otherwise corrupt.
- FWRITE_ERROR returned if XFWRITE failed to write to the file.
- BAD_MUTEX_E returned if there was a mutex lock failure.

Example

```
const char* fname;
...
if(wolfSSL_save_session_cache(fname) != SSL_SUCCESS){
    // Fail to write to file.
}
```

```
int wolfSSL_restore_session_cache(
    const char * fname
)
```

This function restores the persistent session cache from file. It does not use memstore because of additional memory use.

Parameters:

- **fname** a constant char pointer file input that will be read.

See:

- XFREAD
- XFOPEN

Return:

- SSL_SUCCESS returned if the function executed without error.
- SSL_BAD_FILE returned if the file passed into the function was corrupted and could not be opened by XFOPEN.
- FREAD_ERROR returned if the file had a read error from XFREAD.
- CACHE_MATCH_ERROR returned if the session cache header match failed.
- BAD_MUTEX_E returned if there was a mutex lock failure.

Example

```
const char *fname;
...
if(wolfSSL_restore_session_cache(fname) != SSL_SUCCESS){
    // Failure case. The function did not return SSL_SUCCESS.
}
```

```
int wolfSSL_memsave_session_cache(
    void * mem,
    int sz
)
```

This function persists session cache to memory.

Parameters:

- **mem** a void pointer representing the destination for the memory copy, XMEMCPY().
- **sz** an int type representing the size of mem.

See:

- XMEMCPY
- wolfSSL_get_session_cache_memsize

Return:

- SSL_SUCCESS returned if the function executed without error. The session cache has been successfully persisted to memory.
- BAD_MUTEX_E returned if there was a mutex lock error.
- BUFFER_E returned if the buffer size was too small.

Example

```
void* mem;
int sz; // Max size of the memory buffer.
…
if(wolfSSL_memsave_session_cache(mem, sz) != SSL_SUCCESS){
    // Failure case, you did not persist the session cache to memory
}
```

```
int wolfSSL_memrestore_session_cache(
    const void * mem,
    int sz
)
```

This function restores the persistent session cache from memory.

Parameters:

- **mem** a constant void pointer containing the source of the restoration.
- **sz** an integer representing the size of the memory buffer.

See: wolfSSL_save_session_cache

Return:

- SSL_SUCCESS returned if the function executed without an error.
- BUFFER_E returned if the memory buffer is too small.
- BAD_MUTEX_E returned if the session cache mutex lock failed.
- CACHE_MATCH_ERROR returned if the session cache header match failed.

Example

```
const void* memoryFile;
int szMf;
...
if(wolfSSL_memrestore_session_cache(memoryFile, szMf) != SSL_SUCCESS){
    // Failure case. SSL_SUCCESS was not returned.
}
```

```
int wolfSSL_get_session_cache_memsize(
    void
)
```

This function returns how large the session cache save buffer should be.

Parameters:

- **none** No parameters.

See: wolfSSL_memrestore_session_cache

Return: int This function returns an integer that represents the size of the session cache save buffer.

Example

```
int sz = // Minimum size for error checking;
...
if(sz < wolfSSL_get_session_cache_memsize()){
    // Memory buffer is too small
}
```

```
int wolfSSL_session_reused(
    WOLFSSL * ssl
)
```

This function returns the resuming member of the options struct. The flag indicates whether or not to reuse a session. If not, a new session must be established.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_SESSION_free
- wolfSSL_GetSessionIndex
- wolfSSL_memsave_session_cache

Return: This function returns an int type held in the Options structure representing the flag for session reuse.

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
…
if(!wolfSSL_session_reused(sslResume)){
    // No session reuse allowed.
}
```

```
const char * wolfSSL_get_version(
    WOLFSSL * ssl
)
```

Returns the SSL version being used as a string.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See: wolfSSL_lib_version

Return:

- "SSLv3" Using SSLv3
- "TLSv1" Using TLSv1
- "TLSv1.1" Using TLSv1.1
- "TLSv1.2" Using TLSv1.2
- "TLSv1.3" Using TLSv1.3
- "DTLS": Using DTLS
- "DTLSv1.2" Using DTLSv1.2
- "unknown" There was a problem determining which version of TLS being used.

Example

```
wolfSSL_Init();
WOLFSSL_CTX* ctx;
WOLFSSL* ssl;
WOLFSSL_METHOD method = // Some wolfSSL method
ctx = wolfSSL_CTX_new(method);
ssl = wolfSSL_new(ctx);
printf(wolfSSL_get_version("Using version: %s", ssl));
```

```
int wolfSSL_get_current_cipher_suite(
    WOLFSSL * ssl
)
```

Returns the current cipher suit an ssl session is using.

Parameters:

- **ssl** The SSL session to check.

See:

- wolfSSL_CIPHER_get_name
- wolfSSL_get_current_cipher
- wolfSSL_get_cipher_list

Return:

- ssl->options.cipherSuite An integer representing the current cipher suite.
- 0 The ssl session provided is null.

Example

```
wolfSSL_Init();
WOLFSSL_CTX* ctx;
WOLFSSL* ssl;
WOLFSSL_METHOD method = // Some wolfSSL method
ctx = wolfSSL_CTX_new(method);
ssl = wolfSSL_new(ctx);

if(wolfSSL_get_current_cipher_suite(ssl) == 0)
{
    // Error getting cipher suite
}
```

```
WOLFSSL_CIPHER * wolfSSL_get_current_cipher(
    WOLFSSL * ssl
)
```

This function returns a pointer to the current cipher in the ssl session.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_get_cipher
- wolfSSL_get_cipher_name_internal
- wolfSSL_get_cipher_name

Return:

- The function returns the address of the cipher member of the WOLFSSL struct. This is a pointer to the WOLFSSL_CIPHER structure.
- NULL returned if the WOLFSSL structure is NULL.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
WOLFSSL* ssl = wolfSSL_new(ctx);
…
WOLFSSL_CIPHER* cipherCurr = wolfSSL_get_current_cipher;

if(!cipherCurr){
    // Failure case.
} else {
    // The cipher was returned to cipherCurr
}
```

```
const char * wolfSSL_CIPHER_get_name(
    const WOLFSSL_CIPHER * cipher
)
```

This function matches the cipher suite in the SSL object with the available suites and returns the string representation.

Parameters:

- **cipher** a constant pointer to a WOLFSSL_CIPHER structure.

See:

- wolfSSL_get_cipher
- wolfSSL_get_current_cipher
- wolfSSL_get_cipher_name_internal
- wolfSSL_get_cipher_name

Return:

- string This function returns the string representation of the matched cipher suite.
- none It will return “None” if there are no suites matched.

Example

```
// gets cipher name in the format DHE_RSA ...
const char* wolfSSL_get_cipher_name_internal(WOLFSSL* ssl){
WOLFSSL_CIPHER* cipher;
const char* fullName;
…
cipher = wolfSSL_get_curent_cipher(ssl);
fullName = wolfSSL_CIPHER_get_name(cipher);
```

```
if(fullName){
    // sanity check on returned cipher
}
```

```
const char * wolfSSL_get_cipher(
    WOLFSSL *
)
```

This function matches the cipher suite in the SSL object with the available suites.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_CIPHER_get_name
- wolfSSL_get_current_cipher

Return: This function returns the string value of the suite matched. It will return “None” if there are no suites matched.

Example

```
#ifdef WOLFSSL_DTLS
…
// make sure a valid suite is used
if(wolfSSL_get_cipher(ssl) == NULL){
    WOLFSSL_MSG(“Can not match cipher suite imported”);
    return MATCH_SUITE_ERROR;
}
…
#endif // WOLFSSL_DTLS
```

```
int wolfSSL_BIO_get_mem_data(
    WOLFSSL_BIO * bio,
    void * p
)
```

This is used to set a byte pointer to the start of the internal memory buffer.

Parameters:

- **bio** WOLFSSL_BIO structure to get memory buffer of.
- **p** byte pointer to set to memory buffer.

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_s_mem
- wolfSSL_BIO_set_fp
- wolfSSL_BIO_free

Return:

- size On success the size of the buffer is returned
- SSL_FATAL_ERROR If an error case was encountered.

Example

```
WOLFSSL_BIO* bio;
const byte* p;
int ret;
bio  = wolfSSL_BIO_new(wolfSSL_BIO_s_mem());
ret  = wolfSSL_BIO_get_mem_data(bio, &p);
// check ret value
```

```
long wolfSSL_BIO_set_fd(
    WOLFSSL_BIO * b,
    int fd,
    int flag
)
```

Sets the file descriptor for bio to use.

Parameters:

- **bio** WOLFSSL_BIO structure to set fd.
- **fd** file descriptor to use.
- **closeF** flag for behavior when closing fd.

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_free

Return: SSL_SUCCESS(1) upon success.

Example

```
WOLFSSL_BIO* bio;
int fd;
// setup bio
wolfSSL_BIO_set_fd(bio, fd, BIO_NOCLOSE);
```

```
int wolfSSL_BIO_set_close(
    WOLFSSL_BIO * b,
    long flag
)
```

Sets the close flag, used to indicate that the i/o stream should be closed when the BIO is freed.

Parameters:

- **bio** WOLFSSL_BIO structure.
- **flag** flag for behavior when closing i/o stream.

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_free

Return: SSL_SUCCESS(1) upon success.

Example

```
WOLFSSL_BIO* bio;
// setup bio
wolfSSL_BIO_set_close(bio, BIO_NOCLOSE);
```

```
WOLFSSL_BIO_METHOD * wolfSSL_BIO_s_socket(
    void
)
```

This is used to get a BIO_SOCKET type WOLFSSL_BIO_METHOD.

Parameters:

- **none** No parameters.

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_s_mem

Return: WOLFSSL_BIO_METHOD pointer to a WOLFSSL_BIO_METHOD structure that is a socket type

Example

```
WOLFSSL_BIO* bio;
bio = wolfSSL_BIO_new(wolfSSL_BIO_s_socket);
```

```
int wolfSSL_BIO_set_write_buf_size(
    WOLFSSL_BIO * b,
    long size
)
```

This is used to set the size of write buffer for a WOLFSSL_BIO. If write buffer has been previously set this function will free it when resetting the size. It is similar to wolfSSL_BIO_reset in that it resets read and write indexes to 0.

Parameters:

- **bio** WOLFSSL_BIO structure to set fd.
- **size** size of buffer to allocate.

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_s_mem
- wolfSSL_BIO_free

Return:

- SSL_SUCCESS On successfully setting the write buffer.
- SSL_FAILURE If an error case was encountered.

Example

```
WOLFSSL_BIO* bio;
int ret;
bio = wolfSSL_BIO_new(wolfSSL_BIO_s_mem());
ret = wolfSSL_BIO_set_write_buf_size(bio, 15000);
// check return value
```

```
int wolfSSL_BIO_make_bio_pair(
    WOLFSSL_BIO * b1,
    WOLFSSL_BIO * b2
)
```

This is used to pair two bios together. A pair of bios acts similar to a two way pipe writing to one can be read by the other and vice versa. It is expected that both bios be in the same thread, this function is not thread safe. Freeing one of the two bios removes both from being paired. If a write buffer size was not previously set for either of the bios it is set to a default size of 17000 (WOLFSSL_BIO_SIZE) before being paired.

Parameters:

- **b1** WOLFSSL_BIO structure to set pair.
- **b2** second WOLFSSL_BIO structure to complete pair.

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_s_mem
- wolfSSL_BIO_free

Return:

- SSL_SUCCESS On successfully pairing the two bios.
- SSL_FAILURE If an error case was encountered.

Example

```
WOLFSSL_BIO* bio;
WOLFSSL_BIO* bio2;
int ret;
bio  = wolfSSL_BIO_new(wolfSSL_BIO_s_bio());
bio2 = wolfSSL_BIO_new(wolfSSL_BIO_s_bio());
ret = wolfSSL_BIO_make_bio_pair(bio, bio2);
// check ret value
```

```
int wolfSSL_BIO_ctrl_reset_read_request(
    WOLFSSL_BIO * bio
)
```

This is used to set the read request flag back to 0.

Parameters:

- **bio** WOLFSSL_BIO structure to set read request flag.

See:

- wolfSSL_BIO_new, wolfSSL_BIO_s_mem
- wolfSSL_BIO_new, wolfSSL_BIO_free

Return:

- SSL_SUCCESS On successfully setting value.
- SSL_FAILURE If an error case was encountered.

Example

```
WOLFSSL_BIO* bio;
int ret;
...
ret = wolfSSL_BIO_ctrl_reset_read_request(bio);
// check ret value
```

```
int wolfSSL_BIO_nread0(
    WOLFSSL_BIO * bio,
    char ** buf
)
```

This is used to get a buffer pointer for reading from. Unlike wolfSSL_BIO_nread the internal read index is not advanced by the number returned from the function call. Reading past the value returned can result in reading out of array bounds.

Parameters:

- **bio** WOLFSSL_BIO structure to read from.
- **buf** pointer to set at beginning of read array.

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_nwrite0

Return: >=0 on success return the number of bytes to read

Example

```
WOLFSSL_BIO* bio;
char* bufPt;
int ret;
// set up bio
ret = wolfSSL_BIO_nread0(bio, &bufPt); // read as many bytes as possible
// handle negative ret check
// read ret bytes from bufPt
```

```
int wolfSSL_BIO_nread(
    WOLFSSL_BIO * bio,
    char ** buf,
    int num
)
```

This is used to get a buffer pointer for reading from. The internal read index is advanced by the number returned from the function call with buf being pointed to the beginning of the buffer to read from. In the case that less bytes are in the read buffer than the value requested with num the lesser value is returned. Reading past the value returned can result in reading out of array bounds.

Parameters:

- **bio** WOLFSSL_BIO structure to read from.
- **buf** pointer to set at beginning of read array.
- **num** number of bytes to try and read.

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_nwrite

Return:

- =0 on success return the number of bytes to read
- WOLFSSL_BIO_ERROR(-1) on error case with nothing to read return -1

Example

```
WOLFSSL_BIO* bio;
char* bufPt;
int ret;

// set up bio
ret = wolfSSL_BIO_nread(bio, &bufPt, 10); // try to read 10 bytes
// handle negative ret check
// read ret bytes from bufPt
```

```
int wolfSSL_BIO_nwrite(
    WOLFSSL_BIO * bio,
    char ** buf,
    int num
)
```

Gets a pointer to the buffer for writing as many bytes as returned by the function. Writing more bytes to the pointer returned then the value returned can result in writing out of bounds.

Parameters:

- **bio** WOLFSSL_BIO structure to write to.
- **buf** pointer to buffer to write to.
- **num** number of bytes desired to be written.

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_free
- wolfSSL_BIO_nread

Return:

- int Returns the number of bytes that can be written to the buffer pointer returned.
- WOLFSSL_BIO_UNSET(-2) in the case that is not part of a bio pair
- WOLFSSL_BIO_ERROR(-1) in the case that there is no more room to write to

Example

```
WOLFSSL_BIO* bio;
char* bufPt;
int ret;
// set up bio
ret = wolfSSL_BIO_nwrite(bio, &bufPt, 10); // try to write 10 bytes
// handle negative ret check
// write ret bytes to bufPt
```

```
int wolfSSL_BIO_reset(
    WOLFSSL_BIO * bio
)
```

Resets bio to an initial state. As an example for type BIO_BIO this resets the read and write index.

Parameters:

- **bio** WOLFSSL_BIO structure to reset.

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_free

Return:

- 0 On successfully resetting the bio.
- WOLFSSL_BIO_ERROR(-1) Returned on bad input or unsuccessful reset.

Example

```
WOLFSSL_BIO* bio;
// setup bio
wolfSSL_BIO_reset(bio);
//use pt
```

```
int wolfSSL_BIO_seek(
    WOLFSSL_BIO * bio,
    int ofs
)
```

This function adjusts the file pointer to the offset given. This is the offset from the head of the file.

Parameters:

- **bio** WOLFSSL_BIO structure to set.

- **ofs** offset into file.

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_s_mem
- wolfSSL_BIO_set_fp
- wolfSSL_BIO_free

Return:

- 0 On successfully seeking.
- -1 If an error case was encountered.

Example

```
WOLFSSL_BIO* bio;
XFILE fp;
int ret;
bio  = wolfSSL_BIO_new(wolfSSL_BIO_s_file());
ret  = wolfSSL_BIO_set_fp(bio, &fp);
// check ret value
ret  = wolfSSL_BIO_seek(bio, 3);
// check ret value
```

```
int wolfSSL_BIO_write_filename(
    WOLFSSL_BIO * bio,
    char * name
)
```

This is used to set and write to a file. WIll overwrite any data currently in the file and is set to close the file when the bio is freed.

Parameters:

- **bio** WOLFSSL_BIO structure to set file.
- **name** name of file to write to.

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_s_file
- wolfSSL_BIO_set_fp
- wolfSSL_BIO_free

Return:

- SSL_SUCCESS On successfully opening and setting file.
- SSL_FAILURE If an error case was encountered.

Example

```
WOLFSSL_BIO* bio;
int ret;
bio  = wolfSSL_BIO_new(wolfSSL_BIO_s_file());
ret  = wolfSSL_BIO_write_filename(bio, “test.txt”);
// check ret value
```

```
long wolfSSL_BIO_set_mem_eof_return(
    WOLFSSL_BIO * bio,
    int v
)
```

This is used to set the end of file value. Common value is -1 so as not to get confused with expected positive values.

Parameters:

- **bio** WOLFSSL_BIO structure to set end of file value.
- **v** value to set in bio.

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_s_mem
- wolfSSL_BIO_set_fp
- wolfSSL_BIO_free

Return: 0 returned on completion

Example

```
WOLFSSL_BIO* bio;
int ret;
bio  = wolfSSL_BIO_new(wolfSSL_BIO_s_mem());
ret  = wolfSSL_BIO_set_mem_eof_return(bio, -1);
// check ret value
```

```
long wolfSSL_BIO_get_mem_ptr(
    WOLFSSL_BIO * bio,
    WOLFSSL_BUF_MEM ** m
)
```

This is a getter function for WOLFSSL_BIO memory pointer.

Parameters:

- **bio** pointer to the WOLFSSL_BIO structure for getting memory pointer.

- **ptr** structure that is currently a char*. Is set to point to bio's memory.

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_s_mem

Return:

- SSL_SUCCESS On successfully getting the pointer SSL_SUCCESS is returned (currently value of 1).
- SSL_FAILURE Returned if NULL arguments are passed in (currently value of 0).

Example

```
WOLFSSL_BIO* bio;
WOLFSSL_BUF_MEM* pt;
// setup bio
wolfSSL_BIO_get_mem_ptr(bio, &pt);
//use pt
```

```
const char * wolfSSL_lib_version(
    void
)
```

This function returns the current library version.

Parameters:

- **none** No parameters.

See: word32_wolfSSL_lib_version_hex

Return: LIBWOLFSSL_VERSION_STRING a const char pointer defining the version.

Example

```
char version[MAXSIZE];
version = wolfSSL_KeepArrays();
…
if(version != ExpectedVersion){
    // Handle the mismatch case
}
```

```
word32 wolfSSL_lib_version_hex(
    void
)
```

This function returns the current library version in hexadecimal notation.

Parameters:

- **none** No parameters.

See: wolfSSL_lib_version

Return: LILBWOLFSSL_VERSION_HEX returns the hexadecimal version defined in wolfssl/version.h.

Example

```
word32 libV;
libV = wolfSSL_lib_version_hex();

if(libV != EXPECTED_HEX){
    // How to handle an unexpected value
} else {
    // The expected result for libV
}
```

```
int wolfSSL_negotiate(
    WOLFSSL * ssl
)
```

Performs the actual connect or accept based on the side of the SSL method. If called from the client side then an wolfSSL_connect() is done while a wolfSSL_accept() is performed if called from the server side.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().

See:

- SSL_connect
- SSL_accept

Return:

- SSL_SUCCESS will be returned if successful. (Note, older versions will return 0.)
- SSL_FATAL_ERROR will be returned if the underlying call resulted in an error. Use wolfSSL_get_error() to get a specific error code.

Example

```
int ret = SSL_FATAL_ERROR;
WOLFSSL* ssl = 0;
...
ret = wolfSSL_negotiate(ssl);
```

```
if (ret == SSL_FATAL_ERROR) {
    // SSL establishment failed
int error_code = wolfSSL_get_error(ssl);
...
}
...
```

```
int wolfSSL_connect_cert(
    WOLFSSL * ssl
)
```

This function is called on the client side and initiates an SSL/TLS handshake with a server only long enough to get the peer's certificate chain. When this function is called, the underlying communication channel has already been set up. wolfSSL_connect_cert() works with both blocking and non-blocking I/O. When the underlying I/O is non-blocking, wolfSSL_connect_cert() will return when the underlying I/O could not satisfy the needs of wolfSSL_connect_cert() to continue the handshake. In this case, a call to wolfSSL_get_error() will yield either SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE. The calling process must then repeat the call to wolfSSL_connect_cert() when the underlying I/O is ready and wolfSSL will pick up where it left off. When using a non_blocking socket, nothing needs to be done, but select() can be used to check for the required condition. If the underlying I/O is blocking, wolfSSL_connect_cert() will only return once the peer's certificate chain has been received.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_get_error
- wolfSSL_connect
- wolfSSL_accept

Return:

- SSL_SUCCESS upon success.
- SSL_FAILURE will be returned if the SSL session parameter is NULL.
- SSL_FATAL_ERROR will be returned if an error occurred. To get a more detailed error code, call wolfSSL_get_error().

Example

```
int ret = 0;
int err = 0;
WOLFSSL* ssl;
char buffer[80];
...
ret = wolfSSL_connect_cert(ssl);
if (ret != SSL_SUCCESS) {
    err = wolfSSL_get_error(ssl, ret);
```

```
    printf("error = %d, %s\n", err, wolfSSL_ERR_error_string(err, buffer));
}



int wolfSSL_writev(
    WOLFSSL * ssl,
    const struct iovec * iov,
    int iovcnt
)
```

Simulates writev semantics but doesn't actually do block at a time because of SSL_write() behavior and because front adds may be small. Makes porting into software that uses writev easier.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().
- **iov** array of I/O vectors to write
- **iovcnt** number of vectors in iov array.

See: wolfSSL_write

Return:

- 0 the number of bytes written upon success.
- 0 will be returned upon failure. Call wolfSSL_get_error() for the specific error code.
- MEMORY_ERROR will be returned if a memory error was encountered.
- SSL_FATAL_ERROR will be returned upon failure when either an error occurred or, when using non-blocking sockets, the SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE error was received and and the application needs to call wolfSSL_write() to get a specific error code.

Example

```
WOLFSSL* ssl = 0;
char *bufA = "hello\n";
char *bufB = "hello world\n";
int iovcnt;
struct iovec iov[2];

iov[0].iov_base = buffA;
iov[0].iov_len = strlen(buffA);
iov[1].iov_base = buffB;
iov[1].iov_len = strlen(buffB);
iovcnt = 2;
...
ret = wolfSSL_writev(ssl, iov, iovcnt);
// wrote "ret" bytes, or error if <= 0.
```

```
unsigned char wolfSSL_SNI_Status(
    WOLFSSL * ssl,
    unsigned char type
)
```

This function gets the status of an SNI object.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **type** the SNI type.

See:

- TLSX_SNI_Status
- TLSX_SNI_find
- TLSX_Find

Return:

- value This function returns the byte value of the SNI struct's status member if the SNI is not NULL.
- 0 if the SNI object is NULL.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
WOLFSSL* ssl = wolfSSL_new(ctx);
…
#define AssertIntEQ(x, y) AssertInt(x, y, ==, !=)
…
Byte type = WOLFSSL_SNI_HOST_NAME;
char* request = (char*)&type;
AssertIntEQ(WOLFSSL_SNI_NO_MATCH, wolfSSL_SNI_Status(ssl, type));
…
```

```
int wolfSSL_UseSecureRenegotiation(
    WOLFSSL * ssl
)
```

This function forces secure renegotiation for the supplied WOLFSSL structure. This is not recommended.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- TLSX_Find

- TLSX_UseSecureRenegotiation

Return:

- SSL_SUCCESS Successfully set secure renegotiation.
- BAD_FUNC_ARG Returns error if ssl is null.
- MEMORY_E Returns error if unable to allocate memory for secure renegotiation.

Example

```
wolfSSL_Init();
WOLFSSL_CTX* ctx;
WOLFSSL* ssl;
WOLFSSL_METHOD method = // Some wolfSSL method
ctx = wolfSSL_CTX_new(method);
ssl = wolfSSL_new(ctx);

if(wolfSSL_UseSecureRenegotiation(ssl) != SSL_SUCCESS)
{
    // Error setting secure renegotiation
}
```

```
int wolfSSL_Rehandshake(
    WOLFSSL * ssl
)
```

This function executes a secure renegotiation handshake; this is user forced as wolfSSL discourages this functionality.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_negotiate
- wc_InitSha512
- wc_InitSha384
- wc_InitSha256
- wc_InitSha
- wc_InitMd5

Return:

- SSL_SUCCESS returned if the function executed without error.
- BAD_FUNC_ARG returned if the WOLFSSL structure was NULL or otherwise if an unacceptable argument was passed in a subroutine.
- SECURE_RENEGOTIATION_E returned if there was an error with renegotiating the handshake.
- SSL_FATAL_ERROR returned if there was an error with the server or client configuration and the renegotiation could not be completed. See wolfSSL_negotiate().

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
...
if(wolfSSL_Rehandshake(ssl) != SSL_SUCCESS){
    // There was an error and the rehandshake is not successful.
}
```

```
int wolfSSL_UseSessionTicket(
    WOLFSSL * ssl
)
```

Force provided WOLFSSL structure to use session ticket. The constant HAVE_SESSION_TICKET should be defined and the constant NO_WOLFSSL_CLIENT should not be defined to use this function.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See: TLSX_UseSessionTicket

Return:

- SSL_SUCCESS Successfully set use session ticket.
- BAD_FUNC_ARG Returned if ssl is null.
- MEMORY_E Error allocating memory for setting session ticket.

Example

```
wolfSSL_Init();
WOLFSSL_CTX* ctx;
WOLFSSL* ssl;
WOLFSSL_METHOD method = // Some wolfSSL method
ctx = wolfSSL_CTX_new(method);
ssl = wolfSSL_new(ctx);

if(wolfSSL_UseSessionTicket(ssl) != SSL_SUCCESS)
{
    // Error setting session ticket
}
```

```
int wolfSSL_get_SessionTicket(
    WOLFSSL * ssl,
    unsigned char * buf,
    word32 * bufSz
)
```

This function copies the ticket member of the Session structure to the buffer.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **buf** a byte pointer representing the memory buffer.
- **bufSz** a word32 pointer representing the buffer size.

See:

- wolfSSL_UseSessionTicket
- wolfSSL_set_SessionTicket

Return:

- SSL_SUCCESS returned if the function executed without error.
- BAD_FUNC_ARG returned if one of the arguments was NULL or if the bufSz argument was 0.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
WOLFSSL* ssl = wolfSSL_new(ctx);
byte* buf;
word32 bufSz;  // Initialize with buf size
…
if(wolfSSL_get_SessionTicket(ssl, buf, bufSz) <= 0){
    // Nothing was written to the buffer
} else {
    // the buffer holds the content from ssl->session->ticket
}
```

```
int wolfSSL_set_SessionTicket(
    WOLFSSL * ssl,
    const unsigned char * buf,
    word32 bufSz
)
```

This function sets the ticket member of the WOLFSSL_SESSION structure within the WOLFSSL struct. The buffer passed into the function is copied to memory.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **buf** a byte pointer that gets loaded into the ticket member of the session structure.
- **bufSz** a word32 type that represents the size of the buffer.

See: wolfSSL_set_SessionTicket_cb

Return:

- SSL_SUCCESS returned on successful execution of the function. The function returned without errors.
- BAD_FUNC_ARG returned if the WOLFSSL structure is NULL. This will also be thrown if the buf argument is NULL but the bufSz argument is not zero.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
WOLFSSL* ssl = wolfSSL_new(ctx);
byte* buffer; // File to load
word32 bufSz;
...
if(wolfSSL_KeepArrays(ssl, buffer, bufSz) != SSL_SUCCESS){
    // There was an error loading the buffer to memory.
}
```

```
int wolfSSL_PrintSessionStats(
    void
)
```

This function prints the statistics from the session.

Parameters:

- **none** No parameters.

See: wolfSSL_get_session_stats

Return:

- SSL_SUCCESS returned if the function and subroutines return without error. The session stats have been successfully retrieved and printed.
- BAD_FUNC_ARG returned if the subroutine wolfSSL_get_session_stats() was passed an unacceptable argument.
- BAD_MUTEX_E returned if there was a mutex error in the subroutine.

Example

```
// You will need to have a session object to retrieve stats from.
if(wolfSSL_PrintSessionStats(void) != SSL_SUCCESS   ){
    // Did not print session stats
}
```

```
int wolfSSL_get_session_stats(
    unsigned int * active,
    unsigned int * total,
    unsigned int * peak,
    unsigned int * maxSessions
)
```

This function gets the statistics for the session.

Parameters:

- **active** a word32 pointer representing the total current sessions.
- **total** a word32 pointer representing the total sessions.
- **peak** a word32 pointer representing the peak sessions.
- **maxSessions** a word32 pointer representing the maximum sessions.

See: wolfSSL_PrintSessionStats

Return:

- SSL_SUCCESS returned if the function and subroutines return without error. The session stats have been successfully retrieved and printed.
- BAD_FUNC_ARG returned if the subroutine wolfSSL_get_session_stats() was passed an unacceptable argument.
- BAD_MUTEX_E returned if there was a mutex error in the subroutine.

Example

```
int wolfSSL_PrintSessionStats(void){
…
ret = wolfSSL_get_session_stats(&totalSessionsNow,
&totalSessionsSeen, &peak, &maxSessions);
…
return ret;
```

```
long wolfSSL_BIO_set_fp(
    WOLFSSL_BIO * bio,
    XFILE fp,
    int c
)
```

This is used to set the internal file pointer for a BIO.

Parameters:

- **bio** WOLFSSL_BIO structure to set pair.
- **fp** file pointer to set in bio.
- **c** close file behavior flag.

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_s_mem
- wolfSSL_BIO_get_fp
- wolfSSL_BIO_free

Return:

- SSL_SUCCESS On successfully setting file pointer.
- SSL_FAILURE If an error case was encountered.

Example

```
WOLFSSL_BIO* bio;
XFILE fp;
int ret;
bio  = wolfSSL_BIO_new(wolfSSL_BIO_s_file());
ret  = wolfSSL_BIO_set_fp(bio, fp, BIO_CLOSE);
// check ret value
```

```
long wolfSSL_BIO_get_fp(
    WOLFSSL_BIO * bio,
    XFILE * fp
)
```

This is used to get the internal file pointer for a BIO.

Parameters:

- **bio** WOLFSSL_BIO structure to set pair.
- **fp** file pointer to set in bio.

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_s_mem
- wolfSSL_BIO_set_fp
- wolfSSL_BIO_free

Return:

- SSL_SUCCESS On successfully getting file pointer.
- SSL_FAILURE If an error case was encountered.

Example

```
WOLFSSL_BIO* bio;
XFILE fp;
int ret;
bio  = wolfSSL_BIO_new(wolfSSL_BIO_s_file());
ret  = wolfSSL_BIO_get_fp(bio, &fp);
// check ret value
```

```
size_t wolfSSL_BIO_ctrl_pending(
    WOLFSSL_BIO * b
)
```

Gets the number of pending bytes to read. If BIO type is BIO_BIO then is the number to read from pair. If BIO contains an SSL object then is pending data from SSL object (wolfSSL_pending(ssl)). If is BIO_MEMORY type then returns the size of memory buffer.

Parameters:

- **bio** pointer to the WOLFSSL_BIO structure that has already been created.

See:

- wolfSSL_BIO_make_bio_pair
- wolfSSL_BIO_new

Return: >=0 number of pending bytes.

Example

```
WOLFSSL_BIO* bio;
int pending;
bio = wolfSSL_BIO_new();
…
pending = wolfSSL_BIO_ctrl_pending(bio);
```

```
int wolfSSL_set_jobject(
    WOLFSSL * ssl,
    void * objPtr
)
```

This function sets the jObjectRef member of the WOLFSSL structure.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **objPtr** a void pointer that will be set to jObjectRef.

See: wolfSSL_get_jobject

Return:

- SSL_SUCCESS returned if jObjectRef is properly set to objPtr.
- SSL_FAILURE returned if the function did not properly execute and jObjectRef is not set.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
WOLFSSL* ssl = WOLFSSL_new();
void* objPtr = &obj;
...
if(wolfSSL_set_jobject(ssl, objPtr)){
    // The success case
}
```

```
void * wolfSSL_get_jobject(
    WOLFSSL * ssl
)
```

This function returns the jObjectRef member of the WOLFSSL structure.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See: wolfSSL_set_jobject

Return:

- value If the WOLFSSL struct is not NULL, the function returns the jObjectRef value.
- NULL returned if the WOLFSSL struct is NULL.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
WOLFSSL* ssl = wolfSSL(ctx);
...
void* jobject = wolfSSL_get_jobject(ssl);

if(jobject != NULL){
    // Success case
}
```

```
int wolfSSL_connect(
    WOLFSSL * ssl
)
```

This function is called on the client side and initiates an SSL/TLS handshake with a server. When this function is called, the underlying communication channel has already been set up. wolfSSL_connect() works with both blocking and non-blocking I/O. When the underlying I/O is non-blocking, wolfSSL_connect() will return when the underlying I/O could not satisfy the needs of wolfSSL_connect to continue the handshake. In this case, a call to wolfSSL_get_error() will yield either SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE. The calling process must then repeat the call to wolfSSL_connect() when the underlying I/O is ready and wolfSSL will pick up where it left off. When using a non_blocking socket, nothing needs to be done, but select() can be used to check for the required condition. If the underlying I/O is blocking, wolfSSL_connect() will only return once the

handshake has been finished or an error occurred. wolfSSL takes a different approach to certificate verification than OpenSSL does. The default policy for the client is to verify the server, this means that if you don't load CAs to verify the server you'll get a connect error, unable to verify (_155). It you want to mimic OpenSSL behavior of having SSL_connect succeed even if verifying the server fails and reducing security you can do this by calling: SSL_CTX_set_verify(ctx, SSL_VERIFY_NONE, 0); before calling SSL_new(); Though it's not recommended.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_get_error
- wolfSSL_accept

Return:

- SSL_SUCCESS If successful.
- SSL_FATAL_ERROR will be returned if an error occurred. To get a more detailed error code, call wolfSSL_get_error().

Example

```
int ret = 0;
int err = 0;
WOLFSSL* ssl;
char buffer[80];
...
ret = wolfSSL_connect(ssl);
if (ret != SSL_SUCCESS) {
err = wolfSSL_get_error(ssl, ret);
printf("error = %d, %s\n", err, wolfSSL_ERR_error_string(err, buffer));
}
```

```
int wolfSSL_update_keys(
    WOLFSSL * ssl
)
```

This function is called on a TLS v1.3 client or server wolfSSL to force the rollover of keys. A KeyUpdate message is sent to the peer and new keys are calculated for encryption. The peer will send back a KeyUpdate message and the new decryption keys will then be calculated. This function can only be called after a handshake has been completed.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See: wolfSSL_write

Return:

- BAD_FUNC_ARG if ssl is NULL or not using TLS v1.3.
- WANT_WRITE if the writing is not ready.
- WOLFSSL_SUCCESS if successful.

Example

```
int ret;
WOLFSSL* ssl;
...
ret = wolfSSL_update_keys(ssl);
if (ret == WANT_WRITE) {
    // need to call again when I/O ready
}
else if (ret != WOLFSSL_SUCCESS) {
    // failed to send key update
}
```

```
int wolfSSL_key_update_response(
    WOLFSSL * ssl,
    int * required
)
```

This function is called on a TLS v1.3 client or server wolfSSL to determine whether a rollover of keys is in progress. When wolfSSL_update_keys() is called, a KeyUpdate message is sent and the encryption key is updated. The decryption key is updated when the response is received.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **required** 0 when no key update response required. 1 when no key update response required.

See: wolfSSL_update_keys

Return:

- 0 on successful.
- BAD_FUNC_ARG if ssl is NULL or not using TLS v1.3.

Example

```
int ret;
WOLFSSL* ssl;
int required;
...
ret = wolfSSL_key_update_response(ssl, &required);
if (ret != 0) {
    // bad parameters
}
if (required) {
```

```
    // encrypt Key updated, awaiting response to change decrypt key
}
```

```
int wolfSSL_request_certificate(
    WOLFSSL * ssl
)
```

This function requests a client certificate from the TLS v1.3 client. This is useful when a web server is serving some pages that require client authentication and others that don't. A maximum of 256 requests can be sent on a connection.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_allow_post_handshake_auth
- wolfSSL_write

Return:

- BAD_FUNC_ARG if ssl is NULL or not using TLS v1.3.
- WANT_WRITE if the writing is not ready.
- SIDE_ERROR if called with a client.
- NOT_READY_ERROR if called when the handshake is not finished.
- POST_HAND_AUTH_ERROR if posthandshake authentication is disallowed.
- MEMORY_E if dynamic memory allocation fails.
- WOLFSSL_SUCCESS if successful.

Example

```
int ret;
WOLFSSL* ssl;
...
ret = wolfSSL_request_certificate(ssl);
if (ret == WANT_WRITE) {
    // need to call again when I/O ready
}
else if (ret != WOLFSSL_SUCCESS) {
    // failed to request a client certificate
}
```

```
int wolfSSL_connect_TLSv13(
    WOLFSSL *
)
```

This function is called on the client side and initiates a TLS v1.3 handshake with a server. When this function is called, the underlying communication channel has already been set up. wolfSSL_connect() will only return once the handshake has been finished or an error occurred. wolfSSL takes a different approach to certificate verification than OpenSSL does. The default policy for the client is to verify the server, this means that if you don't load CAs to verify the server you'll get a connect error, unable to verify (_155). It you want to mimic OpenSSL behavior of having SSL_connect succeed even if verifying the server fails and reducing security you can do this by calling: SSL_CTX_set_verify(ctx, SSL_VERIFY_NONE, 0); before calling SSL_new(); Though it's not recommended.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_get_error
- wolfSSL_connect
- wolfSSL_accept_TLSv13
- wolfSSL_accept

Return:

- SSL_SUCCESS upon success.
- SSL_FATAL_ERROR will be returned if an error occurred. To get a more detailed error code, call wolfSSL_get_error().

Example

```
int ret = 0;
int err = 0;
WOLFSSL* ssl;
char buffer[80];
...

ret = wolfSSL_connect_TLSv13(ssl);
if (ret != SSL_SUCCESS) {
    err = wolfSSL_get_error(ssl, ret);
    printf("error = %d, %s\n", err, wolfSSL_ERR_error_string(err, buffer));
}
```

```
wolfSSL_accept_TLSv13(
    WOLFSSL * ssl
)
```

This function is called on the server side and waits for a SSL/TLS client to initiate the SSL/TLS handshake. When this function is called, the underlying communication channel has already been set up. wolfSSL_accept() will only return once the handshake has been finished or an error occurred. Call this function when expecting a TLS v1.3 connection though older version ClientHello messages are supported.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_get_error
- wolfSSL_connect_TLSv13
- wolfSSL_connect
- wolfSSL_accept_TLSv13
- wolfSSL_accept

Return:

- SSL_SUCCESS upon success.
- SSL_FATAL_ERROR will be returned if an error occurred. To get a more detailed error code, call wolfSSL_get_error().

Example

```
int ret = 0;
int err = 0;
WOLFSSL* ssl;
char buffer[80];
...

ret = wolfSSL_accept_TLSv13(ssl);
if (ret != SSL_SUCCESS) {
    err = wolfSSL_get_error(ssl, ret);
    printf("error = %d, %s\n", err, wolfSSL_ERR_error_string(err, buffer));
}
```

```
int wolfSSL_write_early_data(
    WOLFSSL * ssl,
    const void * data,
    int sz,
    int * outSz
)
```

This function writes early data to the server on resumption. Call this function instead of wolfSSL_connect() to connect to the server and send the data in the handshake. This function is only used with clients.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **data** the buffer holding the early data to write to server.
- **sz** the amount of early data to write in bytes.
- **outSz** the amount of early data written in bytes.

See:

- wolfSSL_read_early_data
- wolfSSL_connect
- wolfSSL_connect_TLSv13

Return:

- BAD_FUNC_ARG if a pointer parameter is NULL, sz is less than 0 or not using TLSv1.3.
- SIDE_ERROR if called with a server.
- WOLFSSL_FATAL_ERROR if the connection is not made.
- WOLFSSL_SUCCESS if successful.

Example

```
int ret = 0;
int err = 0;
WOLFSSL* ssl;
byte earlyData[] = { early data };
int outSz;
char buffer[80];
...

ret = wolfSSL_write_early_data(ssl, earlyData, sizeof(earlyData), &outSz);
if (ret != WOLFSSL_SUCCESS) {
    err = wolfSSL_get_error(ssl, ret);
    printf("error = %d, %s\n", err, wolfSSL_ERR_error_string(err, buffer));
    goto err_label;
}
if (outSz < sizeof(earlyData)) {
    // not all early data was sent
}
ret = wolfSSL_connect_TLSv13(ssl);
if (ret != SSL_SUCCESS) {
    err = wolfSSL_get_error(ssl, ret);
    printf("error = %d, %s\n", err, wolfSSL_ERR_error_string(err, buffer));
}
```

```
int wolfSSL_read_early_data(
    WOLFSSL * ssl,
    void * data,
    int sz,
    int * outSz
)
```

This function reads any early data from a client on resumption. Call this function instead of wolfSSL_accept() to accept a client and read any early data in the handshake. The function should be invoked until wolfSSL_is_init_finished() returns true. Early data may be sent by the client in multiple messages. If there is no early data then the handshake will be processed as normal. This function is only used with servers.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **data** a buffer to hold the early data read from client.
- **sz** size of the buffer in bytes.
- **outSz** number of bytes of early data read.

See:

- wolfSSL_write_early_data
- wolfSSL_accept
- wolfSSL_accept_TLSv13

Return:

- BAD_FUNC_ARG if a pointer parameter is NULL, sz is less than 0 or not using TLSv1.3.
- SIDE_ERROR if called with a client.
- WOLFSSL_FATAL_ERROR if accepting a connection fails.
- Number of early data bytes read (may be zero).

Example

```
int ret = 0;
int err = 0;
WOLFSSL* ssl;
byte earlyData[128];
int outSz;
char buffer[80];
...

do {
    ret = wolfSSL_read_early_data(ssl, earlyData, sizeof(earlyData), &outSz);
    if (ret < 0) {
        err = wolfSSL_get_error(ssl, ret);
        printf("error = %d, %s\n", err, wolfSSL_ERR_error_string(err, buffer));
    }
    if (outSz > 0) {
        // early data available
    }
} while (!wolfSSL_is_init_finished(ssl));
```

```
int wolfSSL_inject(
    WOLFSSL * ssl,
    const void * data,
    int sz
)
```

This function is called to inject data into the WOLFSSL object. This is useful when data needs to be read from a single place and demultiplexed into multiple connections. The caller should then call wolfSSL_read() to extract the plaintext data from the WOLFSSL object.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **data** data to inject into the ssl object.
- **sz** number of bytes of data to inject.

See: wolfSSL_read

Return:

- BAD_FUNC_ARG if any pointer parameter is NULL or sz <= 0
- APP_DATA_READY if there is application data left to read
- MEMORY_E if allocation fails
- WOLFSSL_SUCCESS on success

Example

```
byte buf[2000]
sz = recv(fd, buf, sizeof(buf), 0);
if (sz <= 0)
    // error
if (wolfSSL_inject(ssl, buf, sz) != WOLFSSL_SUCCESS)
    // error
sz = wolfSSL_read(ssl, buf, sizeof(buf);
```

```
void * wolfSSL_GetIOReadCtx(
    WOLFSSL * ssl
)
```

This function returns the IOCB_ReadCtx member of the WOLFSSL struct.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_GetIOWriteCtx
- wolfSSL_SetIOReadFlags
- wolfSSL_SetIOWriteCtx
- wolfSSL_SetIOReadCtx
- wolfSSL_CTX_SetIOSend

Return:

- pointer This function returns a void pointer to the IOCB_ReadCtx member of the WOLFSSL structure.
- NULL returned if the WOLFSSL struct is NULL.

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
void* ioRead;
...
ioRead = wolfSSL_GetIOReadCtx(ssl);
if(ioRead == NULL){
    // Failure case. The ssl object was NULL.
}
```

```
void * wolfSSL_GetIOWriteCtx(
    WOLFSSL * ssl
)
```

This function returns the IOCB_WriteCtx member of the WOLFSSL structure.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_GetIOReadCtx
- wolfSSL_SetIOWriteCtx
- wolfSSL_SetIOReadCtx
- wolfSSL_CTX_SetIOSend

Return:

- pointer This function returns a void pointer to the IOCB_WriteCtx member of the WOLFSSL structure.
- NULL returned if the WOLFSSL struct is NULL.

Example

```
WOLFSSL* ssl;
void* ioWrite;
...
ioWrite = wolfSSL_GetIOWriteCtx(ssl);
if(ioWrite == NULL){
    // The function returned NULL.
}
```

```
void wolfSSL_SetIO_NetX(
    WOLFSSL * ssl,
    NX_TCP_SOCKET * nxsocket,
    ULONG waitoption
)
```

This function sets the nxSocket and nxWait members of the nxCtx struct within the WOLFSSL structure.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **nxSocket** a pointer to type NX_TCP_SOCKET that is set to the nxSocket member of the nxCTX structure.
- **waitOption** a ULONG type that is set to the nxWait member of the nxCtx structure.

See:

- set_fd
- NetX_Send
- NetX_Receive

Return: none No returns.

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
NX_TCP_SOCKET* nxSocket;
ULONG waitOption;
…
if(ssl != NULL || nxSocket != NULL || waitOption <= 0){
wolfSSL_SetIO_NetX(ssl, nxSocket, waitOption);
} else {
    // You need to pass in good parameters.
}
```

A.6 wolfSSL Context and Session Set Up

A.5.2.77 function wolfSSL_SetIO_NetX

A.6.1 Functions

	Name
WOLFSSL_METHOD *	**wolfSSLv23_method**(void)This function returns a WOLFSSL_METHOD similar to wolfSSLv23_client_method except that it is not determined which side yet (server/client).
WOLFSSL_METHOD *	**wolfSSLv3_server_method**(void)The wolfSSLv3_server_method() function is used to indicate that the application is a server and will only support the SSL 3.0 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().

	Name
WOLFSSL_METHOD *	**wolfSSLv3_client_method**(void)The wolfSSLv3_client_method() function is used to indicate that the application is a client and will only support the SSL 3.0 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().
WOLFSSL_METHOD *	**wolfTLSv1_server_method**(void)The wolfTLSv1_server_method() function is used to indicate that the application is a server and will only support the TLS 1.0 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().
WOLFSSL_METHOD *	**wolfTLSv1_client_method**(void)The wolfTLSv1_client_method() function is used to indicate that the application is a client and will only support the TLS 1.0 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().
WOLFSSL_METHOD *	**wolfTLSv1_1_server_method**(void)The wolfTLSv1_1_server_method() function is used to indicate that the application is a server and will only support the TLS 1.1 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().
WOLFSSL_METHOD *	**wolfTLSv1_1_client_method**(void)The wolfTLSv1_1_client_method() function is used to indicate that the application is a client and will only support the TLS 1.0 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().
WOLFSSL_METHOD *	**wolfTLSv1_2_server_method**(void)The wolfTLSv1_2_server_method() function is used to indicate that the application is a server and will only support the TLS 1.2 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().

	Name
WOLFSSL_METHOD *	**wolfTLSv1_2_client_method**(void)The wolfTLSv1_2_client_method() function is used to indicate that the application is a client and will only support the TLS 1.2 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().
WOLFSSL_METHOD *	**wolfDTLSv1_client_method**(void)The wolfDTLSv1_client_method() function is used to indicate that the application is a client and will only support the DTLS 1.0 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new(). This function is only available when wolfSSL has been compiled with DTLS support (–enable_dtls, or by defining wolfSSL_DTLS).
WOLFSSL_METHOD *	**wolfDTLSv1_server_method**(void)The wolfDTLSv1_server_method() function is used to indicate that the application is a server and will only support the DTLS 1.0 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new(). This function is only available when wolfSSL has been compiled with DTLS support (–enable_dtls, or by defining wolfSSL_DTLS).
WOLFSSL_METHOD *	**wolfDTLSv1_3_server_method**(void)The wolfDTLSv1_3_server_method() function is used to indicate that the application is a server and will only support the DTLS 1.3 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new(). This function is only available when wolfSSL has been compiled with DTLSv1.3 support (–enable_dtls13, or by defining wolfSSL_DTLS13).
WOLFSSL_METHOD *	**wolfDTLSv1_3_client_method**(void)The wolfDTLSv1_3_client_method() function is used to indicate that the application is a client and will only support the DTLS 1.3 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new(). This function is only available when wolfSSL has been compiled with DTLSv1.3 support (–enable_dtls13, or by defining wolfSSL_DTLS13).

	Name
WOLFSSL_METHOD *	**wolfDTLS_server_method**(void)The wolfDTLS_server_method() function is used to indicate that the application is a server and will support the highest version of DTLS available and all the version up to the minimum version allowed. The default minimum version allowed is based on the define WOLFSSL_MIN_DTLS_DOWNGRADE and can be changed at runtime using wolfSSL_SetMinVersion(). This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new(). This function is only available when wolfSSL has been compiled with DTLS support (–enable_dtls, or by defining wolfSSL_DTLS).
WOLFSSL_METHOD *	**wolfDTLS_client_method**(void)The wolfDTLS_client_method() function is used to indicate that the application is a client and will support the highest version of DTLS available and all the version up to the minimum version allowed. The default minimum version allowed is based on the define WOLFSSL_MIN_DTLS_DOWNGRADE and can be changed at runtime using wolfSSL_SetMinVersion(). This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new(). This function is only available when wolfSSL has been compiled with DTLS support (–enable_dtls, or by defining wolfSSL_DTLS).
int	**wolfSSL_use_old_poly**(WOLFSSL * ssl, int value)Since there is some differences between the first release and newer versions of chacha-poly AEAD construction we have added an option to communicate with servers/clients using the older version. By default wolfSSL uses the new version.
int	**wolfSSL_CTX_trust_peer_cert**(WOLFSSL_CTX * ctx, const char * file, int type)This function loads a certificate to use for verifying a peer when performing a TLS/SSL handshake. The peer certificate sent during the handshake is compared by using the SKID when available and the signature. If these two things do not match then any loaded CAs are used. Feature is enabled by defining the macro WOLFSSL_TRUST_PEER_CERT. Please see the examples for proper usage.

	Name
long	**wolfSSL_CTX_get_verify_depth**(WOLFSSL_CTX * ctx)This function gets the certificate chaining depth using the CTX structure.
WOLFSSL_CTX *	**wolfSSL_CTX_new**(WOLFSSL_METHOD *)This function creates a new SSL context, taking a desired SSL/TLS protocol method for input.
WOLFSSL *	**wolfSSL_new**(WOLFSSL_CTX *)This function creates a new SSL session, taking an already created SSL context as input.
int	**wolfSSL_set_fd**(WOLFSSL * ssl, int fd)This function assigns a file descriptor (fd) as the input/output facility for the SSL connection. Typically this will be a socket file descriptor.
int	**wolfSSL_set_dtls_fd_connected**(WOLFSSL * ssl, int fd)This function assigns a file descriptor (fd) as the input/output facility for the SSL connection. Typically this will be a socket file descriptor. This is a DTLS specific API because it marks that the socket is connected. recvfrom and sendto calls on this fd will have the addr and addr_len parameters set to NULL.
int	**wolfDTLS_SetChGoodCb**(WOLFSSL * ssl, ClientHelloGoodCb cb, void * user_ctx)Allows setting a callback for a correctly processed and verified DTLS client hello. When using a cookie exchange mechanism (either the HelloVerifyRequest in DTLS 1.2 or the HelloRetryRequest with a cookie extension in DTLS 1.3) this callback is called after the cookie exchange has succeeded. This is useful to use one WOLFSSL object as the listener for new connections and being able to isolate the WOLFSSL object once the ClientHello is verified (either through a cookie exchange or just checking if the ClientHello had the correct format). DTLS 1.2: https://datatracker.ietf.org/doc/html/rfc6347#section_4.2.1 DTLS 1.3: https://www.rfc_editor.org/rfc/rfc8446#section_4.2.2.
void	**wolfSSL_set_using_nonblock**(WOLFSSL * ssl, int nonblock)This function informs the WOLFSSL object that the underlying I/O is non_blocking. After an application creates a WOLFSSL object, if it will be used with a non_blocking socket, call wolfSSL_set_using_nonblock() on it. This lets the WOLFSSL object know that receiving EWOULDBLOCK means that the recvfrom call would block rather than that it timed out.

	Name
void	**wolfSSL_CTX_free**(WOLFSSL_CTX *)This function frees an allocated WOLFSSL_CTX object. This function decrements the CTX reference count and only frees the context when the reference count has reached 0.
void	**wolfSSL_free**(WOLFSSL *)This function frees an allocated wolfSSL object.
int	**wolfSSL_set_session and wolfSSL will try to resume the session. The wolfSSL server code allows session resumption by default. The object returned by wolfSSL_get1_session() needs to be freed after the application is done with it by calling wolfSSL_SESSION_free() on it.
void	**wolfSSL_CTX_set_verify**(WOLFSSL_CTX * ctx, int mode, VerifyCallback verify_callback)This function sets the verification method for remote peers and also allows a verify callback to be registered with the SSL context. The verify callback will be called only when a verification failure has occurred. If no verify callback is desired, the NULL pointer can be used for verify_callback. The verification mode of peer certificates is a logically OR'd list of flags. The possible flag values include: SSL_VERIFY_NONE Client mode: the client will not verify the certificate received from the server and the handshake will continue as normal. Server mode: the server will not send a certificate request to the client. As such, client verification will not be enabled. SSL_VERIFY_PEER Client mode: the client will verify the certificate received from the server during the handshake. This is turned on by default in wolfSSL, therefore, using this option has no effect. Server mode: the server will send a certificate request to the client and verify the client certificate received. SSL_VERIFY_FAIL_IF_NO_PEER_CERT Client mode: no effect when used on the client side. Server mode: the verification will fail on the server side if the client fails to send a certificate when requested to do so (when using SSL_VERIFY_PEER on the SSL server). SSL_VERIFY_FAIL_EXCEPT_PSK Client mode: no effect when used on the client side. Server mode: the verification is the same as SSL_VERIFY_FAIL_IF_NO_PEER_CERT except in the case of a PSK connection. If a PSK connection is being made then the connection will go through without a peer cert.

	Name
void	**wolfSSL_set_verify**(WOLFSSL * ssl, int mode, VerifyCallback verify_callback)This function sets the verification method for remote peers and also allows a verify callback to be registered with the SSL session. The verify callback will be called only when a verification failure has occurred. If no verify callback is desired, the NULL pointer can be used for verify_callback. The verification mode of peer certificates is a logically OR'd list of flags. The possible flag values include: SSL_VERIFY_NONE Client mode: the client will not verify the certificate received from the server and the handshake will continue as normal. Server mode: the server will not send a certificate request to the client. As such, client verification will not be enabled. SSL_VERIFY_PEER Client mode: the client will verify the certificate received from the server during the handshake. This is turned on by default in wolfSSL, therefore, using this option has no effect. Server mode: the server will send a certificate request to the client and verify the client certificate received. SSL_VERIFY_FAIL_IF_NO_PEER_CERT Client mode: no effect when used on the client side. Server mode: the verification will fail on the server side if the client fails to send a certificate when requested to do so (when using SSL_VERIFY_PEER on the SSL server). SSL_VERIFY_FAIL_EXCEPT_PSK Client mode: no effect when used on the client side. Server mode: the verification is the same as SSL_VERIFY_FAIL_IF_NO_PEER_CERT except in the case of a PSK connection. If a PSK connection is being made then the connection will go through without a peer cert.
long	**wolfSSL_CTX_set_session_cache_mode**(WOLFSSL_CTX * ctx, long mode)This function enables or disables SSL session caching. Behavior depends on the value used for mode. The following values for mode are available: SSL_SESS_CACHE_OFF- disable session caching. Session caching is turned on by default. SSL_SESS_CACHE_NO_AUTO_CLEAR - Disable auto-flushing of the session cache. Auto-flushing is turned on by default.
int	**wolfSSL_CTX_memrestore_cert_cache**(WOLFSSL_CTX * ctx, const void * mem, int sz)This function restores the certificate cache from memory.

	Name
int	**wolfSSL_CTX_set_cipher_list**(WOLFSSL_CTX * ctx, const char * list)This function sets cipher suite list for a given WOLFSSL_CTX. This cipher suite list becomes the default list for any new SSL sessions (WOLFSSL) created using this context. The ciphers in the list should be sorted in order of preference from highest to lowest. Each call to wolfSSL_CTX_set_cipher_list() resets the cipher suite list for the specific SSL context to the provided list each time the function is called. The cipher suite list, list, is a null_terminated text string, and a colon_delimited list. For example, one value for list may be “DHE_RSA_AES256_SHA256:DHE_RSA_AES128_SHA256:AES256_SH Valid cipher values are the full name values from the cipher_names[] array in src/internal.c (for a definite list of valid cipher values check src/internal.c)
int	**wolfSSL_set_cipher_list**(WOLFSSL * ssl, const char * list)This function sets cipher suite list for a given WOLFSSL object (SSL session). The ciphers in the list should be sorted in order of preference from highest to lowest. Each call to wolfSSL_set_cipher_list() resets the cipher suite list for the specific SSL session to the provided list each time the function is called. The cipher suite list, list, is a null_terminated text string, and a colon_delimited list. For example, one value for list may be “DHE_RSA_AES256_SHA256:DHE_RSA_AES128_SHA256:AES256_SH Valid cipher values are the full name values from the cipher_names[] array in src/internal.c (for a definite list of valid cipher values check src/internal.c)
void	**wolfSSL_dtls13_set_send_more_acks**(WOLFSSL * ssl, int value)This function sets whether the library should send ACKs to the other peer immediately when detecting disruption or not. Sending ACKs immediately assures minimum latency but it may consume more bandwidth than necessary. If the application manages the timer by itself and this option is set to 0 then application code can use wolfSSL_dtls13_use_quick_timeout() to determine if it should setup a quicker timeout to send those delayed ACKs.
int	**wolfSSL_dtls_set_timeout_init**(WOLFSSL * ssl, int)This function sets the dtls timeout.

	Name
WOLFSSL_SESSION *	**wolfSSL_get1_session and wolfSSL will try to resume the session. The wolfSSL server code allows session resumption by default. The object returned by wolfSSL_get1_session() needs to be freed after the application is done with it by calling wolfSSL_SESSION_free() on it.
WOLFSSL_METHOD *	**wolfSSLv23_client_method. Both wolfSSL clients and servers have robust version downgrade capability. If a specific protocol version method is used on either side, then only that version will be negotiated or an error will be returned. For example, a client that uses TLSv1 and tries to connect to a SSLv3 only server will fail, likewise connecting to a TLSv1.1 will fail as well. To resolve this issue, a client that uses the wolfSSLv23_client_method() function will use the highest protocol version supported by the server and downgrade to SSLv3 if needed. In this case, the client will be able to connect to a server running SSLv3 - TLSv1.3.
WOLFSSL_BIGNUM *	**wolfSSL_ASN1_INTEGER_to_BN**(const WOLFSSL_ASN1_INTEGER * ai, WOLFSSL_BIGNUM * bn)This function is used to copy a WOLFSSL_ASN1_INTEGER value to a WOLFSSL_BIGNUM structure.
long	**wolfSSL_CTX_add_extra_chain_cert**(WOLFSSL_CTX * ctx, WOLFSSL_X509 * x509)This function adds the certificate to the internal chain being built in the WOLFSSL_CTX structure.
int	**wolfSSL_CTX_get_read_ahead**(WOLFSSL_CTX * ctx)This function returns the get read ahead flag from a WOLFSSL_CTX structure.
int	**wolfSSL_CTX_set_read_ahead**(WOLFSSL_CTX * ctx, int v)This function sets the read ahead flag in the WOLFSSL_CTX structure.
long	**wolfSSL_CTX_set_tlsext_status_arg**(WOLFSSL_CTX * ctx, void * arg)This function sets the options argument to use with OCSP.
long	**wolfSSL_CTX_set_tlsext_opaque_prf_input_callback_arg**(WOLF * ctx, void * arg)This function sets the optional argument to be passed to the PRF callback.
long	**wolfSSL_set_options**(WOLFSSL * s, long op)This function sets the options mask in the ssl. Some valid options are, SSL_OP_ALL, SSL_OP_COOKIE_EXCHANGE, SSL_OP_NO_SSLv2, SSL_OP_NO_SSLv3, SSL_OP_NO_TLSv1, SSL_OP_NO_TLSv1_1, SSL_OP_NO_TLSv1_2, SSL_OP_NO_COMPRESSION.
long	**wolfSSL_get_options**(const WOLFSSL * ssl)This function returns the current options mask.

	Name
long	**wolfSSL_set_tlsext_debug_arg**(WOLFSSL * ssl, void * arg)This is used to set the debug argument passed around.
long	**wolfSSL_get_verify_result**(const WOLFSSL * ssl)This is used to get the results after trying to verify the peer's certificate.
int	**wolfSSL_CTX_allow_anon_cipher**(WOLFSSL_CTX *)This function enables the havAnon member of the CTX structure if HAVE_ANON is defined during compilation.
WOLFSSL_METHOD *	**wolfSSLv23_server_method.
int	**wolfSSL_state**(WOLFSSL * ssl)This is used to get the internal error state of the WOLFSSL structure.
int	**wolfSSL_check_domain_name will add a domain name check to the list of checks to perform. dn holds the domain name to check against the peer certificate when it's received.
int	**wolfSSL_set_compression**(WOLFSSL * ssl)Turns on the ability to use compression for the SSL connection. Both sides must have compression turned on otherwise compression will not be used. The zlib library performs the actual data compression. To compile into the library use –with-libz for the configure system and define HAVE_LIBZ otherwise. Keep in mind that while compressing data before sending decreases the actual size of the messages being sent and received, the amount of data saved by compression usually takes longer in time to analyze than it does to send it raw on all but the slowest of networks.
int	**wolfSSL_set_timeout**(WOLFSSL * ssl, unsigned int to)This function sets the SSL session timeout value in seconds.
int	**wolfSSL_CTX_set_timeout**(WOLFSSL_CTX * ctx, unsigned int to)This function sets the timeout value for SSL sessions, in seconds, for the specified SSL context.
int	**wolfSSL_CTX_UnloadCAs**(WOLFSSL_CTX *)This function unloads the CA signer list and frees the whole signer table.
int	**wolfSSL_CTX_UnloadIntermediateCerts**(WOLFSSL_CTX * ctx)This function unloads intermediate certificates added to the CA signer list and frees them.
int	**wolfSSL_CTX_Unload_trust_peers**(WOLFSSL_CTX *)This function is used to unload all previously loaded trusted peer certificates. Feature is enabled by defining the macro WOLFSSL_TRUST_PEER_CERT.

	Name
int	**wolfSSL_CTX_trust_peer_buffer**(WOLFSSL_CTX * ctx, const unsigned char * in, long sz, int format)This function loads a certificate to use for verifying a peer when performing a TLS/SSL handshake. The peer certificate sent during the handshake is compared by using the SKID when available and the signature. If these two things do not match then any loaded CAs are used. Is the same functionality as wolfSSL_CTX_trust_peer_cert except is from a buffer instead of a file. Feature is enabled by defining the macro WOLFSSL_TRUST_PEER_CERT Please see the examples for proper usage.
int	**wolfSSL_CTX_set_group_messages**(WOLFSSL_CTX *)This function turns on grouping of handshake messages where possible.
int	**wolfSSL_set_group_messages**(WOLFSSL *)This function turns on grouping of handshake messages where possible.
int	**wolfSSL_CTX_SetMinVersion**(WOLFSSL_CTX * ctx, int version)This function sets the minimum downgrade version allowed. Applicable only when the connection allows downgrade using (wolfSSLv23_client_method or wolfSSLv23_server_method).
int	**wolfSSL_SetVersion) method type.
int	**wolfSSL_UseALPN**(WOLFSSL * ssl, char * protocol_name_list, unsigned int protocol_name_listSz, unsigned char options)Setup ALPN use for a wolfSSL session.
int	**wolfSSL_CTX_UseSessionTicket**(WOLFSSL_CTX * ctx)This function sets wolfSSL context to use a session ticket.
int	**wolfSSL_check_private_key**(const WOLFSSL * ssl)This function checks that the private key is a match with the certificate being used.
int	**wolfSSL_use_certificate**(WOLFSSL * ssl, WOLFSSL_X509 * x509)his is used to set the certificate for WOLFSSL structure to use during a handshake.
int	**wolfSSL_use_certificate_ASN1**(WOLFSSL * ssl, unsigned char * der, int derSz)This is used to set the certificate for WOLFSSL structure to use during a handshake. A DER formatted buffer is expected.
int	**wolfSSL_SESSION_get_master_key**(const WOLFSSL_SESSION * ses, unsigned char * out, int outSz)This is used to get the master key after completing a handshake.

	Name
int	**wolfSSL_SESSION_get_master_key_length**(const WOLFSSL_SESSION * ses)This is used to get the master secret key length.
void	**wolfSSL_CTX_set_cert_store**(WOLFSSL_CTX * ctx, WOLFSSL_X509_STORE * str)This is a setter function for the WOLFSSL_X509_STORE structure in ctx.
WOLFSSL_X509_STORE *	**wolfSSL_CTX_get_cert_store**(WOLFSSL_CTX * ctx)This is a getter function for the WOLFSSL_X509_STORE structure in ctx.
size_t	**wolfSSL_get_server_random**(const WOLFSSL * ssl, unsigned char * out, size_t outlen)This is used to get the random data sent by the server during the handshake.
size_t	**wolfSSL_get_client_random**(const WOLFSSL * ssl, unsigned char * out, size_t outSz)This is used to get the random data sent by the client during the handshake.
wc_pem_password_cb *	**wolfSSL_CTX_get_default_passwd_cb**(WOLFSSL_CTX * ctx)This is a getter function for the password callback set in ctx.
void *	**wolfSSL_CTX_get_default_passwd_cb_userdata**(WOLFSSL_CTX * ctx)This is a getter function for the password callback user data set in ctx.
long	**wolfSSL_CTX_clear_options**(WOLFSSL_CTX * ctx, long opt)This function resets option bits of WOLFSSL_CTX object.
int	**wolfSSL_set_msg_callback**(WOLFSSL * ssl, SSL_Msg_Cb cb)This function sets a callback in the ssl. The callback is to observe handshake messages. NULL value of cb resets the callback.
int	**wolfSSL_set_msg_callback_arg**(WOLFSSL * ssl, void * arg)This function sets associated callback context value in the ssl. The value is handed over to the callback argument.
int	**wolfSSL_send_hrr_cookie**(WOLFSSL * ssl, const unsigned char * secret, unsigned int secretSz)This function is called on the server side to indicate that a HelloRetryRequest message must contain a Cookie and, in case of using protocol DTLS v1.3, that the handshake will always include a cookie exchange. Please note that when using protocol DTLS v1.3, the cookie exchange is enabled by default. The Cookie holds a hash of the current transcript so that another server process can handle the ClientHello in reply. The secret is used when generating the integrity check on the Cookie data.

	Name
int	**wolfSSL_disable_hrr_cookie**(WOLFSSL * ssl)This function is called on the server side to indicate that a HelloRetryRequest message must NOT contain a Cookie and that, if using protocol DTLS v1.3, a cookie exchange will not be included in the handshake. Please note that not doing a cookie exchange when using protocol DTLS v1.3 can make the server susceptible to DoS/Amplification attacks.
int	**wolfSSL_CTX_no_ticket_TLSv13**(WOLFSSL_CTX * ctx)This function is called on the server to stop it from sending a resumption session ticket once the handshake is complete.
int	**wolfSSL_no_ticket_TLSv13**(WOLFSSL * ssl)This function is called on the server to stop it from sending a resumption session ticket once the handshake is complete.
int	**wolfSSL_CTX_no_dhe_psk**(WOLFSSL_CTX * ctx)This function is called on a TLS v1.3 wolfSSL context to disallow Diffie_Hellman (DH) style key exchanges when handshakes are using pre-shared keys for authentication.
int	**wolfSSL_no_dhe_psk**(WOLFSSL * ssl)This function is called on a TLS v1.3 client or server wolfSSL to disallow Diffie_Hellman (DH) style key exchanges when handshakes are using pre-shared keys for authentication.
int	**wolfSSL_CTX_allow_post_handshake_auth**(WOLFSSL_CTX * ctx)This function is called on a TLS v1.3 client wolfSSL context to allow a client certificate to be sent post handshake upon request from server. This is useful when connecting to a web server that has some pages that require client authentication and others that don't.
int	**wolfSSL_allow_post_handshake_auth**(WOLFSSL * ssl)This function is called on a TLS v1.3 client wolfSSL to allow a client certificate to be sent post handshake upon request from server. A Post-Handshake Client Authentication extension is sent in the ClientHello. This is useful when connecting to a web server that has some pages that require client authentication and others that don't.
int	**wolfSSL_CTX_set1_groups_list**(WOLFSSL_CTX * ctx, char * list)This function sets the list of elliptic curve groups to allow on a wolfSSL context in order of preference. The list is a null-terminated text string, and a colon-delimited list. Call this function to set the key exchange elliptic curve parameters to use with the TLS v1.3 connections.

	Name
int	**wolfSSL_set1_groups_list**(WOLFSSL * ssl, char * list)This function sets the list of elliptic curve groups to allow on a wolfSSL in order of preference. The list is a null-terminated text string, and a colon-delimited list. Call this function to set the key exchange elliptic curve parameters to use with the TLS v1.3 connections.
int	**wolfSSL_CTX_set_groups**(WOLFSSL_CTX * ctx, int * groups, int count)This function sets the list of elliptic curve groups to allow on a wolfSSL context in order of preference. The list is an array of group identifiers with the number of identifiers specified in count. Call this function to set the key exchange elliptic curve parameters to use with the TLS v1.3 connections.
int	**wolfSSL_set_groups**(WOLFSSL * ssl, int * groups, int count)This function sets the list of elliptic curve groups to allow on a wolfSSL. The list is an array of group identifiers with the number of identifiers specified in count. Call this function to set the key exchange elliptic curve parameters to use with the TLS v1.3 connections.
int	**wolfSSL_CTX_set_max_early_data**(WOLFSSL_CTX * ctx, unsigned int sz)This function sets the maximum amount of early data that a TLS v1.3 client or server is willing to exchange using the wolfSSL context. Call this function to limit the amount of early data to process to mitigate replay attacks. Early data is protected by keys derived from those of the connection that the session ticket was sent and therefore will be the same every time a session ticket is used in resumption. The value is included in the session ticket for resumption. A server value of zero indicates no early data is to be sent by client using session tickets. A client value of zero indicates that the client will not send any early data. It is recommended that the number of early data bytes be kept as low as practically possible in the application.

	Name
int	**wolfSSL_set_max_early_data**(WOLFSSL * ssl, unsigned int sz)This function sets the maximum amount of early data that a TLS v1.3 client or server is willing to exchange. Call this function to limit the amount of early data to process to mitigate replay attacks. Early data is protected by keys derived from those of the connection that the session ticket was sent and therefore will be the same every time a session ticket is used in resumption. The value is included in the session ticket for resumption. A server value of zero indicates no early data is to be sent by client using session tickets. A client value of zero indicates that the client will not send any early data. It is recommended that the number of early data bytes be kept as low as practically possible in the application.
void	**wolfSSL_CTX_set_psk_client_tls13_callback**(WOLFSSL_CTX * ctx, wc_psk_client_tls13_callback cb)This function sets the Pre_Shared Key (PSK) client side callback for TLS v1.3 connections. The callback is used to find a PSK identity and return its key and the name of the cipher to use for the handshake. The function sets the client_psk_tls13_cb member of the WOLFSSL_CTX structure.
void	**wolfSSL_set_psk_client_tls13_callback**(WOLFSSL * ssl, wc_psk_client_tls13_callback cb)This function sets the Pre_Shared Key (PSK) client side callback for TLS v1.3 connections. The callback is used to find a PSK identity and return its key and the name of the cipher to use for the handshake. The function sets the client_psk_tls13_cb member of the options field in WOLFSSL structure.
void	**wolfSSL_CTX_set_psk_server_tls13_callback**(WOLFSSL_CTX * ctx, wc_psk_server_tls13_callback cb)This function sets the Pre_Shared Key (PSK) server side callback for TLS v1.3 connections. The callback is used to find a PSK identity and return its key and the name of the cipher to use for the handshake. The function sets the server_psk_tls13_cb member of the WOLFSSL_CTX structure.

	Name
void	**wolfSSL_set_psk_server_tls13_callback**(WOLFSSL * ssl, wc_psk_server_tls13_callback cb)This function sets the Pre_Shared Key (PSK) server side callback for TLS v1.3 connections. The callback is used to find a PSK identity and return its key and the name of the cipher to use for the handshake. The function sets the server_psk_tls13_cb member of the options field in WOLFSSL structure.
int	**wolfSSL_UseKeyShare**(WOLFSSL * ssl, word16 group)This function creates a key share entry from the group including generating a key pair. The KeyShare extension contains all the generated public keys for key exchange. If this function is called, then only the groups specified will be included. Call this function when a preferred group has been previously established for the server.
int	**wolfSSL_NoKeyShares**(WOLFSSL * ssl)This function is called to ensure no key shares are sent in the ClientHello. This will force the server to respond with a HelloRetryRequest if a key exchange is required in the handshake. Call this function when the expected key exchange group is not known and to avoid the generation of keys unnecessarily. Note that an extra round-trip will be required to complete the handshake when a key exchange is required.
WOLFSSL_METHOD *	**wolfTLSv1_3_server_method_ex.
WOLFSSL_METHOD *	**wolfTLSv1_3_client_method_ex.
WOLFSSL_METHOD *	**wolfTLSv1_3_server_method.
WOLFSSL_METHOD *	**wolfTLSv1_3_client_method.
WOLFSSL_METHOD *	**wolfTLSv1_3_method_ex**(void * heap)This function returns a WOLFSSL_METHOD similar to wolfTLSv1_3_client_method except that it is not determined which side yet (server/client).
WOLFSSL_METHOD *	**wolfTLSv1_3_method**(void)This function returns a WOLFSSL_METHOD similar to wolfTLSv1_3_client_method except that it is not determined which side yet (server/client).

	Name
int	**wolfSSL_CTX_set_client_cert_type**(WOLFSSL_CTX * ctx, const char * buf, int len)In case this function is called in a client side, set certificate types that can be sent to its peer. In case called in a server side, set certificate types that can be acceptable from its peer. Put cert types in the buffer with prioritised order. To reset the settings to default, pass NULL for the buffer or pass zero for len. By default, certificate type is only X509. In case both side intend to send or accept “Raw public key” cert, WOLFSSL_CERT_TYPE_RPK should be included in the buffer to set.
int	**wolfSSL_CTX_set_server_cert_type**(WOLFSSL_CTX * ctx, const char * buf, int len)In case this function is called in a server side, set certificate types that can be sent to its peer. In case called in a client side, set certificate types that can be acceptable from its peer. Put cert types in the buffer with prioritised order. To reset the settings to default, pass NULL for the buffer or pass zero for len. By default, certificate type is only X509. In case both side intend to send or accept “Raw public key” cert, WOLFSSL_CERT_TYPE_RPK should be included in the buffer to set.
int	**wolfSSL_set_client_cert_type**(WOLFSSL * ssl, const char * buf, int len)In case this function is called in a client side, set certificate types that can be sent to its peer. In case called in a server side, set certificate types that can be acceptable from its peer. Put cert types in the buffer with prioritised order. To reset the settings to default, pass NULL for the buffer or pass zero for len. By default, certificate type is only X509. In case both side intend to send or accept “Raw public key” cert, WOLFSSL_CERT_TYPE_RPK should be included in the buffer to set.
int	**wolfSSL_set_server_cert_type**(WOLFSSL * ssl, const char * buf, int len)In case this function is called in a server side, set certificate types that can be sent to its peer. In case called in a client side, set certificate types that can be acceptable from its peer. Put cert types in the buffer with prioritised order. To reset the settings to default, pass NULL for the buffer or pass zero for len. By default, certificate type is only X509. In case both side intend to send or accept “Raw public key” cert, WOLFSSL_CERT_TYPE_RPK should be included in the buffer to set.

	Name
void *	**wolfSSL_GetCookieCtx**(WOLFSSL * ssl)This function returns the IOCB_CookieCtx member of the WOLFSSL structure.
int	**wolfSSL_SetIO_ISOTP**(WOLFSSL * ssl, isotp_wolfssl_ctx * ctx, can_recv_fn recv_fn, can_send_fn send_fn, can_delay_fn delay_fn, word32 receive_delay, char * receive_buffer, int receive_buffer_size, void * arg)This function sets up the ISO-TP context if wolfSSL, for use when wolfSSL is compiled with WOLFSSL_ISOTP.
void	**wolfSSL_SSLDisableRead**(WOLFSSL * ssl)This function disables reading from the IO layer.
void	**wolfSSL_SSLEnableRead;.

A.6.2 Functions Documentation

```
WOLFSSL_METHOD * wolfSSLv23_method(
    void
)
```

This function returns a WOLFSSL_METHOD similar to wolfSSLv23_client_method except that it is not determined which side yet (server/client).

Parameters:

- **none** No parameters.

See:

- wolfSSL_new
- wolfSSL_free

Return:

- WOLFSSL_METHOD* On successful creations returns a WOLFSSL_METHOD pointer
- NULL Null if memory allocation error or failure to create method

Example

```
WOLFSSL* ctx;
ctx  = wolfSSL_CTX_new(wolfSSLv23_method());
// check ret value
```

```
WOLFSSL_METHOD * wolfSSLv3_server_method(
    void
)
```

The wolfSSLv3_server_method() function is used to indicate that the application is a server and will only support the SSL 3.0 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().

Parameters:

- **none** No parameters.

See:

- wolfTLSv1_server_method
- wolfTLSv1_1_server_method
- wolfTLSv1_2_server_method
- wolfTLSv1_3_server_method
- wolfDTLSv1_server_method
- wolfSSLv23_server_method
- wolfSSL_CTX_new

Return:

-
  - If successful, the call will return a pointer to the newly created WOLFSSL_METHOD structure.
- FAIL If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfSSLv3_server_method();
if (method == NULL) {
    unable to get method
}

ctx = wolfSSL_CTX_new(method);
...
```

```
WOLFSSL_METHOD * wolfSSLv3_client_method(
    void
)
```

The wolfSSLv3_client_method() function is used to indicate that the application is a client and will only support the SSL 3.0 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().

Parameters:

- **none** No parameters.

See:

- wolfTLSv1_client_method
- wolfTLSv1_1_client_method
- wolfTLSv1_2_client_method
- wolfTLSv1_3_client_method
- wolfDTLSv1_client_method
- wolfSSLv23_client_method
- wolfSSL_CTX_new

Return:

-
  - If successful, the call will return a pointer to the newly created WOLFSSL_METHOD structure.
- FAIL If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfSSLv3_client_method();
if (method == NULL) {
    unable to get method
}

ctx = wolfSSL_CTX_new(method);
...
```

```
WOLFSSL_METHOD * wolfTLSv1_server_method(
    void
)
```

The wolfTLSv1_server_method() function is used to indicate that the application is a server and will only support the TLS 1.0 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().

Parameters:

- **none** No parameters.

See:

- wolfSSLv3_server_method
- wolfTLSv1_1_server_method
- wolfTLSv1_2_server_method
- wolfTLSv1_3_server_method
- wolfDTLSv1_server_method
- wolfSSLv23_server_method
- wolfSSL_CTX_new

Return:

-
  - If successful, the call will return a pointer to the newly created WOLFSSL_METHOD structure.
- FAIL If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfTLSv1_server_method();
if (method == NULL) {
    unable to get method
}

ctx = wolfSSL_CTX_new(method);
...
```

```
WOLFSSL_METHOD * wolfTLSv1_client_method(
    void
)
```

The wolfTLSv1_client_method() function is used to indicate that the application is a client and will only support the TLS 1.0 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().

Parameters:

- **none** No parameters.

See:

- wolfSSLv3_client_method
- wolfTLSv1_1_client_method
- wolfTLSv1_2_client_method
- wolfTLSv1_3_client_method
- wolfDTLSv1_client_method

- wolfSSLv23_client_method
- wolfSSL_CTX_new

Return:

-
  - If successful, the call will return a pointer to the newly created WOLFSSL_METHOD structure.
- FAIL If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfTLSv1_client_method();
if (method == NULL) {
    unable to get method
}

ctx = wolfSSL_CTX_new(method);
...
```

```
WOLFSSL_METHOD * wolfTLSv1_1_server_method(
    void
)
```

The wolfTLSv1_1_server_method() function is used to indicate that the application is a server and will only support the TLS 1.1 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().

Parameters:

- **none** No parameters.

See:

- wolfSSLv3_server_method
- wolfTLSv1_server_method
- wolfTLSv1_2_server_method
- wolfTLSv1_3_server_method
- wolfDTLSv1_server_method
- wolfSSLv23_server_method
- wolfSSL_CTX_new

Return:

-
  - If successful, the call will return a pointer to the newly created WOLFSSL_METHOD structure.
- FAIL If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfTLSv1_1_server_method();
if (method == NULL) {
    // unable to get method
}

ctx = wolfSSL_CTX_new(method);
...
```

```
WOLFSSL_METHOD * wolfTLSv1_1_client_method(
    void
)
```

The wolfTLSv1_1_client_method() function is used to indicate that the application is a client and will only support the TLS 1.0 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().

Parameters:

- **none** No parameters.

See:

- wolfSSLv3_client_method
- wolfTLSv1_client_method
- wolfTLSv1_2_client_method
- wolfTLSv1_3_client_method
- wolfDTLSv1_client_method
- wolfSSLv23_client_method
- wolfSSL_CTX_new

Return:

-
  - If successful, the call will return a pointer to the newly created WOLFSSL_METHOD structure.
- FAIL If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfTLSv1_1_client_method();
if (method == NULL) {
    // unable to get method
}

ctx = wolfSSL_CTX_new(method);
...
```

```
WOLFSSL_METHOD * wolfTLSv1_2_server_method(
    void
)
```

The wolfTLSv1_2_server_method() function is used to indicate that the application is a server and will only support the TLS 1.2 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().

Parameters:

- **none** No parameters.

See:

- wolfSSLv3_server_method
- wolfTLSv1_server_method
- wolfTLSv1_1_server_method
- wolfTLSv1_3_server_method
- wolfDTLSv1_server_method
- wolfSSLv23_server_method
- wolfSSL_CTX_new

Return:

-
  - If successful, the call will return a pointer to the newly created WOLFSSL_METHOD structure.
- FAIL If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Example

```
#include <wolfssl/ssl.h>
```

```
WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfTLSv1_2_server_method();
if (method == NULL) {
    // unable to get method
}

ctx = wolfSSL_CTX_new(method);
...
```

```
WOLFSSL_METHOD * wolfTLSv1_2_client_method(
    void
)
```

The wolfTLSv1_2_client_method() function is used to indicate that the application is a client and will only support the TLS 1.2 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().

Parameters:

- **none** No parameters.

See:

- wolfSSLv3_client_method
- wolfTLSv1_client_method
- wolfTLSv1_1_client_method
- wolfTLSv1_3_client_method
- wolfDTLSv1_client_method
- wolfSSLv23_client_method
- wolfSSL_CTX_new

Return:

-
  - If successful, the call will return a pointer to the newly created WOLFSSL_METHOD structure.
- FAIL If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfTLSv1_2_client_method();
if (method == NULL) {
```

```
    // unable to get method
}

ctx = wolfSSL_CTX_new(method);
...
```

```
WOLFSSL_METHOD * wolfDTLSv1_client_method(
    void
)
```

The wolfDTLSv1_client_method() function is used to indicate that the application is a client and will only support the DTLS 1.0 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new(). This function is only available when wolfSSL has been compiled with DTLS support (–enable-dtls, or by defining wolfSSL_DTLS).

Parameters:

- **none** No parameters.

See:

- wolfSSLv3_client_method
- wolfTLSv1_client_method
- wolfTLSv1_1_client_method
- wolfTLSv1_2_client_method
- wolfTLSv1_3_client_method
- wolfSSLv23_client_method
- wolfSSL_CTX_new

Return:

-
  - If successful, the call will return a pointer to the newly created WOLFSSL_METHOD structure.
- FAIL If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Example

```
WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfDTLSv1_client_method();
if (method == NULL) {
    // unable to get method
}

ctx = wolfSSL_CTX_new(method);
...
```

```
WOLFSSL_METHOD * wolfDTLSv1_server_method(
    void
)
```

The wolfDTLSv1_server_method() function is used to indicate that the application is a server and will only support the DTLS 1.0 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new(). This function is only available when wolfSSL has been compiled with DTLS support (–enable-dtls, or by defining wolfSSL_DTLS).

Parameters:

- **none** No parameters.

See:

- wolfSSLv3_server_method
- wolfTLSv1_server_method
- wolfTLSv1_1_server_method
- wolfTLSv1_2_server_method
- wolfTLSv1_3_server_method
- wolfSSLv23_server_method
- wolfSSL_CTX_new

Return:

-
  - If successful, the call will return a pointer to the newly created WOLFSSL_METHOD structure.
- FAIL If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Example

```
WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfDTLSv1_server_method();
if (method == NULL) {
    // unable to get method
}

ctx = wolfSSL_CTX_new(method);
...
```

```
WOLFSSL_METHOD * wolfDTLSv1_3_server_method(
    void
)
```

The wolfDTLSv1_3_server_method() function is used to indicate that the application is a server and will only support the DTLS 1.3 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new(). This function is only available when wolfSSL has been compiled with DTLSv1.3 support (–enable-dtls13, or by defining wolfSSL_DTLS13).

Parameters:

- **none** No parameters.

See: wolfDTLSv1_3_client_method

Return:

-
  - If successful, the call will return a pointer to the newly created WOLFSSL_METHOD structure.
- FAIL If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Example

```
WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfDTLSv1_3_server_method();
if (method == NULL) {
    // unable to get method
}

ctx = wolfSSL_CTX_new(method);
...
```

```
WOLFSSL_METHOD * wolfDTLSv1_3_client_method(
    void
)
```

The wolfDTLSv1_3_client_method() function is used to indicate that the application is a client and will only support the DTLS 1.3 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new(). This function is only available when wolfSSL has been compiled with DTLSv1.3 support (–enable-dtls13, or by defining wolfSSL_DTLS13).

Parameters:

- **none** No parameters.

See: wolfDTLSv1_3_server_method

Return:

-
  - If successful, the call will return a pointer to the newly created WOLFSSL_METHOD structure.
- FAIL If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Example

```
WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfDTLSv1_3_client_method();
if (method == NULL) {
    // unable to get method
}

ctx = wolfSSL_CTX_new(method);
...
```

```
WOLFSSL_METHOD * wolfDTLS_server_method(
    void
)
```

The wolfDTLS_server_method() function is used to indicate that the application is a server and will support the highest version of DTLS available and all the version up to the minimum version allowed. The default minimum version allowed is based on the define WOLFSSL_MIN_DTLS_DOWNGRADE and can be changed at runtime using wolfSSL_SetMinVersion(). This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new(). This function is only available when wolfSSL has been compiled with DTLS support (–enable-dtls, or by defining wolfSSL_DTLS).

Parameters:

- **none** No parameters.

See:

- wolfDTLS_client_method
- wolfSSL_SetMinVersion

Return:

-
  - If successful, the call will return a pointer to the newly created WOLFSSL_METHOD structure.
- FAIL If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Example

```
WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfDTLS_server_method();
if (method == NULL) {
    // unable to get method
}

ctx = wolfSSL_CTX_new(method);
...
```

```
WOLFSSL_METHOD * wolfDTLS_client_method(
    void
)
```

The wolfDTLS_client_method() function is used to indicate that the application is a client and will support the highest version of DTLS available and all the version up to the minimum version allowed. The default minimum version allowed is based on the define WOLFSSL_MIN_DTLS_DOWNGRADE and can be changed at runtime using wolfSSL_SetMinVersion(). This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new(). This function is only available when wolfSSL has been compiled with DTLS support (–enable-dtls, or by defining wolfSSL_DTLS).

Parameters:

- **none** No parameters.

See:

- wolfDTLS_server_method
- wolfSSL_SetMinVersion

Return:

-
  - If successful, the call will return a pointer to the newly created WOLFSSL_METHOD structure.
- FAIL If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Example

```
WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfDTLS_client_method();
if (method == NULL) {
    // unable to get method
}
```

```
ctx = wolfSSL_CTX_new(method);
...
```

```
int wolfSSL_use_old_poly(
    WOLFSSL * ssl,
    int value
)
```

Since there is some differences between the first release and newer versions of chacha-poly AEAD construction we have added an option to communicate with servers/clients using the older version. By default wolfSSL uses the new version.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **value** whether or not to use the older version of setting up the information for poly1305. Passing a flag value of 1 indicates yes use the old poly AEAD, to switch back to using the new version pass a flag value of 0.

See: none

Return: 0 upon success

Example

```
int ret = 0;
WOLFSSL* ssl;
...

ret = wolfSSL_use_old_poly(ssl, 1);
if (ret != 0) {
    // failed to set poly1305 AEAD version
}
```

```
int wolfSSL_CTX_trust_peer_cert(
    WOLFSSL_CTX * ctx,
    const char * file,
    int type
)
```

This function loads a certificate to use for verifying a peer when performing a TLS/SSL handshake. The peer certificate sent during the handshake is compared by using the SKID when available and the signature. If these two things do not match then any loaded CAs are used. Feature is enabled by defining the macro WOLFSSL_TRUST_PEER_CERT. Please see the examples for proper usage.

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().
- **file** pointer to name of the file containing certificates
- **type** type of certificate being loaded ie SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM.

See:

- wolfSSL_CTX_load_verify_buffer
- wolfSSL_CTX_use_certificate_file
- wolfSSL_CTX_use_PrivateKey_file
- wolfSSL_CTX_use_certificate_chain_file
- wolfSSL_CTX_trust_peer_buffer
- wolfSSL_CTX_Unload_trust_peers
- wolfSSL_use_certificate_file
- wolfSSL_use_PrivateKey_file
- wolfSSL_use_certificate_chain_file

Return:

- SSL_SUCCES upon success.
- SSL_FAILURE will be returned if ctx is NULL, or if both file and type are invalid.
- SSL_BAD_FILETYPE will be returned if the file is the wrong format.
- SSL_BAD_FILE will be returned if the file doesn't exist, can't be read, or is corrupted.
- MEMORY_E will be returned if an out of memory condition occurs.
- ASN_INPUT_E will be returned if Base16 decoding fails on the file.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
...

ret = wolfSSL_CTX_trust_peer_cert(ctx, "./peer-cert.pem",
SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
    // error loading trusted peer cert
}
...
```

```
long wolfSSL_CTX_get_verify_depth(
    WOLFSSL_CTX * ctx
)
```

This function gets the certificate chaining depth using the CTX structure.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().

See:

- wolfSSL_CTX_use_certificate_chain_file
- wolfSSL_get_verify_depth

Return:

- MAX_CHAIN_DEPTH returned if the CTX struct is not NULL. The constant representation of the max certificate chain peer depth.
- BAD_FUNC_ARG returned if the CTX structure is NULL.

Example

```
WOLFSSL_METHOD method; // protocol method
WOLFSSL_CTX* ctx = WOLFSSL_CTX_new(method);
…
long ret = wolfSSL_CTX_get_verify_depth(ctx);

if(ret == EXPECTED){
    //  You have the expected value
} else {
    //  Handle an unexpected depth
}
```

```
WOLFSSL_CTX * wolfSSL_CTX_new(
    WOLFSSL_METHOD *
)
```

This function creates a new SSL context, taking a desired SSL/TLS protocol method for input.

Parameters:

- **method** pointer to the desired WOLFSSL_METHOD to use for the SSL context. This is created using one of the wolfSSLvXX_XXXX_method() functions to specify SSL/TLS/DTLS protocol level.

See: wolfSSL_new

Return:

- pointer If successful the call will return a pointer to the newly-created WOLFSSL_CTX.
- NULL upon failure.

Example

```
WOLFSSL_CTX*    ctx    = 0;
WOLFSSL_METHOD* method = 0;

method = wolfSSLv3_client_method();
if (method == NULL) {
    // unable to get method
}

ctx = wolfSSL_CTX_new(method);
if (ctx == NULL) {
```

```
    // context creation failed
}
```

```
WOLFSSL * wolfSSL_new(
    WOLFSSL_CTX *
)
```

This function creates a new SSL session, taking an already created SSL context as input.

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().

See: wolfSSL_CTX_new

Return:

-
  - If successful the call will return a pointer to the newly-created wolfSSL structure.
- NULL Upon failure.

Example

```
#include <wolfssl/ssl.h>

WOLFSSL*     ssl = NULL;
WOLFSSL_CTX* ctx = 0;

ctx = wolfSSL_CTX_new(method);
if (ctx == NULL) {
    // context creation failed
}

ssl = wolfSSL_new(ctx);
if (ssl == NULL) {
    // SSL object creation failed
}
```

```
int wolfSSL_set_fd(
    WOLFSSL * ssl,
    int fd
)
```

This function assigns a file descriptor (fd) as the input/output facility for the SSL connection. Typically this will be a socket file descriptor.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().
- **fd** file descriptor to use with SSL/TLS connection.

See:

- wolfSSL_CTX_SetIOSend
- wolfSSL_CTX_SetIORecv
- wolfSSL_SetIOReadCtx
- wolfSSL_SetIOWriteCtx

Return:

- SSL_SUCCESS upon success.
- BAD_FUNC_ARG upon failure.

Example

```
int sockfd;
WOLFSSL* ssl = 0;
...

ret = wolfSSL_set_fd(ssl, sockfd);
if (ret != SSL_SUCCESS) {
    // failed to set SSL file descriptor
}
```

```
int wolfSSL_set_dtls_fd_connected(
    WOLFSSL * ssl,
    int fd
)
```

This function assigns a file descriptor (fd) as the input/output facility for the SSL connection. Typically this will be a socket file descriptor. This is a DTLS specific API because it marks that the socket is connected. recvfrom and sendto calls on this fd will have the addr and addr_len parameters set to NULL.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().
- **fd** file descriptor to use with SSL/TLS connection.

See:

- wolfSSL_CTX_SetIOSend
- wolfSSL_CTX_SetIORecv
- wolfSSL_SetIOReadCtx
- wolfSSL_SetIOWriteCtx
- wolfDTLS_SetChGoodCb

Return:

- SSL_SUCCESS upon success.
- BAD_FUNC_ARG upon failure.

Example

```
int sockfd;
WOLFSSL* ssl = 0;
...
if (connect(sockfd, peer_addr, peer_addr_len) != 0) {
    // handle connect error
}
...
ret = wolfSSL_set_dtls_fd_connected(ssl, sockfd);
if (ret != SSL_SUCCESS) {
    // failed to set SSL file descriptor
}
```

```
int wolfDTLS_SetChGoodCb(
    WOLFSSL * ssl,
    ClientHelloGoodCb cb,
    void * user_ctx
)
```

Allows setting a callback for a correctly processed and verified DTLS client hello. When using a cookie exchange mechanism (either the HelloVerifyRequest in DTLS 1.2 or the HelloRetryRequest with a cookie extension in DTLS 1.3) this callback is called after the cookie exchange has succeeded. This is useful to use one WOLFSSL object as the listener for new connections and being able to isolate the WOLFSSL object once the ClientHello is verified (either through a cookie exchange or just checking if the ClientHello had the correct format). DTLS 1.2: https://datatracker.ietf.org/doc/html/rfc6347#section-4.2.1 DTLS 1.3: https://www.rfc-editor.org/rfc/rfc8446#section-4.2.2.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().
- **fd** file descriptor to use with SSL/TLS connection.

See: wolfSSL_set_dtls_fd_connected

Return:

- SSL_SUCCESS upon success.
- BAD_FUNC_ARG upon failure.

Example

```
// Called when we have verified a connection
static int chGoodCb(WOLFSSL* ssl, void* arg)
{
    // setup peer and file descriptors

}

if (wolfDTLS_SetChGoodCb(ssl, chGoodCb, NULL) != WOLFSSL_SUCCESS) {
     // error setting callback
}
```

```
void wolfSSL_set_using_nonblock(
    WOLFSSL * ssl,
    int nonblock
)
```

This function informs the WOLFSSL object that the underlying I/O is non-blocking. After an application creates a WOLFSSL object, if it will be used with a non-blocking socket, call wolfSSL_set_using_nonblock() on it. This lets the WOLFSSL object know that receiving EWOULDBLOCK means that the recvfrom call would block rather than that it timed out.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().
- **nonblock** value used to set non-blocking flag on WOLFSSL object. Use 1 to specify non-blocking, otherwise 0.

See:

- wolfSSL_get_using_nonblock
- wolfSSL_dtls_got_timeout
- wolfSSL_dtls_get_current_timeout

Return: none No return.

Example

```
WOLFSSL* ssl = 0;
...
wolfSSL_set_using_nonblock(ssl, 1);
```

```
void wolfSSL_CTX_free(
    WOLFSSL_CTX *
)
```

This function frees an allocated WOLFSSL_CTX object. This function decrements the CTX reference count and only frees the context when the reference count has reached 0.

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().

See:

- wolfSSL_CTX_new
- wolfSSL_new
- wolfSSL_free

Return: none No return.

Example

```
WOLFSSL_CTX* ctx = 0;
...
wolfSSL_CTX_free(ctx);
```

```
void wolfSSL_free(
    WOLFSSL *
)
```

This function frees an allocated wolfSSL object.

Parameters:

- **ssl** pointer to the SSL object, created with wolfSSL_new().

See:

- wolfSSL_CTX_new
- wolfSSL_new
- wolfSSL_CTX_free

Return: none No return.

Example

```
#include <wolfssl/ssl.h>

WOLFSSL* ssl = 0;
...
wolfSSL_free(ssl);
```

```
int wolfSSL_set_session(
    WOLFSSL * ssl,
    WOLFSSL_SESSION * session
)
```

This function sets the session to be used when the SSL object, ssl, is used to establish a SSL/TLS connection. For session resumption, before calling wolfSSL_shutdown() with your session object, an application should save the session ID from the object with a call to wolfSSL_get1_session(), which returns a pointer to the session. Later, the application should create a new WOLFSSL object and assign the saved session with wolfSSL_set_session(). At this point, the application may call wolfSSL_connect() and wolfSSL will try to resume the session. The wolfSSL server code allows session resumption by default. The object returned by wolfSSL_get1_session() needs to be freed after the application is done with it by calling wolfSSL_SESSION_free() on it.

Parameters:

- **ssl** pointer to the SSL object, created with wolfSSL_new().
- **session** pointer to the WOLFSSL_SESSION used to set the session for ssl.

See: wolfSSL_get1_session

Return:

- SSL_SUCCESS will be returned upon successfully setting the session.
- SSL_FAILURE will be returned on failure. This could be caused by the session cache being disabled, or if the session has timed out.
- When OPENSSL_EXTRA and WOLFSSL_ERROR_CODE_OPENSSL are defined, SSL_SUCCESS will be returned even if the session has timed out.

Example

```
int ret;
WOLFSSL* ssl;
WOLFSSL_SESSION* session;
...
session = wolfSSL_get1_session(ssl);
if (session == NULL) {
    // failed to get session object from ssl object
}
...
ret = wolfSSL_set_session(ssl, session);
if (ret != SSL_SUCCESS) {
    // failed to set the SSL session
}
wolfSSL_SESSION_free(session);
...
```

```
void wolfSSL_CTX_set_verify(
    WOLFSSL_CTX * ctx,
    int mode,
    VerifyCallback verify_callback
)
```

This function sets the verification method for remote peers and also allows a verify callback to be registered with the SSL context. The verify callback will be called only when a verification failure has

occurred. If no verify callback is desired, the NULL pointer can be used for verify_callback. The verification mode of peer certificates is a logically OR'd list of flags. The possible flag values include: SSL_VERIFY_NONE Client mode: the client will not verify the certificate received from the server and the handshake will continue as normal. Server mode: the server will not send a certificate request to the client. As such, client verification will not be enabled. SSL_VERIFY_PEER Client mode: the client will verify the certificate received from the server during the handshake. This is turned on by default in wolfSSL, therefore, using this option has no effect. Server mode: the server will send a certificate request to the client and verify the client certificate received. SSL_VERIFY_FAIL_IF_NO_PEER_CERT Client mode: no effect when used on the client side. Server mode: the verification will fail on the server side if the client fails to send a certificate when requested to do so (when using SSL_VERIFY_PEER on the SSL server). SSL_VERIFY_FAIL_EXCEPT_PSK Client mode: no effect when used on the client side. Server mode: the verification is the same as SSL_VERIFY_FAIL_IF_NO_PEER_CERT except in the case of a PSK connection. If a PSK connection is being made then the connection will go through without a peer cert.

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().
- **mode** flags indicating verification mode for peer's cert.
- **verify_callback** callback to be called when verification fails. If no callback is desired, the NULL pointer can be used for verify_callback.

See: wolfSSL_set_verify

Return: none No return.

Example

```
WOLFSSL_CTX*    ctx    = 0;
...
wolfSSL_CTX_set_verify(ctx, (WOLFSSL_VERIFY_PEER |
                        WOLFSSL_VERIFY_FAIL_IF_NO_PEER_CERT), NULL);
```

```
void wolfSSL_set_verify(
    WOLFSSL * ssl,
    int mode,
    VerifyCallback verify_callback
)
```

This function sets the verification method for remote peers and also allows a verify callback to be registered with the SSL session. The verify callback will be called only when a verification failure has occurred. If no verify callback is desired, the NULL pointer can be used for verify_callback. The verification mode of peer certificates is a logically OR'd list of flags. The possible flag values include: SSL_VERIFY_NONE Client mode: the client will not verify the certificate received from the server and the handshake will continue as normal. Server mode: the server will not send a certificate request to the client. As such, client verification will not be enabled. SSL_VERIFY_PEER Client mode: the client will verify the certificate received from the server during the handshake. This is turned on by default in wolfSSL, therefore, using this option has no effect. Server mode: the server will send a certificate request to the client and verify the client certificate received. SSL_VERIFY_FAIL_IF_NO_PEER_CERT Client mode: no effect when used on the client side. Server mode: the verification will fail on the server side if the client fails to send a certificate when requested to do so (when using SSL_VERIFY_PEER on the SSL server). SSL_VERIFY_FAIL_EXCEPT_PSK Client mode: no effect when used on the client side. Server

mode: the verification is the same as SSL_VERIFY_FAIL_IF_NO_PEER_CERT except in the case of a PSK connection. If a PSK connection is being made then the connection will go through without a peer cert.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().
- **mode** flags indicating verification mode for peer's cert.
- **verify_callback** callback to be called when verification fails. If no callback is desired, the NULL pointer can be used for verify_callback.

See: wolfSSL_CTX_set_verify

Return: none No return.

Example

```
WOLFSSL* ssl = 0;
...
wolfSSL_set_verify(ssl, SSL_VERIFY_PEER | SSL_VERIFY_FAIL_IF_NO_PEER_CERT, 0);
```

```
long wolfSSL_CTX_set_session_cache_mode(
    WOLFSSL_CTX * ctx,
    long mode
)
```

This function enables or disables SSL session caching. Behavior depends on the value used for mode. The following values for mode are available: SSL_SESS_CACHE_OFF- disable session caching. Session caching is turned on by default. SSL_SESS_CACHE_NO_AUTO_CLEAR - Disable auto-flushing of the session cache. Auto-flushing is turned on by default.

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().
- **mode** modifier used to change behavior of the session cache.

See:

- wolfSSL_flush_sessions
- wolfSSL_get1_session
- wolfSSL_set_session
- wolfSSL_get_sessionID
- wolfSSL_CTX_set_timeout

Return: SSL_SUCCESS will be returned upon success.

Example

```
WOLFSSL_CTX* ctx = 0;
...
ret = wolfSSL_CTX_set_session_cache_mode(ctx, SSL_SESS_CACHE_OFF);
```

```
if (ret != SSL_SUCCESS) {
    // failed to turn SSL session caching off
}
```

```
int wolfSSL_CTX_memrestore_cert_cache(
    WOLFSSL_CTX * ctx,
    const void * mem,
    int sz
)
```

This function restores the certificate cache from memory.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().
- **mem** a void pointer with a value that will be restored to the certificate cache.
- **sz** an int type that represents the size of the mem parameter.

See: CM_MemRestoreCertCache

Return:

- SSL_SUCCESS returned if the function and subroutines executed without an error.
- BAD_FUNC_ARG returned if the ctx or mem parameters are NULL or if the sz parameter is less than or equal to zero.
- BUFFER_E returned if the cert cache memory buffer is too small.
- CACHE_MATCH_ERROR returned if there was a cert cache header mismatch.
- BAD_MUTEX_E returned if the lock mutex on failed.

Example

```
WOLFSSL_CTX* ctx = WOLFSSL_CTX_new( protocol method );
WOLFSSL* ssl = WOLFSSL_new(ctx);
void* mem;
int sz = (*int) sizeof(mem);
…
if(wolfSSL_CTX_memrestore_cert_cache(ssl->ctx, mem, sz)){
    // The success case
}
```

```
int wolfSSL_CTX_set_cipher_list(
    WOLFSSL_CTX * ctx,
    const char * list
)
```

This function sets cipher suite list for a given WOLFSSL_CTX. This cipher suite list becomes the default list for any new SSL sessions (WOLFSSL) created using this context. The ciphers in the list should be sorted in order of preference from highest to lowest. Each call to wolfSSL_CTX_set_cipher_list() resets the cipher suite list for the specific SSL context to the provided list each time the function is

called. The cipher suite list, list, is a null-terminated text string, and a colon-delimited list. For example, one value for list may be “DHE-RSA-AES256-SHA256:DHE-RSA-AES128-SHA256:AES256-SHA256” Valid cipher values are the full name values from the cipher_names[] array in src/internal.c (for a definite list of valid cipher values check src/internal.c)

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().
- **list** null-terminated text string and a colon-delimited list of cipher suites to use with the specified SSL context.

See:

- wolfSSL_set_cipher_list
- wolfSSL_CTX_new

Return:

- SSL_SUCCESS will be returned upon successful function completion.
- SSL_FAILURE will be returned on failure.

Example

```
WOLFSSL_CTX* ctx = 0;
...
ret = wolfSSL_CTX_set_cipher_list(ctx,
“DHE-RSA-AES256-SHA256:DHE-RSA-AES128-SHA256:AES256-SHA256”);
if (ret != SSL_SUCCESS) {
    // failed to set cipher suite list
}
```

```
int wolfSSL_set_cipher_list(
    WOLFSSL * ssl,
    const char * list
)
```

This function sets cipher suite list for a given WOLFSSL object (SSL session). The ciphers in the list should be sorted in order of preference from highest to lowest. Each call to wolfSSL_set_cipher_list() resets the cipher suite list for the specific SSL session to the provided list each time the function is called. The cipher suite list, list, is a null-terminated text string, and a colon-delimited list. For example, one value for list may be “DHE-RSA-AES256-SHA256:DHE-RSA-AES128-SHA256:AES256-SHA256”. Valid cipher values are the full name values from the cipher_names[] array in src/internal.c (for a definite list of valid cipher values check src/internal.c)

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().
- **list** null-terminated text string and a colon-delimited list of cipher suites to use with the specified SSL session.

See:

- wolfSSL_CTX_set_cipher_list
- wolfSSL_new

Return:

- SSL_SUCCESS will be returned upon successful function completion.
- SSL_FAILURE will be returned on failure.

Example

```
int ret = 0;
WOLFSSL* ssl = 0;
...
ret = wolfSSL_set_cipher_list(ssl,
“DHE-RSA-AES256-SHA256:DHE-RSA-AES128-SHA256:AES256-SHA256”);
if (ret != SSL_SUCCESS) {
    // failed to set cipher suite list
}
```

```
void wolfSSL_dtls13_set_send_more_acks(
    WOLFSSL * ssl,
    int value
)
```

This function sets whether the library should send ACKs to the other peer immediately when detecting disruption or not. Sending ACKs immediately assures minimum latency but it may consume more bandwidth than necessary. If the application manages the timer by itself and this option is set to 0 then application code can use wolfSSL_dtls13_use_quick_timeout() to determine if it should setup a quicker timeout to send those delayed ACKs.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **value** 1 to set the option, 0 to disable the option

See:

- wolfSSL_dtls
- wolfSSL_dtls_get_peer
- wolfSSL_dtls_got_timeout
- wolfSSL_dtls_set_peer
- wolfSSL_dtls13_use_quick_timeout

```
int wolfSSL_dtls_set_timeout_init(
    WOLFSSL * ssl,
    int
)
```

This function sets the dtls timeout.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **timeout** an int type that will be set to the dtls_timeout_init member of the WOLFSSL structure.

See:

- wolfSSL_dtls_set_timeout_max
- wolfSSL_dtls_got_timeout

Return:

- SSL_SUCCESS returned if the function executes without an error. The dtls_timeout_init and the dtls_timeout members of SSL have been set.
- BAD_FUNC_ARG returned if the WOLFSSL struct is NULL or if the timeout is not greater than 0. It will also return if the timeout argument exceeds the maximum value allowed.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
int timeout = TIMEOUT;
...
if(wolfSSL_dtls_set_timeout_init(ssl, timeout)){
    // the dtls timeout was set
} else {
    // Failed to set DTLS timeout.
}
```

```
WOLFSSL_SESSION * wolfSSL_get1_session(
    WOLFSSL * ssl
)
```

This function returns the WOLFSSL_SESSION from the WOLFSSL structure as a reference type. This requires calling wolfSSL_SESSION_free to release the session reference. The WOLFSSL_SESSION pointed to contains all the necessary information required to perform a session resumption and reestablish the connection without a new handshake. For session resumption, before calling wolfSSL_shutdown() with your session object, an application should save the session ID from the object with a call to wolfSSL_get1_session(), which returns a pointer to the session. Later, the application should create a new WOLFSSL object and assign the saved session with wolfSSL_set_session() and wolfSSL will try to resume the session. The wolfSSL server code allows session resumption by default. The object returned by wolfSSL_get1_session() needs to be freed after the application is done with it by calling wolfSSL_SESSION_free() on it.

Parameters:

- **ssl** WOLFSSL structure to get session from.

See:

- wolfSSL_new
- wolfSSL_free
- wolfSSL_SESSION_free

Return:

- WOLFSSL_SESSION On success return session pointer.
- NULL will be returned if ssl is NULL, the SSL session cache is disabled, wolfSSL doesn't have the Session ID available, or mutex functions fail.

Example

```
WOLFSSL* ssl;
WOLFSSL_SESSION* ses;
// attempt/complete handshake
wolfSSL_connect(ssl);
ses  = wolfSSL_get1_session(ssl);
// check ses information
// disconnect / setup new SSL instance
wolfSSL_set_session(ssl, ses);
// attempt/resume handshake
wolfSSL_SESSION_free(ses);
```

```
WOLFSSL_METHOD * wolfSSLv23_client_method(
    void
)
```

The wolfSSLv23_client_method() function is used to indicate that the application is a client and will support the highest protocol version supported by the server between SSL 3.0 - TLS 1.3. This function allocates memory for and initializes a new WOLFSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new(). Both wolfSSL clients and servers have robust version downgrade capability. If a specific protocol version method is used on either side, then only that version will be negotiated or an error will be returned. For example, a client that uses TLSv1 and tries to connect to a SSLv3 only server will fail, likewise connecting to a TLSv1.1 will fail as well. To resolve this issue, a client that uses the wolfSSLv23_client_method() function will use the highest protocol version supported by the server and downgrade to SSLv3 if needed. In this case, the client will be able to connect to a server running SSLv3 - TLSv1.3.

Parameters:

- **none** No parameters

See:

- wolfSSLv3_client_method
- wolfTLSv1_client_method
- wolfTLSv1_1_client_method
- wolfTLSv1_2_client_method
- wolfTLSv1_3_client_method
- wolfDTLSv1_client_method
- wolfSSL_CTX_new

Return:

- pointer upon success a pointer to a WOLFSSL_METHOD.
- Failure If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Example

```
WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;
method = wolfSSLv23_client_method();
if (method == NULL) {
// unable to get method
}

ctx = wolfSSL_CTX_new(method);
...
```

```
WOLFSSL_BIGNUM * wolfSSL_ASN1_INTEGER_to_BN(
    const WOLFSSL_ASN1_INTEGER * ai,
    WOLFSSL_BIGNUM * bn
)
```

This function is used to copy a WOLFSSL_ASN1_INTEGER value to a WOLFSSL_BIGNUM structure.

Parameters:

- **ai** WOLFSSL_ASN1_INTEGER structure to copy from.
- **bn** if wanting to copy into an already existing WOLFSSL_BIGNUM struct then pass in a pointer to it. Optionally this can be NULL and a new WOLFSSL_BIGNUM structure will be created.

See: none

Return:

- pointer On successfully copying the WOLFSSL_ASN1_INTEGER value a WOLFSSL_BIGNUM pointer is returned.
- Null upon failure.

Example

```
WOLFSSL_ASN1_INTEGER* ai;
WOLFSSL_BIGNUM* bn;
// create ai
bn = wolfSSL_ASN1_INTEGER_to_BN(ai, NULL);

// or if having already created bn and wanting to reuse structure
// wolfSSL_ASN1_INTEGER_to_BN(ai, bn);
// check bn is or return value is not NULL
```

```
long wolfSSL_CTX_add_extra_chain_cert(
    WOLFSSL_CTX * ctx,
    WOLFSSL_X509 * x509
)
```

This function adds the certificate to the internal chain being built in the WOLFSSL_CTX structure.

Parameters:

- **ctx** WOLFSSL_CTX structure to add certificate to.
- **x509** certificate to add to the chain.

See:

- wolfSSL_CTX_new
- wolfSSL_CTX_free

Return:

- SSL_SUCCESS after successfully adding the certificate.
- SSL_FAILURE if failing to add the certificate to the chain.

Example

```
WOLFSSL_CTX* ctx;
WOLFSSL_X509* x509;
int ret;
// create ctx
ret = wolfSSL_CTX_add_extra_chain_cert(ctx, x509);
// check ret value
```

```
int wolfSSL_CTX_get_read_ahead(
    WOLFSSL_CTX * ctx
)
```

This function returns the get read ahead flag from a WOLFSSL_CTX structure.

Parameters:

- **ctx** WOLFSSL_CTX structure to get read ahead flag from.

See:

- wolfSSL_CTX_new
- wolfSSL_CTX_free
- wolfSSL_CTX_set_read_ahead

Return:

- flag On success returns the read ahead flag.
- SSL_FAILURE If ctx is NULL then SSL_FAILURE is returned.

Example

```
WOLFSSL_CTX* ctx;
int flag;
// setup ctx
flag = wolfSSL_CTX_get_read_ahead(ctx);
//check flag
```

```
int wolfSSL_CTX_set_read_ahead(
    WOLFSSL_CTX * ctx,
    int v
)
```

This function sets the read ahead flag in the WOLFSSL_CTX structure.

Parameters:

- **ctx** WOLFSSL_CTX structure to set read ahead flag.
- **v** read ahead flag

See:

- wolfSSL_CTX_new
- wolfSSL_CTX_free
- wolfSSL_CTX_get_read_ahead

Return:

- SSL_SUCCESS If ctx read ahead flag set.
- SSL_FAILURE If ctx is NULL then SSL_FAILURE is returned.

Example

```
WOLFSSL_CTX* ctx;
int flag;
int ret;
// setup ctx
ret = wolfSSL_CTX_set_read_ahead(ctx, flag);
// check return value
```

```
long wolfSSL_CTX_set_tlsext_status_arg(
    WOLFSSL_CTX * ctx,
    void * arg
)
```

This function sets the options argument to use with OCSP.

Parameters:

- **ctx** WOLFSSL_CTX structure to set user argument.
- **arg** user argument.

See:

- wolfSSL_CTX_new
- wolfSSL_CTX_free

Return:

- SSL_FAILURE If ctx or it's cert manager is NULL.
- SSL_SUCCESS If successfully set.

Example

```
WOLFSSL_CTX* ctx;
void* data;
int ret;
// setup ctx
ret = wolfSSL_CTX_set_tlsext_status_arg(ctx, data);

//check ret value
```

```
long wolfSSL_CTX_set_tlsext_opaque_prf_input_callback_arg(
    WOLFSSL_CTX * ctx,
    void * arg
)
```

This function sets the optional argument to be passed to the PRF callback.

Parameters:

- **ctx** WOLFSSL_CTX structure to set user argument.
- **arg** user argument.

See:

- wolfSSL_CTX_new
- wolfSSL_CTX_free

Return:

- SSL_FAILURE If ctx is NULL.
- SSL_SUCCESS If successfully set.

Example

```
WOLFSSL_CTX* ctx;
void* data;
int ret;
// setup ctx
ret = wolfSSL_CTX_set_tlsext_opaques_prf_input_callback_arg(ctx, data);
//check ret value
```

```
long wolfSSL_set_options(
    WOLFSSL * s,
    long op
)
```

This function sets the options mask in the ssl. Some valid options are, SSL_OP_ALL, SSL_OP_COOKIE_EXCHANGE, SSL_OP_NO_SSLv2, SSL_OP_NO_SSLv3, SSL_OP_NO_TLSv1, SSL_OP_NO_TLSv1_1, SSL_OP_NO_TLSv1_2, SSL_OP_NO_COMPRESSION.

Parameters:

- **s** WOLFSSL structure to set options mask.
- **op** This function sets the options mask in the ssl. Some valid options are: SSL_OP_ALL SSL_OP_COOKIE_EXCHANGE SSL_OP_NO_SSLv2 SSL_OP_NO_SSLv3 SSL_OP_NO_TLSv1 SSL_OP_NO_TLSv1_1 SSL_OP_NO_TLSv1_2 SSL_OP_NO_COMPRESSION

See:

- wolfSSL_new
- wolfSSL_free
- wolfSSL_get_options

Return: val Returns the updated options mask value stored in ssl.

Example

```
WOLFSSL* ssl;
unsigned long mask;
mask = SSL_OP_NO_TLSv1
mask  = wolfSSL_set_options(ssl, mask);
// check mask
```

```
long wolfSSL_get_options(
    const WOLFSSL * ssl
)
```

This function returns the current options mask.

Parameters:

- **ssl** WOLFSSL structure to get options mask from.

See:

- wolfSSL_new
- wolfSSL_free
- wolfSSL_set_options

Return: val Returns the mask value stored in ssl.

Example

```
WOLFSSL* ssl;
unsigned long mask;
mask  = wolfSSL_get_options(ssl);
// check mask
```

```
long wolfSSL_set_tlsext_debug_arg(
    WOLFSSL * ssl,
    void * arg
)
```

This is used to set the debug argument passed around.

Parameters:

- **ssl** WOLFSSL structure to set argument in.
- **arg** argument to use.

See:

- wolfSSL_new
- wolfSSL_free

Return:

- SSL_SUCCESS On successful setting argument.
- SSL_FAILURE If an NULL ssl passed in.

Example

```
WOLFSSL* ssl;
void* args;
int ret;
// create ssl object
ret  = wolfSSL_set_tlsext_debug_arg(ssl, args);
// check ret value
```

```
long wolfSSL_get_verify_result(
    const WOLFSSL * ssl
)
```

This is used to get the results after trying to verify the peer's certificate.

Parameters:

- **ssl** WOLFSSL structure to get verification results from.

See:

- wolfSSL_new
- wolfSSL_free

Return:

- X509_V_OK On successful verification.
- SSL_FAILURE If an NULL ssl passed in.

Example

```
WOLFSSL* ssl;
long ret;
// attempt/complete handshake
ret  = wolfSSL_get_verify_result(ssl);
// check ret value
```

```
int wolfSSL_CTX_allow_anon_cipher(
    WOLFSSL_CTX *
)
```

This function enables the havAnon member of the CTX structure if HAVE_ANON is defined during compilation.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().

See: none

Return:

- SSL_SUCCESS returned if the function executed successfully and the haveAnnon member of the CTX is set to 1.
- SSL_FAILURE returned if the CTX structure was NULL.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
WOLFSSL* ssl = wolfSSL_new(ctx);
...
#ifdef HAVE_ANON
if(cipherList == NULL){
    wolfSSL_CTX_allow_anon_cipher(ctx);
    if(wolfSSL_CTX_set_cipher_list(ctx, “ADH_AES128_SHA”) != SSL_SUCCESS){
        // failure case
    }
}
#endif
```

```
WOLFSSL_METHOD * wolfSSLv23_server_method(
    void
)
```

The wolfSSLv23_server_method() function is used to indicate that the application is a server and will support clients connecting with protocol version from SSL 3.0 - TLS 1.3. This function allocates memory for and initializes a new WOLFSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().

Parameters:

- **none** No parameters

See:

- wolfSSLv3_server_method
- wolfTLSv1_server_method

- wolfTLSv1_1_server_method
- wolfTLSv1_2_server_method
- wolfTLSv1_3_server_method
- wolfDTLSv1_server_method
- wolfSSL_CTX_new

Return:

- pointer If successful, the call will return a pointer to the newly created WOLFSSL_METHOD structure.
- Failure If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Example

```
WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfSSLv23_server_method();
if (method == NULL) {
    // unable to get method
}

ctx = wolfSSL_CTX_new(method);
...
```

```
int wolfSSL_state(
    WOLFSSL * ssl
)
```

This is used to get the internal error state of the WOLFSSL structure.

See:

- wolfSSL_new
- wolfSSL_free

Return:

- wolfssl_error returns ssl error state, usually a negative
- BAD_FUNC_ARG if ssl is NULL.
- ssl WOLFSSL structure to get state from.

Example

```
WOLFSSL* ssl;
int ret;
// create ssl object
ret  = wolfSSL_state(ssl);
// check ret value
```

```
int wolfSSL_check_domain_name(
    WOLFSSL * ssl,
    const char * dn
)
```

wolfSSL by default checks the peer certificate for a valid date range and a verified signature. Calling this function before wolfSSL_connect() will add a domain name check to the list of checks to perform. dn holds the domain name to check against the peer certificate when it's received.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **dn** domain name to check against the peer certificate when received.

See: none

Return:

- SSL_SUCCESS upon success.
- SSL_FAILURE will be returned if a memory error was encountered.

Example

```
int ret = 0;
WOLFSSL* ssl;
char* domain = (char*) "www.yassl.com";
...

ret = wolfSSL_check_domain_name(ssl, domain);
if (ret != SSL_SUCCESS) {
   // failed to enable domain name check
}
```

```
int wolfSSL_set_compression(
    WOLFSSL * ssl
)
```

Turns on the ability to use compression for the SSL connection. Both sides must have compression turned on otherwise compression will not be used. The zlib library performs the actual data compression. To compile into the library use –with-libz for the configure system and define HAVE_LIBZ otherwise. Keep in mind that while compressing data before sending decreases the actual size of the messages being sent and received, the amount of data saved by compression usually takes longer in time to analyze than it does to send it raw on all but the slowest of networks.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().

See: none

Return:

- SSL_SUCCESS upon success.
- NOT_COMPILED_IN will be returned if compression support wasn't built into the library.

Example

```
int ret = 0;
WOLFSSL* ssl = 0;
...
ret = wolfSSL_set_compression(ssl);
if (ret == SSL_SUCCESS) {
    // successfully enabled compression for SSL session
}
```

```
int wolfSSL_set_timeout(
    WOLFSSL * ssl,
    unsigned int to
)
```

This function sets the SSL session timeout value in seconds.

Parameters:

- **ssl** pointer to the SSL object, created with wolfSSL_new().
- **to** value, in seconds, used to set the SSL session timeout.

See:

- wolfSSL_get1_session
- wolfSSL_set_session

Return:

- SSL_SUCCESS will be returned upon successfully setting the session.
- BAD_FUNC_ARG will be returned if ssl is NULL.

Example

```
int ret = 0;
WOLFSSL* ssl = 0;
...

ret = wolfSSL_set_timeout(ssl, 500);
if (ret != SSL_SUCCESS) {
    // failed to set session timeout value
```

```
}
...
```

```
int wolfSSL_CTX_set_timeout(
    WOLFSSL_CTX * ctx,
    unsigned int to
)
```

This function sets the timeout value for SSL sessions, in seconds, for the specified SSL context.

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().
- **to** session timeout value in seconds.

See:

- wolfSSL_flush_sessions
- wolfSSL_get1_session
- wolfSSL_set_session
- wolfSSL_get_sessionID
- wolfSSL_CTX_set_session_cache_mode

Return:

- the previous timeout value, if WOLFSSL_ERROR_CODE_OPENSSL is
- defined on success. If not defined, SSL_SUCCESS will be returned.
- BAD_FUNC_ARG will be returned when the input context (ctx) is null.

Example

```
WOLFSSL_CTX*    ctx    = 0;
...
ret = wolfSSL_CTX_set_timeout(ctx, 500);
if (ret != SSL_SUCCESS) {
    // failed to set session timeout value
}
```

```
int wolfSSL_CTX_UnloadCAs(
    WOLFSSL_CTX *
)
```

This function unloads the CA signer list and frees the whole signer table.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().

See:

- wolfSSL_CertManagerUnloadCAs
- LockMutex
- UnlockMutex

Return:

- SSL_SUCCESS returned on successful execution of the function.
- BAD_FUNC_ARG returned if the WOLFSSL_CTX struct is NULL or there are otherwise unpermitted argument values passed in a subroutine.
- BAD_MUTEX_E returned if there was a mutex error. The LockMutex() did not return 0.

Example

```
WOLFSSL_METHOD method = wolfTLSv1_2_client_method();
WOLFSSL_CTX* ctx = WOLFSSL_CTX_new(method);
…
if(wolfSSL_CTX_UnloadCAs(ctx) != SSL_SUCCESS){
    // The function did not unload CAs
}
```

```
int wolfSSL_CTX_UnloadIntermediateCerts(
    WOLFSSL_CTX * ctx
)
```

This function unloads intermediate certificates added to the CA signer list and frees them.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().

See:

- wolfSSL_CTX_UnloadCAs
- wolfSSL_CertManagerUnloadIntermediateCerts

Return:

- SSL_SUCCESS returned on successful execution of the function.
- BAD_FUNC_ARG returned if the WOLFSSL_CTX struct is NULL or there are otherwise unpermitted argument values passed in a subroutine.
- BAD_STATE_E returned if the WOLFSSL_CTX has a reference count > 1.
- BAD_MUTEX_E returned if there was a mutex error. The LockMutex() did not return 0.

Example

```
WOLFSSL_METHOD method = wolfTLSv1_2_client_method();
WOLFSSL_CTX* ctx = WOLFSSL_CTX_new(method);
…
if(wolfSSL_CTX_UnloadIntermediateCerts(ctx) != NULL){
    // The function did not unload CAs
}
```

```
int wolfSSL_CTX_Unload_trust_peers(
    WOLFSSL_CTX *
)
```

This function is used to unload all previously loaded trusted peer certificates. Feature is enabled by defining the macro WOLFSSL_TRUST_PEER_CERT.

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().

See:

- wolfSSL_CTX_trust_peer_buffer
- wolfSSL_CTX_trust_peer_cert

Return:

- SSL_SUCCESS upon success.
- BAD_FUNC_ARG will be returned if ctx is NULL.
- SSL_BAD_FILE will be returned if the file doesn't exist, can't be read, or is corrupted.
- MEMORY_E will be returned if an out of memory condition occurs.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx;
...
ret = wolfSSL_CTX_Unload_trust_peers(ctx);
if (ret != SSL_SUCCESS) {
    // error unloading trusted peer certs
}
...
```

```
int wolfSSL_CTX_trust_peer_buffer(
    WOLFSSL_CTX * ctx,
    const unsigned char * in,
    long sz,
    int format
)
```

This function loads a certificate to use for verifying a peer when performing a TLS/SSL handshake. The peer certificate sent during the handshake is compared by using the SKID when available and the signature. If these two things do not match then any loaded CAs are used. Is the same functionality as wolfSSL_CTX_trust_peer_cert except is from a buffer instead of a file. Feature is enabled by defining the macro WOLFSSL_TRUST_PEER_CERT Please see the examples for proper usage.

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().
- **buffer** pointer to the buffer containing certificates.
- **sz** length of the buffer input.
- **type** type of certificate being loaded i.e. SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM.

See:

- wolfSSL_CTX_load_verify_buffer
- wolfSSL_CTX_use_certificate_file
- wolfSSL_CTX_use_PrivateKey_file
- wolfSSL_CTX_use_certificate_chain_file
- wolfSSL_CTX_trust_peer_cert
- wolfSSL_CTX_Unload_trust_peers
- wolfSSL_use_certificate_file
- wolfSSL_use_PrivateKey_file
- wolfSSL_use_certificate_chain_file

Return:

- SSL_SUCCESS upon success
- SSL_FAILURE will be returned if ctx is NULL, or if both file and type are invalid.
- SSL_BAD_FILETYPE will be returned if the file is the wrong format.
- SSL_BAD_FILE will be returned if the file doesn't exist, can't be read, or is corrupted.
- MEMORY_E will be returned if an out of memory condition occurs.
- ASN_INPUT_E will be returned if Base16 decoding fails on the file.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx;
...

ret = wolfSSL_CTX_trust_peer_buffer(ctx, bufferPtr, bufferSz,
SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
// error loading trusted peer cert
}
...
```

```
int wolfSSL_CTX_set_group_messages(
    WOLFSSL_CTX *
)
```

This function turns on grouping of handshake messages where possible.

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().

See:

- wolfSSL_set_group_messages
- wolfSSL_CTX_new

Return:

- SSL_SUCCESS will be returned upon success.
- BAD_FUNC_ARG will be returned if the input context is null.

Example

```
WOLFSSL_CTX* ctx = 0;
...
ret = wolfSSL_CTX_set_group_messages(ctx);
if (ret != SSL_SUCCESS) {
    // failed to set handshake message grouping
}
```

```
int wolfSSL_set_group_messages(
    WOLFSSL *
)
```

This function turns on grouping of handshake messages where possible.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().

See:

- wolfSSL_CTX_set_group_messages
- wolfSSL_new

Return:

- SSL_SUCCESS will be returned upon success.
- BAD_FUNC_ARG will be returned if the input context is null.

Example

```
WOLFSSL* ssl = 0;
...
ret = wolfSSL_set_group_messages(ssl);
if (ret != SSL_SUCCESS) {
// failed to set handshake message grouping
}
```

```
int wolfSSL_CTX_SetMinVersion(
    WOLFSSL_CTX * ctx,
    int version
)
```

This function sets the minimum downgrade version allowed. Applicable only when the connection allows downgrade using (wolfSSLv23_client_method or wolfSSLv23_server_method).

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().
- **version** an integer representation of the version to be set as the minimum: WOLFSSL_SSLV3 = 0, WOLFSSL_TLSV1 = 1, WOLFSSL_TLSV1_1 = 2 or WOLFSSL_TLSV1_2 = 3.

See: SetMinVersionHelper

Return:

- SSL_SUCCESS returned if the function returned without error and the minimum version is set.
- BAD_FUNC_ARG returned if the WOLFSSL_CTX structure was NULL or if the minimum version is not supported.

Example

```
WOLFSSL_CTX* ctx = WOLFSSL_CTX_new( protocol method );
WOLFSSL* ssl = WOLFSSL_new(ctx);
int version; // macrop representation
…
if(wolfSSL_CTX_SetMinVersion(ssl->ctx, version) != SSL_SUCCESS){
    // Failed to set min version
}
```

```
int wolfSSL_SetVersion(
    WOLFSSL * ssl,
    int version
)
```

This function sets the SSL/TLS protocol version for the specified SSL session (WOLFSSL object) using the version as specified by version. This will override the protocol setting for the SSL session (ssl) - originally defined and set by the SSL context (wolfSSL_CTX_new()) method type.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **version** SSL/TLS protocol version. Possible values include WOLFSSL_SSLV3, WOLFSSL_TLSV1, WOLFSSL_TLSV1_1, WOLFSSL_TLSV1_2.

See: wolfSSL_CTX_new

Return:

- SSL_SUCCESS upon success.
- BAD_FUNC_ARG will be returned if the input SSL object is NULL or an incorrect protocol version is given for version.

Example

```
int ret = 0;
WOLFSSL* ssl;
...

ret = wolfSSL_SetVersion(ssl, WOLFSSL_TLSV1);
if (ret != SSL_SUCCESS) {
    // failed to set SSL session protocol version
}
```

```
int wolfSSL_UseALPN(
    WOLFSSL * ssl,
    char * protocol_name_list,
    unsigned int protocol_name_listSz,
    unsigned char options
)
```

Setup ALPN use for a wolfSSL session.

Parameters:

- **ssl** The wolfSSL session to use.
- **protocol_name_list** List of protocol names to use. Comma delimited string is required.
- **protocol_name_listSz** Size of the list of protocol names.
- **options** WOLFSSL_ALPN_CONTINUE_ON_MISMATCH or WOLFSSL_ALPN_FAILED_ON_MISMATCH.

See: TLSX_UseALPN

Return:

- WOLFSSL_SUCCESS: upon success.
- BAD_FUNC_ARG Returned if ssl or protocol_name_list is null or protocol_name_listSz is too large or options contain something not supported.
- MEMORY_ERROR Error allocating memory for protocol list.
- SSL_FAILURE upon failure.

Example

```
wolfSSL_Init();
WOLFSSL_CTX* ctx;
WOLFSSL* ssl;
WOLFSSL_METHOD method = // Some wolfSSL method
ctx = wolfSSL_CTX_new(method);
ssl = wolfSSL_new(ctx);

char alpn_list[] = {};

if (wolfSSL_UseALPN(ssl, alpn_list, sizeof(alpn_list),
    WOLFSSL_APN_FAILED_ON_MISMATCH) != WOLFSSL_SUCCESS)
{
   // Error setting session ticket
}
```

```
int wolfSSL_CTX_UseSessionTicket(
    WOLFSSL_CTX * ctx
)
```

This function sets wolfSSL context to use a session ticket.

Parameters:

- **ctx** The WOLFSSL_CTX structure to use.

See: TLSX_UseSessionTicket

Return:

- SSL_SUCCESS Function executed successfully.
- BAD_FUNC_ARG Returned if ctx is null.
- MEMORY_E Error allocating memory in internal function.

Example

```
wolfSSL_Init();
WOLFSSL_CTX* ctx;
WOLFSSL_METHOD method = // Some wolfSSL method ;
ctx = wolfSSL_CTX_new(method);

if(wolfSSL_CTX_UseSessionTicket(ctx) != SSL_SUCCESS)
{
    // Error setting session ticket
}
```

```
int wolfSSL_check_private_key(
    const WOLFSSL * ssl
)
```

This function checks that the private key is a match with the certificate being used.

Parameters:

- **ssl** WOLFSSL structure to check.

See:

- wolfSSL_new
- wolfSSL_free

Return:

- SSL_SUCCESS On successfully match.
- SSL_FAILURE If an error case was encountered.
- <0 All error cases other than SSL_FAILURE are negative values.

Example

```
WOLFSSL* ssl;
int ret;
// create and set up ssl
ret  = wolfSSL_check_private_key(ssl);
// check ret value
```

```
int wolfSSL_use_certificate(
    WOLFSSL * ssl,
    WOLFSSL_X509 * x509
)
```

his is used to set the certificate for WOLFSSL structure to use during a handshake.

Parameters:

- **ssl** WOLFSSL structure to set certificate in.
- **x509** certificate to use.

See:

- wolfSSL_new
- wolfSSL_free

Return:

- SSL_SUCCESS On successful setting argument.
- SSL_FAILURE If a NULL argument passed in.

Example

```
 WOLFSSL* ssl;
WOLFSSL_X509* x509
int ret;
// create ssl object and x509
ret  = wolfSSL_use_certificate(ssl, x509);
// check ret value
```

```
int wolfSSL_use_certificate_ASN1(
    WOLFSSL * ssl,
    unsigned char * der,
    int derSz
)
```

This is used to set the certificate for WOLFSSL structure to use during a handshake. A DER formatted buffer is expected.

Parameters:

- **ssl** WOLFSSL structure to set certificate in.
- **der** DER certificate to use.
- **derSz** size of the DER buffer passed in.

See:

- wolfSSL_new
- wolfSSL_free

Return:

- SSL_SUCCESS On successful setting argument.
- SSL_FAILURE If a NULL argument passed in.

Example

```
WOLFSSL* ssl;
unsigned char* der;
int derSz;
int ret;
// create ssl object and set DER variables
ret  = wolfSSL_use_certificate_ASN1(ssl, der, derSz);
// check ret value
```

```
int wolfSSL_SESSION_get_master_key(
    const WOLFSSL_SESSION * ses,
    unsigned char * out,
    int outSz
)
```

This is used to get the master key after completing a handshake.

Parameters:

- **ses** WOLFSSL_SESSION structure to get master secret buffer from.
- **out** buffer to hold data.
- **outSz** size of out buffer passed in. (if 0 function will return max buffer size needed)

See:

- wolfSSL_new
- wolfSSL_free

Return:

- 0 On successfully getting data returns a value greater than 0
- 0 If no random data buffer or an error state returns 0
- max If outSz passed in is 0 then the maximum buffer size needed is returned

Example

```
WOLFSSL_SESSION ssl;
unsigned char* buffer;
size_t bufferSz;
size_t ret;
// complete handshake and get session structure
bufferSz  = wolfSSL_SESSION_get_master_secret(ses, NULL, 0);
buffer = malloc(bufferSz);
ret  = wolfSSL_SESSION_get_master_secret(ses, buffer, bufferSz);
// check ret value
```

```
int wolfSSL_SESSION_get_master_key_length(
    const WOLFSSL_SESSION * ses
)
```

This is used to get the master secret key length.

Parameters:

- **ses** WOLFSSL_SESSION structure to get master secret buffer from.

See:

- wolfSSL_new
- wolfSSL_free

Return: size Returns master secret key size.

Example

```
WOLFSSL_SESSION ssl;
unsigned char* buffer;
size_t bufferSz;
size_t ret;
// complete handshake and get session structure
bufferSz  = wolfSSL_SESSION_get_master_secret_length(ses);
buffer = malloc(bufferSz);
// check ret value
```

```
void wolfSSL_CTX_set_cert_store(
    WOLFSSL_CTX * ctx,
    WOLFSSL_X509_STORE * str
)
```

This is a setter function for the WOLFSSL_X509_STORE structure in ctx.

Parameters:

- **ctx** pointer to the WOLFSSL_CTX structure for setting cert store pointer.
- **str** pointer to the WOLFSSL_X509_STORE to set in ctx.

See:

- wolfSSL_CTX_new
- wolfSSL_CTX_free

Return: none No return.

Example

```
WOLFSSL_CTX ctx;
WOLFSSL_X509_STORE* st;
// setup ctx and st
st = wolfSSL_CTX_set_cert_store(ctx, st);
//use st
```

```
WOLFSSL_X509_STORE * wolfSSL_CTX_get_cert_store(
    WOLFSSL_CTX * ctx
)
```

This is a getter function for the WOLFSSL_X509_STORE structure in ctx.

Parameters:

- **ctx** pointer to the WOLFSSL_CTX structure for getting cert store pointer.

See:

- wolfSSL_CTX_new
- wolfSSL_CTX_free
- wolfSSL_CTX_set_cert_store

Return:

- WOLFSSL_X509_STORE* On successfully getting the pointer.
- NULL Returned if NULL arguments are passed in.

Example

```
WOLFSSL_CTX ctx;
WOLFSSL_X509_STORE* st;
// setup ctx
st = wolfSSL_CTX_get_cert_store(ctx);
//use st
```

```
size_t wolfSSL_get_server_random(
    const WOLFSSL * ssl,
    unsigned char * out,
    size_t outlen
)
```

This is used to get the random data sent by the server during the handshake.

Parameters:

- **ssl** WOLFSSL structure to get clients random data buffer from.
- **out** buffer to hold random data.
- **outSz** size of out buffer passed in. (if 0 function will return max buffer size needed)

See:

- wolfSSL_new
- wolfSSL_free

Return:

- 0 On successfully getting data returns a value greater than 0
- 0 If no random data buffer or an error state returns 0
- max If outSz passed in is 0 then the maximum buffer size needed is returned

Example

```
WOLFSSL ssl;
unsigned char* buffer;
size_t bufferSz;
size_t ret;
bufferSz  = wolfSSL_get_server_random(ssl, NULL, 0);
buffer = malloc(bufferSz);
ret  = wolfSSL_get_server_random(ssl, buffer, bufferSz);
// check ret value
```

```
size_t wolfSSL_get_client_random(
    const WOLFSSL * ssl,
    unsigned char * out,
    size_t outSz
)
```

This is used to get the random data sent by the client during the handshake.

Parameters:

- **ssl** WOLFSSL structure to get clients random data buffer from.
- **out** buffer to hold random data.
- **outSz** size of out buffer passed in. (if 0 function will return max buffer size needed)

See:

- wolfSSL_new
- wolfSSL_free

Return:

- 0 On successfully getting data returns a value greater than 0
- 0 If no random data buffer or an error state returns 0
- max If outSz passed in is 0 then the maximum buffer size needed is returned

Example

```
WOLFSSL ssl;
unsigned char* buffer;
size_t bufferSz;
size_t ret;
bufferSz  = wolfSSL_get_client_random(ssl, NULL, 0);
buffer = malloc(bufferSz);
ret  = wolfSSL_get_client_random(ssl, buffer, bufferSz);
// check ret value
```

```
wc_pem_password_cb * wolfSSL_CTX_get_default_passwd_cb(
    WOLFSSL_CTX * ctx
)
```

This is a getter function for the password callback set in ctx.

Parameters:

- **ctx** WOLFSSL_CTX structure to get call back from.

See:

- wolfSSL_CTX_new
- wolfSSL_CTX_free

Return:

- func On success returns the callback function.
- NULL If ctx is NULL then NULL is returned.

Example

```
WOLFSSL_CTX* ctx;
wc_pem_password_cb cb;
// setup ctx
cb = wolfSSL_CTX_get_default_passwd_cb(ctx);
//use cb
```

```
void * wolfSSL_CTX_get_default_passwd_cb_userdata(
    WOLFSSL_CTX * ctx
)
```

This is a getter function for the password callback user data set in ctx.

Parameters:

- **ctx** WOLFSSL_CTX structure to get user data from.

See:

- wolfSSL_CTX_new
- wolfSSL_CTX_free

Return:

- pointer On success returns the user data pointer.
- NULL If ctx is NULL then NULL is returned.

Example

```
WOLFSSL_CTX* ctx;
void* data;
// setup ctx
data = wolfSSL_CTX_get_default_passwd_cb(ctx);
//use data
```

```
long wolfSSL_CTX_clear_options(
    WOLFSSL_CTX * ctx,
    long opt
)
```

This function resets option bits of WOLFSSL_CTX object.

Parameters:

- **ctx** pointer to the SSL context.

See:

- wolfSSL_CTX_new
- wolfSSL_new
- wolfSSL_free

Return: option new option bits

Example

```
WOLFSSL_CTX* ctx = 0;
...
wolfSSL_CTX_clear_options(ctx, SSL_OP_NO_TLSv1);
```

```
int wolfSSL_set_msg_callback(
    WOLFSSL * ssl,
    SSL_Msg_Cb cb
)
```

This function sets a callback in the ssl. The callback is to observe handshake messages. NULL value of cb resets the callback.

Parameters:

- **ssl** WOLFSSL structure to set callback argument.

See: wolfSSL_set_msg_callback_arg

Return:

- SSL_SUCCESS On success.
- SSL_FAILURE If an NULL ssl passed in.

Example

```
static cb(int write_p, int version, int content_type,
const void *buf, size_t len, WOLFSSL *ssl, void *arg)
…
WOLFSSL* ssl;
ret  = wolfSSL_set_msg_callback(ssl, cb);
// check ret
```

```
int wolfSSL_set_msg_callback_arg(
    WOLFSSL * ssl,
    void * arg
)
```

This function sets associated callback context value in the ssl. The value is handed over to the callback argument.

Parameters:

- **ssl** WOLFSSL structure to set callback argument.

See: wolfSSL_set_msg_callback

Return: none No return.

Example

```
static cb(int write_p, int version, int content_type,
const void *buf, size_t len, WOLFSSL *ssl, void *arg)
…
WOLFSSL* ssl;
ret  = wolfSSL_set_msg_callback(ssl, cb);
// check ret
wolfSSL_set_msg_callback(ssl, arg);
```

```
int wolfSSL_send_hrr_cookie(
    WOLFSSL * ssl,
    const unsigned char * secret,
    unsigned int secretSz
)
```

This function is called on the server side to indicate that a HelloRetryRequest message must contain a Cookie and, in case of using protocol DTLS v1.3, that the handshake will always include a cookie exchange. Please note that when using protocol DTLS v1.3, the cookie exchange is enabled by default. The Cookie holds a hash of the current transcript so that another server process can handle the ClientHello in reply. The secret is used when generating the integrity check on the Cookie data.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **secret** a pointer to a buffer holding the secret. Passing NULL indicates to generate a new random secret.
- **secretSz** Size of the secret in bytes. Passing 0 indicates to use the default size: WC_SHA256_DIGEST_SIZE (or WC_SHA_DIGEST_SIZE when SHA-256 not available).

See:

- wolfSSL_new
- wolfSSL_disable_hrr_cookie

Return:

- BAD_FUNC_ARG if ssl is NULL or not using TLS v1.3.
- SIDE_ERROR if called with a client.
- WOLFSSL_SUCCESS if successful.
- MEMORY_ERROR if allocating dynamic memory for storing secret failed.
- Another -ve value on internal error.

Example

```
int ret;
WOLFSSL* ssl;
char secret[32];
...
ret = wolfSSL__send_hrr_cookie(ssl, secret, sizeof(secret));
if (ret != WOLFSSL_SUCCESS) {
    // failed to set use of Cookie and secret
}
```

```
int wolfSSL_disable_hrr_cookie(
    WOLFSSL * ssl
)
```

This function is called on the server side to indicate that a HelloRetryRequest message must NOT contain a Cookie and that, if using protocol DTLS v1.3, a cookie exchange will not be included in the handshake. Please note that not doing a cookie exchange when using protocol DTLS v1.3 can make the server susceptible to DoS/Amplification attacks.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See: wolfSSL_send_hrr_cookie

Return:

- WOLFSSL_SUCCESS if successful
- BAD_FUNC_ARG if ssl is NULL or not using TLS v1.3
- SIDE_ERROR if invoked on client

```
int wolfSSL_CTX_no_ticket_TLSv13(
    WOLFSSL_CTX * ctx
)
```

This function is called on the server to stop it from sending a resumption session ticket once the handshake is complete.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created with wolfSSL_CTX_new().

See: wolfSSL_no_ticket_TLSv13

Return:

- BAD_FUNC_ARG if ctx is NULL or not using TLS v1.3.
- SIDE_ERROR if called with a client.
- 0 if successful.

Example

```
int ret;
WOLFSSL_CTX* ctx;
...
ret = wolfSSL_CTX_no_ticket_TLSv13(ctx);
if (ret != 0) {
    // failed to set no ticket
}
```

```
int wolfSSL_no_ticket_TLSv13(
    WOLFSSL * ssl
)
```

This function is called on the server to stop it from sending a resumption session ticket once the handshake is complete.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See: wolfSSL_CTX_no_ticket_TLSv13

Return:

- BAD_FUNC_ARG if ssl is NULL or not using TLS v1.3.

- SIDE_ERROR if called with a client.
- 0 if successful.

Example

```
int ret;
WOLFSSL* ssl;
...
ret = wolfSSL_no_ticket_TLSv13(ssl);
if (ret != 0) {
    // failed to set no ticket
}
```

```
int wolfSSL_CTX_no_dhe_psk(
    WOLFSSL_CTX * ctx
)
```

This function is called on a TLS v1.3 wolfSSL context to disallow Diffie-Hellman (DH) style key exchanges when handshakes are using pre-shared keys for authentication.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created with wolfSSL_CTX_new().

See: wolfSSL_no_dhe_psk

Return:

- BAD_FUNC_ARG if ctx is NULL or not using TLS v1.3.
- 0 if successful.

Example

```
int ret;
WOLFSSL_CTX* ctx;
...
ret = wolfSSL_CTX_no_dhe_psk(ctx);
if (ret != 0) {
    // failed to set no DHE for PSK handshakes
}
```

```
int wolfSSL_no_dhe_psk(
    WOLFSSL * ssl
)
```

This function is called on a TLS v1.3 client or server wolfSSL to disallow Diffie-Hellman (DH) style key exchanges when handshakes are using pre-shared keys for authentication.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See: wolfSSL_CTX_no_dhe_psk

Return:

- BAD_FUNC_ARG if ssl is NULL or not using TLS v1.3.
- 0 if successful.

Example

```
int ret;
WOLFSSL* ssl;
...
ret = wolfSSL_no_dhe_psk(ssl);
if (ret != 0) {
    // failed to set no DHE for PSK handshakes
}
```

```
int wolfSSL_CTX_allow_post_handshake_auth(
    WOLFSSL_CTX * ctx
)
```

This function is called on a TLS v1.3 client wolfSSL context to allow a client certificate to be sent post handshake upon request from server. This is useful when connecting to a web server that has some pages that require client authentication and others that don't.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created with wolfSSL_CTX_new().

See:

- wolfSSL_allow_post_handshake_auth
- wolfSSL_request_certificate

Return:

- BAD_FUNC_ARG if ctx is NULL or not using TLS v1.3.
- SIDE_ERROR if called with a server.
- 0 if successful.

Example

```
int ret;
WOLFSSL_CTX* ctx;
...
ret = wolfSSL_allow_post_handshake_auth(ctx);
```

```
if (ret != 0) {
    // failed to allow post handshake authentication
}
```

```
int wolfSSL_allow_post_handshake_auth(
    WOLFSSL * ssl
)
```

This function is called on a TLS v1.3 client wolfSSL to allow a client certificate to be sent post handshake upon request from server. A Post-Handshake Client Authentication extension is sent in the ClientHello. This is useful when connecting to a web server that has some pages that require client authentication and others that don't.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_CTX_allow_post_handshake_auth
- wolfSSL_request_certificate

Return:

- BAD_FUNC_ARG if ssl is NULL or not using TLS v1.3.
- SIDE_ERROR if called with a server.
- 0 if successful.

Example

```
int ret;
WOLFSSL* ssl;
...
ret = wolfSSL_allow_post_handshake_auth(ssl);
if (ret != 0) {
    // failed to allow post handshake authentication
}
```

```
int wolfSSL_CTX_set1_groups_list(
    WOLFSSL_CTX * ctx,
    char * list
)
```

This function sets the list of elliptic curve groups to allow on a wolfSSL context in order of preference. The list is a null-terminated text string, and a colon-delimited list. Call this function to set the key exchange elliptic curve parameters to use with the TLS v1.3 connections.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created with wolfSSL_CTX_new().
- **list** a string that is a colon-delimited list of elliptic curve groups.

See:

- wolfSSL_set1_groups_list
- wolfSSL_CTX_set_groups
- wolfSSL_set_groups
- wolfSSL_UseKeyShare
- wolfSSL_preferred_group

Return:

- WOLFSSL_FAILURE if pointer parameters are NULL, there are more than WOLFSSL_MAX_GROUP_COUNT groups, a group name is not recognized or not using TLS v1.3.
- WOLFSSL_SUCCESS if successful.

Example

```
int ret;
WOLFSSL_CTX* ctx;
const char* list = "P-384:P-256";
...
ret = wolfSSL_CTX_set1_groups_list(ctx, list);
if (ret != WOLFSSL_SUCCESS) {
    // failed to set group list
}
```

```
int wolfSSL_set1_groups_list(
    WOLFSSL * ssl,
    char * list
)
```

This function sets the list of elliptic curve groups to allow on a wolfSSL in order of preference. The list is a null-terminated text string, and a colon-delimited list. Call this function to set the key exchange elliptic curve parameters to use with the TLS v1.3 connections.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **list** a string that is a colon separated list of key exchange groups.

See:

- wolfSSL_CTX_set1_groups_list
- wolfSSL_CTX_set_groups
- wolfSSL_set_groups
- wolfSSL_UseKeyShare
- wolfSSL_preferred_group

Return:

- WOLFSSL_FAILURE if pointer parameters are NULL, there are more than WOLFSSL_MAX_GROUP_COUNT groups, a group name is not recognized or not using TLS v1.3.
- WOLFSSL_SUCCESS if successful.

Example

```
int ret;
WOLFSSL* ssl;
const char* list = "P-384:P-256";
...
ret = wolfSSL_CTX_set1_groups_list(ssl, list);
if (ret != WOLFSSL_SUCCESS) {
    // failed to set group list
}
```

```
int wolfSSL_CTX_set_groups(
    WOLFSSL_CTX * ctx,
    int * groups,
    int count
)
```

This function sets the list of elliptic curve groups to allow on a wolfSSL context in order of preference. The list is an array of group identifiers with the number of identifiers specified in count. Call this function to set the key exchange elliptic curve parameters to use with the TLS v1.3 connections.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created with wolfSSL_CTX_new().
- **groups** a list of key exchange groups by identifier.
- **count** the number of key exchange groups in groups.

See:

- wolfSSL_set_groups
- wolfSSL_UseKeyShare
- wolfSSL_CTX_set_groups
- wolfSSL_set_groups
- wolfSSL_CTX_set1_groups_list
- wolfSSL_set1_groups_list
- wolfSSL_preferred_group

Return:

- BAD_FUNC_ARG if a pointer parameter is null, the number of groups exceeds WOLFSSL_MAX_GROUP_COUNT or not using TLS v1.3.
- WOLFSSL_SUCCESS if successful.

Example

```
int ret;
WOLFSSL_CTX* ctx;
int* groups = { WOLFSSL_ECC_X25519, WOLFSSL_ECC_SECP256R1 };
int count = 2;
...
ret = wolfSSL_CTX_set1_groups_list(ctx, groups, count);
if (ret != WOLFSSL_SUCCESS) {
    // failed to set group list
}
```

```
int wolfSSL_set_groups(
    WOLFSSL * ssl,
    int * groups,
    int count
)
```

This function sets the list of elliptic curve groups to allow on a wolfSSL. The list is an array of group identifiers with the number of identifiers specified in count. Call this function to set the key exchange elliptic curve parameters to use with the TLS v1.3 connections.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **groups** a list of key exchange groups by identifier.
- **count** the number of key exchange groups in groups.

See:

- wolfSSL_CTX_set_groups
- wolfSSL_UseKeyShare
- wolfSSL_CTX_set_groups
- wolfSSL_set_groups
- wolfSSL_CTX_set1_groups_list
- wolfSSL_set1_groups_list
- wolfSSL_preferred_group

Return:

- BAD_FUNC_ARG if a pointer parameter is null, the number of groups exceeds WOLFSSL_MAX_GROUP_COUNT, any of the identifiers are unrecognized or not using TLS v1.3.
- WOLFSSL_SUCCESS if successful.

Example

```
int ret;
WOLFSSL* ssl;
int* groups = { WOLFSSL_ECC_X25519, WOLFSSL_ECC_SECP256R1 };
```

```
int count = 2;
...
ret = wolfSSL_set_groups(ssl, groups, count);
if (ret != WOLFSSL_SUCCESS) {
    // failed to set group list
}
```

```
int wolfSSL_CTX_set_max_early_data(
    WOLFSSL_CTX * ctx,
    unsigned int sz
)
```

This function sets the maximum amount of early data that a TLS v1.3 client or server is willing to exchange using the wolfSSL context. Call this function to limit the amount of early data to process to mitigate replay attacks. Early data is protected by keys derived from those of the connection that the session ticket was sent and therefore will be the same every time a session ticket is used in resumption. The value is included in the session ticket for resumption. A server value of zero indicates no early data is to be sent by client using session tickets. A client value of zero indicates that the client will not send any early data. It is recommended that the number of early data bytes be kept as low as practically possible in the application.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created with wolfSSL_CTX_new().
- **sz** the amount of early data to accept in bytes.

See:

- wolfSSL_set_max_early_data
- wolfSSL_write_early_data
- wolfSSL_read_early_data

Return:

- BAD_FUNC_ARG if ctx is NULL or not using TLS v1.3.
- 0 if successful.

Example

```
int ret;
WOLFSSL_CTX* ctx;
...
ret = wolfSSL_CTX_set_max_early_data(ctx, 128);
if (ret != WOLFSSL_SUCCESS) {
    // failed to set group list
}
```

```
int wolfSSL_set_max_early_data(
    WOLFSSL * ssl,
    unsigned int sz
)
```

This function sets the maximum amount of early data that a TLS v1.3 client or server is willing to exchange. Call this function to limit the amount of early data to process to mitigate replay attacks. Early data is protected by keys derived from those of the connection that the session ticket was sent and therefore will be the same every time a session ticket is used in resumption. The value is included in the session ticket for resumption. A server value of zero indicates no early data is to be sent by client using session tickets. A client value of zero indicates that the client will not send any early data. It is recommended that the number of early data bytes be kept as low as practically possible in the application.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **sz** the amount of early data to accept from client in bytes.

See:

- wolfSSL_CTX_set_max_early_data
- wolfSSL_write_early_data
- wolfSSL_read_early_data

Return:

- BAD_FUNC_ARG if ssl is NULL or not using TLS v1.3.
- 0 if successful.

Example

```
int ret;
WOLFSSL* ssl;
...
ret = wolfSSL_set_max_early_data(ssl, 128);
if (ret != WOLFSSL_SUCCESS) {
    // failed to set group list
}
```

```
void wolfSSL_CTX_set_psk_client_tls13_callback(
    WOLFSSL_CTX * ctx,
    wc_psk_client_tls13_callback cb
)
```

This function sets the Pre-Shared Key (PSK) client side callback for TLS v1.3 connections. The callback is used to find a PSK identity and return its key and the name of the cipher to use for the handshake. The function sets the client_psk_tls13_cb member of the WOLFSSL_CTX structure.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created with wolfSSL_CTX_new().
- **cb** a Pre-Shared Key (PSK) callback for a TLS 1.3 client.

See:

- wolfSSL_set_psk_client_tls13_callback
- wolfSSL_CTX_set_psk_server_tls13_callback
- wolfSSL_set_psk_server_tls13_callback

Example

```
WOLFSSL_CTX* ctx;
...
wolfSSL_CTX_set_psk_client_tls13_callback(ctx, my_psk_client_tls13_cb);
```

```
void wolfSSL_set_psk_client_tls13_callback(
    WOLFSSL * ssl,
    wc_psk_client_tls13_callback cb
)
```

This function sets the Pre-Shared Key (PSK) client side callback for TLS v1.3 connections. The callback is used to find a PSK identity and return its key and the name of the cipher to use for the handshake. The function sets the client_psk_tls13_cb member of the options field in WOLFSSL structure.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **cb** a Pre-Shared Key (PSK) callback for a TLS 1.3 client.

See:

- wolfSSL_CTX_set_psk_client_tls13_callback
- wolfSSL_CTX_set_psk_server_tls13_callback
- wolfSSL_set_psk_server_tls13_callback

Example

```
WOLFSSL* ssl;
...
wolfSSL_set_psk_client_tls13_callback(ssl, my_psk_client_tls13_cb);
```

```
void wolfSSL_CTX_set_psk_server_tls13_callback(
    WOLFSSL_CTX * ctx,
    wc_psk_server_tls13_callback cb
)
```

This function sets the Pre-Shared Key (PSK) server side callback for TLS v1.3 connections. The callback is used to find a PSK identity and return its key and the name of the cipher to use for the handshake. The function sets the server_psk_tls13_cb member of the WOLFSSL_CTX structure.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created with wolfSSL_CTX_new().
- **cb** a Pre-Shared Key (PSK) callback for a TLS 1.3 server.

See:

- wolfSSL_CTX_set_psk_client_tls13_callback
- wolfSSL_set_psk_client_tls13_callback
- wolfSSL_set_psk_server_tls13_callback

Example

```
WOLFSSL_CTX* ctx;
...
wolfSSL_CTX_set_psk_server_tls13_callback(ctx, my_psk_client_tls13_cb);
```

```
void wolfSSL_set_psk_server_tls13_callback(
    WOLFSSL * ssl,
    wc_psk_server_tls13_callback cb
)
```

This function sets the Pre-Shared Key (PSK) server side callback for TLS v1.3 connections. The callback is used to find a PSK identity and return its key and the name of the cipher to use for the handshake. The function sets the server_psk_tls13_cb member of the options field in WOLFSSL structure.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **cb** a Pre-Shared Key (PSK) callback for a TLS 1.3 server.

See:

- wolfSSL_CTX_set_psk_client_tls13_callback
- wolfSSL_set_psk_client_tls13_callback
- wolfSSL_CTX_set_psk_server_tls13_callback

Example

```
WOLFSSL* ssl;
...
wolfSSL_set_psk_server_tls13_callback(ssl, my_psk_server_tls13_cb);
```

```
int wolfSSL_UseKeyShare(
    WOLFSSL * ssl,
    word16 group
)
```

This function creates a key share entry from the group including generating a key pair. The KeyShare extension contains all the generated public keys for key exchange. If this function is called, then only the groups specified will be included. Call this function when a preferred group has been previously established for the server.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **group** a key exchange group identifier.

See:

- wolfSSL_preferred_group
- wolfSSL_CTX_set1_groups_list
- wolfSSL_set1_groups_list
- wolfSSL_CTX_set_groups
- wolfSSL_set_groups
- wolfSSL_NoKeyShares

Return:

- BAD_FUNC_ARG if ssl is NULL.
- MEMORY_E when dynamic memory allocation fails.
- WOLFSSL_SUCCESS if successful.

Example

```
int ret;
WOLFSSL* ssl;
...
ret = wolfSSL_UseKeyShare(ssl, WOLFSSL_ECC_X25519);
if (ret != WOLFSSL_SUCCESS) {
    // failed to set key share
}
```

```
int wolfSSL_NoKeyShares(
    WOLFSSL * ssl
)
```

This function is called to ensure no key shares are sent in the ClientHello. This will force the server to respond with a HelloRetryRequest if a key exchange is required in the handshake. Call this function when the expected key exchange group is not known and to avoid the generation of keys unnecessarily. Note that an extra round-trip will be required to complete the handshake when a key exchange is required.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See: wolfSSL_UseKeyShare

Return:

- BAD_FUNC_ARG if ssl is NULL.
- SIDE_ERROR if called with a server.
- WOLFSSL_SUCCESS if successful.

Example

```
int ret;
WOLFSSL* ssl;
...
ret = wolfSSL_NoKeyShares(ssl);
if (ret != WOLFSSL_SUCCESS) {
    // failed to set no key shares
}
```

```
WOLFSSL_METHOD * wolfTLSv1_3_server_method_ex(
    void * heap
)
```

This function is used to indicate that the application is a server and will only support the TLS 1.3 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().

Parameters:

- **heap** a pointer to a buffer that the static memory allocator will use during dynamic memory allocation.

See:

- wolfSSLv3_server_method
- wolfTLSv1_server_method
- wolfTLSv1_1_server_method
- wolfTLSv1_2_server_method
- wolfTLSv1_3_server_method
- wolfDTLSv1_server_method
- wolfSSLv23_server_method
- wolfSSL_CTX_new

Return:

- If successful, the call will return a pointer to the newly created WOLFSSL_METHOD structure.
- FAIL If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfTLSv1_3_server_method_ex(NULL);
if (method == NULL) {
    // unable to get method
}

ctx = wolfSSL_CTX_new(method);
...
```

```
WOLFSSL_METHOD * wolfTLSv1_3_client_method_ex(
    void * heap
)
```

This function is used to indicate that the application is a client and will only support the TLS 1.3 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().

Parameters:

- **heap** a pointer to a buffer that the static memory allocator will use during dynamic memory allocation.

See:

- wolfSSLv3_client_method
- wolfTLSv1_client_method
- wolfTLSv1_1_client_method
- wolfTLSv1_2_client_method
- wolfTLSv1_3_client_method
- wolfDTLSv1_client_method
- wolfSSLv23_client_method
- wolfSSL_CTX_new

Return:

- If successful, the call will return a pointer to the newly created WOLFSSL_METHOD structure.
- FAIL If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Example

```
#include <wolfssl/ssl.h>
```

```
WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfTLSv1_3_client_method_ex(NULL);
if (method == NULL) {
    // unable to get method
}

ctx = wolfSSL_CTX_new(method);
...
```

```
WOLFSSL_METHOD * wolfTLSv1_3_server_method(
    void
)
```

This function is used to indicate that the application is a server and will only support the TLS 1.3 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().

See:

- wolfSSLv3_server_method
- wolfTLSv1_server_method
- wolfTLSv1_1_server_method
- wolfTLSv1_2_server_method
- wolfTLSv1_3_server_method_ex
- wolfDTLSv1_server_method
- wolfSSLv23_server_method
- wolfSSL_CTX_new

Return:

- If successful, the call will return a pointer to the newly created WOLFSSL_METHOD structure.
- FAIL If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfTLSv1_3_server_method();
if (method == NULL) {
    // unable to get method
}

ctx = wolfSSL_CTX_new(method);
...
```

```
WOLFSSL_METHOD * wolfTLSv1_3_client_method(
    void
)
```

This function is used to indicate that the application is a client and will only support the TLS 1.3 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().

See:

- wolfSSLv3_client_method
- wolfTLSv1_client_method
- wolfTLSv1_1_client_method
- wolfTLSv1_2_client_method
- wolfTLSv1_3_client_method_ex
- wolfDTLSv1_client_method
- wolfSSLv23_client_method
- wolfSSL_CTX_new

Return:

- If successful, the call will return a pointer to the newly created WOLFSSL_METHOD structure.
- FAIL If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfTLSv1_3_client_method();
if (method == NULL) {
    // unable to get method
}

ctx = wolfSSL_CTX_new(method);
...
```

```
WOLFSSL_METHOD * wolfTLSv1_3_method_ex(
    void * heap
)
```

This function returns a WOLFSSL_METHOD similar to wolfTLSv1_3_client_method except that it is not determined which side yet (server/client).

Parameters:

- **heap** a pointer to a buffer that the static memory allocator will use during dynamic memory allocation.

See:

- wolfSSL_new
- wolfSSL_free

Return:

- WOLFSSL_METHOD On successful creations returns a WOLFSSL_METHOD pointer
- NULL Null if memory allocation error or failure to create method

Example

```
WOLFSSL* ctx;
ctx  = wolfSSL_CTX_new(wolfTLSv1_3_method_ex(NULL));
// check ret value
```

```
WOLFSSL_METHOD * wolfTLSv1_3_method(
    void
)
```

This function returns a WOLFSSL_METHOD similar to wolfTLSv1_3_client_method except that it is not determined which side yet (server/client).

See:

- wolfSSL_new
- wolfSSL_free

Return:

- WOLFSSL_METHOD On successful creations returns a WOLFSSL_METHOD pointer
- NULL Null if memory allocation error or failure to create method

Example

```
WOLFSSL* ctx;
ctx  = wolfSSL_CTX_new(wolfTLSv1_3_method());
// check ret value
```

```
int wolfSSL_CTX_set_client_cert_type(
    WOLFSSL_CTX * ctx,
    const char * buf,
    int len
)
```

In case this function is called in a client side, set certificate types that can be sent to its peer. In case called in a server side, set certificate types that can be acceptable from its peer. Put cert types in the buffer with prioritised order. To reset the settings to default, pass NULL for the buffer or pass zero for len. By default, certificate type is only X509. In case both side intend to send or accept “Raw public key” cert, WOLFSSL_CERT_TYPE_RPK should be included in the buffer to set.

Parameters:

- **ctx** WOLFSSL_CTX object pointer
- **buf** A buffer where certificate types are stored
- **len** buf size in bytes (same as number of certificate types included) *Example*

```
int ret;
WOLFSSL_CTX* ctx;
char buf[] = {WOLFSSL_CERT_TYPE_RPK, WOLFSSL_CERT_TYPE_X509};
int len = sizeof(buf)/sizeof(char);
...

ret = wolfSSL_CTX_set_client_cert_type(ctx, buf, len);
```

See:

- wolfSSL_set_client_cert_type
- wolfSSL_CTX_set_server_cert_type
- wolfSSL_set_server_cert_type
- wolfSSL_get_negotiated_client_cert_type
- wolfSSL_get_negotiated_server_cert_type

Return:

- WOLFSSL_SUCCESS if cert types set successfully
- BAD_FUNC_ARG if NULL was passed for ctx, illegal value was specified as cert type, buf size exceed MAX_CLIENT_CERT_TYPE_CNT was specified or a duplicate value is found in buf.

```
int wolfSSL_CTX_set_server_cert_type(
    WOLFSSL_CTX * ctx,
    const char * buf,
    int len
)
```

In case this function is called in a server side, set certificate types that can be sent to its peer. In case called in a client side, set certificate types that can be acceptable from its peer. Put cert types in the buffer with prioritised order. To reset the settings to default, pass NULL for the buffer or pass zero for len. By default, certificate type is only X509. In case both side intend to send or accept “Raw public key” cert, WOLFSSL_CERT_TYPE_RPK should be included in the buffer to set.

Parameters:

- **ctx** WOLFSSL_CTX object pointer

- **buf** A buffer where certificate types are stored
- **len** buf size in bytes (same as number of certificate types included) *Example*

```
int ret;
WOLFSSL_CTX* ctx;
char buf[] = {WOLFSSL_CERT_TYPE_RPK, WOLFSSL_CERT_TYPE_X509};
int len = sizeof(buf)/sizeof(char);
...

ret = wolfSSL_CTX_set_server_cert_type(ctx, buf, len);
```

See:

- wolfSSL_set_client_cert_type
- wolfSSL_CTX_set_client_cert_type
- wolfSSL_set_server_cert_type
- wolfSSL_get_negotiated_client_cert_type
- wolfSSL_get_negotiated_server_cert_type

Return:

- WOLFSSL_SUCCESS if cert types set successfully
- BAD_FUNC_ARG if NULL was passed for ctx, illegal value was specified as cert type, buf size exceed MAX_SERVER_CERT_TYPE_CNT was specified or a duplicate value is found in buf.

```
int wolfSSL_set_client_cert_type(
    WOLFSSL * ssl,
    const char * buf,
    int len
)
```

In case this function is called in a client side, set certificate types that can be sent to its peer. In case called in a server side, set certificate types that can be acceptable from its peer. Put cert types in the buffer with prioritised order. To reset the settings to default, pass NULL for the buffer or pass zero for len. By default, certificate type is only X509. In case both side intend to send or accept “Raw public key” cert, WOLFSSL_CERT_TYPE_RPK should be included in the buffer to set.

Parameters:

- **ssl** WOLFSSL object pointer
- **buf** A buffer where certificate types are stored
- **len** buf size in bytes (same as number of certificate types included) *Example*

```
int ret;
WOLFSSL* ssl;
char buf[] = {WOLFSSL_CERT_TYPE_RPK, WOLFSSL_CERT_TYPE_X509};
int len = sizeof(buf)/sizeof(char);
...
```

```
ret = wolfSSL_set_client_cert_type(ssl, buf, len);
```

See:

- wolfSSL_CTX_set_client_cert_type
- wolfSSL_CTX_set_server_cert_type
- wolfSSL_set_server_cert_type
- wolfSSL_get_negotiated_client_cert_type
- wolfSSL_get_negotiated_server_cert_type

Return:

- WOLFSSL_SUCCESS if cert types set successfully
- BAD_FUNC_ARG if NULL was passed for ctx, illegal value was specified as cert type, buf size exceed MAX_CLIENT_CERT_TYPE_CNT was specified or a duplicate value is found in buf.

```
int wolfSSL_set_server_cert_type(
    WOLFSSL * ssl,
    const char * buf,
    int len
)
```

In case this function is called in a server side, set certificate types that can be sent to its peer. In case called in a client side, set certificate types that can be acceptable from its peer. Put cert types in the buffer with prioritised order. To reset the settings to default, pass NULL for the buffer or pass zero for len. By default, certificate type is only X509. In case both side intend to send or accept “Raw public key” cert, WOLFSSL_CERT_TYPE_RPK should be included in the buffer to set.

Parameters:

- **ctx** WOLFSSL_CTX object pointer
- **buf** A buffer where certificate types are stored
- **len** buf size in bytes (same as number of certificate types included) *Example*

```
int ret;
WOLFSSL* ssl;
char buf[] = {WOLFSSL_CERT_TYPE_RPK, WOLFSSL_CERT_TYPE_X509};
int len = sizeof(buf)/sizeof(char);
...

ret = wolfSSL_set_server_cert_type(ssl, buf, len);
```

See:

- wolfSSL_set_client_cert_type
- wolfSSL_CTX_set_server_cert_type
- wolfSSL_set_server_cert_type
- wolfSSL_get_negotiated_client_cert_type

- wolfSSL_get_negotiated_server_cert_type

Return:

- WOLFSSL_SUCCESS if cert types set successfully
- BAD_FUNC_ARG if NULL was passed for ctx, illegal value was specified as cert type, buf size exceed MAX_SERVER_CERT_TYPE_CNT was specified or a duplicate value is found in buf.

```
void * wolfSSL_GetCookieCtx(
    WOLFSSL * ssl
)
```

This function returns the IOCB_CookieCtx member of the WOLFSSL structure.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_SetCookieCtx
- wolfSSL_CTX_SetGenCookie

Return:

- pointer The function returns a void pointer value stored in the IOCB_CookieCtx.
- NULL if the WOLFSSL struct is NULL

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
void* cookie;
...
cookie = wolfSSL_GetCookieCtx(ssl);
if(cookie != NULL){
// You have the cookie
}
```

```
int wolfSSL_SetIO_ISOTP(
    WOLFSSL * ssl,
    isotp_wolfssl_ctx * ctx,
    can_recv_fn recv_fn,
    can_send_fn send_fn,
    can_delay_fn delay_fn,
    word32 receive_delay,
    char * receive_buffer,
    int receive_buffer_size,
```

```
    void * arg
)
```

This function sets up the ISO-TP context if wolfSSL, for use when wolfSSL is compiled with WOLFSSL_ISOTP.

Parameters:

- **ssl** the wolfSSL context
- **ctx** a user created ISOTP context which this function initializes
- **recv_fn** a user CAN bus receive callback
- **send_fn** a user CAN bus send callback
- **delay_fn** a user microsecond granularity delay function
- **receive_delay** a set amount of microseconds to delay each CAN bus packet
- **receive_buffer** a user supplied buffer to receive data, recommended that is allocated to ISOTP_DEFAULT_BUFFER_SIZE bytes
- **receive_buffer_size** - The size of receive_buffer
- **arg** an arbitrary pointer sent to recv_fn and send_fn

Return: 0 on success, WOLFSSL_CBIO_ERR_GENERAL on failure

Example

```
struct can_info can_con_info;
isotp_wolfssl_ctx isotp_ctx;
char *receive_buffer = malloc(ISOTP_DEFAULT_BUFFER_SIZE);
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(method);
WOLFSSL* ssl = wolfSSL_new(ctx);
...
wolfSSL_SetIO_ISOTP(ssl, &isotp_ctx, can_receive, can_send, can_delay, 0,
        receive_buffer, ISOTP_DEFAULT_BUFFER_SIZE, &can_con_info);
```

```
void wolfSSL_SSLDisableRead(
    WOLFSSL * ssl
)
```

This function disables reading from the IO layer.

Parameters:

- **ssl** the wolfSSL context

See:

- wolfSSL_CTX_SetIORecv
- wolfSSL_SSLSetIORecv
- wolfSSL_SSLEnableRead

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(method);
WOLFSSL* ssl = wolfSSL_new(ctx);
wolfSSL_SSLDisableRead(ssl);
```

```
void wolfSSL_SSLEnableRead(
    WOLFSSL * ssl
)
```

This function enables reading from the IO layer. Reading is enabled by default and should be used to undo wolfSSL_SSLDisableRead();.

Parameters:

- **ssl** the wolfSSL context

See:

- wolfSSL_CTX_SetIORecv
- wolfSSL_SSLSetIORecv
- wolfSSL_SSLEnableRead

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(method);
WOLFSSL* ssl = wolfSSL_new(ctx);
wolfSSL_SSLDisableRead(ssl);
...
wolfSSL_SSLEnableRead(ssl);
```

A.7 wolfSSL Error Handling and Reporting

A.6.2.112 function wolfSSL_SSLEnableRead

A.7.1 Functions

	Name
int	**wolfSSL_Debugging_ON**(void)If logging has been enabled at build time this function turns on logging at runtime. To enable logging at build time use –enable-debug or define DEBUG_WOLFSSL.
void	**wolfSSL_Debugging_OFF**(void)This function turns off runtime logging messages. If they're already off, no action is taken.

	Name
int	**wolfSSL_get_error**(WOLFSSL * ssl, int ret)This function returns a unique error code describing why the previous API function call (wolfSSL_connect, wolfSSL_accept, wolfSSL_read, wolfSSL_write, etc.) resulted in an error return code (SSL_FAILURE). The return value of the previous function is passed to wolfSSL_get_error through ret. After wolfSSL_get_error is called and returns the unique error code, wolfSSL_ERR_error_string() may be called to get a human_readable error string. See wolfSSL_ERR_error_string() for more information.
void	**wolfSSL_load_error_strings**(void)This function is for OpenSSL compatibility (SSL_load_error_string) only and takes no action.
char *	**wolfSSL_ERR_error_string and data is the storage buffer which the error string will be placed in. The maximum length of data is 80 characters by default, as defined by MAX_ERROR_SZ is wolfssl/wolfcrypt/error.h.
void	**wolfSSL_ERR_error_string_n into a more human-readable error string. The human-readable string is placed in buf.
void	**wolfSSL_ERR_print_errors_fp and fp is the file which the error string will be placed in.
void	**wolfSSL_ERR_print_errors_cb**(int*)(const char* str, size_t len, void *u) cb, void* u)This function uses the provided callback to handle error reporting. The callback function is executed for each error line. The string, length, and userdata are passed into the callback parameters.
int	**wolfSSL_want_read and getting SSL_ERROR_WANT_READ in return. If the underlying error state is SSL_ERROR_WANT_READ, this function will return 1, otherwise, 0.
int	**wolfSSL_want_write and getting SSL_ERROR_WANT_WRITE in return. If the underlying error state is SSL_ERROR_WANT_WRITE, this function will return 1, otherwise, 0.
unsigned long	**wolfSSL_ERR_peek_last_error**(void)This function returns the absolute value of the last error from WOLFSSL_ERROR encountered.

A.7.2 Functions Documentation

```
int wolfSSL_Debugging_ON(
    void
```

```
)
```

If logging has been enabled at build time this function turns on logging at runtime. To enable logging at build time use –enable-debug or define DEBUG_WOLFSSL.

Parameters:

- **none** No parameters.

See:

- wolfSSL_Debugging_OFF
- wolfSSL_SetLoggingCb

Return:

- 0 upon success.
- NOT_COMPILED_IN is the error that will be returned if logging isn't enabled for this build.

Example

```
wolfSSL_Debugging_ON();
```

```
void wolfSSL_Debugging_OFF(
    void
)
```

This function turns off runtime logging messages. If they're already off, no action is taken.

Parameters:

- **none** No parameters.

See:

- wolfSSL_Debugging_ON
- wolfSSL_SetLoggingCb

Return: none No returns.

Example

```
wolfSSL_Debugging_OFF();
```

```
int wolfSSL_get_error(
    WOLFSSL * ssl,
    int ret
)
```

This function returns a unique error code describing why the previous API function call (wolfSSL_connect, wolfSSL_accept, wolfSSL_read, wolfSSL_write, etc.) resulted in an error return code (SSL_FAILURE). The return value of the previous function is passed to wolfSSL_get_error through ret. After wolfSSL_get_error is called and returns the unique error code, wolfSSL_ERR_error_string() may be called to get a human-readable error string. See wolfSSL_ERR_error_string() for more information.

Parameters:

- **ssl** pointer to the SSL object, created with wolfSSL_new().
- **ret** return value of the previous function that resulted in an error return code.

See:

- wolfSSL_ERR_error_string
- wolfSSL_ERR_error_string_n
- wolfSSL_ERR_print_errors_fp
- wolfSSL_load_error_strings

Return:

- On successful completion, this function will return the unique error code describing why the previous API function failed.
- SSL_ERROR_NONE will be returned if ret > 0. For ret <= 0, there are some cases when this value can also be returned when a previous API appeared to return an error code but no error actually occurred. An example is calling wolfSSL_read() is called afterwards, SSL_ERROR_NONE will be returned.

Example

```
int err = 0;
WOLFSSL* ssl;
char buffer[80];
...
err = wolfSSL_get_error(ssl, 0);
wolfSSL_ERR_error_string(err, buffer);
printf("err = %d, %s\n", err, buffer);
```

```
void wolfSSL_load_error_strings(
    void
)
```

This function is for OpenSSL compatibility (SSL_load_error_string) only and takes no action.

Parameters:

- **none** No parameters.

See:

- wolfSSL_get_error
- wolfSSL_ERR_error_string
- wolfSSL_ERR_error_string_n
- wolfSSL_ERR_print_errors_fp
- wolfSSL_load_error_strings

Return: none No returns.

Example

```
wolfSSL_load_error_strings();
```

```
char * wolfSSL_ERR_error_string(
    unsigned long errNumber,
    char * data
)
```

This function converts an error code returned by wolfSSL_get_error() and data is the storage buffer which the error string will be placed in. The maximum length of data is 80 characters by default, as defined by MAX_ERROR_SZ is wolfssl/wolfcrypt/error.h.

Parameters:

- **errNumber** error code returned by wolfSSL_get_error().
- **data** output buffer containing human-readable error string matching errNumber.

See:

- wolfSSL_get_error
- wolfSSL_ERR_error_string_n
- wolfSSL_ERR_print_errors_fp
- wolfSSL_load_error_strings

Return:

- success On successful completion, this function returns the same string as is returned in data.
- failure Upon failure, this function returns a string with the appropriate failure reason, msg.

Example

```
int err = 0;
WOLFSSL* ssl;
char buffer[80];
...
err = wolfSSL_get_error(ssl, 0);
```

```
wolfSSL_ERR_error_string(err, buffer);
printf("err = %d, %s\n", err, buffer);
```

```
void wolfSSL_ERR_error_string_n(
    unsigned long e,
    char * buf,
    unsigned long sz
)
```

This function is a version of wolfSSL_ERR_error_string() into a more human-readable error string. The human-readable string is placed in buf.

Parameters:

- **e** error code returned by wolfSSL_get_error().
- **buff** output buffer containing human-readable error string matching e.
- **len** maximum length in characters which may be written to buf.

See:

- wolfSSL_get_error
- wolfSSL_ERR_error_string
- wolfSSL_ERR_print_errors_fp
- wolfSSL_load_error_strings

Return: none No returns.

Example

```
int err = 0;
WOLFSSL* ssl;
char buffer[80];
...
err = wolfSSL_get_error(ssl, 0);
wolfSSL_ERR_error_string_n(err, buffer, 80);
printf("err = %d, %s\n", err, buffer);
```

```
void wolfSSL_ERR_print_errors_fp(
    XFILE fp,
    int err
)
```

This function converts an error code returned by wolfSSL_get_error() and fp is the file which the error string will be placed in.

Parameters:

- **fp** output file for human-readable error string to be written to.
- **err** error code returned by wolfSSL_get_error().

See:

- wolfSSL_get_error
- wolfSSL_ERR_error_string
- wolfSSL_ERR_error_string_n
- wolfSSL_load_error_strings

Return: none No returns.

Example

```
int err = 0;
WOLFSSL* ssl;
FILE* fp = ...
...
err = wolfSSL_get_error(ssl, 0);
wolfSSL_ERR_print_errors_fp(fp, err);
```

```
void wolfSSL_ERR_print_errors_cb(
    int(*)(const char *str, size_t len, void *u) cb,
    void * u
)
```

This function uses the provided callback to handle error reporting. The callback function is executed for each error line. The string, length, and userdata are passed into the callback parameters.

Parameters:

- **cb** the callback function.
- **u** userdata to pass into the callback function.

See:

- wolfSSL_get_error
- wolfSSL_ERR_error_string
- wolfSSL_ERR_error_string_n
- wolfSSL_load_error_strings

Return: none No returns.

Example

```
int error_cb(const char *str, size_t len, void *u)
{ fprintf((FILE*)u, "%-*.*s\n", (int)len, (int)len, str); return 0; }
...
FILE* fp = ...
wolfSSL_ERR_print_errors_cb(error_cb, fp);
```

```
int wolfSSL_want_read(
    WOLFSSL *
)
```

This function is similar to calling wolfSSL_get_error() and getting SSL_ERROR_WANT_READ in return. If the underlying error state is SSL_ERROR_WANT_READ, this function will return 1, otherwise, 0.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().

See:

- wolfSSL_want_write
- wolfSSL_get_error

Return:

- 1 wolfSSL_get_error() would return SSL_ERROR_WANT_READ, the underlying I/O has data available for reading.
- 0 There is no SSL_ERROR_WANT_READ error state.

Example

```
int ret;
WOLFSSL* ssl = 0;
...

ret = wolfSSL_want_read(ssl);
if (ret == 1) {
    // underlying I/O has data available for reading (SSL_ERROR_WANT_READ)
}
```

```
int wolfSSL_want_write(
    WOLFSSL *
)
```

This function is similar to calling wolfSSL_get_error() and getting SSL_ERROR_WANT_WRITE in return. If the underlying error state is SSL_ERROR_WANT_WRITE, this function will return 1, otherwise, 0.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().

See:

- wolfSSL_want_read
- wolfSSL_get_error

Return:

- 1 wolfSSL_get_error() would return SSL_ERROR_WANT_WRITE, the underlying I/O needs data to be written in order for progress to be made in the underlying SSL connection.
- 0 There is no SSL_ERROR_WANT_WRITE error state.

Example

```
int ret;
WOLFSSL* ssl = 0;
...
ret = wolfSSL_want_write(ssl);
if (ret == 1) {
    // underlying I/O needs data to be written (SSL_ERROR_WANT_WRITE)
}
```

```
unsigned long wolfSSL_ERR_peek_last_error(
    void
)
```

This function returns the absolute value of the last error from WOLFSSL_ERROR encountered.

Parameters:

- **none** No parameters.

See: wolfSSL_ERR_print_errors_fp

Return: error Returns absolute value of last error.

Example

```
unsigned long err;
...
err = wolfSSL_ERR_peek_last_error();
// inspect err value
```

A.8 wolfSSL Initialization/Shutdown

A.7.2.11 function wolfSSL_ERR_peek_last_error

A.8.1 Functions

	Name
int	**wolfSSL_shutdown will yield either SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE. The calling process must then repeat the call to wolfSSL_shutdown() when the underlying I/O is ready.
int	**wolfSSL_SetServerID**(WOLFSSL * ssl, const unsigned char * id, int len, int newSession)This function associates the client session with the server id. If the newSession flag is on, an existing session won't be reused.
int	**wolfSSL_library_init. This function is a wrapper around wolfSSL_Init() and exists for OpenSSL compatibility (SSL_library_init) when wolfSSL has been compiled with OpenSSL compatibility layer. wolfSSL_Init() is the more typically-used wolfSSL initialization function.
int	**wolfSSL_get_shutdown**(const WOLFSSL * ssl)This function checks the shutdown conditions in closeNotify or connReset or sentNotify members of the Options structure. The Options structure is within the WOLFSSL structure.
int	**wolfSSL_is_init_finished**(WOLFSSL * ssl)This function checks to see if the connection is established.
int	**wolfSSL_Init**(void)Initializes the wolfSSL library for use. Must be called once per application and before any other call to the library.
int	**wolfSSL_Cleanup**(void)Un-initializes the wolfSSL library from further use. Doesn't have to be called, though it will free any resources used by the library.
int	**wolfSSL_SetMinVersion**(WOLFSSL * ssl, int version)This function sets the minimum downgrade version allowed. Applicable only when the connection allows downgrade using (wolfSSLv23_client_method or wolfSSLv23_server_method).
int	**wolfSSL_ALPN_GetProtocol**(WOLFSSL * ssl, char ** protocol_name, unsigned short * size)This function gets the protocol name set by the server.
int	**wolfSSL_ALPN_GetPeerProtocol**(WOLFSSL * ssl, char ** list, unsigned short * listSz)This function copies the alpn_client_list data from the SSL object to the buffer.

	Name
int	**wolfSSL_MakeTlsMasterSecret**(unsigned char * ms, word32 msLen, const unsigned char * pms, word32 pmsLen, const unsigned char * cr, const unsigned char * sr, int tls1_2, int hash_type)This function copies the values of cr and sr then passes through to wc_PRF (pseudo random function) and returns that value.
int	**wolfSSL_preferred_group**(WOLFSSL * ssl)This function returns the key exchange group the client prefers to use in the TLS v1.3 handshake. Call this function to after a handshake is complete to determine which group the server prefers so that this information can be used in future connections to pre-generate a key pair for key exchange.
int	**wolfSSL_get_client_suites_sigalgs**(const WOLFSSL * ssl, const byte ** suites, word16 * suiteSz, const byte ** hashSigAlgo, word16 * hashSigAlgoSz)This function returns the raw list of ciphersuites and signature algorithms offered by the client. The lists are only stored and returned inside a callback setup with wolfSSL_CTX_set_cert_cb(). This is useful to be able to dynamically load certificates and keys based on the available ciphersuites and signature algorithms.
WOLFSSL_CIPHERSUITE_INFO	**wolfSSL_get_ciphersuite_info**(byte first, byte second)This returns information about the ciphersuite directly from the raw ciphersuite bytes.
int	**wolfSSL_get_sigalg_info**(byte first, byte second, int * hashAlgo, int * sigAlgo)This returns information about the hash and signature algorithm directly from the raw ciphersuite bytes.

A.8.2 Functions Documentation

```
int wolfSSL_shutdown(
    WOLFSSL *
)
```

This function shuts down an active SSL/TLS connection using the SSL session, ssl. This function will try to send a “close notify” alert to the peer. The calling application can choose to wait for the peer to send its “close notify” alert in response or just go ahead and shut down the underlying connection after directly calling wolfSSL_shutdown (to save resources). Either option is allowed by the TLS specification. If the underlying connection will be used again in the future, the complete two-directional shutdown procedure must be performed to keep synchronization intact between the peers. wolfSSL_shutdown() works with both blocking and non-blocking I/O. When the underlying I/O is non-blocking, wolfSSL_shutdown() will return an error if the underlying I/O could not satisfy the needs of wolfSSL_shutdown() to continue. In this case, a call to wolfSSL_get_error() will yield either SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE. The calling process must then repeat the call

to wolfSSL_shutdown() when the underlying I/O is ready.

Parameters:

- **ssl** pointer to the SSL session created with wolfSSL_new().

See:

- wolfSSL_free
- wolfSSL_CTX_free

Return:

- SSL_SUCCESS will be returned upon success.
- SSL_SHUTDOWN_NOT_DONE will be returned when shutdown has not finished, and the function should be called again.
- SSL_FATAL_ERROR will be returned upon failure. Call wolfSSL_get_error() for a more specific error code.

Example

```
#include <wolfssl/ssl.h>

int ret = 0;
WOLFSSL* ssl = 0;
...
ret = wolfSSL_shutdown(ssl);
if (ret != 0) {
    // failed to shut down SSL connection
}
```

```
int wolfSSL_SetServerID(
    WOLFSSL * ssl,
    const unsigned char * id,
    int len,
    int newSession
)
```

This function associates the client session with the server id. If the newSession flag is on, an existing session won't be reused.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **id** a constant byte pointer that will be copied to the serverID member of the WOLFSSL_SESSION structure.
- **len** an int type representing the length of the session id parameter.
- **newSession** an int type representing the flag to denote whether to reuse a session or not.

See: wolfSSL_set_session

Return:

- SSL_SUCCESS returned if the function executed without error.
- BAD_FUNC_ARG returned if the WOLFSSL struct or id parameter is NULL or if len is not greater than zero.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol );
WOLFSSL* ssl = WOLFSSL_new(ctx);
const byte id[MAX_SIZE];  // or dynamically create space
int len = 0; // initialize length
int newSession = 0; // flag to allow
…
int ret = wolfSSL_SetServerID(ssl, id, len, newSession);

if (ret == WOLFSSL_SUCCESS) {
    // The Id was successfully set
}
```

```
int wolfSSL_library_init(
    void
)
```

This function is called internally in wolfSSL_CTX_new(). This function is a wrapper around wolfSSL_Init() and exists for OpenSSL compatibility (SSL_library_init) when wolfSSL has been compiled with OpenSSL compatibility layer. wolfSSL_Init() is the more typically-used wolfSSL initialization function.

Parameters:

- **none** No parameters.

See:

- wolfSSL_Init
- wolfSSL_Cleanup

Return:

- SSL_SUCCESS If successful the call will return.
- SSL_FATAL_ERROR is returned upon failure.

Example

```
int ret = 0;
ret = wolfSSL_library_init();
if (ret != SSL_SUCCESS) {
```

```
    failed to initialize wolfSSL
}
...
```

```
int wolfSSL_get_shutdown(
    const WOLFSSL * ssl
)
```

This function checks the shutdown conditions in closeNotify or connReset or sentNotify members of the Options structure. The Options structure is within the WOLFSSL structure.

Parameters:

- **ssl** a constant pointer to a WOLFSSL structure, created using wolfSSL_new().

See: wolfSSL_SESSION_free

Return:

- 1 SSL_SENT_SHUTDOWN is returned.
- 2 SSL_RECEIVED_SHUTDOWN is returned.

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_CTX* ctx = WOLFSSL_CTX_new( protocol method );
WOLFSSL* ssl = WOLFSSL_new(ctx);
…
int ret;
ret = wolfSSL_get_shutdown(ssl);

if(ret == 1){
    SSL_SENT_SHUTDOWN
} else if(ret == 2){
    SSL_RECEIVED_SHUTDOWN
} else {
    Fatal error.
}
```

```
int wolfSSL_is_init_finished(
    WOLFSSL * ssl
)
```

This function checks to see if the connection is established.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_set_accept_state
- wolfSSL_get_keys
- wolfSSL_set_shutdown

Return:

- 0 returned if the connection is not established, i.e. the WOLFSSL struct is NULL or the handshake is not done.
- 1 returned if the connection is established i.e. the WOLFSSL handshake is done.

EXAMPLE

```
#include <wolfssl/ssl.h>

WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
WOLFSSL* ssl = wolfSSL_new(ctx);
...
if(wolfSSL_is_init_finished(ssl)){
    Handshake is done and connection is established
}
```

```
int wolfSSL_Init(
    void
)
```

Initializes the wolfSSL library for use. Must be called once per application and before any other call to the library.

See: wolfSSL_Cleanup

Return:

- SSL_SUCCESS If successful the call will return.
- BAD_MUTEX_E is an error that may be returned.
- WC_INIT_E wolfCrypt initialization error returned.

Example

```
int ret = 0;
ret = wolfSSL_Init();
if (ret != SSL_SUCCESS) {
    failed to initialize wolfSSL library
}
```

```
int wolfSSL_Cleanup(
    void
)
```

Un-initializes the wolfSSL library from further use. Doesn't have to be called, though it will free any resources used by the library.

See: wolfSSL_Init

Return:

- SSL_SUCCESS return no errors.
- BAD_MUTEX_E a mutex error return.]

Example

```
wolfSSL_Cleanup();
```

```
int wolfSSL_SetMinVersion(
    WOLFSSL * ssl,
    int version
)
```

This function sets the minimum downgrade version allowed. Applicable only when the connection allows downgrade using (wolfSSLv23_client_method or wolfSSLv23_server_method).

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **version** an integer representation of the version to be set as the minimum: WOLFSSL_SSLV3 = 0, WOLFSSL_TLSV1 = 1, WOLFSSL_TLSV1_1 = 2 or WOLFSSL_TLSV1_2 = 3.

See: SetMinVersionHelper

Return:

- SSL_SUCCESS returned if this function and its subroutine executes without error.
- BAD_FUNC_ARG returned if the SSL object is NULL. In the subroutine this error is thrown if there is not a good version match.

Example

```
WOLFSSL_CTX* ctx = WOLFSSL_CTX_new(protocol method);
WOLFSSL* ssl = WOLFSSL_new(ctx);
int version;  macro representation
…
if(wolfSSL_CTX_SetMinVersion(ssl->ctx, version) != SSL_SUCCESS){
    Failed to set min version
}
```

```
int wolfSSL_ALPN_GetProtocol(
    WOLFSSL * ssl,
    char ** protocol_name,
    unsigned short * size
)
```

This function gets the protocol name set by the server.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **protocol_name** a pointer to a char that represents the protocol name and will be held in the ALPN structure.
- **size** a word16 type that represents the size of the protocol_name.

See:

- TLSX_ALPN_GetRequest
- TLSX_Find

Return:

- SSL_SUCCESS returned on successful execution where no errors were thrown.
- SSL_FATAL_ERROR returned if the extension was not found or if there was no protocol match with peer. There will also be an error thrown if there is more than one protocol name accepted.
- SSL_ALPN_NOT_FOUND returned signifying that no protocol match with peer was found.
- BAD_FUNC_ARG returned if there was a NULL argument passed into the function.

Example

```
WOLFSSL_CTX* ctx = WOLFSSL_CTX_new( protocol method );
WOLFSSL* ssl = WOLFSSL_new(ctx);
...
int err;
char* protocol_name = NULL;
Word16 protocol_nameSz = 0;
err = wolfSSL_ALPN_GetProtocol(ssl, &protocol_name, &protocol_nameSz);

if(err == SSL_SUCCESS){
    // Sent ALPN protocol
}
```

```
int wolfSSL_ALPN_GetPeerProtocol(
    WOLFSSL * ssl,
    char ** list,
    unsigned short * listSz
)
```

This function copies the alpn_client_list data from the SSL object to the buffer.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **list** a pointer to the buffer. The data from the SSL object will be copied into it.
- **listSz** the buffer size.

See: wolfSSL_UseALPN

Return:

- SSL_SUCCESS returned if the function executed without error. The alpn_client_list member of the SSL object has been copied to the list parameter.
- BAD_FUNC_ARG returned if the list or listSz parameter is NULL.
- BUFFER_ERROR returned if there will be a problem with the list buffer (either it's NULL or the size is 0).
- MEMORY_ERROR returned if there was a problem dynamically allocating memory.

Example

```
#import <wolfssl/ssl.h>

WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method);
WOLFSSL* ssl = wolfSSL_new(ctx);
…
#ifdef HAVE_ALPN
char* list = NULL;
word16 listSz = 0;
…
err = wolfSSL_ALPN_GetPeerProtocol(ssl, &list, &listSz);

if(err == SSL_SUCCESS){
    List of protocols names sent by client
}
```

```
int wolfSSL_MakeTlsMasterSecret(
    unsigned char * ms,
    word32 msLen,
    const unsigned char * pms,
    word32 pmsLen,
    const unsigned char * cr,
    const unsigned char * sr,
    int tls1_2,
    int hash_type
)
```

This function copies the values of cr and sr then passes through to wc_PRF (pseudo random function) and returns that value.

Parameters:

- **ms** the master secret held in the Arrays structure.
- **msLen** the length of the master secret.

- **pms** the pre-master secret held in the Arrays structure.
- **pmsLen** the length of the pre-master secret.
- **cr** the client random.
- **sr** the server random.
- **tls1_2** signifies that the version is at least tls version 1.2.
- **hash_type** signifies the hash type.

See:

- wc_PRF
- MakeTlsMasterSecret

Return:

- 0 on success
- BUFFER_E returned if there will be an error with the size of the buffer.
- MEMORY_E returned if a subroutine failed to allocate dynamic memory.

Example

```
WOLFSSL* ssl;

called in MakeTlsMasterSecret and retrieves the necessary
information as follows:

int MakeTlsMasterSecret(WOLFSSL* ssl){
int ret;
ret = wolfSSL_makeTlsMasterSecret(ssl->arrays->masterSecret, SECRET_LEN,
ssl->arrays->preMasterSecret, ssl->arrays->preMasterSz,
ssl->arrays->clientRandom, ssl->arrays->serverRandom,
IsAtLeastTLSv1_2(ssl), ssl->specs.mac_algorithm);
…
return ret;

}
```

```
int wolfSSL_preferred_group(
    WOLFSSL * ssl
)
```

This function returns the key exchange group the client prefers to use in the TLS v1.3 handshake. Call this function to after a handshake is complete to determine which group the server prefers so that this information can be used in future connections to pre-generate a key pair for key exchange.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_UseKeyShare
- wolfSSL_CTX_set_groups
- wolfSSL_set_groups
- wolfSSL_CTX_set1_groups_list
- wolfSSL_set1_groups_list

Return:

- BAD_FUNC_ARG if ssl is NULL or not using TLS v1.3.
- SIDE_ERROR if called with a server.
- NOT_READY_ERROR if called before handshake is complete.
- Group identifier if successful.

Example

```
int ret;
int group;
WOLFSSL* ssl;
...
ret = wolfSSL_CTX_set1_groups_list(ssl)
if (ret < 0) {
    // failed to get group
}
group = ret;
```

```
int wolfSSL_get_client_suites_sigalgs(
    const WOLFSSL * ssl,
    const byte ** suites,
    word16 * suiteSz,
    const byte ** hashSigAlgo,
    word16 * hashSigAlgoSz
)
```

This function returns the raw list of ciphersuites and signature algorithms offered by the client. The lists are only stored and returned inside a callback setup with wolfSSL_CTX_set_cert_cb(). This is useful to be able to dynamically load certificates and keys based on the available ciphersuites and signature algorithms.

Parameters:

- **ssl** The WOLFSSL object to extract the lists from.
- **optional** suites Raw and unfiltered list of client ciphersuites
- **optional** suiteSz Size of suites in bytes
- **optional** hashSigAlgo Raw and unfiltered list of client signature algorithms
- **optional** hashSigAlgoSz Size of hashSigAlgo in bytes

See:

- wolfSSL_get_ciphersuite_info
- wolfSSL_get_sigalg_info

Return:

- WOLFSSL_SUCCESS when suites available
- WOLFSSL_FAILURE when suites not available

Example

```
int certCB(WOLFSSL* ssl, void* arg)
{
    const byte* suites = NULL;
    word16 suiteSz = 0;
    const byte* hashSigAlgo = NULL;
    word16 hashSigAlgoSz = 0;

    wolfSSL_get_client_suites_sigalgs(ssl, &suites, &suiteSz, &hashSigAlgo,
            &hashSigAlgoSz);

    // Choose certificate to load based on ciphersuites and sigalgs
}

WOLFSSL* ctx;
ctx  = wolfSSL_CTX_new(wolfTLSv1_3_method_ex(NULL));
wolfSSL_CTX_set_cert_cb(ctx, certCB, NULL);
```

```
WOLFSSL_CIPHERSUITE_INFO wolfSSL_get_ciphersuite_info(
    byte first,
    byte second
)
```

This returns information about the ciphersuite directly from the raw ciphersuite bytes.

Parameters:

- **first** First byte of the ciphersuite
- **second** Second byte of the ciphersuite

See:

- wolfSSL_get_client_suites_sigalgs
- wolfSSL_get_sigalg_info

Return: WOLFSSL_CIPHERSUITE_INFO A struct containing information about the type of authentication used in the ciphersuite.

Example

```
WOLFSSL_CIPHERSUITE_INFO info =
        wolfSSL_get_ciphersuite_info(suites[0], suites[1]);
if (info.rsaAuth)
    haveRSA = 1;
```

```
else if (info.eccAuth)
    haveECC = 1;



int wolfSSL_get_sigalg_info(
    byte first,
    byte second,
    int * hashAlgo,
    int * sigAlgo
)
```

This returns information about the hash and signature algorithm directly from the raw ciphersuite bytes.

Parameters:

- **first** First byte of the hash and signature algorithm
- **second** Second byte of the hash and signature algorithm
- **hashAlgo** The enum wc_HashType of the MAC algorithm
- **sigAlgo** The enum Key_Sum of the authentication algorithm

See:

- wolfSSL_get_client_suites_sigalgs
- wolfSSL_get_ciphersuite_info

Return:

- 0 when info was correctly set
- BAD_FUNC_ARG when either input parameters are NULL or the bytes are not a recognized sigalg suite

Example

```
enum wc_HashType hashAlgo;
enum Key_Sum sigAlgo;

wolfSSL_get_sigalg_info(hashSigAlgo[idx+0], hashSigAlgo[idx+1],
        &hashAlgo, &sigAlgo);

if (sigAlgo == RSAk || sigAlgo == RSAPSSk)
    haveRSA = 1;
else if (sigAlgo == ECDSAk)
    haveECC = 1;
```

B wolfCrypt API Reference

B.1 ASN.1

B.1.1 Functions

	Name
int	**wc_InitCert**(Cert *)This function initializes a default cert, with the default options: version = 3 (0x2), serial = 0, sigType = SHA_WITH_RSA, issuer = blank, daysValid = 500, selfSigned = 1 (true) use subject as issuer, subject = blank.
Cert *	**wc_CertNew. When the application is finished using the allocated Cert structure wc_CertFree() must be called.
void	**wc_CertFree.
int	**wc_MakeCert**(Cert * cert, byte * derBuffer, word32 derSz, RsaKey * rsaKey, ecc_key * eccKey, WC_RNG * rng)Used to make CA signed certs. Called after the subject information has been entered. This function makes an x509 Certificate v3 RSA or ECC from a cert input. It then writes this cert to derBuffer. It takes in either an rsaKey or an eccKey to generate the certificate. The certificate must be initialized with wc_InitCert before this method is called.
int	**wc_MakeCertReq**(Cert * cert, byte * derBuffer, word32 derSz, RsaKey * rsaKey, ecc_key * eccKey)This function makes a certificate signing request using the input certificate and writes the output to derBuffer. It takes in either an rsaKey or an eccKey to generate the certificate request. wc_SignCert() will need to be called after this function to sign the certificate request. Please see the wolfCrypt test application (./wolfcrypt/test/test.c) for an example usage of this function.
int	**wc_SignCert if creating a CA signed cert.
int	**wc_MakeSelfCert**(Cert * cert, byte * derBuffer, word32 derSz, RsaKey * key, WC_RNG * rng)This function is a combination of the previous two functions, wc_MakeCert and wc_SignCert for self signing (the previous functions may be used for CA requests). It makes a certificate, and then signs it, generating a self-signed certificate.

	Name
int	**wc_SetIssuer**(Cert * cert, const char * issuerFile)This function sets the issuer for a certificate to the issuer in the provided pem issuerFile. It also changes the certificate's self-signed attribute to false. The issuer specified in issuerFile is verified prior to setting the cert issuer. This method is used to set fields prior to signing.
int	**wc_SetSubject**(Cert * cert, const char * subjectFile)This function sets the subject for a certificate to the subject in the provided pem subjectFile. This method is used to set fields prior to signing.
int	**wc_SetSubjectRaw**(Cert * cert, const byte * der, int derSz)This function sets the raw subject for a certificate from the subject in the provided der buffer. This method is used to set the raw subject field prior to signing.
int	**wc_GetSubjectRaw**(byte ** subjectRaw, Cert * cert)This function gets the raw subject from the certificate structure.
int	**wc_SetAltNames**(Cert * cert, const char * file)This function sets the alternate names for a certificate to the alternate names in the provided pem file. This is useful in the case that one wishes to secure multiple domains with the same certificate. This method is used to set fields prior to signing.
int	**wc_SetIssuerBuffer**(Cert * cert, const byte * der, int derSz)This function sets the issuer for a certificate from the issuer in the provided der buffer. It also changes the certificate's self-signed attribute to false. This method is used to set fields prior to signing.
int	**wc_SetIssuerRaw**(Cert * cert, const byte * der, int derSz)This function sets the raw issuer for a certificate from the issuer in the provided der buffer. This method is used to set the raw issuer field prior to signing.
int	**wc_SetSubjectBuffer**(Cert * cert, const byte * der, int derSz)This function sets the subject for a certificate from the subject in the provided der buffer. This method is used to set fields prior to signing.
int	**wc_SetAltNamesBuffer**(Cert * cert, const byte * der, int derSz)This function sets the alternate names for a certificate from the alternate names in the provided der buffer. This is useful in the case that one wishes to secure multiple domains with the same certificate. This method is used to set fields prior to signing.

	Name
int	**wc_SetDatesBuffer**(Cert * cert, const byte * der, int derSz)This function sets the dates for a certificate from the date range in the provided der buffer. This method is used to set fields prior to signing.
int	**wc_SetAuthKeyIdFromPublicKey**(Cert * cert, RsaKey * rsakey, ecc_key * eckey)Set AKID from either an RSA or ECC public key. note: Only set one of rsakey or eckey, not both.
int	**wc_SetAuthKeyIdFromCert**(Cert * cert, const byte * der, int derSz)Set AKID from from DER encoded certificate.
int	**wc_SetAuthKeyId**(Cert * cert, const char * file)Set AKID from certificate file in PEM format.
int	**wc_SetSubjectKeyIdFromPublicKey**(Cert * cert, RsaKey * rsakey, ecc_key * eckey)Set SKID from RSA or ECC public key.
int	**wc_SetSubjectKeyId**(Cert * cert, const char * file)Set SKID from public key file in PEM format. Both arguments are required.
int	**wc_PemPubKeyToDer**(const char * fileName, unsigned char * derBuf, int derSz)Loads a PEM key from a file and converts to a DER encoded buffer.
int	**wc_PubKeyPemToDer**(const unsigned char * pem, int pemSz, unsigned char * buff, int buffSz)Convert a PEM encoded public key to DER. Returns the number of bytes written to the buffer or a negative value for an error.
int	**wc_PemCertToDer**(const char * fileName, unsigned char * derBuf, int derSz)This function converts a pem certificate to a der certificate, and places the resulting certificate in the derBuf buffer provided.
int	**wc_DerToPem**(const byte * der, word32 derSz, byte * output, word32 outputSz, int type)This function converts a der formatted input certificate, contained in the der buffer, into a pem formatted output certificate, contained in the output buffer. It should be noted that this is not an in place conversion, and a separate buffer must be utilized to store the pem formatted output.

	Name
int	**wc_DerToPemEx**(const byte * der, word32 derSz, byte * output, word32 outputSz, byte * cipherIno, int type)This function converts a der formatted input certificate, contained in the der buffer, into a pem formatted output certificate, contained in the output buffer. It should be noted that this is not an in place conversion, and a separate buffer must be utilized to store the pem formatted output. Allows setting cipher info.
int	**wc_EccPrivateKeyDecode**(const byte * input, word32 * inOutIdx, ecc_key * key, word32 inSz)This function reads in an ECC private key from the input buffer, input, parses the private key, and uses it to generate an ecc_key object, which it stores in key.
int	**wc_EccKeyToDer**(ecc_key * key, byte * output, word32 inLen)This function writes a private ECC key to der format.
int	**wc_EccPublicKeyDecode**(const byte * input, word32 * inOutIdx, ecc_key * key, word32 inSz)Decodes an ECC public key from an input buffer. It will parse an ASN sequence to retrieve the ECC key.
int	**wc_EccPublicKeyToDer**(ecc_key * key, byte * output, word32 inLen, int with_AlgCurve)This function converts the ECC public key to DER format. It returns the size of buffer used. The public ECC key in DER format is stored in output buffer. The with_AlgCurve flag will include a header that has the Algorithm and Curve information.
int	**wc_EccPublicKeyToDer_ex**(ecc_key * key, byte * output, word32 inLen, int with_AlgCurve, int comp)This function converts the ECC public key to DER format. It returns the size of buffer used. The public ECC key in DER format is stored in output buffer. The with_AlgCurve flag will include a header that has the Algorithm and Curve information. The comp parameter determines if the public key will be exported as compressed.
int	**wc_Curve25519PrivateKeyDecode**(const byte * input, word32 * inOutIdx, curve25519_key * key, word32 inSz)This function decodes a Curve25519 private key (only) from a DER encoded buffer.
int	**wc_Curve25519PublicKeyDecode**(const byte * input, word32 * inOutIdx, curve25519_key * key, word32 inSz)This function decodes a Curve25519 public key (only) from a DER encoded buffer.

	Name
int	**wc_Curve25519KeyDecode**(const byte * input, word32 * inOutIdx, curve25519_key * key, word32 inSz)This function decodes a Curve25519 key from a DER encoded buffer. It can decode either a private key, a public key, or both.
int	**wc_Curve25519PrivateKeyToDer**(curve25519_key * key, byte * output, word32 inLen)This function encodes a Curve25519 private key to DER format. If the input key structure contains a public key, it will be ignored.
int	**wc_Curve25519PublicKeyToDer**(curve25519_key * key, byte * output, word32 inLen, int withAlg)This function encodes a Curve25519 public key to DER format. If the input key structure contains a private key, it will be ignored.
int	**wc_Curve25519KeyToDer**(curve25519_key * key, byte * output, word32 inLen, int withAlg)This function encodes a Curve25519 key to DER format. It can encode either a private key, a public key, or both.
word32	**wc_EncodeSignature**(byte * out, const byte * digest, word32 digSz, int hashOID)This function encodes a digital signature into the output buffer, and returns the size of the encoded signature created.
int	**wc_GetCTC_HashOID**(int type)This function returns the hash OID that corresponds to a hashing type. For example, when given the type: WC_SHA512, this function returns the identifier corresponding to a SHA512 hash, SHA512h.
void	**wc_SetCert_Free**(Cert * cert)This function cleans up memory and resources used by the certificate structure's decoded cert cache. When WOLFSSL_CERT_GEN_CACHE is defined the decoded cert structure is cached in the certificate structure. This allows subsequent calls to certificate set functions to avoid parsing the decoded cert on each call.
int	**wc_GetPkcs8TraditionalOffset**(byte * input, word32 * inOutIdx, word32 sz)This function finds the beginning of the traditional private key inside a PKCS#8 unencrypted buffer.
int	**wc_CreatePKCS8Key**(byte * out, word32 * outSz, byte * key, word32 keySz, int algoID, const byte * curveOID, word32 oidSz)This function takes in a DER private key and converts it to PKCS#8 format. Also used in creating PKCS#12 shrouded key bags. See RFC 5208.

	Name
int	**wc_EncryptPKCS8Key**(byte * key, word32 keySz, byte * out, word32 * outSz, const char * password, int passwordSz, int vPKCS, int pbeOid, int encAlgId, byte * salt, word32 saltSz, int itt, WC_RNG * rng, void * heap)This function takes in an unencrypted PKCS#8 DER key (e.g. one created by wc_CreatePKCS8Key) and converts it to PKCS#8 encrypted format. The resulting encrypted key can be decrypted using wc_DecryptPKCS8Key. See RFC 5208.
int	**wc_DecryptPKCS8Key**(byte * input, word32 sz, const char * password, int passwordSz)This function takes an encrypted PKCS#8 DER key and decrypts it to PKCS#8 unencrypted DER. Undoes the encryption done by wc_EncryptPKCS8Key. See RFC5208. The input buffer is overwritten with the decrypted data.
int	**wc_CreateEncryptedPKCS8Key**(byte * key, word32 keySz, byte * out, word32 * outSz, const char * password, int passwordSz, int vPKCS, int pbeOid, int encAlgId, byte * salt, word32 saltSz, int itt, WC_RNG * rng, void * heap)This function takes a traditional, DER key, converts it to PKCS#8 format, and encrypts it. It uses wc_CreatePKCS8Key and wc_EncryptPKCS8Key to do this.
void	**wc_InitDecodedCert**(struct DecodedCert * cert, const byte * source, word32 inSz, void * heap)This function initializes the DecodedCert pointed to by the “cert” parameter. It saves the “source” pointer to a DER-encoded certificate of length “inSz.” This certificate can be parsed by a subsequent call to wc_ParseCert.
int	**wc_ParseCert**(DecodedCert * cert, int type, int verify, void * cm)This function parses the DER-encoded certificate saved in the DecodedCert object and populates the fields of that object. The DecodedCert must have been initialized with a prior call to wc_InitDecodedCert. This function takes an optional pointer to a CertificateManager object, which is used to populate the certificate authority information of the DecodedCert, if the CA is found in the CertificateManager.
void	**wc_FreeDecodedCert**(struct DecodedCert * cert)This function frees a DecodedCert that was previously initialized with wc_InitDecodedCert.

	Name
int	**wc_SetTimeCb**(wc_time_cb f)This function registers a time callback that will be used anytime wolfSSL needs to get the current time. The prototype of the callback should be the same as the “time” function from the C standard library.
time_t	**wc_Time**(time_t * t)This function gets the current time. By default, it uses the XTIME macro, which varies between platforms. The user can use a function of their choosing instead via the wc_SetTimeCb function.
int	**wc_SetCustomExtension**(Cert * cert, int critical, const char * oid, const byte * der, word32 derSz)This function injects a custom extension in to an X.509 certificate. note: The content at the address pointed to by any of the parameters that are pointers must not be modified until the certificate is generated and you have the der output. This function does NOT copy the contents to another buffer.
int	**wc_SetUnknownExtCallback**(DecodedCert * cert, wc_UnknownExtCallback cb)This function registers a callback that will be used anytime wolfSSL encounters an unknown X.509 extension in a certificate while parsing a certificate. The prototype of the callback should be:
int	**wc_CheckCertSigPubKey**(const byte * cert, word32 certSz, void * heap, const byte * pubKey, word32 pubKeySz, int pubKeyOID)This function verifies the signature in the der form of an X.509 certificate against a public key. The public key is expected to be the full subject public key info in der form.
int	**wc_Asn1PrintOptions_Init**(Asn1PrintOptions * opts)This function initializes the ASN.1 print options.
int	**wc_Asn1PrintOptions_Set**(Asn1PrintOptions * opts, enum Asn1PrintOpt opt, word32 val)This function sets a print option into an ASN.1 print options object.
int	**wc_Asn1_Init**(Asn1 * asn1)This function initializes an ASN.1 parsing object.
int	**wc_Asn1_SetFile**(Asn1 * asn1, XFILE file)This function sets the file to use when printing into an ASN.1 parsing object.
int	**wc_Asn1_PrintAll**(Asn1 * asn1, Asn1PrintOptions * opts, unsigned char * data, word32 len)Print all ASN.1 items.

B.1.2 Functions Documentation

```
int wc_InitCert(
    Cert *
)
```

This function initializes a default cert, with the default options: version = 3 (0x2), serial = 0, sigType = SHA_WITH_RSA, issuer = blank, daysValid = 500, selfSigned = 1 (true) use subject as issuer, subject = blank.

Parameters:

- **cert** pointer to an uninitialized cert structure to initialize

See:

- wc_MakeCert
- wc_MakeCertReq

Return: none No returns.

Example

```
Cert myCert;
wc_InitCert(&myCert);
```

```
Cert * wc_CertNew(
    void * heap
)
```

This function allocates a new Cert structure for use during cert operations without the application having to allocate the structure itself. The Cert structure is also initialized by this function thus removing the need to call wc_InitCert(). When the application is finished using the allocated Cert structure wc_CertFree() must be called.

Parameters:

- **A** pointer to the heap used for dynamic allocation. Can be NULL.

See:

- wc_InitCert
- wc_MakeCert
- wc_CertFree

Return:

- pointer If successful the call will return a pointer to the newly allocated and initialized Cert.
- NULL On a memory allocation failure.

Example

```
Cert*   myCert;

myCert = wc_CertNew(NULL);
if (myCert == NULL) {
    // Cert creation failure
}
```

```
void wc_CertFree(
    Cert * cert
)
```

This function frees the memory allocated for a cert structure by a previous call to wc_CertNew().

Parameters:

- **A** pointer to the cert structure to free.

See:

- wc_InitCert
- wc_MakeCert
- wc_CertNew

Return: None.

Example

```
Cert*   myCert;

myCert = wc_CertNew(NULL);

// Perform cert operations.

wc_CertFree(myCert);
```

```
int wc_MakeCert(
    Cert * cert,
    byte * derBuffer,
    word32 derSz,
    RsaKey * rsaKey,
    ecc_key * eccKey,
    WC_RNG * rng
)
```

Used to make CA signed certs. Called after the subject information has been entered. This function makes an x509 Certificate v3 RSA or ECC from a cert input. It then writes this cert to derBuffer. It takes in either an rsaKey or an eccKey to generate the certificate. The certificate must be initialized with wc_InitCert before this method is called.

Parameters:

- **cert** pointer to an initialized cert structure
- **derBuffer** pointer to the buffer in which to hold the generated cert
- **derSz** size of the buffer in which to store the cert
- **rsaKey** pointer to an RsaKey structure containing the rsa key used to generate the certificate
- **eccKey** pointer to an EccKey structure containing the ecc key used to generate the certificate
- **rng** pointer to the random number generator used to make the cert

See:

- wc_InitCert
- wc_MakeCertReq

Return:

- Success On successfully making an x509 certificate from the specified input cert, returns the size of the cert generated.
- MEMORY_E Returned if there is an error allocating memory with XMALLOC
- BUFFER_E Returned if the provided derBuffer is too small to store the generated certificate
- Others Additional error messages may be returned if the cert generation is not successful.

Example

```
Cert myCert;
wc_InitCert(&myCert);
WC_RNG rng;
//initialize rng;
RsaKey key;
//initialize key;
byte * derCert = malloc(FOURK_BUF);
word32 certSz;
certSz = wc_MakeCert(&myCert, derCert, FOURK_BUF, &key, NULL, &rng);
```

```
int wc_MakeCertReq(
    Cert * cert,
    byte * derBuffer,
    word32 derSz,
    RsaKey * rsaKey,
    ecc_key * eccKey
)
```

This function makes a certificate signing request using the input certificate and writes the output to derBuffer. It takes in either an rsaKey or an eccKey to generate the certificate request. wc_SignCert() will need to be called after this function to sign the certificate request. Please see the wolfCrypt test application (./wolfcrypt/test/test.c) for an example usage of this function.

Parameters:

- **cert** pointer to an initialized cert structure
- **derBuffer** pointer to the buffer in which to hold the generated certificate request

- **derSz** size of the buffer in which to store the certificate request
- **rsaKey** pointer to an RsaKey structure containing the rsa key used to generate the certificate request
- **eccKey** pointer to an EccKey structure containing the ecc key used to generate the certificate request

See:

- wc_InitCert
- wc_MakeCert

Return:

- Success On successfully making an X.509 certificate request from the specified input cert, returns the size of the certificate request generated.
- MEMORY_E Returned if there is an error allocating memory with XMALLOC
- BUFFER_E Returned if the provided derBuffer is too small to store the generated certificate
- Other Additional error messages may be returned if the certificate request generation is not successful.

Example

```
Cert myCert;
// initialize myCert
EccKey key;
//initialize key;
byte* derCert = (byte*)malloc(FOURK_BUF);

word32 certSz;
certSz = wc_MakeCertReq(&myCert, derCert, FOURK_BUF, NULL, &key);
```

```
int wc_SignCert(
    int requestSz,
    int sigType,
    byte * derBuffer,
    word32 derSz,
    RsaKey * rsaKey,
    ecc_key * eccKey,
    WC_RNG * rng
)
```

This function signs buffer and adds the signature to the end of buffer. It takes in a signature type. Must be called after wc_MakeCert() if creating a CA signed cert.

Parameters:

- **requestSz** the size of the certificate body we're requesting to have signed
- **sType** Type of signature to create. Valid options are: CTC_MD5wRSA, CTC_SHAwRSA, CTC_SHAwECDSA, CTC_SHA256wECDSA, and CTC_SHA256wRSA

- **buffer** pointer to the buffer containing the certificate to be signed. On success: will hold the newly signed certificate
- **buffSz** the (total) size of the buffer in which to store the newly signed certificate
- **rsaKey** pointer to an RsaKey structure containing the rsa key to used to sign the certificate
- **eccKey** pointer to an EccKey structure containing the ecc key to used to sign the certificate
- **rng** pointer to the random number generator used to sign the certificate

See:

- wc_InitCert
- wc_MakeCert

Return:

- Success On successfully signing the certificate, returns the new size of the cert (including signature).
- MEMORY_E Returned if there is an error allocating memory with XMALLOC
- BUFFER_E Returned if the provided buffer is too small to store the generated certificate
- Other Additional error messages may be returned if the cert generation is not successful.

Example

```
Cert myCert;
byte* derCert = (byte*)malloc(FOURK_BUF);
// initialize myCert, derCert
RsaKey key;
// initialize key;
WC_RNG rng;
// initialize rng

word32 certSz;
certSz = wc_SignCert(myCert.bodySz, myCert.sigType,derCert,FOURK_BUF,
&key, NULL,
&rng);
```

```
int wc_MakeSelfCert(
    Cert * cert,
    byte * derBuffer,
    word32 derSz,
    RsaKey * key,
    WC_RNG * rng
)
```

This function is a combination of the previous two functions, wc_MakeCert and wc_SignCert for self signing (the previous functions may be used for CA requests). It makes a certificate, and then signs it, generating a self-signed certificate.

Parameters:

- **cert** pointer to the cert to make and sign

- **buffer** pointer to the buffer in which to hold the signed certificate
- **buffSz** size of the buffer in which to store the signed certificate
- **key** pointer to an RsaKey structure containing the rsa key to used to sign the certificate
- **rng** pointer to the random number generator used to generate and sign the certificate

See:

- wc_InitCert
- wc_MakeCert
- wc_SignCert

Return:

- Success On successfully signing the certificate, returns the new size of the cert.
- MEMORY_E Returned if there is an error allocating memory with XMALLOC
- BUFFER_E Returned if the provided buffer is too small to store the generated certificate
- Other Additional error messages may be returned if the cert generation is not successful.

Example

```
Cert myCert;
byte* derCert = (byte*)malloc(FOURK_BUF);
// initialize myCert, derCert
RsaKey key;
// initialize key;
WC_RNG rng;
// initialize rng

word32 certSz;
certSz = wc_MakeSelfCert(&myCert, derCert, FOURK_BUF, &key, NULL, &rng);
```

```
int wc_SetIssuer(
    Cert * cert,
    const char * issuerFile
)
```

This function sets the issuer for a certificate to the issuer in the provided pem issuerFile. It also changes the certificate's self-signed attribute to false. The issuer specified in issuerFile is verified prior to setting the cert issuer. This method is used to set fields prior to signing.

Parameters:

- **cert** pointer to the cert for which to set the issuer
- **issuerFile** path of the file containing the pem formatted certificate

See:

- wc_InitCert
- wc_SetSubject
- wc_SetIssuerBuffer

Return:

- 0 Returned on successfully setting the issuer for the certificate
- MEMORY_E Returned if there is an error allocating memory with XMALLOC
- ASN_PARSE_E Returned if there is an error parsing the cert header file
- ASN_OBJECT_ID_E Returned if there is an error parsing the encryption type from the cert
- ASN_EXPECT_0_E Returned if there is a formatting error in the encryption specification of the cert file
- ASN_BEFORE_DATE_E Returned if the date is before the certificate start date
- ASN_AFTER_DATE_E Returned if the date is after the certificate expiration date
- ASN_BITSTR_E Returned if there is an error parsing a bit string from the certificate
- ECC_CURVE_OID_E Returned if there is an error parsing the ECC key from the certificate
- ASN_UNKNOWN_OID_E Returned if the certificate is using an unknown key object id
- ASN_VERSION_E Returned if the ALLOW_V1_EXTENSIONS option is not defined and the certificate is a V1 or V2 certificate
- BAD_FUNC_ARG Returned if there is an error processing the certificate extension
- ASN_CRIT_EXT_E Returned if an unfamiliar critical extension is encountered in processing the certificate
- ASN_SIG_OID_E Returned if the signature encryption type is not the same as the encryption type of the certificate in the provided file
- ASN_SIG_CONFIRM_E Returned if confirming the certification signature fails
- ASN_NAME_INVALID_E Returned if the certificate's name is not permitted by the CA name constraints
- ASN_NO_SIGNER_E Returned if there is no CA signer to verify the certificate's authenticity

Example

```
Cert myCert;
// initialize myCert
if(wc_SetIssuer(&myCert, "./path/to/ca-cert.pem") != 0) {
    // error setting issuer
}
```

```
int wc_SetSubject(
    Cert * cert,
    const char * subjectFile
)
```

This function sets the subject for a certificate to the subject in the provided pem subjectFile. This method is used to set fields prior to signing.

Parameters:

- **cert** pointer to the cert for which to set the issuer
- **subjectFile** path of the file containing the pem formatted certificate

See:

- wc_InitCert
- wc_SetIssuer

Return:

- 0 Returned on successfully setting the issuer for the certificate
- MEMORY_E Returned if there is an error allocating memory with XMALLOC
- ASN_PARSE_E Returned if there is an error parsing the cert header file
- ASN_OBJECT_ID_E Returned if there is an error parsing the encryption type from the cert
- ASN_EXPECT_0_E Returned if there is a formatting error in the encryption specification of the cert file
- ASN_BEFORE_DATE_E Returned if the date is before the certificate start date
- ASN_AFTER_DATE_E Returned if the date is after the certificate expiration date
- ASN_BITSTR_E Returned if there is an error parsing a bit string from the certificate
- ECC_CURVE_OID_E Returned if there is an error parsing the ECC key from the certificate
- ASN_UNKNOWN_OID_E Returned if the certificate is using an unknown key object id
- ASN_VERSION_E Returned if the ALLOW_V1_EXTENSIONS option is not defined and the certificate is a V1 or V2 certificate
- BAD_FUNC_ARG Returned if there is an error processing the certificate extension
- ASN_CRIT_EXT_E Returned if an unfamiliar critical extension is encountered in processing the certificate
- ASN_SIG_OID_E Returned if the signature encryption type is not the same as the encryption type of the certificate in the provided file
- ASN_SIG_CONFIRM_E Returned if confirming the certification signature fails
- ASN_NAME_INVALID_E Returned if the certificate's name is not permitted by the CA name constraints
- ASN_NO_SIGNER_E Returned if there is no CA signer to verify the certificate's authenticity

Example

```
Cert myCert;
// initialize myCert
if(wc_SetSubject(&myCert, "./path/to/ca-cert.pem") != 0) {
    // error setting subject
}
```

```
int wc_SetSubjectRaw(
    Cert * cert,
    const byte * der,
    int derSz
)
```

This function sets the raw subject for a certificate from the subject in the provided der buffer. This method is used to set the raw subject field prior to signing.

Parameters:

- **cert** pointer to the cert for which to set the raw subject
- **der** pointer to the buffer containing the der formatted certificate from which to grab the subject
- **derSz** size of the buffer containing the der formatted certificate from which to grab the subject

See:

- wc_InitCert
- wc_SetSubject

Return:

- 0 Returned on successfully setting the subject for the certificate
- MEMORY_E Returned if there is an error allocating memory with XMALLOC
- ASN_PARSE_E Returned if there is an error parsing the cert header file
- ASN_OBJECT_ID_E Returned if there is an error parsing the encryption type from the cert
- ASN_EXPECT_0_E Returned if there is a formatting error in the encryption specification of the cert file
- ASN_BEFORE_DATE_E Returned if the date is before the certificate start date
- ASN_AFTER_DATE_E Returned if the date is after the certificate expiration date
- ASN_BITSTR_E Returned if there is an error parsing a bit string from the certificate
- ECC_CURVE_OID_E Returned if there is an error parsing the ECC key from the certificate
- ASN_UNKNOWN_OID_E Returned if the certificate is using an unknown key object id
- ASN_VERSION_E Returned if the ALLOW_V1_EXTENSIONS option is not defined and the certificate is a V1 or V2 certificate
- BAD_FUNC_ARG Returned if there is an error processing the certificate extension
- ASN_CRIT_EXT_E Returned if an unfamiliar critical extension is encountered in processing the certificate
- ASN_SIG_OID_E Returned if the signature encryption type is not the same as the encryption type of the certificate in the provided file
- ASN_SIG_CONFIRM_E Returned if confirming the certification signature fails
- ASN_NAME_INVALID_E Returned if the certificate's name is not permitted by the CA name constraints
- ASN_NO_SIGNER_E Returned if there is no CA signer to verify the certificate's authenticity

Example

```
Cert myCert;
// initialize myCert
byte* der;
der = (byte*)malloc(FOURK_BUF);
// initialize der
if(wc_SetSubjectRaw(&myCert, der, FOURK_BUF) != 0) {
    // error setting subject
}
```

```
int wc_GetSubjectRaw(
    byte ** subjectRaw,
    Cert * cert
)
```

This function gets the raw subject from the certificate structure.

Parameters:

- **subjectRaw** pointer-pointer to the raw subject upon successful return
- **cert** pointer to the cert from which to get the raw subject

See:

- wc_InitCert
- wc_SetSubjectRaw

Return:

- 0 Returned on successfully getting the subject from the certificate
- BAD_FUNC_ARG Returned if there is an error processing the certificate extension

Example

```
Cert myCert;
byte *subjRaw;
// initialize myCert

if(wc_GetSubjectRaw(&subjRaw, &myCert) != 0) {
    // error setting subject
}
```

```
int wc_SetAltNames(
    Cert * cert,
    const char * file
)
```

This function sets the alternate names for a certificate to the alternate names in the provided pem file. This is useful in the case that one wishes to secure multiple domains with the same certificate. This method is used to set fields prior to signing.

Parameters:

- **cert** pointer to the cert for which to set the alt names
- **file** path of the file containing the pem formatted certificate

See:

- wc_InitCert
- wc_SetIssuer

Return:

- 0 Returned on successfully setting the alt names for the certificate
- MEMORY_E Returned if there is an error allocating memory with XMALLOC
- ASN_PARSE_E Returned if there is an error parsing the cert header file
- ASN_OBJECT_ID_E Returned if there is an error parsing the encryption type from the cert
- ASN_EXPECT_0_E Returned if there is a formatting error in the encryption specification of the cert file
- ASN_BEFORE_DATE_E Returned if the date is before the certificate start date
- ASN_AFTER_DATE_E Returned if the date is after the certificate expiration date

- ASN_BITSTR_E Returned if there is an error parsing a bit string from the certificate
- ECC_CURVE_OID_E Returned if there is an error parsing the ECC key from the certificate
- ASN_UNKNOWN_OID_E Returned if the certificate is using an unknown key object id
- ASN_VERSION_E Returned if the ALLOW_V1_EXTENSIONS option is not defined and the certificate is a V1 or V2 certificate
- BAD_FUNC_ARG Returned if there is an error processing the certificate extension
- ASN_CRIT_EXT_E Returned if an unfamiliar critical extension is encountered in processing the certificate
- ASN_SIG_OID_E Returned if the signature encryption type is not the same as the encryption type of the certificate in the provided file
- ASN_SIG_CONFIRM_E Returned if confirming the certification signature fails
- ASN_NAME_INVALID_E Returned if the certificate's name is not permitted by the CA name constraints
- ASN_NO_SIGNER_E Returned if there is no CA signer to verify the certificate's authenticity

Example

```
Cert myCert;
// initialize myCert
if(wc_SetSubject(&myCert, "./path/to/ca-cert.pem") != 0) {
    // error setting alt names
}
```

```
int wc_SetIssuerBuffer(
    Cert * cert,
    const byte * der,
    int derSz
)
```

This function sets the issuer for a certificate from the issuer in the provided der buffer. It also changes the certificate's self-signed attribute to false. This method is used to set fields prior to signing.

Parameters:

- **cert** pointer to the cert for which to set the issuer
- **der** pointer to the buffer containing the der formatted certificate from which to grab the issuer
- **derSz** size of the buffer containing the der formatted certificate from which to grab the issuer

See:

- wc_InitCert
- wc_SetIssuer

Return:

- 0 Returned on successfully setting the issuer for the certificate
- MEMORY_E Returned if there is an error allocating memory with XMALLOC
- ASN_PARSE_E Returned if there is an error parsing the cert header file
- ASN_OBJECT_ID_E Returned if there is an error parsing the encryption type from the cert

- ASN_EXPECT_0_E Returned if there is a formatting error in the encryption specification of the cert file
- ASN_BEFORE_DATE_E Returned if the date is before the certificate start date
- ASN_AFTER_DATE_E Returned if the date is after the certificate expiration date
- ASN_BITSTR_E Returned if there is an error parsing a bit string from the certificate
- ECC_CURVE_OID_E Returned if there is an error parsing the ECC key from the certificate
- ASN_UNKNOWN_OID_E Returned if the certificate is using an unknown key object id
- ASN_VERSION_E Returned if the ALLOW_V1_EXTENSIONS option is not defined and the certificate is a V1 or V2 certificate
- BAD_FUNC_ARG Returned if there is an error processing the certificate extension
- ASN_CRIT_EXT_E Returned if an unfamiliar critical extension is encountered in processing the certificate
- ASN_SIG_OID_E Returned if the signature encryption type is not the same as the encryption type of the certificate in the provided file
- ASN_SIG_CONFIRM_E Returned if confirming the certification signature fails
- ASN_NAME_INVALID_E Returned if the certificate's name is not permitted by the CA name constraints
- ASN_NO_SIGNER_E Returned if there is no CA signer to verify the certificate's authenticity

Example

```
Cert myCert;
// initialize myCert
byte* der;
der = (byte*)malloc(FOURK_BUF);
// initialize der
if(wc_SetIssuerBuffer(&myCert, der, FOURK_BUF) != 0) {
    // error setting issuer
}
```

```
int wc_SetIssuerRaw(
    Cert * cert,
    const byte * der,
    int derSz
)
```

This function sets the raw issuer for a certificate from the issuer in the provided der buffer. This method is used to set the raw issuer field prior to signing.

Parameters:

- **cert** pointer to the cert for which to set the raw issuer
- **der** pointer to the buffer containing the der formatted certificate from which to grab the subject
- **derSz** size of the buffer containing the der formatted certificate from which to grab the subject

See:

- wc_InitCert
- wc_SetIssuer

Return:

- 0 Returned on successfully setting the issuer for the certificate
- MEMORY_E Returned if there is an error allocating memory with XMALLOC
- ASN_PARSE_E Returned if there is an error parsing the cert header file
- ASN_OBJECT_ID_E Returned if there is an error parsing the encryption type from the cert
- ASN_EXPECT_0_E Returned if there is a formatting error in the encryption specification of the cert file
- ASN_BEFORE_DATE_E Returned if the date is before the certificate start date
- ASN_AFTER_DATE_E Returned if the date is after the certificate expiration date
- ASN_BITSTR_E Returned if there is an error parsing a bit string from the certificate
- ECC_CURVE_OID_E Returned if there is an error parsing the ECC key from the certificate
- ASN_UNKNOWN_OID_E Returned if the certificate is using an unknown key object id
- ASN_VERSION_E Returned if the ALLOW_V1_EXTENSIONS option is not defined and the certificate is a V1 or V2 certificate
- BAD_FUNC_ARG Returned if there is an error processing the certificate extension
- ASN_CRIT_EXT_E Returned if an unfamiliar critical extension is encountered in processing the certificate
- ASN_SIG_OID_E Returned if the signature encryption type is not the same as the encryption type of the certificate in the provided file
- ASN_SIG_CONFIRM_E Returned if confirming the certification signature fails
- ASN_NAME_INVALID_E Returned if the certificate's name is not permitted by the CA name constraints
- ASN_NO_SIGNER_E Returned if there is no CA signer to verify the certificate's authenticity

Example

```
Cert myCert;
// initialize myCert
byte* der;
der = (byte*)malloc(FOURK_BUF);
// initialize der
if(wc_SetIssuerRaw(&myCert, der, FOURK_BUF) != 0) {
    // error setting subject
}
```

```
int wc_SetSubjectBuffer(
    Cert * cert,
    const byte * der,
    int derSz
)
```

This function sets the subject for a certificate from the subject in the provided der buffer. This method is used to set fields prior to signing.

Parameters:

- **cert** pointer to the cert for which to set the subject
- **der** pointer to the buffer containing the der formatted certificate from which to grab the subject
- **derSz** size of the buffer containing the der formatted certificate from which to grab the subject

See:

- wc_InitCert
- wc_SetSubject

Return:

- 0 Returned on successfully setting the subject for the certificate
- MEMORY_E Returned if there is an error allocating memory with XMALLOC
- ASN_PARSE_E Returned if there is an error parsing the cert header file
- ASN_OBJECT_ID_E Returned if there is an error parsing the encryption type from the cert
- ASN_EXPECT_0_E Returned if there is a formatting error in the encryption specification of the cert file
- ASN_BEFORE_DATE_E Returned if the date is before the certificate start date
- ASN_AFTER_DATE_E Returned if the date is after the certificate expiration date
- ASN_BITSTR_E Returned if there is an error parsing a bit string from the certificate
- ECC_CURVE_OID_E Returned if there is an error parsing the ECC key from the certificate
- ASN_UNKNOWN_OID_E Returned if the certificate is using an unknown key object id
- ASN_VERSION_E Returned if the ALLOW_V1_EXTENSIONS option is not defined and the certificate is a V1 or V2 certificate
- BAD_FUNC_ARG Returned if there is an error processing the certificate extension
- ASN_CRIT_EXT_E Returned if an unfamiliar critical extension is encountered in processing the certificate
- ASN_SIG_OID_E Returned if the signature encryption type is not the same as the encryption type of the certificate in the provided file
- ASN_SIG_CONFIRM_E Returned if confirming the certification signature fails
- ASN_NAME_INVALID_E Returned if the certificate's name is not permitted by the CA name constraints
- ASN_NO_SIGNER_E Returned if there is no CA signer to verify the certificate's authenticity

Example

```
Cert myCert;
// initialize myCert
byte* der;
der = (byte*)malloc(FOURK_BUF);
// initialize der
if(wc_SetSubjectBuffer(&myCert, der, FOURK_BUF) != 0) {
    // error setting subject
}
```

```
int wc_SetAltNamesBuffer(
    Cert * cert,
    const byte * der,
    int derSz
)
```

This function sets the alternate names for a certificate from the alternate names in the provided der buffer. This is useful in the case that one wishes to secure multiple domains with the same certificate. This method is used to set fields prior to signing.

Parameters:

- **cert** pointer to the cert for which to set the alternate names
- **der** pointer to the buffer containing the der formatted certificate from which to grab the alternate names
- **derSz** size of the buffer containing the der formatted certificate from which to grab the alternate names

See:

- wc_InitCert
- wc_SetAltNames

Return:

- 0 Returned on successfully setting the alternate names for the certificate
- MEMORY_E Returned if there is an error allocating memory with XMALLOC
- ASN_PARSE_E Returned if there is an error parsing the cert header file
- ASN_OBJECT_ID_E Returned if there is an error parsing the encryption type from the cert
- ASN_EXPECT_0_E Returned if there is a formatting error in the encryption specification of the cert file
- ASN_BEFORE_DATE_E Returned if the date is before the certificate start date
- ASN_AFTER_DATE_E Returned if the date is after the certificate expiration date
- ASN_BITSTR_E Returned if there is an error parsing a bit string from the certificate
- ECC_CURVE_OID_E Returned if there is an error parsing the ECC key from the certificate
- ASN_UNKNOWN_OID_E Returned if the certificate is using an unknown key object id
- ASN_VERSION_E Returned if the ALLOW_V1_EXTENSIONS option is not defined and the certificate is a V1 or V2 certificate
- BAD_FUNC_ARG Returned if there is an error processing the certificate extension
- ASN_CRIT_EXT_E Returned if an unfamiliar critical extension is encountered in processing the certificate
- ASN_SIG_OID_E Returned if the signature encryption type is not the same as the encryption type of the certificate in the provided file
- ASN_SIG_CONFIRM_E Returned if confirming the certification signature fails
- ASN_NAME_INVALID_E Returned if the certificate's name is not permitted by the CA name constraints
- ASN_NO_SIGNER_E Returned if there is no CA signer to verify the certificate's authenticity

Example

```
Cert myCert;
// initialize myCert
byte* der;
der = (byte*)malloc(FOURK_BUF);
// initialize der
if(wc_SetAltNamesBuffer(&myCert, der, FOURK_BUF) != 0) {
    // error setting subject
}
```

```
int wc_SetDatesBuffer(
    Cert * cert,
```

```
    const byte * der,
    int derSz
)
```

This function sets the dates for a certificate from the date range in the provided der buffer. This method is used to set fields prior to signing.

Parameters:

- **cert** pointer to the cert for which to set the dates
- **der** pointer to the buffer containing the der formatted certificate from which to grab the date range
- **derSz** size of the buffer containing the der formatted certificate from which to grab the date range

See: wc_InitCert

Return:

- 0 Returned on successfully setting the dates for the certificate
- MEMORY_E Returned if there is an error allocating memory with XMALLOC
- ASN_PARSE_E Returned if there is an error parsing the cert header file
- ASN_OBJECT_ID_E Returned if there is an error parsing the encryption type from the cert
- ASN_EXPECT_0_E Returned if there is a formatting error in the encryption specification of the cert file
- ASN_BEFORE_DATE_E Returned if the date is before the certificate start date
- ASN_AFTER_DATE_E Returned if the date is after the certificate expiration date
- ASN_BITSTR_E Returned if there is an error parsing a bit string from the certificate
- ECC_CURVE_OID_E Returned if there is an error parsing the ECC key from the certificate
- ASN_UNKNOWN_OID_E Returned if the certificate is using an unknown key object id
- ASN_VERSION_E Returned if the ALLOW_V1_EXTENSIONS option is not defined and the certificate is a V1 or V2 certificate
- BAD_FUNC_ARG Returned if there is an error processing the certificate extension
- ASN_CRIT_EXT_E Returned if an unfamiliar critical extension is encountered in processing the certificate
- ASN_SIG_OID_E Returned if the signature encryption type is not the same as the encryption type of the certificate in the provided file
- ASN_SIG_CONFIRM_E Returned if confirming the certification signature fails
- ASN_NAME_INVALID_E Returned if the certificate's name is not permitted by the CA name constraints
- ASN_NO_SIGNER_E Returned if there is no CA signer to verify the certificate's authenticity

Example

```
Cert myCert;
// initialize myCert
byte* der;
der = (byte*)malloc(FOURK_BUF);
// initialize der
if(wc_SetDatesBuffer(&myCert, der, FOURK_BUF) != 0) {
    // error setting subject
}
```

```
int wc_SetAuthKeyIdFromPublicKey(
    Cert * cert,
    RsaKey * rsakey,
    ecc_key * eckey
)
```

Set AKID from either an RSA or ECC public key. note: Only set one of rsakey or eckey, not both.

Parameters:

- **cert** Pointer to the certificate to set the SKID.
- **rsakey** Pointer to the RsaKey struct to read from.
- **eckey** Pointer to the ecc_key to read from.

See:

- wc_SetSubjectKeyId
- wc_SetAuthKeyId
- wc_SetAuthKeyIdFromCert

Return:

- 0 Success
- BAD_FUNC_ARG Either cert is null or both rsakey and eckey are null.
- MEMORY_E Error allocating memory.
- PUBLIC_KEY_E Error writing to the key.

Example

```
Cert myCert;
RsaKey keypub;

wc_InitRsaKey(&keypub, 0);

if (wc_SetAuthKeyIdFromPublicKey(&myCert, &keypub, NULL) != 0)
{
    // Handle error
}
```

```
int wc_SetAuthKeyIdFromCert(
    Cert * cert,
    const byte * der,
    int derSz
)
```

Set AKID from from DER encoded certificate.

Parameters:

- **cert** The Cert struct to write to.

- **der** The DER encoded certificate buffer.
- **derSz** Size of der in bytes.

See:

- wc_SetAuthKeyIdFromPublicKey
- wc_SetAuthKeyId

Return:

- 0 Success
- BAD_FUNC_ARG Error if any argument is null or derSz is less than 0.
- MEMORY_E Error if problem allocating memory.
- ASN_NO_SKID No subject key ID found.

Example

```
Cert some_cert;
byte some_der[] = { // Initialize a DER buffer };
wc_InitCert(&some_cert);
if(wc_SetAuthKeyIdFromCert(&some_cert, some_der, sizeof(some_der) != 0)
{
    // Handle error
}
```

```
int wc_SetAuthKeyId(
    Cert * cert,
    const char * file
)
```

Set AKID from certificate file in PEM format.

Parameters:

- **cert** Cert struct you want to set the AKID of.
- **file** Buffer containing PEM cert file.

See:

- wc_SetAuthKeyIdFromPublicKey
- wc_SetAuthKeyIdFromCert

Return:

- 0 Success
- BAD_FUNC_ARG Error if cert or file is null.
- MEMORY_E Error if problem allocating memory.

Example

```
char* file_name = "/path/to/file";
cert some_cert;
wc_InitCert(&some_cert);

if(wc_SetAuthKeyId(&some_cert, file_name) != 0)
{
    // Handle Error
}
```

```
int wc_SetSubjectKeyIdFromPublicKey(
    Cert * cert,
    RsaKey * rsakey,
    ecc_key * eckey
)
```

Set SKID from RSA or ECC public key.

Parameters:

- **cert** Pointer to a Cert structure to be used.
- **rsakey** Pointer to an RsaKey structure
- **eckey** Pointer to an ecc_key structure

See: wc_SetSubjectKeyId

Return:

- 0 Success
- BAD_FUNC_ARG Returned if cert or rsakey and eckey are null.
- MEMORY_E Returned if there is an error allocating memory.
- PUBLIC_KEY_E Returned if there is an error getting the public key.

Example

```
Cert some_cert;
RsaKey some_key;
wc_InitCert(&some_cert);
wc_InitRsaKey(&some_key);

if(wc_SetSubjectKeyIdFromPublicKey(&some_cert,&some_key, NULL) != 0)
{
    // Handle Error
}
```

```
int wc_SetSubjectKeyId(
    Cert * cert,
    const char * file
)
```

Set SKID from public key file in PEM format. Both arguments are required.

Parameters:

- **cert** Cert structure to set the SKID of.
- **file** Contains the PEM encoded file.

See: wc_SetSubjectKeyIdFromPublicKey

Return:

- 0 Success
- BAD_FUNC_ARG Returns if cert or file is null.
- MEMORY_E Returns if there is a problem allocating memory for key.
- PUBLIC_KEY_E Returns if there is an error decoding the public key.

Example

```
const char* file_name = "path/to/file";
Cert some_cert;
wc_InitCert(&some_cert);

if(wc_SetSubjectKeyId(&some_cert, file_name) != 0)
{
    // Handle Error
}
```

```
int wc_PemPubKeyToDer(
    const char * fileName,
    unsigned char * derBuf,
    int derSz
)
```

Loads a PEM key from a file and converts to a DER encoded buffer.

Parameters:

- **fileName** Name of the file to load.
- **derBuf** Buffer for DER encoded key.
- **derSz** Size of DER buffer.

See: wc_PubKeyPemToDer

Return:

- 0 Success

- <0 Error
- SSL_BAD_FILE There is a problem with opening the file.
- MEMORY_E There is an error allocating memory for the file buffer.
- BUFFER_E derBuf is not large enough to hold the converted key.

Example

```
char* some_file = "filename";
unsigned char der[];

if(wc_PemPubKeyToDer(some_file, der, sizeof(der)) != 0)
{
    //Handle Error
}
```

```
int wc_PubKeyPemToDer(
    const unsigned char * pem,
    int pemSz,
    unsigned char * buff,
    int buffSz
)
```

Convert a PEM encoded public key to DER. Returns the number of bytes written to the buffer or a negative value for an error.

Parameters:

- **pem** PEM encoded key
- **pemSz** Size of pem
- **buff** Pointer to buffer for output.
- **buffSz** Size of buffer.

See: wc_PemPubKeyToDer

Return:

- 0 Success, number of bytes written.
- BAD_FUNC_ARG Returns if pem, buff, or buffSz are null
- <0 An error occurred in the function.

Example

```
byte some_pem[] = { Initialize with PEM key }
unsigned char out_buffer[1024]; // Ensure buffer is large enough to fit DER

if(wc_PubKeyPemToDer(some_pem, sizeof(some_pem), out_buffer,
sizeof(out_buffer)) < 0)
{
```

```
    // Handle error
}
```

```
int wc_PemCertToDer(
    const char * fileName,
    unsigned char * derBuf,
    int derSz
)
```

This function converts a pem certificate to a der certificate, and places the resulting certificate in the derBuf buffer provided.

Parameters:

- **fileName** path to the file containing a pem certificate to convert to a der certificate
- **derBuf** pointer to a char buffer in which to store the converted certificate
- **derSz** size of the char buffer in which to store the converted certificate

See: none

Return:

- Success On success returns the size of the derBuf generated
- BUFFER_E Returned if the size of derBuf is too small to hold the certificate generated
- MEMORY_E Returned if the call to XMALLOC fails

Example

```
char * file = "./certs/client-cert.pem";
int derSz;
byte* der = (byte*)XMALLOC((8*1024), NULL, DYNAMIC_TYPE_CERT);

derSz = wc_PemCertToDer(file, der, (8*1024));
if (derSz <= 0) {
    //PemCertToDer error
}
```

```
int wc_DerToPem(
    const byte * der,
    word32 derSz,
    byte * output,
    word32 outputSz,
    int type
)
```

This function converts a der formatted input certificate, contained in the der buffer, into a pem formatted output certificate, contained in the output buffer. It should be noted that this is not an in place conversion, and a separate buffer must be utilized to store the pem formatted output.

Parameters:

- **der** pointer to the buffer of the certificate to convert
- **derSz** size of the the certificate to convert
- **output** pointer to the buffer in which to store the pem formatted certificate
- **outSz** size of the buffer in which to store the pem formatted certificate
- **type** the type of certificate to generate. Valid types are: CERT_TYPE, PRIVATEKEY_TYPE, ECC_PRIVATEKEY_TYPE, and CERTREQ_TYPE.

See: wc_PemCertToDer

Return:

- Success On successfully making a pem certificate from the input der cert, returns the size of the pem cert generated.
- BAD_FUNC_ARG Returned if there is an error parsing the der file and storing it as a pem file
- MEMORY_E Returned if there is an error allocating memory with XMALLOC
- ASN_INPUT_E Returned in the case of a base64 encoding error
- BUFFER_E May be returned if the output buffer is too small to store the pem formatted certificate

Example

```
byte* der;
// initialize der with certificate
byte* pemFormatted[FOURK_BUF];

word32 pemSz;
pemSz = wc_DerToPem(der, derSz,pemFormatted,FOURK_BUF, CERT_TYPE);
```

```
int wc_DerToPemEx(
    const byte * der,
    word32 derSz,
    byte * output,
    word32 outputSz,
    byte * cipherIno,
    int type
)
```

This function converts a der formatted input certificate, contained in the der buffer, into a pem formatted output certificate, contained in the output buffer. It should be noted that this is not an in place conversion, and a separate buffer must be utilized to store the pem formatted output. Allows setting cipher info.

Parameters:

- **der** pointer to the buffer of the certificate to convert
- **derSz** size of the the certificate to convert
- **output** pointer to the buffer in which to store the pem formatted certificate
- **outSz** size of the buffer in which to store the pem formatted certificate
- **cipher_inf** Additional cipher information.
- **type** the type of certificate to generate. Valid types are: CERT_TYPE, PRIVATEKEY_TYPE, ECC_PRIVATEKEY_TYPE, and CERTREQ_TYPE.

See: wc_PemCertToDer

Return:

- Success On successfully making a pem certificate from the input der cert, returns the size of the pem cert generated.
- BAD_FUNC_ARG Returned if there is an error parsing the der file and storing it as a pem file
- MEMORY_E Returned if there is an error allocating memory with XMALLOC
- ASN_INPUT_E Returned in the case of a base64 encoding error
- BUFFER_E May be returned if the output buffer is too small to store the pem formatted certificate

Example

```
byte* der;
// initialize der with certificate
byte* pemFormatted[FOURK_BUF];

word32 pemSz;
byte* cipher_info[] { Additional cipher info. }
pemSz = wc_DerToPemEx(der, derSz, pemFormatted, FOURK_BUF, cipher_info,
 ↪  CERT_TYPE);
```

```
int wc_EccPrivateKeyDecode(
    const byte * input,
    word32 * inOutIdx,
    ecc_key * key,
    word32 inSz
)
```

This function reads in an ECC private key from the input buffer, input, parses the private key, and uses it to generate an ecc_key object, which it stores in key.

Parameters:

- **input** pointer to the buffer containing the input private key
- **inOutIdx** pointer to a word32 object containing the index in the buffer at which to start
- **key** pointer to an initialized ecc object, on which to store the decoded private key
- **inSz** size of the input buffer containing the private key

See: wc_RSA_PrivateKeyDecode

Return:

- 0 On successfully decoding the private key and storing the result in the ecc_key struct
- ASN_PARSE_E: Returned if there is an error parsing the der file and storing it as a pem file
- MEMORY_E Returned if there is an error allocating memory with XMALLOC
- BUFFER_E Returned if the certificate to convert is large than the specified max certificate size
- ASN_OBJECT_ID_E Returned if the certificate encoding has an invalid object id
- ECC_CURVE_OID_E Returned if the ECC curve of the provided key is not supported
- ECC_BAD_ARG_E Returned if there is an error in the ECC key format

- NOT_COMPILED_IN Returned if the private key is compressed, and no compression key is provided
- MP_MEM Returned if there is an error in the math library used while parsing the private key
- MP_VAL Returned if there is an error in the math library used while parsing the private key
- MP_RANGE Returned if there is an error in the math library used while parsing the private key

Example

```
int ret, idx=0;
ecc_key key; // to store key in

byte* tmp; // tmp buffer to read key from
tmp = (byte*) malloc(FOURK_BUF);

int inSz;
inSz = fread(tmp, 1, FOURK_BUF, privateKeyFile);
// read key into tmp buffer

wc_ecc_init(&key); // initialize key
ret = wc_EccPrivateKeyDecode(tmp, &idx, &key, (word32)inSz);
if(ret < 0) {
    // error decoding ecc key
}
```

```
int wc_EccKeyToDer(
    ecc_key * key,
    byte * output,
    word32 inLen
)
```

This function writes a private ECC key to der format.

Parameters:

- **key** pointer to the buffer containing the input ecc key
- **output** pointer to a buffer in which to store the der formatted key
- **inLen** the length of the buffer in which to store the der formatted key

See: wc_RsaKeyToDer

Return:

- Success On successfully writing the ECC key to der format, returns the length written to the buffer
- BAD_FUNC_ARG Returned if key or output is null, or inLen equals zero
- MEMORY_E Returned if there is an error allocating memory with XMALLOC
- BUFFER_E Returned if the converted certificate is too large to store in the output buffer
- ASN_UNKNOWN_OID_E Returned if the ECC key used is of an unknown type
- MP_MEM Returned if there is an error in the math library used while parsing the private key
- MP_VAL Returned if there is an error in the math library used while parsing the private key
- MP_RANGE Returned if there is an error in the math library used while parsing the private key

Example

```
int derSz;
ecc_key key;
// initialize and make key
byte der[FOURK_BUF];
// store der formatted key here

derSz = wc_EccKeyToDer(&key, der, FOURK_BUF);
if(derSz < 0) {
    // error converting ecc key to der buffer
}
```

```
int wc_EccPublicKeyDecode(
    const byte * input,
    word32 * inOutIdx,
    ecc_key * key,
    word32 inSz
)
```

Decodes an ECC public key from an input buffer. It will parse an ASN sequence to retrieve the ECC key.

Parameters:

- **input** Buffer containing DER encoded key to decode.
- **inOutIdx** Index to start reading input buffer from. On output, index is set to last position parsed of input buffer.
- **key** Pointer to ecc_key struct to store the public key.
- **inSz** Size of the input buffer.

See: wc_ecc_import_x963

Return:

- 0 Success
- BAD_FUNC_ARG Returns if any arguments are null.
- ASN_PARSE_E Returns if there is an error parsing
- ASN_ECC_KEY_E Returns if there is an error importing the key. See wc_ecc_import_x963 for possible reasons.

Example

```
int ret;
word32 idx = 0;
byte buff[] = { // initialize with key };
ecc_key pubKey;
wc_ecc_init(&pubKey);
if ( wc_EccPublicKeyDecode(buff, &idx, &pubKey, sizeof(buff)) != 0) {
        // error decoding key
}
```

```
int wc_EccPublicKeyToDer(
    ecc_key * key,
    byte * output,
    word32 inLen,
    int with_AlgCurve
)
```

This function converts the ECC public key to DER format. It returns the size of buffer used. The public ECC key in DER format is stored in output buffer. The with_AlgCurve flag will include a header that has the Algorithm and Curve information.

Parameters:

- **key** Pointer to ECC key
- **output** Pointer to output buffer to write to.
- **inLen** Size of buffer.
- **with_AlgCurve** a flag for when to include a header that has the Algorithm and Curve information.

See:

- wc_EccKeyToDer
- wc_EccPrivateKeyDecode

Return:

- 0 Success, size of buffer used
- BAD_FUNC_ARG Returned if output or key is null.
- LENGTH_ONLY_E Error in getting ECC public key size.
- BUFFER_E Returned when output buffer is too small.

Example

```
ecc_key key;
wc_ecc_init(&key);
WC_RNG rng;
wc_InitRng(&rng);
wc_ecc_make_key(&rng, 32, &key);
int derSz = // Some appropriate size for der;
byte der[derSz];

if(wc_EccPublicKeyToDer(&key, der, derSz, 1) < 0)
{
    // Error converting ECC public key to der
}
```

```
int wc_EccPublicKeyToDer_ex(
    ecc_key * key,
    byte * output,
    word32 inLen,
    int with_AlgCurve,
    int comp
)
```

This function converts the ECC public key to DER format. It returns the size of buffer used. The public ECC key in DER format is stored in output buffer. The with_AlgCurve flag will include a header that has the Algorithm and Curve information. The comp parameter determines if the public key will be exported as compressed.

Parameters:

- **key** Pointer to ECC key
- **output** Pointer to output buffer to write to.
- **inLen** Size of buffer.
- **with_AlgCurve** a flag for when to include a header that has the Algorithm and Curve information.
- **comp** If 1 (non-zero) the ECC public key will be written in compressed form. If 0 it will be written in an uncompressed format.

See:

- wc_EccKeyToDer
- wc_EccPublicKeyDecode

Return:

- 0 Success, size of buffer used
- BAD_FUNC_ARG Returned if output or key is null.
- LENGTH_ONLY_E Error in getting ECC public key size.
- BUFFER_E Returned when output buffer is too small.

Example

```
ecc_key key;
wc_ecc_init(&key);
WC_RNG rng;
wc_InitRng(&rng);
wc_ecc_make_key(&rng, 32, &key);
int derSz = // Some appropriate size for der;
byte der[derSz];

// Write out a compressed ECC key
if(wc_EccPublicKeyToDer_ex(&key, der, derSz, 1, 1) < 0)
{
    // Error converting ECC public key to der
}
```

```
int wc_Curve25519PrivateKeyDecode(
    const byte * input,
    word32 * inOutIdx,
    curve25519_key * key,
    word32 inSz
)
```

This function decodes a Curve25519 private key (only) from a DER encoded buffer.

Parameters:

- **input** Pointer to buffer containing DER encoded private key
- **inOutIdx** Index to start reading input buffer from. On output, index is set to last position parsed of input buffer.
- **key** Pointer to curve25519_key structure to store decoded key
- **inSz** Size of input DER buffer

See:

- wc_Curve25519KeyDecode
- wc_Curve25519PublicKeyDecode

Return:

- 0 Success
- BAD_FUNC_ARG Returns if input, inOutIdx or key is null
- ASN_PARSE_E Returns if there is an error parsing the DER encoded data
- ECC_BAD_ARG_E Returns if the key length is not CURVE25519_KEYSIZE or the DER key contains other issues despite being properly formatted.
- BUFFER_E Returns if the input buffer is too small to contain a valid DER encoded key.

Example

```
byte der[] = { // DER encoded key };
word32 idx = 0;
curve25519_key key;
wc_curve25519_init(&key);

if (wc_Curve25519PrivateKeyDecode(der, &idx, &key, sizeof(der)) != 0) {
    // Error decoding private key
}
```

```
int wc_Curve25519PublicKeyDecode(
    const byte * input,
    word32 * inOutIdx,
    curve25519_key * key,
    word32 inSz
)
```

This function decodes a Curve25519 public key (only) from a DER encoded buffer.

Parameters:

- **input** Pointer to buffer containing DER encoded public key
- **inOutIdx** Index to start reading input buffer from. On output, index is set to last position parsed of input buffer.
- **key** Pointer to curve25519_key structure to store decoded key
- **inSz** Size of input DER buffer

See:

- wc_Curve25519KeyDecode
- wc_Curve25519PrivateKeyDecode

Return:

- 0 Success
- BAD_FUNC_ARG Returns if input, inOutIdx or key is null
- ASN_PARSE_E Returns if there is an error parsing the DER encoded data
- ECC_BAD_ARG_E Returns if the key length is not CURVE25519_KEYSIZE or the DER key contains other issues despite being properly formatted.
- BUFFER_E Returns if the input buffer is too small to contain a valid DER encoded key.

Example

```
byte der[] = { // DER encoded key };
word32 idx = 0;
curve25519_key key;
wc_curve25519_init(&key);
if (wc_Curve25519PublicKeyDecode(der, &idx, &key, sizeof(der)) != 0) {
    // Error decoding public key
}
```

```
int wc_Curve25519KeyDecode(
    const byte * input,
    word32 * inOutIdx,
    curve25519_key * key,
    word32 inSz
)
```

This function decodes a Curve25519 key from a DER encoded buffer. It can decode either a private key, a public key, or both.

Parameters:

- **input** Pointer to buffer containing DER encoded key
- **inOutIdx** Index to start reading input buffer from. On output, index is set to last position parsed of input buffer.
- **key** Pointer to curve25519_key structure to store decoded key

- **inSz** Size of input DER buffer

See:

- wc_Curve25519PrivateKeyDecode
- wc_Curve25519PublicKeyDecode

Return:

- 0 Success
- BAD_FUNC_ARG Returns if input, inOutIdx or key is null
- ASN_PARSE_E Returns if there is an error parsing the DER encoded data
- ECC_BAD_ARG_E Returns if the key length is not CURVE25519_KEYSIZE or the DER key contains other issues despite being properly formatted.
- BUFFER_E Returns if the input buffer is too small to contain a valid DER encoded key.

Example

```
byte der[] = { // DER encoded key };
word32 idx = 0;
curve25519_key key;
wc_curve25519_init(&key);
if (wc_Curve25519KeyDecode(der, &idx, &key, sizeof(der)) != 0) {
    // Error decoding key
}
```

```
int wc_Curve25519PrivateKeyToDer(
    curve25519_key * key,
    byte * output,
    word32 inLen
)
```

This function encodes a Curve25519 private key to DER format. If the input key structure contains a public key, it will be ignored.

Parameters:

- **key** Pointer to curve25519_key structure containing private key to encode
- **output** Buffer to hold DER encoding
- **inLen** Size of output buffer

See:

- wc_Curve25519KeyToDer
- wc_Curve25519PublicKeyToDer

Return:

-

0 Success, length of DER encoding

- BAD_FUNC_ARG Returns if key or output is null
- MEMORY_E Returns if there is an allocation failure
- BUFFER_E Returns if output buffer is too small

Example

```
curve25519_key key;
wc_curve25519_init(&key);
...
int derSz = 128; // Some appropriate size for output DER
byte der[derSz];
wc_Curve25519PrivateKeyToDer(&key, der, derSz);
```

```
int wc_Curve25519PublicKeyToDer(
    curve25519_key * key,
    byte * output,
    word32 inLen,
    int withAlg
)
```

This function encodes a Curve25519 public key to DER format. If the input key structure contains a private key, it will be ignored.

Parameters:

- **key** Pointer to curve25519_key structure containing public key to encode
- **output** Buffer to hold DER encoding
- **inLen** Size of output buffer
- **withAlg** Whether to include algorithm identifier in the DER encoding

See:

- wc_Curve25519KeyToDer
- wc_Curve25519PrivateKeyToDer

Return:

-

0 Success, length of DER encoding

- BAD_FUNC_ARG Returns if key or output is null
- MEMORY_E Returns if there is an allocation failure
- BUFFER_E Returns if output buffer is too small

Example

```
curve25519_key key;
wc_curve25519_init(&key);
...
int derSz = 128; // Some appropriate size for output DER
byte der[derSz];
wc_Curve25519PublicKeyToDer(&key, der, derSz, 1);
```

```
int wc_Curve25519KeyToDer(
    curve25519_key * key,
    byte * output,
    word32 inLen,
    int withAlg
)
```

This function encodes a Curve25519 key to DER format. It can encode either a private key, a public key, or both.

Parameters:

- **key** Pointer to curve25519_key structure containing key to encode
- **output** Buffer to hold DER encoding
- **inLen** Size of output buffer
- **withAlg** Whether to include algorithm identifier in the DER encoding

See:

- wc_Curve25519PrivateKeyToDer
- wc_Curve25519PublicKeyToDer

Return:

- 0 Success, length of DER encoding
- BAD_FUNC_ARG Returns if key or output is null
- MEMORY_E Returns if there is an allocation failure
- BUFFER_E Returns if output buffer is too small

Example

```
curve25519_key key;
wc_curve25519_init(&key);
...
int derSz = 128; // Some appropriate size for output DER
byte der[derSz];
wc_Curve25519KeyToDer(&key, der, derSz, 1);
```

```
word32 wc_EncodeSignature(
    byte * out,
    const byte * digest,
    word32 digSz,
    int hashOID
)
```

This function encodes a digital signature into the output buffer, and returns the size of the encoded signature created.

Parameters:

- **out** pointer to the buffer where the encoded signature will be written
- **digest** pointer to the digest to use to encode the signature
- **digSz** the length of the buffer containing the digest
- **hashOID** OID identifying the hash type used to generate the signature. Valid options, depending on build configurations, are: SHAh, SHA256h, SHA384h, SHA512h, MD2h, MD5h, DESb, DES3b, CTC_MD5wRSA, CTC_SHAwRSA, CTC_SHA256wRSA, CTC_SHA384wRSA, CTC_SHA512wRSA, CTC_SHAwECDSA, CTC_SHA256wECDSA, CTC_SHA384wECDSA, and CTC_SHA512wECDSA.

See: none

Return: Success On successfully writing the encoded signature to output, returns the length written to the buffer

```
int signSz;
byte encodedSig[MAX_ENCODED_SIG_SZ];
Sha256 sha256;
// initialize sha256 for hashing

byte* dig = = (byte*)malloc(WC_SHA256_DIGEST_SIZE);
// perform hashing and hash updating so dig stores SHA-256 hash
// (see wc_InitSha256, wc_Sha256Update and wc_Sha256Final)
signSz = wc_EncodeSignature(encodedSig, dig, WC_SHA256_DIGEST_SIZE, SHA256h);
```

```
int wc_GetCTC_HashOID(
    int type
)
```

This function returns the hash OID that corresponds to a hashing type. For example, when given the type: WC_SHA512, this function returns the identifier corresponding to a SHA512 hash, SHA512h.

Parameters:

- **type** the hash type for which to find the OID. Valid options, depending on build configuration, include: WC_MD5, WC_SHA, WC_SHA256, WC_SHA384, WC_SHA512, WC_SHA3_224, WC_SHA3_256, WC_SHA3_384 or WC_SHA3_512

See: none

Return:

- Success On success, returns the OID corresponding to the appropriate hash to use with that encryption type.
- 0 Returned if an unrecognized hash type is passed in as argument.

Example

```
int hashOID;

hashOID = wc_GetCTC_HashOID(WC_SHA512);
if (hashOID == 0) {
    // WOLFSSL_SHA512 not defined
}
```

```
void wc_SetCert_Free(
    Cert * cert
)
```

This function cleans up memory and resources used by the certificate structure's decoded cert cache. When WOLFSSL_CERT_GEN_CACHE is defined the decoded cert structure is cached in the certificate structure. This allows subsequent calls to certificate set functions to avoid parsing the decoded cert on each call.

Parameters:

- **cert** pointer to an uninitialized certificate information structure.

See:

- wc_SetAuthKeyIdFromCert
- wc_SetIssuerBuffer
- wc_SetSubjectBuffer
- wc_SetSubjectRaw
- wc_SetIssuerRaw
- wc_SetAltNamesBuffer
- wc_SetDatesBuffer

Return:

- 0 on success.
- BAD_FUNC_ARG Returned if invalid pointer is passed in as argument.

Example

```
Cert cert; // Initialized certificate structure

wc_SetCert_Free(&cert);
```

```
int wc_GetPkcs8TraditionalOffset(
    byte * input,
    word32 * inOutIdx,
    word32 sz
)
```

This function finds the beginning of the traditional private key inside a PKCS#8 unencrypted buffer.

Parameters:

- **input** Buffer containing unencrypted PKCS#8 private key.
- **inOutIdx** Index into the input buffer. On input, it should be a byte offset to the beginning of the the PKCS#8 buffer. On output, it will be the byte offset to the traditional private key within the input buffer.
- **sz** The number of bytes in the input buffer.

See:

- wc_CreatePKCS8Key
- wc_EncryptPKCS8Key
- wc_DecryptPKCS8Key
- wc_CreateEncryptedPKCS8Key

Return:

- Length of traditional private key on success.
- Negative values on failure.

Example

```
byte* pkcs8Buf; // Buffer containing PKCS#8 key.
word32 idx = 0;
word32 sz; // Size of pkcs8Buf.
...
ret = wc_GetPkcs8TraditionalOffset(pkcs8Buf, &idx, sz);
// pkcs8Buf + idx is now the beginning of the traditional private key bytes.
```

```
int wc_CreatePKCS8Key(
    byte * out,
    word32 * outSz,
    byte * key,
    word32 keySz,
    int algoID,
    const byte * curveOID,
    word32 oidSz
)
```

This function takes in a DER private key and converts it to PKCS#8 format. Also used in creating PKCS#12 shrouded key bags. See RFC 5208.

Parameters:

- **out** Buffer to place result in. If NULL, required out buffer size returned in outSz.
- **outSz** Size of out buffer.
- **key** Buffer with traditional DER key.
- **keySz** Size of key buffer.
- **algoID** Algorithm ID (e.g. RSAk).
- **curveOID** ECC curve OID if used. Should be NULL for RSA keys.
- **oidSz** Size of curve OID. Is set to 0 if curveOID is NULL.

See:

- wc_GetPkcs8TraditionalOffset
- wc_EncryptPKCS8Key
- wc_DecryptPKCS8Key
- wc_CreateEncryptedPKCS8Key

Return:

- The size of the PKCS#8 key placed into out on success.
- LENGTH_ONLY_E if out is NULL, with required output buffer size in outSz.
- Other negative values on failure.

Example

```
ecc_key eccKey;              // wolfSSL ECC key object.
byte* der;                   // DER-encoded ECC key.
word32 derSize;              // Size of der.
const byte* curveOid = NULL; // OID of curve used by eccKey.
word32 curveOidSz = 0;       // Size of curve OID.
byte* pkcs8;                 // Output buffer for PKCS#8 key.
word32 pkcs8Sz;              // Size of output buffer.

derSize = wc_EccKeyDerSize(&eccKey, 1);
...
derSize = wc_EccKeyToDer(&eccKey, der, derSize);
...
ret = wc_ecc_get_oid(eccKey.dp->oidSum, &curveOid, &curveOidSz);
...
ret = wc_CreatePKCS8Key(NULL, &pkcs8Sz, der,
    derSize, ECDSAk, curveOid, curveOidSz); // Get size needed in pkcs8Sz.
...
ret = wc_CreatePKCS8Key(pkcs8, &pkcs8Sz, der,
    derSize, ECDSAk, curveOid, curveOidSz);
```

```
int wc_EncryptPKCS8Key(
    byte * key,
    word32 keySz,
    byte * out,
    word32 * outSz,
    const char * password,
```

```
    int passwordSz,
    int vPKCS,
    int pbeOid,
    int encAlgId,
    byte * salt,
    word32 saltSz,
    int itt,
    WC_RNG * rng,
    void * heap
)
```

This function takes in an unencrypted PKCS#8 DER key (e.g. one created by wc_CreatePKCS8Key) and converts it to PKCS#8 encrypted format. The resulting encrypted key can be decrypted using wc_DecryptPKCS8Key. See RFC 5208.

Parameters:

- **key** Buffer with traditional DER key.
- **keySz** Size of key buffer.
- **out** Buffer to place result in. If NULL, required out buffer size returned in outSz.
- **outSz** Size of out buffer.
- **password** The password to use for the password-based encryption algorithm.
- **passwordSz** The length of the password (not including the NULL terminator).
- **vPKCS** The PKCS version to use. Can be 1 for PKCS12 or PKCS5.
- **pbeOid** The OID of the PBE scheme to use (e.g. PBES2 or one of the OIDs for PBES1 in RFC 2898 A.3).
- **encAlgId** The encryption algorithm ID to use (e.g. AES256CBCb).
- **salt** The salt buffer to use. If NULL, a random salt will be used.
- **saltSz** The length of the salt buffer. Can be 0 if passing NULL for salt.
- **itt** The number of iterations to use for the KDF.
- **rng** A pointer to an initialized WC_RNG object.
- **heap** A pointer to the heap used for dynamic allocation. Can be NULL.

See:

- wc_GetPkcs8TraditionalOffset
- wc_CreatePKCS8Key
- wc_DecryptPKCS8Key
- wc_CreateEncryptedPKCS8Key

Return:

- The size of the encrypted key placed in out on success.
- LENGTH_ONLY_E if out is NULL, with required output buffer size in outSz.
- Other negative values on failure.

Example

```
byte* pkcs8;          // Unencrypted PKCS#8 key.
word32 pkcs8Sz;       // Size of pkcs8.
byte* pkcs8Enc;       // Encrypted PKCS#8 key.
word32 pkcs8EncSz;    // Size of pkcs8Enc.
```

```
const char* password; // Password to use for encryption.
int passwordSz;       // Length of password (not including NULL terminator).
WC_RNG rng;

// The following produces an encrypted version of pkcs8 in pkcs8Enc. The
// encryption uses password-based encryption scheme 2 (PBE2) from PKCS#5 and
// the AES cipher in CBC mode with a 256-bit key. See RFC 8018 for more on
// PKCS#5.
ret = wc_EncryptPKCS8Key(pkcs8, pkcs8Sz, pkcs8Enc, &pkcs8EncSz, password,
        passwordSz, PKCS5, PBES2, AES256CBCb, NULL, 0,
        WC_PKCS12_ITT_DEFAULT, &rng, NULL);



int wc_DecryptPKCS8Key(
    byte * input,
    word32 sz,
    const char * password,
    int passwordSz
)
```

This function takes an encrypted PKCS#8 DER key and decrypts it to PKCS#8 unencrypted DER. Undoes the encryption done by wc_EncryptPKCS8Key. See RFC5208. The input buffer is overwritten with the decrypted data.

Parameters:

- **input** On input, buffer containing encrypted PKCS#8 key. On successful output, contains the decrypted key.
- **sz** Size of the input buffer.
- **password** The password used to encrypt the key.
- **passwordSz** The length of the password (not including NULL terminator).

See:

- wc_GetPkcs8TraditionalOffset
- wc_CreatePKCS8Key
- wc_EncryptPKCS8Key
- wc_CreateEncryptedPKCS8Key

Return:

- The length of the decrypted buffer on success.
- Negative values on failure.

Example

```
byte* pkcs8Enc;       // Encrypted PKCS#8 key made with wc_EncryptPKCS8Key.
word32 pkcs8EncSz;    // Size of pkcs8Enc.
const char* password; // Password to use for decryption.
int passwordSz;       // Length of password (not including NULL terminator).
```

```
ret = wc_DecryptPKCS8Key(pkcs8Enc, pkcs8EncSz, password, passwordSz);
```

```
int wc_CreateEncryptedPKCS8Key(
    byte * key,
    word32 keySz,
    byte * out,
    word32 * outSz,
    const char * password,
    int passwordSz,
    int vPKCS,
    int pbeOid,
    int encAlgId,
    byte * salt,
    word32 saltSz,
    int itt,
    WC_RNG * rng,
    void * heap
)
```

This function takes a traditional, DER key, converts it to PKCS#8 format, and encrypts it. It uses wc_CreatePKCS8Key and wc_EncryptPKCS8Key to do this.

Parameters:

- **key** Buffer with traditional DER key.
- **keySz** Size of key buffer.
- **out** Buffer to place result in. If NULL, required out buffer size returned in outSz.
- **outSz** Size of out buffer.
- **password** The password to use for the password-based encryption algorithm.
- **passwordSz** The length of the password (not including the NULL terminator).
- **vPKCS** The PKCS version to use. Can be 1 for PKCS12 or PKCS5.
- **pbeOid** The OID of the PBE scheme to use (e.g. PBES2 or one of the OIDs for PBES1 in RFC 2898 A.3).
- **encAlgId** The encryption algorithm ID to use (e.g. AES256CBCb).
- **salt** The salt buffer to use. If NULL, a random salt will be used.
- **saltSz** The length of the salt buffer. Can be 0 if passing NULL for salt.
- **itt** The number of iterations to use for the KDF.
- **rng** A pointer to an initialized WC_RNG object.
- **heap** A pointer to the heap used for dynamic allocation. Can be NULL.

See:

- wc_GetPkcs8TraditionalOffset
- wc_CreatePKCS8Key
- wc_EncryptPKCS8Key
- wc_DecryptPKCS8Key

Return:

- The size of the encrypted key placed in out on success.

- LENGTH_ONLY_E if out is NULL, with required output buffer size in outSz.
- Other negative values on failure.

Example

```
byte* key;            // Traditional private key (DER formatted).
word32 keySz;         // Size of key.
byte* pkcs8Enc;       // Encrypted PKCS#8 key.
word32 pkcs8EncSz;    // Size of pkcs8Enc.
const char* password; // Password to use for encryption.
int passwordSz;       // Length of password (not including NULL terminator).
WC_RNG rng;

// The following produces an encrypted, PKCS#8 version of key in pkcs8Enc.
// The encryption uses password-based encryption scheme 2 (PBE2) from PKCS#5
// and the AES cipher in CBC mode with a 256-bit key. See RFC 8018 for more
// on PKCS#5.
ret = wc_CreateEncryptedPKCS8Key(key, keySz, pkcs8Enc, &pkcs8EncSz,
        password, passwordSz, PKCS5, PBES2, AES256CBCb, NULL, 0,
        WC_PKCS12_ITT_DEFAULT, &rng, NULL);
```

```
void wc_InitDecodedCert(
    struct DecodedCert * cert,
    const byte * source,
    word32 inSz,
    void * heap
)
```

This function initializes the DecodedCert pointed to by the “cert” parameter. It saves the “source” pointer to a DER-encoded certificate of length “inSz.” This certificate can be parsed by a subsequent call to wc_ParseCert.

Parameters:

- **cert** Pointer to an allocated DecodedCert object.
- **source** Pointer to a DER-encoded certificate.
- **inSz** Length of the DER-encoded certificate in bytes.
- **heap** A pointer to the heap used for dynamic allocation. Can be NULL.

See:

- wc_ParseCert
- wc_FreeDecodedCert

Example

```
DecodedCert decodedCert; // Decoded certificate object.
byte* certBuf;           // DER-encoded certificate buffer.
word32 certBufSz;        // Size of certBuf in bytes.
```

```
wc_InitDecodedCert(&decodedCert, certBuf, certBufSz, NULL);
```

```
int wc_ParseCert(
    DecodedCert * cert,
    int type,
    int verify,
    void * cm
)
```

This function parses the DER-encoded certificate saved in the DecodedCert object and populates the fields of that object. The DecodedCert must have been initialized with a prior call to wc_InitDecodedCert. This function takes an optional pointer to a CertificateManager object, which is used to populate the certificate authority information of the DecodedCert, if the CA is found in the CertificateManager.

Parameters:

- **cert** Pointer to an initialized DecodedCert object.
- **type** Type of certificate. See the CertType enum in asn_public.h.
- **verify** Flag that, if set, indicates the user wants to verify the validity of the certificate.
- **cm** An optional pointer to a CertificateManager. Can be NULL.

See:

- wc_InitDecodedCert
- wc_FreeDecodedCert

Return:

- 0 on success.
- Other negative values on failure.

Example

```
int ret;
DecodedCert decodedCert; // Decoded certificate object.
byte* certBuf;           // DER-encoded certificate buffer.
word32 certBufSz;        // Size of certBuf in bytes.

wc_InitDecodedCert(&decodedCert, certBuf, certBufSz, NULL);
ret = wc_ParseCert(&decodedCert, CERT_TYPE, NO_VERIFY, NULL);
if (ret != 0) {
    fprintf(stderr, "wc_ParseCert failed.\n");
}
```

```
void wc_FreeDecodedCert(
    struct DecodedCert * cert
)
```

This function frees a DecodedCert that was previously initialized with wc_InitDecodedCert.

Parameters:

- **cert** Pointer to an initialized DecodedCert object.

See:

- wc_InitDecodedCert
- wc_ParseCert

Example

```
int ret;
DecodedCert decodedCert; // Decoded certificate object.
byte* certBuf;           // DER-encoded certificate buffer.
word32 certBufSz;        // Size of certBuf in bytes.

wc_InitDecodedCert(&decodedCert, certBuf, certBufSz, NULL);
ret = wc_ParseCert(&decodedCert, CERT_TYPE, NO_VERIFY, NULL);
if (ret != 0) {
    fprintf(stderr, "wc_ParseCert failed.\n");
}
wc_FreeDecodedCert(&decodedCert);
```

```
int wc_SetTimeCb(
    wc_time_cb f
)
```

This function registers a time callback that will be used anytime wolfSSL needs to get the current time. The prototype of the callback should be the same as the “time” function from the C standard library.

Parameters:

- **f** function to register as the time callback.

See: wc_Time

Return: 0 Returned on success.

Example

```
int ret = 0;
// Time callback prototype
time_t my_time_cb(time_t* t);
// Register it
ret = wc_SetTimeCb(my_time_cb);
```

```
if (ret != 0) {
    // failed to set time callback
}
time_t my_time_cb(time_t* t)
{
    // custom time function
}
```

```
time_t wc_Time(
    time_t * t
)
```

This function gets the current time. By default, it uses the XTIME macro, which varies between platforms. The user can use a function of their choosing instead via the wc_SetTimeCb function.

Parameters:

- **t** Optional time_t pointer to populate with current time.

See: wc_SetTimeCb

Return: Time Current time returned on success.

Example

```
time_t currentTime = 0;
currentTime = wc_Time(NULL);
wc_Time(&currentTime);
```

```
int wc_SetCustomExtension(
    Cert * cert,
    int critical,
    const char * oid,
    const byte * der,
    word32 derSz
)
```

This function injects a custom extension in to an X.509 certificate. note: The content at the address pointed to by any of the parameters that are pointers must not be modified until the certificate is generated and you have the der output. This function does NOT copy the contents to another buffer.

Parameters:

- **cert** Pointer to an initialized DecodedCert object.
- **critical** If 0, the extension will not be marked critical, otherwise it will be marked critical.
- **oid** Dot separated oid as a string. For example “1.2.840.10045.3.1.7”
- **der** The der encoding of the content of the extension.
- **derSz** The size in bytes of the der encoding.

See:

- wc_InitCert
- wc_SetUnknownExtCallback

Return:

- 0 Returned on success.
- Other negative values on failure.

Example

```
int ret = 0;
Cert newCert;
wc_InitCert(&newCert);

// Code to setup subject, public key, issuer, and other things goes here.

ret = wc_SetCustomExtension(&newCert, 1, "1.2.3.4.5",
          (const byte *)"This is a critical extension", 28);
if (ret < 0) {
    // Failed to set the extension.
}

ret = wc_SetCustomExtension(&newCert, 0, "1.2.3.4.6",
          (const byte *)"This is NOT a critical extension", 32)
if (ret < 0) {
    // Failed to set the extension.
}

// Code to sign the certificate and then write it out goes here.
```

```
int wc_SetUnknownExtCallback(
    DecodedCert * cert,
    wc_UnknownExtCallback cb
)
```

This function registers a callback that will be used anytime wolfSSL encounters an unknown X.509 extension in a certificate while parsing a certificate. The prototype of the callback should be:

Parameters:

- **cert** the DecodedCert struct that is to be associated with this callback.
- **cb** function to register as the time callback.

See:

- ParseCert
- wc_SetCustomExtension

Return:

- 0 Returned on success.
- Other negative values on failure.

Example

```
int ret = 0;
// Unknown extension callback prototype
int myUnknownExtCallback(const word16* oid, word32 oidSz, int crit,
                         const unsigned char* der, word32 derSz);

// Register it
ret = wc_SetUnknownExtCallback(cert, myUnknownExtCallback);
if (ret != 0) {
    // failed to set the callback
}

// oid: Array of integers that are the dot separated values in an oid.
// oidSz: Number of values in oid.
// crit: Whether the extension was mark critical.
// der: The der encoding of the content of the extension.
// derSz: The size in bytes of the der encoding.
int myCustomExtCallback(const word16* oid, word32 oidSz, int crit,
                        const unsigned char* der, word32 derSz) {

    // Logic to parse extension goes here.

    // NOTE: by returning zero, we are accepting this extension and
    // informing wolfSSL that it is acceptable. If you find an extension
    // that you do not find acceptable, you should return an error. The
    // standard behavior upon encountering an unknown extension with the
    // critical flag set is to return ASN_CRIT_EXT_E. For the sake of
    // brevity, this example is always accepting every extension; you
    // should use different logic.
    return 0;
}
```

```
int wc_CheckCertSigPubKey(
    const byte * cert,
    word32 certSz,
    void * heap,
    const byte * pubKey,
    word32 pubKeySz,
    int pubKeyOID
)
```

This function verifies the signature in the der form of an X.509 certificate against a public key. The public key is expected to be the full subject public key info in der form.

Parameters:

- **cert** The der encoding of the X.509 certificate.
- **certSz** The size in bytes of cert.
- **heap** A pointer to the heap used for dynamic allocation. Can be NULL.
- **pubKey** The der encoding of the public key.
- **pubKeySz** The size in bytes of pubKey.
- **pubKeyOID** OID identifying the algorithm of the public key. (ie: ECDSAk, DSAk or RSAk)

Return:

- 0 Returned on success.
- Other negative values on failure.

```
int wc_Asn1PrintOptions_Init(
    Asn1PrintOptions * opts
)
```

This function initializes the ASN.1 print options.

Parameters:

- **opts** The ASN.1 options for printing.

See:

- wc_Asn1PrintOptions_Set
- wc_Asn1_PrintAll

Return:

- 0 on success.
- BAD_FUNC_ARG when asn1 is NULL.

Example

```
Asn1PrintOptions opt;

// Initialize ASN.1 print options before use.
wc_Asn1PrintOptions_Init(&opt);
```

```
int wc_Asn1PrintOptions_Set(
    Asn1PrintOptions * opts,
    enum Asn1PrintOpt opt,
    word32 val
)
```

This function sets a print option into an ASN.1 print options object.

Parameters:

- **opts** The ASN.1 options for printing.
- **opt** An option to set value for.
- **val** The value to set.

See:

- wc_Asn1PrintOptions_Init
- wc_Asn1_PrintAll

Return:

- 0 on success.
- BAD_FUNC_ARG when asn1 is NULL.
- BAD_FUNC_ARG when val is out of range for option.

Example

```
Asn1PrintOptions opt;

// Initialize ASN.1 print options before use.
wc_Asn1PrintOptions_Init(&opt);
// Set the number of indents when printing tag name to be 1.
wc_Asn1PrintOptions_Set(&opt, ASN1_PRINT_OPT_INDENT, 1);
```

```
int wc_Asn1_Init(
    Asn1 * asn1
)
```

This function initializes an ASN.1 parsing object.

Parameters:

- **asn1** ASN.1 parse object.

See:

- wc_Asn1_SetFile
- wc_Asn1_PrintAll

Return:

- 0 on success.
- BAD_FUNC_ARG when asn1 is NULL.

Example

```
Asn1 asn1;

// Initialize ASN.1 parse object before use.
wc_Asn1_Init(&asn1);
```

```
int wc_Asn1_SetFile(
    Asn1 * asn1,
    XFILE file
)
```

This function sets the file to use when printing into an ASN.1 parsing object.

Parameters:

- **asn1** The ASN.1 parse object.
- **file** File to print to.

See:

- wc_Asn1_Init
- wc_Asn1_PrintAll

Return:

- 0 on success.
- BAD_FUNC_ARG when asn1 is NULL.
- BAD_FUNC_ARG when file is XBADFILE.

Example

```
Asn1 asn1;

// Initialize ASN.1 parse object before use.
wc_Asn1_Init(&asn1);
// Set standard out to be the file descriptor to write to.
wc_Asn1_SetFile(&asn1, stdout);
```

```
int wc_Asn1_PrintAll(
    Asn1 * asn1,
    Asn1PrintOptions * opts,
    unsigned char * data,
    word32 len
)
```

Print all ASN.1 items.

Parameters:

- **asn1** The ASN.1 parse object.
- **opts** The ASN.1 print options.
- **data** Buffer containing BER/DER data to print.
- **len** Length of data to print in bytes.

See:

- wc_Asn1_Init
- wc_Asn1_SetFile

Return:

- 0 on success.
- BAD_FUNC_ARG when asn1 or opts is NULL.
- ASN_LEN_E when ASN.1 item's length too long.
- ASN_DEPTH_E when end offset invalid.
- ASN_PARSE_E when not all of an ASN.1 item parsed.

```
Asn1PrintOptions opts;
Asn1 asn1;
unsigned char data[] = { Initialize with DER/BER data };
word32 len = sizeof(data);

// Initialize ASN.1 print options before use.
wc_Asn1PrintOptions_Init(&opt);
// Set the number of indents when printing tag name to be 1.
wc_Asn1PrintOptions_Set(&opt, ASN1_PRINT_OPT_INDENT, 1);

// Initialize ASN.1 parse object before use.
wc_Asn1_Init(&asn1);
// Set standard out to be the file descriptor to write to.
wc_Asn1_SetFile(&asn1, stdout);
// Print all ASN.1 items in buffer with the specified print options.
wc_Asn1_PrintAll(&asn1, &opts, data, len);
```

B.2 Base Encoding

B.1.2.59 function wc_Asn1_PrintAll

B.2.1 Functions

	Name
int	**Base64_Decode**(const byte * in, word32 inLen, byte * out, word32 * outLen)This function decodes the given Base64 encoded input, in, and stores the result in the output buffer out. It also sets the size written to the output buffer in the variable outLen.
int	**Base64_Encode**(const byte * in, word32 inLen, byte * out, word32 * outLen)This function encodes the given input, in, and stores the Base64 encoded result in the output buffer out. It writes the data with the traditional ‘ ’ line endings, instead of escaped %0A line endings. Upon successfully completing, this function also sets outLen to the number of bytes written to the output buffer.
int	**Base64_EncodeEsc**(const byte * in, word32 inLen, byte * out, word32 * outLen)This function encodes the given input, in, and stores the Base64 encoded result in the output buffer out. It writes the data with %0A escaped line endings instead of ‘ ’ line endings. Upon successfully completing, this function also sets outLen to the number of bytes written to the output buffer.
int	**Base64_Encode_NoNl**(const byte * in, word32 inLen, byte * out, word32 * outLen)This function encodes the given input, in, and stores the Base64 encoded result in the output buffer out. It writes the data with no new lines. Upon successfully completing, this function also sets outLen to the number of bytes written to the output buffer.
int	**Base16_Decode**(const byte * in, word32 inLen, byte * out, word32 * outLen)This function decodes the given Base16 encoded input, in, and stores the result in the output buffer out. It also sets the size written to the output buffer in the variable outLen.
int	**Base16_Encode**(const byte * in, word32 inLen, byte * out, word32 * outLen)Encode input to base16 output.

B.2.2 Functions Documentation

```
int Base64_Decode(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 * outLen
)
```

This function decodes the given Base64 encoded input, in, and stores the result in the output buffer

out. It also sets the size written to the output buffer in the variable outLen.

Parameters:

- **in** pointer to the input buffer to decode
- **inLen** length of the input buffer to decode
- **out** pointer to the output buffer in which to store the decoded message
- **outLen** pointer to the length of the output buffer. Updated with the bytes written at the end of the function call

See:

- Base64_Encode
- Base16_Decode

Return:

- 0 Returned upon successfully decoding the Base64 encoded input
- BAD_FUNC_ARG Returned if the output buffer is too small to store the decoded input
- ASN_INPUT_E Returned if a character in the input buffer falls outside of the Base64 range ([A-Za-z0-9+/=]) or if there is an invalid line ending in the Base64 encoded input

Example

```
byte encoded[] = { // initialize text to decode };
byte decoded[sizeof(encoded)];
// requires at least (sizeof(encoded) * 3 + 3) / 4 room

int outLen = sizeof(decoded);

if( Base64_Decode(encoded,sizeof(encoded), decoded, &outLen) != 0 ) {
    // error decoding input buffer
}
```

```
int Base64_Encode(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 * outLen
)
```

This function encodes the given input, in, and stores the Base64 encoded result in the output buffer out. It writes the data with the traditional ‘ ’ line endings, instead of escaped %0A line endings. Upon successfully completing, this function also sets outLen to the number of bytes written to the output buffer.

Parameters:

- **in** pointer to the input buffer to encode
- **inLen** length of the input buffer to encode

- **out** pointer to the output buffer in which to store the encoded message
- **outLen** pointer to the length of the output buffer in which to store the encoded message

See:

- Base64_EncodeEsc
- Base64_Decode

Return:

- 0 Returned upon successfully decoding the Base64 encoded input
- BAD_FUNC_ARG Returned if the output buffer is too small to store the encoded input
- BUFFER_E Returned if the output buffer runs out of room while encoding

Example

```
byte plain[] = { // initialize text to encode };
byte encoded[MAX_BUFFER_SIZE];

int outLen = sizeof(encoded);

if( Base64_Encode(plain, sizeof(plain), encoded, &outLen) != 0 ) {
    // error encoding input buffer
}
```

```
int Base64_EncodeEsc(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 * outLen
)
```

This function encodes the given input, in, and stores the Base64 encoded result in the output buffer out. It writes the data with %0A escaped line endings instead of ‘ ’ line endings. Upon successfully completing, this function also sets outLen to the number of bytes written to the output buffer.

Parameters:

- **in** pointer to the input buffer to encode
- **inLen** length of the input buffer to encode
- **out** pointer to the output buffer in which to store the encoded message
- **outLen** pointer to the length of the output buffer in which to store the encoded message

See:

- Base64_Encode
- Base64_Decode

Return:

- 0 Returned upon successfully decoding the Base64 encoded input
- BAD_FUNC_ARG Returned if the output buffer is too small to store the encoded input
- BUFFER_E Returned if the output buffer runs out of room while encoding
- ASN_INPUT_E Returned if there is an error processing the decode on the input message

Example

```
byte plain[] = { // initialize text to encode };
byte encoded[MAX_BUFFER_SIZE];

int outLen = sizeof(encoded);

if( Base64_EncodeEsc(plain, sizeof(plain), encoded, &outLen) != 0 ) {
    // error encoding input buffer
}
```

```
int Base64_Encode_NoNl(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 * outLen
)
```

This function encodes the given input, in, and stores the Base64 encoded result in the output buffer out. It writes the data with no new lines. Upon successfully completing, this function also sets outLen to the number of bytes written to the output buffer.

Parameters:

- **in** pointer to the input buffer to encode
- **inLen** length of the input buffer to encode
- **out** pointer to the output buffer in which to store the encoded message
- **outLen** pointer to the length of the output buffer in which to store the encoded message

See:

- Base64_Encode
- Base64_Decode

Return:

- 0 Returned upon successfully decoding the Base64 encoded input
- BAD_FUNC_ARG Returned if the output buffer is too small to store the encoded input
- BUFFER_E Returned if the output buffer runs out of room while encoding
- ASN_INPUT_E Returned if there is an error processing the decode on the input message

Example

```
byte plain[] = { // initialize text to encode };
byte encoded[MAX_BUFFER_SIZE];
int outLen = sizeof(encoded);
if( Base64_Encode_NoNl(plain, sizeof(plain), encoded, &outLen) != 0 ) {
    // error encoding input buffer
}
```

```
int Base16_Decode(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 * outLen
)
```

This function decodes the given Base16 encoded input, in, and stores the result in the output buffer out. It also sets the size written to the output buffer in the variable outLen.

Parameters:

- **in** pointer to the input buffer to decode
- **inLen** length of the input buffer to decode
- **out** pointer to the output buffer in which to store the decoded message
- **outLen** pointer to the length of the output buffer. Updated with the bytes written at the end of the function call

See:

- Base64_Encode
- Base64_Decode
- Base16_Encode

Return:

- 0 Returned upon successfully decoding the Base16 encoded input
- BAD_FUNC_ARG Returned if the output buffer is too small to store the decoded input or if the input length is not a multiple of two
- ASN_INPUT_E Returned if a character in the input buffer falls outside of the Base16 range ([0-9A-F])

Example

```
byte encoded[] = { // initialize text to decode };
byte decoded[sizeof(encoded)];
int outLen = sizeof(decoded);

if( Base16_Decode(encoded,sizeof(encoded), decoded, &outLen) != 0 ) {
    // error decoding input buffer
}
```

```
int Base16_Encode(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 * outLen
)
```

Encode input to base16 output.

Parameters:

- **in** Pointer to input buffer to be encoded.
- **inLen** Length of input buffer.
- **out** Pointer to output buffer.
- **outLen** Length of output buffer. Is set to len of encoded output.

See:

- Base64_Encode
- Base64_Decode
- Base16_Decode

Return:

- 0 Success
- BAD_FUNC_ARG Returns if in, out, or outLen is null or if outLen is less than 2 times inLen plus 1.

Example

```
byte in[] = { // Contents of something to be encoded };
byte out[NECESSARY_OUTPUT_SIZE];
word32 outSz = sizeof(out);

if(Base16_Encode(in, sizeof(in), out, &outSz) != 0)
{
    // Handle encode error
}
```

B.3 Compression

B.2.2.6 function Base16_Encode

B.3.1 Functions

	Name
int	**wc_Compress**(byte * out, word32 outSz, const byte * in, word32 inSz, word32 flags)This function compresses the given input data using Huffman coding and stores the output in out. Note that the output buffer should still be larger than the input buffer because there exists a certain input for which there will be no compression possible, which will still require a lookup table. It is recommended that one allocate srcSz + 0.1% + 12 for the output buffer.
int	**wc_DeCompress**(byte * out, word32 outSz, const byte * in, word32 inSz)This function decompresses the given compressed data using Huffman coding and stores the output in out.

B.3.2 Functions Documentation

```
int wc_Compress(
    byte * out,
    word32 outSz,
    const byte * in,
    word32 inSz,
    word32 flags
)
```

This function compresses the given input data using Huffman coding and stores the output in out. Note that the output buffer should still be larger than the input buffer because there exists a certain input for which there will be no compression possible, which will still require a lookup table. It is recommended that one allocate srcSz + 0.1% + 12 for the output buffer.

Parameters:

- **out** pointer to the output buffer in which to store the compressed data
- **outSz** size available in the output buffer for storage
- **in** pointer to the buffer containing the message to compress
- **inSz** size of the input message to compress
- **flags** flags to control how compression operates. Use 0 for normal decompression

See: wc_DeCompress

Return:

- On successfully compressing the input data, returns the number of bytes stored in the output buffer
- COMPRESS_INIT_E Returned if there is an error initializing the stream for compression
- COMPRESS_E Returned if an error occurs during compression

Example

```
byte message[] = { // initialize text to compress };
byte compressed[(sizeof(message) + sizeof(message) * .001 + 12 )];
// Recommends at least srcSz + .1% + 12

if( wc_Compress(compressed, sizeof(compressed), message, sizeof(message),
0) != 0){
    // error compressing data
}
```

```
int wc_DeCompress(
    byte * out,
    word32 outSz,
    const byte * in,
    word32 inSz
)
```

This function decompresses the given compressed data using Huffman coding and stores the output in out.

Parameters:

- **out** pointer to the output buffer in which to store the decompressed data
- **outSz** size available in the output buffer for storage
- **in** pointer to the buffer containing the message to decompress
- **inSz** size of the input message to decompress

See: wc_Compress

Return:

- Success On successfully decompressing the input data, returns the number of bytes stored in the output buffer
- COMPRESS_INIT_E: Returned if there is an error initializing the stream for compression
- COMPRESS_E: Returned if an error occurs during compression

Example

```
byte compressed[] = { // initialize compressed message };
byte decompressed[MAX_MESSAGE_SIZE];

if( wc_DeCompress(decompressed, sizeof(decompressed),
compressed, sizeof(compressed)) != 0 ) {
    // error decompressing data
}
```

B.4 Error Reporting

B.3.2.2 function wc_DeCompress

B.4.1 Functions

	Name
void	**wc_ErrorString**(int err, char * buff)This function stores the error string for a particular error code in the given buffer.
const char *	**wc_GetErrorString**(int error)This function returns the error string for a particular error code.

B.4.2 Functions Documentation

```
void wc_ErrorString(
    int err,
    char * buff
)
```

This function stores the error string for a particular error code in the given buffer.

Parameters:

- **error** error code for which to get the string
- **buffer** buffer in which to store the error string. Buffer should be at least WOLFSSL_MAX_ERROR_SZ (80 bytes) long

See: wc_GetErrorString

Return: none No returns.

Example

```
char errorMsg[WOLFSSL_MAX_ERROR_SZ];
int err = wc_some_function();

if( err != 0) { // error occurred
    wc_ErrorString(err, errorMsg);
}
```

```
const char * wc_GetErrorString(
    int error
)
```

This function returns the error string for a particular error code.

Parameters:

- **error** error code for which to get the string

See: wc_ErrorString

Return: string Returns the error string for an error code as a string literal.

Example

```
char * errorMsg;
int err = wc_some_function();

if( err != 0) { // error occurred
    errorMsg = wc_GetErrorString(err);
}
```

B.5 IoT-Safe Module

B.4.2.2 function wc_GetErrorString More...

B.5.1 Functions

	Name
int	**wolfSSL_CTX_iotsafe_enable**(WOLFSSL_CTX * ctx)This function enables the IoT-Safe support on the given context.
int	**wolfSSL_iotsafe_on**(WOLFSSL * ssl, byte privkey_id, byte ecdh_keypair_slot, byte peer_pubkey_slot, byte peer_cert_slot)This function connects the IoT-Safe TLS callbacks to the given SSL session.
int	**wolfSSL_iotsafe_on_ex except that the IDs for the IoT-SAFE slots can be passed by reference, and the length of the ID fields can be specified via the parameter “id_size”.
void	**wolfIoTSafe_SetCSIM_read_cb**(wolfSSL_IOTSafe_CSIM_read_cb rf)Associates a read callback for the AT+CSIM commands. This input function is usually associated to a read event of a UART channel communicating with the modem. The read callback associated is global and changes for all the contexts that use IoT-safe support at the same time.
void	**wolfIoTSafe_SetCSIM_write_cb**(wolfSSL_IOTSafe_CSIM_write_cb wf)Associates a write callback for the AT+CSIM commands. This output function is usually associated to a write event on a UART channel communicating with the modem. The write callback associated is global and changes for all the contexts that use IoT-safe support at the same time.
int	**wolfIoTSafe_GetRandom**(unsigned char * out, word32 sz)Generate a random buffer of given size, using the IoT-Safe function GetRandom. This function is automatically used by the wolfCrypt RNG object.

	Name
int	**wolfIoTSafe_GetCert**(uint8_t id, unsigned char * output, unsigned long sz)Import a certificate stored in a file on IoT-Safe applet, and store it locally in memory. Works with one-byte file ID field.
int	**wolfIoTSafe_GetCert_ex, except that it can be invoked with a file ID of two or more bytes.
int	**wc_iotsafe_ecc_import_public**(ecc_key * key, byte key_id)Import an ECC 256-bit public key, stored in the IoT-Safe applet, into an ecc_key object.
int	**wc_iotsafe_ecc_export_public**(ecc_key * key, byte key_id)Export an ECC 256-bit public key, from ecc_key object to a writable public-key slot into the IoT-Safe applet.
int	**wc_iotsafe_ecc_import_public_ex, except that it can be invoked with a key ID of two or more bytes.
int	**wc_iotsafe_ecc_export_private**(ecc_key * key, byte key_id)Export an ECC 256-bit key, from ecc_key object to a writable private-key slot into the IoT-Safe applet.
int	**wc_iotsafe_ecc_export_private_ex, except that it can be invoked with a key ID of two or more bytes.
int	**wc_iotsafe_ecc_sign_hash**(byte * in, word32 inlen, byte * out, word32 * outlen, byte key_id)Sign a pre-computed HASH, using a private key previously stored, or pre-provisioned, in the IoT-Safe applet.
int	**wc_iotsafe_ecc_sign_hash_ex, except that it can be invoked with a key ID of two or more bytes.
int	**wc_iotsafe_ecc_verify_hash**(byte * sig, word32 siglen, byte * hash, word32 hashlen, int * res, byte key_id)Verify an ECC signature against a pre-computed HASH, using a public key previously stored, or pre-provisioned, in the IoT-Safe applet. Result is written to res. 1 is valid, 0 is invalid. Note: Do not use the return value to test for valid. Only use res.
int	**wc_iotsafe_ecc_verify_hash_ex, except that it can be invoked with a key ID of two or more bytes.
int	**wc_iotsafe_ecc_gen_k**(byte key_id)Generate an ECC 256_bit keypair and store it in a (writable) slot into the IoT-Safe applet.

B.5.2 Detailed Description

IoT-Safe (IoT-SIM Applet For Secure End-2-End Communication) is a technology that leverage the SIM as robust, scalable and standardized hardware Root of Trust to protect data communication.

IoT-Safe SSL sessions use the SIM as Hardware Security Module, offloading all the crypto public key operations and reducing the attack surface by restricting access to certificate and keys to the SIM.

IoT-Safe support can be enabled on an existing WOLFSSL_CTX context, using wolfSSL_CTX_iotsafe_enable().

Session created within the context can set the parameters for IoT-Safe key and files usage, and enable the public keys callback, with wolfSSL_iotsafe_on().

If compiled in, the module supports IoT-Safe random number generator as source of entropy for wolfCrypt.

B.5.3 Functions Documentation

```
int wolfSSL_CTX_iotsafe_enable(
    WOLFSSL_CTX * ctx
)
```

This function enables the IoT-Safe support on the given context.

Parameters:

- **ctx** pointer to the WOLFSSL_CTX object on which the IoT-safe support must be enabled

See:

- wolfSSL_iotsafe_on
- wolfIoTSafe_SetCSIM_read_cb
- wolfIoTSafe_SetCSIM_write_cb

Return:

- 0 on success
- WC_HW_E on hardware error

Example

```
WOLFSSL_CTX *ctx;
ctx = wolfSSL_CTX_new(wolfTLSv1_2_client_method());
if (!ctx)
    return NULL;
wolfSSL_CTX_iotsafe_enable(ctx);
```

```
int wolfSSL_iotsafe_on(
    WOLFSSL * ssl,
    byte privkey_id,
    byte ecdh_keypair_slot,
    byte peer_pubkey_slot,
    byte peer_cert_slot
)
```

This function connects the IoT-Safe TLS callbacks to the given SSL session.

Parameters:

- **ssl** pointer to the WOLFSSL object where the callbacks will be enabled
- **privkey_id** id of the iot-safe applet slot containing the private key for the host
- **ecdh_keypair_slot** id of the iot-safe applet slot to store the ECDH keypair
- **peer_pubkey_slot** id of the iot-safe applet slot to store the other endpoint's public key for ECDH
- **peer_cert_slot** id of the iot-safe applet slot to store the other endpoint's public key for verification

See:

- wolfSSL_iotsafe_on_ex
- wolfSSL_CTX_iotsafe_enable

Return:

- 0 upon success
- NOT_COMPILED_IN if HAVE_PK_CALLBACKS is disabled
- BAD_FUNC_ARG if the ssl pointer is invalid

This should be called to connect a SSL session to IoT-Safe applet when the ID of the slots are one-byte long. If IoT-SAFE slots have an ID of two or more bytes, wolfSSL_iotsafe_on_ex() should be used instead.

Example

```
// Define key ids for IoT-Safe
#define PRIVKEY_ID 0x02
#define ECDH_KEYPAIR_ID 0x03
#define PEER_PUBKEY_ID 0x04
#define PEER_CERT_ID 0x05
// Create new ssl session
WOLFSSL *ssl;
ssl = wolfSSL_new(ctx);
if (!ssl)
    return NULL;
// Enable IoT-Safe and associate key slots
ret = wolfSSL_CTX_iotsafe_enable(ctx);
if (ret == 0) {
    ret = wolfSSL_iotsafe_on(ssl, PRIVKEY_ID, ECDH_KEYPAIR_ID, PEER_PUBKEY_ID,
 ↪  PEER_CERT_ID);
}
```

```
int wolfSSL_iotsafe_on_ex(
    WOLFSSL * ssl,
    byte * privkey_id,
    byte * ecdh_keypair_slot,
    byte * peer_pubkey_slot,
    byte * peer_cert_slot,
    word16 id_size
```

)

This function connects the IoT-Safe TLS callbacks to the given SSL session. This is equivalent to wolfSSL_iotsafe_on except that the IDs for the IoT-SAFE slots can be passed by reference, and the length of the ID fields can be specified via the parameter “id_size”.

Parameters:

- **ssl** pointer to the WOLFSSL object where the callbacks will be enabled
- **privkey_id** pointer to the id of the iot-safe applet slot containing the private key for the host
- **ecdh_keypair_slot** pointer to the id of the iot-safe applet slot to store the ECDH keypair
- **peer_pubkey_slot** pointer to the of id the iot-safe applet slot to store the other endpoint’s public key for ECDH
- **peer_cert_slot** pointer to the id of the iot-safe applet slot to store the other endpoint’s public key for verification
- **id_size** size of each slot ID

See:

- wolfSSL_iotsafe_on
- wolfSSL_CTX_iotsafe_enable

Return:

- 0 upon success
- NOT_COMPILED_IN if HAVE_PK_CALLBACKS is disabled
- BAD_FUNC_ARG if the ssl pointer is invalid

Example

```
// Define key ids for IoT-Safe (16 bit, little endian)
#define PRIVKEY_ID 0x0201
#define ECDH_KEYPAIR_ID 0x0301
#define PEER_PUBKEY_ID 0x0401
#define PEER_CERT_ID 0x0501
#define ID_SIZE (sizeof(word16))

word16 privkey = PRIVKEY_ID,
         ecdh_keypair = ECDH_KEYPAIR_ID,
         peer_pubkey = PEER_PUBKEY_ID,
         peer_cert = PEER_CERT_ID;



// Create new ssl session
WOLFSSL *ssl;
ssl = wolfSSL_new(ctx);
if (!ssl)
    return NULL;
// Enable IoT-Safe and associate key slots
ret = wolfSSL_CTX_iotsafe_enable(ctx);
if (ret == 0) {
```

```
    ret = wolfSSL_CTX_iotsafe_on_ex(ssl, &privkey, &ecdh_keypair, &peer_pubkey,
 ↪  &peer_cert, ID_SIZE);
}
```

```
void wolfIoTSafe_SetCSIM_read_cb(
    wolfSSL_IOTSafe_CSIM_read_cb rf
)
```

Associates a read callback for the AT+CSIM commands. This input function is usually associated to a read event of a UART channel communicating with the modem. The read callback associated is global and changes for all the contexts that use IoT-safe support at the same time.

Parameters:

- **rf** Read callback associated to a UART read event. The callback function takes two arguments (buf, len) and return the number of characters read, up to len. When a newline is encountered, the callback should return the number of characters received so far, including the newline character.

See: wolfIoTSafe_SetCSIM_write_cb

Example

```
// USART read function, defined elsewhere
int usart_read(char *buf, int len);

wolfIoTSafe_SetCSIM_read_cb(usart_read);
```

```
void wolfIoTSafe_SetCSIM_write_cb(
    wolfSSL_IOTSafe_CSIM_write_cb wf
)
```

Associates a write callback for the AT+CSIM commands. This output function is usually associated to a write event on a UART channel communicating with the modem. The write callback associated is global and changes for all the contexts that use IoT-safe support at the same time.

Parameters:

- **rf** Write callback associated to a UART write event. The callback function takes two arguments (buf, len) and return the number of characters written, up to len.

See: wolfIoTSafe_SetCSIM_read_cb

Example

```
// USART write function, defined elsewhere
int usart_write(const char *buf, int len);
wolfIoTSafe_SetCSIM_write_cb(usart_write);
```

```
int wolfIoTSafe_GetRandom(
    unsigned char * out,
    word32 sz
)
```

Generate a random buffer of given size, using the IoT-Safe function GetRandom. This function is automatically used by the wolfCrypt RNG object.

Parameters:

- **out** the buffer where the random sequence of bytes is stored.
- **sz** the size of the random sequence to generate, in bytes

Return: 0 upon success

```
int wolfIoTSafe_GetCert(
    uint8_t id,
    unsigned char * output,
    unsigned long sz
)
```

Import a certificate stored in a file on IoT-Safe applet, and store it locally in memory. Works with one-byte file ID field.

Parameters:

- **id** The file id in the IoT-Safe applet where the certificate is stored
- **output** the buffer where the certificate will be imported
- **sz** the maximum size available in the buffer output

Return:

- the length of the certificate imported
- < 0 in case of failure

Example

```
#define CRT_CLIENT_FILE_ID 0x03
unsigned char cert_buffer[2048];
// Get the certificate into the buffer
cert_buffer_size = wolfIoTSafe_GetCert(CRT_CLIENT_FILE_ID, cert_buffer, 2048);
if (cert_buffer_size < 1) {
    printf("Bad cli cert\n");
    return -1;
}
printf("Loaded Client certificate from IoT-Safe, size = %lu\n",
 ↪  cert_buffer_size);

// Use the certificate buffer as identity for the TLS client context
if (wolfSSL_CTX_use_certificate_buffer(cli_ctx, cert_buffer,
            cert_buffer_size, SSL_FILETYPE_ASN1) != SSL_SUCCESS) {
```

```
    printf("Cannot load client cert\n");
    return -1;
}
printf("Client certificate successfully imported.\n");
```

```
int wolfIoTSafe_GetCert_ex(
    uint8_t * id,
    uint16_t id_sz,
    unsigned char * output,
    unsigned long sz
)
```

Import a certificate stored in a file on IoT-Safe applet, and store it locally in memory. Equivalent to wolfIoTSafe_GetCert, except that it can be invoked with a file ID of two or more bytes.

Parameters:

- **id** Pointer to the file id in the IoT-Safe applet where the certificate is stored
- **id_sz** Size of the file id in bytes
- **output** the buffer where the certificate will be imported
- **sz** the maximum size available in the buffer output

Return:

- the length of the certificate imported
- < 0 in case of failure

Example

```
#define CRT_CLIENT_FILE_ID 0x0302
#define ID_SIZE (sizeof(word16))
unsigned char cert_buffer[2048];
word16 client_file_id = CRT_CLIENT_FILE_ID;
```

```
// Get the certificate into the buffer
cert_buffer_size = wolfIoTSafe_GetCert_ex(&client_file_id, ID_SIZE,
 ↪  cert_buffer, 2048);
if (cert_buffer_size < 1) {
    printf("Bad cli cert\n");
    return -1;
}
printf("Loaded Client certificate from IoT-Safe, size = %lu\n",
 ↪  cert_buffer_size);

// Use the certificate buffer as identity for the TLS client context
if (wolfSSL_CTX_use_certificate_buffer(cli_ctx, cert_buffer,
            cert_buffer_size, SSL_FILETYPE_ASN1) != SSL_SUCCESS) {
    printf("Cannot load client cert\n");
    return -1;
```

```
}
printf("Client certificate successfully imported.\n");
```

```
int wc_iotsafe_ecc_import_public(
    ecc_key * key,
    byte key_id
)
```

Import an ECC 256-bit public key, stored in the IoT-Safe applet, into an ecc_key object.

Parameters:

- **key** the ecc_key object that will contain the key imported from the IoT-Safe applet
- **id** The key id in the IoT-Safe applet where the public key is stored

See:

- wc_iotsafe_ecc_export_public
- wc_iotsafe_ecc_export_private

Return:

- 0 upon success
- < 0 in case of failure

```
int wc_iotsafe_ecc_export_public(
    ecc_key * key,
    byte key_id
)
```

Export an ECC 256-bit public key, from ecc_key object to a writable public-key slot into the IoT-Safe applet.

Parameters:

- **key** the ecc_key object containing the key to be exported
- **id** The key id in the IoT-Safe applet where the public key will be stored

See:

- wc_iotsafe_ecc_import_public_ex
- wc_iotsafe_ecc_export_private

Return:

- 0 upon success
- < 0 in case of failure

```
int wc_iotsafe_ecc_import_public_ex(
    ecc_key * key,
    byte * key_id,
    word16 id_size
)
```

Export an ECC 256-bit public key, from ecc_key object to a writable public-key slot into the IoT-Safe applet. Equivalent to wc_iotsafe_ecc_import_public, except that it can be invoked with a key ID of two or more bytes.

Parameters:

- **key** the ecc_key object containing the key to be exported
- **id** The pointer to the key id in the IoT-Safe applet where the public key will be stored
- **id_size** The key id size

See:

- wc_iotsafe_ecc_import_public
- wc_iotsafe_ecc_export_private

Return:

- 0 upon success
- < 0 in case of failure

```
int wc_iotsafe_ecc_export_private(
    ecc_key * key,
    byte key_id
)
```

Export an ECC 256-bit key, from ecc_key object to a writable private-key slot into the IoT-Safe applet.

Parameters:

- **key** the ecc_key object containing the key to be exported
- **id** The key id in the IoT-Safe applet where the private key will be stored

See:

- wc_iotsafe_ecc_export_private_ex
- wc_iotsafe_ecc_import_public
- wc_iotsafe_ecc_export_public

Return:

- 0 upon success
- < 0 in case of failure

```
int wc_iotsafe_ecc_export_private_ex(
    ecc_key * key,
    byte * key_id,
    word16 id_size
)
```

Export an ECC 256-bit key, from ecc_key object to a writable private-key slot into the IoT-Safe applet. Equivalent to wc_iotsafe_ecc_export_private, except that it can be invoked with a key ID of two or more bytes.

Parameters:

- **key** the ecc_key object containing the key to be exported
- **id** The pointer to the key id in the IoT-Safe applet where the private key will be stored
- **id_size** The key id size

See:

- wc_iotsafe_ecc_export_private
- wc_iotsafe_ecc_import_public
- wc_iotsafe_ecc_export_public

Return:

- 0 upon success
- < 0 in case of failure

```
int wc_iotsafe_ecc_sign_hash(
    byte * in,
    word32 inlen,
    byte * out,
    word32 * outlen,
    byte key_id
)
```

Sign a pre-computed HASH, using a private key previously stored, or pre-provisioned, in the IoT-Safe applet.

Parameters:

- **in** pointer to the buffer containing the message hash to sign
- **inlen** length of the message hash to sign
- **out** buffer in which to store the generated signature
- **outlen** max length of the output buffer. Will store the bytes
- **id** key id in the IoT-Safe applet for the slot containing the private key to sign the payload written to out upon successfully generating a message signature

See:

- wc_iotsafe_ecc_sign_hash_ex

- wc_iotsafe_ecc_verify_hash
- wc_iotsafe_ecc_gen_k

Return:

- 0 upon success
- < 0 in case of failure

```
int wc_iotsafe_ecc_sign_hash_ex(
    byte * in,
    word32 inlen,
    byte * out,
    word32 * outlen,
    byte * key_id,
    word16 id_size
)
```

Sign a pre-computed HASH, using a private key previously stored, or pre-provisioned, in the IoT-Safe applet. Equivalent to wc_iotsafe_ecc_sign_hash, except that it can be invoked with a key ID of two or more bytes.

Parameters:

- **in** pointer to the buffer containing the message hash to sign
- **inlen** length of the message hash to sign
- **out** buffer in which to store the generated signature
- **outlen** max length of the output buffer. Will store the bytes
- **id** pointer to a key id in the IoT-Safe applet for the slot containing the private key to sign the payload written to out upon successfully generating a message signature
- **id_size** The key id size

See:

- wc_iotsafe_ecc_sign_hash
- wc_iotsafe_ecc_verify_hash
- wc_iotsafe_ecc_gen_k

Return:

- 0 upon success
- < 0 in case of failure

```
int wc_iotsafe_ecc_verify_hash(
    byte * sig,
    word32 siglen,
    byte * hash,
    word32 hashlen,
    int * res,
```

```
    byte key_id
)
```

Verify an ECC signature against a pre-computed HASH, using a public key previously stored, or pre-provisioned, in the IoT-Safe applet. Result is written to res. 1 is valid, 0 is invalid. Note: Do not use the return value to test for valid. Only use res.

Parameters:

- **sig** buffer containing the signature to verify
- **hash** The hash (message digest) that was signed
- **hashlen** The length of the hash (octets)
- **res** Result of signature, 1==valid, 0==invalid
- **key_id** The id of the slot where the public ECC key is stored in the IoT-Safe applet

See:

- wc_iotsafe_ecc_verify_hash_ex
- wc_iotsafe_ecc_sign_hash
- wc_iotsafe_ecc_gen_k

Return:

- 0 upon success (even if the signature is not valid)
- < 0 in case of failure.

```
int wc_iotsafe_ecc_verify_hash_ex(
    byte * sig,
    word32 siglen,
    byte * hash,
    word32 hashlen,
    int * res,
    byte * key_id,
    word16 id_size
)
```

Verify an ECC signature against a pre-computed HASH, using a public key previously stored, or pre-provisioned, in the IoT-Safe applet. Result is written to res. 1 is valid, 0 is invalid. Note: Do not use the return value to test for valid. Only use res. Equivalent to wc_iotsafe_ecc_verify_hash, except that it can be invoked with a key ID of two or more bytes.

Parameters:

- **sig** buffer containing the signature to verify
- **hash** The hash (message digest) that was signed
- **hashlen** The length of the hash (octets)
- **res** Result of signature, 1==valid, 0==invalid
- **key_id** The id of the slot where the public ECC key is stored in the IoT-Safe applet
- **id_size** The key id size

See:

- wc_iotsafe_ecc_verify_hash
- wc_iotsafe_ecc_sign_hash
- wc_iotsafe_ecc_gen_k

Return:

- 0 upon success (even if the signature is not valid)
- < 0 in case of failure.

```
int wc_iotsafe_ecc_gen_k(
    byte key_id
)
```

Generate an ECC 256-bit keypair and store it in a (writable) slot into the IoT-Safe applet.

Parameters:

- **key_id** The id of the slot where the ECC key pair is stored in the IoT-Safe applet.
- **key_id** The id of the slot where the ECC key pair is stored in the IoT-Safe applet.
- **id_size** The key id size

See:

- wc_iotsafe_ecc_gen_k_ex
- wc_iotsafe_ecc_sign_hash
- wc_iotsafe_ecc_verify_hash
- wc_iotsafe_ecc_gen_k
- wc_iotsafe_ecc_sign_hash_ex
- wc_iotsafe_ecc_verify_hash_ex

Return:

- 0 upon success
- < 0 in case of failure.
- 0 upon success
- < 0 in case of failure.

Generate an ECC 256-bit keypair and store it in a (writable) slot into the IoT-Safe applet. Equivalent to wc_iotsafe_ecc_gen_k, except that it can be invoked with a key ID of two or more bytes.

B.6 Key and Cert Conversion

B.5.3.18 function wc_iotsafe_ecc_gen_k

B.7 Logging

B.7.1 Functions

	Name
int	**wolfSSL_SetLoggingCb**(wolfSSL_Logging_cb log_function)This function registers a logging callback that will be used to handle the wolfSSL log message. By default, if the system supports it fprintf() to stderr is used but by using this function anything can be done by the user.

B.7.2 Functions Documentation

```
int wolfSSL_SetLoggingCb(
    wolfSSL_Logging_cb log_function
)
```

This function registers a logging callback that will be used to handle the wolfSSL log message. By default, if the system supports it fprintf() to stderr is used but by using this function anything can be done by the user.

Parameters:

- **log_function** function to register as a logging callback. Function signature must follow the above prototype.

See:

- wolfSSL_Debugging_ON
- wolfSSL_Debugging_OFF

Return:

- Success If successful this function will return 0.
- BAD_FUNC_ARG is the error that will be returned if a function pointer is not provided.

Example

```
int ret = 0;
// Logging callback prototype
void MyLoggingCallback(const int logLevel, const char* const logMessage);
// Register the custom logging callback with wolfSSL
ret = wolfSSL_SetLoggingCb(MyLoggingCallback);
if (ret != 0) {
    // failed to set logging callback
}
void MyLoggingCallback(const int logLevel, const char* const logMessage)
{
// custom logging function
}
```

B.8 Math API

B.7.2.1 function wolfSSL_SetLoggingCb

B.8.1 Functions

	Name
word32	**CheckRunTimeFastMath**(void)This function checks the runtime fastmath settings for the maximum size of an integer. It is important when a user is using a wolfCrypt library independently, as the FP_SIZE must match for each library in order for math to work correctly. This check is defined as CheckFastMathSettings(), which simply compares CheckRunTimeFastMath and FP_SIZE, returning 0 if there is a mismatch, or 1 if they match.
word32	**CheckRunTimeSettings**(void)This function checks the compile time class settings. It is important when a user is using a wolfCrypt library independently, as the settings must match between libraries for math to work correctly. This check is defined as CheckCtcSettings(), which simply compares CheckRunTimeSettings and CTC_SETTINGS, returning 0 if there is a mismatch, or 1 if they match.

B.8.2 Functions Documentation

```
word32 CheckRunTimeFastMath(
    void
)
```

This function checks the runtime fastmath settings for the maximum size of an integer. It is important when a user is using a wolfCrypt library independently, as the FP_SIZE must match for each library in order for math to work correctly. This check is defined as CheckFastMathSettings(), which simply compares CheckRunTimeFastMath and FP_SIZE, returning 0 if there is a mismatch, or 1 if they match.

Parameters:

- **none** No parameters.

See: CheckRunTimeSettings

Return: FP_SIZE Returns FP_SIZE, corresponding to the max size available for the math library.

Example

```
if (CheckFastMathSettings() != 1) {
return err_sys("Build vs. runtime fastmath FP_MAX_BITS mismatch\n");
}
// This is converted by the preprocessor to:
// if ( (CheckRunTimeFastMath() == FP_SIZE) != 1) {
// and confirms that the fast math settings match
// the compile time settings
```

```
word32 CheckRunTimeSettings(
    void
)
```

This function checks the compile time class settings. It is important when a user is using a wolfCrypt library independently, as the settings must match between libraries for math to work correctly. This check is defined as CheckCtcSettings(), which simply compares CheckRunTimeSettings and CTC_SETTINGS, returning 0 if there is a mismatch, or 1 if they match.

Parameters:

- **none** No Parameters.

See: CheckRunTimeFastMath

Return: settings Returns the runtime CTC_SETTINGS (Compile Time Settings)

Example

```
if (CheckCtcSettings() != 1) {
    return err_sys("Build vs. runtime math mismatch\n");
}
// This is converted by the preprocessor to:
// if ( (CheckCtcSettings() == CTC_SETTINGS) != 1) {
// and will compare whether the compile time class settings
// match the current settings
```

B.9 Random Number Generation

B.8.2.2 function CheckRunTimeSettings

B.9.1 Functions

	Name
int	**wc_InitNetRandom**(const char * configFile, wnr_hmac_key hmac_cb, int timeout)Init global Whitewood netRandom context.
int	**wc_FreeNetRandom**(void)Free global Whitewood netRandom context.

	Name
int	**wc_InitRng**(WC_RNG *)Gets the seed (from OS) and key cipher for rng. rng_>drbg (deterministic random bit generator) allocated (should be deallocated with wc_FreeRng). This is a blocking operation.
int	**wc_RNG_GenerateBlock**(WC_RNG * rng, byte * b, word32 sz)Copies a sz bytes of pseudorandom data to output. Will reseed rng if needed (blocking).
WC_RNG *	**wc_rng_new**(byte * nonce, word32 nonceSz, void * heap)Creates a new WC_RNG structure.
int	**wc_FreeRng**(WC_RNG *)Should be called when RNG no longer needed in order to securely free drgb. Zeros and XFREEs rng-drbg.
WC_RNG *	**wc_rng_free**(WC_RNG * rng)Should be called when RNG no longer needed in order to securely free rng.
int	**wc_RNG_HealthTest**(int reseed, const byte * entropyA, word32 entropyASz, const byte * entropyB, word32 entropyBSz, byte * output, word32 outputSz)Creates and tests functionality of drbg.

B.9.2 Functions Documentation

```
int wc_InitNetRandom(
    const char * configFile,
    wnr_hmac_key hmac_cb,
    int timeout
)
```

Init global Whitewood netRandom context.

Parameters:

- **configFile** Path to configuration file
- **hmac_cb** Optional to create HMAC callback.
- **timeout** A timeout duration.

See: wc_FreeNetRandom

Return:

- 0 Success
- BAD_FUNC_ARG Either configFile is null or timeout is negative.
- RNG_FAILURE_E There was a failure initializing the rng.

Example

```
char* config = "path/to/config/example.conf";
int time = // Some sufficient timeout value;
```

```
if (wc_InitNetRandom(config, NULL, time) != 0)
{
    // Some error occurred
}
```

```
int wc_FreeNetRandom(
    void
)
```

Free global Whitewood netRandom context.

Parameters:

- **none** No returns.

See: wc_InitNetRandom

Return:

- 0 Success
- BAD_MUTEX_E Error locking mutex on wnr_mutex

Example

```
int ret = wc_FreeNetRandom();
if(ret != 0)
{
    // Handle the error
}
```

```
int wc_InitRng(
    WC_RNG *
)
```

Gets the seed (from OS) and key cipher for rng. rng->drbg (deterministic random bit generator) allocated (should be deallocated with wc_FreeRng). This is a blocking operation.

Parameters:

- **rng** random number generator to be initialized for use with a seed and key cipher

See:

- wc_InitRngCavium
- wc_RNG_GenerateBlock
- wc_RNG_GenerateByte
- wc_FreeRng

- wc_RNG_HealthTest

Return:

- 0 on success.
- MEMORY_E XMALLOC failed
- WINCRYPT_E wc_GenerateSeed: failed to acquire context
- CRYPTGEN_E wc_GenerateSeed: failed to get random
- BAD_FUNC_ARG wc_RNG_GenerateBlock input is null or sz exceeds MAX_REQUEST_LEN
- DRBG_CONT_FIPS_E wc_RNG_GenerateBlock: Hash_gen returned DRBG_CONT_FAILURE
- RNG_FAILURE_E wc_RNG_GenerateBlock: Default error. rng's status originally not ok, or set to DRBG_FAILED

Example

```
RNG  rng;
int ret;

#ifdef HAVE_CAVIUM
ret = wc_InitRngCavium(&rng, CAVIUM_DEV_ID);
if (ret != 0){
    printf("RNG Nitrox init for device: %d failed", CAVIUM_DEV_ID);
    return -1;
}
#endif
ret = wc_InitRng(&rng);
if (ret != 0){
    printf("RNG init failed");
    return -1;
}
```

```
int wc_RNG_GenerateBlock(
    WC_RNG * rng,
    byte * b,
    word32 sz
)
```

Copies a sz bytes of pseudorandom data to output. Will reseed rng if needed (blocking).

Parameters:

- **rng** random number generator initialized with wc_InitRng
- **output** buffer to which the block is copied
- **sz** size of output in bytes

See:

- wc_InitRngCavium, wc_InitRng
- wc_RNG_GenerateByte
- wc_FreeRng

- wc_RNG_HealthTest

Return:

- 0 on success
- BAD_FUNC_ARG an input is null or sz exceeds MAX_REQUEST_LEN
- DRBG_CONT_FIPS_E Hash_gen returned DRBG_CONT_FAILURE
- RNG_FAILURE_E Default error. rng's status originally not ok, or set to DRBG_FAILED

Example

```
RNG  rng;
int  sz = 32;
byte block[sz];

int ret = wc_InitRng(&rng);
if (ret != 0) {
    return -1; //init of rng failed!
}

ret = wc_RNG_GenerateBlock(&rng, block, sz);
if (ret != 0) {
    return -1; //generating block failed!
}
```

```
WC_RNG * wc_rng_new(
    byte * nonce,
    word32 nonceSz,
    void * heap
)
```

Creates a new WC_RNG structure.

Parameters:

- **heap** pointer to a heap identifier
- **nonce** pointer to the buffer containing the nonce
- **nonceSz** length of the nonce
- **rng** random number generator initialized with wc_InitRng
- **b** one byte buffer to which the block is copied

See:

- wc_InitRng
- wc_rng_free
- wc_FreeRng
- wc_RNG_HealthTest
- wc_InitRngCavium
- wc_InitRng
- wc_RNG_GenerateBlock

- wc_FreeRng
- wc_RNG_HealthTest

Return:

- WC_RNG structure on success
- NULL on error
- 0 on success
- BAD_FUNC_ARG an input is null or sz exceeds MAX_REQUEST_LEN
- DRBG_CONT_FIPS_E Hash_gen returned DRBG_CONT_FAILURE
- RNG_FAILURE_E Default error. rng's status originally not ok, or set to DRBG_FAILED

Example

```
RNG  rng;
byte nonce[] = { initialize nonce };
word32 nonceSz = sizeof(nonce);

wc_rng_new(&nonce, nonceSz, &heap);
```

Calls wc_RNG_GenerateBlock to copy a byte of pseudorandom data to b. Will reseed rng if needed.

Example

```
RNG  rng;
int  sz = 32;
byte b[1];

int ret = wc_InitRng(&rng);
if (ret != 0) {
    return -1; //init of rng failed!
}

ret = wc_RNG_GenerateByte(&rng, b);
if (ret != 0) {
    return -1; //generating block failed!
}
```

```
int wc_FreeRng(
    WC_RNG *
)
```

Should be called when RNG no longer needed in order to securely free drgb. Zeros and XFREEs rng-drbg.

Parameters:

- **rng** random number generator initialized with wc_InitRng

See:

- wc_InitRngCavium
- wc_InitRng
- wc_RNG_GenerateBlock
- wc_RNG_GenerateByte,
- wc_RNG_HealthTest

Return:

- 0 on success
- BAD_FUNC_ARG rng or rng->drgb null
- RNG_FAILURE_E Failed to deallocated drbg

Example

```
RNG  rng;
int ret = wc_InitRng(&rng);
if (ret != 0) {
    return -1; //init of rng failed!
}

int ret = wc_FreeRng(&rng);
if (ret != 0) {
    return -1; //free of rng failed!
}
```

```
WC_RNG * wc_rng_free(
    WC_RNG * rng
)
```

Should be called when RNG no longer needed in order to securely free rng.

Parameters:

- **rng** random number generator initialized with wc_InitRng

See:

- wc_InitRng
- wc_rng_new
- wc_FreeRng
- wc_RNG_HealthTest

Example

```
RNG  rng;
byte nonce[] = { initialize nonce };
word32 nonceSz = sizeof(nonce);

rng = wc_rng_new(&nonce, nonceSz, &heap);
```

```
// use rng

wc_rng_free(&rng);
```

```
int wc_RNG_HealthTest(
    int reseed,
    const byte * entropyA,
    word32 entropyASz,
    const byte * entropyB,
    word32 entropyBSz,
    byte * output,
    word32 outputSz
)
```

Creates and tests functionality of drbg.

Parameters:

- **int** reseed: if set, will test reseed functionality
- **entropyA** entropy to instantiate drgb with
- **entropyASz** size of entropyA in bytes
- **entropyB** If reseed set, drbg will be reseeded with entropyB
- **entropyBSz** size of entropyB in bytes
- **output** initialized to random data seeded with entropyB if seedrandom is set, and entropyA otherwise
- **outputSz** length of output in bytes

See:

- wc_InitRngCavium
- wc_InitRng
- wc_RNG_GenerateBlock
- wc_RNG_GenerateByte
- wc_FreeRng

Return:

- 0 on success
- BAD_FUNC_ARG entropyA and output must not be null. If reseed set entropyB must not be null
- -1 test failed

Example

```
byte output[SHA256_DIGEST_SIZE * 4];
const byte test1EntropyB[] = ....; // test input for reseed false
const byte test1Output[] = ....;   // testvector: expected output of
                                   // reseed false
ret = wc_RNG_HealthTest(0, test1Entropy, sizeof(test1Entropy), NULL, 0,
                    output, sizeof(output));
if (ret != 0)
```

```
    return -1;//healthtest without reseed failed

if (XMEMCMP(test1Output, output, sizeof(output)) != 0)
    return -1; //compare to testvector failed: unexpected output

const byte test2EntropyB[] = ....; // test input for reseed
const byte test2Output[] = ....;   // testvector expected output of reseed
ret = wc_RNG_HealthTest(1, test2EntropyA, sizeof(test2EntropyA),
                    test2EntropyB, sizeof(test2EntropyB),
                    output, sizeof(output));

if (XMEMCMP(test2Output, output, sizeof(output)) != 0)
    return -1; //compare to testvector failed
```

B.10 Signature API

B.9.2.8 function wc_RNG_HealthTest

B.10.1 Functions

	Name
int	**wc_SignatureGetSize**(enum wc_SignatureType sig_type, const void * key, word32 key_len)This function returns the maximum size of the resulting signature.
int	**wc_SignatureVerify**(enum wc_HashType hash_type, enum wc_SignatureType sig_type, const byte * data, word32 data_len, const byte * sig, word32 sig_len, const void * key, word32 key_len)This function validates a signature by hashing the data and using the resulting hash and key to verify the signature.
int	**wc_SignatureGenerate**(enum wc_HashType hash_type, enum wc_SignatureType sig_type, const byte * data, word32 data_len, byte * sig, word32 * sig_len, const void * key, word32 key_len, WC_RNG * rng)This function generates a signature from the data using a key. It first creates a hash of the data then signs the hash using the key.

B.10.2 Functions Documentation

```
int wc_SignatureGetSize(
    enum wc_SignatureType sig_type,
    const void * key,
    word32 key_len
)
```

This function returns the maximum size of the resulting signature.

Parameters:

- **sig_type** A signature type enum value such as WC_SIGNATURE_TYPE_ECC or WC_SIGNATURE_TYPE_RSA.
- **key** Pointer to a key structure such as ecc_key or RsaKey.
- **key_len** Size of the key structure.

See:

- wc_HashGetDigestSize
- wc_SignatureGenerate
- wc_SignatureVerify

Return: Returns SIG_TYPE_E if sig_type is not supported. Returns BAD_FUNC_ARG if sig_type was invalid. A positive return value indicates the maximum size of a signature.

Example

```
// Get signature length
enum wc_SignatureType sig_type = WC_SIGNATURE_TYPE_ECC;
ecc_key eccKey;
word32 sigLen;
wc_ecc_init(&eccKey);
sigLen = wc_SignatureGetSize(sig_type, &eccKey, sizeof(eccKey));
if (sigLen > 0) {
    // Success
}
```

```
int wc_SignatureVerify(
    enum wc_HashType hash_type,
    enum wc_SignatureType sig_type,
    const byte * data,
    word32 data_len,
    const byte * sig,
    word32 sig_len,
    const void * key,
    word32 key_len
)
```

This function validates a signature by hashing the data and using the resulting hash and key to verify the signature.

Parameters:

- **hash_type** A hash type from the “enum wc_HashType” such as “WC_HASH_TYPE_SHA256”.
- **sig_type** A signature type enum value such as WC_SIGNATURE_TYPE_ECC or WC_SIGNATURE_TYPE_RSA.
- **data** Pointer to buffer containing the data to hash.
- **data_len** Length of the data buffer.
- **sig** Pointer to buffer to output signature.
- **sig_len** Length of the signature output buffer.
- **key** Pointer to a key structure such as ecc_key or RsaKey.

- **key_len** Size of the key structure.

See:

- wc_SignatureGetSize
- wc_SignatureGenerate

Return:

- 0 Success
- SIG_TYPE_E -231, signature type not enabled/ available
- BAD_FUNC_ARG -173, bad function argument provided
- BUFFER_E -132, output buffer too small or input too large.

Example

```
int ret;
ecc_key eccKey;

// Import the public key
wc_ecc_init(&eccKey);
ret = wc_ecc_import_x963(eccPubKeyBuf, eccPubKeyLen, &eccKey);
// Perform signature verification using public key
ret = wc_SignatureVerify(
WC_HASH_TYPE_SHA256, WC_SIGNATURE_TYPE_ECC,
fileBuf, fileLen,
sigBuf, sigLen,
&eccKey, sizeof(eccKey));
printf("Signature Verification: %s
(%d)\n", (ret == 0) ? "Pass" : "Fail", ret);
wc_ecc_free(&eccKey);
```

```
int wc_SignatureGenerate(
    enum wc_HashType hash_type,
    enum wc_SignatureType sig_type,
    const byte * data,
    word32 data_len,
    byte * sig,
    word32 * sig_len,
    const void * key,
    word32 key_len,
    WC_RNG * rng
)
```

This function generates a signature from the data using a key. It first creates a hash of the data then signs the hash using the key.

Parameters:

- **hash_type** A hash type from the “enum wc_HashType” such as “WC_HASH_TYPE_SHA256”.

- **sig_type** A signature type enum value such as WC_SIGNATURE_TYPE_ECC or WC_SIGNATURE_TYPE_RSA.
- **data** Pointer to buffer containing the data to hash.
- **data_len** Length of the data buffer.
- **sig** Pointer to buffer to output signature.
- **sig_len** Length of the signature output buffer.
- **key** Pointer to a key structure such as ecc_key or RsaKey.
- **key_len** Size of the key structure.
- **rng** Pointer to an initialized RNG structure.

See:

- wc_SignatureGetSize
- wc_SignatureVerify

Return:

- 0 Success
- SIG_TYPE_E -231, signature type not enabled/ available
- BAD_FUNC_ARG -173, bad function argument provided
- BUFFER_E -132, output buffer too small or input too large.

Example

```
int ret;
WC_RNG rng;
ecc_key eccKey;

wc_InitRng(&rng);
wc_ecc_init(&eccKey);

// Generate key
ret = wc_ecc_make_key(&rng, 32, &eccKey);

// Get signature length and allocate buffer
sigLen = wc_SignatureGetSize(sig_type, &eccKey, sizeof(eccKey));
sigBuf = malloc(sigLen);

// Perform signature verification using public key
ret = wc_SignatureGenerate(
    WC_HASH_TYPE_SHA256, WC_SIGNATURE_TYPE_ECC,
    fileBuf, fileLen,
    sigBuf, &sigLen,
    &eccKey, sizeof(eccKey),
    &rng);
printf("Signature Generation: %s
(%d)\n", (ret == 0) ? "Pass" : "Fail", ret);

free(sigBuf);
wc_ecc_free(&eccKey);
wc_FreeRng(&rng);
```

B.11 wolfCrypt Init and Cleanup

B.10.2.3 function wc_SignatureGenerate

B.11.1 Functions

	Name
int	**wc_HashGetOID**(enum wc_HashType hash_type)This function will return the OID for the wc_HashType provided.
int	**wc_HashGetDigestSize**(enum wc_HashType hash_type)This function returns the size of the digest (output) for a hash_type. The returns size is used to make sure the output buffer provided to wc_Hash is large enough.
int	**wc_Hash**(enum wc_HashType hash_type, const byte * data, word32 data_len, byte * hash, word32 hash_len)This function performs a hash on the provided data buffer and returns it in the hash buffer provided.
int	**wolfCrypt_Init**(void)Used to initialize resources used by wolfCrypt.
int	**wolfCrypt_Cleanup**(void)Used to clean up resources used by wolfCrypt.

B.11.2 Functions Documentation

```
int wc_HashGetOID(
    enum wc_HashType hash_type
)
```

This function will return the OID for the wc_HashType provided.

Parameters:

- **hash_type** A hash type from the “enum wc_HashType” such as “WC_HASH_TYPE_SHA256”.

See:

- wc_HashGetDigestSize
- wc_Hash

Return:

- OID returns value greater than 0
- HASH_TYPE_E hash type not supported.
- BAD_FUNC_ARG one of the provided arguments is incorrect.

Example

```
enum wc_HashType hash_type = WC_HASH_TYPE_SHA256;
int oid = wc_HashGetOID(hash_type);
if (oid > 0) {
    // Success
}
```

```
int wc_HashGetDigestSize(
    enum wc_HashType hash_type
)
```

This function returns the size of the digest (output) for a hash_type. The returns size is used to make sure the output buffer provided to wc_Hash is large enough.

Parameters:

- **hash_type** A hash type from the “enum wc_HashType” such as “WC_HASH_TYPE_SHA256”.

See: wc_Hash

Return:

- Success A positive return value indicates the digest size for the hash.
- Error Returns HASH_TYPE_E if hash_type is not supported.
- Failure Returns BAD_FUNC_ARG if an invalid hash_type was used.

Example

```
int hash_len = wc_HashGetDigestSize(hash_type);
if (hash_len <= 0) {
WOLFSSL_MSG("Invalid hash type/len");
return BAD_FUNC_ARG;
}
```

```
int wc_Hash(
    enum wc_HashType hash_type,
    const byte * data,
    word32 data_len,
    byte * hash,
    word32 hash_len
)
```

This function performs a hash on the provided data buffer and returns it in the hash buffer provided.

Parameters:

- **hash_type** A hash type from the “enum wc_HashType” such as “WC_HASH_TYPE_SHA256”.
- **data** Pointer to buffer containing the data to hash.
- **data_len** Length of the data buffer.
- **hash** Pointer to buffer used to output the final hash to.

- **hash_len** Length of the hash buffer.

See: wc_HashGetDigestSize

Return: 0 Success, else error (such as BAD_FUNC_ARG or BUFFER_E).

Example

```
enum wc_HashType hash_type = WC_HASH_TYPE_SHA256;
int hash_len = wc_HashGetDigestSize(hash_type);
if (hash_len > 0) {
    int ret = wc_Hash(hash_type, data, data_len, hash_data, hash_len);
    if(ret == 0) {
        // Success
    }
}
```

```
int wolfCrypt_Init(
    void
)
```

Used to initialize resources used by wolfCrypt.

Parameters:

- **none** No parameters.

See: wolfCrypt_Cleanup

Return:

- 0 upon success.
- <0 upon failure of init resources.

Example

```
...
if (wolfCrypt_Init() != 0) {
    WOLFSSL_MSG("Error with wolfCrypt_Init call");
}
```

```
int wolfCrypt_Cleanup(
    void
)
```

Used to clean up resources used by wolfCrypt.

Parameters:

- **none** No parameters.

See: wolfCrypt_Init

Return:

- 0 upon success.
- <0 upon failure of cleaning up resources.

Example

```
...
if (wolfCrypt_Cleanup() != 0) {
    WOLFSSL_MSG("Error with wolfCrypt_Cleanup call");
}
```

B.12 Algorithms - 3DES

B.11.2.5 function wolfCrypt_Cleanup

B.12.1 Functions

	Name
int	**wc_Des_SetKey**(Des * des, const byte * key, const byte * iv, int dir)This function sets the key and initialization vector (iv) for the Des structure given as argument. It also initializes and allocates space for the buffers needed for encryption and decryption, if these have not yet been initialized. Note: If no iv is provided (i.e. iv == NULL) the initialization vector defaults to an iv of 0.
void	**wc_Des_SetIV**(Des * des, const byte * iv)This function sets the initialization vector (iv) for the Des structure given as argument. When passed a NULL iv, it sets the initialization vector to 0.
int	**wc_Des_CbcEncrypt**(Des * des, byte * out, const byte * in, word32 sz)This function encrypts the input message, in, and stores the result in the output buffer, out. It uses DES encryption with cipher block chaining (CBC) mode.
int	**wc_Des_CbcDecrypt**(Des * des, byte * out, const byte * in, word32 sz)This function decrypts the input ciphertext, in, and stores the resulting plaintext in the output buffer, out. It uses DES encryption with cipher block chaining (CBC) mode.

	Name
int	**wc_Des_EcbEncrypt**(Des * des, byte * out, const byte * in, word32 sz)This function encrypts the input message, in, and stores the result in the output buffer, out. It uses Des encryption with Electronic Codebook (ECB) mode.
int	**wc_Des3_EcbEncrypt**(Des3 * des, byte * out, const byte * in, word32 sz)This function encrypts the input message, in, and stores the result in the output buffer, out. It uses Des3 encryption with Electronic Codebook (ECB) mode. Warning: In nearly all use cases ECB mode is considered to be less secure. Please avoid using ECB API's directly whenever possible.
int	**wc_Des3_SetKey**(Des3 * des, const byte * key, const byte * iv, int dir)This function sets the key and initialization vector (iv) for the Des3 structure given as argument. It also initializes and allocates space for the buffers needed for encryption and decryption, if these have not yet been initialized. Note: If no iv is provided (i.e. iv == NULL) the initialization vector defaults to an iv of 0.
int	**wc_Des3_SetIV**(Des3 * des, const byte * iv)This function sets the initialization vector (iv) for the Des3 structure given as argument. When passed a NULL iv, it sets the initialization vector to 0.
int	**wc_Des3_CbcEncrypt**(Des3 * des, byte * out, const byte * in, word32 sz)This function encrypts the input message, in, and stores the result in the output buffer, out. It uses Triple Des (3DES) encryption with cipher block chaining (CBC) mode.
int	**wc_Des3_CbcDecrypt**(Des3 * des, byte * out, const byte * in, word32 sz)This function decrypts the input ciphertext, in, and stores the resulting plaintext in the output buffer, out. It uses Triple Des (3DES) encryption with cipher block chaining (CBC) mode.
int	**wc_Des_CbcDecryptWithKey**(byte * out, const byte * in, word32 sz, const byte * key, const byte * iv)This function decrypts the input ciphertext, in, and stores the resulting plaintext in the output buffer, out. It uses DES encryption with cipher block chaining (CBC) mode. This function is a substitute for wc_Des_CbcDecrypt, allowing the user to decrypt a message without directly instantiating a Des structure.

	Name
int	**wc_Des_CbcEncryptWithKey**(byte * out, const byte * in, word32 sz, const byte * key, const byte * iv)This function encrypts the input plaintext, in, and stores the resulting ciphertext in the output buffer, out. It uses DES encryption with cipher block chaining (CBC) mode. This function is a substitute for wc_Des_CbcEncrypt, allowing the user to encrypt a message without directly instantiating a Des structure.
int	**wc_Des3_CbcEncryptWithKey**(byte * out, const byte * in, word32 sz, const byte * key, const byte * iv)This function encrypts the input plaintext, in, and stores the resulting ciphertext in the output buffer, out. It uses Triple DES (3DES) encryption with cipher block chaining (CBC) mode. This function is a substitute for wc_Des3_CbcEncrypt, allowing the user to encrypt a message without directly instantiating a Des3 structure.
int	**wc_Des3_CbcDecryptWithKey**(byte * out, const byte * in, word32 sz, const byte * key, const byte * iv)This function decrypts the input ciphertext, in, and stores the resulting plaintext in the output buffer, out. It uses Triple Des (3DES) encryption with cipher block chaining (CBC) mode. This function is a substitute for wc_Des3_CbcDecrypt, allowing the user to decrypt a message without directly instantiating a Des3 structure.

B.12.2 Functions Documentation

```
int wc_Des_SetKey(
    Des * des,
    const byte * key,
    const byte * iv,
    int dir
)
```

This function sets the key and initialization vector (iv) for the Des structure given as argument. It also initializes and allocates space for the buffers needed for encryption and decryption, if these have not yet been initialized. Note: If no iv is provided (i.e. iv == NULL) the initialization vector defaults to an iv of 0.

Parameters:

- **des** pointer to the Des structure to initialize
- **key** pointer to the buffer containing the 8 byte key with which to initialize the Des structure
- **iv** pointer to the buffer containing the 8 byte iv with which to initialize the Des structure. If this is not provided, the iv defaults to 0
- **dir** direction of encryption. Valid options are: DES_ENCRYPTION, and DES_DECRYPTION

See:

- wc_Des_SetIV
- wc_Des3_SetKey

Return: 0 On successfully setting the key and initialization vector for the Des structure

3

Example

```
Des enc; // Des structure used for encryption
int ret;
byte key[] = { // initialize with 8 byte key };
byte iv[]  = { // initialize with 8 byte iv };

ret = wc_Des_SetKey(&des, key, iv, DES_ENCRYPTION);
if (ret != 0) {
    // error initializing des structure
}
```

```
void wc_Des_SetIV(
    Des * des,
    const byte * iv
)
```

This function sets the initialization vector (iv) for the Des structure given as argument. When passed a NULL iv, it sets the initialization vector to 0.

Parameters:

- **des** pointer to the Des structure for which to set the iv
- **iv** pointer to the buffer containing the 8 byte iv with which to initialize the Des structure. If this is not provided, the iv defaults to 0

See: wc_Des_SetKey

Return: none No returns.

3

Example

```
Des enc; // Des structure used for encryption
// initialize enc with wc_Des_SetKey
byte iv[]  = { // initialize with 8 byte iv };
wc_Des_SetIV(&enc, iv);
}
```

```
int wc_Des_CbcEncrypt(
    Des * des,
    byte * out,
    const byte * in,
    word32 sz
)
```

This function encrypts the input message, in, and stores the result in the output buffer, out. It uses DES encryption with cipher block chaining (CBC) mode.

Parameters:

- **des** pointer to the Des structure to use for encryption
- **out** pointer to the buffer in which to store the encrypted ciphertext
- **in** pointer to the input buffer containing the message to encrypt
- **sz** length of the message to encrypt

See:

- wc_Des_SetKey
- wc_Des_CbcDecrypt

Return: 0 Returned upon successfully encrypting the given input message

3

Example

```
Des enc; // Des structure used for encryption
// initialize enc with wc_Des_SetKey, use mode DES_ENCRYPTION

byte plain[]  = { // initialize with message };
byte cipher[sizeof(plain)];

if ( wc_Des_CbcEncrypt(&enc, cipher, plain, sizeof(plain)) != 0) {
    // error encrypting message
}
```

```
int wc_Des_CbcDecrypt(
    Des * des,
    byte * out,
    const byte * in,
    word32 sz
)
```

This function decrypts the input ciphertext, in, and stores the resulting plaintext in the output buffer, out. It uses DES encryption with cipher block chaining (CBC) mode.

Parameters:

- **des** pointer to the Des structure to use for decryption

- **out** pointer to the buffer in which to store the decrypted plaintext
- **in** pointer to the input buffer containing the encrypted ciphertext
- **sz** length of the ciphertext to decrypt

See:

- wc_Des_SetKey
- wc_Des_CbcEncrypt

Return: 0 Returned upon successfully decrypting the given ciphertext

3

Example

```
Des dec; // Des structure used for decryption
// initialize dec with wc_Des_SetKey, use mode DES_DECRYPTION

byte cipher[]  = { // initialize with ciphertext };
byte decoded[sizeof(cipher)];

if ( wc_Des_CbcDecrypt(&dec, decoded, cipher, sizeof(cipher)) != 0) {
    // error decrypting message
}
```

```
int wc_Des_EcbEncrypt(
    Des * des,
    byte * out,
    const byte * in,
    word32 sz
)
```

This function encrypts the input message, in, and stores the result in the output buffer, out. It uses Des encryption with Electronic Codebook (ECB) mode.

Parameters:

- **des** pointer to the Des structure to use for encryption
- **out** pointer to the buffer in which to store the encrypted message
- **in** pointer to the input buffer containing the plaintext to encrypt
- **sz** length of the plaintext to encrypt

See: wc_Des_SetKe

Return: 0: Returned upon successfully encrypting the given plaintext.

3

Example

```
Des enc; // Des structure used for encryption
// initialize enc with wc_Des_SetKey, use mode DES_ENCRYPTION
```

```
byte plain[]  = { // initialize with message to encrypt };
byte cipher[sizeof(plain)];

if ( wc_Des_EcbEncrypt(&enc,cipher, plain, sizeof(plain)) != 0) {
    // error encrypting message
}
```

```
int wc_Des3_EcbEncrypt(
    Des3 * des,
    byte * out,
    const byte * in,
    word32 sz
)
```

This function encrypts the input message, in, and stores the result in the output buffer, out. It uses Des3 encryption with Electronic Codebook (ECB) mode. Warning: In nearly all use cases ECB mode is considered to be less secure. Please avoid using ECB API's directly whenever possible.

Parameters:

- **des3** pointer to the Des3 structure to use for encryption
- **out** pointer to the buffer in which to store the encrypted message
- **in** pointer to the input buffer containing the plaintext to encrypt
- **sz** length of the plaintext to encrypt

See: wc_Des3_SetKey

Return: 0 Returned upon successfully encrypting the given plaintext

3

Example

```
Des3 enc; // Des3 structure used for encryption
// initialize enc with wc_Des3_SetKey, use mode DES_ENCRYPTION

byte plain[]  = { // initialize with message to encrypt };
byte cipher[sizeof(plain)];

if ( wc_Des3_EcbEncrypt(&enc,cipher, plain, sizeof(plain)) != 0) {
    // error encrypting message
}
```

```
int wc_Des3_SetKey(
    Des3 * des,
    const byte * key,
    const byte * iv,
    int dir
)
```

This function sets the key and initialization vector (iv) for the Des3 structure given as argument. It also initializes and allocates space for the buffers needed for encryption and decryption, if these have not yet been initialized. Note: If no iv is provided (i.e. iv == NULL) the initialization vector defaults to an iv of 0.

Parameters:

- **des3** pointer to the Des3 structure to initialize
- **key** pointer to the buffer containing the 24 byte key with which to initialize the Des3 structure
- **iv** pointer to the buffer containing the 8 byte iv with which to initialize the Des3 structure. If this is not provided, the iv defaults to 0
- **dir** direction of encryption. Valid options are: DES_ENCRYPTION, and DES_DECRYPTION

See:

- wc_Des3_SetIV
- wc_Des3_CbcEncrypt
- wc_Des3_CbcDecrypt

Return: 0 On successfully setting the key and initialization vector for the Des structure

3

Example

```
Des3 enc; // Des3 structure used for encryption
int ret;
byte key[] = { // initialize with 24 byte key };
byte iv[]  = { // initialize with 8 byte iv };

ret = wc_Des3_SetKey(&des, key, iv, DES_ENCRYPTION);
if (ret != 0) {
    // error initializing des structure
}
```

```
int wc_Des3_SetIV(
    Des3 * des,
    const byte * iv
)
```

This function sets the initialization vector (iv) for the Des3 structure given as argument. When passed a NULL iv, it sets the initialization vector to 0.

Parameters:

- **des** pointer to the Des3 structure for which to set the iv
- **iv** pointer to the buffer containing the 8 byte iv with which to initialize the Des3 structure. If this is not provided, the iv defaults to 0

See: wc_Des3_SetKey

Return: none No returns.

3

Example

```
Des3 enc; // Des3 structure used for encryption
// initialize enc with wc_Des3_SetKey

byte iv[]  = { // initialize with 8 byte iv };

wc_Des3_SetIV(&enc, iv);
}
```

```
int wc_Des3_CbcEncrypt(
    Des3 * des,
    byte * out,
    const byte * in,
    word32 sz
)
```

This function encrypts the input message, in, and stores the result in the output buffer, out. It uses Triple Des (3DES) encryption with cipher block chaining (CBC) mode.

Parameters:

- **des** pointer to the Des3 structure to use for encryption
- **out** pointer to the buffer in which to store the encrypted ciphertext
- **in** pointer to the input buffer containing the message to encrypt
- **sz** length of the message to encrypt

See:

- wc_Des3_SetKey
- wc_Des3_CbcDecrypt

Return: 0 Returned upon successfully encrypting the given input message

3

Example

```
Des3 enc; // Des3 structure used for encryption
// initialize enc with wc_Des3_SetKey, use mode DES_ENCRYPTION

byte plain[]  = { // initialize with message };
byte cipher[sizeof(plain)];

if ( wc_Des3_CbcEncrypt(&enc, cipher, plain, sizeof(plain)) != 0) {
    // error encrypting message
}
```

```
int wc_Des3_CbcDecrypt(
    Des3 * des,
    byte * out,
    const byte * in,
    word32 sz
)
```

This function decrypts the input ciphertext, in, and stores the resulting plaintext in the output buffer, out. It uses Triple Des (3DES) encryption with cipher block chaining (CBC) mode.

Parameters:

- **des** pointer to the Des3 structure to use for decryption
- **out** pointer to the buffer in which to store the decrypted plaintext
- **in** pointer to the input buffer containing the encrypted ciphertext
- **sz** length of the ciphertext to decrypt

See:

- wc_Des3_SetKey
- wc_Des3_CbcEncrypt

Return: 0 Returned upon successfully decrypting the given ciphertext

3

Example

```
Des3 dec; // Des structure used for decryption
// initialize dec with wc_Des3_SetKey, use mode DES_DECRYPTION

byte cipher[]  = { // initialize with ciphertext };
byte decoded[sizeof(cipher)];

if ( wc_Des3_CbcDecrypt(&dec, decoded, cipher, sizeof(cipher)) != 0) {
    // error decrypting message
}
```

```
int wc_Des_CbcDecryptWithKey(
    byte * out,
    const byte * in,
    word32 sz,
    const byte * key,
    const byte * iv
)
```

This function decrypts the input ciphertext, in, and stores the resulting plaintext in the output buffer, out. It uses DES encryption with cipher block chaining (CBC) mode. This function is a substitute for wc_Des_CbcDecrypt, allowing the user to decrypt a message without directly instantiating a Des structure.

Parameters:

- **out** pointer to the buffer in which to store the decrypted plaintext
- **in** pointer to the input buffer containing the encrypted ciphertext
- **sz** length of the ciphertext to decrypt
- **key** pointer to the buffer containing the 8 byte key to use for decryption
- **iv** pointer to the buffer containing the 8 byte iv to use for decryption. If no iv is provided, the iv defaults to 0

See: wc_Des_CbcDecrypt

Return:

- 0 Returned upon successfully decrypting the given ciphertext
- MEMORY_E Returned if there is an error allocating space for a Des structure

3

Example

```
int ret;
byte key[] = { // initialize with 8 byte key };
byte iv[]  = { // initialize with 8 byte iv };

byte cipher[]  = { // initialize with ciphertext };
byte decoded[sizeof(cipher)];

if ( wc_Des_CbcDecryptWithKey(decoded, cipher, sizeof(cipher), key,
iv) != 0) {
    // error decrypting message
}
```

```
int wc_Des_CbcEncryptWithKey(
    byte * out,
    const byte * in,
    word32 sz,
    const byte * key,
    const byte * iv
)
```

This function encrypts the input plaintext, in, and stores the resulting ciphertext in the output buffer, out. It uses DES encryption with cipher block chaining (CBC) mode. This function is a substitute for wc_Des_CbcEncrypt, allowing the user to encrypt a message without directly instantiating a Des structure.

Parameters:

- **out** Final encrypted data
- **in** Data to be encrypted, must be padded to Des block size.
- **sz** Size of input buffer.
- **key** Pointer to the key to use for encryption.
- **iv** Initialization vector

See:

- wc_Des_CbcDecryptWithKey
- wc_Des_CbcEncrypt

Return:

- 0 Returned after successfully encrypting data.
- MEMORY_E Returned if there's an error allocating memory for a Des structure.
- <0 Returned on any error during encryption.

3

Example

```
byte key[] = { // initialize with 8 byte key };
byte iv[]  = { // initialize with 8 byte iv };
byte in[] = { // Initialize with plaintext };
byte out[sizeof(in)];
if ( wc_Des_CbcEncryptWithKey(&out, in, sizeof(in), key, iv) != 0)
{
    // error encrypting message
}
```

```
int wc_Des3_CbcEncryptWithKey(
    byte * out,
    const byte * in,
    word32 sz,
    const byte * key,
    const byte * iv
)
```

This function encrypts the input plaintext, in, and stores the resulting ciphertext in the output buffer, out. It uses Triple DES (3DES) encryption with cipher block chaining (CBC) mode. This function is a substitute for wc_Des3_CbcEncrypt, allowing the user to encrypt a message without directly instantiating a Des3 structure.

Parameters:

- **out** Final encrypted data
- **in** Data to be encrypted, must be padded to Des block size.
- **sz** Size of input buffer.
- **key** Pointer to the key to use for encryption.
- **iv** Initialization vector

See:

- wc_Des3_CbcDecryptWithKey
- wc_Des_CbcEncryptWithKey
- wc_Des_CbcDecryptWithKey

Return:

- 0 Returned after successfully encrypting data.
- MEMORY_E Returned if there's an error allocating memory for a Des structure.
- <0 Returned on any error during encryption.

3

Example

```
byte key[] = { // initialize with 8 byte key };
byte iv[]  = { // initialize with 8 byte iv };

byte in[] = { // Initialize with plaintext };
byte out[sizeof(in)];

if ( wc_Des3_CbcEncryptWithKey(&out, in, sizeof(in), key, iv) != 0)
{
    // error encrypting message
}
```

```
int wc_Des3_CbcDecryptWithKey(
    byte * out,
    const byte * in,
    word32 sz,
    const byte * key,
    const byte * iv
)
```

This function decrypts the input ciphertext, in, and stores the resulting plaintext in the output buffer, out. It uses Triple Des (3DES) encryption with cipher block chaining (CBC) mode. This function is a substitute for wc_Des3_CbcDecrypt, allowing the user to decrypt a message without directly instantiating a Des3 structure.

Parameters:

- **out** pointer to the buffer in which to store the decrypted plaintext
- **in** pointer to the input buffer containing the encrypted ciphertext
- **sz** length of the ciphertext to decrypt
- **key** pointer to the buffer containing the 24 byte key to use for decryption
- **iv** pointer to the buffer containing the 8 byte iv to use for decryption. If no iv is provided, the iv defaults to 0

See: wc_Des3_CbcDecrypt

Return:

- 0 Returned upon successfully decrypting the given ciphertext
- MEMORY_E Returned if there is an error allocating space for a Des structure

3

Example

```
int ret;
byte key[] = { // initialize with 24 byte key };
byte iv[]  = { // initialize with 8 byte iv };

byte cipher[]  = { // initialize with ciphertext };
byte decoded[sizeof(cipher)];

if ( wc_Des3_CbcDecryptWithKey(decoded, cipher, sizeof(cipher),
key, iv) != 0) {
    // error decrypting message
}
```

B.13 Algorithms - AES

B.12.2.14 function wc_Des3_CbcDecryptWithKey

B.13.1 Functions

	Name
int	**wc_AesSetKey**(Aes * aes, const byte * key, word32 len, const byte * iv, int dir)This function initializes an AES structure by setting the key and then setting the initialization vector.
int	**wc_AesSetIV**(Aes * aes, const byte * iv)This function sets the initialization vector for a particular AES object. The AES object should be initialized before calling this function.

	Name
int	**wc_AesCbcEncrypt**(Aes * aes, byte * out, const byte * in, word32 sz)Encrypts a plaintext message from the input buffer in, and places the resulting cipher text in the output buffer out using cipher block chaining with AES. This function requires that the AES object has been initialized by calling AesSetKey before a message is able to be encrypted. This function assumes that the input message is AES block length aligned, and expects the input length to be a multiple of the block length, which will optionally be checked and enforced if WOLFSSL_AES_CBC_LENGTH_CHECKS is defined in the build configuration. In order to assure block-multiple input, PKCS#7 style padding should be added beforehand. This differs from the OpenSSL AES-CBC methods which add the padding for you. To make the wolfSSL and corresponding OpenSSL functions interoperate, one should specify the -nopad option in the OpenSSL command line function so that it behaves like the wolfSSL AesCbcEncrypt method and does not add extra padding during encryption.
int	**wc_AesCbcDecrypt**(Aes * aes, byte * out, const byte * in, word32 sz)Decrypts a cipher from the input buffer in, and places the resulting plain text in the output buffer out using cipher block chaining with AES. This function requires that the AES structure has been initialized by calling AesSetKey before a message is able to be decrypted. This function assumes that the original message was AES block length aligned, and expects the input length to be a multiple of the block length, which will optionally be checked and enforced if WOLFSSL_AES_CBC_LENGTH_CHECKS is defined in the build configuration. This differs from the OpenSSL AES-CBC methods, which add PKCS#7 padding automatically, and so do not require block-multiple input. To make the wolfSSL function and equivalent OpenSSL functions interoperate, one should specify the -nopad option in the OpenSSL command line function so that it behaves like the wolfSSL AesCbcEncrypt method and does not create errors during decryption.

	Name
int	**wc_AesCtrEncrypt**(Aes * aes, byte * out, const byte * in, word32 sz)Encrypts/Decrypts a message from the input buffer in, and places the resulting cipher text in the output buffer out using CTR mode with AES. This function is only enabled if WOLFSSL_AES_COUNTER is enabled at compile time. The AES structure should be initialized through AesSetKey before calling this function. Note that this function is used for both decryption and encryption. *NOTE:* Regarding using same API for encryption and decryption. User should differentiate between Aes structures for encrypt/decrypt.
int	**wc_AesEncryptDirect**(Aes * aes, byte * out, const byte * in)This function is a one-block encrypt of the input block, in, into the output block, out. It uses the key of the provided AES structure, which should be initialized with wc_AesSetKey before calling this function. wc_AesSetKey should have been called with the iv set to NULL. This is only enabled if the configure option WOLFSSL_AES_DIRECT is enabled. **Warning:** In nearly all use cases ECB mode is considered to be less secure. Please avoid using ECB API's directly whenever possible.
int	**wc_AesDecryptDirect**(Aes * aes, byte * out, const byte * in)This function is a one-block decrypt of the input block, in, into the output block, out. It uses the key of the provided AES structure, which should be initialized with wc_AesSetKey before calling this function. wc_AesSetKey should have been called with the iv set to NULL. This is only enabled if the configure option WOLFSSL_AES_DIRECT is enabled. **Warning:** In nearly all use cases ECB mode is considered to be less secure. Please avoid using ECB API's directly whenever possible.
int	**wc_AesSetKeyDirect**(Aes * aes, const byte * key, word32 len, const byte * iv, int dir)This function is used to set the AES keys for CTR mode with AES. It initializes an AES object with the given key, iv (initialization vector), and encryption dir (direction). It is only enabled if the configure option WOLFSSL_AES_DIRECT is enabled. Currently wc_AesSetKeyDirect uses wc_AesSetKey internally. **Warning:** In nearly all use cases ECB mode is considered to be less secure. Please avoid using ECB API's directly whenever possible.

	Name
int	**wc_AesGcmSetKey**(Aes * aes, const byte * key, word32 len)This function is used to set the key for AES GCM (Galois/Counter Mode). It initializes an AES object with the given key. It is only enabled if the configure option HAVE_AESGCM is enabled at compile time.
int	**wc_AesGcmEncrypt**(Aes * aes, byte * out, const byte * in, word32 sz, const byte * iv, word32 ivSz, byte * authTag, word32 authTagSz, const byte * authIn, word32 authInSz)This function encrypts the input message, held in the buffer in, and stores the resulting cipher text in the output buffer out. It requires a new iv (initialization vector) for each call to encrypt. It also encodes the input authentication vector, authIn, into the authentication tag, authTag.
int	**wc_AesGcmDecrypt**(Aes * aes, byte * out, const byte * in, word32 sz, const byte * iv, word32 ivSz, const byte * authTag, word32 authTagSz, const byte * authIn, word32 authInSz)This function decrypts the input cipher text, held in the buffer in, and stores the resulting message text in the output buffer out. It also checks the input authentication vector, authIn, against the supplied authentication tag, authTag. If a nonzero error code is returned, the output data is undefined. However, callers must unconditionally zeroize the output buffer to guard against leakage of cleartext data.
int	**wc_GmacSetKey**(Gmac * gmac, const byte * key, word32 len)This function initializes and sets the key for a GMAC object to be used for Galois Message Authentication.
int	**wc_GmacUpdate**(Gmac * gmac, const byte * iv, word32 ivSz, const byte * authIn, word32 authInSz, byte * authTag, word32 authTagSz)This function generates the Gmac hash of the authIn input and stores the result in the authTag buffer. After running wc_GmacUpdate, one should compare the generated authTag to a known authentication tag to verify the authenticity of a message.
int	**wc_AesCcmSetKey**(Aes * aes, const byte * key, word32 keySz)This function sets the key for an AES object using CCM (Counter with CBC_MAC). It takes a pointer to an AES structure and initializes it with supplied key.

	Name
int	**wc_AesCcmEncrypt**(Aes * aes, byte * out, const byte * in, word32 inSz, const byte * nonce, word32 nonceSz, byte * authTag, word32 authTagSz, const byte * authIn, word32 authInSz)This function encrypts the input message, in, into the output buffer, out, using CCM (Counter with CBC_MAC). It subsequently calculates and stores the authorization tag, authTag, from the authIn input.
int	**wc_AesCcmDecrypt**(Aes * aes, byte * out, const byte * in, word32 inSz, const byte * nonce, word32 nonceSz, const byte * authTag, word32 authTagSz, const byte * authIn, word32 authInSz)This function decrypts the input cipher text, in, into the output buffer, out, using CCM (Counter with CBC_MAC). It subsequently calculates the authorization tag, authTag, from the authIn input. If a nonzero error code is returned, the output data is undefined. However, callers must unconditionally zeroize the output buffer to guard against leakage of cleartext data.
int	**wc_AesXtsInit**(XtsAes * aes, void * heap, int devId)This is to initialize an AES-XTS context. It is up to user to call wc_AesXtsFree on aes key when done.
int	**wc_AesXtsSetKeyNoInit. It is up to user to call wc_AesXtsFree on aes key when done.
int	**wc_AesXtsSetKey**(XtsAes * aes, const byte * key, word32 len, int dir, void * heap, int devId)This is to help with setting keys to correct encrypt or decrypt type. It is up to user to call wc_AesXtsFree on aes key when done.
int	**wc_AesXtsEncryptSector**(XtsAes * aes, byte * out, const byte * in, word32 sz, word64 sector)Same process as wc_AesXtsEncrypt but uses a word64 type as the tweak value instead of a byte array. This just converts the word64 to a byte array and calls wc_AesXtsEncrypt.
int	**wc_AesXtsDecryptSector**(XtsAes * aes, byte * out, const byte * in, word32 sz, word64 sector)Same process as wc_AesXtsDecrypt but uses a word64 type as the tweak value instead of a byte array. This just converts the word64 to a byte array.
int	**wc_AesXtsEncrypt**(XtsAes * aes, byte * out, const byte * in, word32 sz, const byte * i, word32 iSz)AES with XTS mode. (XTS) XEX encryption with Tweak and cipher text Stealing.

	Name
int	**wc_AesXtsDecrypt**(XtsAes * aes, byte * out, const byte * in, word32 sz, const byte * i, word32 iSz)Same process as encryption but Aes key is AES_DECRYPTION type.
int	**wc_AesXtsFree**(XtsAes * aes)This is to free up any resources used by the XtsAes structure.
int	**wc_AesInit**(Aes * aes, void * heap, int devId)Initialize Aes structure. Sets heap hint to be used and ID for use with async hardware. It is up to the user to call wc_AesFree on the Aes structure when done.
int	**wc_AesFree**(Aes * aes)free resources associated with the Aes structure when applicable. Internally may sometimes be a no_op but still recommended to call in all cases as a general best_practice (IE if application code is ported for use on new environments where the call is applicable).
int	**wc_AesCfbEncrypt**(Aes * aes, byte * out, const byte * in, word32 sz)AES with CFB mode.
int	**wc_AesCfbDecrypt**(Aes * aes, byte * out, const byte * in, word32 sz)AES with CFB mode.
int	**wc_AesSivEncrypt**(const byte * key, word32 keySz, const byte * assoc, word32 assocSz, const byte * nonce, word32 nonceSz, const byte * in, word32 inSz, byte * siv, byte * out)This function performs SIV (synthetic initialization vector) encryption as described in RFC 5297.
int	**wc_AesSivDecrypt**(const byte * key, word32 keySz, const byte * assoc, word32 assocSz, const byte * nonce, word32 nonceSz, const byte * in, word32 inSz, byte * siv, byte * out)This function performs SIV (synthetic initialization vector) decryption as described in RFC 5297. If a nonzero error code is returned, the output data is undefined. However, callers must unconditionally zeroize the output buffer to guard against leakage of cleartext data.
WOLFSSL_API int	**wc_AesEaxEncryptAuth**(const byte * key, word32 keySz, byte * out, const byte * in, word32 inSz, const byte * nonce, word32 nonceSz, byte * authTag, word32 authTagSz, const byte * authIn, word32 authInSz)This function performs AES EAX encryption and authentication as described in “EAX: A Conventional Authenticated_Encryption Mode” (https://eprint.iacr.org/2003/069). It is a “one-shot” API that performs all encryption and authentication operations in one function call.

	Name
WOLFSSL_API int	**wc_AesEaxDecryptAuth**(const byte * key, word32 keySz, byte * out, const byte * in, word32 inSz, const byte * nonce, word32 nonceSz, const byte * authTag, word32 authTagSz, const byte * authIn, word32 authInSz)This function performs AES EAX decryption and authentication as described in "EAX: A Conventional Authenticated_Encryption Mode" (https://eprint.iacr.org/2003/069). It is a "one-shot" API that performs all decryption and authentication operations in one function call. If a nonzero error code is returned, the output data is undefined. However, callers must unconditionally zeroize the output buffer to guard against leakage of cleartext data.
WOLFSSL_API int	**wc_AesEaxInit**(AesEax * eax, const byte * key, word32 keySz, const byte * nonce, word32 nonceSz, const byte * authIn, word32 authInSz)This function initializes an AesEax object for use in authenticated encryption or decryption. This function must be called on an AesEax object before using it with any of the AES EAX incremental API functions. It does not need to be called if using the one_shot EAX API functions. All AesEax instances initialized with this function need to be freed with a call to wc_AesEaxFree() when done using the instance.
WOLFSSL_API int	**wc_AesEaxEncryptUpdate.
WOLFSSL_API int	**wc_AesEaxDecryptUpdate.
WOLFSSL_API int	**wc_AesEaxAuthDataUpdate.
WOLFSSL_API int	**wc_AesEaxEncryptFinal. When done using the AesEax context structure, make sure to free it using wc_AesEaxFree.
WOLFSSL_API int	**wc_AesEaxDecryptFinal. When done using the AesEax context structure, make sure to free it using wc_AesEaxFree.
WOLFSSL_API int	**wc_AesEaxFree**(AesEax * eax)This frees up any resources, specifically keys, used by the Aes instance inside the AesEax wrapper struct. It should be called on the AesEax struct after it has been initialized with wc_AesEaxInit, and all desired EAX operations are complete.
int	**wc_AesCtsEncrypt**(const byte * key, word32 keySz, byte * out, const byte * in, word32 inSz, const byte * iv)This function performs AES encryption using Ciphertext Stealing (CTS) mode. It is a one-shot API that handles all operations in a single call.

	Name
int	**wc_AesCtsDecrypt**(const byte * key, word32 keySz, byte * out, const byte * in, word32 inSz, const byte * iv)This function performs AES decryption using Ciphertext Stealing (CTS) mode. It is a one-shot API that handles all operations in a single call.
int	**wc_AesCtsEncryptUpdate**(Aes * aes, byte * out, word32 * outSz, const byte * in, word32 inSz)This function performs an update step of the AES CTS encryption. It processes a chunk of plaintext and stores intermediate data.
int	**wc_AesCtsEncryptFinal**(Aes * aes, byte * out, word32 * outSz)This function finalizes the AES CTS encryption operation. It processes any remaining plaintext and completes the encryption.
int	**wc_AesCtsDecryptUpdate**(Aes * aes, byte * out, word32 * outSz, const byte * in, word32 inSz)This function performs an update step of the AES CTS decryption. It processes a chunk of ciphertext and stores intermediate data.
int	**wc_AesCtsDecryptFinal**(Aes * aes, byte * out, word32 * outSz)This function finalizes the AES CTS decryption operation. It processes any remaining ciphertext and completes the decryption.
int	**wc_AesCbcDecryptWithKey**(byte * out, const byte * in, word32 inSz, const byte * key, word32 keySz, const byte * iv)Decrypts a cipher from the input buffer in, and places the resulting plain text in the output buffer out using cipher block chaining with AES. This function does not require an AES structure to be initialized. Instead, it takes in a key and an iv (initialization vector) and uses these to initialize an AES object and then decrypt the cipher text.

B.13.2 Functions Documentation

```
int wc_AesSetKey(
    Aes * aes,
    const byte * key,
    word32 len,
    const byte * iv,
    int dir
)
```

This function initializes an AES structure by setting the key and then setting the initialization vector.

Parameters:

- **aes** pointer to the AES structure to modify

- **key** 16, 24, or 32 byte secret key for encryption and decryption
- **len** length of the key passed in
- **iv** pointer to the initialization vector used to initialize the key
- **dir** Cipher direction. Set AES_ENCRYPTION to encrypt, or AES_DECRYPTION to decrypt. Direction for some modes (CFB and CTR) is always AES_ENCRYPTION.

See:

- wc_AesSetKeyDirect
- wc_AesSetIV

Return:

- 0 On successfully setting key and initialization vector.
- BAD_FUNC_ARG Returned if key length is invalid.

Example

```
Aes enc;
int ret = 0;
byte key[] = { some 16, 24 or 32 byte key };
byte iv[]  = { some 16 byte iv };
if (ret = wc_AesInit(&enc, HEAP_HINT, INVALID_DEVID) != 0) {
    // failed to initialize aes key
}
if (ret = wc_AesSetKey(&enc, key, AES_BLOCK_SIZE, iv,
AES_ENCRYPTION) != 0) {
// failed to set aes key
}
```

```
int wc_AesSetIV(
    Aes * aes,
    const byte * iv
)
```

This function sets the initialization vector for a particular AES object. The AES object should be initialized before calling this function.

Parameters:

- **aes** pointer to the AES structure on which to set the initialization vector
- **iv** initialization vector used to initialize the AES structure. If the value is NULL, the default action initializes the iv to 0.

See:

- wc_AesSetKeyDirect
- wc_AesSetKey

Return:

- 0 On successfully setting initialization vector.
- BAD_FUNC_ARG Returned if AES pointer is NULL.

Example

```
Aes enc;
// set enc key
byte iv[]  = { some 16 byte iv };
if (ret = wc_AesSetIV(&enc, iv) != 0) {
// failed to set aes iv
}
```

```
int wc_AesCbcEncrypt(
    Aes * aes,
    byte * out,
    const byte * in,
    word32 sz
)
```

Encrypts a plaintext message from the input buffer in, and places the resulting cipher text in the output buffer out using cipher block chaining with AES. This function requires that the AES object has been initialized by calling AesSetKey before a message is able to be encrypted. This function assumes that the input message is AES block length aligned, and expects the input length to be a multiple of the block length, which will optionally be checked and enforced if WOLFSSL_AES_CBC_LENGTH_CHECKS is defined in the build configuration. In order to assure block-multiple input, PKCS#7 style padding should be added beforehand. This differs from the OpenSSL AES-CBC methods which add the padding for you. To make the wolfSSL and corresponding OpenSSL functions interoperate, one should specify the -nopad option in the OpenSSL command line function so that it behaves like the wolfSSL AesCbcEncrypt method and does not add extra padding during encryption.

Parameters:

- **aes** pointer to the AES object used to encrypt data
- **out** pointer to the output buffer in which to store the ciphertext of the encrypted message
- **in** pointer to the input buffer containing message to be encrypted
- **sz** size of input message

See:

- wc_AesInit
- wc_AesSetKey
- wc_AesSetIV
- wc_AesCbcDecrypt

Return:

- 0 On successfully encrypting message.

- BAD_ALIGN_E: may be returned on block align error
- BAD_LENGTH_E will be returned if the input length isn't a multiple of the AES block length, when the library is built with WOLFSSL_AES_CBC_LENGTH_CHECKS.

Example

```
Aes enc;
int ret = 0;
// initialize enc with wc_AesInit and wc_AesSetKey, using direction
// AES_ENCRYPTION
byte msg[AES_BLOCK_SIZE * n]; // multiple of 16 bytes
// fill msg with data
byte cipher[AES_BLOCK_SIZE * n]; // Some multiple of 16 bytes
if ((ret = wc_AesCbcEncrypt(&enc, cipher, message, sizeof(msg))) != 0 ) {
// block align error
}
```

```
int wc_AesCbcDecrypt(
    Aes * aes,
    byte * out,
    const byte * in,
    word32 sz
)
```

Decrypts a cipher from the input buffer in, and places the resulting plain text in the output buffer out using cipher block chaining with AES. This function requires that the AES structure has been initialized by calling AesSetKey before a message is able to be decrypted. This function assumes that the original message was AES block length aligned, and expects the input length to be a multiple of the block length, which will optionally be checked and enforced if WOLFSSL_AES_CBC_LENGTH_CHECKS is defined in the build configuration. This differs from the OpenSSL AES-CBC methods, which add PKCS#7 padding automatically, and so do not require block-multiple input. To make the wolfSSL function and equivalent OpenSSL functions interoperate, one should specify the -nopad option in the OpenSSL command line function so that it behaves like the wolfSSL AesCbcEncrypt method and does not create errors during decryption.

Parameters:

- **aes** pointer to the AES object used to decrypt data.
- **out** pointer to the output buffer in which to store the plain text of the decrypted message. size must be a multiple of AES_BLOCK_LENGTH, padded if necessary
- **in** pointer to the input buffer containing cipher text to be decrypted. size must be a multiple of AES_BLOCK_LENGTH, padded if necessary
- **sz** size of input message.

See:

- wc_AesInit
- wc_AesSetKey
- wc_AesCbcEncrypt

Return:

- 0 On successfully decrypting message.
- BAD_ALIGN_E may be returned on block align error.
- BAD_LENGTH_E will be returned if the input length isn't a multiple of the AES block length, when the library is built with WOLFSSL_AES_CBC_LENGTH_CHECKS.

Example

```
Aes dec;
int ret = 0;
// initialize dec with wc_AesInit and wc_AesSetKey, using direction
// AES_DECRYPTION
byte cipher[AES_BLOCK_SIZE * n]; // some multiple of 16 bytes
// fill cipher with cipher text
byte plain [AES_BLOCK_SIZE * n];
if ((ret = wc_AesCbcDecrypt(&dec, plain, cipher, sizeof(cipher))) != 0 ) {
// block align error
}
```

```
int wc_AesCtrEncrypt(
    Aes * aes,
    byte * out,
    const byte * in,
    word32 sz
)
```

Encrypts/Decrypts a message from the input buffer in, and places the resulting cipher text in the output buffer out using CTR mode with AES. This function is only enabled if WOLFSSL_AES_COUNTER is enabled at compile time. The AES structure should be initialized through AesSetKey before calling this function. Note that this function is used for both decryption and encryption. *NOTE:* Regarding using same API for encryption and decryption. User should differentiate between Aes structures for encrypt/decrypt.

Parameters:

- **aes** pointer to the AES object used to decrypt data
- **out** pointer to the output buffer in which to store the cipher text of the encrypted message size must be a multiple of AES_BLOCK_LENGTH, padded if necessary
- **in** pointer to the input buffer containing plain text to be encrypted size must be a multiple of AES_BLOCK_LENGTH, padded if necessary
- **sz** size of the input plain text

See: wc_AesSetKey

Return: int integer values corresponding to wolfSSL error or success status

Example

```
Aes enc;
Aes dec;
// initialize enc and dec with wc_AesInit and wc_AesSetKeyDirect, using
```

```
// direction AES_ENCRYPTION since the underlying API only calls Encrypt
// and by default calling encrypt on a cipher results in a decryption of
// the cipher

byte msg[AES_BLOCK_SIZE * n]; //n being a positive integer making msg
some multiple of 16 bytes
// fill plain with message text
byte cipher[AES_BLOCK_SIZE * n];
byte decrypted[AES_BLOCK_SIZE * n];
wc_AesCtrEncrypt(&enc, cipher, msg, sizeof(msg)); // encrypt plain
wc_AesCtrEncrypt(&dec, decrypted, cipher, sizeof(cipher));
// decrypt cipher text
```

```
int wc_AesEncryptDirect(
    Aes * aes,
    byte * out,
    const byte * in
)
```

This function is a one-block encrypt of the input block, in, into the output block, out. It uses the key of the provided AES structure, which should be initialized with wc_AesSetKey before calling this function. wc_AesSetKey should have been called with the iv set to NULL. This is only enabled if the configure option WOLFSSL_AES_DIRECT is enabled. **Warning:** In nearly all use cases ECB mode is considered to be less secure. Please avoid using ECB API's directly whenever possible.

Parameters:

- **aes** pointer to the AES object used to encrypt data
- **out** pointer to the output buffer in which to store the cipher text of the encrypted message
- **in** pointer to the input buffer containing plain text to be encrypted

See:

- wc_AesDecryptDirect
- wc_AesSetKeyDirect

Return: int integer values corresponding to wolfSSL error or success status

Example

```
Aes enc;
// initialize enc with wc_AesInit and wc_AesSetKey, using direction
// AES_ENCRYPTION
byte msg [AES_BLOCK_SIZE]; // 16 bytes
// initialize msg with plain text to encrypt
byte cipher[AES_BLOCK_SIZE];
wc_AesEncryptDirect(&enc, cipher, msg);
```

```
int wc_AesDecryptDirect(
    Aes * aes,
    byte * out,
    const byte * in
)
```

This function is a one-block decrypt of the input block, in, into the output block, out. It uses the key of the provided AES structure, which should be initialized with wc_AesSetKey before calling this function. wc_AesSetKey should have been called with the iv set to NULL. This is only enabled if the configure option WOLFSSL_AES_DIRECT is enabled. **Warning:** In nearly all use cases ECB mode is considered to be less secure. Please avoid using ECB API's directly whenever possible.

Parameters:

- **aes** pointer to the AES object used to encrypt data
- **out** pointer to the output buffer in which to store the plain text of the decrypted cipher text
- **in** pointer to the input buffer containing cipher text to be decrypted

See:

- wc_AesEncryptDirect
- wc_AesSetKeyDirect

Return: int integer values corresponding to wolfSSL error or success status

Example

```
Aes dec;
// initialize enc with wc_AesInit and wc_AesSetKey, using direction
// AES_DECRYPTION
byte cipher [AES_BLOCK_SIZE]; // 16 bytes
// initialize cipher with cipher text to decrypt
byte msg[AES_BLOCK_SIZE];
wc_AesDecryptDirect(&dec, msg, cipher);
```

```
int wc_AesSetKeyDirect(
    Aes * aes,
    const byte * key,
    word32 len,
    const byte * iv,
    int dir
)
```

This function is used to set the AES keys for CTR mode with AES. It initializes an AES object with the given key, iv (initialization vector), and encryption dir (direction). It is only enabled if the configure option WOLFSSL_AES_DIRECT is enabled. Currently wc_AesSetKeyDirect uses wc_AesSetKey internally. **Warning:** In nearly all use cases ECB mode is considered to be less secure. Please avoid using ECB API's directly whenever possible.

Parameters:

- **aes** pointer to the AES object used to encrypt data
- **key** 16, 24, or 32 byte secret key for encryption and decryption
- **len** length of the key passed in
- **iv** initialization vector used to initialize the key
- **dir** Cipher direction. Set AES_ENCRYPTION to encrypt, or AES_DECRYPTION to decrypt. (See enum in wolfssl/wolfcrypt/aes.h) (NOTE: If using wc_AesSetKeyDirect with Aes Counter mode (Stream cipher) only use AES_ENCRYPTION for both encrypting and decrypting)

See:

- wc_AesEncryptDirect
- wc_AesDecryptDirect
- wc_AesSetKey

Return:

- 0 On successfully setting the key.
- BAD_FUNC_ARG Returned if the given key is an invalid length.

Example

```
Aes enc;
int ret = 0;
byte key[] = { some 16, 24, or 32 byte key };
byte iv[]  = { some 16 byte iv };

if (ret = wc_AesInit(&enc, HEAP_HINT, INVALID_DEVID) != 0) {
    // failed to initialize aes key
}
if (ret = wc_AesSetKeyDirect(&enc, key, sizeof(key), iv,
AES_ENCRYPTION) != 0) {
// failed to set aes key
}
```

```
int wc_AesGcmSetKey(
    Aes * aes,
    const byte * key,
    word32 len
)
```

This function is used to set the key for AES GCM (Galois/Counter Mode). It initializes an AES object with the given key. It is only enabled if the configure option HAVE_AESGCM is enabled at compile time.

Parameters:

- **aes** pointer to the AES object used to encrypt data
- **key** 16, 24, or 32 byte secret key for encryption and decryption
- **len** length of the key passed in

See:

- wc_AesGcmEncrypt
- wc_AesGcmDecrypt

Return:

- 0 On successfully setting the key.
- BAD_FUNC_ARG Returned if the given key is an invalid length.

Example

```
Aes enc;
int ret = 0;
byte key[] = { some 16, 24,32 byte key };
if (ret = wc_AesInit(&enc, HEAP_HINT, INVALID_DEVID) != 0) {
    // failed to initialize aes key
}
if (ret = wc_AesGcmSetKey(&enc, key, sizeof(key)) != 0) {
// failed to set aes key
}
```

```
int wc_AesGcmEncrypt(
    Aes * aes,
    byte * out,
    const byte * in,
    word32 sz,
    const byte * iv,
    word32 ivSz,
    byte * authTag,
    word32 authTagSz,
    const byte * authIn,
    word32 authInSz
)
```

This function encrypts the input message, held in the buffer in, and stores the resulting cipher text in the output buffer out. It requires a new iv (initialization vector) for each call to encrypt. It also encodes the input authentication vector, authIn, into the authentication tag, authTag.

Parameters:

- **aes** - pointer to the AES object used to encrypt data
- **out** pointer to the output buffer in which to store the cipher text size must match in's size (sz)
- **in** pointer to the input buffer holding the message to encrypt size must be a multiple of AES_BLOCK_LENGTH, padded if necessary
- **sz** length of the input message to encrypt
- **iv** pointer to the buffer containing the initialization vector
- **ivSz** length of the initialization vector
- **authTag** pointer to the buffer in which to store the authentication tag
- **authTagSz** length of the desired authentication tag
- **authIn** pointer to the buffer containing the input authentication vector

- **authInSz** length of the input authentication vector

See:

- wc_AesGcmSetKey
- wc_AesGcmDecrypt

Return: 0 On successfully encrypting the input message

Example

```
Aes enc;
// initialize Aes structure by calling wc_AesInit() and wc_AesGcmSetKey

byte plain[AES_BLOCK_LENGTH * n]; //n being a positive integer
making plain some multiple of 16 bytes
// initialize plain with msg to encrypt
byte cipher[sizeof(plain)];
byte iv[] = // some 16 byte iv
byte authTag[AUTH_TAG_LENGTH];
byte authIn[] = // Authentication Vector

wc_AesGcmEncrypt(&enc, cipher, plain, sizeof(cipher), iv, sizeof(iv),
        authTag, sizeof(authTag), authIn, sizeof(authIn));
```

```
int wc_AesGcmDecrypt(
    Aes * aes,
    byte * out,
    const byte * in,
    word32 sz,
    const byte * iv,
    word32 ivSz,
    const byte * authTag,
    word32 authTagSz,
    const byte * authIn,
    word32 authInSz
)
```

This function decrypts the input cipher text, held in the buffer in, and stores the resulting message text in the output buffer out. It also checks the input authentication vector, authIn, against the supplied authentication tag, authTag. If a nonzero error code is returned, the output data is undefined. However, callers must unconditionally zeroize the output buffer to guard against leakage of cleartext data.

Parameters:

- **aes** pointer to the AES object used to encrypt data
- **out** pointer to the output buffer in which to store the message text size must match in's size (sz)
- **in** pointer to the input buffer holding the cipher text to decrypt size must be a multiple of AES_BLOCK_LENGTH, padded if necessary
- **sz** length of the cipher text to decrypt

- **iv** pointer to the buffer containing the initialization vector
- **ivSz** length of the initialization vector
- **authTag** pointer to the buffer containing the authentication tag
- **authTagSz** length of the desired authentication tag
- **authIn** pointer to the buffer containing the input authentication vector
- **authInSz** length of the input authentication vector

See:

- wc_AesGcmSetKey
- wc_AesGcmEncrypt

Return:

- 0 On successfully decrypting and authenticating the input message
- AES_GCM_AUTH_E If the authentication tag does not match the supplied authentication code vector, authTag.

Example

```
Aes enc; //can use the same struct as was passed to wc_AesGcmEncrypt
// initialize aes structure by calling wc_AesInit and wc_AesGcmSetKey
// if not already done

byte cipher[AES_BLOCK_LENGTH * n]; //n being a positive integer
making cipher some multiple of 16 bytes
// initialize cipher with cipher text to decrypt
byte output[sizeof(cipher)];
byte iv[] = // some 16 byte iv
byte authTag[AUTH_TAG_LENGTH];
byte authIn[] = // Authentication Vector

wc_AesGcmDecrypt(&enc, output, cipher, sizeof(cipher), iv, sizeof(iv),
        authTag, sizeof(authTag), authIn, sizeof(authIn));
```

```
int wc_GmacSetKey(
    Gmac * gmac,
    const byte * key,
    word32 len
)
```

This function initializes and sets the key for a GMAC object to be used for Galois Message Authentication.

Parameters:

- **gmac** pointer to the gmac object used for authentication
- **key** 16, 24, or 32 byte secret key for authentication
- **len** length of the key

See:

- wc_GmacUpdate
- wc_AesInit

Return:

- 0 On successfully setting the key
- BAD_FUNC_ARG Returned if key length is invalid.

Example

```
Gmac gmac;
key[] = { some 16, 24, or 32 byte length key };
wc_AesInit(gmac.aes, HEAP_HINT, INVALID_DEVID); // Make sure devId updated
wc_GmacSetKey(&gmac, key, sizeof(key));
```

```
int wc_GmacUpdate(
    Gmac * gmac,
    const byte * iv,
    word32 ivSz,
    const byte * authIn,
    word32 authInSz,
    byte * authTag,
    word32 authTagSz
)
```

This function generates the Gmac hash of the authIn input and stores the result in the authTag buffer. After running wc_GmacUpdate, one should compare the generated authTag to a known authentication tag to verify the authenticity of a message.

Parameters:

- **gmac** pointer to the gmac object used for authentication
- **iv** initialization vector used for the hash
- **ivSz** size of the initialization vector used
- **authIn** pointer to the buffer containing the authentication vector to verify
- **authInSz** size of the authentication vector
- **authTag** pointer to the output buffer in which to store the Gmac hash
- **authTagSz** the size of the output buffer used to store the Gmac hash

See:

- wc_GmacSetKey
- wc_AesInit

Return: 0 On successfully computing the Gmac hash.

Example

```
Gmac gmac;
key[] = { some 16, 24, or 32 byte length key };
iv[] = { some 16 byte length iv };

wc_AesInit(gmac.aes, HEAP_HINT, INVALID_DEVID); // Make sure devId updated
wc_GmacSetKey(&gmac, key, sizeof(key));
authIn[] = { some 16 byte authentication input };
tag[AES_BLOCK_SIZE]; // will store authentication code

wc_GmacUpdate(&gmac, iv, sizeof(iv), authIn, sizeof(authIn), tag,
sizeof(tag));
```

```
int wc_AesCcmSetKey(
    Aes * aes,
    const byte * key,
    word32 keySz
)
```

This function sets the key for an AES object using CCM (Counter with CBC-MAC). It takes a pointer to an AES structure and initializes it with supplied key.

Parameters:

- **aes** aes structure in which to store the supplied key
- **key** 16, 24, or 32 byte secret key for encryption and decryption
- **keySz** size of the supplied key

See:

- wc_AesCcmEncrypt
- wc_AesCcmDecrypt

Return: none

Example

```
Aes enc;
key[] = { some 16, 24, or 32 byte length key };

wc_AesInit(&enc, HEAP_HINT, INVALID_DEVID); // Make sure devId updated
wc_AesCcmSetKey(&enc, key, sizeof(key));
```

```
int wc_AesCcmEncrypt(
    Aes * aes,
    byte * out,
    const byte * in,
    word32 inSz,
    const byte * nonce,
    word32 nonceSz,
```

```
    byte * authTag,
    word32 authTagSz,
    const byte * authIn,
    word32 authInSz
)
```

This function encrypts the input message, in, into the output buffer, out, using CCM (Counter with CBC-MAC). It subsequently calculates and stores the authorization tag, authTag, from the authIn input.

Parameters:

- **aes** pointer to the AES object used to encrypt data
- **out** pointer to the output buffer in which to store the cipher text
- **in** pointer to the input buffer holding the message to encrypt
- **sz** length of the input message to encrypt
- **nonce** pointer to the buffer containing the nonce (number only used once)
- **nonceSz** length of the nonce
- **authTag** pointer to the buffer in which to store the authentication tag
- **authTagSz** length of the desired authentication tag
- **authIn** pointer to the buffer containing the input authentication vector
- **authInSz** length of the input authentication vector

See:

- wc_AesCcmSetKey
- wc_AesCcmDecrypt

Return: none

Example

```
Aes enc;
// initialize enc with wc_AesInit and wc_AesCcmSetKey

nonce[] = { initialize nonce };
plain[] = { some plain text message };
cipher[sizeof(plain)];

authIn[] = { some 16 byte authentication input };
tag[AES_BLOCK_SIZE]; // will store authentication code

wc_AesCcmEncrypt(&enc, cipher, plain, sizeof(plain), nonce, sizeof(nonce),
        tag, sizeof(tag), authIn, sizeof(authIn));
```

```
int wc_AesCcmDecrypt(
    Aes * aes,
    byte * out,
    const byte * in,
    word32 inSz,
    const byte * nonce,
    word32 nonceSz,
```

```
    const byte * authTag,
    word32 authTagSz,
    const byte * authIn,
    word32 authInSz
)
```

This function decrypts the input cipher text, in, into the output buffer, out, using CCM (Counter with CBC-MAC). It subsequently calculates the authorization tag, authTag, from the authIn input. If a nonzero error code is returned, the output data is undefined. However, callers must unconditionally zeroize the output buffer to guard against leakage of cleartext data.

Parameters:

- **aes** pointer to the AES object used to encrypt data
- **out** pointer to the output buffer in which to store the cipher text
- **in** pointer to the input buffer holding the message to encrypt
- **sz** length of the input cipher text to decrypt
- **nonce** pointer to the buffer containing the nonce (number only used once)
- **nonceSz** length of the nonce
- **authTag** pointer to the buffer in which to store the authentication tag
- **authTagSz** length of the desired authentication tag
- **authIn** pointer to the buffer containing the input authentication vector
- **authInSz** length of the input authentication vector

See:

- wc_AesCcmSetKey
- wc_AesCcmEncrypt

Return:

- 0 On successfully decrypting the input message
- AES_CCM_AUTH_E If the authentication tag does not match the supplied authentication code vector, authTag.

Example

```
Aes dec;
// initialize dec with wc_AesInit and wc_AesCcmSetKey

nonce[] = { initialize nonce };
cipher[] = { encrypted message };
plain[sizeof(cipher)];

authIn[] = { some 16 byte authentication input };
tag[AES_BLOCK_SIZE] = { authentication tag received for verification };

int return = wc_AesCcmDecrypt(&dec, plain, cipher, sizeof(cipher),
nonce, sizeof(nonce),tag, sizeof(tag), authIn, sizeof(authIn));
if(return != 0) {
// decrypt error, invalid authentication code
}
```

```
int wc_AesXtsInit(
    XtsAes * aes,
    void * heap,
    int devId
)
```

This is to initialize an AES-XTS context. It is up to user to call wc_AesXtsFree on aes key when done.

Parameters:

- **aes** AES keys for encrypt/decrypt process
- **heap** heap hint to use for memory. Can be NULL
- **devId** ID to use with crypto callbacks or async hardware. Set to INVALID_DEVID (-2) if not used

See:

- wc_AesXtsSetKey
- wc_AesXtsSetKeyNoInit
- wc_AesXtsEncrypt
- wc_AesXtsDecrypt
- wc_AesXtsFree

Return: 0 Success

Example

```
XtsAes aes;

if(wc_AesXtsInit(&aes, NULL, INVALID_DEVID) != 0)
{
    // Handle error
}
if(wc_AesXtsSetKeyNoInit(&aes, key, sizeof(key), AES_ENCRYPTION) != 0)
{
    // Handle error
}
wc_AesXtsFree(&aes);
```

```
int wc_AesXtsSetKeyNoInit(
    XtsAes * aes,
    const byte * key,
    word32 len,
    int dir
)
```

This is to help with setting keys to correct encrypt or decrypt type, after first calling wc_AesXtsInit(). It is up to user to call wc_AesXtsFree on aes key when done.

Parameters:

- **aes** AES keys for encrypt/decrypt process
- **key** buffer holding aes key | tweak key
- **len** length of key buffer in bytes. Should be twice that of key size. i.e. 32 for a 16 byte key.
- **dir** direction, either AES_ENCRYPTION or AES_DECRYPTION

See:

- wc_AesXtsEncrypt
- wc_AesXtsDecrypt
- wc_AesXtsFree

Return: 0 Success

Example

```
XtsAes aes;

if(wc_AesXtsInit(&aes, NULL, 0) != 0)
{
    // Handle error
}
if(wc_AesXtsSetKeyNoInit(&aes, key, sizeof(key), AES_ENCRYPTION, NULL, 0)
   != 0)
{
    // Handle error
}
wc_AesXtsFree(&aes);
```

```
int wc_AesXtsSetKey(
    XtsAes * aes,
    const byte * key,
    word32 len,
    int dir,
    void * heap,
    int devId
)
```

This is to help with setting keys to correct encrypt or decrypt type. It is up to user to call wc_AesXtsFree on aes key when done.

Parameters:

- **aes** AES keys for encrypt/decrypt process
- **key** buffer holding aes key | tweak key
- **len** length of key buffer in bytes. Should be twice that of key size. i.e. 32 for a 16 byte key.
- **dir** direction, either AES_ENCRYPTION or AES_DECRYPTION
- **heap** heap hint to use for memory. Can be NULL
- **devId** ID to use with crypto callbacks or async hardware. Set to INVALID_DEVID (-2) if not used

See:

- wc_AesXtsInit
- wc_AesXtsSetKeyNoInit
- wc_AesXtsEncrypt
- wc_AesXtsDecrypt
- wc_AesXtsFree

Return: 0 Success

Example

```
XtsAes aes;

if(wc_AesXtsSetKey(&aes, key, sizeof(key), AES_ENCRYPTION, NULL, INVALID_DEVID)
 ↪  != 0)
{
    // Handle error
}
wc_AesXtsFree(&aes);
```

```
int wc_AesXtsEncryptSector(
    XtsAes * aes,
    byte * out,
    const byte * in,
    word32 sz,
    word64 sector
)
```

Same process as wc_AesXtsEncrypt but uses a word64 type as the tweak value instead of a byte array. This just converts the word64 to a byte array and calls wc_AesXtsEncrypt.

Parameters:

- **aes** AES keys to use for block encrypt/decrypt
- **out** output buffer to hold cipher text
- **in** input plain text buffer to encrypt
- **sz** size of both out and in buffers
- **sector** value to use for tweak

See:

- wc_AesXtsEncrypt
- wc_AesXtsDecrypt
- wc_AesXtsInit
- wc_AesXtsSetKeyNoInit
- wc_AesXtsSetKey
- wc_AesXtsFree

Return: 0 Success

Example

```
XtsAes aes;
unsigned char plain[SIZE];
unsigned char cipher[SIZE];
word64 s = VALUE;

//set up keys with AES_ENCRYPTION as dir

if(wc_AesXtsEncryptSector(&aes, cipher, plain, SIZE, s) != 0)
{
    // Handle error
}
wc_AesXtsFree(&aes);
```

```
int wc_AesXtsDecryptSector(
    XtsAes * aes,
    byte * out,
    const byte * in,
    word32 sz,
    word64 sector
)
```

Same process as wc_AesXtsDecrypt but uses a word64 type as the tweak value instead of a byte array. This just converts the word64 to a byte array.

Parameters:

- **aes** AES keys to use for block encrypt/decrypt
- **out** output buffer to hold plain text
- **in** input cipher text buffer to decrypt
- **sz** size of both out and in buffers
- **sector** value to use for tweak

See:

- wc_AesXtsEncrypt
- wc_AesXtsDecrypt
- wc_AesXtsInit
- wc_AesXtsSetKeyNoInit
- wc_AesXtsSetKey
- wc_AesXtsFree

Return: 0 Success

Example

```
XtsAes aes;
unsigned char plain[SIZE];
unsigned char cipher[SIZE];
```

```
word64 s = VALUE;

//set up aes key with AES_DECRYPTION as dir and tweak with AES_ENCRYPTION

if(wc_AesXtsDecryptSector(&aes, plain, cipher, SIZE, s) != 0)
{
    // Handle error
}
wc_AesXtsFree(&aes);
```

```
int wc_AesXtsEncrypt(
    XtsAes * aes,
    byte * out,
    const byte * in,
    word32 sz,
    const byte * i,
    word32 iSz
)
```

AES with XTS mode. (XTS) XEX encryption with Tweak and cipher text Stealing.

Parameters:

- **aes** AES keys to use for block encrypt/decrypt
- **out** output buffer to hold cipher text
- **in** input plain text buffer to encrypt
- **sz** size of both out and in buffers
- **i** value to use for tweak
- **iSz** size of i buffer, should always be AES_BLOCK_SIZE but having this input adds a sanity check on how the user calls the function.

See:

- wc_AesXtsDecrypt
- wc_AesXtsInit
- wc_AesXtsSetKeyNoInit
- wc_AesXtsSetKey
- wc_AesXtsFree

Return: 0 Success

Example

```
XtsAes aes;
unsigned char plain[SIZE];
unsigned char cipher[SIZE];
unsigned char i[AES_BLOCK_SIZE];

//set up key with AES_ENCRYPTION as dir

if(wc_AesXtsEncrypt(&aes, cipher, plain, SIZE, i, sizeof(i)) != 0)
```

```
{
    // Handle error
}
wc_AesXtsFree(&aes);
```

```
int wc_AesXtsDecrypt(
    XtsAes * aes,
    byte * out,
    const byte * in,
    word32 sz,
    const byte * i,
    word32 iSz
)
```

Same process as encryption but Aes key is AES_DECRYPTION type.

Parameters:

- **aes** AES keys to use for block encrypt/decrypt
- **out** output buffer to hold plain text
- **in** input cipher text buffer to decrypt
- **sz** size of both out and in buffers
- **i** value to use for tweak
- **iSz** size of i buffer, should always be AES_BLOCK_SIZE but having this input adds a sanity check on how the user calls the function.

See:

- wc_AesXtsEncrypt
- wc_AesXtsInit
- wc_AesXtsSetKeyNoInit
- wc_AesXtsSetKey
- wc_AesXtsFree

Return: 0 Success

Example

```
XtsAes aes;
unsigned char plain[SIZE];
unsigned char cipher[SIZE];
unsigned char i[AES_BLOCK_SIZE];

//set up key with AES_DECRYPTION as dir and tweak with AES_ENCRYPTION

if(wc_AesXtsDecrypt(&aes, plain, cipher, SIZE, i, sizeof(i)) != 0)
{
    // Handle error
}
wc_AesXtsFree(&aes);
```

```
int wc_AesXtsFree(
    XtsAes * aes
)
```

This is to free up any resources used by the XtsAes structure.

Parameters:

- **aes** AES keys to free

See:

- wc_AesXtsEncrypt
- wc_AesXtsDecrypt
- wc_AesXtsInit
- wc_AesXtsSetKeyNoInit
- wc_AesXtsSetKey

Return: 0 Success

Example

```
XtsAes aes;

if(wc_AesXtsSetKey(&aes, key, sizeof(key), AES_ENCRYPTION, NULL, 0) != 0)
{
    // Handle error
}
wc_AesXtsFree(&aes);
```

```
int wc_AesInit(
    Aes * aes,
    void * heap,
    int devId
)
```

Initialize Aes structure. Sets heap hint to be used and ID for use with async hardware. It is up to the user to call wc_AesFree on the Aes structure when done.

Parameters:

- **aes** aes structure in to initialize
- **heap** heap hint to use for malloc / free if needed
- **devId** ID to use with crypto callbacks or async hardware. Set to INVALID_DEVID (-2) if not used

See:

- wc_AesSetKey
- wc_AesSetIV
- wc_AesFree

Return: 0 Success

Example

```
Aes enc;
void* hint = NULL;
int devId = INVALID_DEVID; //if not using async INVALID_DEVID is default

//heap hint could be set here if used

wc_AesInit(&enc, hint, devId);
```

```
int wc_AesFree(
    Aes * aes
)
```

free resources associated with the Aes structure when applicable. Internally may sometimes be a no-op but still recommended to call in all cases as a general best-practice (IE if application code is ported for use on new environments where the call is applicable).

Parameters:

- **aes** aes structure in to free

See: wc_AesInit

Return: no return (void function)

Example

```
Aes enc;
void* hint = NULL;
int devId = INVALID_DEVID; //if not using async INVALID_DEVID is default

//heap hint could be set here if used

wc_AesInit(&enc, hint, devId);
// ... do some interesting things ...
wc_AesFree(&enc);
```

```
int wc_AesCfbEncrypt(
    Aes * aes,
    byte * out,
    const byte * in,
    word32 sz
)
```

AES with CFB mode.

Parameters:

- **aes** AES keys to use for block encrypt/decrypt
- **out** output buffer to hold cipher text must be at least as large as inputbuffer)
- **in** input plain text buffer to encrypt
- **sz** size of input buffer

See:

- wc_AesCfbDecrypt
- wc_AesSetKey

Return: 0 Success and negative error values on failure

Example

```
Aes aes;
unsigned char plain[SIZE];
unsigned char cipher[SIZE];

//set up key with AES_ENCRYPTION as dir for both encrypt and decrypt

if(wc_AesCfbEncrypt(&aes, cipher, plain, SIZE) != 0)
{
    // Handle error
}
```

```
int wc_AesCfbDecrypt(
    Aes * aes,
    byte * out,
    const byte * in,
    word32 sz
)
```

AES with CFB mode.

Parameters:

- **aes** AES keys to use for block encrypt/decrypt
- **out** output buffer to hold decrypted text must be at least as large as inputbuffer)
- **in** input buffer to decrypt
- **sz** size of input buffer

See:

- wc_AesCfbEncrypt
- wc_AesSetKey

Return: 0 Success and negative error values on failure

Example

```
Aes aes;
unsigned char plain[SIZE];
unsigned char cipher[SIZE];

//set up key with AES_ENCRYPTION as dir for both encrypt and decrypt

if(wc_AesCfbDecrypt(&aes, plain, cipher, SIZE) != 0)
{
    // Handle error
}
```

```
int wc_AesSivEncrypt(
    const byte * key,
    word32 keySz,
    const byte * assoc,
    word32 assocSz,
    const byte * nonce,
    word32 nonceSz,
    const byte * in,
    word32 inSz,
    byte * siv,
    byte * out
)
```

This function performs SIV (synthetic initialization vector) encryption as described in RFC 5297.

Parameters:

- **key** Byte buffer containing the key to use.
- **keySz** Length of the key buffer in bytes.
- **assoc** Additional, authenticated associated data (AD).
- **assocSz** Length of AD buffer in bytes.
- **nonce** A number used once. Used by the algorithm in the same manner as the AD.
- **nonceSz** Length of nonce buffer in bytes.
- **in** Plaintext buffer to encrypt.
- **inSz** Length of plaintext buffer.
- **siv** The SIV output by S2V (see RFC 5297 2.4).
- **out** Buffer to hold the ciphertext. Should be the same length as the plaintext buffer.

See: wc_AesSivDecrypt

Return:

- 0 On successful encryption.
- BAD_FUNC_ARG If key, SIV, or output buffer are NULL. Also returned if the key size isn't 32, 48, or 64 bytes.
- Other Other negative error values returned if AES or CMAC operations fail.

Example

```
byte key[] = { some 32, 48, or 64 byte key };
byte assoc[] = {0x01, 0x2, 0x3};
byte nonce[] = {0x04, 0x5, 0x6};
byte plainText[] = {0xDE, 0xAD, 0xBE, 0xEF};
byte siv[AES_BLOCK_SIZE];
byte cipherText[sizeof(plainText)];
if (wc_AesSivEncrypt(key, sizeof(key), assoc, sizeof(assoc), nonce,
    sizeof(nonce), plainText, sizeof(plainText), siv, cipherText) != 0) {
    // failed to encrypt
}
```

```
int wc_AesSivDecrypt(
    const byte * key,
    word32 keySz,
    const byte * assoc,
    word32 assocSz,
    const byte * nonce,
    word32 nonceSz,
    const byte * in,
    word32 inSz,
    byte * siv,
    byte * out
)
```

This function performs SIV (synthetic initialization vector) decryption as described in RFC 5297. If a nonzero error code is returned, the output data is undefined. However, callers must unconditionally zeroize the output buffer to guard against leakage of cleartext data.

Parameters:

- **key** Byte buffer containing the key to use.
- **keySz** Length of the key buffer in bytes.
- **assoc** Additional, authenticated associated data (AD).
- **assocSz** Length of AD buffer in bytes.
- **nonce** A number used once. Used by the underlying algorithm in the same manner as the AD.
- **nonceSz** Length of nonce buffer in bytes.
- **in** Ciphertext buffer to decrypt.
- **inSz** Length of ciphertext buffer.
- **siv** The SIV that accompanies the ciphertext (see RFC 5297 2.4).
- **out** Buffer to hold the decrypted plaintext. Should be the same length as the ciphertext buffer.

See: wc_AesSivEncrypt

Return:

- 0 On successful decryption.
- BAD_FUNC_ARG If key, SIV, or output buffer are NULL. Also returned if the key size isn't 32, 48, or 64 bytes.
- AES_SIV_AUTH_E If the SIV derived by S2V doesn't match the input SIV (see RFC 5297 2.7).
- Other Other negative error values returned if AES or CMAC operations fail.

Example

```
byte key[] = { some 32, 48, or 64 byte key };
byte assoc[] = {0x01, 0x2, 0x3};
byte nonce[] = {0x04, 0x5, 0x6};
byte cipherText[] = {0xDE, 0xAD, 0xBE, 0xEF};
byte siv[AES_BLOCK_SIZE] = { the SIV that came with the ciphertext };
byte plainText[sizeof(cipherText)];
if (wc_AesSivDecrypt(key, sizeof(key), assoc, sizeof(assoc), nonce,
    sizeof(nonce), cipherText, sizeof(cipherText), siv, plainText) != 0) {
    // failed to decrypt
}
```

```
WOLFSSL_API int wc_AesEaxEncryptAuth(
    const byte * key,
    word32 keySz,
    byte * out,
    const byte * in,
    word32 inSz,
    const byte * nonce,
    word32 nonceSz,
    byte * authTag,
    word32 authTagSz,
    const byte * authIn,
    word32 authInSz
)
```

This function performs AES EAX encryption and authentication as described in “EAX: A Conventional Authenticated-Encryption Mode” (https://eprint.iacr.org/2003/069). It is a “one-shot” API that performs all encryption and authentication operations in one function call.

Parameters:

- **key** buffer containing the key to use
- **keySz** length of the key buffer in bytes
- **out** buffer to hold the ciphertext. Should be the same length as the plaintext buffer
- **in** plaintext buffer to encrypt
- **inSz** length of plaintext buffer
- **nonce** the cryptographic nonce to use for EAX operations
- **nonceSz** length of nonce buffer in bytes
- **authTag** pointer to the buffer in which to store the authentication tag
- **authTagSz** length of the desired authentication tag
- **authIn** pointer to the buffer containing input data to authenticate
- **authInSz** length of the input authentication data

See: wc_AesEaxDecryptAuth

Return:

- 0 on successful encryption.
- BAD_FUNC_ARG if input or output buffers are NULL. Also returned if the key size isn’t a valid AES key size (16, 24, or 32 bytes)

- other negative error values returned if AES or CMAC operations fail.

Example

```
byte key[] = { some 32, 48, or 64 byte key };
byte nonce[] = {0x04, 0x5, 0x6};
byte plainText[] = {0xDE, 0xAD, 0xBE, 0xEF};
byte authIn[] = {0x01, 0x2, 0x3};

byte cipherText[sizeof(plainText)]; // output ciphertext
byte authTag[length, up to AES_BLOCK_SIZE]; // output authTag

if (wc_AesEaxEncrypt(key, sizeof(key),
                     cipherText, plainText, sizeof(plainText),
                     nonce, sizeof(nonce),
                     authTag, sizeof(authTag),
                     authIn, sizeof(authIn)) != 0) {
    // failed to encrypt
}
```

```
WOLFSSL_API int wc_AesEaxDecryptAuth(
    const byte * key,
    word32 keySz,
    byte * out,
    const byte * in,
    word32 inSz,
    const byte * nonce,
    word32 nonceSz,
    const byte * authTag,
    word32 authTagSz,
    const byte * authIn,
    word32 authInSz
)
```

This function performs AES EAX decryption and authentication as described in "EAX: A Conventional Authenticated-Encryption Mode" (https://eprint.iacr.org/2003/069). It is a "one-shot" API that performs all decryption and authentication operations in one function call. If a nonzero error code is returned, the output data is undefined. However, callers must unconditionally zeroize the output buffer to guard against leakage of cleartext data.

Parameters:

- **key** byte buffer containing the key to use
- **keySz** length of the key buffer in bytes
- **out** buffer to hold the plaintext. Should be the same length as the input ciphertext buffer
- **in** ciphertext buffer to decrypt
- **inSz** length of ciphertext buffer
- **nonce** the cryptographic nonce to use for EAX operations
- **nonceSz** length of nonce buffer in bytes
- **authTag** buffer that holds the authentication tag to check the authenticity of the data against
- **authTagSz** Length of the input authentication tag

- **authIn** pointer to the buffer containing input data to authenticate
- **authInSz** length of the input authentication data

See: wc_AesEaxEncryptAuth

Return:

- 0 on successful decryption
- BAD_FUNC_ARG if input or output buffers are NULL. Also returned if the key size isn't a valid AES key size (16, 24, or 32 bytes)
- AES_EAX_AUTH_E If the authentication tag does not match the supplied authentication code vector authTag
- other negative error values returned if AES or CMAC operations fail.

Example

```
byte key[] = { some 32, 48, or 64 byte key };
byte nonce[] = {0x04, 0x5, 0x6};
byte cipherText[] = {0xDE, 0xAD, 0xBE, 0xEF};
byte authIn[] = {0x01, 0x2, 0x3};

byte plainText[sizeof(cipherText)]; // output plaintext
byte authTag[length, up to AES_BLOCK_SIZE]; // output authTag

if (wc_AesEaxDecrypt(key, sizeof(key),
                     cipherText, plainText, sizeof(plainText),
                     nonce, sizeof(nonce),
                     authTag, sizeof(authTag),
                     authIn, sizeof(authIn)) != 0) {
    // failed to encrypt
}
```

```
WOLFSSL_API int wc_AesEaxInit(
    AesEax * eax,
    const byte * key,
    word32 keySz,
    const byte * nonce,
    word32 nonceSz,
    const byte * authIn,
    word32 authInSz
)
```

This function initializes an AesEax object for use in authenticated encryption or decryption. This function must be called on an AesEax object before using it with any of the AES EAX incremental API functions. It does not need to be called if using the one-shot EAX API functions. All AesEax instances initialized with this function need to be freed with a call to wc_AesEaxFree() when done using the instance.

Parameters:

- **eax** AES EAX structure holding the context of the AEAD operation

- **key** 16, 24, or 32 byte secret key for encryption and decryption
- **keySz** length of the supplied key in bytes
- **nonce** the cryptographic nonce to use for EAX operations
- **nonceSz** length of nonce buffer in bytes
- **authIn** (optional) input data to add to the authentication stream This argument should be NULL if not used
- **authInSz** size in bytes of the input authentication data

See:

- wc_AesEaxEncryptUpdate
- wc_AesEaxDecryptUpdate
- wc_AesEaxAuthDataUpdate
- wc_AesEaxEncryptFinal
- wc_AesEaxDecryptFinal
- wc_AesEaxFree

Return:

- 0 on success
- error code on failure

Example

```
AesEax eax;
key[]   = { some 16, 24, or 32 byte length key };
nonce[] = { some arbitrary length nonce };
authIn[] = { some data to add to the authentication stream };
plainText[] = {some plaintext data to encrypt};

cipherText[sizeof(plainText)]; // buffer to hold cipherText
authTag[length, up to AES_BLOCK_SIZE]; // buffer to hold computed auth data

AesEax eax;

if ((ret = wc_AesEaxInit(eax,
                         key, keySz,
                         nonce, nonceSz,
                         authIn, authInSz)) != 0) {
    goto cleanup;
}

// if we wanted to add more auth data, we could provide it at this point,
// otherwise we use NULL for the authIn parameter, with authIn size of 0
if ((ret = wc_AesEaxEncryptUpdate(eax,
                                  cipherText, plainText, sizeof(plainText),
                                  NULL, 0)) != 0) {
    goto cleanup;
}

if ((ret = wc_AesEaxEncryptFinal(eax, authTag, sizeof(authTag))) != 0) {
    goto cleanup;
}
```

```
cleanup:
    wc_AesEaxFree(eax);



WOLFSSL_API int wc_AesEaxEncryptUpdate(
    AesEax * eax,
    byte * out,
    const byte * in,
    word32 inSz,
    const byte * authIn,
    word32 authInSz
)
```

This function uses AES EAX to encrypt input data, and optionally, add more input data to the authentication stream. eax must have been previously initialized with a call to wc_AesEaxInit.

Parameters:

- **eax** AES EAX structure holding the context of the AEAD operation
- **out** output buffer holding the ciphertext
- **in** input buffer holding the plaintext to encrypt
- **inSz** size in bytes of the input data buffer
- **authIn** (optional) input data to add to the authentication stream This argument should be NULL if not used
- **authInSz** size in bytes of the input authentication data

See:

- wc_AesEaxInit
- wc_AesEaxDecryptUpdate
- wc_AesEaxAuthDataUpdate
- wc_AesEaxEncryptFinal
- wc_AesEaxDecryptFinal
- wc_AesEaxFree

Return:

- 0 on success
- error code on failure

Example

```
AesEax eax;
key[]   = { some 16, 24, or 32 byte length key };
nonce[] = { some arbitrary length nonce };
authIn[] = { some data to add to the authentication stream };
plainText[] = {some plaintext data to encrypt};

cipherText[sizeof(plainText)]; // buffer to hold cipherText
authTag[length, up to AES_BLOCK_SIZE]; // buffer to hold computed auth data
```

```
AesEax eax;

if ((ret = wc_AesEaxInit(eax,
                         key, keySz,
                         nonce, nonceSz,
                         authIn, authInSz)) != 0) {
    goto cleanup;
}

// if we wanted to add more auth data, we could provide it at this point,
// otherwise we use NULL for the authIn parameter, with authInSz of 0
if ((ret = wc_AesEaxEncryptUpdate(eax,
                                  cipherText, plainText, sizeof(plainText),
                                  NULL, 0)) != 0) {
    goto cleanup;
}

if ((ret = wc_AesEaxEncryptFinal(eax, authTag, sizeof(authTag))) != 0) {
    goto cleanup;
}

cleanup:
    wc_AesEaxFree(eax);



WOLFSSL_API int wc_AesEaxDecryptUpdate(
    AesEax * eax,
    byte * out,
    const byte * in,
    word32 inSz,
    const byte * authIn,
    word32 authInSz
)
```

This function uses AES EAX to decrypt input data, and optionally, add more input data to the authentication stream. eax must have been previously initialized with a call to wc_AesEaxInit.

Parameters:

- **eax** AES EAX structure holding the context of the AEAD operation
- **out** output buffer holding the decrypted plaintext
- **in** input buffer holding the ciphertext
- **inSz** size in bytes of the input data buffer
- **authIn** (optional) input data to add to the authentication stream This argument should be NULL if not used
- **authInSz** size in bytes of the input authentication data

See:

- wc_AesEaxInit
- wc_AesEaxEncryptUpdate

- wc_AesEaxAuthDataUpdate
- wc_AesEaxEncryptFinal
- wc_AesEaxDecryptFinal
- wc_AesEaxFree

Return:

- 0 on success
- error code on failure

Example

```
AesEax eax;
key[]   = { some 16, 24, or 32 byte length key };
nonce[] = { some arbitrary length nonce };
authIn[] = { some data to add to the authentication stream };
cipherText[] = {some encrypted data};

plainText[sizeof(cipherText)]; // buffer to hold decrypted data
// auth tag is generated elsewhere by the encrypt AEAD operation
authTag[length, up to AES_BLOCK_SIZE] = { the auth tag };

AesEax eax;

if ((ret = wc_AesEaxInit(eax,
                         key, keySz,
                         nonce, nonceSz,
                         authIn, authInSz)) != 0) {
    goto cleanup;
}

// if we wanted to add more auth data, we could provide it at this point,
// otherwise we use NULL for the authIn parameter, with authInSz of 0
if ((ret = wc_AesEaxDecryptUpdate(eax,
                                  plainText, cipherText, sizeof(cipherText),
                                  NULL, 0)) != 0) {
    goto cleanup;
}

if ((ret = wc_AesEaxDecryptFinal(eax, authTag, sizeof(authTag))) != 0) {
    goto cleanup;
}

cleanup:
    wc_AesEaxFree(eax);
```

```
WOLFSSL_API int wc_AesEaxAuthDataUpdate(
    AesEax * eax,
    const byte * authIn,
    word32 authInSz
)
```

This function adds input data to the authentication stream. eax must have been previously initialized with a call to wc_AesEaxInit.

Parameters:

- **eax** AES EAX structure holding the context of the AEAD operation
- **authIn** input data to add to the authentication stream
- **authInSz** size in bytes of the input authentication data

See:

- wc_AesEaxInit
- wc_AesEaxEncryptUpdate
- wc_AesEaxDecryptUpdate
- wc_AesEaxEncryptFinal
- wc_AesEaxDecryptFinal
- wc_AesEaxFree

Return:

- 0 on success
- error code on failure

Example

```
AesEax eax;
key[]   = { some 16, 24, or 32 byte length key };
nonce[] = { some arbitrary length nonce };
authIn[] = { some data to add to the authentication stream };
cipherText[] = {some encrypted data};

plainText[sizeof(cipherText)]; // buffer to hold decrypted data
// auth tag is generated elsewhere by the encrypt AEAD operation
authTag[length, up to AES_BLOCK_SIZE] = { the auth tag };

AesEax eax;

// No auth data to add here
if ((ret = wc_AesEaxInit(eax,
                         key, keySz,
                         nonce, nonceSz,
                         NULL, 0)) != 0) {
    goto cleanup;
}

// No auth data to add here, added later with wc_AesEaxAuthDataUpdate
if ((ret = wc_AesEaxDecryptUpdate(eax,
                                  plainText, cipherText, sizeof(cipherText),
                                  NULL, 0)) != 0) {
    goto cleanup;
}
```

```
if ((ret = wc_AesEaxAuthDataUpdate(eax, authIn, sizeof(authIn))) != 0) {
    goto cleanup;
}

if ((ret = wc_AesEaxDecryptFinal(eax, authTag, sizeof(authTag))) != 0) {
    goto cleanup;
}

cleanup:
    wc_AesEaxFree(eax);
```

```
WOLFSSL_API int wc_AesEaxEncryptFinal(
    AesEax * eax,
    byte * authTag,
    word32 authTagSz
)
```

This function finalizes the encrypt AEAD operation, producing an auth tag over the current authentication stream. eax must have been previously initialized with a call to wc_AesEaxInit. When done using the AesEax context structure, make sure to free it using wc_AesEaxFree.

Parameters:

- **eax** AES EAX structure holding the context of the AEAD operation
- **authTag[out]** buffer that will hold the computed auth tag
- **authTagSz** size in bytes of authTag

See:

- wc_AesEaxInit
- wc_AesEaxEncryptUpdate
- wc_AesEaxDecryptUpdate
- wc_AesEaxAuthDataUpdate
- wc_AesEaxDecryptFinal
- wc_AesEaxFree

Return:

- 0 on success
- error code on failure

Example

```
AesEax eax;
key[]   = { some 16, 24, or 32 byte length key };
nonce[] = { some arbitrary length nonce };
authIn[] = { some data to add to the authentication stream };
plainText[] = {some plaintext data to encrypt};

cipherText[sizeof(plainText)]; // buffer to hold cipherText
```

```
authTag[length, up to AES_BLOCK_SIZE]; // buffer to hold computed auth data

AesEax eax;

if ((ret = wc_AesEaxInit(eax,
                         key, keySz,
                         nonce, nonceSz,
                         authIn, authInSz)) != 0) {
    goto cleanup;
}

// if we wanted to add more auth data, we could provide it at this point,
// otherwise we use NULL for the authIn parameter, with authInSz of 0
if ((ret = wc_AesEaxEncryptUpdate(eax,
                                  cipherText, plainText, sizeof(plainText),
                                  NULL, 0)) != 0) {
    goto cleanup;
}

if ((ret = wc_AesEaxEncryptFinal(eax, authTag, sizeof(authTag))) != 0) {
    goto cleanup;
}

cleanup:
    wc_AesEaxFree(eax);



WOLFSSL_API int wc_AesEaxDecryptFinal(
    AesEax * eax,
    const byte * authIn,
    word32 authInSz
)
```

This function finalizes the decrypt AEAD operation, finalizing the auth tag computation and checking it for validity against the user supplied tag. eax must have been previously initialized with a call to wc_AesEaxInit. When done using the AesEax context structure, make sure to free it using wc_AesEaxFree.

Parameters:

- **eax** AES EAX structure holding the context of the AEAD operation
- **authIn** input auth tag to check computed auth tag against
- **authInSz** size in bytes of authIn

See:

- wc_AesEaxInit
- wc_AesEaxEncryptUpdate
- wc_AesEaxDecryptUpdate
- wc_AesEaxAuthDataUpdate
- wc_AesEaxEncryptFinal
- wc_AesEaxFree

Return:

- 0 if data is authenticated successfully
- AES_EAX_AUTH_E if the authentication tag does not match the supplied authentication code vector authIn
- other error code on failure

Example

```
AesEax eax;
key[]   = { some 16, 24, or 32 byte length key };
nonce[] = { some arbitrary length nonce };
authIn[] = { some data to add to the authentication stream };
cipherText[] = {some encrypted data};

plainText[sizeof(cipherText)]; // buffer to hold decrypted data
// auth tag is generated elsewhere by the encrypt AEAD operation
authTag[length, up to AES_BLOCK_SIZE] = { the auth tag };

AesEax eax;

if ((ret = wc_AesEaxInit(eax,
                         key, keySz,
                         nonce, nonceSz,
                         authIn, authInSz)) != 0) {
    goto cleanup;
}

// if we wanted to add more auth data, we could provide it at this point,
// otherwise we use NULL for the authIn parameter, with authInSz of 0
if ((ret = wc_AesEaxDecryptUpdate(eax,
                                  plainText, cipherText, sizeof(cipherText),
                                  NULL, 0)) != 0) {
    goto cleanup;
}

if ((ret = wc_AesEaxDecryptFinal(eax, authTag, sizeof(authTag))) != 0) {
    goto cleanup;
}

cleanup:
    wc_AesEaxFree(eax);



WOLFSSL_API int wc_AesEaxFree(
    AesEax * eax
)
```

This frees up any resources, specifically keys, used by the Aes instance inside the AesEax wrapper struct. It should be called on the AesEax struct after it has been initialized with wc_AesEaxInit, and all desired EAX operations are complete.

Parameters:

- **eaxAES** EAX instance to free

See:

- wc_AesEaxInit
- wc_AesEaxEncryptUpdate
- wc_AesEaxDecryptUpdate
- wc_AesEaxAuthDataUpdate
- wc_AesEaxEncryptFinal
- wc_AesEaxDecryptFinal

Return: 0 Success

Example

```
AesEax eax;

if(wc_AesEaxInit(eax, key, keySz, nonce, nonceSz, authIn, authInSz) != 0) {
    // handle errors, then free
    wc_AesEaxFree(&eax);
}
```

```
int wc_AesCtsEncrypt(
    const byte * key,
    word32 keySz,
    byte * out,
    const byte * in,
    word32 inSz,
    const byte * iv
)
```

This function performs AES encryption using Ciphertext Stealing (CTS) mode. It is a one-shot API that handles all operations in a single call.

Parameters:

- **key** pointer to the AES key used for encryption.
- **keySz** size of the AES key in bytes (16, 24, or 32 bytes).
- **out** buffer to hold the encrypted ciphertext. Must be at least the size of the input.
- **in** pointer to the plaintext input data to encrypt.
- **inSz** size of the plaintext input data in bytes.
- **iv** pointer to the initialization vector (IV) used for encryption. Must be 16 bytes.
- **key** pointer to the AES key used for encryption.
- **keySz** size of the AES key in bytes (16, 24, or 32 bytes).
- **out** buffer to hold the encrypted ciphertext. Must be at least the same size as the input plaintext.
- **in** pointer to the plaintext input data to encrypt.
- **inSz** size of the plaintext input data in bytes.
- **iv** pointer to the initialization vector (IV) used for encryption. Must be 16 bytes. *Example*

```
byte key[16] = { 0 };
byte iv[16] = { 0 };
byte plaintext[] = { 0x01, 0x02, 0x03, 0x04, 0x05 };
byte ciphertext[sizeof(plaintext)];
int ret = wc_AesCtsEncrypt(key, sizeof(key), ciphertext, plaintext,
                           sizeof(plaintext), iv);
if (ret != 0) {
    // handle encryption error
}
```

See:

- wc_AesCtsDecrypt
- wc_AesCtsDecrypt

Return:

- 0 on successful encryption.
- BAD_FUNC_ARG if input arguments are invalid.
- other negative error codes for encryption failures.
- 0 on successful encryption.
- BAD_FUNC_ARG if input arguments are invalid.
- other negative error codes for encryption failures.

Example

```
    byte key[16] = { 0 };
    byte iv[16] = { 0 };
    byte plaintext[] = { 0x01, 0x02, 0x03, 0x04, 0x05 };
    byte ciphertext[sizeof(plaintext)];

    int ret = wc_AesCtsEncrypt(key, sizeof(key), ciphertext, plaintext,
        sizeof(plaintext), iv);
    if (ret != 0) {
    // handle encryption error
}
```

```
int wc_AesCtsDecrypt(
    const byte * key,
    word32 keySz,
    byte * out,
    const byte * in,
    word32 inSz,
    const byte * iv
)
```

This function performs AES decryption using Ciphertext Stealing (CTS) mode. It is a one-shot API that handles all operations in a single call.

Parameters:

- **key** pointer to the AES key used for decryption.
- **keySz** size of the AES key in bytes (16, 24, or 32 bytes).
- **out** buffer to hold the decrypted plaintext. Must be at least the same size as the input ciphertext.
- **in** pointer to the ciphertext input data to decrypt.
- **inSz** size of the ciphertext input data in bytes.
- **iv** pointer to the initialization vector (IV) used for decryption. Must be 16 bytes. *Example*

```
byte key[16] = { 0 };
byte iv[16] = { 0 };
byte ciphertext[] = { 0x01, 0x02, 0x03, 0x04, 0x05 };
byte plaintext[sizeof(ciphertext)];
int ret = wc_AesCtsDecrypt(key, sizeof(key), plaintext, ciphertext,
                           sizeof(ciphertext), iv);
if (ret != 0) {
    // handle decryption error
}
```

See: wc_AesCtsEncrypt

Return:

- 0 on successful decryption.
- BAD_FUNC_ARG if input arguments are invalid.
- other negative error codes for decryption failures.

```
int wc_AesCtsEncryptUpdate(
    Aes * aes,
    byte * out,
    word32 * outSz,
    const byte * in,
    word32 inSz
)
```

This function performs an update step of the AES CTS encryption. It processes a chunk of plaintext and stores intermediate data.

Parameters:

- **aes** pointer to the Aes structure holding the context of the operation.
- **out** buffer to hold the encrypted ciphertext. Must be large enough to store the output from this update step.
- **outSz** size in bytes of the output data written to the `out` buffer. On input, it should contain the maximum number of bytes that can be written to the `out` buffer.
- **in** pointer to the plaintext input data to encrypt.
- **inSz** size of the plaintext input data in bytes. *Example*

```
Aes aes;
wc_AesInit(&aes, NULL, INVALID_DEVID);
```

```
byte key[16] = { 0 };
byte iv[16] = { 0 };
byte plaintext[] = { ... };
byte ciphertext[sizeof(plaintext)];
word32 outSz = sizeof(ciphertext);
wc_AesSetKey(&aes, key, sizeof(key), iv, AES_ENCRYPTION);
int ret = wc_AesCtsEncryptUpdate(&aes, ciphertext, &outSz, plaintext,
 ↪  sizeof(plaintext));
if (ret != 0) {
    // handle error
}
wc_AesFree(&aes);
```

See: wc_AesCtsDecryptUpdate

Return:

- 0 on successful processing.
- BAD_FUNC_ARG if input arguments are invalid.

```
int wc_AesCtsEncryptFinal(
    Aes * aes,
    byte * out,
    word32 * outSz
)
```

This function finalizes the AES CTS encryption operation. It processes any remaining plaintext and completes the encryption.

Parameters:

- **aes** pointer to the Aes structure holding the context of the operation.
- **out** buffer to hold the final encrypted ciphertext. Must be large enough to store any remaining ciphertext from this final step.
- **outSz** size in bytes of the output data written to the out buffer. On input, it should contain the maximum number of bytes that can be written to the out buffer. *Example*

```
Aes aes;
wc_AesInit(&aes, NULL, INVALID_DEVID);
byte key[16] = { 0 };
byte iv[16] = { 0 };
byte plaintext[] = { ... };
byte ciphertext[sizeof(plaintext)];
word32 outSz = sizeof(ciphertext);
wc_AesSetKey(&aes, key, sizeof(key), iv, AES_ENCRYPTION);
// Perform any required update steps using wc_AesCtsEncryptUpdate
int ret = wc_AesCtsEncryptFinal(&aes, ciphertext, &outSz);
if (ret != 0) {
    // handle error
}
wc_AesFree(&aes);
```

See: wc_AesCtsDecryptFinal

Return:

- 0 on successful encryption completion.
- BAD_FUNC_ARG if input arguments are invalid.

```
int wc_AesCtsDecryptUpdate(
    Aes * aes,
    byte * out,
    word32 * outSz,
    const byte * in,
    word32 inSz
)
```

This function performs an update step of the AES CTS decryption. It processes a chunk of ciphertext and stores intermediate data.

Parameters:

- **aes** pointer to the Aes structure holding the context of the operation.
- **out** buffer to hold the decrypted plaintext. Must be large enough to store the output from this update step.
- **outSz** size in bytes of the output data written to the out buffer. On input, it should contain the maximum number of bytes that can be written to the out buffer.
- **in** pointer to the ciphertext input data to decrypt.
- **inSz** size of the ciphertext input data in bytes. *Example*

```
Aes aes;
wc_AesInit(&aes, NULL, INVALID_DEVID);
byte key[16] = { 0 };
byte iv[16] = { 0 };
byte ciphertext[] = { ... };
byte plaintext[sizeof(ciphertext)];
word32 outSz = sizeof(plaintext);
wc_AesSetKey(&aes, key, sizeof(key), iv, AES_DECRYPTION);
int ret = wc_AesCtsDecryptUpdate(&aes, plaintext, &outSz, ciphertext,
 ↪  sizeof(ciphertext));
if (ret != 0) {
    // handle error
}
wc_AesFree(&aes);
```

See: wc_AesCtsEncryptUpdate

Return:

- 0 on successful processing.
- BAD_FUNC_ARG if input arguments are invalid.

```
int wc_AesCtsDecryptFinal(
    Aes * aes,
    byte * out,
    word32 * outSz
)
```

This function finalizes the AES CTS decryption operation. It processes any remaining ciphertext and completes the decryption.

Parameters:

- **aes** pointer to the Aes structure holding the context of the operation.
- **out** buffer to hold the final decrypted plaintext. Must be large enough to store any remaining plaintext from this final step.
- **outSz** size in bytes of the output data written to the out buffer. On input, it should contain the maximum number of bytes that can be written to the out buffer. *Example*

```
Aes aes;
wc_AesInit(&aes, NULL, INVALID_DEVID);
byte key[16] = { 0 };
byte iv[16] = { 0 };
byte ciphertext[] = { ... };
byte plaintext[sizeof(ciphertext)];
word32 outSz = sizeof(plaintext);
wc_AesSetKey(&aes, key, sizeof(key), iv, AES_DECRYPTION);
// Perform any required update steps using wc_AesCtsDecryptUpdate
int ret = wc_AesCtsDecryptFinal(&aes, plaintext, &outSz);
if (ret != 0) {
    // handle error
}
wc_AesFree(&aes);
```

See: wc_AesCtsEncryptFinal

Return:

- 0 on successful decryption completion.
- BAD_FUNC_ARG if input arguments are invalid.

```
int wc_AesCbcDecryptWithKey(
    byte * out,
    const byte * in,
    word32 inSz,
    const byte * key,
    word32 keySz,
    const byte * iv
)
```

Decrypts a cipher from the input buffer in, and places the resulting plain text in the output buffer out using cipher block chaining with AES. This function does not require an AES structure to be initialized.

Instead, it takes in a key and an iv (initialization vector) and uses these to initialize an AES object and then decrypt the cipher text.

Parameters:

- **out** pointer to the output buffer in which to store the plain text of the decrypted message
- **in** pointer to the input buffer containing cipher text to be decrypted
- **inSz** size of input message
- **key** 16, 24, or 32 byte secret key for decryption
- **keySz** size of key used for decryption

See:

- wc_AesSetKey
- wc_AesSetIV
- wc_AesCbcEncrypt
- wc_AesCbcDecrypt

Return:

- 0 On successfully decrypting message
- BAD_ALIGN_E Returned on block align error
- BAD_FUNC_ARG Returned if key length is invalid or AES object is null during AesSetIV
- MEMORY_E Returned if WOLFSSL_SMALL_STACK is enabled and XMALLOC fails to instantiate an AES object.

Example

```
int ret = 0;
byte key[] = { some 16, 24, or 32 byte key };
byte iv[]  = { some 16 byte iv };
byte cipher[AES_BLOCK_SIZE * n]; //n being a positive integer making
cipher some multiple of 16 bytes
// fill cipher with cipher text
byte plain [AES_BLOCK_SIZE * n];
if ((ret = wc_AesCbcDecryptWithKey(plain, cipher, AES_BLOCK_SIZE, key,
AES_BLOCK_SIZE, iv)) != 0 ) {
// Decrypt Error
}
```

B.14 Algorithms - ARC4

B.13.2.46 function wc_AesCbcDecryptWithKey

B.14.1 Functions

	Name
int	**wc_Arc4Process**(Arc4 * arc4, byte * out, const byte * in, word32 length)This function encrypts an input message from the buffer in, placing the ciphertext in the output buffer out, or decrypts a ciphertext from the buffer in, placing the plaintext in the output buffer out, using ARC4 encryption. This function is used for both encryption and decryption. Before this method may be called, one must first initialize the ARC4 structure using wc_Arc4SetKey.
int	**wc_Arc4SetKey**(Arc4 * arc4, const byte * key, word32 length)This function sets the key for a ARC4 object, initializing it for use as a cipher. It should be called before using it for encryption with wc_Arc4Process.

B.14.2 Functions Documentation

```
int wc_Arc4Process(
    Arc4 * arc4,
    byte * out,
    const byte * in,
    word32 length
)
```

This function encrypts an input message from the buffer in, placing the ciphertext in the output buffer out, or decrypts a ciphertext from the buffer in, placing the plaintext in the output buffer out, using ARC4 encryption. This function is used for both encryption and decryption. Before this method may be called, one must first initialize the ARC4 structure using wc_Arc4SetKey.

Parameters:

- **arc4** pointer to the ARC4 structure used to process the message
- **out** pointer to the output buffer in which to store the processed message
- **in** pointer to the input buffer containing the message to process
- **length** length of the message to process

See: wc_Arc4SetKey

Return: none

Example

```
Arc4 enc;
byte key[] = { key to use for encryption };
wc_Arc4SetKey(&enc, key, sizeof(key));

byte plain[] = { plain text to encode };
byte cipher[sizeof(plain)];
byte decrypted[sizeof(plain)];
// encrypt the plain into cipher
wc_Arc4Process(&enc, cipher, plain, sizeof(plain));
```

```
// decrypt the cipher
wc_Arc4Process(&enc, decrypted, cipher, sizeof(cipher));
```

```
int wc_Arc4SetKey(
    Arc4 * arc4,
    const byte * key,
    word32 length
)
```

This function sets the key for a ARC4 object, initializing it for use as a cipher. It should be called before using it for encryption with wc_Arc4Process.

Parameters:

- **arc4** pointer to an arc4 structure to be used for encryption
- **key** key with which to initialize the arc4 structure
- **length** length of the key used to initialize the arc4 structure

See: wc_Arc4Process

Return: none

Example

```
Arc4 enc;
byte key[] = { initialize with key to use for encryption };
wc_Arc4SetKey(&enc, key, sizeof(key));
```

B.15 Algorithms - BLAKE2

B.14.2.2 function wc_Arc4SetKey

B.15.1 Functions

	Name
int	**wc_InitBlake2b**(Blake2b * b2b, word32 digestSz)This function initializes a Blake2b structure for use with the Blake2 hash function.
int	**wc_Blake2bUpdate**(Blake2b * b2b, const byte * data, word32 sz)This function updates the Blake2b hash with the given input data. This function should be called after wc_InitBlake2b, and repeated until one is ready for the final hash: wc_Blake2bFinal.

	Name
int	**wc_Blake2bFinal**(Blake2b * b2b, byte * final, word32 requestSz)This function computes the Blake2b hash of the previously supplied input data. The output hash will be of length requestSz, or, if requestSz==0, the digestSz of the b2b structure. This function should be called after wc_InitBlake2b and wc_Blake2bUpdate has been processed for each piece of input data desired.

B.15.2 Functions Documentation

```
int wc_InitBlake2b(
    Blake2b * b2b,
    word32 digestSz
)
```

This function initializes a Blake2b structure for use with the Blake2 hash function.

Parameters:

- **b2b** pointer to the Blake2b structure to initialize
- **digestSz** length of the blake 2 digest to implement

See: wc_Blake2bUpdate

Return: 0 Returned upon successfully initializing the Blake2b structure and setting the digest size.

Example

```
Blake2b b2b;
// initialize Blake2b structure with 64 byte digest
wc_InitBlake2b(&b2b, 64);
```

```
int wc_Blake2bUpdate(
    Blake2b * b2b,
    const byte * data,
    word32 sz
)
```

This function updates the Blake2b hash with the given input data. This function should be called after wc_InitBlake2b, and repeated until one is ready for the final hash: wc_Blake2bFinal.

Parameters:

- **b2b** pointer to the Blake2b structure to update
- **data** pointer to a buffer containing the data to append
- **sz** length of the input data to append

See:

- wc_InitBlake2b
- wc_Blake2bFinal

Return:

- 0 Returned upon successfully update the Blake2b structure with the given data
- -1 Returned if there is a failure while compressing the input data

Example

```
int ret;
Blake2b b2b;
// initialize Blake2b structure with 64 byte digest
wc_InitBlake2b(&b2b, 64);

byte plain[] = { // initialize input };

ret = wc_Blake2bUpdate(&b2b, plain, sizeof(plain));
if( ret != 0) {
    // error updating blake2b
}
```

```
int wc_Blake2bFinal(
    Blake2b * b2b,
    byte * final,
    word32 requestSz
)
```

This function computes the Blake2b hash of the previously supplied input data. The output hash will be of length requestSz, or, if requestSz==0, the digestSz of the b2b structure. This function should be called after wc_InitBlake2b and wc_Blake2bUpdate has been processed for each piece of input data desired.

Parameters:

- **b2b** pointer to the Blake2b structure to update
- **final** pointer to a buffer in which to store the blake2b hash. Should be of length requestSz
- **requestSz** length of the digest to compute. When this is zero, b2b->digestSz will be used instead

See:

- wc_InitBlake2b
- wc_Blake2bUpdate

Return:

- 0 Returned upon successfully computing the Blake2b hash

- -1 Returned if there is a failure while parsing the Blake2b hash

Example

```
int ret;
Blake2b b2b;
byte hash[64];
// initialize Blake2b structure with 64 byte digest
wc_InitBlake2b(&b2b, 64);
... // call wc_Blake2bUpdate to add data to hash

ret = wc_Blake2bFinal(&b2b, hash, 64);
if( ret != 0) {
    // error generating blake2b hash
}
```

B.16 Algorithms - Camellia

B.15.2.3 function wc_Blake2bFinal

B.16.1 Functions

	Name
int	**wc_CamelliaSetKey**(Camellia * cam, const byte * key, word32 len, const byte * iv)This function sets the key and initialization vector for a camellia object, initializing it for use as a cipher.
int	**wc_CamelliaSetIV**(Camellia * cam, const byte * iv)This function sets the initialization vector for a camellia object.
int	**wc_CamelliaEncryptDirect**(Camellia * cam, byte * out, const byte * in)This function does a one-block encrypt using the provided camellia object. It parses the first 16 byte block from the buffer in and stores the encrypted result in the buffer out. Before using this function, one should initialize the camellia object using wc_CamelliaSetKey.
int	**wc_CamelliaDecryptDirect**(Camellia * cam, byte * out, const byte * in)This function does a one-block decrypt using the provided camellia object. It parses the first 16 byte block from the buffer in, decrypts it, and stores the result in the buffer out. Before using this function, one should initialize the camellia object using wc_CamelliaSetKey.

	Name
int	**wc_CamelliaCbcEncrypt**(Camellia * cam, byte * out, const byte * in, word32 sz)This function encrypts the plaintext from the buffer in and stores the output in the buffer out. It performs this encryption using Camellia with Cipher Block Chaining (CBC).
int	**wc_CamelliaCbcDecrypt**(Camellia * cam, byte * out, const byte * in, word32 sz)This function decrypts the ciphertext from the buffer in and stores the output in the buffer out. It performs this decryption using Camellia with Cipher Block Chaining (CBC).

B.16.2 Functions Documentation

```
int wc_CamelliaSetKey(
    Camellia * cam,
    const byte * key,
    word32 len,
    const byte * iv
)
```

This function sets the key and initialization vector for a camellia object, initializing it for use as a cipher.

Parameters:

- **cam** pointer to the camellia structure on which to set the key and iv
- **key** pointer to the buffer containing the 16, 24, or 32 byte key to use for encryption and decryption
- **len** length of the key passed in
- **iv** pointer to the buffer containing the 16 byte initialization vector for use with this camellia structure

See:

- wc_CamelliaEncryptDirect
- wc_CamelliaDecryptDirect
- wc_CamelliaCbcEncrypt
- wc_CamelliaCbcDecrypt

Return:

- 0 Returned upon successfully setting the key and initialization vector
- BAD_FUNC_ARG returned if there is an error processing one of the input arguments
- MEMORY_E returned if there is an error allocating memory with XMALLOC

Example

```
Camellia cam;
byte key[32];
```

```
// initialize key
byte iv[16];
// initialize iv
if( wc_CamelliaSetKey(&cam, key, sizeof(key), iv) != 0) {
    // error initializing camellia structure
}
```

```
int wc_CamelliaSetIV(
    Camellia * cam,
    const byte * iv
)
```

This function sets the initialization vector for a camellia object.

Parameters:

- **cam** pointer to the camellia structure on which to set the iv
- **iv** pointer to the buffer containing the 16 byte initialization vector for use with this camellia structure

See: wc_CamelliaSetKey

Return:

- 0 Returned upon successfully setting the key and initialization vector
- BAD_FUNC_ARG returned if there is an error processing one of the input arguments

Example

```
Camellia cam;
byte iv[16];
// initialize iv
if( wc_CamelliaSetIV(&cam, iv) != 0) {
// error initializing camellia structure
}
```

```
int wc_CamelliaEncryptDirect(
    Camellia * cam,
    byte * out,
    const byte * in
)
```

This function does a one-block encrypt using the provided camellia object. It parses the first 16 byte block from the buffer in and stores the encrypted result in the buffer out. Before using this function, one should initialize the camellia object using wc_CamelliaSetKey.

Parameters:

- **cam** pointer to the camellia structure to use for encryption

- **out** pointer to the buffer in which to store the encrypted block
- **in** pointer to the buffer containing the plaintext block to encrypt

See: wc_CamelliaDecryptDirect

Return: none No returns.

Example

```
Camellia cam;
// initialize cam structure with key and iv
byte plain[] = { // initialize with message to encrypt };
byte cipher[16];

wc_CamelliaEncryptDirect(&ca, cipher, plain);
```

```
int wc_CamelliaDecryptDirect(
    Camellia * cam,
    byte * out,
    const byte * in
)
```

This function does a one-block decrypt using the provided camellia object. It parses the first 16 byte block from the buffer in, decrypts it, and stores the result in the buffer out. Before using this function, one should initialize the camellia object using wc_CamelliaSetKey.

Parameters:

- **cam** pointer to the camellia structure to use for encryption
- **out** pointer to the buffer in which to store the decrypted plaintext block
- **in** pointer to the buffer containing the ciphertext block to decrypt

See: wc_CamelliaEncryptDirect

Return: none No returns.

Example

```
Camellia cam;
// initialize cam structure with key and iv
byte cipher[] = { // initialize with encrypted message to decrypt };
byte decrypted[16];

wc_CamelliaDecryptDirect(&cam, decrypted, cipher);
```

```
int wc_CamelliaCbcEncrypt(
    Camellia * cam,
    byte * out,
    const byte * in,
    word32 sz
)
```

This function encrypts the plaintext from the buffer in and stores the output in the buffer out. It performs this encryption using Camellia with Cipher Block Chaining (CBC).

Parameters:

- **cam** pointer to the camellia structure to use for encryption
- **out** pointer to the buffer in which to store the encrypted ciphertext
- **in** pointer to the buffer containing the plaintext to encrypt
- **sz** the size of the message to encrypt

See: wc_CamelliaCbcDecrypt

Return: none No returns.

Example

```
Camellia cam;
// initialize cam structure with key and iv
byte plain[] = { // initialize with encrypted message to decrypt };
byte cipher[sizeof(plain)];

wc_CamelliaCbcEncrypt(&cam, cipher, plain, sizeof(plain));
```

```
int wc_CamelliaCbcDecrypt(
    Camellia * cam,
    byte * out,
    const byte * in,
    word32 sz
)
```

This function decrypts the ciphertext from the buffer in and stores the output in the buffer out. It performs this decryption using Camellia with Cipher Block Chaining (CBC).

Parameters:

- **cam** pointer to the camellia structure to use for encryption
- **out** pointer to the buffer in which to store the decrypted message
- **in** pointer to the buffer containing the encrypted ciphertext
- **sz** the size of the message to encrypt

See: wc_CamelliaCbcEncrypt

Return: none No returns.

Example

```
Camellia cam;
// initialize cam structure with key and iv
byte cipher[] = { // initialize with encrypted message to decrypt };
byte decrypted[sizeof(cipher)];

wc_CamelliaCbcDecrypt(&cam, decrypted, cipher, sizeof(cipher));
```

B.17 Algorithms - ChaCha

B.16.2.6 function wc_CamelliaCbcDecrypt

B.17.1 Functions

	Name
int	**wc_Chacha_SetIV**(ChaCha * ctx, const byte * inIv, word32 counter)This function sets the initialization vector (nonce) for a ChaCha object, initializing it for use as a cipher. It should be called after the key has been set, using wc_Chacha_SetKey. A difference nonce should be used for each round of encryption.
int	**wc_Chacha_Process**(ChaCha * ctx, byte * cipher, const byte * plain, word32 msglen)This function processes the text from the buffer input, encrypts or decrypts it, and stores the result in the buffer output.
int	**wc_Chacha_SetKey**(ChaCha * ctx, const byte * key, word32 keySz)This function sets the key for a ChaCha object, initializing it for use as a cipher. It should be called before setting the nonce with wc_Chacha_SetIV, and before using it for encryption with wc_Chacha_Process.

B.17.2 Functions Documentation

```
int wc_Chacha_SetIV(
    ChaCha * ctx,
    const byte * inIv,
    word32 counter
)
```

This function sets the initialization vector (nonce) for a ChaCha object, initializing it for use as a cipher. It should be called after the key has been set, using wc_Chacha_SetKey. A difference nonce should be used for each round of encryption.

Parameters:

- **ctx** pointer to the ChaCha structure on which to set the iv
- **inIv** pointer to a buffer containing the 12 byte initialization vector with which to initialize the ChaCha structure
- **counter** the value at which the block counter should start–usually zero.

See:

- wc_Chacha_SetKey
- wc_Chacha_Process

Return:

- 0 Returned upon successfully setting the initialization vector
- BAD_FUNC_ARG returned if there is an error processing the ctx input argument

Example

```
ChaCha enc;
// initialize enc with wc_Chacha_SetKey
byte iv[12];
// initialize iv
if( wc_Chacha_SetIV(&enc, iv, 0) != 0) {
    // error initializing ChaCha structure
}
```

```
int wc_Chacha_Process(
    ChaCha * ctx,
    byte * cipher,
    const byte * plain,
    word32 msglen
)
```

This function processes the text from the buffer input, encrypts or decrypts it, and stores the result in the buffer output.

Parameters:

- **ctx** pointer to the ChaCha structure on which to set the iv
- **output** pointer to a buffer in which to store the output ciphertext or decrypted plaintext
- **input** pointer to the buffer containing the input plaintext to encrypt or the input ciphertext to decrypt
- **msglen** length of the message to encrypt or the ciphertext to decrypt

See:

- wc_Chacha_SetKey
- wc_Chacha_Process

Return:

- 0 Returned upon successfully encrypting or decrypting the input
- BAD_FUNC_ARG returned if there is an error processing the ctx input argument

Example

```
ChaCha enc;
// initialize enc with wc_Chacha_SetKey and wc_Chacha_SetIV

byte plain[] = { // initialize plaintext };
```

```
byte cipher[sizeof(plain)];
if( wc_Chacha_Process(&enc, cipher, plain, sizeof(plain)) != 0) {
    // error processing ChaCha cipher
}
```

```
int wc_Chacha_SetKey(
    ChaCha * ctx,
    const byte * key,
    word32 keySz
)
```

This function sets the key for a ChaCha object, initializing it for use as a cipher. It should be called before setting the nonce with wc_Chacha_SetIV, and before using it for encryption with wc_Chacha_Process.

Parameters:

- **ctx** pointer to the ChaCha structure in which to set the key
- **key** pointer to a buffer containing the 16 or 32 byte key with which to initialize the ChaCha structure
- **keySz** the length of the key passed in

See:

- wc_Chacha_SetIV
- wc_Chacha_Process

Return:

- 0 Returned upon successfully setting the key
- BAD_FUNC_ARG returned if there is an error processing the ctx input argument or if the key is not 16 or 32 bytes long

Example

```
ChaCha enc;
byte key[] = { // initialize key };

if( wc_Chacha_SetKey(&enc, key, sizeof(key)) != 0) {
    // error initializing ChaCha structure
}
```

B.18 Algorithms - ChaCha20_Poly1305

B.17.2.3 function wc_Chacha_SetKey

B.18.1 Functions

	Name
int	**wc_ChaCha20Poly1305_Encrypt**(const byte inKey[CHACHA20_POLY1305_AEAD_KEYSIZE], const byte inIV[CHACHA20_POLY1305_AEAD_IV_SIZE], const byte * inAAD, const word32 inAADLen, const byte * inPlaintext, const word32 inPlaintextLen, byte * outCiphertext, byte outAuth-Tag[CHACHA20_POLY1305_AEAD_AUTHTAG_SIZE])This function encrypts an input message, inPlaintext, using the ChaCha20 stream cipher, into the output buffer, outCiphertext. It also performs Poly_1305 authentication (on the cipher text), and stores the generated authentication tag in the output buffer, outAuthTag.
int	**wc_ChaCha20Poly1305_Decrypt**(const byte inKey[CHACHA20_POLY1305_AEAD_KEYSIZE], const byte inIV[CHACHA20_POLY1305_AEAD_IV_SIZE], const byte * inAAD, const word32 inAADLen, const byte * inCiphertext, const word32 inCiphertextLen, const byte inAuth-Tag[CHACHA20_POLY1305_AEAD_AUTHTAG_SIZE], byte * outPlaintext)This function decrypts input ciphertext, inCiphertext, using the ChaCha20 stream cipher, into the output buffer, outPlaintext. It also performs Poly_1305 authentication, comparing the given inAuthTag to an authentication generated with the inAAD (arbitrary length additional authentication data). If a nonzero error code is returned, the output data, outPlaintext, is undefined. However, callers must unconditionally zeroize the output buffer to guard against leakage of cleartext data.

B.18.2 Functions Documentation

```
int wc_ChaCha20Poly1305_Encrypt(
    const byte inKey[CHACHA20_POLY1305_AEAD_KEYSIZE],
    const byte inIV[CHACHA20_POLY1305_AEAD_IV_SIZE],
    const byte * inAAD,
    const word32 inAADLen,
    const byte * inPlaintext,
    const word32 inPlaintextLen,
    byte * outCiphertext,
    byte outAuthTag[CHACHA20_POLY1305_AEAD_AUTHTAG_SIZE]
)
```

This function encrypts an input message, inPlaintext, using the ChaCha20 stream cipher, into the output buffer, outCiphertext. It also performs Poly-1305 authentication (on the cipher text), and stores

the generated authentication tag in the output buffer, outAuthTag.

Parameters:

- **inKey** pointer to a buffer containing the 32 byte key to use for encryption
- **inIv** pointer to a buffer containing the 12 byte iv to use for encryption
- **inAAD** pointer to the buffer containing arbitrary length additional authenticated data (AAD)
- **inAADLen** length of the input AAD
- **inPlaintext** pointer to the buffer containing the plaintext to encrypt
- **inPlaintextLen** the length of the plain text to encrypt
- **outCiphertext** pointer to the buffer in which to store the ciphertext
- **outAuthTag** pointer to a 16 byte wide buffer in which to store the authentication tag

See:

- wc_ChaCha20Poly1305_Decrypt
- wc_ChaCha_*
- wc_Poly1305*

Return:

- 0 Returned upon successfully encrypting the message
- BAD_FUNC_ARG returned if there is an error during the encryption process

Example

```
byte key[] = { // initialize 32 byte key };
byte iv[]  = { // initialize 12 byte key };
byte inAAD[] = { // initialize AAD };

byte plain[] = { // initialize message to encrypt };
byte cipher[sizeof(plain)];
byte authTag[16];

int ret = wc_ChaCha20Poly1305_Encrypt(key, iv, inAAD, sizeof(inAAD),
plain, sizeof(plain), cipher, authTag);

if(ret != 0) {
    // error running encrypt
}
```

```
int wc_ChaCha20Poly1305_Decrypt(
    const byte inKey[CHACHA20_POLY1305_AEAD_KEYSIZE],
    const byte inIV[CHACHA20_POLY1305_AEAD_IV_SIZE],
    const byte * inAAD,
    const word32 inAADLen,
    const byte * inCiphertext,
    const word32 inCiphertextLen,
    const byte inAuthTag[CHACHA20_POLY1305_AEAD_AUTHTAG_SIZE],
    byte * outPlaintext
```

)

This function decrypts input ciphertext, inCiphertext, using the ChaCha20 stream cipher, into the output buffer, outPlaintext. It also performs Poly-1305 authentication, comparing the given inAuthTag to an authentication generated with the inAAD (arbitrary length additional authentication data). If a nonzero error code is returned, the output data, outPlaintext, is undefined. However, callers must unconditionally zeroize the output buffer to guard against leakage of cleartext data.

Parameters:

- **inKey** pointer to a buffer containing the 32 byte key to use for decryption
- **inIv** pointer to a buffer containing the 12 byte iv to use for decryption
- **inAAD** pointer to the buffer containing arbitrary length additional authenticated data (AAD)
- **inAADLen** length of the input AAD
- **inCiphertext** pointer to the buffer containing the ciphertext to decrypt
- **outCiphertextLen** the length of the ciphertext to decrypt
- **inAuthTag** pointer to the buffer containing the 16 byte digest for authentication
- **outPlaintext** pointer to the buffer in which to store the plaintext

See:

- wc_ChaCha20Poly1305_Encrypt
- wc_ChaCha_*
- wc_Poly1305*

Return:

- 0 Returned upon successfully decrypting and authenticating the message
- BAD_FUNC_ARG Returned if any of the function arguments do not match what is expected
- MAC_CMP_FAILED_E Returned if the generated authentication tag does not match the supplied inAuthTag.
- MEMORY_E Returned if internal buffer allocation failed.
- CHACHA_POLY_OVERFLOW Can be returned if input is corrupted.

Example

```
byte key[]   = { // initialize 32 byte key };
byte iv[]    = { // initialize 12 byte key };
byte inAAD[] = { // initialize AAD };

byte cipher[]    = { // initialize with received ciphertext };
byte authTag[16] = { // initialize with received authentication tag };

byte plain[sizeof(cipher)];

int ret = wc_ChaCha20Poly1305_Decrypt(key, iv, inAAD, sizeof(inAAD),
cipher, sizeof(cipher), authTag, plain);

if(ret == MAC_CMP_FAILED_E) {
    // error during authentication
} else if( ret != 0) {
```

```
    // error with function arguments
}
```

B.19 Callbacks - CryptoCb

B.18.2.2 function wc_ChaCha20Poly1305_Decrypt

B.20 Algorithms - Curve25519

B.20.1 Functions

	Name
int	**wc_curve25519_make_key**(WC_RNG * rng, int keysize, curve25519_key * key)This function generates a Curve25519 key using the given random number generator, rng, of the size given (keysize), and stores it in the given curve25519_key structure. It should be called after the key structure has been initialized through wc_curve25519_init().
int	**wc_curve25519_shared_secret**(curve25519_key * private_key, curve25519_key * public_key, byte * out, word32 * outlen)This function computes a shared secret key given a secret private key and a received public key. It stores the generated secret key in the buffer out and assigns the variable of the secret key to outlen. Only supports big endian.
int	**wc_curve25519_shared_secret_ex**(curve25519_key * private_key, curve25519_key * public_key, byte * out, word32 * outlen, int endian)This function computes a shared secret key given a secret private key and a received public key. It stores the generated secret key in the buffer out and assigns the variable of the secret key to outlen. Supports both big and little endian.
int	**wc_curve25519_init**(curve25519_key * key)This function initializes a Curve25519 key. It should be called before generating a key for the structure.
void	**wc_curve25519_free**(curve25519_key * key)This function frees a Curve25519 object.
int	**wc_curve25519_import_private**(const byte * priv, word32 privSz, curve25519_key * key)This function imports a curve25519 private key only. (Big endian).
int	**wc_curve25519_import_private_ex**(const byte * priv, word32 privSz, curve25519_key * key, int endian)curve25519 private key import only. (Big or Little endian).

	Name
int	**wc_curve25519_import_private_raw**(const byte * priv, word32 privSz, const byte * pub, word32 pubSz, curve25519_key * key)This function imports a public-private key pair into a curve25519_key structure. Big endian only.
int	**wc_curve25519_import_private_raw_ex**(const byte * priv, word32 privSz, const byte * pub, word32 pubSz, curve25519_key * key, int endian)This function imports a public-private key pair into a curve25519_key structure. Supports both big and little endian.
int	**wc_curve25519_export_private_raw**(curve25519_key * key, byte * out, word32 * outLen)This function exports a private key from a curve25519_key structure and stores it in the given out buffer. It also sets outLen to be the size of the exported key. Big Endian only.
int	**wc_curve25519_export_private_raw_ex**(curve25519_key * key, byte * out, word32 * outLen, int endian)This function exports a private key from a curve25519_key structure and stores it in the given out buffer. It also sets outLen to be the size of the exported key. Can specify whether it's big or little endian.
int	**wc_curve25519_import_public**(const byte * in, word32 inLen, curve25519_key * key)This function imports a public key from the given in buffer and stores it in the curve25519_key structure.
int	**wc_curve25519_import_public_ex**(const byte * in, word32 inLen, curve25519_key * key, int endian)This function imports a public key from the given in buffer and stores it in the curve25519_key structure.
int	**wc_curve25519_check_public**(const byte * pub, word32 pubSz, int endian)This function checks that a public key buffer holds a valid Curve25519 key value given the endian ordering.
int	**wc_curve25519_export_public**(curve25519_key * key, byte * out, word32 * outLen)This function exports a public key from the given key structure and stores the result in the out buffer. Big endian only.
int	**wc_curve25519_export_public_ex**(curve25519_key * key, byte * out, word32 * outLen, int endian)This function exports a public key from the given key structure and stores the result in the out buffer. Supports both big and little endian.

	Name
int	**wc_curve25519_export_key_raw**(curve25519_key * key, byte * priv, word32 * privSz, byte * pub, word32 * pubSz)Export Curve25519 key pair. Big endian only.
int	**wc_curve25519_export_key_raw_ex**(curve25519_key * key, byte * priv, word32 * privSz, byte * pub, word32 * pubSz, int endian)Export curve25519 key pair. Big or little endian.
int	**wc_curve25519_size**(curve25519_key * key)This function returns the key size of the given key structure.

B.20.2 Functions Documentation

```
int wc_curve25519_make_key(
    WC_RNG * rng,
    int keysize,
    curve25519_key * key
)
```

This function generates a Curve25519 key using the given random number generator, rng, of the size given (keysize), and stores it in the given curve25519_key structure. It should be called after the key structure has been initialized through wc_curve25519_init().

Parameters:

- **rng** Pointer to the RNG object used to generate the ecc key.
- **keysize** Size of the key to generate. Must be 32 bytes for curve25519.
- **key** Pointer to the curve25519_key structure in which to store the generated key.

See: wc_curve25519_init

Return:

- 0 Returned on successfully generating the key and and storing it in the given curve25519_key structure.
- ECC_BAD_ARG_E Returned if the input keysize does not correspond to the keysize for a curve25519 key (32 bytes).
- RNG_FAILURE_E Returned if the rng internal status is not DRBG_OK or if there is in generating the next random block with rng.
- BAD_FUNC_ARG Returned if any of the input parameters passed in are NULL.

Example

```
int ret;

curve25519_key key;
wc_curve25519_init(&key); // initialize key
WC_RNG rng;
wc_InitRng(&rng); // initialize random number generator
```

```
ret = wc_curve25519_make_key(&rng, 32, &key);
if (ret != 0) {
    // error making Curve25519 key
}
```

```
int wc_curve25519_shared_secret(
    curve25519_key * private_key,
    curve25519_key * public_key,
    byte * out,
    word32 * outlen
)
```

This function computes a shared secret key given a secret private key and a received public key. It stores the generated secret key in the buffer out and assigns the variable of the secret key to outlen. Only supports big endian.

Parameters:

- **private_key** Pointer to the curve25519_key structure initialized with the user's private key.
- **public_key** Pointer to the curve25519_key structure containing the received public key.
- **out** Pointer to a buffer in which to store the 32 byte computed secret key.
- **outlen** Pointer in which to store the length written to the output buffer.

See:

- wc_curve25519_init
- wc_curve25519_make_key
- wc_curve25519_shared_secret_ex

Return:

- 0 Returned on successfully computing a shared secret key.
- BAD_FUNC_ARG Returned if any of the input parameters passed in are NULL.
- ECC_BAD_ARG_E Returned if the first bit of the public key is set, to avoid implementation fingerprinting.

Example

```
int ret;

byte sharedKey[32];
word32 keySz;
curve25519_key privKey, pubKey;
// initialize both keys

ret = wc_curve25519_shared_secret(&privKey, &pubKey, sharedKey, &keySz);
if (ret != 0) {
    // error generating shared key
}
```

```
int wc_curve25519_shared_secret_ex(
    curve25519_key * private_key,
    curve25519_key * public_key,
    byte * out,
    word32 * outlen,
    int endian
)
```

This function computes a shared secret key given a secret private key and a received public key. It stores the generated secret key in the buffer out and assigns the variable of the secret key to outlen. Supports both big and little endian.

Parameters:

- **private_key** Pointer to the curve25519_key structure initialized with the user's private key.
- **public_key** Pointer to the curve25519_key structure containing the received public key.
- **out** Pointer to a buffer in which to store the 32 byte computed secret key.
- **pinout]** outlen Pointer in which to store the length written to the output buffer.
- **endian** EC25519_BIG_ENDIAN or EC25519_LITTLE_ENDIAN to set which form to use.

See:

- wc_curve25519_init
- wc_curve25519_make_key
- wc_curve25519_shared_secret

Return:

- 0 Returned on successfully computing a shared secret key.
- BAD_FUNC_ARG Returned if any of the input parameters passed in are NULL.
- ECC_BAD_ARG_E Returned if the first bit of the public key is set, to avoid implementation fingerprinting.

Example

```
int ret;

byte sharedKey[32];
word32 keySz;

curve25519_key privKey, pubKey;
// initialize both keys

ret = wc_curve25519_shared_secret_ex(&privKey, &pubKey, sharedKey, &keySz,
        EC25519_BIG_ENDIAN);
if (ret != 0) {
    // error generating shared key
}
```

```
int wc_curve25519_init(
    curve25519_key * key
)
```

This function initializes a Curve25519 key. It should be called before generating a key for the structure.

Parameters:

- **key** Pointer to the curve25519_key structure to initialize.

See: wc_curve25519_make_key

Return:

- 0 Returned on successfully initializing the curve25519_key structure.
- BAD_FUNC_ARG Returned when key is NULL.

Example

```
curve25519_key key;
wc_curve25519_init(&key); // initialize key
// make key and proceed to encryption
```

```
void wc_curve25519_free(
    curve25519_key * key
)
```

This function frees a Curve25519 object.

Parameters:

- **key** Pointer to the key object to free.

See:

- wc_curve25519_init
- wc_curve25519_make_key

Example

```
curve25519_key privKey;
// initialize key, use it to generate shared secret key
wc_curve25519_free(&privKey);
```

```
int wc_curve25519_import_private(
    const byte * priv,
    word32 privSz,
    curve25519_key * key
)
```

This function imports a curve25519 private key only. (Big endian).

Parameters:

- **priv** Pointer to a buffer containing the private key to import.
- **privSz** Length of the private key to import.
- **key** Pointer to the structure in which to store the imported key.

See:

- wc_curve25519_import_private_ex
- wc_curve25519_size

Return:

- 0 Returned on successfully importing private key.
- BAD_FUNC_ARG Returns if key or priv is null.
- ECC_BAD_ARG_E Returns if privSz is not equal to CURVE25519_KEY_SIZE.

Example

```
int ret;

byte priv[] = { Contents of private key };
curve25519_key key;
wc_curve25519_init(&key);

ret = wc_curve25519_import_private(priv, sizeof(priv), &key);
if (ret != 0) {
    // error importing keys
}
```

```
int wc_curve25519_import_private_ex(
    const byte * priv,
    word32 privSz,
    curve25519_key * key,
    int endian
)
```

curve25519 private key import only. (Big or Little endian).

Parameters:

- **priv** Pointer to a buffer containing the private key to import.

- **privSz** Length of the private key to import.
- **key** Pointer to the structure in which to store the imported key.
- **endian** EC25519_BIG_ENDIAN or EC25519_LITTLE_ENDIAN to set which form to use.

See:

- wc_curve25519_import_private
- wc_curve25519_size

Return:

- 0 Returned on successfully importing private key.
- BAD_FUNC_ARG Returns if key or priv is null.
- ECC_BAD_ARG_E Returns if privSz is not equal to CURVE25519_KEY_SIZE.

Example

```
int ret;

byte priv[] = { // Contents of private key };
curve25519_key key;
wc_curve25519_init(&key);

ret = wc_curve25519_import_private_ex(priv, sizeof(priv), &key,
        EC25519_BIG_ENDIAN);
if (ret != 0) {
    // error importing keys
}
```

```
int wc_curve25519_import_private_raw(
    const byte * priv,
    word32 privSz,
    const byte * pub,
    word32 pubSz,
    curve25519_key * key
)
```

This function imports a public-private key pair into a curve25519_key structure. Big endian only.

Parameters:

- **priv** Pointer to a buffer containing the private key to import.
- **privSz** Length of the private key to import.
- **pub** Pointer to a buffer containing the public key to import.
- **pubSz** Length of the public key to import.
- **key** Pointer to the structure in which to store the imported keys.

See:

- wc_curve25519_init

- wc_curve25519_make_key
- wc_curve25519_import_public
- wc_curve25519_export_private_raw

Return:

- 0 Returned on importing into the curve25519_key structure
- BAD_FUNC_ARG Returns if any of the input parameters are null.
- ECC_BAD_ARG_E Returned if the input key's key size does not match the public or private key sizes.

Example

```
int ret;

byte priv[32];
byte pub[32];
// initialize with public and private keys
curve25519_key key;

wc_curve25519_init(&key);
// initialize key

ret = wc_curve25519_import_private_raw(&priv, sizeof(priv), pub,
        sizeof(pub), &key);
if (ret != 0) {
    // error importing keys
}
```

```
int wc_curve25519_import_private_raw_ex(
    const byte * priv,
    word32 privSz,
    const byte * pub,
    word32 pubSz,
    curve25519_key * key,
    int endian
)
```

This function imports a public-private key pair into a curve25519_key structure. Supports both big and little endian.

Parameters:

- **priv** Pointer to a buffer containing the private key to import.
- **privSz** Length of the private key to import.
- **pub** Pointer to a buffer containing the public key to import.
- **pubSz** Length of the public key to import.
- **key** Pointer to the structure in which to store the imported keys.
- **endian** EC25519_BIG_ENDIAN or EC25519_LITTLE_ENDIAN to set which form to use.

See:

- wc_curve25519_init
- wc_curve25519_make_key
- wc_curve25519_import_public
- wc_curve25519_export_private_raw
- wc_curve25519_import_private_raw

Return:

- 0 Returned on importing into the curve25519_key structure
- BAD_FUNC_ARG Returns if any of the input parameters are null.
- ECC_BAD_ARG_E Returned if or the input key's key size does not match the public or private key sizes

Example

```
int ret;
byte priv[32];
byte pub[32];
// initialize with public and private keys
curve25519_key key;

wc_curve25519_init(&key);
// initialize key

ret = wc_curve25519_import_private_raw_ex(&priv, sizeof(priv), pub,
        sizeof(pub), &key, EC25519_BIG_ENDIAN);
if (ret != 0) {
    // error importing keys
}
```

```
int wc_curve25519_export_private_raw(
    curve25519_key * key,
    byte * out,
    word32 * outLen
)
```

This function exports a private key from a curve25519_key structure and stores it in the given out buffer. It also sets outLen to be the size of the exported key. Big Endian only.

Parameters:

- **key** Pointer to the structure from which to export the key.
- **out** Pointer to the buffer in which to store the exported key.
- **outLen** On in, is the size of the out in bytes. On out, will store the bytes written to the output buffer.

See:

- wc_curve25519_init

- wc_curve25519_make_key
- wc_curve25519_import_private_raw
- wc_curve25519_export_private_raw_ex

Return:

- 0 Returned on successfully exporting the private key from the curve25519_key structure.
- BAD_FUNC_ARG Returned if any input parameters are NULL.
- ECC_BAD_ARG_E Returned if wc_curve25519_size() is not equal to key.

Example

```
int ret;
byte priv[32];
int privSz;

curve25519_key key;
// initialize and make key

ret = wc_curve25519_export_private_raw(&key, priv, &privSz);
if (ret != 0) {
    // error exporting key
}
```

```
int wc_curve25519_export_private_raw_ex(
    curve25519_key * key,
    byte * out,
    word32 * outLen,
    int endian
)
```

This function exports a private key from a curve25519_key structure and stores it in the given out buffer. It also sets outLen to be the size of the exported key. Can specify whether it's big or little endian.

Parameters:

- **key** Pointer to the structure from which to export the key.
- **out** Pointer to the buffer in which to store the exported key.
- **outLen** On in, is the size of the out in bytes. On out, will store the bytes written to the output buffer.
- **endian** EC25519_BIG_ENDIAN or EC25519_LITTLE_ENDIAN to set which form to use.

See:

- wc_curve25519_init
- wc_curve25519_make_key
- wc_curve25519_import_private_raw
- wc_curve25519_export_private_raw
- wc_curve25519_size

Return:

- 0 Returned on successfully exporting the private key from the curve25519_key structure.
- BAD_FUNC_ARG Returned if any input parameters are NULL.
- ECC_BAD_ARG_E Returned if wc_curve25519_size() is not equal to key.

Example

```
int ret;

byte priv[32];
int privSz;
curve25519_key key;
// initialize and make key
ret = wc_curve25519_export_private_raw_ex(&key, priv, &privSz,
        EC25519_BIG_ENDIAN);
if (ret != 0) {
    // error exporting key
}
```

```
int wc_curve25519_import_public(
    const byte * in,
    word32 inLen,
    curve25519_key * key
)
```

This function imports a public key from the given in buffer and stores it in the curve25519_key structure.

Parameters:

- **in** Pointer to the buffer containing the public key to import.
- **inLen** Length of the public key to import.
- **key** Pointer to the curve25519_key structure in which to store the key.

See:

- wc_curve25519_init
- wc_curve25519_export_public
- wc_curve25519_import_private_raw
- wc_curve25519_import_public_ex
- wc_curve25519_check_public
- wc_curve25519_size

Return:

- 0 Returned on successfully importing the public key into the curve25519_key structure.
- ECC_BAD_ARG_E Returned if the inLen parameter does not match the key size of the key structure.

- BAD_FUNC_ARG Returned if any of the input parameters are NULL.

Example

```
int ret;

byte pub[32];
// initialize pub with public key

curve25519_key key;
// initialize key

ret = wc_curve25519_import_public(pub,sizeof(pub), &key);
if (ret != 0) {
    // error importing key
}
```

```
int wc_curve25519_import_public_ex(
    const byte * in,
    word32 inLen,
    curve25519_key * key,
    int endian
)
```

This function imports a public key from the given in buffer and stores it in the curve25519_key structure.

Parameters:

- **in** Pointer to the buffer containing the public key to import.
- **inLen** Length of the public key to import.
- **key** Pointer to the curve25519_key structure in which to store the key.
- **endian** EC25519_BIG_ENDIAN or EC25519_LITTLE_ENDIAN to set which form to use.

See:

- wc_curve25519_init
- wc_curve25519_export_public
- wc_curve25519_import_private_raw
- wc_curve25519_import_public
- wc_curve25519_check_public
- wc_curve25519_size

Return:

- 0 Returned on successfully importing the public key into the curve25519_key structure.
- ECC_BAD_ARG_E Returned if the inLen parameter does not match the key size of the key structure.
- BAD_FUNC_ARG Returned if any of the input parameters are NULL.

Example

```
int ret;

byte pub[32];
// initialize pub with public key
curve25519_key key;
// initialize key

ret = wc_curve25519_import_public_ex(pub, sizeof(pub), &key,
        EC25519_BIG_ENDIAN);
if (ret != 0) {
    // error importing key
}
```

```
int wc_curve25519_check_public(
    const byte * pub,
    word32 pubSz,
    int endian
)
```

This function checks that a public key buffer holds a valid Curve25519 key value given the endian ordering.

Parameters:

- **pub** Pointer to the buffer containing the public key to check.
- **pubLen** Length of the public key to check.
- **endian** EC25519_BIG_ENDIAN or EC25519_LITTLE_ENDIAN to set which form to use.

See:

- wc_curve25519_init
- wc_curve25519_import_public
- wc_curve25519_import_public_ex
- wc_curve25519_size

Return:

- 0 Returned when the public key value is valid.
- ECC_BAD_ARG_E Returned if the public key value is not valid.
- BAD_FUNC_ARG Returned if any of the input parameters are NULL.

Example

```
int ret;

byte pub[] = { Contents of public key };
```

```
ret = wc_curve25519_check_public_ex(pub, sizeof(pub), EC25519_BIG_ENDIAN);
if (ret != 0) {
    // error importing key
}
```

```
int wc_curve25519_export_public(
    curve25519_key * key,
    byte * out,
    word32 * outLen
)
```

This function exports a public key from the given key structure and stores the result in the out buffer. Big endian only.

Parameters:

- **key** Pointer to the curve25519_key structure in from which to export the key.
- **out** Pointer to the buffer in which to store the public key.
- **outLen** On in, is the size of the out in bytes. On out, will store the bytes written to the output buffer.

See:

- wc_curve25519_init
- wc_curve25519_export_private_raw
- wc_curve25519_import_public

Return:

- 0 Returned on successfully exporting the public key from the curve25519_key structure.
- ECC_BAD_ARG_E Returned if outLen is less than CURVE25519_PUB_KEY_SIZE.
- BAD_FUNC_ARG Returned if any of the input parameters are NULL.

Example

```
int ret;

byte pub[32];
int pubSz;

curve25519_key key;
// initialize and make key
ret = wc_curve25519_export_public(&key, pub, &pubSz);
if (ret != 0) {
    // error exporting key
}
```

```
int wc_curve25519_export_public_ex(
    curve25519_key * key,
    byte * out,
    word32 * outLen,
    int endian
)
```

This function exports a public key from the given key structure and stores the result in the out buffer. Supports both big and little endian.

Parameters:

- **key** Pointer to the curve25519_key structure in from which to export the key.
- **out** Pointer to the buffer in which to store the public key.
- **outLen** On in, is the size of the out in bytes. On out, will store the bytes written to the output buffer.
- **endian** EC25519_BIG_ENDIAN or EC25519_LITTLE_ENDIAN to set which form to use.

See:

- wc_curve25519_init
- wc_curve25519_export_private_raw
- wc_curve25519_import_public

Return:

- 0 Returned on successfully exporting the public key from the curve25519_key structure.
- ECC_BAD_ARG_E Returned if outLen is less than CURVE25519_PUB_KEY_SIZE.
- BAD_FUNC_ARG Returned if any of the input parameters are NULL.

Example

```
int ret;

byte pub[32];
int pubSz;

curve25519_key key;
// initialize and make key

ret = wc_curve25519_export_public_ex(&key, pub, &pubSz, EC25519_BIG_ENDIAN);
if (ret != 0) {
    // error exporting key
}
```

```
int wc_curve25519_export_key_raw(
    curve25519_key * key,
    byte * priv,
    word32 * privSz,
    byte * pub,
```

```
    word32 * pubSz
)
```

Export Curve25519 key pair. Big endian only.

Parameters:

- **key** Pointer to the curve448_key structure in from which to export the key pair.
- **priv** Pointer to the buffer in which to store the private key.
- **privSz** On in, is the size of the priv buffer in bytes. On out, will store the bytes written to the priv buffer.
- **pub** Pointer to the buffer in which to store the public key.
- **pubSz** On in, is the size of the pub buffer in bytes. On out, will store the bytes written to the pub buffer.

See:

- wc_curve25519_export_key_raw_ex
- wc_curve25519_export_private_raw

Return:

- 0 Returned on successfully exporting the key pair from the curve25519_key structure.
- BAD_FUNC_ARG Returned if any input parameters are NULL.
- ECC_BAD_ARG_E Returned if privSz is less than CURVE25519_KEY_SIZE or pubSz is less than CURVE25519_PUB_KEY_SIZE.

Example

```
int ret;

byte pub[32];
byte priv[32];
int pubSz;
int privSz;

curve25519_key key;
// initialize and make key

ret = wc_curve25519_export_key_raw(&key, priv, &privSz, pub, &pubSz);
if (ret != 0) {
    // error exporting key
}
```

```
int wc_curve25519_export_key_raw_ex(
    curve25519_key * key,
    byte * priv,
    word32 * privSz,
    byte * pub,
    word32 * pubSz,
```

```
    int endian
)
```

Export curve25519 key pair. Big or little endian.

Parameters:

- **key** Pointer to the curve448_key structure in from which to export the key pair.
- **priv** Pointer to the buffer in which to store the private key.
- **privSz** On in, is the size of the priv buffer in bytes. On out, will store the bytes written to the priv buffer.
- **pub** Pointer to the buffer in which to store the public key.
- **pubSz** On in, is the size of the pub buffer in bytes. On out, will store the bytes written to the pub buffer.
- **endian** EC25519_BIG_ENDIAN or EC25519_LITTLE_ENDIAN to set which form to use.

See:

- wc_curve25519_export_key_raw
- wc_curve25519_export_private_raw_ex
- wc_curve25519_export_public_ex

Return:

- 0 Returned on successfully exporting the key pair from the curve25519_key structure.
- BAD_FUNC_ARG Returned if any input parameters are NULL.
- ECC_BAD_ARG_E Returned if privSz is less than CURVE25519_KEY_SIZE or pubSz is less than CURVE25519_PUB_KEY_SIZE.

Example

```
int ret;

byte pub[32];
byte priv[32];
int pubSz;
int privSz;

curve25519_key key;
// initialize and make key

ret = wc_curve25519_export_key_raw_ex(&key,priv, &privSz, pub, &pubSz,
        EC25519_BIG_ENDIAN);
if (ret != 0) {
    // error exporting key
}
```

```
int wc_curve25519_size(
    curve25519_key * key
)
```

This function returns the key size of the given key structure.

Parameters:

- **key** Pointer to the curve25519_key structure in for which to determine the key size.

See:

- wc_curve25519_init
- wc_curve25519_make_key

Return:

- Success Given a valid, initialized curve25519_key structure, returns the size of the key.
- 0 Returned if key is NULL

Example

```
int keySz;

curve25519_key key;
// initialize and make key

keySz = wc_curve25519_size(&key);
```

B.21 Algorithms - Curve448

B.20.2.19 function wc_curve25519_size

B.21.1 Functions

	Name
int	**wc_curve448_make_key**(WC_RNG * rng, int keysize, curve448_key * key)This function generates a Curve448 key using the given random number generator, rng, of the size given (keysize), and stores it in the given curve448_key structure. It should be called after the key structure has been initialized through wc_curve448_init().
int	**wc_curve448_shared_secret**(curve448_key * private_key, curve448_key * public_key, byte * out, word32 * outlen)This function computes a shared secret key given a secret private key and a received public key. It stores the generated secret key in the buffer out and assigns the variable of the secret key to outlen. Only supports big endian.

	Name
int	**wc_curve448_shared_secret_ex**(curve448_key * private_key, curve448_key * public_key, byte * out, word32 * outlen, int endian)This function computes a shared secret key given a secret private key and a received public key. It stores the generated secret key in the buffer out and assigns the variable of the secret key to outlen. Supports both big and little endian.
int	**wc_curve448_init**(curve448_key * key)This function initializes a Curve448 key. It should be called before generating a key for the structure.
void	**wc_curve448_free**(curve448_key * key)This function frees a Curve448 object.
int	**wc_curve448_import_private**(const byte * priv, word32 privSz, curve448_key * key)This function imports a curve448 private key only. (Big endian).
int	**wc_curve448_import_private_ex**(const byte * priv, word32 privSz, curve448_key * key, int endian)curve448 private key import only. (Big or Little endian).
int	**wc_curve448_import_private_raw**(const byte * priv, word32 privSz, const byte * pub, word32 pubSz, curve448_key * key)This function imports a public-private key pair into a curve448_key structure. Big endian only.
int	**wc_curve448_import_private_raw_ex**(const byte * priv, word32 privSz, const byte * pub, word32 pubSz, curve448_key * key, int endian)This function imports a public-private key pair into a curve448_key structure. Supports both big and little endian.
int	**wc_curve448_export_private_raw**(curve448_key * key, byte * out, word32 * outLen)This function exports a private key from a curve448_key structure and stores it in the given out buffer. It also sets outLen to be the size of the exported key. Big Endian only.
int	**wc_curve448_export_private_raw_ex**(curve448_key * key, byte * out, word32 * outLen, int endian)This function exports a private key from a curve448_key structure and stores it in the given out buffer. It also sets outLen to be the size of the exported key. Can specify whether it's big or little endian.
int	**wc_curve448_import_public**(const byte * in, word32 inLen, curve448_key * key)This function imports a public key from the given in buffer and stores it in the curve448_key structure.

	Name
int	**wc_curve448_import_public_ex**(const byte * in, word32 inLen, curve448_key * key, int endian)This function imports a public key from the given in buffer and stores it in the curve448_key structure.
int	**wc_curve448_check_public**(const byte * pub, word32 pubSz, int endian)This function checks that a public key buffer holds a valid Curve448 key value given the endian ordering.
int	**wc_curve448_export_public**(curve448_key * key, byte * out, word32 * outLen)This function exports a public key from the given key structure and stores the result in the out buffer. Big endian only.
int	**wc_curve448_export_public_ex**(curve448_key * key, byte * out, word32 * outLen, int endian)This function exports a public key from the given key structure and stores the result in the out buffer. Supports both big and little endian.
int	**wc_curve448_export_key_raw**(curve448_key * key, byte * priv, word32 * privSz, byte * pub, word32 * pubSz)This function exports a key pair from the given key structure and stores the result in the out buffer. Big endian only.
int	**wc_curve448_export_key_raw_ex**(curve448_key * key, byte * priv, word32 * privSz, byte * pub, word32 * pubSz, int endian)Export curve448 key pair. Big or little endian.
int	**wc_curve448_size**(curve448_key * key)This function returns the key size of the given key structure.

B.21.2 Functions Documentation

```
int wc_curve448_make_key(
    WC_RNG * rng,
    int keysize,
    curve448_key * key
)
```

This function generates a Curve448 key using the given random number generator, rng, of the size given (keysize), and stores it in the given curve448_key structure. It should be called after the key structure has been initialized through wc_curve448_init().

Parameters:

- **rng** Pointer to the RNG object used to generate the ecc key.
- **keysize** Size of the key to generate. Must be 56 bytes for curve448.
- **key** Pointer to the curve448_key structure in which to store the generated key.

See: wc_curve448_init

Return:

- 0 Returned on successfully generating the key and and storing it in the given curve448_key structure.
- ECC_BAD_ARG_E Returned if the input keysize does not correspond to the keysize for a curve448 key (56 bytes).
- RNG_FAILURE_E Returned if the rng internal status is not DRBG_OK or if there is in generating the next random block with rng.
- BAD_FUNC_ARG Returned if any of the input parameters passed in are NULL.

Example

```
int ret;

curve448_key key;
wc_curve448_init(&key); // initialize key
WC_RNG rng;
wc_InitRng(&rng); // initialize random number generator

ret = wc_curve448_make_key(&rng, 56, &key);
if (ret != 0) {
    // error making Curve448 key
}
```

```
int wc_curve448_shared_secret(
    curve448_key * private_key,
    curve448_key * public_key,
    byte * out,
    word32 * outlen
)
```

This function computes a shared secret key given a secret private key and a received public key. It stores the generated secret key in the buffer out and assigns the variable of the secret key to outlen. Only supports big endian.

Parameters:

- **private_key** Pointer to the curve448_key structure initialized with the user's private key.
- **public_key** Pointer to the curve448_key structure containing the received public key.
- **out** Pointer to a buffer in which to store the 56 byte computed secret key.
- **outlen** Pointer in which to store the length written to the output buffer.

See:

- wc_curve448_init
- wc_curve448_make_key
- wc_curve448_shared_secret_ex

Return:

- 0 Returned on successfully computing a shared secret key
- BAD_FUNC_ARG Returned if any of the input parameters passed in are NULL

Example

```
int ret;

byte sharedKey[56];
word32 keySz;
curve448_key privKey, pubKey;
// initialize both keys

ret = wc_curve448_shared_secret(&privKey, &pubKey, sharedKey, &keySz);
if (ret != 0) {
    // error generating shared key
}
```

```
int wc_curve448_shared_secret_ex(
    curve448_key * private_key,
    curve448_key * public_key,
    byte * out,
    word32 * outlen,
    int endian
)
```

This function computes a shared secret key given a secret private key and a received public key. It stores the generated secret key in the buffer out and assigns the variable of the secret key to outlen. Supports both big and little endian.

Parameters:

- **private_key** Pointer to the curve448_key structure initialized with the user's private key.
- **public_key** Pointer to the curve448_key structure containing the received public key.
- **out** Pointer to a buffer in which to store the 56 byte computed secret key.
- **outlen** Pointer in which to store the length written to the output buffer.
- **endian** EC448_BIG_ENDIAN or EC448_LITTLE_ENDIAN to set which form to use.

See:

- wc_curve448_init
- wc_curve448_make_key
- wc_curve448_shared_secret

Return:

- 0 Returned on successfully computing a shared secret key.
- BAD_FUNC_ARG Returned if any of the input parameters passed in are NULL.

Example

```
int ret;

byte sharedKey[56];
word32 keySz;

curve448_key privKey, pubKey;
// initialize both keys

ret = wc_curve448_shared_secret_ex(&privKey, &pubKey, sharedKey, &keySz,
        EC448_BIG_ENDIAN);
if (ret != 0) {
    // error generating shared key
}
```

```
int wc_curve448_init(
    curve448_key * key
)
```

This function initializes a Curve448 key. It should be called before generating a key for the structure.

Parameters:

- **key** Pointer to the curve448_key structure to initialize.

See: wc_curve448_make_key

Return:

- 0 Returned on successfully initializing the curve448_key structure.
- BAD_FUNC_ARG Returned when key is NULL.

Example

```
curve448_key key;
wc_curve448_init(&key); // initialize key
// make key and proceed to encryption
```

```
void wc_curve448_free(
    curve448_key * key
)
```

This function frees a Curve448 object.

Parameters:

- **key** Pointer to the key object to free.

See:

- wc_curve448_init
- wc_curve448_make_key

Example

```
curve448_key privKey;
// initialize key, use it to generate shared secret key
wc_curve448_free(&privKey);
```

```
int wc_curve448_import_private(
    const byte * priv,
    word32 privSz,
    curve448_key * key
)
```

This function imports a curve448 private key only. (Big endian).

Parameters:

- **priv** Pointer to a buffer containing the private key to import.
- **privSz** Length of the private key to import.
- **key** Pointer to the structure in which to store the imported key.

See:

- wc_curve448_import_private_ex
- wc_curve448_size

Return:

- 0 Returned on successfully importing private key.
- BAD_FUNC_ARG Returns if key or priv is null.
- ECC_BAD_ARG_E Returns if privSz is not equal to CURVE448_KEY_SIZE.

Example

```
int ret;

byte priv[] = { Contents of private key };
curve448_key key;
wc_curve448_init(&key);

ret = wc_curve448_import_private(priv, sizeof(priv), &key);
if (ret != 0) {
    // error importing key
}
```

```
int wc_curve448_import_private_ex(
    const byte * priv,
    word32 privSz,
    curve448_key * key,
    int endian
)
```

curve448 private key import only. (Big or Little endian).

Parameters:

- **priv** Pointer to a buffer containing the private key to import.
- **privSz** Length of the private key to import.
- **key** Pointer to the structure in which to store the imported key.
- **endian** EC448_BIG_ENDIAN or EC448_LITTLE_ENDIAN to set which form to use.

See:

- wc_curve448_import_private
- wc_curve448_size

Return:

- 0 Returned on successfully importing private key.
- BAD_FUNC_ARG Returns if key or priv is null.
- ECC_BAD_ARG_E Returns if privSz is not equal to CURVE448_KEY_SIZE.

Example

```
int ret;

byte priv[] = { // Contents of private key };
curve448_key key;
wc_curve448_init(&key);

ret = wc_curve448_import_private_ex(priv, sizeof(priv), &key,
        EC448_BIG_ENDIAN);
if (ret != 0) {
    // error importing key
}
```

```
int wc_curve448_import_private_raw(
    const byte * priv,
    word32 privSz,
    const byte * pub,
    word32 pubSz,
    curve448_key * key
)
```

This function imports a public-private key pair into a curve448_key structure. Big endian only.

Parameters:

- **priv** Pointer to a buffer containing the private key to import.
- **privSz** Length of the private key to import.
- **pub** Pointer to a buffer containing the public key to import.
- **pubSz** Length of the public key to import.
- **key** Pointer to the structure in which to store the imported keys

See:

- wc_curve448_init
- wc_curve448_make_key
- wc_curve448_import_public
- wc_curve448_export_private_raw

Return:

- 0 Returned on importing into the curve448_key structure.
- BAD_FUNC_ARG Returns if any of the input parameters are null.
- ECC_BAD_ARG_E Returned if the input key's key size does not match the public or private key sizes.

Example

```
int ret;

byte priv[56];
byte pub[56];
// initialize with public and private keys
curve448_key key;

wc_curve448_init(&key);
// initialize key

ret = wc_curve448_import_private_raw(&priv, sizeof(priv), pub, sizeof(pub),
        &key);
if (ret != 0) {
    // error importing keys
}
```

```
int wc_curve448_import_private_raw_ex(
    const byte * priv,
    word32 privSz,
    const byte * pub,
    word32 pubSz,
    curve448_key * key,
    int endian
)
```

This function imports a public-private key pair into a curve448_key structure. Supports both big and little endian.

Parameters:

- **priv** Pointer to a buffer containing the private key to import.
- **privSz** Length of the private key to import.
- **pub** Pointer to a buffer containing the public key to import.
- **pubSz** Length of the public key to import.
- **key** Pointer to the structure in which to store the imported keys.
- **endian** EC448_BIG_ENDIAN or EC448_LITTLE_ENDIAN to set which form to use.

See:

- wc_curve448_init
- wc_curve448_make_key
- wc_curve448_import_public
- wc_curve448_export_private_raw
- wc_curve448_import_private_raw

Return:

- 0 Returned on importing into the curve448_key structure.
- BAD_FUNC_ARG Returns if any of the input parameters are null.
- ECC_BAD_ARG_E Returned if the input key's key size does not match the public or private key sizes.

Example

```
int ret;

byte priv[56];
byte pub[56];
// initialize with public and private keys
curve448_key key;

wc_curve448_init(&key);
// initialize key

ret = wc_curve448_import_private_raw_ex(&priv, sizeof(priv), pub,
        sizeof(pub), &key, EC448_BIG_ENDIAN);
if (ret != 0) {
    // error importing keys
}
```

```
int wc_curve448_export_private_raw(
    curve448_key * key,
    byte * out,
    word32 * outLen
)
```

This function exports a private key from a curve448_key structure and stores it in the given out buffer. It also sets outLen to be the size of the exported key. Big Endian only.

Parameters:

- **key** Pointer to the structure from which to export the key.
- **out** Pointer to the buffer in which to store the exported key.
- **outLen** On in, is the size of the out in bytes. On out, will store the bytes written to the output buffer.

See:

- wc_curve448_init
- wc_curve448_make_key
- wc_curve448_import_private_raw
- wc_curve448_export_private_raw_ex

Return:

- 0 Returned on successfully exporting the private key from the curve448_key structure.
- BAD_FUNC_ARG Returned if any input parameters are NULL.
- ECC_BAD_ARG_E Returned if wc_curve448_size() is not equal to key.

Example

```
int ret;
byte priv[56];
int privSz;

curve448_key key;
// initialize and make key

ret = wc_curve448_export_private_raw(&key, priv, &privSz);
if (ret != 0) {
    // error exporting key
}
```

```
int wc_curve448_export_private_raw_ex(
    curve448_key * key,
    byte * out,
    word32 * outLen,
    int endian
)
```

This function exports a private key from a curve448_key structure and stores it in the given out buffer. It also sets outLen to be the size of the exported key. Can specify whether it's big or little endian.

Parameters:

- **key** Pointer to the structure from which to export the key.

- **out** Pointer to the buffer in which to store the exported key.
- **outLen** On in, is the size of the out in bytes. On out, will store the bytes written to the output buffer.
- **endian** EC448_BIG_ENDIAN or EC448_LITTLE_ENDIAN to set which form to use.

See:

- wc_curve448_init
- wc_curve448_make_key
- wc_curve448_import_private_raw
- wc_curve448_export_private_raw
- wc_curve448_size

Return:

- 0 Returned on successfully exporting the private key from the curve448_key structure.
- BAD_FUNC_ARG Returned if any input parameters are NULL.
- ECC_BAD_ARG_E Returned if wc_curve448_size() is not equal to key.

Example

```
int ret;

byte priv[56];
int privSz;
curve448_key key;
// initialize and make key
ret = wc_curve448_export_private_raw_ex(&key, priv, &privSz,
        EC448_BIG_ENDIAN);
if (ret != 0) {
    // error exporting key
}
```

```
int wc_curve448_import_public(
    const byte * in,
    word32 inLen,
    curve448_key * key
)
```

This function imports a public key from the given in buffer and stores it in the curve448_key structure.

Parameters:

- **in** Pointer to the buffer containing the public key to import.
- **inLen** Length of the public key to import.
- **key** Pointer to the curve448_key structure in which to store the key.

See:

- wc_curve448_init

- wc_curve448_export_public
- wc_curve448_import_private_raw
- wc_curve448_import_public_ex
- wc_curve448_check_public
- wc_curve448_size

Return:

- 0 Returned on successfully importing the public key into the curve448_key structure.
- ECC_BAD_ARG_E Returned if the inLen parameter does not match the key size of the key structure.
- BAD_FUNC_ARG Returned if any of the input parameters are NULL.

Example

```
int ret;

byte pub[56];
// initialize pub with public key

curve448_key key;
// initialize key

ret = wc_curve448_import_public(pub,sizeof(pub), &key);
if (ret != 0) {
    // error importing key
}
```

```
int wc_curve448_import_public_ex(
    const byte * in,
    word32 inLen,
    curve448_key * key,
    int endian
)
```

This function imports a public key from the given in buffer and stores it in the curve448_key structure.

Parameters:

- **in** Pointer to the buffer containing the public key to import.
- **inLen** Length of the public key to import.
- **key** Pointer to the curve448_key structure in which to store the key.
- **endian** EC448_BIG_ENDIAN or EC448_LITTLE_ENDIAN to set which form to use.

See:

- wc_curve448_init
- wc_curve448_export_public
- wc_curve448_import_private_raw
- wc_curve448_import_public

- wc_curve448_check_public
- wc_curve448_size

Return:

- 0 Returned on successfully importing the public key into the curve448_key structure.
- ECC_BAD_ARG_E Returned if the inLen parameter does not match the key size of the key structure.
- BAD_FUNC_ARG Returned if any of the input parameters are NULL.

Example

```
int ret;

byte pub[56];
// initialize pub with public key
curve448_key key;
// initialize key

ret = wc_curve448_import_public_ex(pub, sizeof(pub), &key,
        EC448_BIG_ENDIAN);
if (ret != 0) {
    // error importing key
}
```

```
int wc_curve448_check_public(
    const byte * pub,
    word32 pubSz,
    int endian
)
```

This function checks that a public key buffer holds a valid Curve448 key value given the endian ordering.

Parameters:

- **pub** Pointer to the buffer containing the public key to check.
- **pubLen** Length of the public key to check.
- **endian** EC448_BIG_ENDIAN or EC448_LITTLE_ENDIAN to set which form to use.

See:

- wc_curve448_init
- wc_curve448_import_public
- wc_curve448_import_public_ex
- wc_curve448_size

Return:

- 0 Returned when the public key value is valid.

- ECC_BAD_ARG_E Returned if the public key value is not valid.
- BAD_FUNC_ARG Returned if any of the input parameters are NULL.

Example

```
int ret;

byte pub[] = { Contents of public key };

ret = wc_curve448_check_public_ex(pub, sizeof(pub), EC448_BIG_ENDIAN);
if (ret != 0) {
    // error importing key
}
```

```
int wc_curve448_export_public(
    curve448_key * key,
    byte * out,
    word32 * outLen
)
```

This function exports a public key from the given key structure and stores the result in the out buffer. Big endian only.

Parameters:

- **key** Pointer to the curve448_key structure in from which to export the key.
- **out** Pointer to the buffer in which to store the public key.
- **outLen** On in, is the size of the out in bytes. On out, will store the bytes written to the output buffer.

See:

- wc_curve448_init
- wc_curve448_export_private_raw
- wc_curve448_import_public

Return:

- 0 Returned on successfully exporting the public key from the curve448_key structure.
- ECC_BAD_ARG_E Returned if outLen is less than CURVE448_PUB_KEY_SIZE.
- BAD_FUNC_ARG Returned if any of the input parameters are NULL.

Example

```
int ret;

byte pub[56];
int pubSz;

curve448_key key;
```

```
// initialize and make key

ret = wc_curve448_export_public(&key, pub, &pubSz);
if (ret != 0) {
    // error exporting key
}
```

```
int wc_curve448_export_public_ex(
    curve448_key * key,
    byte * out,
    word32 * outLen,
    int endian
)
```

This function exports a public key from the given key structure and stores the result in the out buffer. Supports both big and little endian.

Parameters:

- **key** Pointer to the curve448_key structure in from which to export the key.
- **out** Pointer to the buffer in which to store the public key.
- **outLen** On in, is the size of the out in bytes. On out, will store the bytes written to the output buffer.
- **endian** EC448_BIG_ENDIAN or EC448_LITTLE_ENDIAN to set which form to use.

See:

- wc_curve448_init
- wc_curve448_export_private_raw
- wc_curve448_import_public

Return:

- 0 Returned on successfully exporting the public key from the curve448_key structure.
- ECC_BAD_ARG_E Returned if outLen is less than CURVE448_PUB_KEY_SIZE.
- BAD_FUNC_ARG Returned if any of the input parameters are NULL.

Example

```
int ret;

byte pub[56];
int pubSz;

curve448_key key;
// initialize and make key

ret = wc_curve448_export_public_ex(&key, pub, &pubSz, EC448_BIG_ENDIAN);
if (ret != 0) {
```

```
    // error exporting key
}



int wc_curve448_export_key_raw(
    curve448_key * key,
    byte * priv,
    word32 * privSz,
    byte * pub,
    word32 * pubSz
)
```

This function exports a key pair from the given key structure and stores the result in the out buffer. Big endian only.

Parameters:

- **key** Pointer to the curve448_key structure in from which to export the key pair.
- **priv** Pointer to the buffer in which to store the private key.
- **privSz** On in, is the size of the priv buffer in bytes. On out, will store the bytes written to the priv buffer.
- **pub** Pointer to the buffer in which to store the public key.
- **pubSz** On in, is the size of the pub buffer in bytes. On out, will store the bytes written to the pub buffer.

See:

- wc_curve448_export_key_raw_ex
- wc_curve448_export_private_raw

Return:

- 0 Returned on successfully exporting the key pair from the curve448_key structure.
- BAD_FUNC_ARG Returned if any input parameters are NULL.
- ECC_BAD_ARG_E Returned if privSz is less than CURVE448_KEY_SIZE or pubSz is less than CURVE448_PUB_KEY_SIZE.

Example

```
int ret;

byte pub[56];
byte priv[56];
int pubSz;
int privSz;

curve448_key key;
// initialize and make key

ret = wc_curve448_export_key_raw(&key, priv, &privSz, pub, &pubSz);
if (ret != 0) {
```

```
    // error exporting key
}



int wc_curve448_export_key_raw_ex(
    curve448_key * key,
    byte * priv,
    word32 * privSz,
    byte * pub,
    word32 * pubSz,
    int endian
)
```

Export curve448 key pair. Big or little endian.

Parameters:

- **key** Pointer to the curve448_key structure in from which to export the key pair.
- **priv** Pointer to the buffer in which to store the private key.
- **privSz** On in, is the size of the priv buffer in bytes. On out, will store the bytes written to the priv buffer.
- **pub** Pointer to the buffer in which to store the public key.
- **pubSz** On in, is the size of the pub buffer in bytes. On out, will store the bytes written to the pub buffer.
- **endian** EC448_BIG_ENDIAN or EC448_LITTLE_ENDIAN to set which form to use.

See:

- wc_curve448_export_key_raw
- wc_curve448_export_private_raw_ex
- wc_curve448_export_public_ex

Return:

- 0 Success
- BAD_FUNC_ARG Returned if any input parameters are NULL.
- ECC_BAD_ARG_E Returned if privSz is less than CURVE448_KEY_SIZE or pubSz is less than CURVE448_PUB_KEY_SIZE.

This function exports a key pair from the given key structure and stores the result in the out buffer. Big or little endian.

Example

```
int ret;

byte pub[56];
byte priv[56];
int pubSz;
int privSz;

curve448_key key;
```

```
// initialize and make key

ret = wc_curve448_export_key_raw_ex(&key,priv, &privSz, pub, &pubSz,
        EC448_BIG_ENDIAN);
if (ret != 0) {
    // error exporting key
}
```

```
int wc_curve448_size(
    curve448_key * key
)
```

This function returns the key size of the given key structure.

Parameters:

- **key** Pointer to the curve448_key structure in for which to determine the key size.

See:

- wc_curve448_init
- wc_curve448_make_key

Return:

- Success Given a valid, initialized curve448_key structure, returns the size of the key.
- 0 Returned if key is NULL.

Example

```
int keySz;

curve448_key key;
// initialize and make key

keySz = wc_curve448_size(&key);
```

B.22 Algorithms - DSA

B.21.2.19 function wc_curve448_size

B.22.1 Functions

	Name
int	**wc_InitDsaKey**(DsaKey * key)This function initializes a DsaKey object in order to use it for authentication via the Digital Signature Algorithm (DSA).
void	**wc_FreeDsaKey**(DsaKey * key)This function frees a DsaKey object after it has been used.
int	**wc_DsaSign**(const byte * digest, byte * out, DsaKey * key, WC_RNG * rng)This function signs the input digest and stores the result in the output buffer, out.
int	**wc_DsaVerify**(const byte * digest, const byte * sig, DsaKey * key, int * answer)This function verifies the signature of a digest, given a private key. It stores whether the key properly verifies in the answer parameter, with 1 corresponding to a successful verification, and 0 corresponding to failed verification.
int	**wc_DsaPublicKeyDecode**(const byte * input, word32 * inOutIdx, DsaKey * key, word32 inSz)This function decodes a DER formatted certificate buffer containing a DSA public key, and stores the key in the given DsaKey structure. It also sets the inOutIdx parameter according to the length of the input read.
int	**wc_DsaPrivateKeyDecode**(const byte * input, word32 * inOutIdx, DsaKey * key, word32 inSz)This function decodes a DER formatted certificate buffer containing a DSA private key, and stores the key in the given DsaKey structure. It also sets the inOutIdx parameter according to the length of the input read.
int	**wc_DsaKeyToDer**(DsaKey * key, byte * output, word32 inLen)Convert DsaKey key to DER format, write to output (inLen), return bytes written.
int	**wc_MakeDsaKey**(WC_RNG * rng, DsaKey * dsa)Create a DSA key.
int	**wc_MakeDsaParameters**(WC_RNG * rng, int modulus_size, DsaKey * dsa)FIPS 186_4 defines valid for modulus_size values as (1024, 160) (2048, 256) (3072, 256)

B.22.2 Functions Documentation

```
int wc_InitDsaKey(
    DsaKey * key
)
```

This function initializes a DsaKey object in order to use it for authentication via the Digital Signature Algorithm (DSA).

Parameters:

- **key** pointer to the DsaKey structure to initialize

See: wc_FreeDsaKey

Return:

- 0 Returned on success.
- BAD_FUNC_ARG Returned if a NULL key is passed in.

Example

```
DsaKey key;
int ret;
ret = wc_InitDsaKey(&key); // initialize DSA key
```

```
void wc_FreeDsaKey(
    DsaKey * key
)
```

This function frees a DsaKey object after it has been used.

Parameters:

- **key** pointer to the DsaKey structure to free

See: wc_FreeDsaKey

Return: none No returns.

Example

```
DsaKey key;
// initialize key, use for authentication
...
wc_FreeDsaKey(&key); // free DSA key
```

```
int wc_DsaSign(
    const byte * digest,
    byte * out,
    DsaKey * key,
    WC_RNG * rng
)
```

This function signs the input digest and stores the result in the output buffer, out.

Parameters:

- **digest** pointer to the hash to sign

- **out** pointer to the buffer in which to store the signature
- **key** pointer to the initialized DsaKey structure with which to generate the signature
- **rng** pointer to an initialized RNG to use with the signature generation

See: wc_DsaVerify

Return:

- 0 Returned on successfully signing the input digest
- MP_INIT_E may be returned if there is an error in processing the DSA signature.
- MP_READ_E may be returned if there is an error in processing the DSA signature.
- MP_CMP_E may be returned if there is an error in processing the DSA signature.
- MP_INVMOD_E may be returned if there is an error in processing the DSA signature.
- MP_EXPTMOD_E may be returned if there is an error in processing the DSA signature.
- MP_MOD_E may be returned if there is an error in processing the DSA signature.
- MP_MUL_E may be returned if there is an error in processing the DSA signature.
- MP_ADD_E may be returned if there is an error in processing the DSA signature.
- MP_MULMOD_E may be returned if there is an error in processing the DSA signature.
- MP_TO_E may be returned if there is an error in processing the DSA signature.
- MP_MEM may be returned if there is an error in processing the DSA signature.

Example

```
DsaKey key;
// initialize DSA key, load private Key
int ret;
WC_RNG rng;
wc_InitRng(&rng);
byte hash[] = { // initialize with hash digest };
byte signature[40]; // signature will be 40 bytes (320 bits)

ret = wc_DsaSign(hash, signature, &key, &rng);
if (ret != 0) {
    // error generating DSA signature
}
```

```
int wc_DsaVerify(
    const byte * digest,
    const byte * sig,
    DsaKey * key,
    int * answer
)
```

This function verifies the signature of a digest, given a private key. It stores whether the key properly verifies in the answer parameter, with 1 corresponding to a successful verification, and 0 corresponding to failed verification.

Parameters:

- **digest** pointer to the digest containing the subject of the signature
- **sig** pointer to the buffer containing the signature to verify

- **key** pointer to the initialized DsaKey structure with which to verify the signature
- **answer** pointer to an integer which will store whether the verification was successful

See: wc_DsaSign

Return:

- 0 Returned on successfully processing the verify request. Note: this does not mean that the signature is verified, only that the function succeeded
- MP_INIT_E may be returned if there is an error in processing the DSA signature.
- MP_READ_E may be returned if there is an error in processing the DSA signature.
- MP_CMP_E may be returned if there is an error in processing the DSA signature.
- MP_INVMOD_E may be returned if there is an error in processing the DSA signature.
- MP_EXPTMOD_E may be returned if there is an error in processing the DSA signature.
- MP_MOD_E may be returned if there is an error in processing the DSA signature.
- MP_MUL_E may be returned if there is an error in processing the DSA signature.
- MP_ADD_E may be returned if there is an error in processing the DSA signature.
- MP_MULMOD_E may be returned if there is an error in processing the DSA signature.
- MP_TO_E may be returned if there is an error in processing the DSA signature.
- MP_MEM may be returned if there is an error in processing the DSA signature.

Example

```
DsaKey key;
// initialize DSA key, load public Key

int ret;
int verified;
byte hash[] = { // initialize with hash digest };
byte signature[] = { // initialize with signature to verify };
ret = wc_DsaVerify(hash, signature, &key, &verified);
if (ret != 0) {
    // error processing verify request
} else if (answer == 0) {
    // invalid signature
}
```

```
int wc_DsaPublicKeyDecode(
    const byte * input,
    word32 * inOutIdx,
    DsaKey * key,
    word32 inSz
)
```

This function decodes a DER formatted certificate buffer containing a DSA public key, and stores the key in the given DsaKey structure. It also sets the inOutIdx parameter according to the length of the input read.

Parameters:

- **input** pointer to the buffer containing the DER formatted DSA public key

- **inOutIdx** pointer to an integer in which to store the final index of the certificate read
- **key** pointer to the DsaKey structure in which to store the public key
- **inSz** size of the input buffer

See:

- wc_InitDsaKey
- wc_DsaPrivateKeyDecode

Return:

- 0 Returned on successfully setting the public key for the DsaKey object
- ASN_PARSE_E Returned if there is an error in the encoding while reading the certificate buffer
- ASN_DH_KEY_E Returned if one of the DSA parameters is incorrectly formatted

Example

```
int ret, idx=0;

DsaKey key;
wc_InitDsaKey(&key);
byte derBuff[] = { // DSA public key};
ret = wc_DsaPublicKeyDecode(derBuff, &idx, &key, inSz);
if (ret != 0) {
    // error reading public key
}
```

```
int wc_DsaPrivateKeyDecode(
    const byte * input,
    word32 * inOutIdx,
    DsaKey * key,
    word32 inSz
)
```

This function decodes a DER formatted certificate buffer containing a DSA private key, and stores the key in the given DsaKey structure. It also sets the inOutIdx parameter according to the length of the input read.

Parameters:

- **input** pointer to the buffer containing the DER formatted DSA private key
- **inOutIdx** pointer to an integer in which to store the final index of the certificate read
- **key** pointer to the DsaKey structure in which to store the private key
- **inSz** size of the input buffer

See:

- wc_InitDsaKey
- wc_DsaPublicKeyDecode

Return:

- 0 Returned on successfully setting the private key for the DsaKey object
- ASN_PARSE_E Returned if there is an error in the encoding while reading the certificate buffer
- ASN_DH_KEY_E Returned if one of the DSA parameters is incorrectly formatted

Example

```
int ret, idx=0;

DsaKey key;
wc_InitDsaKey(&key);
byte derBuff[] = { // DSA private key };
ret = wc_DsaPrivateKeyDecode(derBuff, &idx, &key, inSz);
if (ret != 0) {
    // error reading private key
}
```

```
int wc_DsaKeyToDer(
    DsaKey * key,
    byte * output,
    word32 inLen
)
```

Convert DsaKey key to DER format, write to output (inLen), return bytes written.

Parameters:

- **key** Pointer to DsaKey structure to convert.
- **output** Pointer to output buffer for converted key.
- **inLen** Length of key input.

See:

- wc_InitDsaKey
- wc_FreeDsaKey
- wc_MakeDsaKey

Return:

- outLen Success, number of bytes written
- BAD_FUNC_ARG key or output are null or key->type is not DSA_PRIVATE.
- MEMORY_E Error allocating memory.

Example

```
DsaKey key;
WC_RNG rng;
int derSz;
```

```
int bufferSize = // Sufficient buffer size;
byte der[bufferSize];

wc_InitDsaKey(&key);
wc_InitRng(&rng);
wc_MakeDsaKey(&rng, &key);
derSz = wc_DsaKeyToDer(&key, der, bufferSize);
```

```
int wc_MakeDsaKey(
    WC_RNG * rng,
    DsaKey * dsa
)
```

Create a DSA key.

Parameters:

- **rng** Pointer to WC_RNG structure.
- **dsa** Pointer to DsaKey structure.

See:

- wc_InitDsaKey
- wc_FreeDsaKey
- wc_DsaSign

Return:

- MP_OKAY Success
- BAD_FUNC_ARG Either rng or dsa is null.
- MEMORY_E Couldn't allocate memory for buffer.
- MP_INIT_E Error initializing mp_int

Example

```
WC_RNG rng;
DsaKey dsa;
wc_InitRng(&rng);
wc_InitDsa(&dsa);
if(wc_MakeDsaKey(&rng, &dsa) != 0)
{
    // Error creating key
}
```

```
int wc_MakeDsaParameters(
    WC_RNG * rng,
    int modulus_size,
    DsaKey * dsa
)
```

FIPS 186-4 defines valid for modulus_size values as (1024, 160) (2048, 256) (3072, 256)

Parameters:

- **rng** pointer to wolfCrypt rng.
- **modulus_size** 1024, 2048, or 3072 are valid values.
- **dsa** Pointer to a DsaKey structure.

See:

- wc_MakeDsaKey
- wc_DsaKeyToDer
- wc_InitDsaKey

Return:

- 0 Success
- BAD_FUNC_ARG rng or dsa is null or modulus_size is invalid.
- MEMORY_E Error attempting to allocate memory.

Example

```
DsaKey key;
WC_RNG rng;
wc_InitDsaKey(&key);
wc_InitRng(&rng);
if(wc_MakeDsaParameters(&rng, 1024, &genKey) != 0)
{
    // Handle error
}
```

B.23 Algorithms - Diffie-Hellman

B.22.2.9 function wc_MakeDsaParameters

B.23.1 Functions

	Name
int	**wc_InitDhKey**(DhKey * key)This function initializes a Diffie-Hellman key for use in negotiating a secure secret key with the Diffie-Hellman exchange protocol.
void	**wc_FreeDhKey**(DhKey * key)This function frees a Diffie-Hellman key after it has been used to negotiate a secure secret key with the Diffie-Hellman exchange protocol.

	Name
int	**wc_DhGenerateKeyPair**(DhKey * key, WC_RNG * rng, byte * priv, word32 * privSz, byte * pub, word32 * pubSz)This function generates a public/private key pair based on the Diffie-Hellman public parameters, storing the private key in priv and the public key in pub. It takes an initialized Diffie-Hellman key and an initialized rng structure.
int	**wc_DhAgree**(DhKey * key, byte * agree, word32 * agreeSz, const byte * priv, word32 privSz, const byte * otherPub, word32 pubSz)This function generates an agreed upon secret key based on a local private key and a received public key. If completed on both sides of an exchange, this function generates an agreed upon secret key for symmetric communication. On successfully generating a shared secret key, the size of the secret key written will be stored in agreeSz.
int	**wc_DhKeyDecode**(const byte * input, word32 * inOutIdx, DhKey * key, word32)This function decodes a Diffie-Hellman key from the given input buffer containing the key in DER format. It stores the result in the DhKey structure.
int	**wc_DhSetKey**(DhKey * key, const byte * p, word32 pSz, const byte * g, word32 gSz)This function sets the key for a DhKey structure using the input private key parameters. Unlike wc_DhKeyDecode, this function does not require that the input key be formatted in DER format, and instead simply accepts the parsed input parameters p (prime) and g (base).
int	**wc_DhParamsLoad**(const byte * input, word32 inSz, byte * p, word32 * pInOutSz, byte * g, word32 * gInOutSz)This function loads the Diffie_Hellman parameters, p (prime) and g (base) out of the given input buffer, DER formatted.
const DhParams *	**wc_Dh_ffdhe2048_Get**(void)This function returns ... and requires that HAVE_FFDHE_2048 be defined.
const DhParams *	**wc_Dh_ffdhe3072_Get**(void)This function returns ... and requires that HAVE_FFDHE_3072 be defined.
const DhParams *	**wc_Dh_ffdhe4096_Get**(void)This function returns ... and requires that HAVE_FFDHE_4096 be defined.
const DhParams *	**wc_Dh_ffdhe6144_Get**(void)This function returns ... and requires that HAVE_FFDHE_6144 be defined.

	Name
const DhParams *	**wc_Dh_ffdhe8192_Get**(void)This function returns ... and requires that HAVE_FFDHE_8192 be defined.
int	**wc_DhCheckKeyPair**(DhKey * key, const byte * pub, word32 pubSz, const byte * priv, word32 privSz)Checks DH keys for pair_wise consistency per process in SP 800_56Ar3, section 5.6.2.1.4, method (b) for FFC.
int	**wc_DhCheckPrivKey**(DhKey * key, const byte * priv, word32 pubSz)Check DH private key for invalid numbers.
int	**wc_DhCheckPrivKey_ex**(DhKey * key, const byte * priv, word32 pubSz, const byte * prime, word32 primeSz)
int	**wc_DhCheckPubKey**(DhKey * key, const byte * pub, word32 pubSz)
int	**wc_DhCheckPubKey_ex**(DhKey * key, const byte * pub, word32 pubSz, const byte * prime, word32 primeSz)
int	**wc_DhExportParamsRaw**(DhKey * dh, byte * p, word32 * pSz, byte * q, word32 * qSz, byte * g, word32 * gSz)
int	**wc_DhGenerateParams**(WC_RNG * rng, int modSz, DhKey * dh)
int	**wc_DhSetCheckKey**(DhKey * key, const byte * p, word32 pSz, const byte * g, word32 gSz, const byte * q, word32 qSz, int trusted, WC_RNG * rng)
int	**wc_DhSetKey_ex**(DhKey * key, const byte * p, word32 pSz, const byte * g, word32 gSz, const byte * q, word32 qSz)

B.23.2 Functions Documentation

```
int wc_InitDhKey(
    DhKey * key
)
```

This function initializes a Diffie-Hellman key for use in negotiating a secure secret key with the Diffie-Hellman exchange protocol.

Parameters:

- **key** pointer to the DhKey structure to initialize for use with secure key exchanges

See:

- wc_FreeDhKey
- wc_DhGenerateKeyPair

Return: none No returns.

Example

```
DhKey key;
wc_InitDhKey(&key); // initialize DH key
```

```
void wc_FreeDhKey(
    DhKey * key
)
```

This function frees a Diffie-Hellman key after it has been used to negotiate a secure secret key with the Diffie-Hellman exchange protocol.

Parameters:

- **key** pointer to the DhKey structure to free

See: wc_InitDhKey

Return: none No returns.

Example

```
DhKey key;
// initialize key, perform key exchange

wc_FreeDhKey(&key); // free DH key to avoid memory leaks
```

```
int wc_DhGenerateKeyPair(
    DhKey * key,
    WC_RNG * rng,
    byte * priv,
    word32 * privSz,
    byte * pub,
    word32 * pubSz
)
```

This function generates a public/private key pair based on the Diffie-Hellman public parameters, storing the private key in priv and the public key in pub. It takes an initialized Diffie-Hellman key and an initialized rng structure.

Parameters:

- **key** pointer to the DhKey structure from which to generate the key pair
- **rng** pointer to an initialized random number generator (rng) with which to generate the keys
- **priv** pointer to a buffer in which to store the private key
- **privSz** will store the size of the private key written to priv
- **pub** pointer to a buffer in which to store the public key
- **pubSz** will store the size of the private key written to pub

See:

- wc_InitDhKey
- wc_DhSetKey
- wc_DhKeyDecode

Return:

- BAD_FUNC_ARG Returned if there is an error parsing one of the inputs to this function
- RNG_FAILURE_E Returned if there is an error generating a random number using rng
- MP_INIT_E May be returned if there is an error in the math library while generating the public key
- MP_READ_E May be returned if there is an error in the math library while generating the public key
- MP_EXPTMOD_E May be returned if there is an error in the math library while generating the public key
- MP_TO_E May be returned if there is an error in the math library while generating the public key

Example

```
DhKey key;
int ret;
byte priv[256];
byte pub[256];
word32 privSz, pubSz;

wc_InitDhKey(&key); // initialize key
// Set DH parameters using wc_DhSetKey or wc_DhKeyDecode
WC_RNG rng;
wc_InitRng(&rng); // initialize rng
ret = wc_DhGenerateKeyPair(&key, &rng, priv, &privSz, pub, &pubSz);
```

```
int wc_DhAgree(
    DhKey * key,
    byte * agree,
    word32 * agreeSz,
    const byte * priv,
    word32 privSz,
    const byte * otherPub,
    word32 pubSz
)
```

This function generates an agreed upon secret key based on a local private key and a received public key. If completed on both sides of an exchange, this function generates an agreed upon secret key for symmetric communication. On successfully generating a shared secret key, the size of the secret key written will be stored in agreeSz.

Parameters:

- **key** pointer to the DhKey structure to use to compute the shared key

- **agree** pointer to the buffer in which to store the secret key
- **agreeSz** will hold the size of the secret key after successful generation
- **priv** pointer to the buffer containing the local secret key
- **privSz** size of the local secret key
- **otherPub** pointer to a buffer containing the received public key
- **pubSz** size of the received public key

See: wc_DhGenerateKeyPair

Return:

- 0 Returned on successfully generating an agreed upon secret key
- MP_INIT_E May be returned if there is an error while generating the shared secret key
- MP_READ_E May be returned if there is an error while generating the shared secret key
- MP_EXPTMOD_E May be returned if there is an error while generating the shared secret key
- MP_TO_E May be returned if there is an error while generating the shared secret key

Example

```
DhKey key;
int ret;
byte priv[256];
byte agree[256];
word32 agreeSz;

// initialize key, set key prime and base
// wc_DhGenerateKeyPair -- store private key in priv
byte pub[] = { // initialized with the received public key };
ret = wc_DhAgree(&key, agree, &agreeSz, priv, sizeof(priv), pub,
sizeof(pub));
if ( ret != 0 ) {
    // error generating shared key
}
```

```
int wc_DhKeyDecode(
    const byte * input,
    word32 * inOutIdx,
    DhKey * key,
    word32
)
```

This function decodes a Diffie-Hellman key from the given input buffer containing the key in DER format. It stores the result in the DhKey structure.

Parameters:

- **input** pointer to the buffer containing the DER formatted Diffie-Hellman key
- **inOutIdx** pointer to an integer in which to store the index parsed to while decoding the key
- **key** pointer to the DhKey structure to initialize with the input key
- **inSz** length of the input buffer. Gives the max length that may be read

See: wc_DhSetKey

Return:

- 0 Returned on successfully decoding the input key
- ASN_PARSE_E Returned if there is an error parsing the sequence of the input
- ASN_DH_KEY_E Returned if there is an error reading the private key parameters from the parsed input

Example

```
DhKey key;
word32 idx = 0;

byte keyBuff[1024];
// initialize with DER formatted key
wc_DhKeyInit(&key);
ret = wc_DhKeyDecode(keyBuff, &idx, &key, sizeof(keyBuff));

if ( ret != 0 ) {
    // error decoding key
}
```

```
int wc_DhSetKey(
    DhKey * key,
    const byte * p,
    word32 pSz,
    const byte * g,
    word32 gSz
)
```

This function sets the key for a DhKey structure using the input private key parameters. Unlike wc_DhKeyDecode, this function does not require that the input key be formatted in DER format, and instead simply accepts the parsed input parameters p (prime) and g (base).

Parameters:

- **key** pointer to the DhKey structure on which to set the key
- **p** pointer to the buffer containing the prime for use with the key
- **pSz** length of the input prime
- **g** pointer to the buffer containing the base for use with the key
- **gSz** length of the input base

See: wc_DhKeyDecode

Return:

- 0 Returned on successfully setting the key
- BAD_FUNC_ARG Returned if any of the input parameters evaluate to NULL
- MP_INIT_E Returned if there is an error initializing the key parameters for storage
- ASN_DH_KEY_E Returned if there is an error reading in the DH key parameters p and g

Example

```
DhKey key;

byte p[] = { // initialize with prime };
byte g[] = { // initialize with base };
wc_DhKeyInit(&key);
ret = wc_DhSetKey(key, p, sizeof(p), g, sizeof(g));

if ( ret != 0 ) {
    // error setting key
}
```

```
int wc_DhParamsLoad(
    const byte * input,
    word32 inSz,
    byte * p,
    word32 * pInOutSz,
    byte * g,
    word32 * gInOutSz
)
```

This function loads the Diffie-Hellman parameters, p (prime) and g (base) out of the given input buffer, DER formatted.

Parameters:

- **input** pointer to a buffer containing a DER formatted Diffie-Hellman certificate to parse
- **inSz** size of the input buffer
- **p** pointer to a buffer in which to store the parsed prime
- **pInOutSz** pointer to a word32 object containing the available size in the p buffer. Will be overwritten with the number of bytes written to the buffer after completing the function call
- **g** pointer to a buffer in which to store the parsed base
- **gInOutSz** pointer to a word32 object containing the available size in the g buffer. Will be overwritten with the number of bytes written to the buffer after completing the function call

See:

- wc_DhSetKey
- wc_DhKeyDecode

Return:

- 0 Returned on successfully extracting the DH parameters
- ASN_PARSE_E Returned if an error occurs while parsing the DER formatted DH certificate
- BUFFER_E Returned if there is inadequate space in p or g to store the parsed parameters

Example

```
byte dhCert[] = { initialize with DER formatted certificate };
byte p[MAX_DH_SIZE];
byte g[MAX_DH_SIZE];
word32 pSz = MAX_DH_SIZE;
word32 gSz = MAX_DH_SIZE;

ret = wc_DhParamsLoad(dhCert, sizeof(dhCert), p, &pSz, g, &gSz);
if ( ret != 0 ) {
    // error parsing inputs
}
```

```
const DhParams * wc_Dh_ffdhe2048_Get(
    void
)
```

This function returns ... and requires that HAVE_FFDHE_2048 be defined.

See:

- wc_Dh_ffdhe3072_Get
- wc_Dh_ffdhe4096_Get
- wc_Dh_ffdhe6144_Get
- wc_Dh_ffdhe8192_Get

```
const DhParams * wc_Dh_ffdhe3072_Get(
    void
)
```

This function returns ... and requires that HAVE_FFDHE_3072 be defined.

See:

- wc_Dh_ffdhe2048_Get
- wc_Dh_ffdhe4096_Get
- wc_Dh_ffdhe6144_Get
- wc_Dh_ffdhe8192_Get

```
const DhParams * wc_Dh_ffdhe4096_Get(
    void
)
```

This function returns ... and requires that HAVE_FFDHE_4096 be defined.

See:

- wc_Dh_ffdhe2048_Get
- wc_Dh_ffdhe3072_Get

- wc_Dh_ffdhe6144_Get
- wc_Dh_ffdhe8192_Get

```
const DhParams * wc_Dh_ffdhe6144_Get(
    void
)
```

This function returns ... and requires that HAVE_FFDHE_6144 be defined.

See:

- wc_Dh_ffdhe2048_Get
- wc_Dh_ffdhe3072_Get
- wc_Dh_ffdhe4096_Get
- wc_Dh_ffdhe8192_Get

```
const DhParams * wc_Dh_ffdhe8192_Get(
    void
)
```

This function returns ... and requires that HAVE_FFDHE_8192 be defined.

See:

- wc_Dh_ffdhe2048_Get
- wc_Dh_ffdhe3072_Get
- wc_Dh_ffdhe4096_Get
- wc_Dh_ffdhe6144_Get

```
int wc_DhCheckKeyPair(
    DhKey * key,
    const byte * pub,
    word32 pubSz,
    const byte * priv,
    word32 privSz
)
```

Checks DH keys for pair-wise consistency per process in SP 800-56Ar3, section 5.6.2.1.4, method (b) for FFC.

```
int wc_DhCheckPrivKey(
    DhKey * key,
    const byte * priv,
    word32 pubSz
)
```

Check DH private key for invalid numbers.

```
int wc_DhCheckPrivKey_ex(
    DhKey * key,
    const byte * priv,
    word32 pubSz,
    const byte * prime,
    word32 primeSz
)
```

```
int wc_DhCheckPubKey(
    DhKey * key,
    const byte * pub,
    word32 pubSz
)
```

```
int wc_DhCheckPubKey_ex(
    DhKey * key,
    const byte * pub,
    word32 pubSz,
    const byte * prime,
    word32 primeSz
)
```

```
int wc_DhExportParamsRaw(
    DhKey * dh,
    byte * p,
    word32 * pSz,
    byte * q,
    word32 * qSz,
    byte * g,
    word32 * gSz
)
```

```
int wc_DhGenerateParams(
    WC_RNG * rng,
    int modSz,
    DhKey * dh
)
```

```
int wc_DhSetCheckKey(
    DhKey * key,
    const byte * p,
    word32 pSz,
    const byte * g,
    word32 gSz,
    const byte * q,
    word32 qSz,
    int trusted,
    WC_RNG * rng
)
```

```
int wc_DhSetKey_ex(
    DhKey * key,
    const byte * p,
    word32 pSz,
    const byte * g,
    word32 gSz,
    const byte * q,
    word32 qSz
)
```

B.24 Algorithms - ECC

B.23.2.21 function wc_DhSetKey_ex

B.24.1 Functions

	Name
int	**wc_ecc_make_key**(WC_RNG * rng, int keysize, ecc_key * key)This function generates a new ecc_key and stores it in key.
int	**wc_ecc_make_key_ex**(WC_RNG * rng, int keysize, ecc_key * key, int curve_id)This function generates a new ecc_key and stores it in key.
int	**wc_ecc_check_key**(ecc_key * key)Perform sanity checks on ecc key validity.
void	**wc_ecc_key_free**(ecc_key * key)This function frees an ecc_key key after it has been used.
int	**wc_ecc_shared_secret**(ecc_key * private_key, ecc_key * public_key, byte * out, word32 * outlen)This function generates a new secret key using a local private key and a received public key. It stores this shared secret key in the buffer out and updates outlen to hold the number of bytes written to the output buffer.

	Name
int	**wc_ecc_shared_secret_ex**(ecc_key * private_key, ecc_point * point, byte * out, word32 * outlen)Create an ECC shared secret between private key and public point.
int	**wc_ecc_sign_hash**(const byte * in, word32 inlen, byte * out, word32 * outlen, WC_RNG * rng, ecc_key * key)This function signs a message digest using an ecc_key object to guarantee authenticity.
int	**wc_ecc_sign_hash_ex**(const byte * in, word32 inlen, WC_RNG * rng, ecc_key * key, mp_int * r, mp_int * s)Sign a message digest.
int	**wc_ecc_verify_hash**(const byte * sig, word32 siglen, const byte * hash, word32 hashlen, int * stat, ecc_key * key)This function verifies the ECC signature of a hash to ensure authenticity. It returns the answer through stat, with 1 corresponding to a valid signature, and 0 corresponding to an invalid signature.
int	**wc_ecc_verify_hash_ex**(mp_int * r, mp_int * s, const byte * hash, word32 hashlen, int * stat, ecc_key * key)Verify an ECC signature. Result is written to stat. 1 is valid, 0 is invalid. Note: Do not use the return value to test for valid. Only use stat.
int	**wc_ecc_init**(ecc_key * key)This function initializes an ecc_key object for future use with message verification or key negotiation.
int	**wc_ecc_init_ex**(ecc_key * key, void * heap, int devId)This function initializes an ecc_key object for future use with message verification or key negotiation.
ecc_key *	**wc_ecc_key_new**(void * heap)This function uses a user defined heap and allocates space for the key structure.
int	**wc_ecc_free**(ecc_key * key)This function frees an ecc_key object after it has been used.
void	**wc_ecc_fp_free**(void)This function frees the fixed_point cache, which can be used with ecc to speed up computation times. To use this functionality, FP_ECC (fixed_point ecc), should be defined. Threaded applications should call this function before exiting the thread.
int	**wc_ecc_is_valid_idx**(int n)Checks if an ECC idx is valid.
ecc_point *	**wc_ecc_new_point**(void)Allocate a new ECC point.
void	**wc_ecc_del_point**(ecc_point * p)Free an ECC point from memory.
int	**wc_ecc_copy_point**(ecc_point * p, ecc_point * r)Copy the value of one point to another one.

	Name
int	**wc_ecc_cmp_point**(ecc_point * a, ecc_point * b)Compare the value of a point with another one.
int	**wc_ecc_point_is_at_infinity**(ecc_point * p)Checks if a point is at infinity. Returns 1 if point is at infinity, 0 if not, < 0 on error.
int	**wc_ecc_mulmod**(mp_int * k, ecc_point * G, ecc_point * R, mp_int * a, mp_int * modulus, int map)Perform ECC Fixed Point multiplication.
int	**wc_ecc_export_x963**(ecc_key * key, byte * out, word32 * outLen)This function exports the ECC key from the ecc_key structure, storing the result in out. The key will be stored in ANSI X9.63 format. It stores the bytes written to the output buffer in outLen.
int	**wc_ecc_export_x963_ex**(ecc_key * key, byte * out, word32 * outLen, int compressed)This function exports the ECC key from the ecc_key structure, storing the result in out. The key will be stored in ANSI X9.63 format. It stores the bytes written to the output buffer in outLen. This function allows the additional option of compressing the certificate through the compressed parameter. When this parameter is true, the key will be stored in ANSI X9.63 compressed format.
int	**wc_ecc_import_x963**(const byte * in, word32 inLen, ecc_key * key)This function imports a public ECC key from a buffer containing the key stored in ANSI X9.63 format. This function will handle both compressed and uncompressed keys, as long as compressed keys are enabled at compile time through the HAVE_COMP_KEY option.
int	**wc_ecc_import_private_key**(const byte * priv, word32 privSz, const byte * pub, word32 pubSz, ecc_key * key)This function imports a public/private ECC key pair from a buffer containing the raw private key, and a second buffer containing the ANSI X9.63 formatted public key. This function will handle both compressed and uncompressed keys, as long as compressed keys are enabled at compile time through the HAVE_COMP_KEY option.
int	**wc_ecc_rs_to_sig**(const char * r, const char * s, byte * out, word32 * outlen)This function converts the R and S portions of an ECC signature into a DER-encoded ECDSA signature. This function also stores the length written to the output buffer, out, in outlen.

	Name
int	**wc_ecc_import_raw**(ecc_key * key, const char * qx, const char * qy, const char * d, const char * curveName)This function fills an ecc_key structure with the raw components of an ECC signature.
int	**wc_ecc_export_private_only**(ecc_key * key, byte * out, word32 * outLen)This function exports only the private key from an ecc_key structure. It stores the private key in the buffer out, and sets the bytes written to this buffer in outLen.
int	**wc_ecc_export_point_der**(const int curve_idx, ecc_point * point, byte * out, word32 * outLen)Export point to der.
int	**wc_ecc_import_point_der**(byte * in, word32 inLen, const int curve_idx, ecc_point * point)Import point from der format.
int	**wc_ecc_size**(ecc_key * key)This function returns the key size of an ecc_key structure in octets.
int	**wc_ecc_sig_size_calc**(int sz)This function returns the worst case size for an ECC signature, given by: (keySz * 2) + SIG_HEADER_SZ + ECC_MAX_PAD_SZ. The actual signature size can be computed with wc_ecc_sign_hash.
int	**wc_ecc_sig_size**(ecc_key * key)This function returns the worst case size for an ECC signature, given by: (keySz * 2) + SIG_HEADER_SZ + ECC_MAX_PAD_SZ. The actual signature size can be computed with wc_ecc_sign_hash.
ecEncCtx *	**wc_ecc_ctx_new**(int flags, WC_RNG * rng)This function allocates and initializes space for a new ECC context object to allow secure message exchange with ECC.
void	**wc_ecc_ctx_free**(ecEncCtx *)This function frees the ecEncCtx object used for encrypting and decrypting messages.
int	**wc_ecc_ctx_reset**(ecEncCtx * ctx, WC_RNG * rng)This function resets an ecEncCtx structure to avoid having to free and allocate a new context object.
int	**wc_ecc_ctx_set_algo**(ecEncCtx * ctx, byte encAlgo, byte kdfAlgo, byte macAlgo)This function can optionally be called after wc_ecc_ctx_new. It sets the encryption, KDF, and MAC algorithms into an ecEncCtx object.
const byte *	**wc_ecc_ctx_get_own_salt**(ecEncCtx *)This function returns the salt of an ecEncCtx object. This function should only be called when the ecEncCtx's state is ecSRV_INIT or ecCLI_INIT.

	Name
int	**wc_ecc_ctx_set_peer_salt**(ecEncCtx * ctx, const byte * salt)This function sets the peer salt of an ecEncCtx object.
int	**wc_ecc_ctx_set_kdf_salt**(ecEncCtx * ctx, const byte * salt, word32 sz)This function sets the salt pointer and length to use with KDF into the ecEncCtx object.
int	**wc_ecc_ctx_set_info**(ecEncCtx * ctx, const byte * info, int sz)This function can optionally be called before or after wc_ecc_ctx_set_peer_salt. It sets optional information for an ecEncCtx object.
int	**wc_ecc_encrypt**(ecc_key * privKey, ecc_key * pubKey, const byte * msg, word32 msgSz, byte * out, word32 * outSz, ecEncCtx * ctx)This function encrypts the given input message from msg to out. This function takes an optional ctx object as parameter. When supplied, encryption proceeds based on the ecEncCtx's encAlgo, kdfAlgo, and macAlgo. If ctx is not supplied, processing completes with the default algorithms, ecAES_128_CBC, ecHKDF_SHA256 and ecHMAC_SHA256. This function requires that the messages are padded according to the encryption type specified by ctx.
int	**wc_ecc_encrypt_ex**(ecc_key * privKey, ecc_key * pubKey, const byte * msg, word32 msgSz, byte * out, word32 * outSz, ecEncCtx * ctx, int compressed)This function encrypts the given input message from msg to out. This function takes an optional ctx object as parameter. When supplied, encryption proceeds based on the ecEncCtx's encAlgo, kdfAlgo, and macAlgo. If ctx is not supplied, processing completes with the default algorithms, ecAES_128_CBC, ecHKDF_SHA256 and ecHMAC_SHA256. This function requires that the messages are padded according to the encryption type specified by ctx.

	Name
int	**wc_ecc_decrypt**(ecc_key * privKey, ecc_key * pubKey, const byte * msg, word32 msgSz, byte * out, word32 * outSz, ecEncCtx * ctx)This function decrypts the ciphertext from msg to out. This function takes an optional ctx object as parameter. When supplied, encryption proceeds based on the ecEncCtx's encAlgo, kdfAlgo, and macAlgo. If ctx is not supplied, processing completes with the default algorithms, ecAES_128_CBC, ecHKDF_SHA256 and ecHMAC_SHA256. This function requires that the messages are padded according to the encryption type specified by ctx.
int	**wc_ecc_set_nonblock**(ecc_key * key, ecc_nb_ctx_t * ctx)Enable ECC support for non_blocking operations. Supported for Single Precision (SP) math with the following build options: WOLFSSL_SP_NONBLOCK WOLFSSL_SP_SMALL WOLFSSL_SP_NO_MALLOC WC_ECC_NONBLOCK.
int	**wc_ecc_set_curve**(ecc_key * key, int keysize, int curve_id)Compare a curve which has larger key than specified size or the curve matched curve ID, set a curve with smaller key size to the key.

B.24.2 Functions Documentation

```
int wc_ecc_make_key(
    WC_RNG * rng,
    int keysize,
    ecc_key * key
)
```

This function generates a new ecc_key and stores it in key.

Parameters:

- **rng** pointer to an initialized RNG object with which to generate the key
- **keysize** desired length for the ecc_key
- **key** pointer to the ecc_key for which to generate a key

See:

- wc_ecc_init
- wc_ecc_shared_secret

Return:

- 0 Returned on success.
- ECC_BAD_ARG_E Returned if rng or key evaluate to NULL

- BAD_FUNC_ARG Returned if the specified key size is not in the correct range of supported keys
- MEMORY_E Returned if there is an error allocating memory while computing the ecc key
- MP_INIT_E may be returned if there is an error while computing the ecc key
- MP_READ_E may be returned if there is an error while computing the ecc key
- MP_CMP_E may be returned if there is an error while computing the ecc key
- MP_INVMOD_E may be returned if there is an error while computing the ecc key
- MP_EXPTMOD_E may be returned if there is an error while computing the ecc key
- MP_MOD_E may be returned if there is an error while computing the ecc key
- MP_MUL_E may be returned if there is an error while computing the ecc key
- MP_ADD_E may be returned if there is an error while computing the ecc key
- MP_MULMOD_E may be returned if there is an error while computing the ecc key
- MP_TO_E may be returned if there is an error while computing the ecc key
- MP_MEM may be returned if there is an error while computing the ecc key

Example

```
ecc_key key;
wc_ecc_init(&key);
WC_RNG rng;
wc_InitRng(&rng);
wc_ecc_make_key(&rng, 32, &key); // initialize 32 byte ecc key
```

```
int wc_ecc_make_key_ex(
    WC_RNG * rng,
    int keysize,
    ecc_key * key,
    int curve_id
)
```

This function generates a new ecc_key and stores it in key.

Parameters:

- **key** Pointer to store the created key.
- **keysize** size of key to be created in bytes, set based on curveId
- **rng** Rng to be used in key creation
- **curve_id** Curve to use for key

See:

- wc_ecc_make_key
- wc_ecc_get_curve_size_from_id

Return:

- 0 Returned on success.
- ECC_BAD_ARG_E Returned if rng or key evaluate to NULL
- BAD_FUNC_ARG Returned if the specified key size is not in the correct range of supported keys
- MEMORY_E Returned if there is an error allocating memory while computing the ecc key
- MP_INIT_E may be returned if there is an error while computing the ecc key

- MP_READ_E may be returned if there is an error while computing the ecc key
- MP_CMP_E may be returned if there is an error while computing the ecc key
- MP_INVMOD_E may be returned if there is an error while computing the ecc key
- MP_EXPTMOD_E may be returned if there is an error while computing the ecc key
- MP_MOD_E may be returned if there is an error while computing the ecc key
- MP_MUL_E may be returned if there is an error while computing the ecc key
- MP_ADD_E may be returned if there is an error while computing the ecc key
- MP_MULMOD_E may be returned if there is an error while computing the ecc key
- MP_TO_E may be returned if there is an error while computing the ecc key
- MP_MEM may be returned if there is an error while computing the ecc key

Example

```
ecc_key key;
int ret;
WC_RNG rng;
wc_ecc_init(&key);
wc_InitRng(&rng);
int curveId = ECC_SECP521R1;
int keySize = wc_ecc_get_curve_size_from_id(curveId);
ret = wc_ecc_make_key_ex(&rng, keySize, &key, curveId);
if (ret != MP_OKAY) {
    // error handling
}
```

```
int wc_ecc_check_key(
    ecc_key * key
)
```

Perform sanity checks on ecc key validity.

Parameters:

- **key** Pointer to key to check.

See: wc_ecc_point_is_at_infinity

Return:

- MP_OKAY Success, key is OK.
- BAD_FUNC_ARG Returns if key is NULL.
- ECC_INF_E Returns if wc_ecc_point_is_at_infinity returns 1.

Example

```
ecc_key key;
WC_RNG rng;
int check_result;
wc_ecc_init(&key);
wc_InitRng(&rng);
wc_ecc_make_key(&rng, 32, &key);
```

```
check_result = wc_ecc_check_key(&key);

if (check_result == MP_OKAY)
{
    // key check succeeded
}
else
{
    // key check failed
}
```

```
void wc_ecc_key_free(
    ecc_key * key
)
```

This function frees an ecc_key key after it has been used.

Parameters:

- **key** pointer to the ecc_key structure to free

See:

- wc_ecc_key_new
- wc_ecc_init_ex

Example

```
// initialize key and perform ECC operations
...
wc_ecc_key_free(&key);
```

```
int wc_ecc_shared_secret(
    ecc_key * private_key,
    ecc_key * public_key,
    byte * out,
    word32 * outlen
)
```

This function generates a new secret key using a local private key and a received public key. It stores this shared secret key in the buffer out and updates outlen to hold the number of bytes written to the output buffer.

Parameters:

- **private_key** pointer to the ecc_key structure containing the local private key
- **public_key** pointer to the ecc_key structure containing the received public key
- **out** pointer to an output buffer in which to store the generated shared secret key

- **outlen** pointer to the word32 object containing the length of the output buffer. Will be overwritten with the length written to the output buffer upon successfully generating a shared secret key

See:

- wc_ecc_init
- wc_ecc_make_key

Return:

- 0 Returned upon successfully generating a shared secret key
- BAD_FUNC_ARG Returned if any of the input parameters evaluate to NULL
- ECC_BAD_ARG_E Returned if the type of the private key given as argument, private_key, is not ECC_PRIVATEKEY, or if the public and private key types (given by ecc->dp) are not equivalent
- MEMORY_E Returned if there is an error generating a new ecc point
- BUFFER_E Returned if the generated shared secret key is too long to store in the provided buffer
- MP_INIT_E may be returned if there is an error while computing the shared key
- MP_READ_E may be returned if there is an error while computing the shared key
- MP_CMP_E may be returned if there is an error while computing the shared key
- MP_INVMOD_E may be returned if there is an error while computing the shared key
- MP_EXPTMOD_E may be returned if there is an error while computing the shared key
- MP_MOD_E may be returned if there is an error while computing the shared key
- MP_MUL_E may be returned if there is an error while computing the shared key
- MP_ADD_E may be returned if there is an error while computing the shared key
- MP_MULMOD_E may be returned if there is an error while computing the shared key
- MP_TO_E may be returned if there is an error while computing the shared key
- MP_MEM may be returned if there is an error while computing the shared key

Example

```
ecc_key priv, pub;
WC_RNG rng;
byte secret[1024]; // can hold 1024 byte shared secret key
word32 secretSz = sizeof(secret);
int ret;

wc_InitRng(&rng); // initialize rng
wc_ecc_init(&priv); // initialize key
wc_ecc_make_key(&rng, 32, &priv); // make public/private key pair
// receive public key, and initialise into pub
ret = wc_ecc_shared_secret(&priv, &pub, secret, &secretSz);
// generate secret key
if ( ret != 0 ) {
    // error generating shared secret key
}
```

```
int wc_ecc_shared_secret_ex(
    ecc_key * private_key,
    ecc_point * point,
    byte * out,
```

```
    word32 * outlen
)
```

Create an ECC shared secret between private key and public point.

Parameters:

- **private_key** The private ECC key.
- **point** The point to use (public key).
- **out** Output destination of the shared secret. Conforms to EC-DH from ANSI X9.63.
- **outlen** Input the max size and output the resulting size of the shared secret.

See: wc_ecc_verify_hash_ex

Return:

- MP_OKAY Indicates success.
- BAD_FUNC_ARG Error returned when any arguments are null.
- ECC_BAD_ARG_E Error returned if private_key->type is not ECC_PRIVATEKEY or private_key->idx fails to validate.
- BUFFER_E Error when outlen is too small.
- MEMORY_E Error to create a new point.
- MP_VAL possible when an initialization failure occurs.
- MP_MEM possible when an initialization failure occurs.

Example

```
ecc_key key;
ecc_point* point;
byte shared_secret[];
int secret_size;
int result;

point = wc_ecc_new_point();

result = wc_ecc_shared_secret_ex(&key, point,
&shared_secret, &secret_size);

if (result != MP_OKAY)
{
    // Handle error
}
```

```
int wc_ecc_sign_hash(
    const byte * in,
    word32 inlen,
    byte * out,
    word32 * outlen,
    WC_RNG * rng,
    ecc_key * key
)
```

This function signs a message digest using an ecc_key object to guarantee authenticity.

Parameters:

- **in** pointer to the buffer containing the message hash to sign
- **inlen** length of the message hash to sign
- **out** buffer in which to store the generated signature
- **outlen** max length of the output buffer. Will store the bytes written to out upon successfully generating a message signature
- **key** pointer to a private ECC key with which to generate the signature

See: wc_ecc_verify_hash

Return:

- 0 Returned upon successfully generating a signature for the message digest
- BAD_FUNC_ARG Returned if any of the input parameters evaluate to NULL, or if the output buffer is too small to store the generated signature
- ECC_BAD_ARG_E Returned if the input key is not a private key, or if the ECC OID is invalid
- RNG_FAILURE_E Returned if the rng cannot successfully generate a satisfactory key
- MP_INIT_E may be returned if there is an error while computing the message signature
- MP_READ_E may be returned if there is an error while computing the message signature
- MP_CMP_E may be returned if there is an error while computing the message signature
- MP_INVMOD_E may be returned if there is an error while computing the message signature
- MP_EXPTMOD_E may be returned if there is an error while computing the message signature
- MP_MOD_E may be returned if there is an error while computing the message signature
- MP_MUL_E may be returned if there is an error while computing the message signature
- MP_ADD_E may be returned if there is an error while computing the message signature
- MP_MULMOD_E may be returned if there is an error while computing the message signature
- MP_TO_E may be returned if there is an error while computing the message signature
- MP_MEM may be returned if there is an error while computing the message signature

Example

```
ecc_key key;
WC_RNG rng;
int ret, sigSz;

byte sig[512]; // will hold generated signature
sigSz = sizeof(sig);
byte digest[] = { // initialize with message hash };
wc_InitRng(&rng); // initialize rng
wc_ecc_init(&key); // initialize key
wc_ecc_make_key(&rng, 32, &key); // make public/private key pair
ret = wc_ecc_sign_hash(digest, sizeof(digest), sig, &sigSz, &key);
if ( ret != 0 ) {
    // error generating message signature
}
```

```
int wc_ecc_sign_hash_ex(
    const byte * in,
```

```
    word32 inlen,
    WC_RNG * rng,
    ecc_key * key,
    mp_int * r,
    mp_int * s
)
```

Sign a message digest.

Parameters:

- **in** The message digest to sign.
- **inlen** The length of the digest.
- **rng** Pointer to WC_RNG struct.
- **key** A private ECC key.
- **r** The destination for r component of the signature.
- **s** The destination for s component of the signature.

See: wc_ecc_verify_hash_ex

Return:

- MP_OKAY Returned upon successfully generating a signature for the message digest
- ECC_BAD_ARG_E Returned if the input key is not a private key, or if the ECC IDX is invalid, or if any of the input parameters evaluate to NULL, or if the output buffer is too small to store the generated signature
- RNG_FAILURE_E Returned if the rng cannot successfully generate a satisfactory key
- MP_INIT_E may be returned if there is an error while computing the message signature
- MP_READ_E may be returned if there is an error while computing the message signature
- MP_CMP_E may be returned if there is an error while computing the message signature
- MP_INVMOD_E may be returned if there is an error while computing the message signature
- MP_EXPTMOD_E may be returned if there is an error while computing the message signature
- MP_MOD_E may be returned if there is an error while computing the message signature
- MP_MUL_E may be returned if there is an error while computing the message signature
- MP_ADD_E may be returned if there is an error while computing the message signature
- MP_MULMOD_E may be returned if there is an error while computing the message signature
- MP_TO_E may be returned if there is an error while computing the message signature
- MP_MEM may be returned if there is an error while computing the message signature

Example

```
ecc_key key;
WC_RNG rng;
int ret, sigSz;
mp_int r; // destination for r component of signature.
mp_int s; // destination for s component of signature.

byte sig[512]; // will hold generated signature
sigSz = sizeof(sig);
byte digest[] = { initialize with message hash };
wc_InitRng(&rng); // initialize rng
wc_ecc_init(&key); // initialize key
mp_init(&r); // initialize r component
```

```
mp_init(&s); // initialize s component
wc_ecc_make_key(&rng, 32, &key); // make public/private key pair
ret = wc_ecc_sign_hash_ex(digest, sizeof(digest), &rng, &key, &r, &s);

if ( ret != MP_OKAY ) {
    // error generating message signature
}
```

```
int wc_ecc_verify_hash(
    const byte * sig,
    word32 siglen,
    const byte * hash,
    word32 hashlen,
    int * stat,
    ecc_key * key
)
```

This function verifies the ECC signature of a hash to ensure authenticity. It returns the answer through stat, with 1 corresponding to a valid signature, and 0 corresponding to an invalid signature.

Parameters:

- **sig** pointer to the buffer containing the signature to verify
- **siglen** length of the signature to verify
- **hash** pointer to the buffer containing the hash of the message verified
- **hashlen** length of the hash of the message verified
- **stat** pointer to the result of the verification. 1 indicates the message was successfully verified
- **key** pointer to a public ECC key with which to verify the signature

See:

- wc_ecc_sign_hash
- wc_ecc_verify_hash_ex

Return:

- 0 Returned upon successfully performing the signature verification. Note: This does not mean that the signature is verified. The authenticity information is stored instead in stat
- BAD_FUNC_ARG Returned any of the input parameters evaluate to NULL
- MEMORY_E Returned if there is an error allocating memory
- MP_INIT_E may be returned if there is an error while computing the message signature
- MP_READ_E may be returned if there is an error while computing the message signature
- MP_CMP_E may be returned if there is an error while computing the message signature
- MP_INVMOD_E may be returned if there is an error while computing the message signature
- MP_EXPTMOD_E may be returned if there is an error while computing the message signature
- MP_MOD_E may be returned if there is an error while computing the message signature
- MP_MUL_E may be returned if there is an error while computing the message signature
- MP_ADD_E may be returned if there is an error while computing the message signature
- MP_MULMOD_E may be returned if there is an error while computing the message signature
- MP_TO_E may be returned if there is an error while computing the message signature
- MP_MEM may be returned if there is an error while computing the message signature

Example

```
ecc_key key;
int ret, verified = 0;

byte sig[1024] { initialize with received signature };
byte digest[] = { initialize with message hash };
// initialize key with received public key
ret = wc_ecc_verify_hash(sig, sizeof(sig), digest,sizeof(digest),
&verified, &key);
if ( ret != 0 ) {
    // error performing verification
} else if ( verified == 0 ) {
    // the signature is invalid
}
```

```
int wc_ecc_verify_hash_ex(
    mp_int * r,
    mp_int * s,
    const byte * hash,
    word32 hashlen,
    int * stat,
    ecc_key * key
)
```

Verify an ECC signature. Result is written to stat. 1 is valid, 0 is invalid. Note: Do not use the return value to test for valid. Only use stat.

Parameters:

- **r** The signature R component to verify
- **s** The signature S component to verify
- **hash** The hash (message digest) that was signed
- **hashlen** The length of the hash (octets)
- **stat** Result of signature, 1==valid, 0==invalid
- **key** The corresponding public ECC key

See: wc_ecc_verify_hash

Return:

- MP_OKAY If successful (even if the signature is not valid)
- ECC_BAD_ARG_E Returns if arguments are null or if key-idx is invalid.
- MEMORY_E Error allocating ints or points.

Example

```
mp_int r;
mp_int s;
int stat;
```

```
byte hash[] = { Some hash }
ecc_key key;

if(wc_ecc_verify_hash_ex(&r, &s, hash, hashlen, &stat, &key) == MP_OKAY)
{
    // Check stat
}
```

```
int wc_ecc_init(
    ecc_key * key
)
```

This function initializes an ecc_key object for future use with message verification or key negotiation.

Parameters:

- **key** pointer to the ecc_key object to initialize

See:

- wc_ecc_make_key
- wc_ecc_free

Return:

- 0 Returned upon successfully initializing the ecc_key object
- MEMORY_E Returned if there is an error allocating memory

Example

```
ecc_key key;
wc_ecc_init(&key);
```

```
int wc_ecc_init_ex(
    ecc_key * key,
    void * heap,
    int devId
)
```

This function initializes an ecc_key object for future use with message verification or key negotiation.

Parameters:

- **key** pointer to the ecc_key object to initialize
- **heap** pointer to a heap identifier
- **devId** ID to use with crypto callbacks or async hardware. Set to INVALID_DEVID (-2) if not used

See:

- wc_ecc_make_key
- wc_ecc_free
- wc_ecc_init

Return:

- 0 Returned upon successfully initializing the ecc_key object
- MEMORY_E Returned if there is an error allocating memory

Example

```
ecc_key key;
wc_ecc_init_ex(&key, heap, devId);
```

```
ecc_key * wc_ecc_key_new(
    void * heap
)
```

This function uses a user defined heap and allocates space for the key structure.

See:

- wc_ecc_make_key
- wc_ecc_key_free
- wc_ecc_init

Return:

- 0 Returned upon successfully initializing the ecc_key object
- MEMORY_E Returned if there is an error allocating memory

Example

```
wc_ecc_key_new(&heap);
```

```
int wc_ecc_free(
    ecc_key * key
)
```

This function frees an ecc_key object after it has been used.

Parameters:

- **key** pointer to the ecc_key object to free

See: wc_ecc_init

Return: int integer returned indicating wolfSSL error or success status.

Example

```
// initialize key and perform secure exchanges
...
wc_ecc_free(&key);
```

```
void wc_ecc_fp_free(
    void
)
```

This function frees the fixed-point cache, which can be used with ecc to speed up computation times. To use this functionality, FP_ECC (fixed-point ecc), should be defined. Threaded applications should call this function before exiting the thread.

Parameters:

- **none** No parameters.

See: wc_ecc_free

Return: none No returns.

Example

```
ecc_key key;
// initialize key and perform secure exchanges
...

wc_ecc_fp_free();
```

```
int wc_ecc_is_valid_idx(
    int n
)
```

Checks if an ECC idx is valid.

Parameters:

- **n** The idx number to check.

See: none

Return:

- 1 Return if valid.
- 0 Return if not valid.

Example

```
ecc_key key;
WC_RNG rng;
int is_valid;
wc_ecc_init(&key);
wc_InitRng(&rng);
wc_ecc_make_key(&rng, 32, &key);
is_valid = wc_ecc_is_valid_idx(key.idx);
if (is_valid == 1)
{
    // idx is valid
}
else if (is_valid == 0)
{
    // idx is not valid
}
```

```
ecc_point * wc_ecc_new_point(
    void
)
```

Allocate a new ECC point.

Parameters:

- **none** No parameters.

See:

- wc_ecc_del_point
- wc_ecc_cmp_point
- wc_ecc_copy_point

Return:

- p A newly allocated point.
- NULL Returns NULL on error.

Example

```
ecc_point* point;
point = wc_ecc_new_point();
if (point == NULL)
{
    // Handle point creation error
}
// Do stuff with point
```

```
void wc_ecc_del_point(
    ecc_point * p
)
```

Free an ECC point from memory.

Parameters:

- **p** The point to free.

See:

- wc_ecc_new_point
- wc_ecc_cmp_point
- wc_ecc_copy_point

Return: none No returns.

Example

```
ecc_point* point;
point = wc_ecc_new_point();
if (point == NULL)
{
    // Handle point creation error
}
// Do stuff with point
wc_ecc_del_point(point);
```

```
int wc_ecc_copy_point(
    ecc_point * p,
    ecc_point * r
)
```

Copy the value of one point to another one.

Parameters:

- **p** The point to copy.
- **r** The created point.

See:

- wc_ecc_new_point
- wc_ecc_cmp_point
- wc_ecc_del_point

Return:

- ECC_BAD_ARG_E Error thrown when p or r is null.

- MP_OKAY Point copied successfully
- ret Error from internal functions. Can be...

Example

```
ecc_point* point;
ecc_point* copied_point;
int copy_return;

point = wc_ecc_new_point();
copy_return = wc_ecc_copy_point(point, copied_point);
if (copy_return != MP_OKAY)
{
    // Handle error
}
```

```
int wc_ecc_cmp_point(
    ecc_point * a,
    ecc_point * b
)
```

Compare the value of a point with another one.

Parameters:

- **a** First point to compare.
- **b** Second point to compare.

See:

- wc_ecc_new_point
- wc_ecc_del_point
- wc_ecc_copy_point

Return:

- BAD_FUNC_ARG One or both arguments are NULL.
- MP_EQ The points are equal.
- ret Either MP_LT or MP_GT and signifies that the points are not equal.

Example

```
ecc_point* point;
ecc_point* point_to_compare;
int cmp_result;

point = wc_ecc_new_point();
point_to_compare = wc_ecc_new_point();
cmp_result = wc_ecc_cmp_point(point, point_to_compare);
if (cmp_result == BAD_FUNC_ARG)
```

```
{
    // arguments are invalid
}
else if (cmp_result == MP_EQ)
{
    // Points are equal
}
else
{
    // Points are not equal
}
```

```
int wc_ecc_point_is_at_infinity(
    ecc_point * p
)
```

Checks if a point is at infinity. Returns 1 if point is at infinity, 0 if not, < 0 on error.

Parameters:

- **p** The point to check.

See:

- wc_ecc_new_point
- wc_ecc_del_point
- wc_ecc_cmp_point
- wc_ecc_copy_point

Return:

- 1 p is at infinity.
- 0 p is not at infinity.
- <0 Error.

Example

```
ecc_point* point;
int is_infinity;
point = wc_ecc_new_point();

is_infinity = wc_ecc_point_is_at_infinity(point);
if (is_infinity < 0)
{
    // Handle error
}
else if (is_infinity == 0)
{
    // Point is not at infinity
}
```

```
else if (is_infinity == 1)
{
    // Point is at infinity
}
```

```
int wc_ecc_mulmod(
    mp_int * k,
    ecc_point * G,
    ecc_point * R,
    mp_int * a,
    mp_int * modulus,
    int map
)
```

Perform ECC Fixed Point multiplication.

Parameters:

- **k** The multiplicand.
- **G** Base point to multiply.
- **R** Destination of product.
- **modulus** The modulus for the curve.
- **map** If non-zero maps the point back to affine coordinates, otherwise it's left in jacobian-montgomery form.

See: none

Return:

- MP_OKAY Returns on successful operation.
- MP_INIT_E Returned if there is an error initializing an integer for use with the multiple precision integer (mp_int) library.

Example

```
ecc_point* base;
ecc_point* destination;
// Initialize points
base = wc_ecc_new_point();
destination = wc_ecc_new_point();
// Setup other arguments
mp_int multiplicand;
mp_int modulus;
int map;
```

```
int wc_ecc_export_x963(
    ecc_key * key,
    byte * out,
    word32 * outLen
)
```

This function exports the ECC key from the ecc_key structure, storing the result in out. The key will be stored in ANSI X9.63 format. It stores the bytes written to the output buffer in outLen.

Parameters:

- **key** pointer to the ecc_key object to export
- **out** pointer to the buffer in which to store the ANSI X9.63 formatted key
- **outLen** size of the output buffer. On successfully storing the key, will hold the bytes written to the output buffer

See:

- wc_ecc_export_x963_ex
- wc_ecc_import_x963

Return:

- 0 Returned on successfully exporting the ecc_key
- LENGTH_ONLY_E Returned if the output buffer evaluates to NULL, but the other two input parameters are valid. Indicates that the function is only returning the length required to store the key
- ECC_BAD_ARG_E Returned if any of the input parameters are NULL, or the key is unsupported (has an invalid index)
- BUFFER_E Returned if the output buffer is too small to store the ecc key. If the output buffer is too small, the size needed will be returned in outLen
- MEMORY_E Returned if there is an error allocating memory with XMALLOC
- MP_INIT_E may be returned if there is an error processing the ecc_key
- MP_READ_E may be returned if there is an error processing the ecc_key
- MP_CMP_E may be returned if there is an error processing the ecc_key
- MP_INVMOD_E may be returned if there is an error processing the ecc_key
- MP_EXPTMOD_E may be returned if there is an error processing the ecc_key
- MP_MOD_E may be returned if there is an error processing the ecc_key
- MP_MUL_E may be returned if there is an error processing the ecc_key
- MP_ADD_E may be returned if there is an error processing the ecc_key
- MP_MULMOD_E may be returned if there is an error processing the ecc_key
- MP_TO_E may be returned if there is an error processing the ecc_key
- MP_MEM may be returned if there is an error processing the ecc_key

Example

```
int ret;
byte buff[1024];
word32 buffSz = sizeof(buff);

ecc_key key;
// initialize key, make key
ret = wc_ecc_export_x963(&key, buff, &buffSz);
if ( ret != 0) {
    // error exporting key
}
```

```
int wc_ecc_export_x963_ex(
    ecc_key * key,
    byte * out,
    word32 * outLen,
    int compressed
)
```

This function exports the ECC key from the ecc_key structure, storing the result in out. The key will be stored in ANSI X9.63 format. It stores the bytes written to the output buffer in outLen. This function allows the additional option of compressing the certificate through the compressed parameter. When this parameter is true, the key will be stored in ANSI X9.63 compressed format.

Parameters:

- **key** pointer to the ecc_key object to export
- **out** pointer to the buffer in which to store the ANSI X9.63 formatted key
- **outLen** size of the output buffer. On successfully storing the key, will hold the bytes written to the output buffer
- **compressed** indicator of whether to store the key in compressed format. 1==compressed, 0==uncompressed

See:

- wc_ecc_export_x963
- wc_ecc_import_x963

Return:

- 0 Returned on successfully exporting the ecc_key
- NOT_COMPILED_IN Returned if the HAVE_COMP_KEY was not enabled at compile time, but the key was requested in compressed format
- LENGTH_ONLY_E Returned if the output buffer evaluates to NULL, but the other two input parameters are valid. Indicates that the function is only returning the length required to store the key
- ECC_BAD_ARG_E Returned if any of the input parameters are NULL, or the key is unsupported (has an invalid index)
- BUFFER_E Returned if the output buffer is too small to store the ecc key. If the output buffer is too small, the size needed will be returned in outLen
- MEMORY_E Returned if there is an error allocating memory with XMALLOC
- MP_INIT_E may be returned if there is an error processing the ecc_key
- MP_READ_E may be returned if there is an error processing the ecc_key
- MP_CMP_E may be returned if there is an error processing the ecc_key
- MP_INVMOD_E may be returned if there is an error processing the ecc_key
- MP_EXPTMOD_E may be returned if there is an error processing the ecc_key
- MP_MOD_E may be returned if there is an error processing the ecc_key
- MP_MUL_E may be returned if there is an error processing the ecc_key
- MP_ADD_E may be returned if there is an error processing the ecc_key
- MP_MULMOD_E may be returned if there is an error processing the ecc_key
- MP_TO_E may be returned if there is an error processing the ecc_key
- MP_MEM may be returned if there is an error processing the ecc_key

Example

```
int ret;
byte buff[1024];
word32 buffSz = sizeof(buff);
ecc_key key;
// initialize key, make key
ret = wc_ecc_export_x963_ex(&key, buff, &buffSz, 1);
if ( ret != 0) {
    // error exporting key
}
```

```
int wc_ecc_import_x963(
    const byte * in,
    word32 inLen,
    ecc_key * key
)
```

This function imports a public ECC key from a buffer containing the key stored in ANSI X9.63 format. This function will handle both compressed and uncompressed keys, as long as compressed keys are enabled at compile time through the HAVE_COMP_KEY option.

Parameters:

- **in** pointer to the buffer containing the ANSI x9.63 formatted ECC key
- **inLen** length of the input buffer
- **key** pointer to the ecc_key object in which to store the imported key

See:

- wc_ecc_export_x963
- wc_ecc_import_private_key

Return:

- 0 Returned on successfully importing the ecc_key
- NOT_COMPILED_IN Returned if the HAVE_COMP_KEY was not enabled at compile time, but the key is stored in compressed format
- ECC_BAD_ARG_E Returned if in or key evaluate to NULL, or the inLen is even (according to the x9.63 standard, the key must be odd)
- MEMORY_E Returned if there is an error allocating memory
- ASN_PARSE_E Returned if there is an error parsing the ECC key; may indicate that the ECC key is not stored in valid ANSI X9.63 format
- IS_POINT_E Returned if the public key exported is not a point on the ECC curve
- MP_INIT_E may be returned if there is an error processing the ecc_key
- MP_READ_E may be returned if there is an error processing the ecc_key
- MP_CMP_E may be returned if there is an error processing the ecc_key
- MP_INVMOD_E may be returned if there is an error processing the ecc_key
- MP_EXPTMOD_E may be returned if there is an error processing the ecc_key
- MP_MOD_E may be returned if there is an error processing the ecc_key
- MP_MUL_E may be returned if there is an error processing the ecc_key
- MP_ADD_E may be returned if there is an error processing the ecc_key
- MP_MULMOD_E may be returned if there is an error processing the ecc_key

- MP_TO_E may be returned if there is an error processing the ecc_key
- MP_MEM may be returned if there is an error processing the ecc_key

Example

```
int ret;
byte buff[] = { initialize with ANSI X9.63 formatted key };

ecc_key pubKey;
wc_ecc_init(&pubKey);

ret = wc_ecc_import_x963(buff, sizeof(buff), &pubKey);
if ( ret != 0) {
    // error importing key
}
```

```
int wc_ecc_import_private_key(
    const byte * priv,
    word32 privSz,
    const byte * pub,
    word32 pubSz,
    ecc_key * key
)
```

This function imports a public/private ECC key pair from a buffer containing the raw private key, and a second buffer containing the ANSI X9.63 formatted public key. This function will handle both compressed and uncompressed keys, as long as compressed keys are enabled at compile time through the HAVE_COMP_KEY option.

Parameters:

- **priv** pointer to the buffer containing the raw private key
- **privSz** size of the private key buffer
- **pub** pointer to the buffer containing the ANSI x9.63 formatted ECC public key
- **pubSz** length of the public key input buffer
- **key** pointer to the ecc_key object in which to store the imported private/public key pair

See:

- wc_ecc_export_x963
- wc_ecc_import_private_key

Return:

- 0 Returned on successfully importing the ecc_key NOT_COMPILED_IN Returned if the HAVE_COMP_KEY was not enabled at compile time, but the key is stored in compressed format
- ECC_BAD_ARG_E Returned if in or key evaluate to NULL, or the inLen is even (according to the x9.63 standard, the key must be odd)
- MEMORY_E Returned if there is an error allocating memory

- ASN_PARSE_E Returned if there is an error parsing the ECC key; may indicate that the ECC key is not stored in valid ANSI X9.63 format
- IS_POINT_E Returned if the public key exported is not a point on the ECC curve
- MP_INIT_E may be returned if there is an error processing the ecc_key
- MP_READ_E may be returned if there is an error processing the ecc_key
- MP_CMP_E may be returned if there is an error processing the ecc_key
- MP_INVMOD_E may be returned if there is an error processing the ecc_key
- MP_EXPTMOD_E may be returned if there is an error processing the ecc_key
- MP_MOD_E may be returned if there is an error processing the ecc_key
- MP_MUL_E may be returned if there is an error processing the ecc_key
- MP_ADD_E may be returned if there is an error processing the ecc_key
- MP_MULMOD_E may be returned if there is an error processing the ecc_key
- MP_TO_E may be returned if there is an error processing the ecc_key
- MP_MEM may be returned if there is an error processing the ecc_key

Example

```
int ret;
byte pub[] = { initialize with ANSI X9.63 formatted key };
byte priv[] = { initialize with the raw private key };

ecc_key key;
wc_ecc_init(&key);
ret = wc_ecc_import_private_key(priv, sizeof(priv), pub, sizeof(pub),
&key);
if ( ret != 0) {
    // error importing key
}
```

```
int wc_ecc_rs_to_sig(
    const char * r,
    const char * s,
    byte * out,
    word32 * outlen
)
```

This function converts the R and S portions of an ECC signature into a DER-encoded ECDSA signature. This function also stores the length written to the output buffer, out, in outlen.

Parameters:

- **r** pointer to the buffer containing the R portion of the signature as a string
- **s** pointer to the buffer containing the S portion of the signature as a string
- **out** pointer to the buffer in which to store the DER-encoded ECDSA signature
- **outlen** length of the output buffer available. Will store the bytes written to the buffer after successfully converting the signature to ECDSA format

See:

- wc_ecc_sign_hash
- wc_ecc_sig_size

Return:

- 0 Returned on successfully converting the signature
- ECC_BAD_ARG_E Returned if any of the input parameters evaluate to NULL, or if the input buffer is not large enough to hold the DER-encoded ECDSA signature
- MP_INIT_E may be returned if there is an error processing the ecc_key
- MP_READ_E may be returned if there is an error processing the ecc_key
- MP_CMP_E may be returned if there is an error processing the ecc_key
- MP_INVMOD_E may be returned if there is an error processing the ecc_key
- MP_EXPTMOD_E may be returned if there is an error processing the ecc_key
- MP_MOD_E may be returned if there is an error processing the ecc_key
- MP_MUL_E may be returned if there is an error processing the ecc_key
- MP_ADD_E may be returned if there is an error processing the ecc_key
- MP_MULMOD_E may be returned if there is an error processing the ecc_key
- MP_TO_E may be returned if there is an error processing the ecc_key
- MP_MEM may be returned if there is an error processing the ecc_key

Example

```
int ret;
ecc_key key;
// initialize key, generate R and S

char r[] = { initialize with R };
char s[] = { initialize with S };
byte sig[wc_ecc_sig_size(key)];
// signature size will be 2 * ECC key size + ~10 bytes for ASN.1 overhead
word32 sigSz = sizeof(sig);
ret = wc_ecc_rs_to_sig(r, s, sig, &sigSz);
if ( ret != 0) {
    // error converting parameters to signature
}
```

```
int wc_ecc_import_raw(
    ecc_key * key,
    const char * qx,
    const char * qy,
    const char * d,
    const char * curveName
)
```

This function fills an ecc_key structure with the raw components of an ECC signature.

Parameters:

- **key** pointer to an ecc_key structure to fill
- **qx** pointer to a buffer containing the x component of the base point as an ASCII hex string
- **qy** pointer to a buffer containing the y component of the base point as an ASCII hex string
- **d** pointer to a buffer containing the private key as an ASCII hex string
- **curveName** pointer to a string containing the ECC curve name, as found in ecc_sets

See: wc_ecc_import_private_key

Return:

- 0 Returned upon successfully importing into the ecc_key structure
- ECC_BAD_ARG_E Returned if any of the input values evaluate to NULL
- MEMORY_E Returned if there is an error initializing space to store the parameters of the ecc_key
- ASN_PARSE_E Returned if the input curveName is not defined in ecc_sets
- MP_INIT_E may be returned if there is an error processing the input parameters
- MP_READ_E may be returned if there is an error processing the input parameters
- MP_CMP_E may be returned if there is an error processing the input parameters
- MP_INVMOD_E may be returned if there is an error processing the input parameters
- MP_EXPTMOD_E may be returned if there is an error processing the input parameters
- MP_MOD_E may be returned if there is an error processing the input parameters
- MP_MUL_E may be returned if there is an error processing the input parameters
- MP_ADD_E may be returned if there is an error processing the input parameters
- MP_MULMOD_E may be returned if there is an error processing the input parameters
- MP_TO_E may be returned if there is an error processing the input parameters
- MP_MEM may be returned if there is an error processing the input parameters

Example

```
int ret;
ecc_key key;
wc_ecc_init(&key);

char qx[] = { initialize with x component of base point };
char qy[] = { initialize with y component of base point };
char d[]  = { initialize with private key };
ret = wc_ecc_import_raw(&key,qx, qy, d, "ECC-256");
if ( ret != 0) {
    // error initializing key with given inputs
}
```

```
int wc_ecc_export_private_only(
    ecc_key * key,
    byte * out,
    word32 * outLen
)
```

This function exports only the private key from an ecc_key structure. It stores the private key in the buffer out, and sets the bytes written to this buffer in outLen.

Parameters:

- **key** pointer to an ecc_key structure from which to export the private key
- **out** pointer to the buffer in which to store the private key
- **outLen** pointer to a word32 object with the size available in out. Set with the number of bytes written to out after successfully exporting the private key

See: wc_ecc_import_private_key

Return:

- 0 Returned upon successfully exporting the private key
- ECC_BAD_ARG_E Returned if any of the input values evaluate to NULL
- MEMORY_E Returned if there is an error initializing space to store the parameters of the ecc_key
- ASN_PARSE_E Returned if the input curveName is not defined in ecc_sets
- MP_INIT_E may be returned if there is an error processing the input parameters
- MP_READ_E may be returned if there is an error processing the input parameters
- MP_CMP_E may be returned if there is an error processing the input parameters
- MP_INVMOD_E may be returned if there is an error processing the input parameters
- MP_EXPTMOD_E may be returned if there is an error processing the input parameters
- MP_MOD_E may be returned if there is an error processing the input parameters
- MP_MUL_E may be returned if there is an error processing the input parameters
- MP_ADD_E may be returned if there is an error processing the input parameters
- MP_MULMOD_E may be returned if there is an error processing the input parameters
- MP_TO_E may be returned if there is an error processing the input parameters
- MP_MEM may be returned if there is an error processing the input parameters

Example

```
int ret;
ecc_key key;
// initialize key, make key

char priv[ECC_KEY_SIZE];
word32 privSz = sizeof(priv);
ret = wc_ecc_export_private_only(&key, priv, &privSz);
if ( ret != 0) {
    // error exporting private key
}
```

```
int wc_ecc_export_point_der(
    const int curve_idx,
    ecc_point * point,
    byte * out,
    word32 * outLen
)
```

Export point to der.

Parameters:

- **curve_idx** Index of the curve used from ecc_sets.
- **point** Point to export to der.
- **out** Destination for the output.
- **outLen** Maxsize allowed for output, destination for final size of output

See: wc_ecc_import_point_der

Return:

- 0 Returned on success.
- ECC_BAD_ARG_E Returns if curve_idx is less than 0 or invalid. Also returns when
- LENGTH_ONLY_E outLen is set but nothing else.
- BUFFER_E Returns if outLen is less than 1 + 2 * the curve size.
- MEMORY_E Returns if there is a problem allocating memory.

Example

```
int curve_idx;
ecc_point* point;
byte out[];
word32 outLen;
wc_ecc_export_point_der(curve_idx, point, out, &outLen);
```

```
int wc_ecc_import_point_der(
    byte * in,
    word32 inLen,
    const int curve_idx,
    ecc_point * point
)
```

Import point from der format.

Parameters:

- **in** der buffer to import point from.
- **inLen** Length of der buffer.
- **curve_idx** Index of curve.
- **point** Destination for point.

See: wc_ecc_export_point_der

Return:

- ECC_BAD_ARG_E Returns if any arguments are null or if inLen is even.
- MEMORY_E Returns if there is an error initializing
- NOT_COMPILED_IN Returned if HAVE_COMP_KEY is not true and in is a compressed cert
- MP_OKAY Successful operation.

Example

```
byte in[];
word32 inLen;
int curve_idx;
ecc_point* point;
wc_ecc_import_point_der(in, inLen, curve_idx, point);
```

```
int wc_ecc_size(
    ecc_key * key
)
```

This function returns the key size of an ecc_key structure in octets.

Parameters:

- **key** pointer to an ecc_key structure for which to get the key size

See: wc_ecc_make_key

Return:

- Given a valid key, returns the key size in octets
- 0 Returned if the given key is NULL

Example

```
int keySz;
ecc_key key;
// initialize key, make key
keySz = wc_ecc_size(&key);
if ( keySz == 0) {
    // error determining key size
}
```

```
int wc_ecc_sig_size_calc(
    int sz
)
```

This function returns the worst case size for an ECC signature, given by: (keySz * 2) + SIG_HEADER_SZ + ECC_MAX_PAD_SZ. The actual signature size can be computed with wc_ecc_sign_hash.

Parameters:

- **key** size

See:

- wc_ecc_sign_hash
- wc_ecc_sig_size

Return: returns the maximum signature size, in octets

Example

```
int sigSz = wc_ecc_sig_size_calc(32);
if ( sigSz == 0) {
    // error determining sig size
}
```

```
int wc_ecc_sig_size(
    ecc_key * key
)
```

This function returns the worst case size for an ECC signature, given by: (keySz * 2) + SIG_HEADER_SZ + ECC_MAX_PAD_SZ. The actual signature size can be computed with wc_ecc_sign_hash.

Parameters:

- **key** pointer to an ecc_key structure for which to get the signature size

See:

- wc_ecc_sign_hash
- wc_ecc_sig_size_calc

Return:

- Success Given a valid key, returns the maximum signature size, in octets
- 0 Returned if the given key is NULL

Example

```
int sigSz;
ecc_key key;
// initialize key, make key

sigSz = wc_ecc_sig_size(&key);
if ( sigSz == 0) {
    // error determining sig size
}
```

```
ecEncCtx * wc_ecc_ctx_new(
    int flags,
    WC_RNG * rng
)
```

This function allocates and initializes space for a new ECC context object to allow secure message exchange with ECC.

Parameters:

- **flags** indicate whether this is a server or client context Options are: REQ_RESP_CLIENT, and REQ_RESP_SERVER
- **rng** pointer to a RNG object with which to generate a salt

See:

- wc_ecc_encrypt
- wc_ecc_encrypt_ex
- wc_ecc_decrypt

Return:

- Success On successfully generating a new ecEncCtx object, returns a pointer to that object
- NULL Returned if the function fails to generate a new ecEncCtx object

Example

```
ecEncCtx* ctx;
WC_RNG rng;
wc_InitRng(&rng);
ctx = wc_ecc_ctx_new(REQ_RESP_CLIENT, &rng);
if(ctx == NULL) {
    // error generating new ecEncCtx object
}
```

```
void wc_ecc_ctx_free(
    ecEncCtx *
)
```

This function frees the ecEncCtx object used for encrypting and decrypting messages.

Parameters:

- **ctx** pointer to the ecEncCtx object to free

See: wc_ecc_ctx_new

Return: none Returns.

Example

```
ecEncCtx* ctx;
WC_RNG rng;
wc_InitRng(&rng);
ctx = wc_ecc_ctx_new(REQ_RESP_CLIENT, &rng);
// do secure communication
...
wc_ecc_ctx_free(&ctx);
```

```
int wc_ecc_ctx_reset(
    ecEncCtx * ctx,
    WC_RNG * rng
)
```

This function resets an ecEncCtx structure to avoid having to free and allocate a new context object.

Parameters:

- **ctx** pointer to the ecEncCtx object to reset
- **rng** pointer to an RNG object with which to generate a new salt

See: wc_ecc_ctx_new

Return:

- 0 Returned if the ecEncCtx structure is successfully reset
- BAD_FUNC_ARG Returned if either rng or ctx is NULL
- RNG_FAILURE_E Returned if there is an error generating a new salt for the ECC object

Example

```
ecEncCtx* ctx;
WC_RNG rng;
wc_InitRng(&rng);
ctx = wc_ecc_ctx_new(REQ_RESP_CLIENT, &rng);
// do secure communication
...
wc_ecc_ctx_reset(&ctx, &rng);
// do more secure communication
```

```
int wc_ecc_ctx_set_algo(
    ecEncCtx * ctx,
    byte encAlgo,
    byte kdfAlgo,
    byte macAlgo
)
```

This function can optionally be called after wc_ecc_ctx_new. It sets the encryption, KDF, and MAC algorithms into an ecEncCtx object.

Parameters:

- **ctx** pointer to the ecEncCtx for which to set the info
- **encAlgo** encryption algorithm to use.
- **kdfAlgo** KDF algorithm to use.
- **macAlgo** MAC algorithm to use.

See: wc_ecc_ctx_new

Return:

- 0 Returned upon successfully setting the information for the ecEncCtx object.
- BAD_FUNC_ARG Returned if the given ecEncCtx object is NULL.

Example

```
ecEncCtx* ctx;
// initialize ctx
if(wc_ecc_ctx_set_algo(&ctx, ecAES_128_CTR, ecHKDF_SHA256, ecHMAC_SHA256))) {
    // error setting info
}
```

```
const byte * wc_ecc_ctx_get_own_salt(
    ecEncCtx *
)
```

This function returns the salt of an ecEncCtx object. This function should only be called when the ecEncCtx's state is ecSRV_INIT or ecCLI_INIT.

Parameters:

- **ctx** pointer to the ecEncCtx object from which to get the salt

See:

- wc_ecc_ctx_new
- wc_ecc_ctx_set_peer_salt
- wc_ecc_ctx_set_kdf_salt

Return:

- Success On success, returns the ecEncCtx salt
- NULL Returned if the ecEncCtx object is NULL, or the ecEncCtx's state is not ecSRV_INIT or ecCLI_INIT. In the latter two cases, this function also sets the ecEncCtx's state to ecSRV_BAD_STATE or ecCLI_BAD_STATE, respectively

Example

```
ecEncCtx* ctx;
WC_RNG rng;
const byte* salt;
wc_InitRng(&rng);
ctx = wc_ecc_ctx_new(REQ_RESP_CLIENT, &rng);
salt = wc_ecc_ctx_get_own_salt(&ctx);
if(salt == NULL) {
    // error getting salt
}
```

```
int wc_ecc_ctx_set_peer_salt(
    ecEncCtx * ctx,
    const byte * salt
)
```

This function sets the peer salt of an ecEncCtx object.

Parameters:

- **ctx** pointer to the ecEncCtx for which to set the salt
- **salt** pointer to the peer's salt

See:

- wc_ecc_ctx_get_own_salt
- wc_ecc_ctx_set_kdf_salt

Return:

- 0 Returned upon successfully setting the peer salt for the ecEncCtx object.
- BAD_FUNC_ARG Returned if the given ecEncCtx object is NULL or has an invalid protocol, or if the given salt is NULL
- BAD_ENC_STATE_E Returned if the ecEncCtx's state is ecSRV_SALT_GET or ecCLI_SALT_GET. In the latter two cases, this function also sets the ecEncCtx's state to ecSRV_BAD_STATE or ecCLI_BAD_STATE, respectively

Example

```
ecEncCtx* cliCtx, srvCtx;
WC_RNG rng;
const byte* cliSalt, srvSalt;
int ret;

wc_InitRng(&rng);
cliCtx = wc_ecc_ctx_new(REQ_RESP_CLIENT, &rng);
srvCtx = wc_ecc_ctx_new(REQ_RESP_SERVER, &rng);

cliSalt = wc_ecc_ctx_get_own_salt(&cliCtx);
srvSalt = wc_ecc_ctx_get_own_salt(&srvCtx);
ret = wc_ecc_ctx_set_peer_salt(&cliCtx, srvSalt);
```

```
int wc_ecc_ctx_set_kdf_salt(
    ecEncCtx * ctx,
    const byte * salt,
    word32 sz
)
```

This function sets the salt pointer and length to use with KDF into the ecEncCtx object.

Parameters:

- **ctx** pointer to the ecEncCtx for which to set the salt
- **salt** pointer to salt buffer
- **sz** length salt in bytes

See:

- wc_ecc_ctx_get_own_salt
- wc_ecc_ctx_get_peer_salt

Return:

- 0 Returned upon successfully setting the salt for the ecEncCtx object.
- BAD_FUNC_ARG Returned if the given ecEncCtx object is NULL or if the given salt is NULL and length is not NULL.

Example

```
ecEncCtx* srvCtx;
WC_RNG rng;
byte cliSalt[] = { fixed salt data };
word32 cliSaltLen = (word32)sizeof(cliSalt);
int ret;

wc_InitRng(&rng);
cliCtx = wc_ecc_ctx_new(REQ_RESP_SERVER, &rng);

ret = wc_ecc_ctx_set_kdf_salt(&cliCtx, cliSalt, cliSaltLen);
```

```
int wc_ecc_ctx_set_info(
    ecEncCtx * ctx,
    const byte * info,
    int sz
)
```

This function can optionally be called before or after wc_ecc_ctx_set_peer_salt. It sets optional information for an ecEncCtx object.

Parameters:

- **ctx** pointer to the ecEncCtx for which to set the info
- **info** pointer to a buffer containing the info to set
- **sz** size of the info buffer

See: wc_ecc_ctx_new

Return:

- 0 Returned upon successfully setting the information for the ecEncCtx object.
- BAD_FUNC_ARG Returned if the given ecEncCtx object is NULL, the input info is NULL or it's size is invalid

Example

```
ecEncCtx* ctx;
byte info[] = { initialize with information };
// initialize ctx, get salt,
if(wc_ecc_ctx_set_info(&ctx, info, sizeof(info))) {
    // error setting info
}
```

```
int wc_ecc_encrypt(
    ecc_key * privKey,
    ecc_key * pubKey,
    const byte * msg,
    word32 msgSz,
    byte * out,
    word32 * outSz,
    ecEncCtx * ctx
)
```

This function encrypts the given input message from msg to out. This function takes an optional ctx object as parameter. When supplied, encryption proceeds based on the ecEncCtx's encAlgo, kdfAlgo, and macAlgo. If ctx is not supplied, processing completes with the default algorithms, ecAES_128_CBC, ecHKDF_SHA256 and ecHMAC_SHA256. This function requires that the messages are padded according to the encryption type specified by ctx.

Parameters:

- **privKey** pointer to the ecc_key object containing the private key to use for encryption
- **pubKey** pointer to the ecc_key object containing the public key of the peer with whom one wishes to communicate
- **msg** pointer to the buffer holding the message to encrypt
- **msgSz** size of the buffer to encrypt
- **out** pointer to the buffer in which to store the encrypted ciphertext
- **outSz** pointer to a word32 object containing the available size in the out buffer. Upon successfully encrypting the message, holds the number of bytes written to the output buffer
- **ctx** Optional: pointer to an ecEncCtx object specifying different encryption algorithms to use

See:

- wc_ecc_encrypt_ex
- wc_ecc_decrypt

Return:

- 0 Returned upon successfully encrypting the input message
- BAD_FUNC_ARG Returned if privKey, pubKey, msg, msgSz, out, or outSz are NULL, or the ctx object specifies an unsupported encryption type
- BAD_ENC_STATE_E Returned if the ctx object given is in a state that is not appropriate for encryption
- BUFFER_E Returned if the supplied output buffer is too small to store the encrypted ciphertext
- MEMORY_E Returned if there is an error allocating memory for the shared secret key

Example

```
byte msg[] = { initialize with msg to encrypt. Ensure padded to block size };
byte out[sizeof(msg)];
word32 outSz = sizeof(out);
int ret;
ecc_key cli, serv;
// initialize cli with private key
// initialize serv with received public key

ecEncCtx* cliCtx, servCtx;
// initialize cliCtx and servCtx
// exchange salts
ret = wc_ecc_encrypt(&cli, &serv, msg, sizeof(msg), out, &outSz, cliCtx);
if(ret != 0) {
    // error encrypting message
}
```

```
int wc_ecc_encrypt_ex(
    ecc_key * privKey,
    ecc_key * pubKey,
    const byte * msg,
    word32 msgSz,
    byte * out,
    word32 * outSz,
    ecEncCtx * ctx,
    int compressed
)
```

This function encrypts the given input message from msg to out. This function takes an optional ctx object as parameter. When supplied, encryption proceeds based on the ecEncCtx's encAlgo, kdfAlgo, and macAlgo. If ctx is not supplied, processing completes with the default algorithms, ecAES_128_CBC, ecHKDF_SHA256 and ecHMAC_SHA256. This function requires that the messages are padded according to the encryption type specified by ctx.

Parameters:

- **privKey** pointer to the ecc_key object containing the private key to use for encryption
- **pubKey** pointer to the ecc_key object containing the public key of the peer with whom one wishes to communicate
- **msg** pointer to the buffer holding the message to encrypt
- **msgSz** size of the buffer to encrypt
- **out** pointer to the buffer in which to store the encrypted ciphertext
- **outSz** pointer to a word32 object containing the available size in the out buffer. Upon successfully encrypting the message, holds the number of bytes written to the output buffer
- **ctx** Optional: pointer to an ecEncCtx object specifying different encryption algorithms to use
- **compressed** Public key field is to be output in compressed format.

See:

- wc_ecc_encrypt
- wc_ecc_decrypt

Return:

- 0 Returned upon successfully encrypting the input message
- BAD_FUNC_ARG Returned if privKey, pubKey, msg, msgSz, out, or outSz are NULL, or the ctx object specifies an unsupported encryption type
- BAD_ENC_STATE_E Returned if the ctx object given is in a state that is not appropriate for encryption
- BUFFER_E Returned if the supplied output buffer is too small to store the encrypted ciphertext
- MEMORY_E Returned if there is an error allocating memory for the shared secret key

Example

```
byte msg[] = { initialize with msg to encrypt. Ensure padded to block size };
byte out[sizeof(msg)];
word32 outSz = sizeof(out);
int ret;
ecc_key cli, serv;
// initialize cli with private key
// initialize serv with received public key

ecEncCtx* cliCtx, servCtx;
// initialize cliCtx and servCtx
// exchange salts
ret = wc_ecc_encrypt_ex(&cli, &serv, msg, sizeof(msg), out, &outSz, cliCtx,
    1);
if(ret != 0) {
    // error encrypting message
}
```

```
int wc_ecc_decrypt(
    ecc_key * privKey,
    ecc_key * pubKey,
    const byte * msg,
    word32 msgSz,
    byte * out,
    word32 * outSz,
    ecEncCtx * ctx
)
```

This function decrypts the ciphertext from msg to out. This function takes an optional ctx object as parameter. When supplied, encryption proceeds based on the ecEncCtx's encAlgo, kdfAlgo, and macAlgo. If ctx is not supplied, processing completes with the default algorithms, ecAES_128_CBC, ecHKDF_SHA256 and ecHMAC_SHA256. This function requires that the messages are padded according to the encryption type specified by ctx.

Parameters:

- **privKey** pointer to the ecc_key object containing the private key to use for decryption
- **pubKey** pointer to the ecc_key object containing the public key of the peer with whom one wishes to communicate

- **msg** pointer to the buffer holding the ciphertext to decrypt
- **msgSz** size of the buffer to decrypt
- **out** pointer to the buffer in which to store the decrypted plaintext
- **outSz** pointer to a word32 object containing the available size in the out buffer. Upon successfully decrypting the ciphertext, holds the number of bytes written to the output buffer
- **ctx** Optional: pointer to an ecEncCtx object specifying different decryption algorithms to use

See:

- wc_ecc_encrypt
- wc_ecc_encrypt_ex

Return:

- 0 Returned upon successfully decrypting the input message
- BAD_FUNC_ARG Returned if privKey, pubKey, msg, msgSz, out, or outSz are NULL, or the ctx object specifies an unsupported encryption type
- BAD_ENC_STATE_E Returned if the ctx object given is in a state that is not appropriate for decryption
- BUFFER_E Returned if the supplied output buffer is too small to store the decrypted plaintext
- MEMORY_E Returned if there is an error allocating memory for the shared secret key

Example

```
byte cipher[] = { initialize with
ciphertext to decrypt. Ensure padded to block size };
byte plain[sizeof(cipher)];
word32 plainSz = sizeof(plain);
int ret;
ecc_key cli, serv;
// initialize cli with private key
// initialize serv with received public key
ecEncCtx* cliCtx, servCtx;
// initialize cliCtx and servCtx
// exchange salts
ret = wc_ecc_decrypt(&cli, &serv, cipher, sizeof(cipher),
plain, &plainSz, cliCtx);

if(ret != 0) {
    // error decrypting message
}
```

```
int wc_ecc_set_nonblock(
    ecc_key * key,
    ecc_nb_ctx_t * ctx
)
```

Enable ECC support for non-blocking operations. Supported for Single Precision (SP) math with the following build options: WOLFSSL_SP_NONBLOCK WOLFSSL_SP_SMALL WOLFSSL_SP_NO_MALLOC WC_ECC_NONBLOCK.

Parameters:

- **key** pointer to the ecc_key object
- **ctx** pointer to ecc_nb_ctx_t structure with stack data cache for SP

Return: 0 Returned upon successfully setting the callback context the input message

Example

```
int ret;
ecc_key ecc;
ecc_nb_ctx_t nb_ctx;

ret = wc_ecc_init(&ecc);
if (ret == 0) {
    ret = wc_ecc_set_nonblock(&ecc, &nb_ctx);
    if (ret == 0) {
        do {
            ret = wc_ecc_verify_hash_ex(
                &r, &s,       // r/s as mp_int
                hash, hashSz, // computed hash digest
                &verify_res,  // verification result 1=success
                &key
            );

            // TODO: Real-time work can be called here
        } while (ret == FP_WOULDBLOCK);
    }
    wc_ecc_free(&key);
}
```

```
int wc_ecc_set_curve(
    ecc_key * key,
    int keysize,
    int curve_id
)
```

Compare a curve which has larger key than specified size or the curve matched curve ID, set a curve with smaller key size to the key.

Parameters:

- **keysize** Key size in bytes
- **curve_id** Curve ID

```
 int ret;
ecc_key ecc;
```

```
ret = wc_ecc_init(&ecc);
if (ret != 0)
    return ret;
    ret = wc_ecc_set_curve(&ecc, 32, ECC_SECP256R1));
    if (ret != 0)
        return ret;
```

Return: 0 Returned upon successfully setting the key

B.25 Algorithms - ED25519

B.24.2.47 function wc_ecc_set_curve

B.25.1 Functions

	Name
int	**wc_ed25519_make_public**(ed25519_key * key, unsigned char * pubKey, word32 pubKeySz)This function generates the Ed25519 public key from the private key, stored in the ed25519_key object. It stores the public key in the buffer pubKey.
int	**wc_ed25519_make_key**(WC_RNG * rng, int keysize, ed25519_key * key)This function generates a new Ed25519 key and stores it in key.
int	**wc_ed25519_sign_msg**(const byte * in, word32 inlen, byte * out, word32 * outlen, ed25519_key * key)This function signs a message using an ed25519_key object to guarantee authenticity.
int	**wc_ed25519ctx_sign_msg**(const byte * in, word32 inlen, byte * out, word32 * outlen, ed25519_key * key, const byte * context, byte contextLen)This function signs a message using an ed25519_key object to guarantee authenticity. The context is part of the data signed.
int	**wc_ed25519ph_sign_hash**(const byte * hash, word32 hashLen, byte * out, word32 * outLen, ed25519_key * key, const byte * context, byte contextLen)This function signs a message digest using an ed25519_key object to guarantee authenticity. The context is included as part of the data signed. The message is pre-hashed before signature calculation.

	Name
int	**wc_ed25519ph_sign_msg**(const byte * in, word32 inlen, byte * out, word32 * outlen, ed25519_key * key, const byte * context, byte contextLen)This function signs a message using an ed25519_key object to guarantee authenticity. The context is included as part of the data signed. The message is pre-hashed before signature calculation.
int	**wc_ed25519_verify_msg**(const byte * sig, word32 siglen, const byte * msg, word32 msgLen, int * ret, ed25519_key * key)This function verifies the Ed25519 signature of a message to ensure authenticity. It returns the answer through ret, with 1 corresponding to a valid signature, and 0 corresponding to an invalid signature.
int	**wc_ed25519ctx_verify_msg**(const byte * sig, word32 siglen, const byte * msg, word32 msgLen, int * ret, ed25519_key * key, const byte * context, byte contextLen)This function verifies the Ed25519 signature of a message to ensure authenticity. The context is included as part of the data verified. It returns the answer through ret, with 1 corresponding to a valid signature, and 0 corresponding to an invalid signature.
int	**wc_ed25519ph_verify_hash**(const byte * sig, word32 siglen, const byte * hash, word32 hashLen, int * ret, ed25519_key * key, const byte * context, byte contextLen)This function verifies the Ed25519 signature of the digest of a message to ensure authenticity. The context is included as part of the data verified. The hash is the pre-hashed message before signature calculation. The hash algorithm used to create message digest must be SHA-512. The answer is returned through ret, with 1 corresponding to a valid signature, and 0 corresponding to an invalid signature.
int	**wc_ed25519ph_verify_msg**(const byte * sig, word32 siglen, const byte * msg, word32 msgLen, int * ret, ed25519_key * key, const byte * context, byte contextLen)This function verifies the Ed25519 signature of a message to ensure authenticity. The context is included as part of the data verified. The message is pre-hashed before verification. It returns the answer through ret, with 1 corresponding to a valid signature, and 0 corresponding to an invalid signature.

	Name
int	**wc_ed25519_init**(ed25519_key * key)This function initializes an ed25519_key object for future use with message verification.
void	**wc_ed25519_free**(ed25519_key * key)This function frees an Ed25519 object after it has been used.
int	**wc_ed25519_import_public**(const byte * in, word32 inLen, ed25519_key * key)This function imports a public ed25519_key from a buffer containing the public key. This function will handle both compressed and uncompressed keys. The public key is checked that it matches the private key when one is present.
int	**wc_ed25519_import_public_ex**(const byte * in, word32 inLen, ed25519_key * key, int trusted)This function imports a public ed25519_key from a buffer containing the public key. This function will handle both compressed and uncompressed keys. Check public key matches private key, when present, when not trusted.
int	**wc_ed25519_import_private_only**(const byte * priv, word32 privSz, ed25519_key * key)This function imports an Ed25519 private key only from a buffer.
int	**wc_ed25519_import_private_key**(const byte * priv, word32 privSz, const byte * pub, word32 pubSz, ed25519_key * key)This function imports a public/private Ed25519 key pair from a pair of buffers. This function will handle both compressed and uncompressed keys. The public key is assumed to be untrusted and is checked against the private key.
int	**wc_ed25519_import_private_key_ex**(const byte * priv, word32 privSz, const byte * pub, word32 pubSz, ed25519_key * key, int trusted)This function imports a public/private Ed25519 key pair from a pair of buffers. This function will handle both compressed and uncompressed keys. The public is checked against private key if not trusted.
int	**wc_ed25519_export_public**(ed25519_key * key, byte * out, word32 * outLen)This function exports the private key from an ed25519_key structure. It stores the public key in the buffer out, and sets the bytes written to this buffer in outLen.

	Name
int	**wc_ed25519_export_private_only**(ed25519_key * key, byte * out, word32 * outLen)This function exports only the private key from an ed25519_key structure. It stores the private key in the buffer out, and sets the bytes written to this buffer in outLen.
int	**wc_ed25519_export_private**(ed25519_key * key, byte * out, word32 * outLen)This function exports the key pair from an ed25519_key structure. It stores the key pair in the buffer out, and sets the bytes written to this buffer in outLen.
int	**wc_ed25519_export_key**(ed25519_key * key, byte * priv, word32 * privSz, byte * pub, word32 * pubSz)This function exports the private and public key separately from an ed25519_key structure. It stores the private key in the buffer priv, and sets the bytes written to this buffer in privSz. It stores the public key in the buffer pub, and sets the bytes written to this buffer in pubSz.
int	**wc_ed25519_check_key**(ed25519_key * key)This function checks the public key in ed25519_key structure matches the private key.
int	**wc_ed25519_size**(ed25519_key * key)This function returns the size of an Ed25519 - 32 bytes.
int	**wc_ed25519_priv_size**(ed25519_key * key)This function returns the private key size (secret + public) in bytes.
int	**wc_ed25519_pub_size**(ed25519_key * key)This function returns the compressed key size in bytes (public key).
int	**wc_ed25519_sig_size**(ed25519_key * key)This function returns the size of an Ed25519 signature (64 in bytes).

B.25.2 Functions Documentation

```
int wc_ed25519_make_public(
    ed25519_key * key,
    unsigned char * pubKey,
    word32 pubKeySz
)
```

This function generates the Ed25519 public key from the private key, stored in the ed25519_key object. It stores the public key in the buffer pubKey.

Parameters:

- **key** Pointer to the ed25519_key for which to generate a key.

- **pubKey** Pointer to the buffer in which to store the public key.
- **pubKeySz** Size of the public key. Should be ED25519_PUB_KEY_SIZE.

See:

- wc_ed25519_init
- wc_ed25519_import_private_only
- wc_ed25519_make_key

Return:

- 0 Returned upon successfully making the public key.
- BAD_FUNC_ARG Returned if key or pubKey evaluate to NULL, or if the specified key size is not 32 bytes (Ed25519 has 32 byte keys).
- ECC_PRIV_KEY_E returned if the ed25519_key object does not have the private key in it.
- MEMORY_E Returned if there is an error allocating memory during function execution.

Example

```
int ret;

ed25519_key key;
byte priv[] = { initialize with 32 byte private key };
byte pub[32];
word32 pubSz = sizeof(pub);

wc_ed25519_init(&key);
wc_ed25519_import_private_only(priv, sizeof(priv), &key);
ret = wc_ed25519_make_public(&key, pub, &pubSz);
if (ret != 0) {
    // error making public key
}
```

```
int wc_ed25519_make_key(
    WC_RNG * rng,
    int keysize,
    ed25519_key * key
)
```

This function generates a new Ed25519 key and stores it in key.

Parameters:

- **rng** Pointer to an initialized RNG object with which to generate the key.
- **keysize** Length of key to generate. Should always be 32 for Ed25519.
- **key** Pointer to the ed25519_key for which to generate a key.

See: wc_ed25519_init

Return:

- 0 Returned upon successfully making an ed25519_key.
- BAD_FUNC_ARG Returned if rng or key evaluate to NULL, or if the specified key size is not 32 bytes (Ed25519 has 32 byte keys).
- MEMORY_E Returned if there is an error allocating memory during function execution.

Example

```
int ret;

WC_RNG rng;
ed25519_key key;

wc_InitRng(&rng);
wc_ed25519_init(&key);
wc_ed25519_make_key(&rng, 32, &key);
if (ret != 0) {
    // error making key
}
```

```
int wc_ed25519_sign_msg(
    const byte * in,
    word32 inlen,
    byte * out,
    word32 * outlen,
    ed25519_key * key
)
```

This function signs a message using an ed25519_key object to guarantee authenticity.

Parameters:

- **in** Pointer to the buffer containing the message to sign.
- **inlen** Length of the message to sign.
- **out** Buffer in which to store the generated signature.
- **outlen** Maximum length of the output buffer. Will store the bytes written to out upon successfully generating a message signature.
- **key** Pointer to a private ed25519_key with which to generate the signature.

See:

- wc_ed25519ctx_sign_msg
- wc_ed25519ph_sign_hash
- wc_ed25519ph_sign_msg
- wc_ed25519_verify_msg

Return:

- 0 Returned upon successfully generating a signature for the message.
- BAD_FUNC_ARG Returned if any of the input parameters evaluate to NULL, or if the output buffer is too small to store the generated signature.

- MEMORY_E Returned if there is an error allocating memory during function execution.

Example

```
ed25519_key key;
WC_RNG rng;
int ret, sigSz;

byte sig[64]; // will hold generated signature
sigSz = sizeof(sig);
byte message[] = { initialize with message };

wc_InitRng(&rng); // initialize rng
wc_ed25519_init(&key); // initialize key
wc_ed25519_make_key(&rng, 32, &key); // make public/private key pair
ret = wc_ed25519_sign_msg(message, sizeof(message), sig, &sigSz, &key);
if (ret != 0) {
    // error generating message signature
}
```

```
int wc_ed25519ctx_sign_msg(
    const byte * in,
    word32 inlen,
    byte * out,
    word32 * outlen,
    ed25519_key * key,
    const byte * context,
    byte contextLen
)
```

This function signs a message using an ed25519_key object to guarantee authenticity. The context is part of the data signed.

Parameters:

- **in** Pointer to the buffer containing the message to sign.
- **inlen** Length of the message to sign.
- **out** Buffer in which to store the generated signature.
- **outlen** Maximum length of the output buffer. Will store the bytes written to out upon successfully generating a message signature.
- **key** Pointer to a private ed25519_key with which to generate the signature.
- **context** Pointer to the buffer containing the context for which message is being signed.
- **contextLen** Length of the context buffer.

See:

- wc_ed25519_sign_msg
- wc_ed25519ph_sign_hash
- wc_ed25519ph_sign_msg
- wc_ed25519_verify_msg

Return:

- 0 Returned upon successfully generating a signature for the message.
- BAD_FUNC_ARG Returned any of the input parameters evaluate to NULL, or if the output buffer is too small to store the generated signature.
- MEMORY_E Returned if there is an error allocating memory during function execution.

Example

```
ed25519_key key;
WC_RNG rng;
int ret, sigSz;

byte sig[64]; // will hold generated signature
sigSz = sizeof(sig);
byte message[] = { initialize with message };
byte context[] = { initialize with context of signing };

wc_InitRng(&rng); // initialize rng
wc_ed25519_init(&key); // initialize key
wc_ed25519_make_key(&rng, 32, &key); // make public/private key pair
ret = wc_ed25519ctx_sign_msg(message, sizeof(message), sig, &sigSz, &key,
        context, sizeof(context));
if (ret != 0) {
    // error generating message signature
}
```

```
int wc_ed25519ph_sign_hash(
    const byte * hash,
    word32 hashLen,
    byte * out,
    word32 * outLen,
    ed25519_key * key,
    const byte * context,
    byte contextLen
)
```

This function signs a message digest using an ed25519_key object to guarantee authenticity. The context is included as part of the data signed. The message is pre-hashed before signature calculation.

Parameters:

- **hash** Pointer to the buffer containing the hash of the message to sign.
- **hashLen** Length of the hash of the message to sign.
- **out** Buffer in which to store the generated signature.
- **outlen** Maximum length of the output buffer. Will store the bytes written to out upon successfully generating a message signature.
- **key** Pointer to a private ed25519_key with which to generate the signature.
- **context** Pointer to the buffer containing the context for which message is being signed.
- **contextLen** Length of the context buffer.

See:

- wc_ed25519_sign_msg
- wc_ed25519ctx_sign_msg
- wc_ed25519ph_sign_msg
- wc_ed25519_verify_msg

Return:

- 0 Returned upon successfully generating a signature for the message digest.
- BAD_FUNC_ARG Returned any of the input parameters evaluate to NULL, or if the output buffer is too small to store the generated signature.
- MEMORY_E Returned if there is an error allocating memory during function execution.

Example

```
ed25519_key key;
WC_RNG rng;
int ret, sigSz;

byte sig[64]; // will hold generated signature
sigSz = sizeof(sig);
byte hash[] = { initialize with SHA-512 hash of message };
byte context[] = { initialize with context of signing };

wc_InitRng(&rng); // initialize rng
wc_ed25519_init(&key); // initialize key
wc_ed25519_make_key(&rng, 32, &key); // make public/private key pair
ret = wc_ed25519ph_sign_hash(hash, sizeof(hash), sig, &sigSz, &key,
        context, sizeof(context));
if (ret != 0) {
    // error generating message signature
}
```

```
int wc_ed25519ph_sign_msg(
    const byte * in,
    word32 inlen,
    byte * out,
    word32 * outlen,
    ed25519_key * key,
    const byte * context,
    byte contextLen
)
```

This function signs a message using an ed25519_key object to guarantee authenticity. The context is included as part of the data signed. The message is pre-hashed before signature calculation.

Parameters:

- **in** Pointer to the buffer containing the message to sign.

- **inlen** Length of the message to sign.
- **out** Buffer in which to store the generated signature.
- **outlen** Maximum length of the output buffer. Will store the bytes written to out upon successfully generating a message signature.
- **key** Pointer to a private ed25519_key with which to generate the signature.
- **context** Pointer to the buffer containing the context for which message is being signed.
- **contextLen** Length of the context buffer.

See:

- wc_ed25519_sign_msg
- wc_ed25519ctx_sign_msg
- wc_ed25519ph_sign_hash
- wc_ed25519_verify_msg

Return:

- 0 Returned upon successfully generating a signature for the message.
- BAD_FUNC_ARG Returned any of the input parameters evaluate to NULL, or if the output buffer is too small to store the generated signature.
- MEMORY_E Returned if there is an error allocating memory during function execution.

Example

```
ed25519_key key;
WC_RNG rng;
int ret, sigSz;

byte sig[64]; // will hold generated signature
sigSz = sizeof(sig);
byte message[] = { initialize with message };
byte context[] = { initialize with context of signing };

wc_InitRng(&rng); // initialize rng
wc_ed25519_init(&key); // initialize key
wc_ed25519_make_key(&rng, 32, &key); // make public/private key pair
ret = wc_ed25519ph_sign_msg(message, sizeof(message), sig, &sigSz, &key,
        context, sizeof(context));
if (ret != 0) {
    // error generating message signature
}
```

```
int wc_ed25519_verify_msg(
    const byte * sig,
    word32 siglen,
    const byte * msg,
    word32 msgLen,
    int * ret,
    ed25519_key * key
)
```

This function verifies the Ed25519 signature of a message to ensure authenticity. It returns the answer through ret, with 1 corresponding to a valid signature, and 0 corresponding to an invalid signature.

Parameters:

- **sig** Pointer to the buffer containing the signature to verify.
- **siglen** Length of the signature to verify.
- **msg** Pointer to the buffer containing the message to verify.
- **msgLen** Length of the message to verify.
- **ret** Pointer to the result of the verification. 1 indicates the message was successfully verified.
- **key** Pointer to a public Ed25519 key with which to verify the signature.

See:

- wc_ed25519ctx_verify_msg
- wc_ed25519ph_verify_hash
- wc_ed25519ph_verify_msg
- wc_ed25519_sign_msg

Return:

- 0 Returned upon successfully performing the signature verification and authentication.
- BAD_FUNC_ARG Returned if any of the input parameters evaluate to NULL, or if the siglen does not match the actual length of a signature.
- SIG_VERIFY_E Returned if verification completes, but the signature generated does not match the signature provided.

Example

```
ed25519_key key;
int ret, verified = 0;

byte sig[] { initialize with received signature };
byte msg[] = { initialize with message };
// initialize key with received public key
ret = wc_ed25519_verify_msg(sig, sizeof(sig), msg, sizeof(msg), &verified,
        &key);
if (ret < 0) {
    // error performing verification
} else if (verified == 0)
    // the signature is invalid
}
```

```
int wc_ed25519ctx_verify_msg(
    const byte * sig,
    word32 siglen,
    const byte * msg,
    word32 msgLen,
    int * ret,
    ed25519_key * key,
```

```
    const byte * context,
    byte contextLen
)
```

This function verifies the Ed25519 signature of a message to ensure authenticity. The context is included as part of the data verified. It returns the answer through ret, with 1 corresponding to a valid signature, and 0 corresponding to an invalid signature.

Parameters:

- **sig** Pointer to the buffer containing the signature to verify.
- **siglen** Length of the signature to verify.
- **msg** Pointer to the buffer containing the message to verify.
- **msgLen** Length of the message to verify.
- **ret** Pointer to the result of the verification. 1 indicates the message was successfully verified.
- **key** Pointer to a public Ed25519 key with which to verify the signature.
- **context** Pointer to the buffer containing the context for which the message was signed.
- **contextLen** Length of the context buffer.

See:

- wc_ed25519_verify_msg
- wc_ed25519ph_verify_hash
- wc_ed25519ph_verify_msg
- wc_ed25519_sign_msg

Return:

- 0 Returned upon successfully performing the signature verification and authentication.
- BAD_FUNC_ARG Returned if any of the input parameters evaluate to NULL, or if the siglen does not match the actual length of a signature.
- SIG_VERIFY_E Returned if verification completes, but the signature generated does not match the signature provided.

Example

```
ed25519_key key;
int ret, verified = 0;

byte sig[] { initialize with received signature };
byte msg[] = { initialize with message };
byte context[] = { initialize with context of signature };
// initialize key with received public key
ret = wc_ed25519ctx_verify_msg(sig, sizeof(sig), msg, sizeof(msg),
        &verified, &key, );
if (ret < 0) {
    // error performing verification
} else if (verified == 0)
    // the signature is invalid
}
```

```
int wc_ed25519ph_verify_hash(
    const byte * sig,
    word32 siglen,
    const byte * hash,
    word32 hashLen,
    int * ret,
    ed25519_key * key,
    const byte * context,
    byte contextLen
)
```

This function verifies the Ed25519 signature of the digest of a message to ensure authenticity. The context is included as part of the data verified. The hash is the pre-hashed message before signature calculation. The hash algorithm used to create message digest must be SHA-512. The answer is returned through ret, with 1 corresponding to a valid signature, and 0 corresponding to an invalid signature.

Parameters:

- **sig** Pointer to the buffer containing the signature to verify.
- **siglen** Length of the signature to verify.
- **hash** Pointer to the buffer containing the hash of the message to verify.
- **hashLen** Length of the hash to verify.
- **ret** Pointer to the result of the verification. 1 indicates the message was successfully verified.
- **key** Pointer to a public Ed25519 key with which to verify the signature.
- **context** Pointer to the buffer containing the context for which the message was signed.
- **contextLen** Length of the context buffer.

See:

- wc_ed25519_verify_msg
- wc_ed25519ctx_verify_msg
- wc_ed25519ph_verify_msg
- wc_ed25519_sign_msg

Return:

- 0 Returned upon successfully performing the signature verification and authentication.
- BAD_FUNC_ARG Returned if any of the input parameters evaluate to NULL, or if the siglen does not match the actual length of a signature.
- SIG_VERIFY_E Returned if verification completes, but the signature generated does not match the signature provided.

Example

```
ed25519_key key;
int ret, verified = 0;

byte sig[] { initialize with received signature };
byte hash[] = { initialize with SHA-512 hash of message };
byte context[] = { initialize with context of signature };
// initialize key with received public key
```

```
ret = wc_ed25519ph_verify_hash(sig, sizeof(sig), msg, sizeof(msg),
        &verified, &key, );
if (ret < 0) {
    // error performing verification
} else if (verified == 0)
    // the signature is invalid
}
```

```
int wc_ed25519ph_verify_msg(
    const byte * sig,
    word32 siglen,
    const byte * msg,
    word32 msgLen,
    int * ret,
    ed25519_key * key,
    const byte * context,
    byte contextLen
)
```

This function verifies the Ed25519 signature of a message to ensure authenticity. The context is included as part of the data verified. The message is pre-hashed before verification. It returns the answer through ret, with 1 corresponding to a valid signature, and 0 corresponding to an invalid signature.

Parameters:

- **sig** Pointer to the buffer containing the signature to verify.
- **siglen** Length of the signature to verify.
- **msg** Pointer to the buffer containing the message to verify.
- **msgLen** Length of the message to verify.
- **ret** Pointer to the result of the verification. 1 indicates the message was successfully verified.
- **key** Pointer to a public Ed25519 key with which to verify the signature.
- **context** Pointer to the buffer containing the context for which the message was signed.
- **contextLen** Length of the context buffer.

See:

- wc_ed25519_verify_msg
- wc_ed25519ph_verify_hash
- wc_ed25519ph_verify_msg
- wc_ed25519_sign_msg

Return:

- 0 Returned upon successfully performing the signature verification and authentication.
- BAD_FUNC_ARG Returned if any of the input parameters evaluate to NULL, or if the siglen does not match the actual length of a signature.
- SIG_VERIFY_E Returned if verification completes, but the signature generated does not match the signature provided.

Example

```
ed25519_key key;
int ret, verified = 0;

byte sig[] { initialize with received signature };
byte msg[] = { initialize with message };
byte context[] = { initialize with context of signature };
// initialize key with received public key
ret = wc_ed25519ctx_verify_msg(sig, sizeof(sig), msg, sizeof(msg),
        &verified, &key, );
if (ret < 0) {
    // error performing verification
} else if (verified == 0)
    // the signature is invalid
}
```

```
int wc_ed25519_init(
    ed25519_key * key
)
```

This function initializes an ed25519_key object for future use with message verification.

Parameters:

- **key** Pointer to the ed25519_key object to initialize.

See:

- wc_ed25519_make_key
- wc_ed25519_free

Return:

- 0 Returned upon successfully initializing the ed25519_key object.
- BAD_FUNC_ARG Returned if key is NULL.

Example

```
ed25519_key key;
wc_ed25519_init(&key);
```

```
void wc_ed25519_free(
    ed25519_key * key
)
```

This function frees an Ed25519 object after it has been used.

Parameters:

- **key** Pointer to the ed25519_key object to free

See: wc_ed25519_init

Example

```
ed25519_key key;
// initialize key and perform secure exchanges
...
wc_ed25519_free(&key);
```

```
int wc_ed25519_import_public(
    const byte * in,
    word32 inLen,
    ed25519_key * key
)
```

This function imports a public ed25519_key from a buffer containing the public key. This function will handle both compressed and uncompressed keys. The public key is checked that it matches the private key when one is present.

Parameters:

- **in** Pointer to the buffer containing the public key.
- **inLen** Length of the buffer containing the public key.
- **key** Pointer to the ed25519_key object in which to store the public key.

See:

- wc_ed25519_import_public_ex
- wc_ed25519_import_private_key
- wc_ed25519_import_private_key_ex
- wc_ed25519_export_public

Return:

- 0 Returned on successfully importing the ed25519_key.
- BAD_FUNC_ARG Returned if in or key evaluate to NULL, or inLen is less than the size of an Ed25519 key.

Example

```
int ret;
byte pub[] = { initialize Ed25519 public key };

ed_25519 key;
wc_ed25519_init_key(&key);
ret = wc_ed25519_import_public(pub, sizeof(pub), &key);
if (ret != 0) {
```

```
    // error importing key
}
```

```
int wc_ed25519_import_public_ex(
    const byte * in,
    word32 inLen,
    ed25519_key * key,
    int trusted
)
```

This function imports a public ed25519_key from a buffer containing the public key. This function will handle both compressed and uncompressed keys. Check public key matches private key, when present, when not trusted.

Parameters:

- **in** Pointer to the buffer containing the public key.
- **inLen** Length of the buffer containing the public key.
- **key** Pointer to the ed25519_key object in which to store the public key.
- **trusted** Public key data is trusted or not.

See:

- wc_ed25519_import_public
- wc_ed25519_import_private_key
- wc_ed25519_import_private_key_ex
- wc_ed25519_export_public

Return:

- 0 Returned on successfully importing the ed25519_key.
- BAD_FUNC_ARG Returned if in or key evaluate to NULL, or inLen is less than the size of an Ed25519 key.

Example

```
int ret;
byte pub[] = { initialize Ed25519 public key };

ed_25519 key;
wc_ed25519_init_key(&key);
ret = wc_ed25519_import_public_ex(pub, sizeof(pub), &key, 1);
if (ret != 0) {
    // error importing key
}
```

```
int wc_ed25519_import_private_only(
    const byte * priv,
    word32 privSz,
    ed25519_key * key
)
```

This function imports an Ed25519 private key only from a buffer.

Parameters:

- **priv** Pointer to the buffer containing the private key.
- **privSz** Length of the private key.
- **key** Pointer to the ed25519_key object in which to store the imported private key.

See:

- wc_ed25519_import_public
- wc_ed25519_import_public_ex
- wc_ed25519_import_private_key
- wc_ed25519_import_private_key_ex
- wc_ed25519_export_private_only

Return:

- 0 Returned on successfully importing the Ed25519 key.
- BAD_FUNC_ARG Returned if priv or key evaluate to NULL, or if privSz is not equal to ED25519_KEY_SIZE.

Example

```
int ret;
byte priv[] = { initialize with 32 byte private key };

ed25519_key key;
wc_ed25519_init_key(&key);
ret = wc_ed25519_import_private_only(priv, sizeof(priv), &key);
if (ret != 0) {
    // error importing private key
}
```

```
int wc_ed25519_import_private_key(
    const byte * priv,
    word32 privSz,
    const byte * pub,
    word32 pubSz,
    ed25519_key * key
)
```

This function imports a public/private Ed25519 key pair from a pair of buffers. This function will handle both compressed and uncompressed keys. The public key is assumed to be untrusted and is checked against the private key.

Parameters:

- **priv** Pointer to the buffer containing the private key.
- **privSz** Length of the private key.
- **pub** Pointer to the buffer containing the public key.
- **pubSz** Length of the public key.
- **key** Pointer to the ed25519_key object in which to store the imported private/public key pair.

See:

- wc_ed25519_import_public
- wc_ed25519_import_public_ex
- wc_ed25519_import_private_only
- wc_ed25519_import_private_key_ex
- wc_ed25519_export_private

Return:

- 0 Returned on successfully importing the ed25519_key.
- BAD_FUNC_ARG Returned if priv or key evaluate to NULL; or if either privSz is not equal to ED25519_KEY_SIZE nor ED25519_PRV_KEY_SIZE, or pubSz is less than ED25519_PUB_KEY_SIZE.

Example

```
int ret;
byte priv[] = { initialize with 32 byte private key };
byte pub[]  = { initialize with the corresponding public key };

ed25519_key key;
wc_ed25519_init_key(&key);
ret = wc_ed25519_import_private_key(priv, sizeof(priv), pub, sizeof(pub),
        &key);
if (ret != 0) {
    // error importing key
}
```

```
int wc_ed25519_import_private_key_ex(
    const byte * priv,
    word32 privSz,
    const byte * pub,
    word32 pubSz,
    ed25519_key * key,
    int trusted
)
```

This function imports a public/private Ed25519 key pair from a pair of buffers. This function will handle both compressed and uncompressed keys. The public is checked against private key if not trusted.

Parameters:

- **priv** Pointer to the buffer containing the private key.
- **privSz** Length of the private key.
- **pub** Pointer to the buffer containing the public key.
- **pubSz** Length of the public key.
- **key** Pointer to the ed25519_key object in which to store the imported private/public key pair.
- **trusted** Public key data is trusted or not.

See:

- wc_ed25519_import_public
- wc_ed25519_import_public_ex
- wc_ed25519_import_private_only
- wc_ed25519_import_private_key
- wc_ed25519_export_private

Return:

- 0 Returned on successfully importing the ed25519_key.
- BAD_FUNC_ARG Returned if priv or key evaluate to NULL; or if either privSz is not equal to ED25519_KEY_SIZE nor ED25519_PRV_KEY_SIZE, or pubSz is less than ED25519_PUB_KEY_SIZE.

Example

```
int ret;
byte priv[] = { initialize with 32 byte private key };
byte pub[]  = { initialize with the corresponding public key };

ed25519_key key;
wc_ed25519_init_key(&key);
ret = wc_ed25519_import_private_key(priv, sizeof(priv), pub, sizeof(pub),
        &key, 1);
if (ret != 0) {
    // error importing key
}
```

```
int wc_ed25519_export_public(
    ed25519_key * key,
    byte * out,
    word32 * outLen
)
```

This function exports the private key from an ed25519_key structure. It stores the public key in the buffer out, and sets the bytes written to this buffer in outLen.

Parameters:

- **key** Pointer to an ed25519_key structure from which to export the public key.
- **out** Pointer to the buffer in which to store the public key.
- **outLen** Pointer to a word32 object with the size available in out. Set with the number of bytes written to out after successfully exporting the public key.

See:

- wc_ed25519_import_public
- wc_ed25519_import_public_ex
- wc_ed25519_export_private_only

Return:

- 0 Returned upon successfully exporting the public key.
- BAD_FUNC_ARG Returned if any of the input values evaluate to NULL.
- BUFFER_E Returned if the buffer provided is not large enough to store the private key. Upon returning this error, the function sets the size required in outLen.

Example

```
int ret;
ed25519_key key;
// initialize key, make key

char pub[32];
word32 pubSz = sizeof(pub);

ret = wc_ed25519_export_public(&key, pub, &pubSz);
if (ret != 0) {
    // error exporting public key
}
```

```
int wc_ed25519_export_private_only(
    ed25519_key * key,
    byte * out,
    word32 * outLen
)
```

This function exports only the private key from an ed25519_key structure. It stores the private key in the buffer out, and sets the bytes written to this buffer in outLen.

Parameters:

- **key** Pointer to an ed25519_key structure from which to export the private key.
- **out** Pointer to the buffer in which to store the private key.
- **outLen** Pointer to a word32 object with the size available in out. Set with the number of bytes written to out after successfully exporting the private key.

See:

- wc_ed25519_export_public
- wc_ed25519_import_private_key
- wc_ed25519_import_private_key_ex

Return:

- 0 Returned upon successfully exporting the private key.
- BAD_FUNC_ARG Returned if any of the input values evaluate to NULL.
- BUFFER_E Returned if the buffer provided is not large enough to store the private key.

Example

```
int ret;
ed25519_key key;
// initialize key, make key

char priv[32]; // 32 bytes because only private key
word32 privSz = sizeof(priv);
ret = wc_ed25519_export_private_only(&key, priv, &privSz);
if (ret != 0) {
    // error exporting private key
}
```

```
int wc_ed25519_export_private(
    ed25519_key * key,
    byte * out,
    word32 * outLen
)
```

This function exports the key pair from an ed25519_key structure. It stores the key pair in the buffer out, and sets the bytes written to this buffer in outLen.

Parameters:

- **key** Pointer to an ed25519_key structure from which to export the key pair.
- **out** Pointer to the buffer in which to store the key pair.
- **outLen** Pointer to a word32 object with the size available in out. Set with the number of bytes written to out after successfully exporting the key pair.

See:

- wc_ed25519_import_private_key
- wc_ed25519_import_private_key_ex
- wc_ed25519_export_private_only

Return:

- 0 Returned upon successfully exporting the key pair.
- BAD_FUNC_ARG Returned if any of the input values evaluate to NULL.
- BUFFER_E Returned if the buffer provided is not large enough to store the key pair.

Example

```
ed25519_key key;
wc_ed25519_init(&key);

WC_RNG rng;
wc_InitRng(&rng);

wc_ed25519_make_key(&rng, 32, &key); // initialize 32 byte Ed25519 key

byte out[64]; // out needs to be a sufficient buffer size
word32 outLen = sizeof(out);
int key_size = wc_ed25519_export_private(&key, out, &outLen);
if (key_size == BUFFER_E) {
    // Check size of out compared to outLen to see if function reset outLen
}
```

```
int wc_ed25519_export_key(
    ed25519_key * key,
    byte * priv,
    word32 * privSz,
    byte * pub,
    word32 * pubSz
)
```

This function exports the private and public key separately from an ed25519_key structure. It stores the private key in the buffer priv, and sets the bytes written to this buffer in privSz. It stores the public key in the buffer pub, and sets the bytes written to this buffer in pubSz.

Parameters:

- **key** Pointer to an ed25519_key structure from which to export the key pair.
- **priv** Pointer to the buffer in which to store the private key.
- **privSz** Pointer to a word32 object with the size available in out. Set with the number of bytes written to out after successfully exporting the private key.
- **pub** Pointer to the buffer in which to store the public key.
- **pubSz** Pointer to a word32 object with the size available in out. Set with the number of bytes written to out after successfully exporting the public key.

See:

- wc_ed25519_export_private
- wc_ed25519_export_public

Return:

- 0 Returned upon successfully exporting the key pair.
- BAD_FUNC_ARG Returned if any of the input values evaluate to NULL.
- BUFFER_E Returned if the buffer provided is not large enough to store the key pair.

Example

```
int ret;
ed25519_key key;
// initialize key, make key

char pub[32];
word32 pubSz = sizeof(pub);
char priv[32];
word32 privSz = sizeof(priv);

ret = wc_ed25519_export_key(&key, priv, &pubSz, pub, &pubSz);
if (ret != 0) {
    // error exporting public key
}
```

```
int wc_ed25519_check_key(
    ed25519_key * key
)
```

This function checks the public key in ed25519_key structure matches the private key.

Parameters:

- **key** Pointer to an ed25519_key structure holding a private and public key.

See:

- wc_ed25519_import_private_key
- wc_ed25519_import_private_key_ex

Return:

- 0 Returned if the private and public key matched.
- BAD_FUNC_ARG Returned if the given key is NULL.
- PUBLIC_KEY_E Returned if the no public key available or is invalid.

Example

```
int ret;
byte priv[] = { initialize with 57 byte private key };
byte pub[]  = { initialize with the corresponding public key };

ed25519_key key;
wc_ed25519_init_key(&key);
wc_ed25519_import_private_key_ex(priv, sizeof(priv), pub, sizeof(pub), &key,
    1);
ret = wc_ed25519_check_key(&key);
if (ret != 0) {
    // error checking key
}
```

```
int wc_ed25519_size(
    ed25519_key * key
)
```

This function returns the size of an Ed25519 - 32 bytes.

Parameters:

- **key** Pointer to an ed25519_key structure for which to get the key size.

See: wc_ed25519_make_key

Return:

- ED25519_KEY_SIZE The size of a valid private key (32 bytes).
- BAD_FUNC_ARG Returned if the given key is NULL.

Example

```
int keySz;
ed25519_key key;
// initialize key, make key
keySz = wc_ed25519_size(&key);
if (keySz == 0) {
    // error determining key size
}
```

```
int wc_ed25519_priv_size(
    ed25519_key * key
)
```

This function returns the private key size (secret + public) in bytes.

Parameters:

- **key** Pointer to an ed25519_key structure for which to get the key size.

See: wc_ed25519_pub_size

Return:

- ED25519_PRV_KEY_SIZE The size of the private key (64 bytes).
- BAD_FUNC_ARG Returned if key argument is NULL.

Example

```
ed25519_key key;
wc_ed25519_init(&key);

WC_RNG rng;
```

```
wc_InitRng(&rng);

wc_ed25519_make_key(&rng, 32, &key); // initialize 32 byte Ed25519 key
int key_size = wc_ed25519_priv_size(&key);
```

```
int wc_ed25519_pub_size(
    ed25519_key * key
)
```

This function returns the compressed key size in bytes (public key).

Parameters:

- **key** Pointer to an ed25519_key structure for which to get the key size.

See: wc_ed25519_priv_size

Return:

- ED25519_PUB_KEY_SIZE The size of the compressed public key (32 bytes).
- BAD_FUNC_ARG Returns if key argument is NULL.

Example

```
ed25519_key key;
wc_ed25519_init(&key);
WC_RNG rng;
wc_InitRng(&rng);

wc_ed25519_make_key(&rng, 32, &key); // initialize 32 byte Ed25519 key
int key_size = wc_ed25519_pub_size(&key);
```

```
int wc_ed25519_sig_size(
    ed25519_key * key
)
```

This function returns the size of an Ed25519 signature (64 in bytes).

Parameters:

- **key** Pointer to an ed25519_key structure for which to get the signature size.

See: wc_ed25519_sign_msg

Return:

- ED25519_SIG_SIZE The size of an Ed25519 signature (64 bytes).
- BAD_FUNC_ARG Returns if key argument is NULL.

Example

```
int sigSz;
ed25519_key key;
// initialize key, make key

sigSz = wc_ed25519_sig_size(&key);
if (sigSz == 0) {
    // error determining sig size
}
```

B.26 Algorithms - ED448

B.25.2.26 function wc_ed25519_sig_size

B.26.1 Functions

	Name
int	**wc_ed448_make_public**(ed448_key * key, unsigned char * pubKey, word32 pubKeySz)This function generates the Ed448 public key from the private key. It stores the public key in the buffer pubKey, and sets the bytes written to this buffer in pubKeySz.
int	**wc_ed448_make_key**(WC_RNG * rng, int keysize, ed448_key * key)This function generates a new Ed448 key and stores it in key.
int	**wc_ed448_sign_msg**(const byte * in, word32 inlen, byte * out, word32 * outlen, ed448_key * key)This function signs a message using an ed448_key object to guarantee authenticity.
int	**wc_ed448ph_sign_hash**(const byte * hash, word32 hashLen, byte * out, word32 * outLen, ed448_key * key, const byte * context, byte contextLen)This function signs a message digest using an ed448_key object to guarantee authenticity. The context is included as part of the data signed. The hash is the pre-hashed message before signature calculation.
int	**wc_ed448ph_sign_msg**(const byte * in, word32 inLen, byte * out, word32 * outLen, ed448_key * key, const byte * context, byte contextLen)This function signs a message using an ed448_key object to guarantee authenticity. The context is included as part of the data signed. The message is pre-hashed before signature calculation.

	Name
int	**wc_ed448_verify_msg**(const byte * sig, word32 siglen, const byte * msg, word32 msgLen, int * res, ed448_key * key, const byte * context, byte contextLen)This function verifies the Ed448 signature of a message to ensure authenticity. The context is included as part of the data verified. The answer is returned through res, with 1 corresponding to a valid signature, and 0 corresponding to an invalid signature.
int	**wc_ed448ph_verify_hash**(const byte * sig, word32 siglen, const byte * hash, word32 hashlen, int * res, ed448_key * key, const byte * context, byte contextLen)This function verifies the Ed448 signature of the digest of a message to ensure authenticity. The context is included as part of the data verified. The hash is the pre-hashed message before signature calculation. The answer is returned through res, with 1 corresponding to a valid signature, and 0 corresponding to an invalid signature.
int	**wc_ed448ph_verify_msg**(const byte * sig, word32 siglen, const byte * msg, word32 msgLen, int * res, ed448_key * key, const byte * context, byte contextLen)This function verifies the Ed448 signature of a message to ensure authenticity. The context is included as part of the data verified. The message is pre-hashed before verification. The answer is returned through res, with 1 corresponding to a valid signature, and 0 corresponding to an invalid signature.
int	**wc_ed448_init**(ed448_key * key)This function initializes an ed448_key object for future use with message verification.
void	**wc_ed448_free**(ed448_key * key)This function frees an Ed448 object after it has been used.
int	**wc_ed448_import_public**(const byte * in, word32 inLen, ed448_key * key)This function imports a public ed448_key pair from a buffer containing the public key. This function will handle both compressed and uncompressed keys. The public key is checked that it matches the private key when one is present.
int	**wc_ed448_import_public_ex**(const byte * in, word32 inLen, ed448_key * key, int trusted)This function imports a public ed448_key pair from a buffer containing the public key. This function will handle both compressed and uncompressed keys. Check public key matches private key, when present, when not trusted.

	Name
int	**wc_ed448_import_private_only**(const byte * priv, word32 privSz, ed448_key * key)This function imports an Ed448 private key only from a buffer.
int	**wc_ed448_import_private_key**(const byte * priv, word32 privSz, const byte * pub, word32 pubSz, ed448_key * key)This function imports a public/private Ed448 key pair from a pair of buffers. This function will handle both compressed and uncompressed keys.
int	**wc_ed448_import_private_key_ex**(const byte * priv, word32 privSz, const byte * pub, word32 pubSz, ed448_key * key, int trusted)This function imports a public/private Ed448 key pair from a pair of buffers. This function will handle both compressed and uncompressed keys. The public is checked against private key if not trusted.
int	**wc_ed448_export_public**(ed448_key * key, byte * out, word32 * outLen)This function exports the private key from an ed448_key structure. It stores the public key in the buffer out, and sets the bytes written to this buffer in outLen.
int	**wc_ed448_export_private_only**(ed448_key * key, byte * out, word32 * outLen)This function exports only the private key from an ed448_key structure. It stores the private key in the buffer out, and sets the bytes written to this buffer in outLen.
int	**wc_ed448_export_private**(ed448_key * key, byte * out, word32 * outLen)This function exports the key pair from an ed448_key structure. It stores the key pair in the buffer out, and sets the bytes written to this buffer in outLen.
int	**wc_ed448_export_key**(ed448_key * key, byte * priv, word32 * privSz, byte * pub, word32 * pubSz)This function exports the private and public key separately from an ed448_key structure. It stores the private key in the buffer priv, and sets the bytes written to this buffer in privSz. It stores the public key in the buffer pub, and sets the bytes written to this buffer in pubSz.
int	**wc_ed448_check_key**(ed448_key * key)This function checks the public key in ed448_key structure matches the private key.
int	**wc_ed448_size**(ed448_key * key)This function returns the size of an Ed448 private key - 57 bytes.

	Name
int	**wc_ed448_priv_size**(ed448_key * key)This function returns the private key size (secret + public) in bytes.
int	**wc_ed448_pub_size**(ed448_key * key)This function returns the compressed key size in bytes (public key).
int	**wc_ed448_sig_size**(ed448_key * key)This function returns the size of an Ed448 signature (114 in bytes).

B.26.2 Functions Documentation

```
int wc_ed448_make_public(
    ed448_key * key,
    unsigned char * pubKey,
    word32 pubKeySz
)
```

This function generates the Ed448 public key from the private key. It stores the public key in the buffer pubKey, and sets the bytes written to this buffer in pubKeySz.

Parameters:

- **key** Pointer to the ed448_key for which to generate a key.
- **out** Pointer to the buffer in which to store the public key.
- **outLen** Pointer to a word32 object with the size available in out. Set with the number of bytes written to out after successfully exporting the public key.

See:

- wc_ed448_init
- wc_ed448_import_private_only
- wc_ed448_make_key

Return:

- 0 Returned upon successfully making the public key.
- BAD_FUNC_ARG Returned ifi key or pubKey evaluate to NULL, or if the specified key size is not 57 bytes (Ed448 has 57 byte keys).
- MEMORY_E Returned if there is an error allocating memory during function execution.

Example

```
int ret;

ed448_key key;
byte priv[] = { initialize with 57 byte private key };
byte pub[57];
word32 pubSz = sizeof(pub);
```

```
wc_ed448_init(&key);
wc_ed448_import_private_only(priv, sizeof(priv), &key);
ret = wc_ed448_make_public(&key, pub, &pubSz);
if (ret != 0) {
    // error making public key
}
```

```
int wc_ed448_make_key(
    WC_RNG * rng,
    int keysize,
    ed448_key * key
)
```

This function generates a new Ed448 key and stores it in key.

Parameters:

- **rng** Pointer to an initialized RNG object with which to generate the key.
- **keysize** Length of key to generate. Should always be 57 for Ed448.
- **key** Pointer to the ed448_key for which to generate a key.

See: wc_ed448_init

Return:

- 0 Returned upon successfully making an ed448_key.
- BAD_FUNC_ARG Returned if rng or key evaluate to NULL, or if the specified key size is not 57 bytes (Ed448 has 57 byte keys).
- MEMORY_E Returned if there is an error allocating memory during function execution.

Example

```
int ret;

WC_RNG rng;
ed448_key key;

wc_InitRng(&rng);
wc_ed448_init(&key);
ret = wc_ed448_make_key(&rng, 57, &key);
if (ret != 0) {
    // error making key
}
```

```
int wc_ed448_sign_msg(
    const byte * in,
    word32 inlen,
    byte * out,
    word32 * outlen,
```

```
    ed448_key * key
)
```

This function signs a message using an ed448_key object to guarantee authenticity.

Parameters:

- **in** Pointer to the buffer containing the message to sign.
- **inlen** Length of the message to sign.
- **out** Buffer in which to store the generated signature.
- **outlen** Maximum length of the output buffer. Will store the bytes written to out upon successfully generating a message signature.
- **key** Pointer to a private ed448_key with which to generate the signature.

See:

- wc_ed448ph_sign_hash
- wc_ed448ph_sign_msg
- wc_ed448_verify_msg

Return:

- 0 Returned upon successfully generating a signature for the message.
- BAD_FUNC_ARG Returned if any of the input parameters evaluate to NULL, or if the output buffer is too small to store the generated signature.
- MEMORY_E Returned if there is an error allocating memory during function execution.

Example

```
ed448_key key;
WC_RNG rng;
int ret, sigSz;

byte sig[114]; // will hold generated signature
sigSz = sizeof(sig);
byte message[] = { initialize with message };

wc_InitRng(&rng); // initialize rng
wc_ed448_init(&key); // initialize key
wc_ed448_make_key(&rng, 57, &key); // make public/private key pair
ret = wc_ed448_sign_msg(message, sizeof(message), sig, &sigSz, &key);
if (ret != 0 ) {
    // error generating message signature
}
```

```
int wc_ed448ph_sign_hash(
    const byte * hash,
    word32 hashLen,
    byte * out,
    word32 * outLen,
```

```
    ed448_key * key,
    const byte * context,
    byte contextLen
)
```

This function signs a message digest using an ed448_key object to guarantee authenticity. The context is included as part of the data signed. The hash is the pre-hashed message before signature calculation.

Parameters:

- **hash** Pointer to the buffer containing the hash of the message to sign.
- **hashLen** Length of the hash of the message to sign.
- **out** Buffer in which to store the generated signature.
- **outlen** Maximum length of the output buffer. Will store the bytes written to out upon successfully generating a message signature.
- **key** Pointer to a private ed448_key with which to generate the signature.
- **context** Pointer to the buffer containing the context for which message is being signed.
- **contextLen** Length of the context buffer.

See:

- wc_ed448_sign_msg
- wc_ed448ph_sign_msg
- wc_ed448ph_verify_hash

Return:

- 0 Returned upon successfully generating a signature for the message digest.
- BAD_FUNC_ARG Returned any of the input parameters evaluate to NULL, or if the output buffer is too small to store the generated signature.
- MEMORY_E Returned if there is an error allocating memory during function execution.

Example

```
ed448_key key;
WC_RNG rng;
int ret, sigSz;

byte sig[114]; // will hold generated signature
sigSz = sizeof(sig);
byte hash[] = { initialize hash of message };
byte context[] = { initialize with context of signing };

wc_InitRng(&rng); // initialize rng
wc_ed448_init(&key); // initialize key
wc_ed448_make_key(&rng, 57, &key); // make public/private key pair
ret = wc_ed448ph_sign_hash(hash, sizeof(hash), sig, &sigSz, &key,
        context, sizeof(context));
if (ret != 0) {
    // error generating message signature
}
```

```
int wc_ed448ph_sign_msg(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 * outLen,
    ed448_key * key,
    const byte * context,
    byte contextLen
)
```

This function signs a message using an ed448_key object to guarantee authenticity. The context is included as part of the data signed. The message is pre-hashed before signature calculation.

Parameters:

- **in** Pointer to the buffer containing the message to sign.
- **inlen** Length of the message to sign.
- **out** Buffer in which to store the generated signature.
- **outlen** Maximum length of the output buffer. Will store the bytes written to out upon successfully generating a message signature.
- **key** Pointer to a private ed448_key with which to generate the signature.
- **context** Pointer to the buffer containing the context for which message is being signed.
- **contextLen** Length of the context buffer.

See:

- wc_ed448_sign_msg
- wc_ed448ph_sign_hash
- wc_ed448ph_verify_msg

Return:

- 0 Returned upon successfully generating a signature for the message.
- BAD_FUNC_ARG Returned any of the input parameters evaluate to NULL, or if the output buffer is too small to store the generated signature.
- MEMORY_E Returned if there is an error allocating memory during function execution.

Example

```
ed448_key key;
WC_RNG rng;
int ret, sigSz;

byte sig[114]; // will hold generated signature
sigSz = sizeof(sig);
byte message[] = { initialize with message };
byte context[] = { initialize with context of signing };

wc_InitRng(&rng); // initialize rng
wc_ed448_init(&key); // initialize key
```

```
wc_ed448_make_key(&rng, 57, &key); // make public/private key pair
ret = wc_ed448ph_sign_msg(message, sizeof(message), sig, &sigSz, &key,
        context, sizeof(context));
if (ret != 0) {
    // error generating message signature
}
```

```
int wc_ed448_verify_msg(
    const byte * sig,
    word32 siglen,
    const byte * msg,
    word32 msgLen,
    int * res,
    ed448_key * key,
    const byte * context,
    byte contextLen
)
```

This function verifies the Ed448 signature of a message to ensure authenticity. The context is included as part of the data verified. The answer is returned through res, with 1 corresponding to a valid signature, and 0 corresponding to an invalid signature.

Parameters:

- **sig** Pointer to the buffer containing the signature to verify.
- **siglen** Length of the signature to verify.
- **msg** Pointer to the buffer containing the message to verify.
- **msgLen** Length of the message to verify.
- **key** Pointer to a public Ed448 key with which to verify the signature.
- **context** Pointer to the buffer containing the context for which the message was signed.
- **contextLen** Length of the context buffer.

See:

- wc_ed448ph_verify_hash
- wc_ed448ph_verify_msg
- wc_ed448_sign_msg

Return:

- 0 Returned upon successfully performing the signature verification and authentication.
- BAD_FUNC_ARG Returned if any of the input parameters evaluate to NULL, or if the siglen does not match the actual length of a signature.
- SIG_VERIFY_E Returned if verification completes, but the signature generated does not match the signature provided.

Example

```
ed448_key key;
int ret, verified = 0;
```

```
byte sig[] { initialize with received signature };
byte msg[] = { initialize with message };
byte context[] = { initialize with context of signature };
// initialize key with received public key
ret = wc_ed448_verify_msg(sig, sizeof(sig), msg, sizeof(msg), &verified,
        &key, context, sizeof(context));
if (ret < 0) {
    // error performing verification
} else if (verified == 0)
    // the signature is invalid
}
```

```
int wc_ed448ph_verify_hash(
    const byte * sig,
    word32 siglen,
    const byte * hash,
    word32 hashlen,
    int * res,
    ed448_key * key,
    const byte * context,
    byte contextLen
)
```

This function verifies the Ed448 signature of the digest of a message to ensure authenticity. The context is included as part of the data verified. The hash is the pre-hashed message before signature calculation. The answer is returned through res, with 1 corresponding to a valid signature, and 0 corresponding to an invalid signature.

Parameters:

- **sig** Pointer to the buffer containing the signature to verify.
- **siglen** Length of the signature to verify.
- **hash** Pointer to the buffer containing the hash of the message to verify.
- **hashLen** Length of the hash to verify.
- **key** Pointer to a public Ed448 key with which to verify the signature.
- **context** Pointer to the buffer containing the context for which the message was signed.
- **contextLen** Length of the context buffer.

See:

- wc_ed448_verify_msg
- wc_ed448ph_verify_msg
- wc_ed448ph_sign_hash

Return:

- 0 Returned upon successfully performing the signature verification and authentication.
- BAD_FUNC_ARG Returned if any of the input parameters evaluate to NULL, or if the siglen does not match the actual length of a signature.

- SIG_VERIFY_E Returned if verification completes, but the signature generated does not match the signature provided.

Example

```
ed448_key key;
int ret, verified = 0;

byte sig[] { initialize with received signature };
byte hash[] = { initialize hash of message };
byte context[] = { initialize with context of signature };
// initialize key with received public key
ret = wc_ed448ph_verify_hash(sig, sizeof(sig), hash, sizeof(hash),
        &verified, &key, context, sizeof(context));
if (ret < 0) {
    // error performing verification
} else if (verified == 0)
    // the signature is invalid
}
```

```
int wc_ed448ph_verify_msg(
    const byte * sig,
    word32 siglen,
    const byte * msg,
    word32 msgLen,
    int * res,
    ed448_key * key,
    const byte * context,
    byte contextLen
)
```

This function verifies the Ed448 signature of a message to ensure authenticity. The context is included as part of the data verified. The message is pre-hashed before verification. The answer is returned through res, with 1 corresponding to a valid signature, and 0 corresponding to an invalid signature.

Parameters:

- **sig** Pointer to the buffer containing the signature to verify.
- **siglen** Length of the signature to verify.
- **msg** Pointer to the buffer containing the message to verify.
- **msgLen** Length of the message to verify.
- **key** Pointer to a public Ed448 key with which to verify the signature.
- **context** Pointer to the buffer containing the context for which the message was signed.
- **contextLen** Length of the context buffer.

See:

- wc_ed448_verify_msg
- wc_ed448ph_verify_hash
- wc_ed448ph_sign_msg

Return:

- 0 Returned upon successfully performing the signature verification and authentication.
- BAD_FUNC_ARG Returned if any of the input parameters evaluate to NULL, or if the siglen does not match the actual length of a signature.
- SIG_VERIFY_E Returned if verification completes, but the signature generated does not match the signature provided.

Example

```
ed448_key key;
int ret, verified = 0;

byte sig[] { initialize with received signature };
byte msg[] = { initialize with message };
byte context[] = { initialize with context of signature };
// initialize key with received public key
ret = wc_ed448ph_verify_msg(sig, sizeof(sig), msg, sizeof(msg), &verified,
        &key, context, sizeof(context));
if (ret < 0) {
    // error performing verification
} else if (verified == 0)
    // the signature is invalid
}
```

```
int wc_ed448_init(
    ed448_key * key
)
```

This function initializes an ed448_key object for future use with message verification.

Parameters:

- **key** Pointer to the ed448_key object to initialize.

See:

- wc_ed448_make_key
- wc_ed448_free

Return:

- 0 Returned upon successfully initializing the ed448_key object.
- BAD_FUNC_ARG Returned if key is NULL.

Example

```
ed448_key key;
wc_ed448_init(&key);
```

```
void wc_ed448_free(
    ed448_key * key
)
```

This function frees an Ed448 object after it has been used.

Parameters:

- **key** Pointer to the ed448_key object to free

See: wc_ed448_init

Example

```
ed448_key key;
// initialize key and perform secure exchanges
...
wc_ed448_free(&key);
```

```
int wc_ed448_import_public(
    const byte * in,
    word32 inLen,
    ed448_key * key
)
```

This function imports a public ed448_key pair from a buffer containing the public key. This function will handle both compressed and uncompressed keys. The public key is checked that it matches the private key when one is present.

Parameters:

- **in** Pointer to the buffer containing the public key.
- **inLen** Length of the buffer containing the public key.
- **key** Pointer to the ed448_key object in which to store the public key.

See:

- wc_ed448_import_public_ex
- wc_ed448_import_private_key
- wc_ed448_import_private_key_ex
- wc_ed448_export_public

Return:

- 0 Returned on successfully importing the ed448_key.
- BAD_FUNC_ARG Returned if in or key evaluate to NULL, or inLen is less than the size of an Ed448 key.

Example

```
int ret;
byte pub[] = { initialize Ed448 public key };

ed_448 key;
wc_ed448_init_key(&key);
ret = wc_ed448_import_public(pub, sizeof(pub), &key);
if (ret != 0) {
    // error importing key
}
```

```
int wc_ed448_import_public_ex(
    const byte * in,
    word32 inLen,
    ed448_key * key,
    int trusted
)
```

This function imports a public ed448_key pair from a buffer containing the public key. This function will handle both compressed and uncompressed keys. Check public key matches private key, when present, when not trusted.

Parameters:

- **in** Pointer to the buffer containing the public key.
- **inLen** Length of the buffer containing the public key.
- **key** Pointer to the ed448_key object in which to store the public key.
- **trusted** Public key data is trusted or not.

See:

- wc_ed448_import_public
- wc_ed448_import_private_key
- wc_ed448_import_private_key_ex
- wc_ed448_export_public

Return:

- 0 Returned on successfully importing the ed448_key.
- BAD_FUNC_ARG Returned if in or key evaluate to NULL, or inLen is less than the size of an Ed448 key.

Example

```
int ret;
byte pub[] = { initialize Ed448 public key };

ed_448 key;
```

```
wc_ed448_init_key(&key);
ret = wc_ed448_import_public_ex(pub, sizeof(pub), &key, 1);
if (ret != 0) {
    // error importing key
}
```

```
int wc_ed448_import_private_only(
    const byte * priv,
    word32 privSz,
    ed448_key * key
)
```

This function imports an Ed448 private key only from a buffer.

Parameters:

- **priv** Pointer to the buffer containing the private key.
- **privSz** Length of the private key.
- **key** Pointer to the ed448_key object in which to store the imported private key.

See:

- wc_ed448_import_public
- wc_ed448_import_public_ex
- wc_ed448_import_private_key
- wc_ed448_import_private_key_ex
- wc_ed448_export_private_only

Return:

- 0 Returned on successfully importing the Ed448 private key.
- BAD_FUNC_ARG Returned if in or key evaluate to NULL, or if privSz is less than ED448_KEY_SIZE.

Example

```
int ret;
byte priv[] = { initialize with 57 byte private key };

ed448_key key;
wc_ed448_init_key(&key);
ret = wc_ed448_import_private_only(priv, sizeof(priv), &key);
if (ret != 0) {
    // error importing private key
}
```

```
int wc_ed448_import_private_key(
    const byte * priv,
    word32 privSz,
```

```
    const byte * pub,
    word32 pubSz,
    ed448_key * key
)
```

This function imports a public/private Ed448 key pair from a pair of buffers. This function will handle both compressed and uncompressed keys.

Parameters:

- **priv** Pointer to the buffer containing the private key.
- **privSz** Length of the private key.
- **pub** Pointer to the buffer containing the public key.
- **pubSz** Length of the public key.
- **key** Pointer to the ed448_key object in which to store the imported private/public key pair.

See:

- wc_ed448_import_public
- wc_ed448_import_public_ex
- wc_ed448_import_private_only
- wc_ed448_import_private_key_ex
- wc_ed448_export_private

Return:

- 0 Returned on successfully importing the Ed448 key.
- BAD_FUNC_ARG Returned if in or key evaluate to NULL, or if either privSz is less than ED448_KEY_SIZE or pubSz is less than ED448_PUB_KEY_SIZE.

Example

```
int ret;
byte priv[] = { initialize with 57 byte private key };
byte pub[]  = { initialize with the corresponding public key };

ed448_key key;
wc_ed448_init_key(&key);
ret = wc_ed448_import_private_key(priv, sizeof(priv), pub, sizeof(pub),
        &key);
if (ret != 0) {
    // error importing key
}
```

```
int wc_ed448_import_private_key_ex(
    const byte * priv,
    word32 privSz,
    const byte * pub,
    word32 pubSz,
    ed448_key * key,
```

```
    int trusted
)
```

This function imports a public/private Ed448 key pair from a pair of buffers. This function will handle both compressed and uncompressed keys. The public is checked against private key if not trusted.

Parameters:

- **priv** Pointer to the buffer containing the private key.
- **privSz** Length of the private key.
- **pub** Pointer to the buffer containing the public key.
- **pubSz** Length of the public key.
- **key** Pointer to the ed448_key object in which to store the imported private/public key pair.
- **trusted** Public key data is trusted or not.

See:

- wc_ed448_import_public
- wc_ed448_import_public_ex
- wc_ed448_import_private_only
- wc_ed448_import_private_key
- wc_ed448_export_private

Return:

- 0 Returned on successfully importing the Ed448 key.
- BAD_FUNC_ARG Returned if in or key evaluate to NULL, or if either privSz is less than ED448_KEY_SIZE or pubSz is less than ED448_PUB_KEY_SIZE.

Example

```
int ret;
byte priv[] = { initialize with 57 byte private key };
byte pub[]  = { initialize with the corresponding public key };

ed448_key key;
wc_ed448_init_key(&key);
ret = wc_ed448_import_private_key_ex(priv, sizeof(priv), pub, sizeof(pub),
        &key, 1);
if (ret != 0) {
    // error importing key
}
```

```
int wc_ed448_export_public(
    ed448_key * key,
    byte * out,
    word32 * outLen
)
```

This function exports the private key from an ed448_key structure. It stores the public key in the buffer out, and sets the bytes written to this buffer in outLen.

Parameters:

- **key** Pointer to an ed448_key structure from which to export the public key.
- **out** Pointer to the buffer in which to store the public key.
- **outLen** Pointer to a word32 object with the size available in out. Set with the number of bytes written to out after successfully exporting the public key.

See:

- wc_ed448_import_public
- wc_ed448_import_public_ex
- wc_ed448_export_private_only

Return:

- 0 Returned upon successfully exporting the public key.
- BAD_FUNC_ARG Returned if any of the input values evaluate to NULL.
- BUFFER_E Returned if the buffer provided is not large enough to store the private key. Upon returning this error, the function sets the size required in outLen.

Example

```
int ret;
ed448_key key;
// initialize key, make key

char pub[57];
word32 pubSz = sizeof(pub);

ret = wc_ed448_export_public(&key, pub, &pubSz);
if (ret != 0) {
    // error exporting public key
}
```

```
int wc_ed448_export_private_only(
    ed448_key * key,
    byte * out,
    word32 * outLen
)
```

This function exports only the private key from an ed448_key structure. It stores the private key in the buffer out, and sets the bytes written to this buffer in outLen.

Parameters:

- **key** Pointer to an ed448_key structure from which to export the private key.
- **out** Pointer to the buffer in which to store the private key.

- **outLen** Pointer to a word32 object with the size available in out. Set with the number of bytes written to out after successfully exporting the private key.

See:

- wc_ed448_export_public
- wc_ed448_import_private_key
- wc_ed448_import_private_key_ex

Return:

- 0 Returned upon successfully exporting the private key.
- ECC_BAD_ARG_E Returned if any of the input values evaluate to NULL.
- BUFFER_E Returned if the buffer provided is not large enough to store the private key.

Example

```
int ret;
ed448_key key;
// initialize key, make key

char priv[57]; // 57 bytes because only private key
word32 privSz = sizeof(priv);
ret = wc_ed448_export_private_only(&key, priv, &privSz);
if (ret != 0) {
    // error exporting private key
}
```

```
int wc_ed448_export_private(
    ed448_key * key,
    byte * out,
    word32 * outLen
)
```

This function exports the key pair from an ed448_key structure. It stores the key pair in the buffer out, and sets the bytes written to this buffer in outLen.

Parameters:

- **key** Pointer to an ed448_key structure from which to export the key pair.
- **out** Pointer to the buffer in which to store the key pair.
- **outLen** Pointer to a word32 object with the size available in out. Set with the number of bytes written to out after successfully exporting the key pair.

See:

- wc_ed448_import_private
- wc_ed448_export_private_only

Return:

- 0 Returned upon successfully exporting the key pair.
- ECC_BAD_ARG_E Returned if any of the input values evaluate to NULL.
- BUFFER_E Returned if the buffer provided is not large enough to store the key pair.

Example

```
ed448_key key;
wc_ed448_init(&key);

WC_RNG rng;
wc_InitRng(&rng);

wc_ed448_make_key(&rng, 57, &key); // initialize 57 byte Ed448 key

byte out[114]; // out needs to be a sufficient buffer size
word32 outLen = sizeof(out);
int key_size = wc_ed448_export_private(&key, out, &outLen);
if (key_size == BUFFER_E) {
    // Check size of out compared to outLen to see if function reset outLen
}
```

```
int wc_ed448_export_key(
    ed448_key * key,
    byte * priv,
    word32 * privSz,
    byte * pub,
    word32 * pubSz
)
```

This function exports the private and public key separately from an ed448_key structure. It stores the private key in the buffer priv, and sets the bytes written to this buffer in privSz. It stores the public key in the buffer pub, and sets the bytes written to this buffer in pubSz.

Parameters:

- **key** Pointer to an ed448_key structure from which to export the key pair.
- **priv** Pointer to the buffer in which to store the private key.
- **privSz** Pointer to a word32 object with the size available in out. Set with the number of bytes written to out after successfully exporting the private key.
- **pub** Pointer to the buffer in which to store the public key.
- **pubSz** Pointer to a word32 object with the size available in out. Set with the number of bytes written to out after successfully exporting the public key.

See:

- wc_ed448_export_private
- wc_ed448_export_public

Return:

- 0 Returned upon successfully exporting the key pair.
- ECC_BAD_ARG_E Returned if any of the input values evaluate to NULL.
- BUFFER_E Returned if the buffer provided is not large enough to store the key pair.

Example

```
int ret;
ed448_key key;
// initialize key, make key

char pub[57];
word32 pubSz = sizeof(pub);
char priv[57];
word32 privSz = sizeof(priv);

ret = wc_ed448_export_key(&key, priv, &pubSz, pub, &pubSz);
if (ret != 0) {
    // error exporting private and public key
}
```

```
int wc_ed448_check_key(
    ed448_key * key
)
```

This function checks the public key in ed448_key structure matches the private key.

Parameters:

- **key** Pointer to an ed448_key structure holding a private and public key.

See:

- wc_ed448_import_private_key
- wc_ed448_import_private_key_ex

Return:

- 0 Returned if the private and public key matched.
- BAD_FUNC_ARGS Returned if the given key is NULL.

Example

```
int ret;
byte priv[] = { initialize with 57 byte private key };
byte pub[]  = { initialize with the corresponding public key };

ed448_key key;
```

```
wc_ed448_init_key(&key);
wc_ed448_import_private_key_ex(priv, sizeof(priv), pub, sizeof(pub), &key,
    1);
ret = wc_ed448_check_key(&key);
if (ret != 0) {
    // error checking key
}
```

```
int wc_ed448_size(
    ed448_key * key
)
```

This function returns the size of an Ed448 private key - 57 bytes.

Parameters:

- **key** Pointer to an ed448_key structure for which to get the key size.

See: wc_ed448_make_key

Return:

- ED448_KEY_SIZE The size of a valid private key (57 bytes).
- BAD_FUNC_ARGS Returned if the given key is NULL.

Example

```
int keySz;
ed448_key key;
// initialize key, make key
keySz = wc_ed448_size(&key);
if (keySz == 0) {
    // error determining key size
}
```

```
int wc_ed448_priv_size(
    ed448_key * key
)
```

This function returns the private key size (secret + public) in bytes.

Parameters:

- **key** Pointer to an ed448_key structure for which to get the key size.

See: wc_ed448_pub_size

Return:

- ED448_PRV_KEY_SIZE The size of the private key (114 bytes).
- BAD_FUNC_ARG Returns if key argument is NULL.

Example

```
ed448_key key;
wc_ed448_init(&key);

WC_RNG rng;
wc_InitRng(&rng);

wc_ed448_make_key(&rng, 57, &key); // initialize 57 byte Ed448 key
int key_size = wc_ed448_priv_size(&key);
```

```
int wc_ed448_pub_size(
    ed448_key * key
)
```

This function returns the compressed key size in bytes (public key).

Parameters:

- **key** Pointer to an ed448_key structure for which to get the key size.

See: wc_ed448_priv_size

Return:

- ED448_PUB_KEY_SIZE The size of the compressed public key (57 bytes).
- BAD_FUNC_ARG Returns if key argument is NULL.

Example

```
ed448_key key;
wc_ed448_init(&key);
WC_RNG rng;
wc_InitRng(&rng);

wc_ed448_make_key(&rng, 57, &key); // initialize 57 byte Ed448 key
int key_size = wc_ed448_pub_size(&key);
```

```
int wc_ed448_sig_size(
    ed448_key * key
)
```

This function returns the size of an Ed448 signature (114 in bytes).

Parameters:

- **key** Pointer to an ed448_key structure for which to get the signature size.

See: wc_ed448_sign_msg

Return:

- ED448_SIG_SIZE The size of an Ed448 signature (114 bytes).
- BAD_FUNC_ARG Returns if key argument is NULL.

Example

```
int sigSz;
ed448_key key;
// initialize key, make key

sigSz = wc_ed448_sig_size(&key);
if (sigSz == 0) {
    // error determining sig size
}
```

B.27 Platform Security Architecture (PSA) API

B.26.2.24 function wc_ed448_sig_size

B.27.1 Functions

	Name
int	**wolfSSL_CTX_psa_enable**(WOLFSSL_CTX * ctx)This function enables PSA support on the given context.
int	**wolfSSL_set_psa_ctx**(WOLFSSL * ssl, struct psa_ssl_ctx * ctx)This function setup the PSA context for the given SSL session.
void	**wolfSSL_free_psa_ctx**(struct psa_ssl_ctx * ctx)This function releases the resources used by a PSA context.
int	**wolfSSL_psa_set_private_key_id**(struct psa_ssl_ctx * ctx, psa_key_id_t id)This function set the private key used by an SSL session.

B.27.2 Functions Documentation

```
int wolfSSL_CTX_psa_enable(
    WOLFSSL_CTX * ctx
)
```

This function enables PSA support on the given context.

Parameters:

- **ctx** pointer to the WOLFSSL_CTX object on which the PSA support must be enabled

See: wolfSSL_set_psa_ctx

Return:

- WOLFSSL_SUCCESS on success
- BAD_FUNC_ARG if ctx == NULL

Example

```
WOLFSSL_CTX *ctx;
ctx = wolfSSL_CTX_new(wolfTLSv1_2_client_method());
if (!ctx)
    return NULL;
ret = wolfSSL_CTX_psa_enable(ctx);
if (ret != WOLFSSL_SUCCESS)
    printf("can't enable PSA on ctx");
```

```
int wolfSSL_set_psa_ctx(
    WOLFSSL * ssl,
    struct psa_ssl_ctx * ctx
)
```

This function setup the PSA context for the given SSL session.

Parameters:

- **ssl** pointer to the WOLFSSL where the ctx will be enabled
- **ctx** pointer to a struct psa_ssl_ctx (must be unique for a ssl session)

See:

- wolfSSL_psa_set_private_key_id
- wolfSSL_psa_free_psa_ctx

Return:

- WOLFSSL_SUCCESS on success
- BAD_FUNC_ARG if ssl or ctx are NULL

This function setup the PSA context for the TLS callbacks to the given SSL session. At the end of the session, the resources used by the context should be freed using wolfSSL_free_psa_ctx().

Example

```
// Create new ssl session
WOLFSSL *ssl;
struct psa_ssl_ctx psa_ctx = { 0 };
ssl = wolfSSL_new(ctx);
```

```
if (!ssl)
    return NULL;
// setup PSA context
ret = wolfSSL_set_psa_ctx(ssl, ctx);
```

```
void wolfSSL_free_psa_ctx(
    struct psa_ssl_ctx * ctx
)
```

This function releases the resources used by a PSA context.

Parameters:

- **ctx** pointer to a struct psa_ssl_ctx

See: wolfSSL_set_psa_ctx

```
int wolfSSL_psa_set_private_key_id(
    struct psa_ssl_ctx * ctx,
    psa_key_id_t id
)
```

This function set the private key used by an SSL session.

Parameters:

- **ctx** pointer to a struct psa_ssl_ctx
- **id** PSA id of the key to be used as private key

See: wolfSSL_set_psa_ctx

Example

```
// Create new ssl session
WOLFSSL *ssl;
struct psa_ssl_ctx psa_ctx = { 0 };
psa_key_id_t key_id;

// key provisioning already done
get_private_key_id(&key_id);

ssl = wolfSSL_new(ctx);
if (!ssl)
    return NULL;

wolfSSL_psa_set_private_key_id(&psa_ctx, key_id);
wolfSSL_set_psa_ctx(ssl, ctx);
```

B.28 Algorithm - SipHash

B.27.2.4 function wolfSSL_psa_set_private_key_id

B.28.1 Functions

	Name
int	**wc_InitSipHash**(SipHash * siphash, const unsigned char * key, unsigned char outSz)This function initializes SipHash with a key for a MAC size.
int	**wc_SipHashUpdate**(SipHash * siphash, const unsigned char * in, word32 inSz)Can be called to continually hash the provided byte array of length len.
int	**wc_SipHashFinal**(SipHash * siphash, unsigned char * out, unsigned char outSz)Finalizes MACing of data. Result is placed into out.
int	**wc_SipHash**(const unsigned char * key, const unsigned char * in, word32 inSz, unsigned char * out, unsigned char outSz)This function one-shots the data using SipHash to calculate a MAC based on the key.

B.28.2 Functions Documentation

```
int wc_InitSipHash(
    SipHash * siphash,
    const unsigned char * key,
    unsigned char outSz
)
```

This function initializes SipHash with a key for a MAC size.

Parameters:

- **siphash** pointer to the SipHash structure to use for MACing
- **key** pointer to the 16-byte array
- **outSz** number of bytes to output as MAC

See:

- wc_SipHash
- wc_SipHashUpdate
- wc_SipHashFinal

Return:

- 0 Returned upon successfully initializing
- BAD_FUNC_ARG Returned when siphash or key is NULL
- BAD_FUNC_ARG Returned when outSz is neither 8 nor 16

Example

```
SipHash siphash[1];
unsigned char key[16] = { ... };
byte macSz = 8; // 8 or 16

if ((ret = wc_InitSipHash(siphash, key, macSz)) != 0) {
    WOLFSSL_MSG("wc_InitSipHash failed");
}
else if ((ret = wc_SipHashUpdate(siphash, data, len)) != 0) {
    WOLFSSL_MSG("wc_SipHashUpdate failed");
}
else if ((ret = wc_SipHashFinal(siphash, mac, macSz)) != 0) {
    WOLFSSL_MSG("wc_SipHashFinal failed");
}
```

```
int wc_SipHashUpdate(
    SipHash * siphash,
    const unsigned char * in,
    word32 inSz
)
```

Can be called to continually hash the provided byte array of length len.

Parameters:

- **siphash** pointer to the SipHash structure to use for MACing
- **in** the data to be MACed
- **inSz** size of data to be MACed

See:

- wc_SipHash
- wc_InitSipHash
- wc_SipHashFinal

Return:

- 0 Returned upon successfully adding the data to the MAC
- BAD_FUNC_ARG Returned when siphash is NULL
- BAD_FUNC_ARG Returned when in is NULL and inSz is not zero

Example

```
SipHash siphash[1];
byte data[] = { Data to be MACed };
word32 len = sizeof(data);

if ((ret = wc_InitSipHash(siphash, key, macSz)) != 0) {
```

```
    WOLFSSL_MSG("wc_InitSipHash failed");
}
else if ((ret = wc_SipHashUpdate(siphash, data, len)) != 0) {
    WOLFSSL_MSG("wc_SipHashUpdate failed");
}
else if ((ret = wc_SipHashFinal(siphash, mac, macSz)) != 0) {
    WOLFSSL_MSG("wc_SipHashFinal failed");
}
```

```
int wc_SipHashFinal(
    SipHash * siphash,
    unsigned char * out,
    unsigned char outSz
)
```

Finalizes MACing of data. Result is placed into out.

Parameters:

- **siphash** pointer to the SipHash structure to use for MACing
- **out** Byte array to hold MAC value
- **outSz** number of bytes to output as MAC

See:

- wc_SipHash
- wc_InitSipHash
- wc_SipHashUpdate

Return:

- 0 Returned upon successfully finalizing.
- BAD_FUNC_ARG Returned when siphash of out is NULL
- BAD_FUNC_ARG Returned when outSz is not the same as the initialized value

Example

```
SipHash siphash[1];
byte mac[8] = { ... }; // 8 or 16 bytes
byte macSz = sizeof(mac);

if ((ret = wc_InitSipHash(siphash, key, macSz)) != 0) {
    WOLFSSL_MSG("wc_InitSipHash failed");
}
else if ((ret = wc_SipHashUpdate(siphash, data, len)) != 0) {
    WOLFSSL_MSG("wc_SipHashUpdate failed");
}
else if ((ret = wc_SipHashFinal(siphash, mac, macSz)) != 0) {
    WOLFSSL_MSG("wc_SipHashFinal failed");
}
```

```
int wc_SipHash(
    const unsigned char * key,
    const unsigned char * in,
    word32 inSz,
    unsigned char * out,
    unsigned char outSz
)
```

This function one-shots the data using SipHash to calculate a MAC based on the key.

Parameters:

- **key** pointer to the 16-byte array
- **in** the data to be MACed
- **inSz** size of data to be MACed
- **out** Byte array to hold MAC value
- **outSz** number of bytes to output as MAC

See:

- wc_InitSipHash
- wc_SipHashUpdate
- wc_SipHashFinal

Return:

- 0 Returned upon successfully MACing
- BAD_FUNC_ARG Returned when key or out is NULL
- BAD_FUNC_ARG Returned when in is NULL and inSz is not zero
- BAD_FUNC_ARG Returned when outSz is neither 8 nor 16

Example

```
unsigned char key[16] = { ... };
byte data[] = { Data to be MACed };
word32 len = sizeof(data);
byte mac[8] = { ... }; // 8 or 16 bytes
byte macSz = sizeof(mac);

if ((ret = wc_SipHash(key, data, len, mac, macSz)) != 0) {
    WOLFSSL_MSG("wc_SipHash failed");
}
```

C API Header Files

C.1 dox_comments/header_files/aes.h

C.1.1 Functions

	Name
int	**wc_AesSetKey**(Aes * aes, const byte * key, word32 len, const byte * iv, int dir)This function initializes an AES structure by setting the key and then setting the initialization vector.
int	**wc_AesSetIV**(Aes * aes, const byte * iv)This function sets the initialization vector for a particular AES object. The AES object should be initialized before calling this function.
int	**wc_AesCbcEncrypt**(Aes * aes, byte * out, const byte * in, word32 sz)Encrypts a plaintext message from the input buffer in, and places the resulting cipher text in the output buffer out using cipher block chaining with AES. This function requires that the AES object has been initialized by calling AesSetKey before a message is able to be encrypted. This function assumes that the input message is AES block length aligned, and expects the input length to be a multiple of the block length, which will optionally be checked and enforced if WOLFSSL_AES_CBC_LENGTH_CHECKS is defined in the build configuration. In order to assure block-multiple input, PKCS#7 style padding should be added beforehand. This differs from the OpenSSL AES-CBC methods which add the padding for you. To make the wolfSSL and corresponding OpenSSL functions interoperate, one should specify the -nopad option in the OpenSSL command line function so that it behaves like the wolfSSL AesCbcEncrypt method and does not add extra padding during encryption.

	Name
int	**wc_AesCbcDecrypt**(Aes * aes, byte * out, const byte * in, word32 sz)Decrypts a cipher from the input buffer in, and places the resulting plain text in the output buffer out using cipher block chaining with AES. This function requires that the AES structure has been initialized by calling AesSetKey before a message is able to be decrypted. This function assumes that the original message was AES block length aligned, and expects the input length to be a multiple of the block length, which will optionally be checked and enforced if WOLFSSL_AES_CBC_LENGTH_CHECKS is defined in the build configuration. This differs from the OpenSSL AES-CBC methods, which add PKCS#7 padding automatically, and so do not require block-multiple input. To make the wolfSSL function and equivalent OpenSSL functions interoperate, one should specify the -nopad option in the OpenSSL command line function so that it behaves like the wolfSSL AesCbcEncrypt method and does not create errors during decryption.
int	**wc_AesCtrEncrypt**(Aes * aes, byte * out, const byte * in, word32 sz)Encrypts/Decrypts a message from the input buffer in, and places the resulting cipher text in the output buffer out using CTR mode with AES. This function is only enabled if WOLFSSL_AES_COUNTER is enabled at compile time. The AES structure should be initialized through AesSetKey before calling this function. Note that this function is used for both decryption and encryption. *NOTE:* Regarding using same API for encryption and decryption. User should differentiate between Aes structures for encrypt/decrypt.
int	**wc_AesEncryptDirect**(Aes * aes, byte * out, const byte * in)This function is a one-block encrypt of the input block, in, into the output block, out. It uses the key of the provided AES structure, which should be initialized with wc_AesSetKey before calling this function. wc_AesSetKey should have been called with the iv set to NULL. This is only enabled if the configure option WOLFSSL_AES_DIRECT is enabled. **Warning:** In nearly all use cases ECB mode is considered to be less secure. Please avoid using ECB API's directly whenever possible.

	Name
int	**wc_AesDecryptDirect**(Aes * aes, byte * out, const byte * in)This function is a one-block decrypt of the input block, in, into the output block, out. It uses the key of the provided AES structure, which should be initialized with wc_AesSetKey before calling this function. wc_AesSetKey should have been called with the iv set to NULL. This is only enabled if the configure option WOLFSSL_AES_DIRECT is enabled. **Warning:** In nearly all use cases ECB mode is considered to be less secure. Please avoid using ECB API's directly whenever possible.
int	**wc_AesSetKeyDirect**(Aes * aes, const byte * key, word32 len, const byte * iv, int dir)This function is used to set the AES keys for CTR mode with AES. It initializes an AES object with the given key, iv (initialization vector), and encryption dir (direction). It is only enabled if the configure option WOLFSSL_AES_DIRECT is enabled. Currently wc_AesSetKeyDirect uses wc_AesSetKey internally. **Warning:** In nearly all use cases ECB mode is considered to be less secure. Please avoid using ECB API's directly whenever possible.
int	**wc_AesGcmSetKey**(Aes * aes, const byte * key, word32 len)This function is used to set the key for AES GCM (Galois/Counter Mode). It initializes an AES object with the given key. It is only enabled if the configure option HAVE_AESGCM is enabled at compile time.
int	**wc_AesGcmEncrypt**(Aes * aes, byte * out, const byte * in, word32 sz, const byte * iv, word32 ivSz, byte * authTag, word32 authTagSz, const byte * authIn, word32 authInSz)This function encrypts the input message, held in the buffer in, and stores the resulting cipher text in the output buffer out. It requires a new iv (initialization vector) for each call to encrypt. It also encodes the input authentication vector, authIn, into the authentication tag, authTag.

	Name
int	**wc_AesGcmDecrypt**(Aes * aes, byte * out, const byte * in, word32 sz, const byte * iv, word32 ivSz, const byte * authTag, word32 authTagSz, const byte * authIn, word32 authInSz)This function decrypts the input cipher text, held in the buffer in, and stores the resulting message text in the output buffer out. It also checks the input authentication vector, authIn, against the supplied authentication tag, authTag. If a nonzero error code is returned, the output data is undefined. However, callers must unconditionally zeroize the output buffer to guard against leakage of cleartext data.
int	**wc_GmacSetKey**(Gmac * gmac, const byte * key, word32 len)This function initializes and sets the key for a GMAC object to be used for Galois Message Authentication.
int	**wc_GmacUpdate**(Gmac * gmac, const byte * iv, word32 ivSz, const byte * authIn, word32 authInSz, byte * authTag, word32 authTagSz)This function generates the Gmac hash of the authIn input and stores the result in the authTag buffer. After running wc_GmacUpdate, one should compare the generated authTag to a known authentication tag to verify the authenticity of a message.
int	**wc_AesCcmSetKey**(Aes * aes, const byte * key, word32 keySz)This function sets the key for an AES object using CCM (Counter with CBC_MAC). It takes a pointer to an AES structure and initializes it with supplied key.
int	**wc_AesCcmEncrypt**(Aes * aes, byte * out, const byte * in, word32 inSz, const byte * nonce, word32 nonceSz, byte * authTag, word32 authTagSz, const byte * authIn, word32 authInSz)This function encrypts the input message, in, into the output buffer, out, using CCM (Counter with CBC_MAC). It subsequently calculates and stores the authorization tag, authTag, from the authIn input.

	Name
int	**wc_AesCcmDecrypt**(Aes * aes, byte * out, const byte * in, word32 inSz, const byte * nonce, word32 nonceSz, const byte * authTag, word32 authTagSz, const byte * authIn, word32 authInSz)This function decrypts the input cipher text, in, into the output buffer, out, using CCM (Counter with CBC_MAC). It subsequently calculates the authorization tag, authTag, from the authIn input. If a nonzero error code is returned, the output data is undefined. However, callers must unconditionally zeroize the output buffer to guard against leakage of cleartext data.
int	**wc_AesXtsInit**(XtsAes * aes, void * heap, int devId)This is to initialize an AES-XTS context. It is up to user to call wc_AesXtsFree on aes key when done.
int	**wc_AesXtsSetKeyNoInit. It is up to user to call wc_AesXtsFree on aes key when done.
int	**wc_AesXtsSetKey**(XtsAes * aes, const byte * key, word32 len, int dir, void * heap, int devId)This is to help with setting keys to correct encrypt or decrypt type. It is up to user to call wc_AesXtsFree on aes key when done.
int	**wc_AesXtsEncryptSector**(XtsAes * aes, byte * out, const byte * in, word32 sz, word64 sector)Same process as wc_AesXtsEncrypt but uses a word64 type as the tweak value instead of a byte array. This just converts the word64 to a byte array and calls wc_AesXtsEncrypt.
int	**wc_AesXtsDecryptSector**(XtsAes * aes, byte * out, const byte * in, word32 sz, word64 sector)Same process as wc_AesXtsDecrypt but uses a word64 type as the tweak value instead of a byte array. This just converts the word64 to a byte array.
int	**wc_AesXtsEncrypt**(XtsAes * aes, byte * out, const byte * in, word32 sz, const byte * i, word32 iSz)AES with XTS mode. (XTS) XEX encryption with Tweak and cipher text Stealing.
int	**wc_AesXtsDecrypt**(XtsAes * aes, byte * out, const byte * in, word32 sz, const byte * i, word32 iSz)Same process as encryption but Aes key is AES_DECRYPTION type.
int	**wc_AesXtsFree**(XtsAes * aes)This is to free up any resources used by the XtsAes structure.
int	**wc_AesInit**(Aes * aes, void * heap, int devId)Initialize Aes structure. Sets heap hint to be used and ID for use with async hardware. It is up to the user to call wc_AesFree on the Aes structure when done.

	Name
int	**wc_AesFree**(Aes * aes)free resources associated with the Aes structure when applicable. Internally may sometimes be a no_op but still recommended to call in all cases as a general best_practice (IE if application code is ported for use on new environments where the call is applicable).
int	**wc_AesCfbEncrypt**(Aes * aes, byte * out, const byte * in, word32 sz)AES with CFB mode.
int	**wc_AesCfbDecrypt**(Aes * aes, byte * out, const byte * in, word32 sz)AES with CFB mode.
int	**wc_AesSivEncrypt**(const byte * key, word32 keySz, const byte * assoc, word32 assocSz, const byte * nonce, word32 nonceSz, const byte * in, word32 inSz, byte * siv, byte * out)This function performs SIV (synthetic initialization vector) encryption as described in RFC 5297.
int	**wc_AesSivDecrypt**(const byte * key, word32 keySz, const byte * assoc, word32 assocSz, const byte * nonce, word32 nonceSz, const byte * in, word32 inSz, byte * siv, byte * out)This function performs SIV (synthetic initialization vector) decryption as described in RFC 5297. If a nonzero error code is returned, the output data is undefined. However, callers must unconditionally zeroize the output buffer to guard against leakage of cleartext data.
WOLFSSL_API int	**wc_AesEaxEncryptAuth**(const byte * key, word32 keySz, byte * out, const byte * in, word32 inSz, const byte * nonce, word32 nonceSz, byte * authTag, word32 authTagSz, const byte * authIn, word32 authInSz)This function performs AES EAX encryption and authentication as described in “EAX: A Conventional Authenticated_Encryption Mode” (https://eprint.iacr.org/2003/069). It is a “one-shot” API that performs all encryption and authentication operations in one function call.

	Name
WOLFSSL_API int	**wc_AesEaxDecryptAuth**(const byte * key, word32 keySz, byte * out, const byte * in, word32 inSz, const byte * nonce, word32 nonceSz, const byte * authTag, word32 authTagSz, const byte * authIn, word32 authInSz)This function performs AES EAX decryption and authentication as described in "EAX: A Conventional Authenticated_Encryption Mode" (https://eprint.iacr.org/2003/069). It is a "one-shot" API that performs all decryption and authentication operations in one function call. If a nonzero error code is returned, the output data is undefined. However, callers must unconditionally zeroize the output buffer to guard against leakage of cleartext data.
WOLFSSL_API int	**wc_AesEaxInit when done using the instance.
WOLFSSL_API int	**wc_AesEaxEncryptUpdate.
WOLFSSL_API int	**wc_AesEaxDecryptUpdate.
WOLFSSL_API int	**wc_AesEaxAuthDataUpdate.
WOLFSSL_API int	**wc_AesEaxEncryptFinal.
WOLFSSL_API int	**wc_AesEaxDecryptFinal.
WOLFSSL_API int	**wc_AesEaxFree**(AesEax * eax)This frees up any resources, specifically keys, used by the Aes instance inside the AesEax wrapper struct. It should be called on the AesEax struct after it has been initialized with wc_AesEaxInit, and all desired EAX operations are complete.
int	**wc_AesCtsEncrypt**(const byte * key, word32 keySz, byte * out, const byte * in, word32 inSz, const byte * iv)This function performs AES encryption using Ciphertext Stealing (CTS) mode. It is a one-shot API that handles all operations in a single call.
int	**wc_AesCtsDecrypt**(const byte * key, word32 keySz, byte * out, const byte * in, word32 inSz, const byte * iv)This function performs AES decryption using Ciphertext Stealing (CTS) mode. It is a one-shot API that handles all operations in a single call.
int	**wc_AesCtsEncryptUpdate**(Aes * aes, byte * out, word32 * outSz, const byte * in, word32 inSz)This function performs an update step of the AES CTS encryption. It processes a chunk of plaintext and stores intermediate data.
int	**wc_AesCtsEncryptFinal**(Aes * aes, byte * out, word32 * outSz)This function finalizes the AES CTS encryption operation. It processes any remaining plaintext and completes the encryption.

	Name
int	**wc_AesCtsDecryptUpdate**(Aes * aes, byte * out, word32 * outSz, const byte * in, word32 inSz)This function performs an update step of the AES CTS decryption. It processes a chunk of ciphertext and stores intermediate data.
int	**wc_AesCtsDecryptFinal**(Aes * aes, byte * out, word32 * outSz)This function finalizes the AES CTS decryption operation. It processes any remaining ciphertext and completes the decryption.

C.1.2 Functions Documentation

C.1.2.1 function wc_AesSetKey

```
int wc_AesSetKey(
    Aes * aes,
    const byte * key,
    word32 len,
    const byte * iv,
    int dir
)
```

This function initializes an AES structure by setting the key and then setting the initialization vector.

Parameters:

- **aes** pointer to the AES structure to modify
- **key** 16, 24, or 32 byte secret key for encryption and decryption
- **len** length of the key passed in
- **iv** pointer to the initialization vector used to initialize the key
- **dir** Cipher direction. Set AES_ENCRYPTION to encrypt, or AES_DECRYPTION to decrypt. Direction for some modes (CFB and CTR) is always AES_ENCRYPTION.

See:

- wc_AesSetKeyDirect
- wc_AesSetIV

Return:

- 0 On successfully setting key and initialization vector.
- BAD_FUNC_ARG Returned if key length is invalid.

Example

```
Aes enc;
int ret = 0;
byte key[] = { some 16, 24 or 32 byte key };
byte iv[]  = { some 16 byte iv };
if (ret = wc_AesInit(&enc, HEAP_HINT, INVALID_DEVID) != 0) {
    // failed to initialize aes key
}
if (ret = wc_AesSetKey(&enc, key, AES_BLOCK_SIZE, iv,
AES_ENCRYPTION) != 0) {
// failed to set aes key
}
```

C.1.2.2 function wc_AesSetIV

```
int wc_AesSetIV(
    Aes * aes,
    const byte * iv
)
```

This function sets the initialization vector for a particular AES object. The AES object should be initialized before calling this function.

Parameters:

- **aes** pointer to the AES structure on which to set the initialization vector
- **iv** initialization vector used to initialize the AES structure. If the value is NULL, the default action initializes the iv to 0.

See:

- wc_AesSetKeyDirect
- wc_AesSetKey

Return:

- 0 On successfully setting initialization vector.
- BAD_FUNC_ARG Returned if AES pointer is NULL.

Example

```
Aes enc;
// set enc key
byte iv[]  = { some 16 byte iv };
if (ret = wc_AesSetIV(&enc, iv) != 0) {
// failed to set aes iv
}
```

C.1.2.3 function wc_AesCbcEncrypt

```
int wc_AesCbcEncrypt(
    Aes * aes,
    byte * out,
    const byte * in,
    word32 sz
)
```

Encrypts a plaintext message from the input buffer in, and places the resulting cipher text in the output buffer out using cipher block chaining with AES. This function requires that the AES object has been initialized by calling AesSetKey before a message is able to be encrypted. This function assumes that the input message is AES block length aligned, and expects the input length to be a multiple of the block length, which will optionally be checked and enforced if WOLFSSL_AES_CBC_LENGTH_CHECKS is defined in the build configuration. In order to assure block-multiple input, PKCS#7 style padding should be added beforehand. This differs from the OpenSSL AES-CBC methods which add the padding for you. To make the wolfSSL and corresponding OpenSSL functions interoperate, one should specify the -nopad option in the OpenSSL command line function so that it behaves like the wolfSSL AesCbcEncrypt method and does not add extra padding during encryption.

Parameters:

- **aes** pointer to the AES object used to encrypt data
- **out** pointer to the output buffer in which to store the ciphertext of the encrypted message
- **in** pointer to the input buffer containing message to be encrypted
- **sz** size of input message

See:

- wc_AesInit
- wc_AesSetKey
- wc_AesSetIV
- wc_AesCbcDecrypt

Return:

- 0 On successfully encrypting message.
- BAD_ALIGN_E: may be returned on block align error
- BAD_LENGTH_E will be returned if the input length isn't a multiple of the AES block length, when the library is built with WOLFSSL_AES_CBC_LENGTH_CHECKS.

Example

```
Aes enc;
int ret = 0;
// initialize enc with wc_AesInit and wc_AesSetKey, using direction
// AES_ENCRYPTION
byte msg[AES_BLOCK_SIZE * n]; // multiple of 16 bytes
// fill msg with data
byte cipher[AES_BLOCK_SIZE * n]; // Some multiple of 16 bytes
if ((ret = wc_AesCbcEncrypt(&enc, cipher, message, sizeof(msg))) != 0 ) {
// block align error
}
```

C.1.2.4 function wc_AesCbcDecrypt

```
int wc_AesCbcDecrypt(
    Aes * aes,
    byte * out,
    const byte * in,
    word32 sz
)
```

Decrypts a cipher from the input buffer in, and places the resulting plain text in the output buffer out using cipher block chaining with AES. This function requires that the AES structure has been initialized by calling AesSetKey before a message is able to be decrypted. This function assumes that the original message was AES block length aligned, and expects the input length to be a multiple of the block length, which will optionally be checked and enforced if WOLFSSL_AES_CBC_LENGTH_CHECKS is defined in the build configuration. This differs from the OpenSSL AES-CBC methods, which add PKCS#7 padding automatically, and so do not require block-multiple input. To make the wolfSSL function and equivalent OpenSSL functions interoperate, one should specify the -nopad option in the OpenSSL command line function so that it behaves like the wolfSSL AesCbcEncrypt method and does not create errors during decryption.

Parameters:

- **aes** pointer to the AES object used to decrypt data.
- **out** pointer to the output buffer in which to store the plain text of the decrypted message. size must be a multiple of AES_BLOCK_LENGTH, padded if necessary
- **in** pointer to the input buffer containing cipher text to be decrypted. size must be a multiple of AES_BLOCK_LENGTH, padded if necessary
- **sz** size of input message.

See:

- wc_AesInit

- wc_AesSetKey
- wc_AesCbcEncrypt

Return:

- 0 On successfully decrypting message.
- BAD_ALIGN_E may be returned on block align error.
- BAD_LENGTH_E will be returned if the input length isn't a multiple of the AES block length, when the library is built with WOLFSSL_AES_CBC_LENGTH_CHECKS.

Example

```
Aes dec;
int ret = 0;
// initialize dec with wc_AesInit and wc_AesSetKey, using direction
// AES_DECRYPTION
byte cipher[AES_BLOCK_SIZE * n]; // some multiple of 16 bytes
// fill cipher with cipher text
byte plain [AES_BLOCK_SIZE * n];
if ((ret = wc_AesCbcDecrypt(&dec, plain, cipher, sizeof(cipher))) != 0 ) {
// block align error
}
```

C.1.2.5 function wc_AesCtrEncrypt

```
int wc_AesCtrEncrypt(
    Aes * aes,
    byte * out,
    const byte * in,
    word32 sz
)
```

Encrypts/Decrypts a message from the input buffer in, and places the resulting cipher text in the output buffer out using CTR mode with AES. This function is only enabled if WOLFSSL_AES_COUNTER is enabled at compile time. The AES structure should be initialized through AesSetKey before calling this function. Note that this function is used for both decryption and encryption. *NOTE:* Regarding using same API for encryption and decryption. User should differentiate between Aes structures for encrypt/decrypt.

Parameters:

- **aes** pointer to the AES object used to decrypt data
- **out** pointer to the output buffer in which to store the cipher text of the encrypted message size must be a multiple of AES_BLOCK_LENGTH, padded if necessary
- **in** pointer to the input buffer containing plain text to be encrypted size must be a multiple of AES_BLOCK_LENGTH, padded if necessary
- **sz** size of the input plain text

See: wc_AesSetKey

Return: int integer values corresponding to wolfSSL error or success status

Example

```
Aes enc;
Aes dec;
// initialize enc and dec with wc_AesInit and wc_AesSetKeyDirect, using
// direction AES_ENCRYPTION since the underlying API only calls Encrypt
// and by default calling encrypt on a cipher results in a decryption of
// the cipher
```

```
byte msg[AES_BLOCK_SIZE * n]; //n being a positive integer making msg
some multiple of 16 bytes
// fill plain with message text
byte cipher[AES_BLOCK_SIZE * n];
byte decrypted[AES_BLOCK_SIZE * n];
wc_AesCtrEncrypt(&enc, cipher, msg, sizeof(msg)); // encrypt plain
wc_AesCtrEncrypt(&dec, decrypted, cipher, sizeof(cipher));
// decrypt cipher text
```

C.1.2.6 function wc_AesEncryptDirect

```
int wc_AesEncryptDirect(
    Aes * aes,
    byte * out,
    const byte * in
)
```

This function is a one-block encrypt of the input block, in, into the output block, out. It uses the key of the provided AES structure, which should be initialized with wc_AesSetKey before calling this function. wc_AesSetKey should have been called with the iv set to NULL. This is only enabled if the configure option WOLFSSL_AES_DIRECT is enabled. **Warning:** In nearly all use cases ECB mode is considered to be less secure. Please avoid using ECB API's directly whenever possible.

Parameters:

- **aes** pointer to the AES object used to encrypt data
- **out** pointer to the output buffer in which to store the cipher text of the encrypted message
- **in** pointer to the input buffer containing plain text to be encrypted

See:

- wc_AesDecryptDirect
- wc_AesSetKeyDirect

Return: int integer values corresponding to wolfSSL error or success status

Example

```
Aes enc;
// initialize enc with wc_AesInit and wc_AesSetKey, using direction
// AES_ENCRYPTION
byte msg [AES_BLOCK_SIZE]; // 16 bytes
// initialize msg with plain text to encrypt
byte cipher[AES_BLOCK_SIZE];
wc_AesEncryptDirect(&enc, cipher, msg);
```

C.1.2.7 function wc_AesDecryptDirect

```
int wc_AesDecryptDirect(
    Aes * aes,
    byte * out,
    const byte * in
)
```

This function is a one-block decrypt of the input block, in, into the output block, out. It uses the key of the provided AES structure, which should be initialized with wc_AesSetKey before calling this function. wc_AesSetKey should have been called with the iv set to NULL. This is only enabled if the configure

option WOLFSSL_AES_DIRECT is enabled. **Warning:** In nearly all use cases ECB mode is considered to be less secure. Please avoid using ECB API's directly whenever possible.

Parameters:

- **aes** pointer to the AES object used to encrypt data
- **out** pointer to the output buffer in which to store the plain text of the decrypted cipher text
- **in** pointer to the input buffer containing cipher text to be decrypted

See:

- wc_AesEncryptDirect
- wc_AesSetKeyDirect

Return: int integer values corresponding to wolfSSL error or success status

Example

```
Aes dec;
// initialize enc with wc_AesInit and wc_AesSetKey, using direction
// AES_DECRYPTION
byte cipher [AES_BLOCK_SIZE]; // 16 bytes
// initialize cipher with cipher text to decrypt
byte msg[AES_BLOCK_SIZE];
wc_AesDecryptDirect(&dec, msg, cipher);
```

C.1.2.8 function wc_AesSetKeyDirect

```
int wc_AesSetKeyDirect(
    Aes * aes,
    const byte * key,
    word32 len,
    const byte * iv,
    int dir
)
```

This function is used to set the AES keys for CTR mode with AES. It initializes an AES object with the given key, iv (initialization vector), and encryption dir (direction). It is only enabled if the configure option WOLFSSL_AES_DIRECT is enabled. Currently wc_AesSetKeyDirect uses wc_AesSetKey internally. **Warning:** In nearly all use cases ECB mode is considered to be less secure. Please avoid using ECB API's directly whenever possible.

Parameters:

- **aes** pointer to the AES object used to encrypt data
- **key** 16, 24, or 32 byte secret key for encryption and decryption
- **len** length of the key passed in
- **iv** initialization vector used to initialize the key
- **dir** Cipher direction. Set AES_ENCRYPTION to encrypt, or AES_DECRYPTION to decrypt. (See enum in wolfssl/wolfcrypt/aes.h) (NOTE: If using wc_AesSetKeyDirect with Aes Counter mode (Stream cipher) only use AES_ENCRYPTION for both encrypting and decrypting)

See:

- wc_AesEncryptDirect
- wc_AesDecryptDirect
- wc_AesSetKey

Return:

- 0 On successfully setting the key.

- BAD_FUNC_ARG Returned if the given key is an invalid length.

Example

```
Aes enc;
int ret = 0;
byte key[] = { some 16, 24, or 32 byte key };
byte iv[]  = { some 16 byte iv };

if (ret = wc_AesInit(&enc, HEAP_HINT, INVALID_DEVID) != 0) {
    // failed to initialize aes key
}
if (ret = wc_AesSetKeyDirect(&enc, key, sizeof(key), iv,
AES_ENCRYPTION) != 0) {
// failed to set aes key
}
```

C.1.2.9 function wc_AesGcmSetKey

```
int wc_AesGcmSetKey(
    Aes * aes,
    const byte * key,
    word32 len
)
```

This function is used to set the key for AES GCM (Galois/Counter Mode). It initializes an AES object with the given key. It is only enabled if the configure option HAVE_AESGCM is enabled at compile time.

Parameters:

- **aes** pointer to the AES object used to encrypt data
- **key** 16, 24, or 32 byte secret key for encryption and decryption
- **len** length of the key passed in

See:

- wc_AesGcmEncrypt
- wc_AesGcmDecrypt

Return:

- 0 On successfully setting the key.
- BAD_FUNC_ARG Returned if the given key is an invalid length.

Example

```
Aes enc;
int ret = 0;
byte key[] = { some 16, 24,32 byte key };
if (ret = wc_AesInit(&enc, HEAP_HINT, INVALID_DEVID) != 0) {
    // failed to initialize aes key
}
if (ret = wc_AesGcmSetKey(&enc, key, sizeof(key)) != 0) {
// failed to set aes key
}
```

C.1.2.10 function wc_AesGcmEncrypt

```
int wc_AesGcmEncrypt(
    Aes * aes,
```

```
    byte * out,
    const byte * in,
    word32 sz,
    const byte * iv,
    word32 ivSz,
    byte * authTag,
    word32 authTagSz,
    const byte * authIn,
    word32 authInSz
)
```

This function encrypts the input message, held in the buffer in, and stores the resulting cipher text in the output buffer out. It requires a new iv (initialization vector) for each call to encrypt. It also encodes the input authentication vector, authIn, into the authentication tag, authTag.

Parameters:

- **aes** - pointer to the AES object used to encrypt data
- **out** pointer to the output buffer in which to store the cipher text size must match in's size (sz)
- **in** pointer to the input buffer holding the message to encrypt size must be a multiple of AES_BLOCK_LENGTH, padded if necessary
- **sz** length of the input message to encrypt
- **iv** pointer to the buffer containing the initialization vector
- **ivSz** length of the initialization vector
- **authTag** pointer to the buffer in which to store the authentication tag
- **authTagSz** length of the desired authentication tag
- **authIn** pointer to the buffer containing the input authentication vector
- **authInSz** length of the input authentication vector

See:

- wc_AesGcmSetKey
- wc_AesGcmDecrypt

Return: 0 On successfully encrypting the input message

Example

```
Aes enc;
// initialize Aes structure by calling wc_AesInit() and wc_AesGcmSetKey

byte plain[AES_BLOCK_LENGTH * n]; //n being a positive integer
making plain some multiple of 16 bytes
// initialize plain with msg to encrypt
byte cipher[sizeof(plain)];
byte iv[] = // some 16 byte iv
byte authTag[AUTH_TAG_LENGTH];
byte authIn[] = // Authentication Vector

wc_AesGcmEncrypt(&enc, cipher, plain, sizeof(cipher), iv, sizeof(iv),
        authTag, sizeof(authTag), authIn, sizeof(authIn));
```

C.1.2.11 function wc_AesGcmDecrypt

```
int wc_AesGcmDecrypt(
    Aes * aes,
    byte * out,
    const byte * in,
```

```
    word32 sz,
    const byte * iv,
    word32 ivSz,
    const byte * authTag,
    word32 authTagSz,
    const byte * authIn,
    word32 authInSz
)
```

This function decrypts the input cipher text, held in the buffer in, and stores the resulting message text in the output buffer out. It also checks the input authentication vector, authIn, against the supplied authentication tag, authTag. If a nonzero error code is returned, the output data is undefined. However, callers must unconditionally zeroize the output buffer to guard against leakage of cleartext data.

Parameters:

- **aes** pointer to the AES object used to encrypt data
- **out** pointer to the output buffer in which to store the message text size must match in's size (sz)
- **in** pointer to the input buffer holding the cipher text to decrypt size must be a multiple of AES_BLOCK_LENGTH, padded if necessary
- **sz** length of the cipher text to decrypt
- **iv** pointer to the buffer containing the initialization vector
- **ivSz** length of the initialization vector
- **authTag** pointer to the buffer containing the authentication tag
- **authTagSz** length of the desired authentication tag
- **authIn** pointer to the buffer containing the input authentication vector
- **authInSz** length of the input authentication vector

See:

- wc_AesGcmSetKey
- wc_AesGcmEncrypt

Return:

- 0 On successfully decrypting and authenticating the input message
- AES_GCM_AUTH_E If the authentication tag does not match the supplied authentication code vector, authTag.

Example

```
Aes enc; //can use the same struct as was passed to wc_AesGcmEncrypt
// initialize aes structure by calling wc_AesInit and wc_AesGcmSetKey
// if not already done

byte cipher[AES_BLOCK_LENGTH * n]; //n being a positive integer
making cipher some multiple of 16 bytes
// initialize cipher with cipher text to decrypt
byte output[sizeof(cipher)];
byte iv[] = // some 16 byte iv
byte authTag[AUTH_TAG_LENGTH];
byte authIn[] = // Authentication Vector

wc_AesGcmDecrypt(&enc, output, cipher, sizeof(cipher), iv, sizeof(iv),
        authTag, sizeof(authTag), authIn, sizeof(authIn));
```

C.1.2.12 function wc_GmacSetKey

```
int wc_GmacSetKey(
    Gmac * gmac,
    const byte * key,
    word32 len
)
```

This function initializes and sets the key for a GMAC object to be used for Galois Message Authentication.

Parameters:

- **gmac** pointer to the gmac object used for authentication
- **key** 16, 24, or 32 byte secret key for authentication
- **len** length of the key

See:

- wc_GmacUpdate
- wc_AesInit

Return:

- 0 On successfully setting the key
- BAD_FUNC_ARG Returned if key length is invalid.

Example

```
Gmac gmac;
key[] = { some 16, 24, or 32 byte length key };
wc_AesInit(gmac.aes, HEAP_HINT, INVALID_DEVID); // Make sure devId updated
wc_GmacSetKey(&gmac, key, sizeof(key));
```

C.1.2.13 function wc_GmacUpdate

```
int wc_GmacUpdate(
    Gmac * gmac,
    const byte * iv,
    word32 ivSz,
    const byte * authIn,
    word32 authInSz,
    byte * authTag,
    word32 authTagSz
)
```

This function generates the Gmac hash of the authIn input and stores the result in the authTag buffer. After running wc_GmacUpdate, one should compare the generated authTag to a known authentication tag to verify the authenticity of a message.

Parameters:

- **gmac** pointer to the gmac object used for authentication
- **iv** initialization vector used for the hash
- **ivSz** size of the initialization vector used
- **authIn** pointer to the buffer containing the authentication vector to verify
- **authInSz** size of the authentication vector
- **authTag** pointer to the output buffer in which to store the Gmac hash
- **authTagSz** the size of the output buffer used to store the Gmac hash

See:

- wc_GmacSetKey

- wc_AesInit

Return: 0 On successfully computing the Gmac hash.

Example

```
Gmac gmac;
key[] = { some 16, 24, or 32 byte length key };
iv[] = { some 16 byte length iv };

wc_AesInit(gmac.aes, HEAP_HINT, INVALID_DEVID); // Make sure devId updated
wc_GmacSetKey(&gmac, key, sizeof(key));
authIn[] = { some 16 byte authentication input };
tag[AES_BLOCK_SIZE]; // will store authentication code

wc_GmacUpdate(&gmac, iv, sizeof(iv), authIn, sizeof(authIn), tag,
sizeof(tag));
```

C.1.2.14 function wc_AesCcmSetKey

```
int wc_AesCcmSetKey(
    Aes * aes,
    const byte * key,
    word32 keySz
)
```

This function sets the key for an AES object using CCM (Counter with CBC-MAC). It takes a pointer to an AES structure and initializes it with supplied key.

Parameters:

- **aes** aes structure in which to store the supplied key
- **key** 16, 24, or 32 byte secret key for encryption and decryption
- **keySz** size of the supplied key

See:

- wc_AesCcmEncrypt
- wc_AesCcmDecrypt

Return: none

Example

```
Aes enc;
key[] = { some 16, 24, or 32 byte length key };

wc_AesInit(&enc, HEAP_HINT, INVALID_DEVID); // Make sure devId updated
wc_AesCcmSetKey(&enc, key, sizeof(key));
```

C.1.2.15 function wc_AesCcmEncrypt

```
int wc_AesCcmEncrypt(
    Aes * aes,
    byte * out,
    const byte * in,
    word32 inSz,
    const byte * nonce,
    word32 nonceSz,
    byte * authTag,
```

```
    word32 authTagSz,
    const byte * authIn,
    word32 authInSz
)
```

This function encrypts the input message, in, into the output buffer, out, using CCM (Counter with CBC-MAC). It subsequently calculates and stores the authorization tag, authTag, from the authIn input.

Parameters:

- **aes** pointer to the AES object used to encrypt data
- **out** pointer to the output buffer in which to store the cipher text
- **in** pointer to the input buffer holding the message to encrypt
- **sz** length of the input message to encrypt
- **nonce** pointer to the buffer containing the nonce (number only used once)
- **nonceSz** length of the nonce
- **authTag** pointer to the buffer in which to store the authentication tag
- **authTagSz** length of the desired authentication tag
- **authIn** pointer to the buffer containing the input authentication vector
- **authInSz** length of the input authentication vector

See:

- wc_AesCcmSetKey
- wc_AesCcmDecrypt

Return: none

Example

```
Aes enc;
// initialize enc with wc_AesInit and wc_AesCcmSetKey

nonce[] = { initialize nonce };
plain[] = { some plain text message };
cipher[sizeof(plain)];

authIn[] = { some 16 byte authentication input };
tag[AES_BLOCK_SIZE]; // will store authentication code

wc_AesCcmEncrypt(&enc, cipher, plain, sizeof(plain), nonce, sizeof(nonce),
        tag, sizeof(tag), authIn, sizeof(authIn));
```

C.1.2.16 function wc_AesCcmDecrypt

```
int wc_AesCcmDecrypt(
    Aes * aes,
    byte * out,
    const byte * in,
    word32 inSz,
    const byte * nonce,
    word32 nonceSz,
    const byte * authTag,
    word32 authTagSz,
    const byte * authIn,
    word32 authInSz
)
```

This function decrypts the input cipher text, in, into the output buffer, out, using CCM (Counter with CBC-MAC). It subsequently calculates the authorization tag, authTag, from the authIn input. If a nonzero error code is returned, the output data is undefined. However, callers must unconditionally zeroize the output buffer to guard against leakage of cleartext data.

Parameters:

- **aes** pointer to the AES object used to encrypt data
- **out** pointer to the output buffer in which to store the cipher text
- **in** pointer to the input buffer holding the message to encrypt
- **sz** length of the input cipher text to decrypt
- **nonce** pointer to the buffer containing the nonce (number only used once)
- **nonceSz** length of the nonce
- **authTag** pointer to the buffer in which to store the authentication tag
- **authTagSz** length of the desired authentication tag
- **authIn** pointer to the buffer containing the input authentication vector
- **authInSz** length of the input authentication vector

See:

- wc_AesCcmSetKey
- wc_AesCcmEncrypt

Return:

- 0 On successfully decrypting the input message
- AES_CCM_AUTH_E If the authentication tag does not match the supplied authentication code vector, authTag.

Example

```
Aes dec;
// initialize dec with wc_AesInit and wc_AesCcmSetKey

nonce[] = { initialize nonce };
cipher[] = { encrypted message };
plain[sizeof(cipher)];

authIn[] = { some 16 byte authentication input };
tag[AES_BLOCK_SIZE] = { authentication tag received for verification };

int return = wc_AesCcmDecrypt(&dec, plain, cipher, sizeof(cipher),
nonce, sizeof(nonce),tag, sizeof(tag), authIn, sizeof(authIn));
if(return != 0) {
// decrypt error, invalid authentication code
}
```

C.1.2.17 function wc_AesXtsInit

```
int wc_AesXtsInit(
    XtsAes * aes,
    void * heap,
    int devId
)
```

This is to initialize an AES-XTS context. It is up to user to call wc_AesXtsFree on aes key when done.

Parameters:

- **aes** AES keys for encrypt/decrypt process

- **heap** heap hint to use for memory. Can be NULL
- **devId** ID to use with crypto callbacks or async hardware. Set to INVALID_DEVID (-2) if not used

See:

- wc_AesXtsSetKey
- wc_AesXtsSetKeyNoInit
- wc_AesXtsEncrypt
- wc_AesXtsDecrypt
- wc_AesXtsFree

Return: 0 Success

Example

```
XtsAes aes;

if(wc_AesXtsInit(&aes, NULL, INVALID_DEVID) != 0)
{
    // Handle error
}
if(wc_AesXtsSetKeyNoInit(&aes, key, sizeof(key), AES_ENCRYPTION) != 0)
{
    // Handle error
}
wc_AesXtsFree(&aes);
```

C.1.2.18 function wc_AesXtsSetKeyNoInit

```
int wc_AesXtsSetKeyNoInit(
    XtsAes * aes,
    const byte * key,
    word32 len,
    int dir
)
```

This is to help with setting keys to correct encrypt or decrypt type, after first calling wc_AesXtsInit(). It is up to user to call wc_AesXtsFree on aes key when done.

Parameters:

- **aes** AES keys for encrypt/decrypt process
- **key** buffer holding aes key | tweak key
- **len** length of key buffer in bytes. Should be twice that of key size. i.e. 32 for a 16 byte key.
- **dir** direction, either AES_ENCRYPTION or AES_DECRYPTION

See:

- wc_AesXtsEncrypt
- wc_AesXtsDecrypt
- wc_AesXtsFree

Return: 0 Success

Example

```
XtsAes aes;

if(wc_AesXtsInit(&aes, NULL, 0) != 0)
{
    // Handle error
```

```
}
if(wc_AesXtsSetKeyNoInit(&aes, key, sizeof(key), AES_ENCRYPTION, NULL, 0)
    != 0)
{
    // Handle error
}
wc_AesXtsFree(&aes);
```

C.1.2.19 function wc_AesXtsSetKey

```
int wc_AesXtsSetKey(
    XtsAes * aes,
    const byte * key,
    word32 len,
    int dir,
    void * heap,
    int devId
)
```

This is to help with setting keys to correct encrypt or decrypt type. It is up to user to call wc_AesXtsFree on aes key when done.

Parameters:

- **aes** AES keys for encrypt/decrypt process
- **key** buffer holding aes key | tweak key
- **len** length of key buffer in bytes. Should be twice that of key size. i.e. 32 for a 16 byte key.
- **dir** direction, either AES_ENCRYPTION or AES_DECRYPTION
- **heap** heap hint to use for memory. Can be NULL
- **devId** ID to use with crypto callbacks or async hardware. Set to INVALID_DEVID (-2) if not used

See:

- wc_AesXtsInit
- wc_AesXtsSetKeyNoInit
- wc_AesXtsEncrypt
- wc_AesXtsDecrypt
- wc_AesXtsFree

Return: 0 Success

Example

```
XtsAes aes;

if(wc_AesXtsSetKey(&aes, key, sizeof(key), AES_ENCRYPTION, NULL, INVALID_DEVID)
 ↪  != 0)
{
    // Handle error
}
wc_AesXtsFree(&aes);
```

C.1.2.20 function wc_AesXtsEncryptSector

```
int wc_AesXtsEncryptSector(
    XtsAes * aes,
    byte * out,
    const byte * in,
```

```
    word32 sz,
    word64 sector
)
```

Same process as wc_AesXtsEncrypt but uses a word64 type as the tweak value instead of a byte array. This just converts the word64 to a byte array and calls wc_AesXtsEncrypt.

Parameters:

- **aes** AES keys to use for block encrypt/decrypt
- **out** output buffer to hold cipher text
- **in** input plain text buffer to encrypt
- **sz** size of both out and in buffers
- **sector** value to use for tweak

See:

- wc_AesXtsEncrypt
- wc_AesXtsDecrypt
- wc_AesXtsInit
- wc_AesXtsSetKeyNoInit
- wc_AesXtsSetKey
- wc_AesXtsFree

Return: 0 Success

Example

```
XtsAes aes;
unsigned char plain[SIZE];
unsigned char cipher[SIZE];
word64 s = VALUE;

//set up keys with AES_ENCRYPTION as dir

if(wc_AesXtsEncryptSector(&aes, cipher, plain, SIZE, s) != 0)
{
    // Handle error
}
wc_AesXtsFree(&aes);
```

C.1.2.21 function wc_AesXtsDecryptSector

```
int wc_AesXtsDecryptSector(
    XtsAes * aes,
    byte * out,
    const byte * in,
    word32 sz,
    word64 sector
)
```

Same process as wc_AesXtsDecrypt but uses a word64 type as the tweak value instead of a byte array. This just converts the word64 to a byte array.

Parameters:

- **aes** AES keys to use for block encrypt/decrypt
- **out** output buffer to hold plain text
- **in** input cipher text buffer to decrypt
- **sz** size of both out and in buffers

- **sector** value to use for tweak

See:

- wc_AesXtsEncrypt
- wc_AesXtsDecrypt
- wc_AesXtsInit
- wc_AesXtsSetKeyNoInit
- wc_AesXtsSetKey
- wc_AesXtsFree

Return: 0 Success

Example

```
XtsAes aes;
unsigned char plain[SIZE];
unsigned char cipher[SIZE];
word64 s = VALUE;

//set up aes key with AES_DECRYPTION as dir and tweak with AES_ENCRYPTION

if(wc_AesXtsDecryptSector(&aes, plain, cipher, SIZE, s) != 0)
{
    // Handle error
}
wc_AesXtsFree(&aes);
```

C.1.2.22 function wc_AesXtsEncrypt

```
int wc_AesXtsEncrypt(
    XtsAes * aes,
    byte * out,
    const byte * in,
    word32 sz,
    const byte * i,
    word32 iSz
)
```

AES with XTS mode. (XTS) XEX encryption with Tweak and cipher text Stealing.

Parameters:

- **aes** AES keys to use for block encrypt/decrypt
- **out** output buffer to hold cipher text
- **in** input plain text buffer to encrypt
- **sz** size of both out and in buffers
- **i** value to use for tweak
- **iSz** size of i buffer, should always be AES_BLOCK_SIZE but having this input adds a sanity check on how the user calls the function.

See:

- wc_AesXtsDecrypt
- wc_AesXtsInit
- wc_AesXtsSetKeyNoInit
- wc_AesXtsSetKey
- wc_AesXtsFree

Return: 0 Success

Example

```
XtsAes aes;
unsigned char plain[SIZE];
unsigned char cipher[SIZE];
unsigned char i[AES_BLOCK_SIZE];

//set up key with AES_ENCRYPTION as dir

if(wc_AesXtsEncrypt(&aes, cipher, plain, SIZE, i, sizeof(i)) != 0)
{
    // Handle error
}
wc_AesXtsFree(&aes);
```

C.1.2.23 function wc_AesXtsDecrypt

```
int wc_AesXtsDecrypt(
    XtsAes * aes,
    byte * out,
    const byte * in,
    word32 sz,
    const byte * i,
    word32 iSz
)
```

Same process as encryption but Aes key is AES_DECRYPTION type.

Parameters:

- **aes** AES keys to use for block encrypt/decrypt
- **out** output buffer to hold plain text
- **in** input cipher text buffer to decrypt
- **sz** size of both out and in buffers
- **i** value to use for tweak
- **iSz** size of i buffer, should always be AES_BLOCK_SIZE but having this input adds a sanity check on how the user calls the function.

See:

- wc_AesXtsEncrypt
- wc_AesXtsInit
- wc_AesXtsSetKeyNoInit
- wc_AesXtsSetKey
- wc_AesXtsFree

Return: 0 Success

Example

```
XtsAes aes;
unsigned char plain[SIZE];
unsigned char cipher[SIZE];
unsigned char i[AES_BLOCK_SIZE];

//set up key with AES_DECRYPTION as dir and tweak with AES_ENCRYPTION
```

```
if(wc_AesXtsDecrypt(&aes, plain, cipher, SIZE, i, sizeof(i)) != 0)
{
    // Handle error
}
wc_AesXtsFree(&aes);
```

C.1.2.24 function wc_AesXtsFree

```
int wc_AesXtsFree(
    XtsAes * aes
)
```

This is to free up any resources used by the XtsAes structure.

Parameters:

- **aes** AES keys to free

See:

- wc_AesXtsEncrypt
- wc_AesXtsDecrypt
- wc_AesXtsInit
- wc_AesXtsSetKeyNoInit
- wc_AesXtsSetKey

Return: 0 Success

Example

```
XtsAes aes;

if(wc_AesXtsSetKey(&aes, key, sizeof(key), AES_ENCRYPTION, NULL, 0) != 0)
{
    // Handle error
}
wc_AesXtsFree(&aes);
```

C.1.2.25 function wc_AesInit

```
int wc_AesInit(
    Aes * aes,
    void * heap,
    int devId
)
```

Initialize Aes structure. Sets heap hint to be used and ID for use with async hardware. It is up to the user to call wc_AesFree on the Aes structure when done.

Parameters:

- **aes** aes structure in to initialize
- **heap** heap hint to use for malloc / free if needed
- **devId** ID to use with crypto callbacks or async hardware. Set to INVALID_DEVID (-2) if not used

See:

- wc_AesSetKey
- wc_AesSetIV
- wc_AesFree

Return: 0 Success

Example

```
Aes enc;
void* hint = NULL;
int devId = INVALID_DEVID; //if not using async INVALID_DEVID is default

//heap hint could be set here if used

wc_AesInit(&enc, hint, devId);
```

C.1.2.26 function wc_AesFree

```
int wc_AesFree(
    Aes * aes
)
```

free resources associated with the Aes structure when applicable. Internally may sometimes be a no-op but still recommended to call in all cases as a general best-practice (IE if application code is ported for use on new environments where the call is applicable).

Parameters:

- **aes** aes structure in to free

See: wc_AesInit

Return: no return (void function)

Example

```
Aes enc;
void* hint = NULL;
int devId = INVALID_DEVID; //if not using async INVALID_DEVID is default

//heap hint could be set here if used

wc_AesInit(&enc, hint, devId);
// ... do some interesting things ...
wc_AesFree(&enc);
```

C.1.2.27 function wc_AesCfbEncrypt

```
int wc_AesCfbEncrypt(
    Aes * aes,
    byte * out,
    const byte * in,
    word32 sz
)
```

AES with CFB mode.

Parameters:

- **aes** AES keys to use for block encrypt/decrypt
- **out** output buffer to hold cipher text must be at least as large as inputbuffer)
- **in** input plain text buffer to encrypt
- **sz** size of input buffer

See:

- wc_AesCfbDecrypt
- wc_AesSetKey

Return: 0 Success and negative error values on failure

Example

```
Aes aes;
unsigned char plain[SIZE];
unsigned char cipher[SIZE];

//set up key with AES_ENCRYPTION as dir for both encrypt and decrypt

if(wc_AesCfbEncrypt(&aes, cipher, plain, SIZE) != 0)
{
    // Handle error
}
```

C.1.2.28 function wc_AesCfbDecrypt

```
int wc_AesCfbDecrypt(
    Aes * aes,
    byte * out,
    const byte * in,
    word32 sz
)
```

AES with CFB mode.

Parameters:

- **aes** AES keys to use for block encrypt/decrypt
- **out** output buffer to hold decrypted text must be at least as large as inputbuffer)
- **in** input buffer to decrypt
- **sz** size of input buffer

See:

- wc_AesCfbEncrypt
- wc_AesSetKey

Return: 0 Success and negative error values on failure

Example

```
Aes aes;
unsigned char plain[SIZE];
unsigned char cipher[SIZE];

//set up key with AES_ENCRYPTION as dir for both encrypt and decrypt

if(wc_AesCfbDecrypt(&aes, plain, cipher, SIZE) != 0)
{
    // Handle error
}
```

C.1.2.29 function wc_AesSivEncrypt

```
int wc_AesSivEncrypt(
    const byte * key,
```

```
    word32 keySz,
    const byte * assoc,
    word32 assocSz,
    const byte * nonce,
    word32 nonceSz,
    const byte * in,
    word32 inSz,
    byte * siv,
    byte * out
)
```

This function performs SIV (synthetic initialization vector) encryption as described in RFC 5297.

Parameters:

- **key** Byte buffer containing the key to use.
- **keySz** Length of the key buffer in bytes.
- **assoc** Additional, authenticated associated data (AD).
- **assocSz** Length of AD buffer in bytes.
- **nonce** A number used once. Used by the algorithm in the same manner as the AD.
- **nonceSz** Length of nonce buffer in bytes.
- **in** Plaintext buffer to encrypt.
- **inSz** Length of plaintext buffer.
- **siv** The SIV output by S2V (see RFC 5297 2.4).
- **out** Buffer to hold the ciphertext. Should be the same length as the plaintext buffer.

See: wc_AesSivDecrypt

Return:

- 0 On successful encryption.
- BAD_FUNC_ARG If key, SIV, or output buffer are NULL. Also returned if the key size isn't 32, 48, or 64 bytes.
- Other Other negative error values returned if AES or CMAC operations fail.

Example

```
byte key[] = { some 32, 48, or 64 byte key };
byte assoc[] = {0x01, 0x2, 0x3};
byte nonce[] = {0x04, 0x5, 0x6};
byte plainText[] = {0xDE, 0xAD, 0xBE, 0xEF};
byte siv[AES_BLOCK_SIZE];
byte cipherText[sizeof(plainText)];
if (wc_AesSivEncrypt(key, sizeof(key), assoc, sizeof(assoc), nonce,
    sizeof(nonce), plainText, sizeof(plainText), siv, cipherText) != 0) {
    // failed to encrypt
}
```

C.1.2.30 function wc_AesSivDecrypt

```
int wc_AesSivDecrypt(
    const byte * key,
    word32 keySz,
    const byte * assoc,
    word32 assocSz,
    const byte * nonce,
    word32 nonceSz,
    const byte * in,
```

```
    word32 inSz,
    byte * siv,
    byte * out
)
```

This function performs SIV (synthetic initialization vector) decryption as described in RFC 5297. If a nonzero error code is returned, the output data is undefined. However, callers must unconditionally zeroize the output buffer to guard against leakage of cleartext data.

Parameters:

- **key** Byte buffer containing the key to use.
- **keySz** Length of the key buffer in bytes.
- **assoc** Additional, authenticated associated data (AD).
- **assocSz** Length of AD buffer in bytes.
- **nonce** A number used once. Used by the underlying algorithm in the same manner as the AD.
- **nonceSz** Length of nonce buffer in bytes.
- **in** Ciphertext buffer to decrypt.
- **inSz** Length of ciphertext buffer.
- **siv** The SIV that accompanies the ciphertext (see RFC 5297 2.4).
- **out** Buffer to hold the decrypted plaintext. Should be the same length as the ciphertext buffer.

See: wc_AesSivEncrypt

Return:

- 0 On successful decryption.
- BAD_FUNC_ARG If key, SIV, or output buffer are NULL. Also returned if the key size isn't 32, 48, or 64 bytes.
- AES_SIV_AUTH_E If the SIV derived by S2V doesn't match the input SIV (see RFC 5297 2.7).
- Other Other negative error values returned if AES or CMAC operations fail.

Example

```
byte key[] = { some 32, 48, or 64 byte key };
byte assoc[] = {0x01, 0x2, 0x3};
byte nonce[] = {0x04, 0x5, 0x6};
byte cipherText[] = {0xDE, 0xAD, 0xBE, 0xEF};
byte siv[AES_BLOCK_SIZE] = { the SIV that came with the ciphertext };
byte plainText[sizeof(cipherText)];
if (wc_AesSivDecrypt(key, sizeof(key), assoc, sizeof(assoc), nonce,
    sizeof(nonce), cipherText, sizeof(cipherText), siv, plainText) != 0) {
    // failed to decrypt
}
```

C.1.2.31 function wc_AesEaxEncryptAuth

```
WOLFSSL_API int wc_AesEaxEncryptAuth(
    const byte * key,
    word32 keySz,
    byte * out,
    const byte * in,
    word32 inSz,
    const byte * nonce,
    word32 nonceSz,
    byte * authTag,
    word32 authTagSz,
    const byte * authIn,
```

```
    word32 authInSz
)
```

This function performs AES EAX encryption and authentication as described in “EAX: A Conventional Authenticated-Encryption Mode” (https://eprint.iacr.org/2003/069). It is a “one-shot” API that performs all encryption and authentication operations in one function call.

Parameters:

- **key** buffer containing the key to use
- **keySz** length of the key buffer in bytes
- **out** buffer to hold the ciphertext. Should be the same length as the plaintext buffer
- **in** plaintext buffer to encrypt
- **inSz** length of plaintext buffer
- **nonce** the cryptographic nonce to use for EAX operations
- **nonceSz** length of nonce buffer in bytes
- **authTag** pointer to the buffer in which to store the authentication tag
- **authTagSz** length of the desired authentication tag
- **authIn** pointer to the buffer containing input data to authenticate
- **authInSz** length of the input authentication data

See: wc_AesEaxDecryptAuth

Return:

- 0 on successful encryption.
- BAD_FUNC_ARG if input or output buffers are NULL. Also returned if the key size isn’t a valid AES key size (16, 24, or 32 bytes)
- other negative error values returned if AES or CMAC operations fail.

Example

```
byte key[] = { some 32, 48, or 64 byte key };
byte nonce[] = {0x04, 0x5, 0x6};
byte plainText[] = {0xDE, 0xAD, 0xBE, 0xEF};
byte authIn[] = {0x01, 0x2, 0x3};

byte cipherText[sizeof(plainText)]; // output ciphertext
byte authTag[length, up to AES_BLOCK_SIZE]; // output authTag

if (wc_AesEaxEncrypt(key, sizeof(key),
                     cipherText, plainText, sizeof(plainText),
                     nonce, sizeof(nonce),
                     authTag, sizeof(authTag),
                     authIn, sizeof(authIn)) != 0) {
    // failed to encrypt
}
```

C.1.2.32 function wc_AesEaxDecryptAuth

```
WOLFSSL_API int wc_AesEaxDecryptAuth(
    const byte * key,
    word32 keySz,
    byte * out,
    const byte * in,
    word32 inSz,
    const byte * nonce,
    word32 nonceSz,
```

```
    const byte * authTag,
    word32 authTagSz,
    const byte * authIn,
    word32 authInSz
)
```

This function performs AES EAX decryption and authentication as described in “EAX: A Conventional Authenticated-Encryption Mode” (https://eprint.iacr.org/2003/069). It is a “one-shot” API that performs all decryption and authentication operations in one function call. If a nonzero error code is returned, the output data is undefined. However, callers must unconditionally zeroize the output buffer to guard against leakage of cleartext data.

Parameters:

- **key** byte buffer containing the key to use
- **keySz** length of the key buffer in bytes
- **out** buffer to hold the plaintext. Should be the same length as the input ciphertext buffer
- **in** ciphertext buffer to decrypt
- **inSz** length of ciphertext buffer
- **nonce** the cryptographic nonce to use for EAX operations
- **nonceSz** length of nonce buffer in bytes
- **authTag** buffer that holds the authentication tag to check the authenticity of the data against
- **authTagSz** Length of the input authentication tag
- **authIn** pointer to the buffer containing input data to authenticate
- **authInSz** length of the input authentication data

See: wc_AesEaxEncryptAuth

Return:

- 0 on successful decryption
- BAD_FUNC_ARG if input or output buffers are NULL. Also returned if the key size isn’t a valid AES key size (16, 24, or 32 bytes)
- AES_EAX_AUTH_E If the authentication tag does not match the supplied authentication code vector authTag
- other negative error values returned if AES or CMAC operations fail.

Example

```
byte key[] = { some 32, 48, or 64 byte key };
byte nonce[] = {0x04, 0x5, 0x6};
byte cipherText[] = {0xDE, 0xAD, 0xBE, 0xEF};
byte authIn[] = {0x01, 0x2, 0x3};

byte plainText[sizeof(cipherText)]; // output plaintext
byte authTag[length, up to AES_BLOCK_SIZE]; // output authTag

if (wc_AesEaxDecrypt(key, sizeof(key),
                     cipherText, plainText, sizeof(plainText),
                     nonce, sizeof(nonce),
                     authTag, sizeof(authTag),
                     authIn, sizeof(authIn)) != 0) {
    // failed to encrypt
}
```

C.1.2.33 function wc_AesEaxInit

```
WOLFSSL_API int wc_AesEaxInit(
    AesEax * eax,
    const byte * key,
    word32 keySz,
    const byte * nonce,
    word32 nonceSz,
    const byte * authIn,
    word32 authInSz
)
```

This function initializes an AesEax object for use in authenticated encryption or decryption. This function must be called on an AesEax object before using it with any of the AES EAX incremental API functions. It does not need to be called if using the one-shot EAX API functions. All AesEax instances initialized with this function need to be freed with a call to wc_AesEaxFree() when done using the instance.

Parameters:

- **eax** AES EAX structure holding the context of the AEAD operation
- **key** 16, 24, or 32 byte secret key for encryption and decryption
- **keySz** length of the supplied key in bytes
- **nonce** the cryptographic nonce to use for EAX operations
- **nonceSz** length of nonce buffer in bytes
- **authIn** (optional) input data to add to the authentication stream This argument should be NULL if not used
- **authInSz** size in bytes of the input authentication data

See:

- wc_AesEaxEncryptUpdate
- wc_AesEaxDecryptUpdate
- wc_AesEaxAuthDataUpdate
- wc_AesEaxEncryptFinal
- wc_AesEaxDecryptFinal
- wc_AesEaxFree

Return:

- 0 on success
- error code on failure

Example

```
AesEax eax;
key[]   = { some 16, 24, or 32 byte length key };
nonce[] = { some arbitrary length nonce };
authIn[] = { some data to add to the authentication stream };
plainText[] = {some plaintext data to encrypt};

cipherText[sizeof(plainText)]; // buffer to hold cipherText
authTag[length, up to AES_BLOCK_SIZE]; // buffer to hold computed auth data

AesEax eax;

if ((ret = wc_AesEaxInit(eax,
                         key, keySz,
                         nonce, nonceSz,
                         authIn, authInSz)) != 0) {
    goto cleanup;
```

```
}

// if we wanted to add more auth data, we could provide it at this point,
// otherwise we use NULL for the authIn parameter, with authIn size of 0
if ((ret = wc_AesEaxEncryptUpdate(eax,
                                  cipherText, plainText, sizeof(plainText),
                                  NULL, 0)) != 0) {
    goto cleanup;
}

if ((ret = wc_AesEaxEncryptFinal(eax, authTag, sizeof(authTag))) != 0) {
    goto cleanup;
}

cleanup:
    wc_AesEaxFree(eax);
```

C.1.2.34 function wc_AesEaxEncryptUpdate

```
WOLFSSL_API int wc_AesEaxEncryptUpdate(
    AesEax * eax,
    byte * out,
    const byte * in,
    word32 inSz,
    const byte * authIn,
    word32 authInSz
)
```

This function uses AES EAX to encrypt input data, and optionally, add more input data to the authentication stream. eax must have been previously initialized with a call to wc_AesEaxInit.

Parameters:

- **eax** AES EAX structure holding the context of the AEAD operation
- **out** output buffer holding the ciphertext
- **in** input buffer holding the plaintext to encrypt
- **inSz** size in bytes of the input data buffer
- **authIn** (optional) input data to add to the authentication stream This argument should be NULL if not used
- **authInSz** size in bytes of the input authentication data

See:

- wc_AesEaxInit
- wc_AesEaxDecryptUpdate
- wc_AesEaxAuthDataUpdate
- wc_AesEaxEncryptFinal
- wc_AesEaxDecryptFinal
- wc_AesEaxFree

Return:

- 0 on success
- error code on failure

Example

```
AesEax eax;
key[]   = { some 16, 24, or 32 byte length key };
```

```
nonce[] = { some arbitrary length nonce };
authIn[] = { some data to add to the authentication stream };
plainText[] = {some plaintext data to encrypt};

cipherText[sizeof(plainText)]; // buffer to hold cipherText
authTag[length, up to AES_BLOCK_SIZE]; // buffer to hold computed auth data

AesEax eax;

if ((ret = wc_AesEaxInit(eax,
                         key, keySz,
                         nonce, nonceSz,
                         authIn, authInSz)) != 0) {
    goto cleanup;
}

// if we wanted to add more auth data, we could provide it at this point,
// otherwise we use NULL for the authIn parameter, with authInSz of 0
if ((ret = wc_AesEaxEncryptUpdate(eax,
                                  cipherText, plainText, sizeof(plainText),
                                  NULL, 0)) != 0) {
    goto cleanup;
}

if ((ret = wc_AesEaxEncryptFinal(eax, authTag, sizeof(authTag))) != 0) {
    goto cleanup;
}

cleanup:
    wc_AesEaxFree(eax);
```

C.1.2.35 function wc_AesEaxDecryptUpdate

```
WOLFSSL_API int wc_AesEaxDecryptUpdate(
    AesEax * eax,
    byte * out,
    const byte * in,
    word32 inSz,
    const byte * authIn,
    word32 authInSz
)
```

This function uses AES EAX to decrypt input data, and optionally, add more input data to the authentication stream. eax must have been previously initialized with a call to wc_AesEaxInit.

Parameters:

- **eax** AES EAX structure holding the context of the AEAD operation
- **out** output buffer holding the decrypted plaintext
- **in** input buffer holding the ciphertext
- **inSz** size in bytes of the input data buffer
- **authIn** (optional) input data to add to the authentication stream This argument should be NULL if not used
- **authInSz** size in bytes of the input authentication data

See:

- wc_AesEaxInit
- wc_AesEaxEncryptUpdate
- wc_AesEaxAuthDataUpdate
- wc_AesEaxEncryptFinal
- wc_AesEaxDecryptFinal
- wc_AesEaxFree

Return:

- 0 on success
- error code on failure

Example

```
AesEax eax;
key[]   = { some 16, 24, or 32 byte length key };
nonce[] = { some arbitrary length nonce };
authIn[] = { some data to add to the authentication stream };
cipherText[] = {some encrypted data};

plainText[sizeof(cipherText)]; // buffer to hold decrypted data
// auth tag is generated elsewhere by the encrypt AEAD operation
authTag[length, up to AES_BLOCK_SIZE] = { the auth tag };

AesEax eax;

if ((ret = wc_AesEaxInit(eax,
                         key, keySz,
                         nonce, nonceSz,
                         authIn, authInSz)) != 0) {
    goto cleanup;
}

// if we wanted to add more auth data, we could provide it at this point,
// otherwise we use NULL for the authIn parameter, with authInSz of 0
if ((ret = wc_AesEaxDecryptUpdate(eax,
                                  plainText, cipherText, sizeof(cipherText),
                                  NULL, 0)) != 0) {
    goto cleanup;
}

if ((ret = wc_AesEaxDecryptFinal(eax, authTag, sizeof(authTag))) != 0) {
    goto cleanup;
}

cleanup:
    wc_AesEaxFree(eax);
```

C.1.2.36 function wc_AesEaxAuthDataUpdate

```
WOLFSSL_API int wc_AesEaxAuthDataUpdate(
    AesEax * eax,
    const byte * authIn,
    word32 authInSz
)
```

This function adds input data to the authentication stream. eax must have been previously initialized with a call to wc_AesEaxInit.

Parameters:

- **eax** AES EAX structure holding the context of the AEAD operation
- **authIn** input data to add to the authentication stream
- **authInSz** size in bytes of the input authentication data

See:

- wc_AesEaxInit
- wc_AesEaxEncryptUpdate
- wc_AesEaxDecryptUpdate
- wc_AesEaxEncryptFinal
- wc_AesEaxDecryptFinal
- wc_AesEaxFree

Return:

- 0 on success
- error code on failure

Example

```
AesEax eax;
key[]   = { some 16, 24, or 32 byte length key };
nonce[] = { some arbitrary length nonce };
authIn[] = { some data to add to the authentication stream };
cipherText[] = {some encrypted data};

plainText[sizeof(cipherText)]; // buffer to hold decrypted data
// auth tag is generated elsewhere by the encrypt AEAD operation
authTag[length, up to AES_BLOCK_SIZE] = { the auth tag };

AesEax eax;

// No auth data to add here
if ((ret = wc_AesEaxInit(eax,
                         key, keySz,
                         nonce, nonceSz,
                         NULL, 0)) != 0) {
    goto cleanup;
}

// No auth data to add here, added later with wc_AesEaxAuthDataUpdate
if ((ret = wc_AesEaxDecryptUpdate(eax,
                                  plainText, cipherText, sizeof(cipherText),
                                  NULL, 0)) != 0) {
    goto cleanup;
}

if ((ret = wc_AesEaxAuthDataUpdate(eax, authIn, sizeof(authIn))) != 0) {
    goto cleanup;
}

if ((ret = wc_AesEaxDecryptFinal(eax, authTag, sizeof(authTag))) != 0) {
    goto cleanup;
```

```
}

cleanup:
    wc_AesEaxFree(eax);
```

C.1.2.37 function wc_AesEaxEncryptFinal

```
WOLFSSL_API int wc_AesEaxEncryptFinal(
    AesEax * eax,
    byte * authTag,
    word32 authTagSz
)
```

This function finalizes the encrypt AEAD operation, producing an auth tag over the current authentication stream. eax must have been previously initialized with a call to wc_AesEaxInit.

Parameters:

- **eax** AES EAX structure holding the context of the AEAD operation
- **authTag[out]** buffer that will hold the computed auth tag
- **authTagSz** size in bytes of authTag

See:

- wc_AesEaxInit
- wc_AesEaxEncryptUpdate
- wc_AesEaxDecryptUpdate
- wc_AesEaxAuthDataUpdate
- wc_AesEaxDecryptFinal
- wc_AesEaxFree

Return:

- 0 on success
- error code on failure

Example

```
AesEax eax;
key[]   = { some 16, 24, or 32 byte length key };
nonce[] = { some arbitrary length nonce };
authIn[] = { some data to add to the authentication stream };
plainText[] = {some plaintext data to encrypt};

cipherText[sizeof(plainText)]; // buffer to hold cipherText
authTag[length, up to AES_BLOCK_SIZE]; // buffer to hold computed auth data

AesEax eax;

if ((ret = wc_AesEaxInit(eax,
                         key, keySz,
                         nonce, nonceSz,
                         authIn, authInSz)) != 0) {
    goto cleanup;
}

// if we wanted to add more auth data, we could provide it at this point,
// otherwise we use NULL for the authIn parameter, with authInSz of 0
if ((ret = wc_AesEaxEncryptUpdate(eax,
```

```
                                    cipherText, plainText, sizeof(plainText),
                                    NULL, 0)) != 0) {
    goto cleanup;
}

if ((ret = wc_AesEaxEncryptFinal(eax, authTag, sizeof(authTag))) != 0) {
    goto cleanup;
}

cleanup:
    wc_AesEaxFree(eax);
```

C.1.2.38 function wc_AesEaxDecryptFinal

```
WOLFSSL_API int wc_AesEaxDecryptFinal(
    AesEax * eax,
    const byte * authIn,
    word32 authInSz
)
```

This function finalizes the decrypt AEAD operation, finalizing the auth tag computation and checking it for validity against the user supplied tag. eax must have been previously initialized with a call to wc_AesEaxInit.

Parameters:

- **eax** AES EAX structure holding the context of the AEAD operation
- **authIn** input auth tag to check computed auth tag against
- **authInSz** size in bytes of authIn

See:

- wc_AesEaxInit
- wc_AesEaxEncryptUpdate
- wc_AesEaxDecryptUpdate
- wc_AesEaxAuthDataUpdate
- wc_AesEaxEncryptFinal
- wc_AesEaxFree

Return:

- 0 if data is authenticated successfully
- AES_EAX_AUTH_E if the authentication tag does not match the supplied authentication code vector authIn
- other error code on failure

Example

```
AesEax eax;
key[]   = { some 16, 24, or 32 byte length key };
nonce[] = { some arbitrary length nonce };
authIn[] = { some data to add to the authentication stream };
cipherText[] = {some encrypted data};

plainText[sizeof(cipherText)]; // buffer to hold decrypted data
// auth tag is generated elsewhere by the encrypt AEAD operation
authTag[length, up to AES_BLOCK_SIZE] = { the auth tag };

AesEax eax;
```

```
if ((ret = wc_AesEaxInit(eax,
                         key, keySz,
                         nonce, nonceSz,
                         authIn, authInSz)) != 0) {
    goto cleanup;
}

// if we wanted to add more auth data, we could provide it at this point,
// otherwise we use NULL for the authIn parameter, with authInSz of 0
if ((ret = wc_AesEaxDecryptUpdate(eax,
                                  plainText, cipherText, sizeof(cipherText),
                                  NULL, 0)) != 0) {
    goto cleanup;
}

if ((ret = wc_AesEaxDecryptFinal(eax, authTag, sizeof(authTag))) != 0) {
    goto cleanup;
}

cleanup:
    wc_AesEaxFree(eax);
```

C.1.2.39 function wc_AesEaxFree

```
WOLFSSL_API int wc_AesEaxFree(
    AesEax * eax
)
```

This frees up any resources, specifically keys, used by the Aes instance inside the AesEax wrapper struct. It should be called on the AesEax struct after it has been initialized with wc_AesEaxInit, and all desired EAX operations are complete.

Parameters:

- **eaxAES** EAX instance to free

See:

- wc_AesEaxInit
- wc_AesEaxEncryptUpdate
- wc_AesEaxDecryptUpdate
- wc_AesEaxAuthDataUpdate
- wc_AesEaxEncryptFinal
- wc_AesEaxDecryptFinal

Return: 0 Success

Example

```
AesEax eax;

if(wc_AesEaxInit(eax, key, keySz, nonce, nonceSz, authIn, authInSz) != 0) {
    // handle errors, then free
    wc_AesEaxFree(&eax);
}
```

C.1.2.40 function wc_AesCtsEncrypt

```
int wc_AesCtsEncrypt(
    const byte * key,
    word32 keySz,
    byte * out,
    const byte * in,
    word32 inSz,
    const byte * iv
)
```

This function performs AES encryption using Ciphertext Stealing (CTS) mode. It is a one-shot API that handles all operations in a single call.

Parameters:

- **key** pointer to the AES key used for encryption.
- **keySz** size of the AES key in bytes (16, 24, or 32 bytes).
- **out** buffer to hold the encrypted ciphertext. Must be at least the size of the input.
- **in** pointer to the plaintext input data to encrypt.
- **inSz** size of the plaintext input data in bytes.
- **iv** pointer to the initialization vector (IV) used for encryption. Must be 16 bytes.
- **key** pointer to the AES key used for encryption.
- **keySz** size of the AES key in bytes (16, 24, or 32 bytes).
- **out** buffer to hold the encrypted ciphertext. Must be at least the same size as the input plaintext.
- **in** pointer to the plaintext input data to encrypt.
- **inSz** size of the plaintext input data in bytes.
- **iv** pointer to the initialization vector (IV) used for encryption. Must be 16 bytes. *Example*

```
byte key[16] = { 0 };
byte iv[16] = { 0 };
byte plaintext[] = { 0x01, 0x02, 0x03, 0x04, 0x05 };
byte ciphertext[sizeof(plaintext)];
int ret = wc_AesCtsEncrypt(key, sizeof(key), ciphertext, plaintext,
                           sizeof(plaintext), iv);
if (ret != 0) {
    // handle encryption error
}
```

See:

- wc_AesCtsDecrypt
- wc_AesCtsDecrypt

Return:

- 0 on successful encryption.
- BAD_FUNC_ARG if input arguments are invalid.
- other negative error codes for encryption failures.
- 0 on successful encryption.
- BAD_FUNC_ARG if input arguments are invalid.
- other negative error codes for encryption failures.

Example

```
    byte key[16] = { 0 };
    byte iv[16] = { 0 };
    byte plaintext[] = { 0x01, 0x02, 0x03, 0x04, 0x05 };
    byte ciphertext[sizeof(plaintext)];
```

```
    int ret = wc_AesCtsEncrypt(key, sizeof(key), ciphertext, plaintext,
        sizeof(plaintext), iv);
    if (ret != 0) {
    // handle encryption error
}
```

C.1.2.41 function wc_AesCtsDecrypt

```
int wc_AesCtsDecrypt(
    const byte * key,
    word32 keySz,
    byte * out,
    const byte * in,
    word32 inSz,
    const byte * iv
)
```

This function performs AES decryption using Ciphertext Stealing (CTS) mode. It is a one-shot API that handles all operations in a single call.

Parameters:

- **key** pointer to the AES key used for decryption.
- **keySz** size of the AES key in bytes (16, 24, or 32 bytes).
- **out** buffer to hold the decrypted plaintext. Must be at least the same size as the input ciphertext.
- **in** pointer to the ciphertext input data to decrypt.
- **inSz** size of the ciphertext input data in bytes.
- **iv** pointer to the initialization vector (IV) used for decryption. Must be 16 bytes. *Example*

```
byte key[16] = { 0 };
byte iv[16] = { 0 };
byte ciphertext[] = { 0x01, 0x02, 0x03, 0x04, 0x05 };
byte plaintext[sizeof(ciphertext)];
int ret = wc_AesCtsDecrypt(key, sizeof(key), plaintext, ciphertext,
                           sizeof(ciphertext), iv);
if (ret != 0) {
    // handle decryption error
}
```

See: wc_AesCtsEncrypt

Return:

- 0 on successful decryption.
- BAD_FUNC_ARG if input arguments are invalid.
- other negative error codes for decryption failures.

C.1.2.42 function wc_AesCtsEncryptUpdate

```
int wc_AesCtsEncryptUpdate(
    Aes * aes,
    byte * out,
    word32 * outSz,
    const byte * in,
    word32 inSz
)
```

This function performs an update step of the AES CTS encryption. It processes a chunk of plaintext and stores intermediate data.

Parameters:

- **aes** pointer to the Aes structure holding the context of the operation.
- **out** buffer to hold the encrypted ciphertext. Must be large enough to store the output from this update step.
- **outSz** size in bytes of the output data written to the out buffer. On input, it should contain the maximum number of bytes that can be written to the out buffer.
- **in** pointer to the plaintext input data to encrypt.
- **inSz** size of the plaintext input data in bytes. *Example*

```
Aes aes;
wc_AesInit(&aes, NULL, INVALID_DEVID);
byte key[16] = { 0 };
byte iv[16] = { 0 };
byte plaintext[] = { ... };
byte ciphertext[sizeof(plaintext)];
word32 outSz = sizeof(ciphertext);
wc_AesSetKey(&aes, key, sizeof(key), iv, AES_ENCRYPTION);
int ret = wc_AesCtsEncryptUpdate(&aes, ciphertext, &outSz, plaintext,
 ↪  sizeof(plaintext));
if (ret != 0) {
    // handle error
}
wc_AesFree(&aes);
```

See: wc_AesCtsDecryptUpdate

Return:

- 0 on successful processing.
- BAD_FUNC_ARG if input arguments are invalid.

C.1.2.43 function wc_AesCtsEncryptFinal

```
int wc_AesCtsEncryptFinal(
    Aes * aes,
    byte * out,
    word32 * outSz
)
```

This function finalizes the AES CTS encryption operation. It processes any remaining plaintext and completes the encryption.

Parameters:

- **aes** pointer to the Aes structure holding the context of the operation.
- **out** buffer to hold the final encrypted ciphertext. Must be large enough to store any remaining ciphertext from this final step.
- **outSz** size in bytes of the output data written to the out buffer. On input, it should contain the maximum number of bytes that can be written to the out buffer. *Example*

```
Aes aes;
wc_AesInit(&aes, NULL, INVALID_DEVID);
byte key[16] = { 0 };
byte iv[16] = { 0 };
byte plaintext[] = { ... };
```

```
byte ciphertext[sizeof(plaintext)];
word32 outSz = sizeof(ciphertext);
wc_AesSetKey(&aes, key, sizeof(key), iv, AES_ENCRYPTION);
// Perform any required update steps using wc_AesCtsEncryptUpdate
int ret = wc_AesCtsEncryptFinal(&aes, ciphertext, &outSz);
if (ret != 0) {
    // handle error
}
wc_AesFree(&aes);
```

See: wc_AesCtsDecryptFinal

Return:

- 0 on successful encryption completion.
- BAD_FUNC_ARG if input arguments are invalid.

C.1.2.44 function wc_AesCtsDecryptUpdate

```
int wc_AesCtsDecryptUpdate(
    Aes * aes,
    byte * out,
    word32 * outSz,
    const byte * in,
    word32 inSz
)
```

This function performs an update step of the AES CTS decryption. It processes a chunk of ciphertext and stores intermediate data.

Parameters:

- **aes** pointer to the Aes structure holding the context of the operation.
- **out** buffer to hold the decrypted plaintext. Must be large enough to store the output from this update step.
- **outSz** size in bytes of the output data written to the out buffer. On input, it should contain the maximum number of bytes that can be written to the out buffer.
- **in** pointer to the ciphertext input data to decrypt.
- **inSz** size of the ciphertext input data in bytes. *Example*

```
Aes aes;
wc_AesInit(&aes, NULL, INVALID_DEVID);
byte key[16] = { 0 };
byte iv[16] = { 0 };
byte ciphertext[] = { ... };
byte plaintext[sizeof(ciphertext)];
word32 outSz = sizeof(plaintext);
wc_AesSetKey(&aes, key, sizeof(key), iv, AES_DECRYPTION);
int ret = wc_AesCtsDecryptUpdate(&aes, plaintext, &outSz, ciphertext,
 ↪  sizeof(ciphertext));
if (ret != 0) {
    // handle error
}
wc_AesFree(&aes);
```

See: wc_AesCtsEncryptUpdate

Return:

- 0 on successful processing.
- BAD_FUNC_ARG if input arguments are invalid.

C.1.2.45 function wc_AesCtsDecryptFinal

```
int wc_AesCtsDecryptFinal(
    Aes * aes,
    byte * out,
    word32 * outSz
)
```

This function finalizes the AES CTS decryption operation. It processes any remaining ciphertext and completes the decryption.

Parameters:

- **aes** pointer to the Aes structure holding the context of the operation.
- **out** buffer to hold the final decrypted plaintext. Must be large enough to store any remaining plaintext from this final step.
- **outSz** size in bytes of the output data written to the out buffer. On input, it should contain the maximum number of bytes that can be written to the out buffer. *Example*

```
Aes aes;
wc_AesInit(&aes, NULL, INVALID_DEVID);
byte key[16] = { 0 };
byte iv[16] = { 0 };
byte ciphertext[] = { ... };
byte plaintext[sizeof(ciphertext)];
word32 outSz = sizeof(plaintext);
wc_AesSetKey(&aes, key, sizeof(key), iv, AES_DECRYPTION);
// Perform any required update steps using wc_AesCtsDecryptUpdate
int ret = wc_AesCtsDecryptFinal(&aes, plaintext, &outSz);
if (ret != 0) {
    // handle error
}
wc_AesFree(&aes);
```

See: wc_AesCtsEncryptFinal

Return:

- 0 on successful decryption completion.
- BAD_FUNC_ARG if input arguments are invalid.

C.1.3 Source code

```
int  wc_AesSetKey(Aes* aes, const byte* key, word32 len,
                              const byte* iv, int dir);

int  wc_AesSetIV(Aes* aes, const byte* iv);

int  wc_AesCbcEncrypt(Aes* aes, byte* out,
                                  const byte* in, word32 sz);

int  wc_AesCbcDecrypt(Aes* aes, byte* out,
                                  const byte* in, word32 sz);
```

```
int wc_AesCtrEncrypt(Aes* aes, byte* out,
                                    const byte* in, word32 sz);

int wc_AesEncryptDirect(Aes* aes, byte* out, const byte* in);

int wc_AesDecryptDirect(Aes* aes, byte* out, const byte* in);

int  wc_AesSetKeyDirect(Aes* aes, const byte* key, word32 len,
                                const byte* iv, int dir);

int  wc_AesGcmSetKey(Aes* aes, const byte* key, word32 len);

int  wc_AesGcmEncrypt(Aes* aes, byte* out,
                                    const byte* in, word32 sz,
                                    const byte* iv, word32 ivSz,
                                    byte* authTag, word32 authTagSz,
                                    const byte* authIn, word32 authInSz);

int  wc_AesGcmDecrypt(Aes* aes, byte* out,
                                    const byte* in, word32 sz,
                                    const byte* iv, word32 ivSz,
                                    const byte* authTag, word32 authTagSz,
                                    const byte* authIn, word32 authInSz);

int wc_GmacSetKey(Gmac* gmac, const byte* key, word32 len);

int wc_GmacUpdate(Gmac* gmac, const byte* iv, word32 ivSz,
                               const byte* authIn, word32 authInSz,
                               byte* authTag, word32 authTagSz);

int  wc_AesCcmSetKey(Aes* aes, const byte* key, word32 keySz);

int  wc_AesCcmEncrypt(Aes* aes, byte* out,
                                    const byte* in, word32 inSz,
                                    const byte* nonce, word32 nonceSz,
                                    byte* authTag, word32 authTagSz,
                                    const byte* authIn, word32 authInSz);

int  wc_AesCcmDecrypt(Aes* aes, byte* out,
                                    const byte* in, word32 inSz,
                                    const byte* nonce, word32 nonceSz,
                                    const byte* authTag, word32 authTagSz,
                                    const byte* authIn, word32 authInSz);

int wc_AesXtsInit(XtsAes* aes, void* heap, int devId);


int wc_AesXtsSetKeyNoInit(XtsAes* aes, const byte* key,
         word32 len, int dir);


int wc_AesXtsSetKey(XtsAes* aes, const byte* key,
         word32 len, int dir, void* heap, int devId);
```

```
int wc_AesXtsEncryptSector(XtsAes* aes, byte* out,
        const byte* in, word32 sz, word64 sector);

int wc_AesXtsDecryptSector(XtsAes* aes, byte* out,
        const byte* in, word32 sz, word64 sector);

int wc_AesXtsEncrypt(XtsAes* aes, byte* out,
        const byte* in, word32 sz, const byte* i, word32 iSz);

int wc_AesXtsDecrypt(XtsAes* aes, byte* out,
       const byte* in, word32 sz, const byte* i, word32 iSz);

int wc_AesXtsFree(XtsAes* aes);


int  wc_AesInit(Aes* aes, void* heap, int devId);

int  wc_AesFree(Aes* aes);

int wc_AesCfbEncrypt(Aes* aes, byte* out, const byte* in, word32 sz);

int wc_AesCfbDecrypt(Aes* aes, byte* out, const byte* in, word32 sz);

int wc_AesSivEncrypt(const byte* key, word32 keySz, const byte* assoc,
                     word32 assocSz, const byte* nonce, word32 nonceSz,
                     const byte* in, word32 inSz, byte* siv, byte* out);

int wc_AesSivDecrypt(const byte* key, word32 keySz, const byte* assoc,
                     word32 assocSz, const byte* nonce, word32 nonceSz,
                     const byte* in, word32 inSz, byte* siv, byte* out);




WOLFSSL_API int  wc_AesEaxEncryptAuth(const byte* key, word32 keySz, byte* out,
                                      const byte* in, word32 inSz,
                                      const byte* nonce, word32 nonceSz,
                                      /* output computed auth tag */
                                      byte* authTag, word32 authTagSz,
                                      /* input data to authenticate */
                                      const byte* authIn, word32 authInSz);
WOLFSSL_API int  wc_AesEaxDecryptAuth(const byte* key, word32 keySz, byte* out,
                                      const byte* in, word32 inSz,
                                      const byte* nonce, word32 nonceSz,
                                      /* auth tag to verify against */
                                      const byte* authTag, word32 authTagSz,
                                      /* input data to authenticate */
                                      const byte* authIn, word32 authInSz);

WOLFSSL_API int  wc_AesEaxInit(AesEax* eax,
                               const byte* key, word32 keySz,
```

```
                                const byte* nonce, word32 nonceSz,
                                const byte* authIn, word32 authInSz);

WOLFSSL_API int  wc_AesEaxEncryptUpdate(AesEax* eax, byte* out,
                                        const byte* in, word32 inSz,
                                        const byte* authIn, word32 authInSz);

WOLFSSL_API int  wc_AesEaxDecryptUpdate(AesEax* eax, byte* out,
                                        const byte* in, word32 inSz,
                                        const byte* authIn, word32 authInSz);
WOLFSSL_API int  wc_AesEaxAuthDataUpdate(AesEax* eax,
                                       const byte* authIn, word32 authInSz);

WOLFSSL_API int wc_AesEaxEncryptFinal(AesEax* eax,
                                      byte* authTag, word32 authTagSz);

WOLFSSL_API int wc_AesEaxDecryptFinal(AesEax* eax,
                                      const byte* authIn, word32 authInSz);
WOLFSSL_API int wc_AesEaxFree(AesEax* eax);

int wc_AesCtsEncrypt(const byte* key, word32 keySz, byte* out,
                     const byte* in, word32 inSz,
                     const byte* iv);

int wc_AesCtsEncrypt(const byte* key, word32 keySz, byte* out,
                     const byte* in, word32 inSz,
                     const byte* iv);

int wc_AesCtsDecrypt(const byte* key, word32 keySz, byte* out,
                     const byte* in, word32 inSz,
                     const byte* iv);

int wc_AesCtsEncryptUpdate(Aes* aes, byte* out, word32* outSz,
                           const byte* in, word32 inSz);

int wc_AesCtsEncryptFinal(Aes* aes, byte* out, word32* outSz);

int wc_AesCtsDecryptUpdate(Aes* aes, byte* out, word32* outSz,
                           const byte* in, word32 inSz);

int wc_AesCtsDecryptFinal(Aes* aes, byte* out, word32* outSz);
```

C.2 dox_comments/header_files/arc4.h

C.2.1 Functions

	Name
int	**wc_Arc4Process**(Arc4 * arc4, byte * out, const byte * in, word32 length)This function encrypts an input message from the buffer in, placing the ciphertext in the output buffer out, or decrypts a ciphertext from the buffer in, placing the plaintext in the output buffer out, using ARC4 encryption. This function is used for both encryption and decryption. Before this method may be called, one must first initialize the ARC4 structure using wc_Arc4SetKey.
int	**wc_Arc4SetKey**(Arc4 * arc4, const byte * key, word32 length)This function sets the key for a ARC4 object, initializing it for use as a cipher. It should be called before using it for encryption with wc_Arc4Process.

C.2.2 Functions Documentation

C.2.2.1 function wc_Arc4Process

```
int wc_Arc4Process(
    Arc4 * arc4,
    byte * out,
    const byte * in,
    word32 length
)
```

This function encrypts an input message from the buffer in, placing the ciphertext in the output buffer out, or decrypts a ciphertext from the buffer in, placing the plaintext in the output buffer out, using ARC4 encryption. This function is used for both encryption and decryption. Before this method may be called, one must first initialize the ARC4 structure using wc_Arc4SetKey.

Parameters:

- **arc4** pointer to the ARC4 structure used to process the message
- **out** pointer to the output buffer in which to store the processed message
- **in** pointer to the input buffer containing the message to process
- **length** length of the message to process

See: wc_Arc4SetKey

Return: none

Example

```
Arc4 enc;
byte key[] = { key to use for encryption };
wc_Arc4SetKey(&enc, key, sizeof(key));

byte plain[] = { plain text to encode };
byte cipher[sizeof(plain)];
byte decrypted[sizeof(plain)];
// encrypt the plain into cipher
wc_Arc4Process(&enc, cipher, plain, sizeof(plain));
// decrypt the cipher
wc_Arc4Process(&enc, decrypted, cipher, sizeof(cipher));
```

C.2.2.2 function wc_Arc4SetKey

```
int wc_Arc4SetKey(
    Arc4 * arc4,
    const byte * key,
    word32 length
)
```

This function sets the key for a ARC4 object, initializing it for use as a cipher. It should be called before using it for encryption with wc_Arc4Process.

Parameters:

- **arc4** pointer to an arc4 structure to be used for encryption
- **key** key with which to initialize the arc4 structure
- **length** length of the key used to initialize the arc4 structure

See: wc_Arc4Process

Return: none

Example

```
Arc4 enc;
byte key[] = { initialize with key to use for encryption };
wc_Arc4SetKey(&enc, key, sizeof(key));
```

C.2.3 Source code

```
int wc_Arc4Process(Arc4* arc4, byte* out, const byte* in, word32 length);

int wc_Arc4SetKey(Arc4* arc4, const byte* key, word32 length);
```

C.3 dox_comments/header_files/asn.h

C.4 dox_comments/header_files/asn_public.h

C.4.1 Functions

	Name
int	**wc_InitCert**(Cert *)This function initializes a default cert, with the default options: version = 3 (0x2), serial = 0, sigType = SHA_WITH_RSA, issuer = blank, daysValid = 500, selfSigned = 1 (true) use subject as issuer, subject = blank.
Cert *	**wc_CertNew must be called.
void	**wc_CertFree.

	Name
int	**wc_MakeCert**(Cert * cert, byte * derBuffer, word32 derSz, RsaKey * rsaKey, ecc_key * eccKey, WC_RNG * rng)Used to make CA signed certs. Called after the subject information has been entered. This function makes an x509 Certificate v3 RSA or ECC from a cert input. It then writes this cert to derBuffer. It takes in either an rsaKey or an eccKey to generate the certificate. The certificate must be initialized with wc_InitCert before this method is called.
int	**wc_MakeCertReq will need to be called after this function to sign the certificate request. Please see the wolfCrypt test application (./wolfcrypt/test/test.c) for an example usage of this function.
int	**wc_SignCert if creating a CA signed cert.
int	**wc_MakeSelfCert**(Cert * cert, byte * derBuffer, word32 derSz, RsaKey * key, WC_RNG * rng)This function is a combination of the previous two functions, wc_MakeCert and wc_SignCert for self signing (the previous functions may be used for CA requests). It makes a certificate, and then signs it, generating a self-signed certificate.
int	**wc_SetIssuer**(Cert * cert, const char * issuerFile)This function sets the issuer for a certificate to the issuer in the provided pem issuerFile. It also changes the certificate's self-signed attribute to false. The issuer specified in issuerFile is verified prior to setting the cert issuer. This method is used to set fields prior to signing.
int	**wc_SetSubject**(Cert * cert, const char * subjectFile)This function sets the subject for a certificate to the subject in the provided pem subjectFile. This method is used to set fields prior to signing.
int	**wc_SetSubjectRaw**(Cert * cert, const byte * der, int derSz)This function sets the raw subject for a certificate from the subject in the provided der buffer. This method is used to set the raw subject field prior to signing.
int	**wc_GetSubjectRaw**(byte ** subjectRaw, Cert * cert)This function gets the raw subject from the certificate structure.
int	**wc_SetAltNames**(Cert * cert, const char * file)This function sets the alternate names for a certificate to the alternate names in the provided pem file. This is useful in the case that one wishes to secure multiple domains with the same certificate. This method is used to set fields prior to signing.

	Name
int	**wc_SetIssuerBuffer**(Cert * cert, const byte * der, int derSz)This function sets the issuer for a certificate from the issuer in the provided der buffer. It also changes the certificate's self-signed attribute to false. This method is used to set fields prior to signing.
int	**wc_SetIssuerRaw**(Cert * cert, const byte * der, int derSz)This function sets the raw issuer for a certificate from the issuer in the provided der buffer. This method is used to set the raw issuer field prior to signing.
int	**wc_SetSubjectBuffer**(Cert * cert, const byte * der, int derSz)This function sets the subject for a certificate from the subject in the provided der buffer. This method is used to set fields prior to signing.
int	**wc_SetAltNamesBuffer**(Cert * cert, const byte * der, int derSz)This function sets the alternate names for a certificate from the alternate names in the provided der buffer. This is useful in the case that one wishes to secure multiple domains with the same certificate. This method is used to set fields prior to signing.
int	**wc_SetDatesBuffer**(Cert * cert, const byte * der, int derSz)This function sets the dates for a certificate from the date range in the provided der buffer. This method is used to set fields prior to signing.
int	**wc_SetAuthKeyIdFromPublicKey**(Cert * cert, RsaKey * rsakey, ecc_key * eckey)Set AKID from either an RSA or ECC public key. note: Only set one of rsakey or eckey, not both.
int	**wc_SetAuthKeyIdFromCert**(Cert * cert, const byte * der, int derSz)Set AKID from from DER encoded certificate.
int	**wc_SetAuthKeyId**(Cert * cert, const char * file)Set AKID from certificate file in PEM format.
int	**wc_SetSubjectKeyIdFromPublicKey**(Cert * cert, RsaKey * rsakey, ecc_key * eckey)Set SKID from RSA or ECC public key.
int	**wc_SetSubjectKeyId**(Cert * cert, const char * file)Set SKID from public key file in PEM format. Both arguments are required.

	Name
int	**wc_SetKeyUsage**(Cert * cert, const char * value)This function allows you to set the key usage using a comma delimited string of tokens. Accepted tokens are: digitalSignature, nonRepudiation, contentCommitment, keyCertSign, cRLSign, dataEncipherment, keyAgreement, keyEncipherment, encipherOnly, decipherOnly. Example: “digitalSignature,nonRepudiation” nonRepudiation and contentCommitment are for the same usage.
int	**wc_PemPubKeyToDer**(const char * fileName, unsigned char * derBuf, int derSz)Loads a PEM key from a file and converts to a DER encoded buffer.
int	**wc_PubKeyPemToDer**(const unsigned char * pem, int pemSz, unsigned char * buff, int buffSz)Convert a PEM encoded public key to DER. Returns the number of bytes written to the buffer or a negative value for an error.
int	**wc_PemCertToDer**(const char * fileName, unsigned char * derBuf, int derSz)This function converts a pem certificate to a der certificate, and places the resulting certificate in the derBuf buffer provided.
int	**wc_DerToPem**(const byte * der, word32 derSz, byte * output, word32 outputSz, int type)This function converts a der formatted input certificate, contained in the der buffer, into a pem formatted output certificate, contained in the output buffer. It should be noted that this is not an in place conversion, and a separate buffer must be utilized to store the pem formatted output.
int	**wc_DerToPemEx**(const byte * der, word32 derSz, byte * output, word32 outputSz, byte * cipherIno, int type)This function converts a der formatted input certificate, contained in the der buffer, into a pem formatted output certificate, contained in the output buffer. It should be noted that this is not an in place conversion, and a separate buffer must be utilized to store the pem formatted output. Allows setting cipher info.
int	**wc_KeyPemToDer**(const unsigned char * pem, int pemSz, unsigned char * buff, int buffSz, const char * pass)Converts a key in PEM format to DER format.

	Name
int	**wc_CertPemToDer**(const unsigned char * pem, int pemSz, unsigned char * buff, int buffSz, int type)This function converts a PEM formatted certificate to DER format. Calls OpenSSL function PemToDer.
int	**wc_GetPubKeyDerFromCert.
int	**wc_EccPrivateKeyDecode**(const byte * input, word32 * inOutIdx, ecc_key * key, word32 inSz)This function reads in an ECC private key from the input buffer, input, parses the private key, and uses it to generate an ecc_key object, which it stores in key.
int	**wc_EccKeyToDer**(ecc_key * key, byte * output, word32 inLen)This function writes a private ECC key to der format.
int	**wc_EccPublicKeyDecode**(const byte * input, word32 * inOutIdx, ecc_key * key, word32 inSz)Decodes an ECC public key from an input buffer. It will parse an ASN sequence to retrieve the ECC key.
int	**wc_EccPublicKeyToDer**(ecc_key * key, byte * output, word32 inLen, int with_AlgCurve)This function converts the ECC public key to DER format. It returns the size of buffer used. The public ECC key in DER format is stored in output buffer. The with_AlgCurve flag will include a header that has the Algorithm and Curve information.
int	**wc_EccPublicKeyToDer_ex**(ecc_key * key, byte * output, word32 inLen, int with_AlgCurve, int comp)This function converts the ECC public key to DER format. It returns the size of buffer used. The public ECC key in DER format is stored in output buffer. The with_AlgCurve flag will include a header that has the Algorithm and Curve information. The comp parameter determines if the public key will be exported as compressed.
int	**wc_Curve25519PrivateKeyDecode**(const byte * input, word32 * inOutIdx, curve25519_key * key, word32 inSz)This function decodes a Curve25519 private key (only) from a DER encoded buffer.
int	**wc_Curve25519PublicKeyDecode**(const byte * input, word32 * inOutIdx, curve25519_key * key, word32 inSz)This function decodes a Curve25519 public key (only) from a DER encoded buffer.

	Name
int	**wc_Curve25519KeyDecode**(const byte * input, word32 * inOutIdx, curve25519_key * key, word32 inSz)This function decodes a Curve25519 key from a DER encoded buffer. It can decode either a private key, a public key, or both.
int	**wc_Curve25519PrivateKeyToDer**(curve25519_key * key, byte * output, word32 inLen)This function encodes a Curve25519 private key to DER format. If the input key structure contains a public key, it will be ignored.
int	**wc_Curve25519PublicKeyToDer**(curve25519_key * key, byte * output, word32 inLen, int withAlg)This function encodes a Curve25519 public key to DER format. If the input key structure contains a private key, it will be ignored.
int	**wc_Curve25519KeyToDer**(curve25519_key * key, byte * output, word32 inLen, int withAlg)This function encodes a Curve25519 key to DER format. It can encode either a private key, a public key, or both.
word32	**wc_EncodeSignature**(byte * out, const byte * digest, word32 digSz, int hashOID)This function encodes a digital signature into the output buffer, and returns the size of the encoded signature created.
int	**wc_GetCTC_HashOID**(int type)This function returns the hash OID that corresponds to a hashing type. For example, when given the type: WC_SHA512, this function returns the identifier corresponding to a SHA512 hash, SHA512h.
void	**wc_SetCert_Free**(Cert * cert)This function cleans up memory and resources used by the certificate structure's decoded cert cache. When WOLFSSL_CERT_GEN_CACHE is defined the decoded cert structure is cached in the certificate structure. This allows subsequent calls to certificate set functions to avoid parsing the decoded cert on each call.
int	**wc_GetPkcs8TraditionalOffset**(byte * input, word32 * inOutIdx, word32 sz)This function finds the beginning of the traditional private key inside a PKCS#8 unencrypted buffer.
int	**wc_CreatePKCS8Key**(byte * out, word32 * outSz, byte * key, word32 keySz, int algoID, const byte * curveOID, word32 oidSz)This function takes in a DER private key and converts it to PKCS#8 format. Also used in creating PKCS#12 shrouded key bags. See RFC 5208.

	Name
int	**wc_EncryptPKCS8Key**(byte * key, word32 keySz, byte * out, word32 * outSz, const char * password, int passwordSz, int vPKCS, int pbeOid, int encAlgId, byte * salt, word32 saltSz, int itt, WC_RNG * rng, void * heap)This function takes in an unencrypted PKCS#8 DER key (e.g. one created by wc_CreatePKCS8Key) and converts it to PKCS#8 encrypted format. The resulting encrypted key can be decrypted using wc_DecryptPKCS8Key. See RFC 5208.
int	**wc_DecryptPKCS8Key**(byte * input, word32 sz, const char * password, int passwordSz)This function takes an encrypted PKCS#8 DER key and decrypts it to PKCS#8 unencrypted DER. Undoes the encryption done by wc_EncryptPKCS8Key. See RFC5208. The input buffer is overwritten with the decrypted data.
int	**wc_CreateEncryptedPKCS8Key**(byte * key, word32 keySz, byte * out, word32 * outSz, const char * password, int passwordSz, int vPKCS, int pbeOid, int encAlgId, byte * salt, word32 saltSz, int itt, WC_RNG * rng, void * heap)This function takes a traditional, DER key, converts it to PKCS#8 format, and encrypts it. It uses wc_CreatePKCS8Key and wc_EncryptPKCS8Key to do this.
void	**wc_InitDecodedCert**(struct DecodedCert * cert, const byte * source, word32 inSz, void * heap)This function initializes the DecodedCert pointed to by the “cert” parameter. It saves the “source” pointer to a DER-encoded certificate of length “inSz.” This certificate can be parsed by a subsequent call to wc_ParseCert.
int	**wc_ParseCert**(DecodedCert * cert, int type, int verify, void * cm)This function parses the DER-encoded certificate saved in the DecodedCert object and populates the fields of that object. The DecodedCert must have been initialized with a prior call to wc_InitDecodedCert. This function takes an optional pointer to a CertificateManager object, which is used to populate the certificate authority information of the DecodedCert, if the CA is found in the CertificateManager.
void	**wc_FreeDecodedCert**(struct DecodedCert * cert)This function frees a DecodedCert that was previously initialized with wc_InitDecodedCert.

	Name
int	**wc_SetTimeCb**(wc_time_cb f)This function registers a time callback that will be used anytime wolfSSL needs to get the current time. The prototype of the callback should be the same as the “time” function from the C standard library.
time_t	**wc_Time**(time_t * t)This function gets the current time. By default, it uses the XTIME macro, which varies between platforms. The user can use a function of their choosing instead via the wc_SetTimeCb function.
int	**wc_SetCustomExtension**(Cert * cert, int critical, const char * oid, const byte * der, word32 derSz)This function injects a custom extension in to an X.509 certificate. note: The content at the address pointed to by any of the parameters that are pointers must not be modified until the certificate is generated and you have the der output. This function does NOT copy the contents to another buffer.
int	**wc_SetUnknownExtCallback**(DecodedCert * cert, wc_UnknownExtCallback cb)This function registers a callback that will be used anytime wolfSSL encounters an unknown X.509 extension in a certificate while parsing a certificate. The prototype of the callback should be:
int	**wc_CheckCertSigPubKey**(const byte * cert, word32 certSz, void * heap, const byte * pubKey, word32 pubKeySz, int pubKeyOID)This function verifies the signature in the der form of an X.509 certificate against a public key. The public key is expected to be the full subject public key info in der form.
int	**wc_Asn1PrintOptions_Init**(Asn1PrintOptions * opts)This function initializes the ASN.1 print options.
int	**wc_Asn1PrintOptions_Set**(Asn1PrintOptions * opts, enum Asn1PrintOpt opt, word32 val)This function sets a print option into an ASN.1 print options object.
int	**wc_Asn1_Init**(Asn1 * asn1)This function initializes an ASN.1 parsing object.
int	**wc_Asn1_SetFile**(Asn1 * asn1, XFILE file)This function sets the file to use when printing into an ASN.1 parsing object.
int	**wc_Asn1_PrintAll**(Asn1 * asn1, Asn1PrintOptions * opts, unsigned char * data, word32 len)Print all ASN.1 items.

C.4.2 Functions Documentation

C.4.2.1 function wc_InitCert

```
int wc_InitCert(
    Cert *
)
```

This function initializes a default cert, with the default options: version = 3 (0x2), serial = 0, sigType = SHA_WITH_RSA, issuer = blank, daysValid = 500, selfSigned = 1 (true) use subject as issuer, subject = blank.

Parameters:

- **cert** pointer to an uninitialized cert structure to initialize

See:

- wc_MakeCert
- wc_MakeCertReq

Return: none No returns.

Example

```
Cert myCert;
wc_InitCert(&myCert);
```

C.4.2.2 function wc_CertNew

```
Cert * wc_CertNew(
    void * heap
)
```

This function allocates a new Cert structure for use during cert operations without the application having to allocate the structure itself. The Cert structure is also initialized by this function thus removing the need to call wc_InitCert() must be called.

Parameters:

- **A** pointer to the heap used for dynamic allocation. Can be NULL.

See:

- wc_InitCert
- wc_MakeCert
- wc_CertFree

Return:

- pointer If successful the call will return a pointer to the newly allocated and initialized Cert.
- NULL On a memory allocation failure.

Example

```
Cert*   myCert;

myCert = wc_CertNew(NULL);
if (myCert == NULL) {
    // Cert creation failure
}
```

C.4.2.3 function wc_CertFree

```
void wc_CertFree(
    Cert * cert
)
```

This function frees the memory allocated for a cert structure by a previous call to wc_CertNew().

Parameters:

- **A** pointer to the cert structure to free.

See:

- wc_InitCert
- wc_MakeCert
- wc_CertNew

Return: None.

Example

```
Cert*   myCert;

myCert = wc_CertNew(NULL);

// Perform cert operations.

wc_CertFree(myCert);
```

C.4.2.4 function wc_MakeCert

```
int wc_MakeCert(
    Cert * cert,
    byte * derBuffer,
    word32 derSz,
    RsaKey * rsaKey,
    ecc_key * eccKey,
    WC_RNG * rng
)
```

Used to make CA signed certs. Called after the subject information has been entered. This function makes an x509 Certificate v3 RSA or ECC from a cert input. It then writes this cert to derBuffer. It takes in either an rsaKey or an eccKey to generate the certificate. The certificate must be initialized with wc_InitCert before this method is called.

Parameters:

- **cert** pointer to an initialized cert structure
- **derBuffer** pointer to the buffer in which to hold the generated cert
- **derSz** size of the buffer in which to store the cert
- **rsaKey** pointer to an RsaKey structure containing the rsa key used to generate the certificate
- **eccKey** pointer to an EccKey structure containing the ecc key used to generate the certificate
- **rng** pointer to the random number generator used to make the cert

See:

- wc_InitCert
- wc_MakeCertReq

Return:

- Success On successfully making an x509 certificate from the specified input cert, returns the size of the cert generated.
- MEMORY_E Returned if there is an error allocating memory with XMALLOC
- BUFFER_E Returned if the provided derBuffer is too small to store the generated certificate
- Others Additional error messages may be returned if the cert generation is not successful.

Example

```
Cert myCert;
wc_InitCert(&myCert);
WC_RNG rng;
//initialize rng;
RsaKey key;
//initialize key;
byte * derCert = malloc(FOURK_BUF);
word32 certSz;
certSz = wc_MakeCert(&myCert, derCert, FOURK_BUF, &key, NULL, &rng);
```

C.4.2.5 function wc_MakeCertReq

```
int wc_MakeCertReq(
    Cert * cert,
    byte * derBuffer,
    word32 derSz,
    RsaKey * rsaKey,
    ecc_key * eccKey
)
```

This function makes a certificate signing request using the input certificate and writes the output to derBuffer. It takes in either an rsaKey or an eccKey to generate the certificate request. wc_SignCert() will need to be called after this function to sign the certificate request. Please see the wolfCrypt test application (./wolfcrypt/test/test.c) for an example usage of this function.

Parameters:

- **cert** pointer to an initialized cert structure
- **derBuffer** pointer to the buffer in which to hold the generated certificate request
- **derSz** size of the buffer in which to store the certificate request
- **rsaKey** pointer to an RsaKey structure containing the rsa key used to generate the certificate request
- **eccKey** pointer to an EccKey structure containing the ecc key used to generate the certificate request

See:

- wc_InitCert
- wc_MakeCert

Return:

- Success On successfully making an X.509 certificate request from the specified input cert, returns the size of the certificate request generated.
- MEMORY_E Returned if there is an error allocating memory with XMALLOC
- BUFFER_E Returned if the provided derBuffer is too small to store the generated certificate
- Other Additional error messages may be returned if the certificate request generation is not successful.

Example

```
Cert myCert;
// initialize myCert
EccKey key;
//initialize key;
byte* derCert = (byte*)malloc(FOURK_BUF);

word32 certSz;
certSz = wc_MakeCertReq(&myCert, derCert, FOURK_BUF, NULL, &key);
```

C.4.2.6 function wc_SignCert

```
int wc_SignCert(
    int requestSz,
    int sigType,
    byte * derBuffer,
    word32 derSz,
    RsaKey * rsaKey,
    ecc_key * eccKey,
    WC_RNG * rng
)
```

This function signs buffer and adds the signature to the end of buffer. It takes in a signature type. Must be called after wc_MakeCert() if creating a CA signed cert.

Parameters:

- **requestSz** the size of the certificate body we're requesting to have signed
- **sType** Type of signature to create. Valid options are: CTC_MD5wRSA, CTC_SHAwRSA, CTC_SHAwECDSA, CTC_SHA256wECDSA, and CTC_SHA256wRSA
- **buffer** pointer to the buffer containing the certificate to be signed. On success: will hold the newly signed certificate
- **buffSz** the (total) size of the buffer in which to store the newly signed certificate
- **rsaKey** pointer to an RsaKey structure containing the rsa key to used to sign the certificate
- **eccKey** pointer to an EccKey structure containing the ecc key to used to sign the certificate
- **rng** pointer to the random number generator used to sign the certificate

See:

- wc_InitCert
- wc_MakeCert

Return:

- Success On successfully signing the certificate, returns the new size of the cert (including signature).
- MEMORY_E Returned if there is an error allocating memory with XMALLOC
- BUFFER_E Returned if the provided buffer is too small to store the generated certificate
- Other Additional error messages may be returned if the cert generation is not successful.

Example

```
Cert myCert;
byte* derCert = (byte*)malloc(FOURK_BUF);
// initialize myCert, derCert
RsaKey key;
// initialize key;
WC_RNG rng;
// initialize rng
```

```
word32 certSz;
certSz = wc_SignCert(myCert.bodySz, myCert.sigType,derCert,FOURK_BUF,
&key, NULL,
&rng);
```

C.4.2.7 function wc_MakeSelfCert

```
int wc_MakeSelfCert(
    Cert * cert,
    byte * derBuffer,
    word32 derSz,
    RsaKey * key,
    WC_RNG * rng
)
```

This function is a combination of the previous two functions, wc_MakeCert and wc_SignCert for self signing (the previous functions may be used for CA requests). It makes a certificate, and then signs it, generating a self-signed certificate.

Parameters:

- **cert** pointer to the cert to make and sign
- **buffer** pointer to the buffer in which to hold the signed certificate
- **buffSz** size of the buffer in which to store the signed certificate
- **key** pointer to an RsaKey structure containing the rsa key to used to sign the certificate
- **rng** pointer to the random number generator used to generate and sign the certificate

See:

- wc_InitCert
- wc_MakeCert
- wc_SignCert

Return:

- Success On successfully signing the certificate, returns the new size of the cert.
- MEMORY_E Returned if there is an error allocating memory with XMALLOC
- BUFFER_E Returned if the provided buffer is too small to store the generated certificate
- Other Additional error messages may be returned if the cert generation is not successful.

Example

```
Cert myCert;
byte* derCert = (byte*)malloc(FOURK_BUF);
// initialize myCert, derCert
RsaKey key;
// initialize key;
WC_RNG rng;
// initialize rng

word32 certSz;
certSz = wc_MakeSelfCert(&myCert, derCert, FOURK_BUF, &key, NULL, &rng);
```

C.4.2.8 function wc_SetIssuer

```
int wc_SetIssuer(
    Cert * cert,
    const char * issuerFile
)
```

This function sets the issuer for a certificate to the issuer in the provided pem issuerFile. It also changes the certificate's self-signed attribute to false. The issuer specified in issuerFile is verified prior to setting the cert issuer. This method is used to set fields prior to signing.

Parameters:

- **cert** pointer to the cert for which to set the issuer
- **issuerFile** path of the file containing the pem formatted certificate

See:

- wc_InitCert
- wc_SetSubject
- wc_SetIssuerBuffer

Return:

- 0 Returned on successfully setting the issuer for the certificate
- MEMORY_E Returned if there is an error allocating memory with XMALLOC
- ASN_PARSE_E Returned if there is an error parsing the cert header file
- ASN_OBJECT_ID_E Returned if there is an error parsing the encryption type from the cert
- ASN_EXPECT_0_E Returned if there is a formatting error in the encryption specification of the cert file
- ASN_BEFORE_DATE_E Returned if the date is before the certificate start date
- ASN_AFTER_DATE_E Returned if the date is after the certificate expiration date
- ASN_BITSTR_E Returned if there is an error parsing a bit string from the certificate
- ECC_CURVE_OID_E Returned if there is an error parsing the ECC key from the certificate
- ASN_UNKNOWN_OID_E Returned if the certificate is using an unknown key object id
- ASN_VERSION_E Returned if the ALLOW_V1_EXTENSIONS option is not defined and the certificate is a V1 or V2 certificate
- BAD_FUNC_ARG Returned if there is an error processing the certificate extension
- ASN_CRIT_EXT_E Returned if an unfamiliar critical extension is encountered in processing the certificate
- ASN_SIG_OID_E Returned if the signature encryption type is not the same as the encryption type of the certificate in the provided file
- ASN_SIG_CONFIRM_E Returned if confirming the certification signature fails
- ASN_NAME_INVALID_E Returned if the certificate's name is not permitted by the CA name constraints
- ASN_NO_SIGNER_E Returned if there is no CA signer to verify the certificate's authenticity

Example

```
Cert myCert;
// initialize myCert
if(wc_SetIssuer(&myCert, "./path/to/ca-cert.pem") != 0) {
    // error setting issuer
}
```

C.4.2.9 function wc_SetSubject

```
int wc_SetSubject(
    Cert * cert,
    const char * subjectFile
)
```

This function sets the subject for a certificate to the subject in the provided pem subjectFile. This method is used to set fields prior to signing.

Parameters:

- **cert** pointer to the cert for which to set the issuer
- **subjectFile** path of the file containing the pem formatted certificate

See:

- wc_InitCert
- wc_SetIssuer

Return:

- 0 Returned on successfully setting the issuer for the certificate
- MEMORY_E Returned if there is an error allocating memory with XMALLOC
- ASN_PARSE_E Returned if there is an error parsing the cert header file
- ASN_OBJECT_ID_E Returned if there is an error parsing the encryption type from the cert
- ASN_EXPECT_0_E Returned if there is a formatting error in the encryption specification of the cert file
- ASN_BEFORE_DATE_E Returned if the date is before the certificate start date
- ASN_AFTER_DATE_E Returned if the date is after the certificate expiration date
- ASN_BITSTR_E Returned if there is an error parsing a bit string from the certificate
- ECC_CURVE_OID_E Returned if there is an error parsing the ECC key from the certificate
- ASN_UNKNOWN_OID_E Returned if the certificate is using an unknown key object id
- ASN_VERSION_E Returned if the ALLOW_V1_EXTENSIONS option is not defined and the certificate is a V1 or V2 certificate
- BAD_FUNC_ARG Returned if there is an error processing the certificate extension
- ASN_CRIT_EXT_E Returned if an unfamiliar critical extension is encountered in processing the certificate
- ASN_SIG_OID_E Returned if the signature encryption type is not the same as the encryption type of the certificate in the provided file
- ASN_SIG_CONFIRM_E Returned if confirming the certification signature fails
- ASN_NAME_INVALID_E Returned if the certificate's name is not permitted by the CA name constraints
- ASN_NO_SIGNER_E Returned if there is no CA signer to verify the certificate's authenticity

Example

```
Cert myCert;
// initialize myCert
if(wc_SetSubject(&myCert, "./path/to/ca-cert.pem") != 0) {
    // error setting subject
}
```

C.4.2.10 function wc_SetSubjectRaw

```
int wc_SetSubjectRaw(
    Cert * cert,
    const byte * der,
    int derSz
)
```

This function sets the raw subject for a certificate from the subject in the provided der buffer. This method is used to set the raw subject field prior to signing.

Parameters:

- **cert** pointer to the cert for which to set the raw subject
- **der** pointer to the buffer containing the der formatted certificate from which to grab the subject
- **derSz** size of the buffer containing the der formatted certificate from which to grab the subject

See:

- wc_InitCert
- wc_SetSubject

Return:

- 0 Returned on successfully setting the subject for the certificate
- MEMORY_E Returned if there is an error allocating memory with XMALLOC
- ASN_PARSE_E Returned if there is an error parsing the cert header file
- ASN_OBJECT_ID_E Returned if there is an error parsing the encryption type from the cert
- ASN_EXPECT_0_E Returned if there is a formatting error in the encryption specification of the cert file
- ASN_BEFORE_DATE_E Returned if the date is before the certificate start date
- ASN_AFTER_DATE_E Returned if the date is after the certificate expiration date
- ASN_BITSTR_E Returned if there is an error parsing a bit string from the certificate
- ECC_CURVE_OID_E Returned if there is an error parsing the ECC key from the certificate
- ASN_UNKNOWN_OID_E Returned if the certificate is using an unknown key object id
- ASN_VERSION_E Returned if the ALLOW_V1_EXTENSIONS option is not defined and the certificate is a V1 or V2 certificate
- BAD_FUNC_ARG Returned if there is an error processing the certificate extension
- ASN_CRIT_EXT_E Returned if an unfamiliar critical extension is encountered in processing the certificate
- ASN_SIG_OID_E Returned if the signature encryption type is not the same as the encryption type of the certificate in the provided file
- ASN_SIG_CONFIRM_E Returned if confirming the certification signature fails
- ASN_NAME_INVALID_E Returned if the certificate's name is not permitted by the CA name constraints
- ASN_NO_SIGNER_E Returned if there is no CA signer to verify the certificate's authenticity

Example

```
Cert myCert;
// initialize myCert
byte* der;
der = (byte*)malloc(FOURK_BUF);
// initialize der
if(wc_SetSubjectRaw(&myCert, der, FOURK_BUF) != 0) {
    // error setting subject
}
```

C.4.2.11 function wc_GetSubjectRaw

```
int wc_GetSubjectRaw(
    byte ** subjectRaw,
    Cert * cert
)
```

This function gets the raw subject from the certificate structure.

Parameters:

- **subjectRaw** pointer-pointer to the raw subject upon successful return
- **cert** pointer to the cert from which to get the raw subject

See:

- wc_InitCert
- wc_SetSubjectRaw

Return:

- 0 Returned on successfully getting the subject from the certificate
- BAD_FUNC_ARG Returned if there is an error processing the certificate extension

Example

```
Cert myCert;
byte *subjRaw;
// initialize myCert

if(wc_GetSubjectRaw(&subjRaw, &myCert) != 0) {
    // error setting subject
}
```

C.4.2.12 function wc_SetAltNames

```
int wc_SetAltNames(
    Cert * cert,
    const char * file
)
```

This function sets the alternate names for a certificate to the alternate names in the provided pem file. This is useful in the case that one wishes to secure multiple domains with the same certificate. This method is used to set fields prior to signing.

Parameters:

- **cert** pointer to the cert for which to set the alt names
- **file** path of the file containing the pem formatted certificate

See:

- wc_InitCert
- wc_SetIssuer

Return:

- 0 Returned on successfully setting the alt names for the certificate
- MEMORY_E Returned if there is an error allocating memory with XMALLOC
- ASN_PARSE_E Returned if there is an error parsing the cert header file
- ASN_OBJECT_ID_E Returned if there is an error parsing the encryption type from the cert
- ASN_EXPECT_0_E Returned if there is a formatting error in the encryption specification of the cert file
- ASN_BEFORE_DATE_E Returned if the date is before the certificate start date
- ASN_AFTER_DATE_E Returned if the date is after the certificate expiration date
- ASN_BITSTR_E Returned if there is an error parsing a bit string from the certificate
- ECC_CURVE_OID_E Returned if there is an error parsing the ECC key from the certificate
- ASN_UNKNOWN_OID_E Returned if the certificate is using an unknown key object id
- ASN_VERSION_E Returned if the ALLOW_V1_EXTENSIONS option is not defined and the certificate is a V1 or V2 certificate
- BAD_FUNC_ARG Returned if there is an error processing the certificate extension
- ASN_CRIT_EXT_E Returned if an unfamiliar critical extension is encountered in processing the certificate
- ASN_SIG_OID_E Returned if the signature encryption type is not the same as the encryption type of the certificate in the provided file
- ASN_SIG_CONFIRM_E Returned if confirming the certification signature fails
- ASN_NAME_INVALID_E Returned if the certificate's name is not permitted by the CA name constraints
- ASN_NO_SIGNER_E Returned if there is no CA signer to verify the certificate's authenticity

Example

```
Cert myCert;
// initialize myCert
if(wc_SetSubject(&myCert, "./path/to/ca-cert.pem") != 0) {
    // error setting alt names
}
```

C.4.2.13 function wc_SetIssuerBuffer

```
int wc_SetIssuerBuffer(
    Cert * cert,
    const byte * der,
    int derSz
)
```

This function sets the issuer for a certificate from the issuer in the provided der buffer. It also changes the certificate's self-signed attribute to false. This method is used to set fields prior to signing.

Parameters:

- **cert** pointer to the cert for which to set the issuer
- **der** pointer to the buffer containing the der formatted certificate from which to grab the issuer
- **derSz** size of the buffer containing the der formatted certificate from which to grab the issuer

See:

- wc_InitCert
- wc_SetIssuer

Return:

- 0 Returned on successfully setting the issuer for the certificate
- MEMORY_E Returned if there is an error allocating memory with XMALLOC
- ASN_PARSE_E Returned if there is an error parsing the cert header file
- ASN_OBJECT_ID_E Returned if there is an error parsing the encryption type from the cert
- ASN_EXPECT_0_E Returned if there is a formatting error in the encryption specification of the cert file
- ASN_BEFORE_DATE_E Returned if the date is before the certificate start date
- ASN_AFTER_DATE_E Returned if the date is after the certificate expiration date
- ASN_BITSTR_E Returned if there is an error parsing a bit string from the certificate
- ECC_CURVE_OID_E Returned if there is an error parsing the ECC key from the certificate
- ASN_UNKNOWN_OID_E Returned if the certificate is using an unknown key object id
- ASN_VERSION_E Returned if the ALLOW_V1_EXTENSIONS option is not defined and the certificate is a V1 or V2 certificate
- BAD_FUNC_ARG Returned if there is an error processing the certificate extension
- ASN_CRIT_EXT_E Returned if an unfamiliar critical extension is encountered in processing the certificate
- ASN_SIG_OID_E Returned if the signature encryption type is not the same as the encryption type of the certificate in the provided file
- ASN_SIG_CONFIRM_E Returned if confirming the certification signature fails
- ASN_NAME_INVALID_E Returned if the certificate's name is not permitted by the CA name constraints
- ASN_NO_SIGNER_E Returned if there is no CA signer to verify the certificate's authenticity

Example

```
Cert myCert;
// initialize myCert
```

```
byte* der;
der = (byte*)malloc(FOURK_BUF);
// initialize der
if(wc_SetIssuerBuffer(&myCert, der, FOURK_BUF) != 0) {
    // error setting issuer
}
```

C.4.2.14 function wc_SetIssuerRaw

```
int wc_SetIssuerRaw(
    Cert * cert,
    const byte * der,
    int derSz
)
```

This function sets the raw issuer for a certificate from the issuer in the provided der buffer. This method is used to set the raw issuer field prior to signing.

Parameters:

- **cert** pointer to the cert for which to set the raw issuer
- **der** pointer to the buffer containing the der formatted certificate from which to grab the subject
- **derSz** size of the buffer containing the der formatted certificate from which to grab the subject

See:

- wc_InitCert
- wc_SetIssuer

Return:

- 0 Returned on successfully setting the issuer for the certificate
- MEMORY_E Returned if there is an error allocating memory with XMALLOC
- ASN_PARSE_E Returned if there is an error parsing the cert header file
- ASN_OBJECT_ID_E Returned if there is an error parsing the encryption type from the cert
- ASN_EXPECT_0_E Returned if there is a formatting error in the encryption specification of the cert file
- ASN_BEFORE_DATE_E Returned if the date is before the certificate start date
- ASN_AFTER_DATE_E Returned if the date is after the certificate expiration date
- ASN_BITSTR_E Returned if there is an error parsing a bit string from the certificate
- ECC_CURVE_OID_E Returned if there is an error parsing the ECC key from the certificate
- ASN_UNKNOWN_OID_E Returned if the certificate is using an unknown key object id
- ASN_VERSION_E Returned if the ALLOW_V1_EXTENSIONS option is not defined and the certificate is a V1 or V2 certificate
- BAD_FUNC_ARG Returned if there is an error processing the certificate extension
- ASN_CRIT_EXT_E Returned if an unfamiliar critical extension is encountered in processing the certificate
- ASN_SIG_OID_E Returned if the signature encryption type is not the same as the encryption type of the certificate in the provided file
- ASN_SIG_CONFIRM_E Returned if confirming the certification signature fails
- ASN_NAME_INVALID_E Returned if the certificate's name is not permitted by the CA name constraints
- ASN_NO_SIGNER_E Returned if there is no CA signer to verify the certificate's authenticity

Example

```
Cert myCert;
// initialize myCert
byte* der;
```

```
der = (byte*)malloc(FOURK_BUF);
// initialize der
if(wc_SetIssuerRaw(&myCert, der, FOURK_BUF) != 0) {
    // error setting subject
}
```

C.4.2.15 function wc_SetSubjectBuffer

```
int wc_SetSubjectBuffer(
    Cert * cert,
    const byte * der,
    int derSz
)
```

This function sets the subject for a certificate from the subject in the provided der buffer. This method is used to set fields prior to signing.

Parameters:

- **cert** pointer to the cert for which to set the subject
- **der** pointer to the buffer containing the der formatted certificate from which to grab the subject
- **derSz** size of the buffer containing the der formatted certificate from which to grab the subject

See:

- wc_InitCert
- wc_SetSubject

Return:

- 0 Returned on successfully setting the subject for the certificate
- MEMORY_E Returned if there is an error allocating memory with XMALLOC
- ASN_PARSE_E Returned if there is an error parsing the cert header file
- ASN_OBJECT_ID_E Returned if there is an error parsing the encryption type from the cert
- ASN_EXPECT_0_E Returned if there is a formatting error in the encryption specification of the cert file
- ASN_BEFORE_DATE_E Returned if the date is before the certificate start date
- ASN_AFTER_DATE_E Returned if the date is after the certificate expiration date
- ASN_BITSTR_E Returned if there is an error parsing a bit string from the certificate
- ECC_CURVE_OID_E Returned if there is an error parsing the ECC key from the certificate
- ASN_UNKNOWN_OID_E Returned if the certificate is using an unknown key object id
- ASN_VERSION_E Returned if the ALLOW_V1_EXTENSIONS option is not defined and the certificate is a V1 or V2 certificate
- BAD_FUNC_ARG Returned if there is an error processing the certificate extension
- ASN_CRIT_EXT_E Returned if an unfamiliar critical extension is encountered in processing the certificate
- ASN_SIG_OID_E Returned if the signature encryption type is not the same as the encryption type of the certificate in the provided file
- ASN_SIG_CONFIRM_E Returned if confirming the certification signature fails
- ASN_NAME_INVALID_E Returned if the certificate's name is not permitted by the CA name constraints
- ASN_NO_SIGNER_E Returned if there is no CA signer to verify the certificate's authenticity

Example

```
Cert myCert;
// initialize myCert
byte* der;
der = (byte*)malloc(FOURK_BUF);
```

```
// initialize der
if(wc_SetSubjectBuffer(&myCert, der, FOURK_BUF) != 0) {
    // error setting subject
}
```

C.4.2.16 function wc_SetAltNamesBuffer

```
int wc_SetAltNamesBuffer(
    Cert * cert,
    const byte * der,
    int derSz
)
```

This function sets the alternate names for a certificate from the alternate names in the provided der buffer. This is useful in the case that one wishes to secure multiple domains with the same certificate. This method is used to set fields prior to signing.

Parameters:

- **cert** pointer to the cert for which to set the alternate names
- **der** pointer to the buffer containing the der formatted certificate from which to grab the alternate names
- **derSz** size of the buffer containing the der formatted certificate from which to grab the alternate names

See:

- wc_InitCert
- wc_SetAltNames

Return:

- 0 Returned on successfully setting the alternate names for the certificate
- MEMORY_E Returned if there is an error allocating memory with XMALLOC
- ASN_PARSE_E Returned if there is an error parsing the cert header file
- ASN_OBJECT_ID_E Returned if there is an error parsing the encryption type from the cert
- ASN_EXPECT_0_E Returned if there is a formatting error in the encryption specification of the cert file
- ASN_BEFORE_DATE_E Returned if the date is before the certificate start date
- ASN_AFTER_DATE_E Returned if the date is after the certificate expiration date
- ASN_BITSTR_E Returned if there is an error parsing a bit string from the certificate
- ECC_CURVE_OID_E Returned if there is an error parsing the ECC key from the certificate
- ASN_UNKNOWN_OID_E Returned if the certificate is using an unknown key object id
- ASN_VERSION_E Returned if the ALLOW_V1_EXTENSIONS option is not defined and the certificate is a V1 or V2 certificate
- BAD_FUNC_ARG Returned if there is an error processing the certificate extension
- ASN_CRIT_EXT_E Returned if an unfamiliar critical extension is encountered in processing the certificate
- ASN_SIG_OID_E Returned if the signature encryption type is not the same as the encryption type of the certificate in the provided file
- ASN_SIG_CONFIRM_E Returned if confirming the certification signature fails
- ASN_NAME_INVALID_E Returned if the certificate's name is not permitted by the CA name constraints
- ASN_NO_SIGNER_E Returned if there is no CA signer to verify the certificate's authenticity

Example

```
Cert myCert;
// initialize myCert
```

```
byte* der;
der = (byte*)malloc(FOURK_BUF);
// initialize der
if(wc_SetAltNamesBuffer(&myCert, der, FOURK_BUF) != 0) {
    // error setting subject
}
```

C.4.2.17 function wc_SetDatesBuffer

```
int wc_SetDatesBuffer(
    Cert * cert,
    const byte * der,
    int derSz
)
```

This function sets the dates for a certificate from the date range in the provided der buffer. This method is used to set fields prior to signing.

Parameters:

- **cert** pointer to the cert for which to set the dates
- **der** pointer to the buffer containing the der formatted certificate from which to grab the date range
- **derSz** size of the buffer containing the der formatted certificate from which to grab the date range

See: wc_InitCert

Return:

- 0 Returned on successfully setting the dates for the certificate
- MEMORY_E Returned if there is an error allocating memory with XMALLOC
- ASN_PARSE_E Returned if there is an error parsing the cert header file
- ASN_OBJECT_ID_E Returned if there is an error parsing the encryption type from the cert
- ASN_EXPECT_0_E Returned if there is a formatting error in the encryption specification of the cert file
- ASN_BEFORE_DATE_E Returned if the date is before the certificate start date
- ASN_AFTER_DATE_E Returned if the date is after the certificate expiration date
- ASN_BITSTR_E Returned if there is an error parsing a bit string from the certificate
- ECC_CURVE_OID_E Returned if there is an error parsing the ECC key from the certificate
- ASN_UNKNOWN_OID_E Returned if the certificate is using an unknown key object id
- ASN_VERSION_E Returned if the ALLOW_V1_EXTENSIONS option is not defined and the certificate is a V1 or V2 certificate
- BAD_FUNC_ARG Returned if there is an error processing the certificate extension
- ASN_CRIT_EXT_E Returned if an unfamiliar critical extension is encountered in processing the certificate
- ASN_SIG_OID_E Returned if the signature encryption type is not the same as the encryption type of the certificate in the provided file
- ASN_SIG_CONFIRM_E Returned if confirming the certification signature fails
- ASN_NAME_INVALID_E Returned if the certificate's name is not permitted by the CA name constraints
- ASN_NO_SIGNER_E Returned if there is no CA signer to verify the certificate's authenticity

Example

```
Cert myCert;
// initialize myCert
byte* der;
```

```
der = (byte*)malloc(FOURK_BUF);
// initialize der
if(wc_SetDatesBuffer(&myCert, der, FOURK_BUF) != 0) {
    // error setting subject
}
```

C.4.2.18 function wc_SetAuthKeyIdFromPublicKey

```
int wc_SetAuthKeyIdFromPublicKey(
    Cert * cert,
    RsaKey * rsakey,
    ecc_key * eckey
)
```

Set AKID from either an RSA or ECC public key. note: Only set one of rsakey or eckey, not both.

Parameters:

- **cert** Pointer to the certificate to set the SKID.
- **rsakey** Pointer to the RsaKey struct to read from.
- **eckey** Pointer to the ecc_key to read from.

See:

- wc_SetSubjectKeyId
- wc_SetAuthKeyId
- wc_SetAuthKeyIdFromCert

Return:

- 0 Success
- BAD_FUNC_ARG Either cert is null or both rsakey and eckey are null.
- MEMORY_E Error allocating memory.
- PUBLIC_KEY_E Error writing to the key.

Example

```
Cert myCert;
RsaKey keypub;

wc_InitRsaKey(&keypub, 0);

if (wc_SetAuthKeyIdFromPublicKey(&myCert, &keypub, NULL) != 0)
{
    // Handle error
}
```

C.4.2.19 function wc_SetAuthKeyIdFromCert

```
int wc_SetAuthKeyIdFromCert(
    Cert * cert,
    const byte * der,
    int derSz
)
```

Set AKID from from DER encoded certificate.

Parameters:

- **cert** The Cert struct to write to.

- **der** The DER encoded certificate buffer.
- **derSz** Size of der in bytes.

See:

- wc_SetAuthKeyIdFromPublicKey
- wc_SetAuthKeyId

Return:

- 0 Success
- BAD_FUNC_ARG Error if any argument is null or derSz is less than 0.
- MEMORY_E Error if problem allocating memory.
- ASN_NO_SKID No subject key ID found.

Example

```
Cert some_cert;
byte some_der[] = { // Initialize a DER buffer };
wc_InitCert(&some_cert);
if(wc_SetAuthKeyIdFromCert(&some_cert, some_der, sizeof(some_der) != 0)
{
    // Handle error
}
```

C.4.2.20 function wc_SetAuthKeyId

```
int wc_SetAuthKeyId(
    Cert * cert,
    const char * file
)
```

Set AKID from certificate file in PEM format.

Parameters:

- **cert** Cert struct you want to set the AKID of.
- **file** Buffer containing PEM cert file.

See:

- wc_SetAuthKeyIdFromPublicKey
- wc_SetAuthKeyIdFromCert

Return:

- 0 Success
- BAD_FUNC_ARG Error if cert or file is null.
- MEMORY_E Error if problem allocating memory.

Example

```
char* file_name = "/path/to/file";
cert some_cert;
wc_InitCert(&some_cert);

if(wc_SetAuthKeyId(&some_cert, file_name) != 0)
{
    // Handle Error
}
```

C.4.2.21 function wc_SetSubjectKeyIdFromPublicKey

```
int wc_SetSubjectKeyIdFromPublicKey(
    Cert * cert,
    RsaKey * rsakey,
    ecc_key * eckey
)
```

Set SKID from RSA or ECC public key.

Parameters:

- **cert** Pointer to a Cert structure to be used.
- **rsakey** Pointer to an RsaKey structure
- **eckey** Pointer to an ecc_key structure

See: wc_SetSubjectKeyId

Return:

- 0 Success
- BAD_FUNC_ARG Returned if cert or rsakey and eckey are null.
- MEMORY_E Returned if there is an error allocating memory.
- PUBLIC_KEY_E Returned if there is an error getting the public key.

Example

```
Cert some_cert;
RsaKey some_key;
wc_InitCert(&some_cert);
wc_InitRsaKey(&some_key);

if(wc_SetSubjectKeyIdFromPublicKey(&some_cert,&some_key, NULL) != 0)
{
    // Handle Error
}
```

C.4.2.22 function wc_SetSubjectKeyId

```
int wc_SetSubjectKeyId(
    Cert * cert,
    const char * file
)
```

Set SKID from public key file in PEM format. Both arguments are required.

Parameters:

- **cert** Cert structure to set the SKID of.
- **file** Contains the PEM encoded file.

See: wc_SetSubjectKeyIdFromPublicKey

Return:

- 0 Success
- BAD_FUNC_ARG Returns if cert or file is null.
- MEMORY_E Returns if there is a problem allocating memory for key.
- PUBLIC_KEY_E Returns if there is an error decoding the public key.

Example

```
const char* file_name = "path/to/file";
Cert some_cert;
wc_InitCert(&some_cert);

if(wc_SetSubjectKeyId(&some_cert, file_name) != 0)
{
    // Handle Error
}
```

C.4.2.23 function wc_SetKeyUsage

```
int wc_SetKeyUsage(
    Cert * cert,
    const char * value
)
```

This function allows you to set the key usage using a comma delimited string of tokens. Accepted tokens are: digitalSignature, nonRepudiation, contentCommitment, keyCertSign, cRLSign, dataEncipherment, keyAgreement, keyEncipherment, encipherOnly, decipherOnly. Example: "digitalSignature,nonRepudiation" nonRepudiation and contentCommitment are for the same usage.

Parameters:

- **cert** Pointer to initialized Cert structure.
- **value** Comma delimited string of tokens to set usage.

See:

- wc_InitCert
- wc_MakeRsaKey

Return:

- 0 Success
- BAD_FUNC_ARG Returned when either arg is null.
- MEMORY_E Returned when there is an error allocating memory.
- KEYUSAGE_E Returned if an unrecognized token is entered.

Example

```
Cert cert;
wc_InitCert(&cert);

if(wc_SetKeyUsage(&cert, "cRLSign,keyCertSign") != 0)
{
    // Handle error
}
```

C.4.2.24 function wc_PemPubKeyToDer

```
int wc_PemPubKeyToDer(
    const char * fileName,
    unsigned char * derBuf,
    int derSz
)
```

Loads a PEM key from a file and converts to a DER encoded buffer.

Parameters:

- **fileName** Name of the file to load.
- **derBuf** Buffer for DER encoded key.
- **derSz** Size of DER buffer.

See: wc_PubKeyPemToDer

Return:

- 0 Success
- <0 Error
- SSL_BAD_FILE There is a problem with opening the file.
- MEMORY_E There is an error allocating memory for the file buffer.
- BUFFER_E derBuf is not large enough to hold the converted key.

Example

```
char* some_file = "filename";
unsigned char der[];

if(wc_PemPubKeyToDer(some_file, der, sizeof(der)) != 0)
{
    //Handle Error
}
```

C.4.2.25 function wc_PubKeyPemToDer

```
int wc_PubKeyPemToDer(
    const unsigned char * pem,
    int pemSz,
    unsigned char * buff,
    int buffSz
)
```

Convert a PEM encoded public key to DER. Returns the number of bytes written to the buffer or a negative value for an error.

Parameters:

- **pem** PEM encoded key
- **pemSz** Size of pem
- **buff** Pointer to buffer for output.
- **buffSz** Size of buffer.

See: wc_PemPubKeyToDer

Return:

- 0 Success, number of bytes written.
- BAD_FUNC_ARG Returns if pem, buff, or buffSz are null
- <0 An error occurred in the function.

Example

```
byte some_pem[] = { Initialize with PEM key }
unsigned char out_buffer[1024]; // Ensure buffer is large enough to fit DER

if(wc_PubKeyPemToDer(some_pem, sizeof(some_pem), out_buffer,
sizeof(out_buffer)) < 0)
{
```

```
    // Handle error
}
```

C.4.2.26 function wc_PemCertToDer

```
int wc_PemCertToDer(
    const char * fileName,
    unsigned char * derBuf,
    int derSz
)
```

This function converts a pem certificate to a der certificate, and places the resulting certificate in the derBuf buffer provided.

Parameters:

- **fileName** path to the file containing a pem certificate to convert to a der certificate
- **derBuf** pointer to a char buffer in which to store the converted certificate
- **derSz** size of the char buffer in which to store the converted certificate

See: none

Return:

- Success On success returns the size of the derBuf generated
- BUFFER_E Returned if the size of derBuf is too small to hold the certificate generated
- MEMORY_E Returned if the call to XMALLOC fails

Example

```
char * file = "./certs/client-cert.pem";
int derSz;
byte* der = (byte*)XMALLOC((8*1024), NULL, DYNAMIC_TYPE_CERT);

derSz = wc_PemCertToDer(file, der, (8*1024));
if (derSz <= 0) {
    //PemCertToDer error
}
```

C.4.2.27 function wc_DerToPem

```
int wc_DerToPem(
    const byte * der,
    word32 derSz,
    byte * output,
    word32 outputSz,
    int type
)
```

This function converts a der formatted input certificate, contained in the der buffer, into a pem formatted output certificate, contained in the output buffer. It should be noted that this is not an in place conversion, and a separate buffer must be utilized to store the pem formatted output.

Parameters:

- **der** pointer to the buffer of the certificate to convert
- **derSz** size of the the certificate to convert
- **output** pointer to the buffer in which to store the pem formatted certificate
- **outSz** size of the buffer in which to store the pem formatted certificate

- **type** the type of certificate to generate. Valid types are: CERT_TYPE, PRIVATEKEY_TYPE, ECC_PRIVATEKEY_TYPE, and CERTREQ_TYPE.

See: wc_PemCertToDer

Return:

- Success On successfully making a pem certificate from the input der cert, returns the size of the pem cert generated.
- BAD_FUNC_ARG Returned if there is an error parsing the der file and storing it as a pem file
- MEMORY_E Returned if there is an error allocating memory with XMALLOC
- ASN_INPUT_E Returned in the case of a base64 encoding error
- BUFFER_E May be returned if the output buffer is too small to store the pem formatted certificate

Example

```
byte* der;
// initialize der with certificate
byte* pemFormatted[FOURK_BUF];

word32 pemSz;
pemSz = wc_DerToPem(der, derSz,pemFormatted,FOURK_BUF, CERT_TYPE);
```

C.4.2.28 function wc_DerToPemEx

```
int wc_DerToPemEx(
    const byte * der,
    word32 derSz,
    byte * output,
    word32 outputSz,
    byte * cipherIno,
    int type
)
```

This function converts a der formatted input certificate, contained in the der buffer, into a pem formatted output certificate, contained in the output buffer. It should be noted that this is not an in place conversion, and a separate buffer must be utilized to store the pem formatted output. Allows setting cipher info.

Parameters:

- **der** pointer to the buffer of the certificate to convert
- **derSz** size of the the certificate to convert
- **output** pointer to the buffer in which to store the pem formatted certificate
- **outSz** size of the buffer in which to store the pem formatted certificate
- **cipher_inf** Additional cipher information.
- **type** the type of certificate to generate. Valid types are: CERT_TYPE, PRIVATEKEY_TYPE, ECC_PRIVATEKEY_TYPE, and CERTREQ_TYPE.

See: wc_PemCertToDer

Return:

- Success On successfully making a pem certificate from the input der cert, returns the size of the pem cert generated.
- BAD_FUNC_ARG Returned if there is an error parsing the der file and storing it as a pem file
- MEMORY_E Returned if there is an error allocating memory with XMALLOC
- ASN_INPUT_E Returned in the case of a base64 encoding error
- BUFFER_E May be returned if the output buffer is too small to store the pem formatted certificate

Example

```
byte* der;
// initialize der with certificate
byte* pemFormatted[FOURK_BUF];

word32 pemSz;
byte* cipher_info[] { Additional cipher info. }
pemSz = wc_DerToPemEx(der, derSz, pemFormatted, FOURK_BUF, cipher_info,
 ↪  CERT_TYPE);
```

C.4.2.29 function wc_KeyPemToDer

```
int wc_KeyPemToDer(
    const unsigned char * pem,
    int pemSz,
    unsigned char * buff,
    int buffSz,
    const char * pass
)
```

Converts a key in PEM format to DER format.

Parameters:

- **pem** a pointer to the PEM encoded certificate.
- **pemSz** the size of the PEM buffer (pem)
- **buff** a pointer to the copy of the buffer member of the DerBuffer struct.
- **buffSz** size of the buffer space allocated in the DerBuffer struct.
- **pass** password passed into the function.

See: wc_PemToDer

Return:

- int the function returns the number of bytes written to the buffer on successful execution.
- int negative int returned indicating an error.

Example

```
byte* loadBuf;
long fileSz = 0;
byte* bufSz;
static int LoadKeyFile(byte** keyBuf, word32* keyBufSz,
const char* keyFile,
                int typeKey, const char* password);
…
bufSz = wc_KeyPemToDer(loadBuf, (int)fileSz, saveBuf,
(int)fileSz, password);

if(saveBufSz > 0){
    // Bytes were written to the buffer.
}
```

C.4.2.30 function wc_CertPemToDer

```
int wc_CertPemToDer(
    const unsigned char * pem,
    int pemSz,
```

```
    unsigned char * buff,
    int buffSz,
    int type
)
```

This function converts a PEM formatted certificate to DER format. Calls OpenSSL function PemToDer.

Parameters:

- **pem** pointer PEM formatted certificate.
- **pemSz** size of the certificate.
- **buff** buffer to be copied to DER format.
- **buffSz** size of the buffer.
- **type** Certificate file type found in asn_public.h enum CertType.

See: wc_PemToDer

Return: buffer returns the bytes written to the buffer.

Example

```
const unsigned char* pem;
int pemSz;
unsigned char buff[BUFSIZE];
int buffSz = sizeof(buff)/sizeof(char);
int type;
...
if(wc_CertPemToDer(pem, pemSz, buff, buffSz, type) <= 0) {
    // There were bytes written to buffer
}
```

C.4.2.31 function wc_GetPubKeyDerFromCert

```
int wc_GetPubKeyDerFromCert(
    struct DecodedCert * cert,
    byte * derKey,
    word32 * derKeySz
)
```

This function gets the public key in DER format from a populated DecodedCert struct. Users must call wc_InitDecodedCert().

Parameters:

- **cert** populated DecodedCert struct holding X.509 certificate
- **derKey** output buffer to place DER encoded public key
- **derKeySz** [IN/OUT] size of derKey buffer on input, size of public key on return. If derKey is passed in as NULL, derKeySz will be set to required buffer size for public key and LENGTH_ONLY_E will be returned from function.

See: wc_GetPubKeyDerFromCert

Return: 0 on success, negative on error. LENGTH_ONLY_E if derKey is NULL and returning length only.

C.4.2.32 function wc_EccPrivateKeyDecode

```
int wc_EccPrivateKeyDecode(
    const byte * input,
    word32 * inOutIdx,
    ecc_key * key,
```

```
    word32 inSz
)
```

This function reads in an ECC private key from the input buffer, input, parses the private key, and uses it to generate an ecc_key object, which it stores in key.

Parameters:

- **input** pointer to the buffer containing the input private key
- **inOutIdx** pointer to a word32 object containing the index in the buffer at which to start
- **key** pointer to an initialized ecc object, on which to store the decoded private key
- **inSz** size of the input buffer containing the private key

See: wc_RSA_PrivateKeyDecode

Return:

- 0 On successfully decoding the private key and storing the result in the ecc_key struct
- ASN_PARSE_E: Returned if there is an error parsing the der file and storing it as a pem file
- MEMORY_E Returned if there is an error allocating memory with XMALLOC
- BUFFER_E Returned if the certificate to convert is large than the specified max certificate size
- ASN_OBJECT_ID_E Returned if the certificate encoding has an invalid object id
- ECC_CURVE_OID_E Returned if the ECC curve of the provided key is not supported
- ECC_BAD_ARG_E Returned if there is an error in the ECC key format
- NOT_COMPILED_IN Returned if the private key is compressed, and no compression key is provided
- MP_MEM Returned if there is an error in the math library used while parsing the private key
- MP_VAL Returned if there is an error in the math library used while parsing the private key
- MP_RANGE Returned if there is an error in the math library used while parsing the private key

Example

```
int ret, idx=0;
ecc_key key; // to store key in

byte* tmp; // tmp buffer to read key from
tmp = (byte*) malloc(FOURK_BUF);

int inSz;
inSz = fread(tmp, 1, FOURK_BUF, privateKeyFile);
// read key into tmp buffer

wc_ecc_init(&key); // initialize key
ret = wc_EccPrivateKeyDecode(tmp, &idx, &key, (word32)inSz);
if(ret < 0) {
    // error decoding ecc key
}
```

C.4.2.33 function wc_EccKeyToDer

```
int wc_EccKeyToDer(
    ecc_key * key,
    byte * output,
    word32 inLen
)
```

This function writes a private ECC key to der format.

Parameters:

- **key** pointer to the buffer containing the input ecc key
- **output** pointer to a buffer in which to store the der formatted key
- **inLen** the length of the buffer in which to store the der formatted key

See: wc_RsaKeyToDer

Return:

- Success On successfully writing the ECC key to der format, returns the length written to the buffer
- BAD_FUNC_ARG Returned if key or output is null, or inLen equals zero
- MEMORY_E Returned if there is an error allocating memory with XMALLOC
- BUFFER_E Returned if the converted certificate is too large to store in the output buffer
- ASN_UNKNOWN_OID_E Returned if the ECC key used is of an unknown type
- MP_MEM Returned if there is an error in the math library used while parsing the private key
- MP_VAL Returned if there is an error in the math library used while parsing the private key
- MP_RANGE Returned if there is an error in the math library used while parsing the private key

Example

```
int derSz;
ecc_key key;
// initialize and make key
byte der[FOURK_BUF];
// store der formatted key here

derSz = wc_EccKeyToDer(&key, der, FOURK_BUF);
if(derSz < 0) {
    // error converting ecc key to der buffer
}
```

C.4.2.34 function wc_EccPublicKeyDecode

```
int wc_EccPublicKeyDecode(
    const byte * input,
    word32 * inOutIdx,
    ecc_key * key,
    word32 inSz
)
```

Decodes an ECC public key from an input buffer. It will parse an ASN sequence to retrieve the ECC key.

Parameters:

- **input** Buffer containing DER encoded key to decode.
- **inOutIdx** Index to start reading input buffer from. On output, index is set to last position parsed of input buffer.
- **key** Pointer to ecc_key struct to store the public key.
- **inSz** Size of the input buffer.

See: wc_ecc_import_x963

Return:

- 0 Success
- BAD_FUNC_ARG Returns if any arguments are null.
- ASN_PARSE_E Returns if there is an error parsing
- ASN_ECC_KEY_E Returns if there is an error importing the key. See wc_ecc_import_x963 for possible reasons.

Example

```
int ret;
word32 idx = 0;
byte buff[] = { // initialize with key };
ecc_key pubKey;
wc_ecc_init(&pubKey);
if ( wc_EccPublicKeyDecode(buff, &idx, &pubKey, sizeof(buff)) != 0) {
        // error decoding key
}
```

C.4.2.35 function wc_EccPublicKeyToDer

```
int wc_EccPublicKeyToDer(
    ecc_key * key,
    byte * output,
    word32 inLen,
    int with_AlgCurve
)
```

This function converts the ECC public key to DER format. It returns the size of buffer used. The public ECC key in DER format is stored in output buffer. The with_AlgCurve flag will include a header that has the Algorithm and Curve information.

Parameters:

- **key** Pointer to ECC key
- **output** Pointer to output buffer to write to.
- **inLen** Size of buffer.
- **with_AlgCurve** a flag for when to include a header that has the Algorithm and Curve information.

See:

- wc_EccKeyToDer
- wc_EccPrivateKeyDecode

Return:

- 0 Success, size of buffer used
- BAD_FUNC_ARG Returned if output or key is null.
- LENGTH_ONLY_E Error in getting ECC public key size.
- BUFFER_E Returned when output buffer is too small.

Example

```
ecc_key key;
wc_ecc_init(&key);
WC_RNG rng;
wc_InitRng(&rng);
wc_ecc_make_key(&rng, 32, &key);
int derSz = // Some appropriate size for der;
byte der[derSz];

if(wc_EccPublicKeyToDer(&key, der, derSz, 1) < 0)
{
    // Error converting ECC public key to der
}
```

C.4.2.36 function wc_EccPublicKeyToDer_ex

```
int wc_EccPublicKeyToDer_ex(
    ecc_key * key,
    byte * output,
    word32 inLen,
    int with_AlgCurve,
    int comp
)
```

This function converts the ECC public key to DER format. It returns the size of buffer used. The public ECC key in DER format is stored in output buffer. The with_AlgCurve flag will include a header that has the Algorithm and Curve information. The comp parameter determines if the public key will be exported as compressed.

Parameters:

- **key** Pointer to ECC key
- **output** Pointer to output buffer to write to.
- **inLen** Size of buffer.
- **with_AlgCurve** a flag for when to include a header that has the Algorithm and Curve information.
- **comp** If 1 (non-zero) the ECC public key will be written in compressed form. If 0 it will be written in an uncompressed format.

See:

- wc_EccKeyToDer
- wc_EccPublicKeyDecode

Return:

- 0 Success, size of buffer used
- BAD_FUNC_ARG Returned if output or key is null.
- LENGTH_ONLY_E Error in getting ECC public key size.
- BUFFER_E Returned when output buffer is too small.

Example

```
ecc_key key;
wc_ecc_init(&key);
WC_RNG rng;
wc_InitRng(&rng);
wc_ecc_make_key(&rng, 32, &key);
int derSz = // Some appropriate size for der;
byte der[derSz];

// Write out a compressed ECC key
if(wc_EccPublicKeyToDer_ex(&key, der, derSz, 1, 1) < 0)
{
    // Error converting ECC public key to der
}
```

C.4.2.37 function wc_Curve25519PrivateKeyDecode

```
int wc_Curve25519PrivateKeyDecode(
    const byte * input,
    word32 * inOutIdx,
    curve25519_key * key,
```

```
    word32 inSz
)
```

This function decodes a Curve25519 private key (only) from a DER encoded buffer.

Parameters:

- **input** Pointer to buffer containing DER encoded private key
- **inOutIdx** Index to start reading input buffer from. On output, index is set to last position parsed of input buffer.
- **key** Pointer to curve25519_key structure to store decoded key
- **inSz** Size of input DER buffer

See:

- wc_Curve25519KeyDecode
- wc_Curve25519PublicKeyDecode

Return:

- 0 Success
- BAD_FUNC_ARG Returns if input, inOutIdx or key is null
- ASN_PARSE_E Returns if there is an error parsing the DER encoded data
- ECC_BAD_ARG_E Returns if the key length is not CURVE25519_KEYSIZE or the DER key contains other issues despite being properly formatted.
- BUFFER_E Returns if the input buffer is too small to contain a valid DER encoded key.

Example

```
byte der[] = { // DER encoded key };
word32 idx = 0;
curve25519_key key;
wc_curve25519_init(&key);

if (wc_Curve25519PrivateKeyDecode(der, &idx, &key, sizeof(der)) != 0) {
    // Error decoding private key
}
```

C.4.2.38 function wc_Curve25519PublicKeyDecode

```
int wc_Curve25519PublicKeyDecode(
    const byte * input,
    word32 * inOutIdx,
    curve25519_key * key,
    word32 inSz
)
```

This function decodes a Curve25519 public key (only) from a DER encoded buffer.

Parameters:

- **input** Pointer to buffer containing DER encoded public key
- **inOutIdx** Index to start reading input buffer from. On output, index is set to last position parsed of input buffer.
- **key** Pointer to curve25519_key structure to store decoded key
- **inSz** Size of input DER buffer

See:

- wc_Curve25519KeyDecode
- wc_Curve25519PrivateKeyDecode

Return:

- 0 Success
- BAD_FUNC_ARG Returns if input, inOutIdx or key is null
- ASN_PARSE_E Returns if there is an error parsing the DER encoded data
- ECC_BAD_ARG_E Returns if the key length is not CURVE25519_KEYSIZE or the DER key contains other issues despite being properly formatted.
- BUFFER_E Returns if the input buffer is too small to contain a valid DER encoded key.

Example

```
byte der[] = { // DER encoded key };
word32 idx = 0;
curve25519_key key;
wc_curve25519_init(&key);
if (wc_Curve25519PublicKeyDecode(der, &idx, &key, sizeof(der)) != 0) {
    // Error decoding public key
}
```

C.4.2.39 function wc_Curve25519KeyDecode

```
int wc_Curve25519KeyDecode(
    const byte * input,
    word32 * inOutIdx,
    curve25519_key * key,
    word32 inSz
)
```

This function decodes a Curve25519 key from a DER encoded buffer. It can decode either a private key, a public key, or both.

Parameters:

- **input** Pointer to buffer containing DER encoded key
- **inOutIdx** Index to start reading input buffer from. On output, index is set to last position parsed of input buffer.
- **key** Pointer to curve25519_key structure to store decoded key
- **inSz** Size of input DER buffer

See:

- wc_Curve25519PrivateKeyDecode
- wc_Curve25519PublicKeyDecode

Return:

- 0 Success
- BAD_FUNC_ARG Returns if input, inOutIdx or key is null
- ASN_PARSE_E Returns if there is an error parsing the DER encoded data
- ECC_BAD_ARG_E Returns if the key length is not CURVE25519_KEYSIZE or the DER key contains other issues despite being properly formatted.
- BUFFER_E Returns if the input buffer is too small to contain a valid DER encoded key.

Example

```
byte der[] = { // DER encoded key };
word32 idx = 0;
curve25519_key key;
wc_curve25519_init(&key);
if (wc_Curve25519KeyDecode(der, &idx, &key, sizeof(der)) != 0) {
```

```
    // Error decoding key
}
```

C.4.2.40 function wc_Curve25519PrivateKeyToDer

```
int wc_Curve25519PrivateKeyToDer(
    curve25519_key * key,
    byte * output,
    word32 inLen
)
```

This function encodes a Curve25519 private key to DER format. If the input key structure contains a public key, it will be ignored.

Parameters:

- **key** Pointer to curve25519_key structure containing private key to encode
- **output** Buffer to hold DER encoding
- **inLen** Size of output buffer

See:

- wc_Curve25519KeyToDer
- wc_Curve25519PublicKeyToDer

Return:

- 0 Success, length of DER encoding
- BAD_FUNC_ARG Returns if key or output is null
- MEMORY_E Returns if there is an allocation failure
- BUFFER_E Returns if output buffer is too small

Example

```
curve25519_key key;
wc_curve25519_init(&key);
...
int derSz = 128; // Some appropriate size for output DER
byte der[derSz];
wc_Curve25519PrivateKeyToDer(&key, der, derSz);
```

C.4.2.41 function wc_Curve25519PublicKeyToDer

```
int wc_Curve25519PublicKeyToDer(
    curve25519_key * key,
    byte * output,
    word32 inLen,
    int withAlg
)
```

This function encodes a Curve25519 public key to DER format. If the input key structure contains a private key, it will be ignored.

Parameters:

- **key** Pointer to curve25519_key structure containing public key to encode
- **output** Buffer to hold DER encoding
- **inLen** Size of output buffer
- **withAlg** Whether to include algorithm identifier in the DER encoding

See:

- wc_Curve25519KeyToDer
- wc_Curve25519PrivateKeyToDer

Return:

- 0 Success, length of DER encoding
- BAD_FUNC_ARG Returns if key or output is null
- MEMORY_E Returns if there is an allocation failure
- BUFFER_E Returns if output buffer is too small

Example

```
curve25519_key key;
wc_curve25519_init(&key);
...
int derSz = 128; // Some appropriate size for output DER
byte der[derSz];
wc_Curve25519PublicKeyToDer(&key, der, derSz, 1);
```

C.4.2.42 function wc_Curve25519KeyToDer

```
int wc_Curve25519KeyToDer(
    curve25519_key * key,
    byte * output,
    word32 inLen,
    int withAlg
)
```

This function encodes a Curve25519 key to DER format. It can encode either a private key, a public key, or both.

Parameters:

- **key** Pointer to curve25519_key structure containing key to encode
- **output** Buffer to hold DER encoding
- **inLen** Size of output buffer
- **withAlg** Whether to include algorithm identifier in the DER encoding

See:

- wc_Curve25519PrivateKeyToDer
- wc_Curve25519PublicKeyToDer

Return:

- 0 Success, length of DER encoding
- BAD_FUNC_ARG Returns if key or output is null
- MEMORY_E Returns if there is an allocation failure
- BUFFER_E Returns if output buffer is too small

Example

```
curve25519_key key;
wc_curve25519_init(&key);
...
int derSz = 128; // Some appropriate size for output DER
```

```
byte der[derSz];
wc_Curve25519KeyToDer(&key, der, derSz, 1);
```

C.4.2.43 function wc_EncodeSignature

```
word32 wc_EncodeSignature(
    byte * out,
    const byte * digest,
    word32 digSz,
    int hashOID
)
```

This function encodes a digital signature into the output buffer, and returns the size of the encoded signature created.

Parameters:

- **out** pointer to the buffer where the encoded signature will be written
- **digest** pointer to the digest to use to encode the signature
- **digSz** the length of the buffer containing the digest
- **hashOID** OID identifying the hash type used to generate the signature. Valid options, depending on build configurations, are: SHAh, SHA256h, SHA384h, SHA512h, MD2h, MD5h, DESb, DES3b, CTC_MD5wRSA, CTC_SHAwRSA, CTC_SHA256wRSA, CTC_SHA384wRSA, CTC_SHA512wRSA, CTC_SHAwECDSA, CTC_SHA256wECDSA, CTC_SHA384wECDSA, and CTC_SHA512wECDSA.

See: none

Return: Success On successfully writing the encoded signature to output, returns the length written to the buffer

```
int signSz;
byte encodedSig[MAX_ENCODED_SIG_SZ];
Sha256 sha256;
// initialize sha256 for hashing

byte* dig = = (byte*)malloc(WC_SHA256_DIGEST_SIZE);
// perform hashing and hash updating so dig stores SHA-256 hash
// (see wc_InitSha256, wc_Sha256Update and wc_Sha256Final)
signSz = wc_EncodeSignature(encodedSig, dig, WC_SHA256_DIGEST_SIZE, SHA256h);
```

C.4.2.44 function wc_GetCTC_HashOID

```
int wc_GetCTC_HashOID(
    int type
)
```

This function returns the hash OID that corresponds to a hashing type. For example, when given the type: WC_SHA512, this function returns the identifier corresponding to a SHA512 hash, SHA512h.

Parameters:

- **type** the hash type for which to find the OID. Valid options, depending on build configuration, include: WC_MD5, WC_SHA, WC_SHA256, WC_SHA384, WC_SHA512, WC_SHA3_224, WC_SHA3_256, WC_SHA3_384 or WC_SHA3_512

See: none

Return:

- Success On success, returns the OID corresponding to the appropriate hash to use with that encryption type.
- 0 Returned if an unrecognized hash type is passed in as argument.

Example

```
int hashOID;

hashOID = wc_GetCTC_HashOID(WC_SHA512);
if (hashOID == 0) {
    // WOLFSSL_SHA512 not defined
}
```

C.4.2.45 function wc_SetCert_Free

```
void wc_SetCert_Free(
    Cert * cert
)
```

This function cleans up memory and resources used by the certificate structure's decoded cert cache. When WOLFSSL_CERT_GEN_CACHE is defined the decoded cert structure is cached in the certificate structure. This allows subsequent calls to certificate set functions to avoid parsing the decoded cert on each call.

Parameters:

- **cert** pointer to an uninitialized certificate information structure.

See:

- wc_SetAuthKeyIdFromCert
- wc_SetIssuerBuffer
- wc_SetSubjectBuffer
- wc_SetSubjectRaw
- wc_SetIssuerRaw
- wc_SetAltNamesBuffer
- wc_SetDatesBuffer

Return:

- 0 on success.
- BAD_FUNC_ARG Returned if invalid pointer is passed in as argument.

Example

```
Cert cert; // Initialized certificate structure

wc_SetCert_Free(&cert);
```

C.4.2.46 function wc_GetPkcs8TraditionalOffset

```
int wc_GetPkcs8TraditionalOffset(
    byte * input,
    word32 * inOutIdx,
    word32 sz
)
```

This function finds the beginning of the traditional private key inside a PKCS#8 unencrypted buffer.

Parameters:

- **input** Buffer containing unencrypted PKCS#8 private key.
- **inOutIdx** Index into the input buffer. On input, it should be a byte offset to the beginning of the the PKCS#8 buffer. On output, it will be the byte offset to the traditional private key within the input buffer.
- **sz** The number of bytes in the input buffer.

See:

- wc_CreatePKCS8Key
- wc_EncryptPKCS8Key
- wc_DecryptPKCS8Key
- wc_CreateEncryptedPKCS8Key

Return:

- Length of traditional private key on success.
- Negative values on failure.

Example

```
byte* pkcs8Buf; // Buffer containing PKCS#8 key.
word32 idx = 0;
word32 sz; // Size of pkcs8Buf.
...
ret = wc_GetPkcs8TraditionalOffset(pkcs8Buf, &idx, sz);
// pkcs8Buf + idx is now the beginning of the traditional private key bytes.
```

C.4.2.47 function wc_CreatePKCS8Key

```
int wc_CreatePKCS8Key(
    byte * out,
    word32 * outSz,
    byte * key,
    word32 keySz,
    int algoID,
    const byte * curveOID,
    word32 oidSz
)
```

This function takes in a DER private key and converts it to PKCS#8 format. Also used in creating PKCS#12 shrouded key bags. See RFC 5208.

Parameters:

- **out** Buffer to place result in. If NULL, required out buffer size returned in outSz.
- **outSz** Size of out buffer.
- **key** Buffer with traditional DER key.
- **keySz** Size of key buffer.
- **algoID** Algorithm ID (e.g. RSAk).
- **curveOID** ECC curve OID if used. Should be NULL for RSA keys.
- **oidSz** Size of curve OID. Is set to 0 if curveOID is NULL.

See:

- wc_GetPkcs8TraditionalOffset
- wc_EncryptPKCS8Key
- wc_DecryptPKCS8Key
- wc_CreateEncryptedPKCS8Key

Return:

- The size of the PKCS#8 key placed into out on success.
- LENGTH_ONLY_E if out is NULL, with required output buffer size in outSz.
- Other negative values on failure.

Example

```
ecc_key eccKey;              // wolfSSL ECC key object.
byte* der;                   // DER-encoded ECC key.
word32 derSize;              // Size of der.
const byte* curveOid = NULL; // OID of curve used by eccKey.
word32 curveOidSz = 0;       // Size of curve OID.
byte* pkcs8;                 // Output buffer for PKCS#8 key.
word32 pkcs8Sz;              // Size of output buffer.

derSize = wc_EccKeyDerSize(&eccKey, 1);
...
derSize = wc_EccKeyToDer(&eccKey, der, derSize);
...
ret = wc_ecc_get_oid(eccKey.dp->oidSum, &curveOid, &curveOidSz);
...
ret = wc_CreatePKCS8Key(NULL, &pkcs8Sz, der,
    derSize, ECDSAk, curveOid, curveOidSz); // Get size needed in pkcs8Sz.
...
ret = wc_CreatePKCS8Key(pkcs8, &pkcs8Sz, der,
    derSize, ECDSAk, curveOid, curveOidSz);
```

C.4.2.48 function wc_EncryptPKCS8Key

```
int wc_EncryptPKCS8Key(
    byte * key,
    word32 keySz,
    byte * out,
    word32 * outSz,
    const char * password,
    int passwordSz,
    int vPKCS,
    int pbeOid,
    int encAlgId,
    byte * salt,
    word32 saltSz,
    int itt,
    WC_RNG * rng,
    void * heap
)
```

This function takes in an unencrypted PKCS#8 DER key (e.g. one created by wc_CreatePKCS8Key) and converts it to PKCS#8 encrypted format. The resulting encrypted key can be decrypted using wc_DecryptPKCS8Key. See RFC 5208.

Parameters:

- **key** Buffer with traditional DER key.
- **keySz** Size of key buffer.
- **out** Buffer to place result in. If NULL, required out buffer size returned in outSz.
- **outSz** Size of out buffer.
- **password** The password to use for the password-based encryption algorithm.
- **passwordSz** The length of the password (not including the NULL terminator).

- **vPKCS** The PKCS version to use. Can be 1 for PKCS12 or PKCS5.
- **pbeOid** The OID of the PBE scheme to use (e.g. PBES2 or one of the OIDs for PBES1 in RFC 2898 A.3).
- **encAlgId** The encryption algorithm ID to use (e.g. AES256CBCb).
- **salt** The salt buffer to use. If NULL, a random salt will be used.
- **saltSz** The length of the salt buffer. Can be 0 if passing NULL for salt.
- **itt** The number of iterations to use for the KDF.
- **rng** A pointer to an initialized WC_RNG object.
- **heap** A pointer to the heap used for dynamic allocation. Can be NULL.

See:

- wc_GetPkcs8TraditionalOffset
- wc_CreatePKCS8Key
- wc_DecryptPKCS8Key
- wc_CreateEncryptedPKCS8Key

Return:

- The size of the encrypted key placed in out on success.
- LENGTH_ONLY_E if out is NULL, with required output buffer size in outSz.
- Other negative values on failure.

Example

```
byte* pkcs8;          // Unencrypted PKCS#8 key.
word32 pkcs8Sz;       // Size of pkcs8.
byte* pkcs8Enc;       // Encrypted PKCS#8 key.
word32 pkcs8EncSz;    // Size of pkcs8Enc.
const char* password; // Password to use for encryption.
int passwordSz;       // Length of password (not including NULL terminator).
WC_RNG rng;

// The following produces an encrypted version of pkcs8 in pkcs8Enc. The
// encryption uses password-based encryption scheme 2 (PBE2) from PKCS#5 and
// the AES cipher in CBC mode with a 256-bit key. See RFC 8018 for more on
// PKCS#5.
ret = wc_EncryptPKCS8Key(pkcs8, pkcs8Sz, pkcs8Enc, &pkcs8EncSz, password,
        passwordSz, PKCS5, PBES2, AES256CBCb, NULL, 0,
        WC_PKCS12_ITT_DEFAULT, &rng, NULL);
```

C.4.2.49 function wc_DecryptPKCS8Key

```
int wc_DecryptPKCS8Key(
    byte * input,
    word32 sz,
    const char * password,
    int passwordSz
)
```

This function takes an encrypted PKCS#8 DER key and decrypts it to PKCS#8 unencrypted DER. Undoes the encryption done by wc_EncryptPKCS8Key. See RFC5208. The input buffer is overwritten with the decrypted data.

Parameters:

- **input** On input, buffer containing encrypted PKCS#8 key. On successful output, contains the decrypted key.
- **sz** Size of the input buffer.

- **password** The password used to encrypt the key.
- **passwordSz** The length of the password (not including NULL terminator).

See:

- wc_GetPkcs8TraditionalOffset
- wc_CreatePKCS8Key
- wc_EncryptPKCS8Key
- wc_CreateEncryptedPKCS8Key

Return:

- The length of the decrypted buffer on success.
- Negative values on failure.

Example

```
byte* pkcs8Enc;       // Encrypted PKCS#8 key made with wc_EncryptPKCS8Key.
word32 pkcs8EncSz;    // Size of pkcs8Enc.
const char* password; // Password to use for decryption.
int passwordSz;       // Length of password (not including NULL terminator).

ret = wc_DecryptPKCS8Key(pkcs8Enc, pkcs8EncSz, password, passwordSz);
```

C.4.2.50 function wc_CreateEncryptedPKCS8Key

```
int wc_CreateEncryptedPKCS8Key(
    byte * key,
    word32 keySz,
    byte * out,
    word32 * outSz,
    const char * password,
    int passwordSz,
    int vPKCS,
    int pbeOid,
    int encAlgId,
    byte * salt,
    word32 saltSz,
    int itt,
    WC_RNG * rng,
    void * heap
)
```

This function takes a traditional, DER key, converts it to PKCS#8 format, and encrypts it. It uses wc_CreatePKCS8Key and wc_EncryptPKCS8Key to do this.

Parameters:

- **key** Buffer with traditional DER key.
- **keySz** Size of key buffer.
- **out** Buffer to place result in. If NULL, required out buffer size returned in outSz.
- **outSz** Size of out buffer.
- **password** The password to use for the password-based encryption algorithm.
- **passwordSz** The length of the password (not including the NULL terminator).
- **vPKCS** The PKCS version to use. Can be 1 for PKCS12 or PKCS5.
- **pbeOid** The OID of the PBE scheme to use (e.g. PBES2 or one of the OIDs for PBES1 in RFC 2898 A.3).
- **encAlgId** The encryption algorithm ID to use (e.g. AES256CBCb).
- **salt** The salt buffer to use. If NULL, a random salt will be used.

- **saltSz** The length of the salt buffer. Can be 0 if passing NULL for salt.
- **itt** The number of iterations to use for the KDF.
- **rng** A pointer to an initialized WC_RNG object.
- **heap** A pointer to the heap used for dynamic allocation. Can be NULL.

See:

- wc_GetPkcs8TraditionalOffset
- wc_CreatePKCS8Key
- wc_EncryptPKCS8Key
- wc_DecryptPKCS8Key

Return:

- The size of the encrypted key placed in out on success.
- LENGTH_ONLY_E if out is NULL, with required output buffer size in outSz.
- Other negative values on failure.

Example

```
byte* key;            // Traditional private key (DER formatted).
word32 keySz;         // Size of key.
byte* pkcs8Enc;       // Encrypted PKCS#8 key.
word32 pkcs8EncSz;    // Size of pkcs8Enc.
const char* password; // Password to use for encryption.
int passwordSz;       // Length of password (not including NULL terminator).
WC_RNG rng;

// The following produces an encrypted, PKCS#8 version of key in pkcs8Enc.
// The encryption uses password-based encryption scheme 2 (PBE2) from PKCS#5
// and the AES cipher in CBC mode with a 256-bit key. See RFC 8018 for more
// on PKCS#5.
ret = wc_CreateEncryptedPKCS8Key(key, keySz, pkcs8Enc, &pkcs8EncSz,
        password, passwordSz, PKCS5, PBES2, AES256CBCb, NULL, 0,
        WC_PKCS12_ITT_DEFAULT, &rng, NULL);
```

C.4.2.51 function wc_InitDecodedCert

```
void wc_InitDecodedCert(
    struct DecodedCert * cert,
    const byte * source,
    word32 inSz,
    void * heap
)
```

This function initializes the DecodedCert pointed to by the “cert” parameter. It saves the “source” pointer to a DER-encoded certificate of length “inSz.” This certificate can be parsed by a subsequent call to wc_ParseCert.

Parameters:

- **cert** Pointer to an allocated DecodedCert object.
- **source** Pointer to a DER-encoded certificate.
- **inSz** Length of the DER-encoded certificate in bytes.
- **heap** A pointer to the heap used for dynamic allocation. Can be NULL.

See:

- wc_ParseCert
- wc_FreeDecodedCert

Example

```
DecodedCert decodedCert; // Decoded certificate object.
byte* certBuf;           // DER-encoded certificate buffer.
word32 certBufSz;        // Size of certBuf in bytes.

wc_InitDecodedCert(&decodedCert, certBuf, certBufSz, NULL);
```

C.4.2.52 function wc_ParseCert

```
int wc_ParseCert(
    DecodedCert * cert,
    int type,
    int verify,
    void * cm
)
```

This function parses the DER-encoded certificate saved in the DecodedCert object and populates the fields of that object. The DecodedCert must have been initialized with a prior call to wc_InitDecodedCert. This function takes an optional pointer to a CertificateManager object, which is used to populate the certificate authority information of the DecodedCert, if the CA is found in the CertificateManager.

Parameters:

- **cert** Pointer to an initialized DecodedCert object.
- **type** Type of certificate. See the CertType enum in asn_public.h.
- **verify** Flag that, if set, indicates the user wants to verify the validity of the certificate.
- **cm** An optional pointer to a CertificateManager. Can be NULL.

See:

- wc_InitDecodedCert
- wc_FreeDecodedCert

Return:

- 0 on success.
- Other negative values on failure.

Example

```
int ret;
DecodedCert decodedCert; // Decoded certificate object.
byte* certBuf;           // DER-encoded certificate buffer.
word32 certBufSz;        // Size of certBuf in bytes.

wc_InitDecodedCert(&decodedCert, certBuf, certBufSz, NULL);
ret = wc_ParseCert(&decodedCert, CERT_TYPE, NO_VERIFY, NULL);
if (ret != 0) {
    fprintf(stderr, "wc_ParseCert failed.\n");
}
```

C.4.2.53 function wc_FreeDecodedCert

```
void wc_FreeDecodedCert(
    struct DecodedCert * cert
)
```

This function frees a DecodedCert that was previously initialized with wc_InitDecodedCert.

Parameters:

- **cert** Pointer to an initialized DecodedCert object.

See:

- wc_InitDecodedCert
- wc_ParseCert

Example

```
int ret;
DecodedCert decodedCert; // Decoded certificate object.
byte* certBuf;           // DER-encoded certificate buffer.
word32 certBufSz;        // Size of certBuf in bytes.

wc_InitDecodedCert(&decodedCert, certBuf, certBufSz, NULL);
ret = wc_ParseCert(&decodedCert, CERT_TYPE, NO_VERIFY, NULL);
if (ret != 0) {
    fprintf(stderr, "wc_ParseCert failed.\n");
}
wc_FreeDecodedCert(&decodedCert);
```

C.4.2.54 function wc_SetTimeCb

```
int wc_SetTimeCb(
    wc_time_cb f
)
```

This function registers a time callback that will be used anytime wolfSSL needs to get the current time. The prototype of the callback should be the same as the “time” function from the C standard library.

Parameters:

- **f** function to register as the time callback.

See: wc_Time

Return: 0 Returned on success.

Example

```
int ret = 0;
// Time callback prototype
time_t my_time_cb(time_t* t);
// Register it
ret = wc_SetTimeCb(my_time_cb);
if (ret != 0) {
    // failed to set time callback
}
time_t my_time_cb(time_t* t)
{
    // custom time function
}
```

C.4.2.55 function wc_Time

```
time_t wc_Time(
    time_t * t
)
```

This function gets the current time. By default, it uses the XTIME macro, which varies between platforms. The user can use a function of their choosing instead via the wc_SetTimeCb function.

Parameters:

- **t** Optional time_t pointer to populate with current time.

See: wc_SetTimeCb

Return: Time Current time returned on success.

Example

```
time_t currentTime = 0;
currentTime = wc_Time(NULL);
wc_Time(&currentTime);
```

C.4.2.56 function wc_SetCustomExtension

```
int wc_SetCustomExtension(
    Cert * cert,
    int critical,
    const char * oid,
    const byte * der,
    word32 derSz
)
```

This function injects a custom extension in to an X.509 certificate. note: The content at the address pointed to by any of the parameters that are pointers must not be modified until the certificate is generated and you have the der output. This function does NOT copy the contents to another buffer.

Parameters:

- **cert** Pointer to an initialized DecodedCert object.
- **critical** If 0, the extension will not be marked critical, otherwise it will be marked critical.
- **oid** Dot separated oid as a string. For example “1.2.840.10045.3.1.7”
- **der** The der encoding of the content of the extension.
- **derSz** The size in bytes of the der encoding.

See:

- wc_InitCert
- wc_SetUnknownExtCallback

Return:

- 0 Returned on success.
- Other negative values on failure.

Example

```
int ret = 0;
Cert newCert;
wc_InitCert(&newCert);

// Code to setup subject, public key, issuer, and other things goes here.

ret = wc_SetCustomExtension(&newCert, 1, "1.2.3.4.5",
```

```
          (const byte *)"This is a critical extension", 28);
if (ret < 0) {
    // Failed to set the extension.
}

ret = wc_SetCustomExtension(&newCert, 0, "1.2.3.4.6",
          (const byte *)"This is NOT a critical extension", 32)
if (ret < 0) {
    // Failed to set the extension.
}

// Code to sign the certificate and then write it out goes here.
```

C.4.2.57 function wc_SetUnknownExtCallback

```
int wc_SetUnknownExtCallback(
    DecodedCert * cert,
    wc_UnknownExtCallback cb
)
```

This function registers a callback that will be used anytime wolfSSL encounters an unknown X.509 extension in a certificate while parsing a certificate. The prototype of the callback should be:

Parameters:

- **cert** the DecodedCert struct that is to be associated with this callback.
- **cb** function to register as the time callback.

See:

- ParseCert
- wc_SetCustomExtension

Return:

- 0 Returned on success.
- Other negative values on failure.

Example

```
int ret = 0;
// Unknown extension callback prototype
int myUnknownExtCallback(const word16* oid, word32 oidSz, int crit,
                         const unsigned char* der, word32 derSz);

// Register it
ret = wc_SetUnknownExtCallback(cert, myUnknownExtCallback);
if (ret != 0) {
    // failed to set the callback
}

// oid: Array of integers that are the dot separated values in an oid.
// oidSz: Number of values in oid.
// crit: Whether the extension was mark critical.
// der: The der encoding of the content of the extension.
// derSz: The size in bytes of the der encoding.
int myCustomExtCallback(const word16* oid, word32 oidSz, int crit,
                        const unsigned char* der, word32 derSz) {
```

```
    // Logic to parse extension goes here.

    // NOTE: by returning zero, we are accepting this extension and
    // informing wolfSSL that it is acceptable. If you find an extension
    // that you do not find acceptable, you should return an error. The
    // standard behavior upon encountering an unknown extension with the
    // critical flag set is to return ASN_CRIT_EXT_E. For the sake of
    // brevity, this example is always accepting every extension; you
    // should use different logic.
    return 0;
}
```

C.4.2.58 function wc_CheckCertSigPubKey

```
int wc_CheckCertSigPubKey(
    const byte * cert,
    word32 certSz,
    void * heap,
    const byte * pubKey,
    word32 pubKeySz,
    int pubKeyOID
)
```

This function verifies the signature in the der form of an X.509 certificate against a public key. The public key is expected to be the full subject public key info in der form.

Parameters:

- **cert** The der encoding of the X.509 certificate.
- **certSz** The size in bytes of cert.
- **heap** A pointer to the heap used for dynamic allocation. Can be NULL.
- **pubKey** The der encoding of the public key.
- **pubKeySz** The size in bytes of pubKey.
- **pubKeyOID** OID identifying the algorithm of the public key. (ie: ECDSAk, DSAk or RSAk)

Return:

- 0 Returned on success.
- Other negative values on failure.

C.4.2.59 function wc_Asn1PrintOptions_Init

```
int wc_Asn1PrintOptions_Init(
    Asn1PrintOptions * opts
)
```

This function initializes the ASN.1 print options.

Parameters:

- **opts** The ASN.1 options for printing.

See:

- wc_Asn1PrintOptions_Set
- wc_Asn1_PrintAll

Return:

- 0 on success.

- BAD_FUNC_ARG when asn1 is NULL.

Example

```
Asn1PrintOptions opt;

// Initialize ASN.1 print options before use.
wc_Asn1PrintOptions_Init(&opt);
```

C.4.2.60 function wc_Asn1PrintOptions_Set

```
int wc_Asn1PrintOptions_Set(
    Asn1PrintOptions * opts,
    enum Asn1PrintOpt opt,
    word32 val
)
```

This function sets a print option into an ASN.1 print options object.

Parameters:

- **opts** The ASN.1 options for printing.
- **opt** An option to set value for.
- **val** The value to set.

See:

- wc_Asn1PrintOptions_Init
- wc_Asn1_PrintAll

Return:

- 0 on success.
- BAD_FUNC_ARG when asn1 is NULL.
- BAD_FUNC_ARG when val is out of range for option.

Example

```
Asn1PrintOptions opt;

// Initialize ASN.1 print options before use.
wc_Asn1PrintOptions_Init(&opt);
// Set the number of indents when printing tag name to be 1.
wc_Asn1PrintOptions_Set(&opt, ASN1_PRINT_OPT_INDENT, 1);
```

C.4.2.61 function wc_Asn1_Init

```
int wc_Asn1_Init(
    Asn1 * asn1
)
```

This function initializes an ASN.1 parsing object.

Parameters:

- **asn1** ASN.1 parse object.

See:

- wc_Asn1_SetFile
- wc_Asn1_PrintAll

Return:

- 0 on success.
- BAD_FUNC_ARG when asn1 is NULL.

Example

```
Asn1 asn1;

// Initialize ASN.1 parse object before use.
wc_Asn1_Init(&asn1);
```

C.4.2.62 function wc_Asn1_SetFile

```
int wc_Asn1_SetFile(
    Asn1 * asn1,
    XFILE file
)
```

This function sets the file to use when printing into an ASN.1 parsing object.

Parameters:

- **asn1** The ASN.1 parse object.
- **file** File to print to.

See:

- wc_Asn1_Init
- wc_Asn1_PrintAll

Return:

- 0 on success.
- BAD_FUNC_ARG when asn1 is NULL.
- BAD_FUNC_ARG when file is XBADFILE.

Example

```
Asn1 asn1;

// Initialize ASN.1 parse object before use.
wc_Asn1_Init(&asn1);
// Set standard out to be the file descriptor to write to.
wc_Asn1_SetFile(&asn1, stdout);
```

C.4.2.63 function wc_Asn1_PrintAll

```
int wc_Asn1_PrintAll(
    Asn1 * asn1,
    Asn1PrintOptions * opts,
    unsigned char * data,
    word32 len
)
```

Print all ASN.1 items.

Parameters:

- **asn1** The ASN.1 parse object.
- **opts** The ASN.1 print options.
- **data** Buffer containing BER/DER data to print.
- **len** Length of data to print in bytes.

See:

- wc_Asn1_Init
- wc_Asn1_SetFile

Return:

- 0 on success.
- BAD_FUNC_ARG when asn1 or opts is NULL.
- ASN_LEN_E when ASN.1 item's length too long.
- ASN_DEPTH_E when end offset invalid.
- ASN_PARSE_E when not all of an ASN.1 item parsed.

```
Asn1PrintOptions opts;
Asn1 asn1;
unsigned char data[] = { Initialize with DER/BER data };
word32 len = sizeof(data);

// Initialize ASN.1 print options before use.
wc_Asn1PrintOptions_Init(&opt);
// Set the number of indents when printing tag name to be 1.
wc_Asn1PrintOptions_Set(&opt, ASN1_PRINT_OPT_INDENT, 1);

// Initialize ASN.1 parse object before use.
wc_Asn1_Init(&asn1);
// Set standard out to be the file descriptor to write to.
wc_Asn1_SetFile(&asn1, stdout);
// Print all ASN.1 items in buffer with the specified print options.
wc_Asn1_PrintAll(&asn1, &opts, data, len);
```

C.4.3 Source code

```
int wc_InitCert(Cert*);

Cert* wc_CertNew(void* heap);

void  wc_CertFree(Cert* cert);

int  wc_MakeCert(Cert* cert, byte* derBuffer, word32 derSz, RsaKey* rsaKey,
                             ecc_key* eccKey, WC_RNG* rng);

int  wc_MakeCertReq(Cert* cert, byte* derBuffer, word32 derSz,
                                    RsaKey* rsaKey, ecc_key* eccKey);

int  wc_SignCert(int requestSz, int sigType, byte* derBuffer,
                 word32 derSz, RsaKey* rsaKey, ecc_key* eccKey, WC_RNG* rng);

int  wc_MakeSelfCert(Cert* cert, byte* derBuffer, word32 derSz, RsaKey* key,
                             WC_RNG* rng);

int  wc_SetIssuer(Cert* cert, const char* issuerFile);

int  wc_SetSubject(Cert* cert, const char* subjectFile);
```

```
int  wc_SetSubjectRaw(Cert* cert, const byte* der, int derSz);

int  wc_GetSubjectRaw(byte **subjectRaw, Cert *cert);

int  wc_SetAltNames(Cert* cert, const char* file);

int  wc_SetIssuerBuffer(Cert* cert, const byte* der, int derSz);

int  wc_SetIssuerRaw(Cert* cert, const byte* der, int derSz);

int  wc_SetSubjectBuffer(Cert* cert, const byte* der, int derSz);

int  wc_SetAltNamesBuffer(Cert* cert, const byte* der, int derSz);

int  wc_SetDatesBuffer(Cert* cert, const byte* der, int derSz);

int wc_SetAuthKeyIdFromPublicKey(Cert *cert, RsaKey *rsakey,
                                             ecc_key *eckey);

int wc_SetAuthKeyIdFromCert(Cert *cert, const byte *der, int derSz);

int wc_SetAuthKeyId(Cert *cert, const char* file);

int wc_SetSubjectKeyIdFromPublicKey(Cert *cert, RsaKey *rsakey,
                                                ecc_key *eckey);

int wc_SetSubjectKeyId(Cert *cert, const char* file);

int wc_SetKeyUsage(Cert *cert, const char *value);

int wc_PemPubKeyToDer(const char* fileName,
                                      unsigned char* derBuf, int derSz);

int wc_PubKeyPemToDer(const unsigned char* pem, int pemSz,
                                     unsigned char* buff, int buffSz);

int wc_PemCertToDer(const char* fileName, unsigned char* derBuf, int derSz);

int wc_DerToPem(const byte* der, word32 derSz, byte* output,
                                 word32 outputSz, int type);

int wc_DerToPemEx(const byte* der, word32 derSz, byte* output,
                                 word32 outputSz, byte *cipherIno, int type);

int wc_KeyPemToDer(const unsigned char* pem, int pemSz,
                                    unsigned char* buff, int buffSz, const char*
                                      ↪  pass);

int wc_CertPemToDer(const unsigned char* pem, int pemSz,
                    unsigned char* buff, int buffSz, int type);

int wc_GetPubKeyDerFromCert(struct DecodedCert* cert,
                                       byte* derKey, word32* derKeySz);
```

```
int wc_EccPrivateKeyDecode(const byte* input, word32* inOutIdx,
                                          ecc_key* key, word32 inSz);

int wc_EccKeyToDer(ecc_key* key, byte* output, word32 inLen);

int wc_EccPublicKeyDecode(const byte* input, word32* inOutIdx,
                          ecc_key* key, word32 inSz);

int wc_EccPublicKeyToDer(ecc_key* key, byte* output,
                                       word32 inLen, int with_AlgCurve);

int wc_EccPublicKeyToDer_ex(ecc_key* key, byte* output,
                                    word32 inLen, int with_AlgCurve, int comp);


int wc_Curve25519PrivateKeyDecode(const byte* input, word32* inOutIdx,
                                  curve25519_key* key, word32 inSz);

int wc_Curve25519PublicKeyDecode(const byte* input, word32* inOutIdx,
                                 curve25519_key* key, word32 inSz);

int wc_Curve25519KeyDecode(const byte* input, word32* inOutIdx,
                           curve25519_key* key, word32 inSz);

int wc_Curve25519PrivateKeyToDer(curve25519_key* key, byte* output,
                                 word32 inLen);

int wc_Curve25519PublicKeyToDer(curve25519_key* key, byte* output, word32
 ↪  inLen,
                                int withAlg);

int wc_Curve25519KeyToDer(curve25519_key* key, byte* output, word32 inLen,
                          int withAlg);

word32 wc_EncodeSignature(byte* out, const byte* digest,
                                      word32 digSz, int hashOID);

int wc_GetCTC_HashOID(int type);

void wc_SetCert_Free(Cert* cert);

int wc_GetPkcs8TraditionalOffset(byte* input,
                                            word32* inOutIdx, word32 sz);

int wc_CreatePKCS8Key(byte* out, word32* outSz,
        byte* key, word32 keySz, int algoID, const byte* curveOID,
        word32 oidSz);

int wc_EncryptPKCS8Key(byte* key, word32 keySz, byte* out,
        word32* outSz, const char* password, int passwordSz, int vPKCS,
        int pbeOid, int encAlgId, byte* salt, word32 saltSz, int itt,
        WC_RNG* rng, void* heap);

int wc_DecryptPKCS8Key(byte* input, word32 sz, const char* password,
```

```
        int passwordSz);

int wc_CreateEncryptedPKCS8Key(byte* key, word32 keySz, byte* out,
        word32* outSz, const char* password, int passwordSz, int vPKCS,
        int pbeOid, int encAlgId, byte* salt, word32 saltSz, int itt,
        WC_RNG* rng, void* heap);

void wc_InitDecodedCert(struct DecodedCert* cert,
    const byte* source, word32 inSz, void* heap);

int wc_ParseCert(DecodedCert* cert, int type, int verify, void* cm);

void wc_FreeDecodedCert(struct DecodedCert* cert);

int wc_SetTimeCb(wc_time_cb f);

time_t wc_Time(time_t* t);

int wc_SetCustomExtension(Cert *cert, int critical, const char *oid,
                                      const byte *der, word32 derSz);

int wc_SetUnknownExtCallback(DecodedCert* cert,
                                          wc_UnknownExtCallback cb);
int wc_CheckCertSigPubKey(const byte* cert, word32 certSz,
                                      void* heap, const byte* pubKey,
                                      word32 pubKeySz, int pubKeyOID);

int wc_Asn1PrintOptions_Init(Asn1PrintOptions* opts);

int wc_Asn1PrintOptions_Set(Asn1PrintOptions* opts, enum Asn1PrintOpt opt,
    word32 val);

int wc_Asn1_Init(Asn1* asn1);

int wc_Asn1_SetFile(Asn1* asn1, XFILE file);

int wc_Asn1_PrintAll(Asn1* asn1, Asn1PrintOptions* opts, unsigned char* data,
    word32 len);
```

C.5 dox_comments/header_files/blake2.h

C.5.1 Functions

	Name
int	**wc_InitBlake2b**(Blake2b * b2b, word32 digestSz)This function initializes a Blake2b structure for use with the Blake2 hash function.
int	**wc_Blake2bUpdate**(Blake2b * b2b, const byte * data, word32 sz)This function updates the Blake2b hash with the given input data. This function should be called after wc_InitBlake2b, and repeated until one is ready for the final hash: wc_Blake2bFinal.

	Name
int	**wc_Blake2bFinal**(Blake2b * b2b, byte * final, word32 requestSz)This function computes the Blake2b hash of the previously supplied input data. The output hash will be of length requestSz, or, if requestSz==0, the digestSz of the b2b structure. This function should be called after wc_InitBlake2b and wc_Blake2bUpdate has been processed for each piece of input data desired.

C.5.2 Functions Documentation

C.5.2.1 function wc_InitBlake2b

```
int wc_InitBlake2b(
    Blake2b * b2b,
    word32 digestSz
)
```

This function initializes a Blake2b structure for use with the Blake2 hash function.

Parameters:

- **b2b** pointer to the Blake2b structure to initialize
- **digestSz** length of the blake 2 digest to implement

See: wc_Blake2bUpdate

Return: 0 Returned upon successfully initializing the Blake2b structure and setting the digest size.

Example

```
Blake2b b2b;
// initialize Blake2b structure with 64 byte digest
wc_InitBlake2b(&b2b, 64);
```

C.5.2.2 function wc_Blake2bUpdate

```
int wc_Blake2bUpdate(
    Blake2b * b2b,
    const byte * data,
    word32 sz
)
```

This function updates the Blake2b hash with the given input data. This function should be called after wc_InitBlake2b, and repeated until one is ready for the final hash: wc_Blake2bFinal.

Parameters:

- **b2b** pointer to the Blake2b structure to update
- **data** pointer to a buffer containing the data to append
- **sz** length of the input data to append

See:

- wc_InitBlake2b
- wc_Blake2bFinal

Return:

- 0 Returned upon successfully update the Blake2b structure with the given data
- -1 Returned if there is a failure while compressing the input data

Example

```
int ret;
Blake2b b2b;
// initialize Blake2b structure with 64 byte digest
wc_InitBlake2b(&b2b, 64);

byte plain[] = { // initialize input };

ret = wc_Blake2bUpdate(&b2b, plain, sizeof(plain));
if( ret != 0) {
    // error updating blake2b
}
```

C.5.2.3 function wc_Blake2bFinal

```
int wc_Blake2bFinal(
    Blake2b * b2b,
    byte * final,
    word32 requestSz
)
```

This function computes the Blake2b hash of the previously supplied input data. The output hash will be of length requestSz, or, if requestSz==0, the digestSz of the b2b structure. This function should be called after wc_InitBlake2b and wc_Blake2bUpdate has been processed for each piece of input data desired.

Parameters:

- **b2b** pointer to the Blake2b structure to update
- **final** pointer to a buffer in which to store the blake2b hash. Should be of length requestSz
- **requestSz** length of the digest to compute. When this is zero, b2b->digestSz will be used instead

See:

- wc_InitBlake2b
- wc_Blake2bUpdate

Return:

- 0 Returned upon successfully computing the Blake2b hash
- -1 Returned if there is a failure while parsing the Blake2b hash

Example

```
int ret;
Blake2b b2b;
byte hash[64];
// initialize Blake2b structure with 64 byte digest
wc_InitBlake2b(&b2b, 64);
... // call wc_Blake2bUpdate to add data to hash

ret = wc_Blake2bFinal(&b2b, hash, 64);
if( ret != 0) {
    // error generating blake2b hash
}
```

C.5.3 Source code

```
int wc_InitBlake2b(Blake2b* b2b, word32 digestSz);

int wc_Blake2bUpdate(Blake2b* b2b, const byte* data, word32 sz);

int wc_Blake2bFinal(Blake2b* b2b, byte* final, word32 requestSz);
```

C.6 dox_comments/header_files/bn.h

C.6.1 Functions

	Name
int	**wolfSSL_BN_mod_exp**(WOLFSSL_BIGNUM * r, const WOLFSSL_BIGNUM * a, const WOLFSSL_BIGNUM * p, const WOLFSSL_BIGNUM * m, WOLFSSL_BN_CTX * ctx)This function performs the following math “r = (a^p) % m”.

C.6.2 Functions Documentation

C.6.2.1 function wolfSSL_BN_mod_exp

```
int wolfSSL_BN_mod_exp(
    WOLFSSL_BIGNUM * r,
    const WOLFSSL_BIGNUM * a,
    const WOLFSSL_BIGNUM * p,
    const WOLFSSL_BIGNUM * m,
    WOLFSSL_BN_CTX * ctx
)
```

This function performs the following math “r = (a^p) % m”.

Parameters:

- **r** structure to hold result.
- **a** value to be raised by a power.
- **p** power to raise a by.
- **m** modulus to use.
- **ctx** currently not used with wolfSSL can be NULL.

See:

- wolfSSL_BN_new
- wolfSSL_BN_free

Return:

- SSL_SUCCESS On successfully performing math operation.
- SSL_FAILURE If an error case was encountered.

Example

```
WOLFSSL_BIGNUM r,a,p,m;
int ret;
```

```
// set big number values
ret  = wolfSSL_BN_mod_exp(r, a, p, m, NULL);
// check ret value
```

C.6.3 Source code

```
int wolfSSL_BN_mod_exp(WOLFSSL_BIGNUM *r, const WOLFSSL_BIGNUM *a,
        const WOLFSSL_BIGNUM *p, const WOLFSSL_BIGNUM *m, WOLFSSL_BN_CTX *ctx);
```

C.7 dox_comments/header_files/camellia.h

C.7.1 Functions

	Name
int	**wc_CamelliaSetKey**(Camellia * cam, const byte * key, word32 len, const byte * iv)This function sets the key and initialization vector for a camellia object, initializing it for use as a cipher.
int	**wc_CamelliaSetIV**(Camellia * cam, const byte * iv)This function sets the initialization vector for a camellia object.
int	**wc_CamelliaEncryptDirect**(Camellia * cam, byte * out, const byte * in)This function does a one-block encrypt using the provided camellia object. It parses the first 16 byte block from the buffer in and stores the encrypted result in the buffer out. Before using this function, one should initialize the camellia object using wc_CamelliaSetKey.
int	**wc_CamelliaDecryptDirect**(Camellia * cam, byte * out, const byte * in)This function does a one-block decrypt using the provided camellia object. It parses the first 16 byte block from the buffer in, decrypts it, and stores the result in the buffer out. Before using this function, one should initialize the camellia object using wc_CamelliaSetKey.
int	**wc_CamelliaCbcEncrypt**(Camellia * cam, byte * out, const byte * in, word32 sz)This function encrypts the plaintext from the buffer in and stores the output in the buffer out. It performs this encryption using Camellia with Cipher Block Chaining (CBC).
int	**wc_CamelliaCbcDecrypt**(Camellia * cam, byte * out, const byte * in, word32 sz)This function decrypts the ciphertext from the buffer in and stores the output in the buffer out. It performs this decryption using Camellia with Cipher Block Chaining (CBC).

C.7.2 Functions Documentation

C.7.2.1 function wc_CamelliaSetKey

```
int wc_CamelliaSetKey(
    Camellia * cam,
    const byte * key,
    word32 len,
    const byte * iv
)
```

This function sets the key and initialization vector for a camellia object, initializing it for use as a cipher.

Parameters:

- **cam** pointer to the camellia structure on which to set the key and iv
- **key** pointer to the buffer containing the 16, 24, or 32 byte key to use for encryption and decryption
- **len** length of the key passed in
- **iv** pointer to the buffer containing the 16 byte initialization vector for use with this camellia structure

See:

- wc_CamelliaEncryptDirect
- wc_CamelliaDecryptDirect
- wc_CamelliaCbcEncrypt
- wc_CamelliaCbcDecrypt

Return:

- 0 Returned upon successfully setting the key and initialization vector
- BAD_FUNC_ARG returned if there is an error processing one of the input arguments
- MEMORY_E returned if there is an error allocating memory with XMALLOC

Example

```
Camellia cam;
byte key[32];
// initialize key
byte iv[16];
// initialize iv
if( wc_CamelliaSetKey(&cam, key, sizeof(key), iv) != 0) {
    // error initializing camellia structure
}
```

C.7.2.2 function wc_CamelliaSetIV

```
int wc_CamelliaSetIV(
    Camellia * cam,
    const byte * iv
)
```

This function sets the initialization vector for a camellia object.

Parameters:

- **cam** pointer to the camellia structure on which to set the iv
- **iv** pointer to the buffer containing the 16 byte initialization vector for use with this camellia structure

See: wc_CamelliaSetKey

Return:

- 0 Returned upon successfully setting the key and initialization vector
- BAD_FUNC_ARG returned if there is an error processing one of the input arguments

Example

```
Camellia cam;
byte iv[16];
// initialize iv
if( wc_CamelliaSetIV(&cam, iv) != 0) {
// error initializing camellia structure
}
```

C.7.2.3 function wc_CamelliaEncryptDirect

```
int wc_CamelliaEncryptDirect(
    Camellia * cam,
    byte * out,
    const byte * in
)
```

This function does a one-block encrypt using the provided camellia object. It parses the first 16 byte block from the buffer in and stores the encrypted result in the buffer out. Before using this function, one should initialize the camellia object using wc_CamelliaSetKey.

Parameters:

- **cam** pointer to the camellia structure to use for encryption
- **out** pointer to the buffer in which to store the encrypted block
- **in** pointer to the buffer containing the plaintext block to encrypt

See: wc_CamelliaDecryptDirect

Return: none No returns.

Example

```
Camellia cam;
// initialize cam structure with key and iv
byte plain[] = { // initialize with message to encrypt };
byte cipher[16];

wc_CamelliaEncryptDirect(&ca, cipher, plain);
```

C.7.2.4 function wc_CamelliaDecryptDirect

```
int wc_CamelliaDecryptDirect(
    Camellia * cam,
    byte * out,
    const byte * in
)
```

This function does a one-block decrypt using the provided camellia object. It parses the first 16 byte block from the buffer in, decrypts it, and stores the result in the buffer out. Before using this function, one should initialize the camellia object using wc_CamelliaSetKey.

Parameters:

- **cam** pointer to the camellia structure to use for encryption
- **out** pointer to the buffer in which to store the decrypted plaintext block
- **in** pointer to the buffer containing the ciphertext block to decrypt

See: wc_CamelliaEncryptDirect

Return: none No returns.

Example

```
Camellia cam;
// initialize cam structure with key and iv
byte cipher[] = { // initialize with encrypted message to decrypt };
byte decrypted[16];

wc_CamelliaDecryptDirect(&cam, decrypted, cipher);
```

C.7.2.5 function wc_CamelliaCbcEncrypt

```
int wc_CamelliaCbcEncrypt(
    Camellia * cam,
    byte * out,
    const byte * in,
    word32 sz
)
```

This function encrypts the plaintext from the buffer in and stores the output in the buffer out. It performs this encryption using Camellia with Cipher Block Chaining (CBC).

Parameters:

- **cam** pointer to the camellia structure to use for encryption
- **out** pointer to the buffer in which to store the encrypted ciphertext
- **in** pointer to the buffer containing the plaintext to encrypt
- **sz** the size of the message to encrypt

See: wc_CamelliaCbcDecrypt

Return: none No returns.

Example

```
Camellia cam;
// initialize cam structure with key and iv
byte plain[] = { // initialize with encrypted message to decrypt };
byte cipher[sizeof(plain)];

wc_CamelliaCbcEncrypt(&cam, cipher, plain, sizeof(plain));
```

C.7.2.6 function wc_CamelliaCbcDecrypt

```
int wc_CamelliaCbcDecrypt(
    Camellia * cam,
    byte * out,
    const byte * in,
    word32 sz
)
```

This function decrypts the ciphertext from the buffer in and stores the output in the buffer out. It performs this decryption using Camellia with Cipher Block Chaining (CBC).

Parameters:

- **cam** pointer to the camellia structure to use for encryption
- **out** pointer to the buffer in which to store the decrypted message
- **in** pointer to the buffer containing the encrypted ciphertext
- **sz** the size of the message to encrypt

See: wc_CamelliaCbcEncrypt

Return: none No returns.

Example

```
Camellia cam;
// initialize cam structure with key and iv
byte cipher[] = { // initialize with encrypted message to decrypt };
byte decrypted[sizeof(cipher)];

wc_CamelliaCbcDecrypt(&cam, decrypted, cipher, sizeof(cipher));
```

C.7.3 Source code

```
int  wc_CamelliaSetKey(Camellia* cam,
                                  const byte* key, word32 len, const byte* iv);

int  wc_CamelliaSetIV(Camellia* cam, const byte* iv);

int  wc_CamelliaEncryptDirect(Camellia* cam, byte* out,
                                                              const byte* in);

int  wc_CamelliaDecryptDirect(Camellia* cam, byte* out,
                                                              const byte* in);

int wc_CamelliaCbcEncrypt(Camellia* cam,
                                        byte* out, const byte* in, word32 sz);

int wc_CamelliaCbcDecrypt(Camellia* cam,
                                        byte* out, const byte* in, word32 sz);
```

C.8 dox_comments/header_files/chacha20_poly1305.h

C.8.1 Functions

	Name
int	**wc_ChaCha20Poly1305_Encrypt**(const byte inKey[CHACHA20_POLY1305_AEAD_KEYSIZE], const byte inIV[CHACHA20_POLY1305_AEAD_IV_SIZE], const byte * inAAD, const word32 inAADLen, const byte * inPlaintext, const word32 inPlaintextLen, byte * outCiphertext, byte outAuthTag[CHACHA20_POLY1305_AEAD_AUTHTAG_SIZE])This function encrypts an input message, inPlaintext, using the ChaCha20 stream cipher, into the output buffer, outCiphertext. It also performs Poly_1305 authentication (on the cipher text), and stores the generated authentication tag in the output buffer, outAuthTag.
int	**wc_ChaCha20Poly1305_Decrypt**(const byte inKey[CHACHA20_POLY1305_AEAD_KEYSIZE], const byte inIV[CHACHA20_POLY1305_AEAD_IV_SIZE], const byte * inAAD, const word32 inAADLen, const byte * inCiphertext, const word32 inCiphertextLen, const byte inAuthTag[CHACHA20_POLY1305_AEAD_AUTHTAG_SIZE], byte * outPlaintext)This function decrypts input ciphertext, inCiphertext, using the ChaCha20 stream cipher, into the output buffer, outPlaintext. It also performs Poly_1305 authentication, comparing the given inAuthTag to an authentication generated with the inAAD (arbitrary length additional authentication data). If a nonzero error code is returned, the output data, outPlaintext, is undefined. However, callers must unconditionally zeroize the output buffer to guard against leakage of cleartext data.

C.8.2 Functions Documentation

C.8.2.1 function wc_ChaCha20Poly1305_Encrypt

```
int wc_ChaCha20Poly1305_Encrypt(
    const byte inKey[CHACHA20_POLY1305_AEAD_KEYSIZE],
    const byte inIV[CHACHA20_POLY1305_AEAD_IV_SIZE],
    const byte * inAAD,
    const word32 inAADLen,
    const byte * inPlaintext,
    const word32 inPlaintextLen,
    byte * outCiphertext,
    byte outAuthTag[CHACHA20_POLY1305_AEAD_AUTHTAG_SIZE]
)
```

This function encrypts an input message, inPlaintext, using the ChaCha20 stream cipher, into the out-

put buffer, outCiphertext. It also performs Poly-1305 authentication (on the cipher text), and stores the generated authentication tag in the output buffer, outAuthTag.

Parameters:

- **inKey** pointer to a buffer containing the 32 byte key to use for encryption
- **inIv** pointer to a buffer containing the 12 byte iv to use for encryption
- **inAAD** pointer to the buffer containing arbitrary length additional authenticated data (AAD)
- **inAADLen** length of the input AAD
- **inPlaintext** pointer to the buffer containing the plaintext to encrypt
- **inPlaintextLen** the length of the plain text to encrypt
- **outCiphertext** pointer to the buffer in which to store the ciphertext
- **outAuthTag** pointer to a 16 byte wide buffer in which to store the authentication tag

See:

- wc_ChaCha20Poly1305_Decrypt
- wc_ChaCha_*
- wc_Poly1305*

Return:

- 0 Returned upon successfully encrypting the message
- BAD_FUNC_ARG returned if there is an error during the encryption process

Example

```
byte key[] = { // initialize 32 byte key };
byte iv[]  = { // initialize 12 byte key };
byte inAAD[] = { // initialize AAD };

byte plain[] = { // initialize message to encrypt };
byte cipher[sizeof(plain)];
byte authTag[16];

int ret = wc_ChaCha20Poly1305_Encrypt(key, iv, inAAD, sizeof(inAAD),
plain, sizeof(plain), cipher, authTag);

if(ret != 0) {
    // error running encrypt
}
```

C.8.2.2 function wc_ChaCha20Poly1305_Decrypt

```
int wc_ChaCha20Poly1305_Decrypt(
    const byte inKey[CHACHA20_POLY1305_AEAD_KEYSIZE],
    const byte inIV[CHACHA20_POLY1305_AEAD_IV_SIZE],
    const byte * inAAD,
    const word32 inAADLen,
    const byte * inCiphertext,
    const word32 inCiphertextLen,
    const byte inAuthTag[CHACHA20_POLY1305_AEAD_AUTHTAG_SIZE],
    byte * outPlaintext
)
```

This function decrypts input ciphertext, inCiphertext, using the ChaCha20 stream cipher, into the output buffer, outPlaintext. It also performs Poly-1305 authentication, comparing the given inAuthTag to an authentication generated with the inAAD (arbitrary length additional authentication data). If a

nonzero error code is returned, the output data, outPlaintext, is undefined. However, callers must unconditionally zeroize the output buffer to guard against leakage of cleartext data.

Parameters:

- **inKey** pointer to a buffer containing the 32 byte key to use for decryption
- **inIv** pointer to a buffer containing the 12 byte iv to use for decryption
- **inAAD** pointer to the buffer containing arbitrary length additional authenticated data (AAD)
- **inAADLen** length of the input AAD
- **inCiphertext** pointer to the buffer containing the ciphertext to decrypt
- **outCiphertextLen** the length of the ciphertext to decrypt
- **inAuthTag** pointer to the buffer containing the 16 byte digest for authentication
- **outPlaintext** pointer to the buffer in which to store the plaintext

See:

- wc_ChaCha20Poly1305_Encrypt
- wc_ChaCha_*
- wc_Poly1305*

Return:

- 0 Returned upon successfully decrypting and authenticating the message
- BAD_FUNC_ARG Returned if any of the function arguments do not match what is expected
- MAC_CMP_FAILED_E Returned if the generated authentication tag does not match the supplied inAuthTag.
- MEMORY_E Returned if internal buffer allocation failed.
- CHACHA_POLY_OVERFLOW Can be returned if input is corrupted.

Example

```
byte key[]   = { // initialize 32 byte key };
byte iv[]    = { // initialize 12 byte key };
byte inAAD[] = { // initialize AAD };

byte cipher[]    = { // initialize with received ciphertext };
byte authTag[16] = { // initialize with received authentication tag };

byte plain[sizeof(cipher)];

int ret = wc_ChaCha20Poly1305_Decrypt(key, iv, inAAD, sizeof(inAAD),
cipher, sizeof(cipher), authTag, plain);

if(ret == MAC_CMP_FAILED_E) {
    // error during authentication
} else if( ret != 0) {
    // error with function arguments
}
```

C.8.3 Source code

```
int wc_ChaCha20Poly1305_Encrypt(
                const byte inKey[CHACHA20_POLY1305_AEAD_KEYSIZE],
                const byte inIV[CHACHA20_POLY1305_AEAD_IV_SIZE],
                const byte* inAAD, const word32 inAADLen,
                const byte* inPlaintext, const word32 inPlaintextLen,
                byte* outCiphertext,
```

```
                byte outAuthTag[CHACHA20_POLY1305_AEAD_AUTHTAG_SIZE]);

int wc_ChaCha20Poly1305_Decrypt(
                const byte inKey[CHACHA20_POLY1305_AEAD_KEYSIZE],
                const byte inIV[CHACHA20_POLY1305_AEAD_IV_SIZE],
                const byte* inAAD, const word32 inAADLen,
                const byte* inCiphertext, const word32 inCiphertextLen,
                const byte inAuthTag[CHACHA20_POLY1305_AEAD_AUTHTAG_SIZE],
                byte* outPlaintext);
```

C.9 dox_comments/header_files/chacha.h

C.9.1 Functions

	Name
int	**wc_Chacha_SetIV**(ChaCha * ctx, const byte * inIv, word32 counter)This function sets the initialization vector (nonce) for a ChaCha object, initializing it for use as a cipher. It should be called after the key has been set, using wc_Chacha_SetKey. A difference nonce should be used for each round of encryption.
int	**wc_Chacha_Process**(ChaCha * ctx, byte * cipher, const byte * plain, word32 msglen)This function processes the text from the buffer input, encrypts or decrypts it, and stores the result in the buffer output.
int	**wc_Chacha_SetKey**(ChaCha * ctx, const byte * key, word32 keySz)This function sets the key for a ChaCha object, initializing it for use as a cipher. It should be called before setting the nonce with wc_Chacha_SetIV, and before using it for encryption with wc_Chacha_Process.

C.9.2 Functions Documentation

C.9.2.1 function wc_Chacha_SetIV

```
int wc_Chacha_SetIV(
    ChaCha * ctx,
    const byte * inIv,
    word32 counter
)
```

This function sets the initialization vector (nonce) for a ChaCha object, initializing it for use as a cipher. It should be called after the key has been set, using wc_Chacha_SetKey. A difference nonce should be used for each round of encryption.

Parameters:

- **ctx** pointer to the ChaCha structure on which to set the iv
- **inIv** pointer to a buffer containing the 12 byte initialization vector with which to initialize the ChaCha structure
- **counter** the value at which the block counter should start–usually zero.

See:

- wc_Chacha_SetKey
- wc_Chacha_Process

Return:

- 0 Returned upon successfully setting the initialization vector
- BAD_FUNC_ARG returned if there is an error processing the ctx input argument

Example

```
ChaCha enc;
// initialize enc with wc_Chacha_SetKey
byte iv[12];
// initialize iv
if( wc_Chacha_SetIV(&enc, iv, 0) != 0) {
    // error initializing ChaCha structure
}
```

C.9.2.2 function wc_Chacha_Process

```
int wc_Chacha_Process(
    ChaCha * ctx,
    byte * cipher,
    const byte * plain,
    word32 msglen
)
```

This function processes the text from the buffer input, encrypts or decrypts it, and stores the result in the buffer output.

Parameters:

- **ctx** pointer to the ChaCha structure on which to set the iv
- **output** pointer to a buffer in which to store the output ciphertext or decrypted plaintext
- **input** pointer to the buffer containing the input plaintext to encrypt or the input ciphertext to decrypt
- **msglen** length of the message to encrypt or the ciphertext to decrypt

See:

- wc_Chacha_SetKey
- wc_Chacha_Process

Return:

- 0 Returned upon successfully encrypting or decrypting the input
- BAD_FUNC_ARG returned if there is an error processing the ctx input argument

Example

```
ChaCha enc;
// initialize enc with wc_Chacha_SetKey and wc_Chacha_SetIV

byte plain[] = { // initialize plaintext };
byte cipher[sizeof(plain)];
if( wc_Chacha_Process(&enc, cipher, plain, sizeof(plain)) != 0) {
    // error processing ChaCha cipher
}
```

C.9.2.3 function wc_Chacha_SetKey

```
int wc_Chacha_SetKey(
    ChaCha * ctx,
    const byte * key,
    word32 keySz
)
```

This function sets the key for a ChaCha object, initializing it for use as a cipher. It should be called before setting the nonce with wc_Chacha_SetIV, and before using it for encryption with wc_Chacha_Process.

Parameters:

- **ctx** pointer to the ChaCha structure in which to set the key
- **key** pointer to a buffer containing the 16 or 32 byte key with which to initialize the ChaCha structure
- **keySz** the length of the key passed in

See:

- wc_Chacha_SetIV
- wc_Chacha_Process

Return:

- 0 Returned upon successfully setting the key
- BAD_FUNC_ARG returned if there is an error processing the ctx input argument or if the key is not 16 or 32 bytes long

Example

```
ChaCha enc;
byte key[] = { // initialize key };

if( wc_Chacha_SetKey(&enc, key, sizeof(key)) != 0) {
    // error initializing ChaCha structure
}
```

C.9.3 Source code

```
int wc_Chacha_SetIV(ChaCha* ctx, const byte* inIv, word32 counter);

int wc_Chacha_Process(ChaCha* ctx, byte* cipher, const byte* plain,
                              word32 msglen);

int wc_Chacha_SetKey(ChaCha* ctx, const byte* key, word32 keySz);
```

C.10 dox_comments/header_files/cmac.h

C.10.1 Functions

	Name
int	**wc_InitCmac**(Cmac * cmac, const byte * key, word32 keySz, int type, void * unused)Initialize the Cmac structure with defaults.

	Name
int	**wc_InitCmac_ex**(Cmac * cmac, const byte * key, word32 keySz, int type, void * unused, void * heap, int devId)Initialize the Cmac structure with defaults.
int	**wc_CmacUpdate**(Cmac * cmac, const byte * in, word32 inSz)Add Cipher-based Message Authentication Code input data.
int	**wc_CmacFinalNoFree**(Cmac * cmac, byte * out, word32 * outSz)Generate the final result using Cipher-based Message Authentication Code, deferring context cleanup.
int	**wc_CmacFinal.
int	**wc_CmacFree**(Cmac * cmac)Clean up allocations in a CMAC context.
int	**wc_AesCmacGenerate**(byte * out, word32 * outSz, const byte * in, word32 inSz, const byte * key, word32 keySz)Single shot function for generating a CMAC.
int	**wc_AesCmacVerify**(const byte * check, word32 checkSz, const byte * in, word32 inSz, const byte * key, word32 keySz)Single shot function for validating a CMAC.
int	**wc_CMAC_Grow**(Cmac * cmac, const byte * in, int inSz)Only used with WOLFSSL_HASH_KEEP when hardware requires single-shot and the updates must be cached in memory.

C.10.2 Functions Documentation

C.10.2.1 function wc_InitCmac

```
int wc_InitCmac(
    Cmac * cmac,
    const byte * key,
    word32 keySz,
    int type,
    void * unused
)
```

Initialize the Cmac structure with defaults.

Parameters:

- **cmac** pointer to the Cmac structure
- **key** key pointer
- **keySz** size of the key pointer (16, 24 or 32)
- **type** Always WC_CMAC_AES = 1
- **unused** not used, exists for potential future use around compatibility

See:

- wc_InitCmac_ex
- wc_CmacUpdate
- wc_CmacFinal
- wc_CmacFinalNoFree

- wc_CmacFree

Return: 0 on success

Example

```
Cmac cmac[1];
ret = wc_InitCmac(cmac, key, keySz, WC_CMAC_AES, NULL);
if (ret == 0) {
    ret = wc_CmacUpdate(cmac, in, inSz);
}
if (ret == 0) {
    ret = wc_CmacFinal(cmac, out, outSz);
}
```

C.10.2.2 function wc_InitCmac_ex

```
int wc_InitCmac_ex(
    Cmac * cmac,
    const byte * key,
    word32 keySz,
    int type,
    void * unused,
    void * heap,
    int devId
)
```

Initialize the Cmac structure with defaults.

Parameters:

- **cmac** pointer to the Cmac structure
- **key** key pointer
- **keySz** size of the key pointer (16, 24 or 32)
- **type** Always WC_CMAC_AES = 1
- **unused** not used, exists for potential future use around compatibility
- **heap** pointer to the heap hint used for dynamic allocation. Typically used with our static memory option. Can be NULL.
- **devId** ID to use with crypto callbacks or async hardware. Set to INVALID_DEVID (-2) if not used

See:

- wc_InitCmac_ex
- wc_CmacUpdate
- wc_CmacFinal
- wc_CmacFinalNoFree
- wc_CmacFree

Return: 0 on success

Example

```
Cmac cmac[1];
ret = wc_InitCmac_ex(cmac, key, keySz, WC_CMAC_AES, NULL, NULL, INVALID_DEVID);
if (ret == 0) {
    ret = wc_CmacUpdate(cmac, in, inSz);
}
if (ret == 0) {
    ret = wc_CmacFinal(cmac, out, &outSz);
}
```

C.10.2.3 function wc_CmacUpdate

```
int wc_CmacUpdate(
    Cmac * cmac,
    const byte * in,
    word32 inSz
)
```

Add Cipher-based Message Authentication Code input data.

Parameters:

- **cmac** pointer to the Cmac structure
- **in** input data to process
- **inSz** size of input data

See:

- wc_InitCmac
- wc_CmacFinal
- wc_CmacFinalNoFree
- wc_CmacFree

Return: 0 on success

Example

```
ret = wc_CmacUpdate(cmac, in, inSz);
```

C.10.2.4 function wc_CmacFinalNoFree

```
int wc_CmacFinalNoFree(
    Cmac * cmac,
    byte * out,
    word32 * outSz
)
```

Generate the final result using Cipher-based Message Authentication Code, deferring context cleanup.

Parameters:

- **cmac** pointer to the Cmac structure
- **out** pointer to return the result
- **outSz** pointer size of output (in/out)

See:

- wc_InitCmac
- wc_CmacFinal
- wc_CmacFinalNoFree
- wc_CmacFree

Return: 0 on success

Example

```
ret = wc_CmacFinalNoFree(cmac, out, &outSz);
(void)wc_CmacFree(cmac);
```

C.10.2.5 function wc_CmacFinal

```
int wc_CmacFinal(
    Cmac * cmac,
    byte * out,
    word32 * outSz
)
```

Generate the final result using Cipher-based Message Authentication Code, and clean up the context with wc_CmacFree().

Parameters:

- **cmac** pointer to the Cmac structure
- **out** pointer to return the result
- **outSz** pointer size of output (in/out)

See:

- wc_InitCmac
- wc_CmacFinalNoFree
- wc_CmacFinalNoFree
- wc_CmacFree

Return: 0 on success

Example

```
ret = wc_CmacFinal(cmac, out, &outSz);
```

C.10.2.6 function wc_CmacFree

```
int wc_CmacFree(
    Cmac * cmac
)
```

Clean up allocations in a CMAC context.

Parameters:

- **cmac** pointer to the Cmac structure

See:

- wc_InitCmac
- wc_CmacFinalNoFree
- wc_CmacFinal
- wc_CmacFree

Return: 0 on success

Example

```
ret = wc_CmacFinalNoFree(cmac, out, &outSz);
(void)wc_CmacFree(cmac);
```

C.10.2.7 function wc_AesCmacGenerate

```
int wc_AesCmacGenerate(
    byte * out,
    word32 * outSz,
    const byte * in,
    word32 inSz,
```

```
    const byte * key,
    word32 keySz
)
```

Single shot function for generating a CMAC.

Parameters:

- **out** pointer to return the result
- **outSz** pointer size of output (in/out)
- **in** input data to process
- **inSz** size of input data
- **key** key pointer
- **keySz** size of the key pointer (16, 24 or 32)

See: wc_AesCmacVerify

Return: 0 on success

Example

```
ret = wc_AesCmacGenerate(mac, &macSz, msg, msgSz, key, keySz);
```

C.10.2.8 function wc_AesCmacVerify

```
int wc_AesCmacVerify(
    const byte * check,
    word32 checkSz,
    const byte * in,
    word32 inSz,
    const byte * key,
    word32 keySz
)
```

Single shot function for validating a CMAC.

Parameters:

- **check** pointer to return the result
- **checkSz** size of checkout buffer
- **in** input data to process
- **inSz** size of input data
- **key** key pointer
- **keySz** size of the key pointer (16, 24 or 32)

See: wc_AesCmacGenerate

Return: 0 on success

Example

```
ret = wc_AesCmacVerify(mac, macSz, msg, msgSz, key, keySz);
```

C.10.2.9 function wc_CMAC_Grow

```
int wc_CMAC_Grow(
    Cmac * cmac,
    const byte * in,
    int inSz
)
```

Only used with WOLFSSL_HASH_KEEP when hardware requires single-shot and the updates must be cached in memory.

Parameters:

- **in** input data to process
- **inSz** size of input data

Return: 0 on success

Example

```
ret = wc_CMAC_Grow(cmac, in, inSz)
```

C.10.3 Source code

```
int wc_InitCmac(Cmac* cmac,
                const byte* key, word32 keySz,
                int type, void* unused);

int wc_InitCmac_ex(Cmac* cmac,
                const byte* key, word32 keySz,
                int type, void* unused, void* heap, int devId);

int wc_CmacUpdate(Cmac* cmac,
                  const byte* in, word32 inSz);


int wc_CmacFinalNoFree(Cmac* cmac,
                 byte* out, word32* outSz);

int wc_CmacFinal(Cmac* cmac,
                 byte* out, word32* outSz);

int wc_CmacFree(Cmac* cmac);

int wc_AesCmacGenerate(byte* out, word32* outSz,
                       const byte* in, word32 inSz,
                       const byte* key, word32 keySz);

int wc_AesCmacVerify(const byte* check, word32 checkSz,
                     const byte* in, word32 inSz,
                     const byte* key, word32 keySz);


int wc_CMAC_Grow(Cmac* cmac, const byte* in, int inSz);
```

C.11 dox_comments/header_files/coding.h

C.11.1 Functions

	Name
int	**Base64_Decode**(const byte * in, word32 inLen, byte * out, word32 * outLen)This function decodes the given Base64 encoded input, in, and stores the result in the output buffer out. It also sets the size written to the output buffer in the variable outLen.
int	**Base64_Encode**(const byte * in, word32 inLen, byte * out, word32 * outLen)This function encodes the given input, in, and stores the Base64 encoded result in the output buffer out. It writes the data with the traditional ‘ ’ line endings, instead of escaped %0A line endings. Upon successfully completing, this function also sets outLen to the number of bytes written to the output buffer.
int	**Base64_EncodeEsc**(const byte * in, word32 inLen, byte * out, word32 * outLen)This function encodes the given input, in, and stores the Base64 encoded result in the output buffer out. It writes the data with %0A escaped line endings instead of ‘ ’ line endings. Upon successfully completing, this function also sets outLen to the number of bytes written to the output buffer.
int	**Base64_Encode_NoNl**(const byte * in, word32 inLen, byte * out, word32 * outLen)This function encodes the given input, in, and stores the Base64 encoded result in the output buffer out. It writes the data with no new lines. Upon successfully completing, this function also sets outLen to the number of bytes written to the output buffer.
int	**Base16_Decode**(const byte * in, word32 inLen, byte * out, word32 * outLen)This function decodes the given Base16 encoded input, in, and stores the result in the output buffer out. It also sets the size written to the output buffer in the variable outLen.
int	**Base16_Encode**(const byte * in, word32 inLen, byte * out, word32 * outLen)Encode input to base16 output.

C.11.2 Functions Documentation

C.11.2.1 function Base64_Decode

```
int Base64_Decode(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 * outLen
)
```

This function decodes the given Base64 encoded input, in, and stores the result in the output buffer out. It also sets the size written to the output buffer in the variable outLen.

Parameters:

- **in** pointer to the input buffer to decode
- **inLen** length of the input buffer to decode
- **out** pointer to the output buffer in which to store the decoded message
- **outLen** pointer to the length of the output buffer. Updated with the bytes written at the end of the function call

See:

- Base64_Encode
- Base16_Decode

Return:

- 0 Returned upon successfully decoding the Base64 encoded input
- BAD_FUNC_ARG Returned if the output buffer is too small to store the decoded input
- ASN_INPUT_E Returned if a character in the input buffer falls outside of the Base64 range ([A-Za-z0-9+/=]) or if there is an invalid line ending in the Base64 encoded input

Example

```
byte encoded[] = { // initialize text to decode };
byte decoded[sizeof(encoded)];
// requires at least (sizeof(encoded) * 3 + 3) / 4 room

int outLen = sizeof(decoded);

if( Base64_Decode(encoded,sizeof(encoded), decoded, &outLen) != 0 ) {
    // error decoding input buffer
}
```

C.11.2.2 function Base64_Encode

```
int Base64_Encode(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 * outLen
)
```

This function encodes the given input, in, and stores the Base64 encoded result in the output buffer out. It writes the data with the traditional ‘ ’ line endings, instead of escaped %0A line endings. Upon successfully completing, this function also sets outLen to the number of bytes written to the output buffer.

Parameters:

- **in** pointer to the input buffer to encode
- **inLen** length of the input buffer to encode
- **out** pointer to the output buffer in which to store the encoded message
- **outLen** pointer to the length of the output buffer in which to store the encoded message

See:

- Base64_EncodeEsc
- Base64_Decode

Return:

- 0 Returned upon successfully decoding the Base64 encoded input
- BAD_FUNC_ARG Returned if the output buffer is too small to store the encoded input
- BUFFER_E Returned if the output buffer runs out of room while encoding

Example

```
byte plain[] = { // initialize text to encode };
byte encoded[MAX_BUFFER_SIZE];

int outLen = sizeof(encoded);

if( Base64_Encode(plain, sizeof(plain), encoded, &outLen) != 0 ) {
    // error encoding input buffer
}
```

C.11.2.3 function Base64_EncodeEsc

```
int Base64_EncodeEsc(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 * outLen
)
```

This function encodes the given input, in, and stores the Base64 encoded result in the output buffer out. It writes the data with %0A escaped line endings instead of ‘ ’ line endings. Upon successfully completing, this function also sets outLen to the number of bytes written to the output buffer.

Parameters:

- **in** pointer to the input buffer to encode
- **inLen** length of the input buffer to encode
- **out** pointer to the output buffer in which to store the encoded message
- **outLen** pointer to the length of the output buffer in which to store the encoded message

See:

- Base64_Encode
- Base64_Decode

Return:

- 0 Returned upon successfully decoding the Base64 encoded input
- BAD_FUNC_ARG Returned if the output buffer is too small to store the encoded input
- BUFFER_E Returned if the output buffer runs out of room while encoding
- ASN_INPUT_E Returned if there is an error processing the decode on the input message

Example

```
byte plain[] = { // initialize text to encode };
byte encoded[MAX_BUFFER_SIZE];

int outLen = sizeof(encoded);

if( Base64_EncodeEsc(plain, sizeof(plain), encoded, &outLen) != 0 ) {
    // error encoding input buffer
}
```

C.11.2.4 function Base64_Encode_NoNl

```
int Base64_Encode_NoNl(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 * outLen
)
```

This function encodes the given input, in, and stores the Base64 encoded result in the output buffer out. It writes the data with no new lines. Upon successfully completing, this function also sets outLen to the number of bytes written to the output buffer.

Parameters:

- **in** pointer to the input buffer to encode
- **inLen** length of the input buffer to encode
- **out** pointer to the output buffer in which to store the encoded message
- **outLen** pointer to the length of the output buffer in which to store the encoded message

See:

- Base64_Encode
- Base64_Decode

Return:

- 0 Returned upon successfully decoding the Base64 encoded input
- BAD_FUNC_ARG Returned if the output buffer is too small to store the encoded input
- BUFFER_E Returned if the output buffer runs out of room while encoding
- ASN_INPUT_E Returned if there is an error processing the decode on the input message

Example

```
byte plain[] = { // initialize text to encode };
byte encoded[MAX_BUFFER_SIZE];
int outLen = sizeof(encoded);
if( Base64_Encode_NoNl(plain, sizeof(plain), encoded, &outLen) != 0 ) {
    // error encoding input buffer
}
```

C.11.2.5 function Base16_Decode

```
int Base16_Decode(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 * outLen
)
```

This function decodes the given Base16 encoded input, in, and stores the result in the output buffer out. It also sets the size written to the output buffer in the variable outLen.

Parameters:

- **in** pointer to the input buffer to decode
- **inLen** length of the input buffer to decode
- **out** pointer to the output buffer in which to store the decoded message
- **outLen** pointer to the length of the output buffer. Updated with the bytes written at the end of the function call

See:

- Base64_Encode
- Base64_Decode
- Base16_Encode

Return:

- 0 Returned upon successfully decoding the Base16 encoded input
- BAD_FUNC_ARG Returned if the output buffer is too small to store the decoded input or if the input length is not a multiple of two
- ASN_INPUT_E Returned if a character in the input buffer falls outside of the Base16 range ([0-9A-F])

Example

```
byte encoded[] = { // initialize text to decode };
byte decoded[sizeof(encoded)];
int outLen = sizeof(decoded);

if( Base16_Decode(encoded,sizeof(encoded), decoded, &outLen) != 0 ) {
    // error decoding input buffer
}
```

C.11.2.6 function Base16_Encode

```
int Base16_Encode(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 * outLen
)
```

Encode input to base16 output.

Parameters:

- **in** Pointer to input buffer to be encoded.
- **inLen** Length of input buffer.
- **out** Pointer to output buffer.
- **outLen** Length of output buffer. Is set to len of encoded output.

See:

- Base64_Encode
- Base64_Decode
- Base16_Decode

Return:

- 0 Success
- BAD_FUNC_ARG Returns if in, out, or outLen is null or if outLen is less than 2 times inLen plus 1.

Example

```
byte in[] = { // Contents of something to be encoded };
byte out[NECESSARY_OUTPUT_SIZE];
word32 outSz = sizeof(out);

if(Base16_Encode(in, sizeof(in), out, &outSz) != 0)
{
```

```
    // Handle encode error
}
```

C.11.3 Source code

```
int Base64_Decode(const byte* in, word32 inLen, byte* out,
                                word32* outLen);

int Base64_Encode(const byte* in, word32 inLen, byte* out,
                                word32* outLen);

int Base64_EncodeEsc(const byte* in, word32 inLen, byte* out,
                                word32* outLen);

int Base64_Encode_NoNl(const byte* in, word32 inLen, byte* out,
                                word32* outLen);

int Base16_Decode(const byte* in, word32 inLen, byte* out, word32* outLen);

int Base16_Encode(const byte* in, word32 inLen, byte* out, word32* outLen);
```

C.12 dox_comments/header_files/compress.h

C.12.1 Functions

	Name
int	**wc_Compress**(byte * out, word32 outSz, const byte * in, word32 inSz, word32 flags)This function compresses the given input data using Huffman coding and stores the output in out. Note that the output buffer should still be larger than the input buffer because there exists a certain input for which there will be no compression possible, which will still require a lookup table. It is recommended that one allocate srcSz + 0.1% + 12 for the output buffer.
int	**wc_DeCompress**(byte * out, word32 outSz, const byte * in, word32 inSz)This function decompresses the given compressed data using Huffman coding and stores the output in out.

C.12.2 Functions Documentation

C.12.2.1 function wc_Compress

```
int wc_Compress(
    byte * out,
    word32 outSz,
    const byte * in,
    word32 inSz,
```

```
    word32 flags
)
```

This function compresses the given input data using Huffman coding and stores the output in out. Note that the output buffer should still be larger than the input buffer because there exists a certain input for which there will be no compression possible, which will still require a lookup table. It is recommended that one allocate srcSz + 0.1% + 12 for the output buffer.

Parameters:

- **out** pointer to the output buffer in which to store the compressed data
- **outSz** size available in the output buffer for storage
- **in** pointer to the buffer containing the message to compress
- **inSz** size of the input message to compress
- **flags** flags to control how compression operates. Use 0 for normal decompression

See: wc_DeCompress

Return:

- On successfully compressing the input data, returns the number of bytes stored in the output buffer
- COMPRESS_INIT_E Returned if there is an error initializing the stream for compression
- COMPRESS_E Returned if an error occurs during compression

Example

```
byte message[] = { // initialize text to compress };
byte compressed[(sizeof(message) + sizeof(message) * .001 + 12 )];
// Recommends at least srcSz + .1% + 12

if( wc_Compress(compressed, sizeof(compressed), message, sizeof(message),
0) != 0){
    // error compressing data
}
```

C.12.2.2 function wc_DeCompress

```
int wc_DeCompress(
    byte * out,
    word32 outSz,
    const byte * in,
    word32 inSz
)
```

This function decompresses the given compressed data using Huffman coding and stores the output in out.

Parameters:

- **out** pointer to the output buffer in which to store the decompressed data
- **outSz** size available in the output buffer for storage
- **in** pointer to the buffer containing the message to decompress
- **inSz** size of the input message to decompress

See: wc_Compress

Return:

- Success On successfully decompressing the input data, returns the number of bytes stored in the output buffer

- COMPRESS_INIT_E: Returned if there is an error initializing the stream for compression
- COMPRESS_E: Returned if an error occurs during compression

Example

```
byte compressed[] = { // initialize compressed message };
byte decompressed[MAX_MESSAGE_SIZE];

if( wc_DeCompress(decompressed, sizeof(decompressed),
compressed, sizeof(compressed)) != 0 ) {
    // error decompressing data
}
```

C.12.3 Source code

```
int wc_Compress(byte* out, word32 outSz, const byte* in, word32 inSz, word32
 ↪  flags);

int wc_DeCompress(byte* out, word32 outSz, const byte* in, word32 inSz);
```

C.13 dox_comments/header_files/cryptocb.h

C.13.1 Functions

	Name
int	**wc_CryptoCb_RegisterDevice**(int devId, CryptoDevCallbackFunc cb, void * ctx)This function registers a unique device identifier (devID) and callback function for offloading crypto operations to external hardware such as Key Store, Secure Element, HSM, PKCS11 or TPM.
void	**wc_CryptoCb_UnRegisterDevice**(int devId)This function un_registers a unique device identifier (devID) callback function.

C.13.2 Functions Documentation

C.13.2.1 function wc_CryptoCb_RegisterDevice

```
int wc_CryptoCb_RegisterDevice(
    int devId,
    CryptoDevCallbackFunc cb,
    void * ctx
)
```

This function registers a unique device identifier (devID) and callback function for offloading crypto operations to external hardware such as Key Store, Secure Element, HSM, PKCS11 or TPM.

Parameters:

- **devId** any unique value, not -2 (INVALID_DEVID)
- **cb** a callback function with prototype: typedef int (*CryptoDevCallbackFunc)(int devId, wc_CryptoInfo* info, void* ctx);

See:

- wc_CryptoCb_UnRegisterDevice
- wolfSSL_SetDevId
- wolfSSL_CTX_SetDevId

Return:

- CRYPTOCB_UNAVAILABLE to fallback to using software crypto
- 0 for success
- negative value for failure

For STSAFE with Crypto Callbacks example see wolfcrypt/src/port/st/stsafe.c and the wolfSSL_STSAFE_CryptoDevCb function.

For TPM based crypto callbacks example see the wolfTPM2_CryptoDevCb function in wolfTPM src/tpm2_wrap.c

Example

```
#include <wolfssl/wolfcrypt/settings.h>
#include <wolfssl/wolfcrypt/cryptocb.h>
static int myCryptoCb_Func(int devId, wc_CryptoInfo* info, void* ctx)
{
    int ret = CRYPTOCB_UNAVAILABLE;

    if (info->algo_type == WC_ALGO_TYPE_PK) {
    #ifndef NO_RSA
        if (info->pk.type == WC_PK_TYPE_RSA) {
            switch (info->pk.rsa.type) {
                case RSA_PUBLIC_ENCRYPT:
                case RSA_PUBLIC_DECRYPT:
                    // RSA public op
                    ret = wc_RsaFunction(
                        info->pk.rsa.in, info->pk.rsa.inLen,
                        info->pk.rsa.out, info->pk.rsa.outLen,
                        info->pk.rsa.type, info->pk.rsa.key,
                        info->pk.rsa.rng);
                    break;
                case RSA_PRIVATE_ENCRYPT:
                case RSA_PRIVATE_DECRYPT:
                    // RSA private op
                    ret = wc_RsaFunction(
                        info->pk.rsa.in, info->pk.rsa.inLen,
                        info->pk.rsa.out, info->pk.rsa.outLen,
                        info->pk.rsa.type, info->pk.rsa.key,
                        info->pk.rsa.rng);
                    break;
            }
        }
    #endif
    #ifdef HAVE_ECC
        if (info->pk.type == WC_PK_TYPE_ECDSA_SIGN) {
            // ECDSA
            ret = wc_ecc_sign_hash(
                info->pk.eccsign.in, info->pk.eccsign.inlen,
                info->pk.eccsign.out, info->pk.eccsign.outlen,
                info->pk.eccsign.rng, info->pk.eccsign.key);
```

```
        }
    #endif
    #ifdef HAVE_ED25519
        if (info->pk.type == WC_PK_TYPE_ED25519_SIGN) {
            // ED25519 sign
            ret = wc_ed25519_sign_msg_ex(
                info->pk.ed25519sign.in, info->pk.ed25519sign.inLen,
                info->pk.ed25519sign.out, info->pk.ed25519sign.outLen,
                info->pk.ed25519sign.key, info->pk.ed25519sign.type,
                info->pk.ed25519sign.context,
                info->pk.ed25519sign.contextLen);
        }
    #endif
    }
    return ret;
}

int devId = 1;
wc_CryptoCb_RegisterDevice(devId, myCryptoCb_Func, &myCtx);
wolfSSL_CTX_SetDevId(ctx, devId);
```

C.13.2.2 function wc_CryptoCb_UnRegisterDevice

```
void wc_CryptoCb_UnRegisterDevice(
    int devId
)
```

This function un-registers a unique device identifier (devID) callback function.

Parameters:

- **devId** any unique value, not -2 (INVALID_DEVID)

See:

- wc_CryptoCb_RegisterDevice
- wolfSSL_SetDevId
- wolfSSL_CTX_SetDevId

Return: none No returns.

Example

```
wc_CryptoCb_UnRegisterDevice(devId);
devId = INVALID_DEVID;
wolfSSL_CTX_SetDevId(ctx, devId);
```

C.13.3 Source code

```
int  wc_CryptoCb_RegisterDevice(int devId, CryptoDevCallbackFunc cb, void*
 ↪  ctx);

void wc_CryptoCb_UnRegisterDevice(int devId);
```

C.14 dox_comments/header_files/curve25519.h

C.14.1 Functions

	Name
int	**wc_curve25519_make_key.
int	**wc_curve25519_shared_secret**(curve25519_key * private_key, curve25519_key * public_key, byte * out, word32 * outlen)This function computes a shared secret key given a secret private key and a received public key. It stores the generated secret key in the buffer out and assigns the variable of the secret key to outlen. Only supports big endian.
int	**wc_curve25519_shared_secret_ex**(curve25519_key * private_key, curve25519_key * public_key, byte * out, word32 * outlen, int endian)This function computes a shared secret key given a secret private key and a received public key. It stores the generated secret key in the buffer out and assigns the variable of the secret key to outlen. Supports both big and little endian.
int	**wc_curve25519_init**(curve25519_key * key)This function initializes a Curve25519 key. It should be called before generating a key for the structure.
void	**wc_curve25519_free**(curve25519_key * key)This function frees a Curve25519 object.
int	**wc_curve25519_import_private**(const byte * priv, word32 privSz, curve25519_key * key)This function imports a curve25519 private key only. (Big endian).
int	**wc_curve25519_import_private_ex**(const byte * priv, word32 privSz, curve25519_key * key, int endian)curve25519 private key import only. (Big or Little endian).
int	**wc_curve25519_import_private_raw**(const byte * priv, word32 privSz, const byte * pub, word32 pubSz, curve25519_key * key)This function imports a public-private key pair into a curve25519_key structure. Big endian only.
int	**wc_curve25519_import_private_raw_ex**(const byte * priv, word32 privSz, const byte * pub, word32 pubSz, curve25519_key * key, int endian)This function imports a public-private key pair into a curve25519_key structure. Supports both big and little endian.
int	**wc_curve25519_export_private_raw**(curve25519_key * key, byte * out, word32 * outLen)This function exports a private key from a curve25519_key structure and stores it in the given out buffer. It also sets outLen to be the size of the exported key. Big Endian only.

	Name
int	**wc_curve25519_export_private_raw_ex**(curve25519_key * key, byte * out, word32 * outLen, int endian)This function exports a private key from a curve25519_key structure and stores it in the given out buffer. It also sets outLen to be the size of the exported key. Can specify whether it's big or little endian.
int	**wc_curve25519_import_public**(const byte * in, word32 inLen, curve25519_key * key)This function imports a public key from the given in buffer and stores it in the curve25519_key structure.
int	**wc_curve25519_import_public_ex**(const byte * in, word32 inLen, curve25519_key * key, int endian)This function imports a public key from the given in buffer and stores it in the curve25519_key structure.
int	**wc_curve25519_check_public**(const byte * pub, word32 pubSz, int endian)This function checks that a public key buffer holds a valid Curve25519 key value given the endian ordering.
int	**wc_curve25519_export_public**(curve25519_key * key, byte * out, word32 * outLen)This function exports a public key from the given key structure and stores the result in the out buffer. Big endian only.
int	**wc_curve25519_export_public_ex**(curve25519_key * key, byte * out, word32 * outLen, int endian)This function exports a public key from the given key structure and stores the result in the out buffer. Supports both big and little endian.
int	**wc_curve25519_export_key_raw**(curve25519_key * key, byte * priv, word32 * privSz, byte * pub, word32 * pubSz)Export Curve25519 key pair. Big endian only.
int	**wc_curve25519_export_key_raw_ex**(curve25519_key * key, byte * priv, word32 * privSz, byte * pub, word32 * pubSz, int endian)Export curve25519 key pair. Big or little endian.
int	**wc_curve25519_size**(curve25519_key * key)This function returns the key size of the given key structure.

C.14.2 Functions Documentation

C.14.2.1 function wc_curve25519_make_key

```
int wc_curve25519_make_key(
    WC_RNG * rng,
    int keysize,
```

```
    curve25519_key * key
)
```

This function generates a Curve25519 key using the given random number generator, rng, of the size given (keysize), and stores it in the given curve25519_key structure. It should be called after the key structure has been initialized through wc_curve25519_init().

Parameters:

- **rng** Pointer to the RNG object used to generate the ecc key.
- **keysize** Size of the key to generate. Must be 32 bytes for curve25519.
- **key** Pointer to the curve25519_key structure in which to store the generated key.

See: wc_curve25519_init

Return:

- 0 Returned on successfully generating the key and and storing it in the given curve25519_key structure.
- ECC_BAD_ARG_E Returned if the input keysize does not correspond to the keysize for a curve25519 key (32 bytes).
- RNG_FAILURE_E Returned if the rng internal status is not DRBG_OK or if there is in generating the next random block with rng.
- BAD_FUNC_ARG Returned if any of the input parameters passed in are NULL.

Example

```
int ret;

curve25519_key key;
wc_curve25519_init(&key); // initialize key
WC_RNG rng;
wc_InitRng(&rng); // initialize random number generator

ret = wc_curve25519_make_key(&rng, 32, &key);
if (ret != 0) {
    // error making Curve25519 key
}
```

C.14.2.2 function wc_curve25519_shared_secret

```
int wc_curve25519_shared_secret(
    curve25519_key * private_key,
    curve25519_key * public_key,
    byte * out,
    word32 * outlen
)
```

This function computes a shared secret key given a secret private key and a received public key. It stores the generated secret key in the buffer out and assigns the variable of the secret key to outlen. Only supports big endian.

Parameters:

- **private_key** Pointer to the curve25519_key structure initialized with the user's private key.
- **public_key** Pointer to the curve25519_key structure containing the received public key.
- **out** Pointer to a buffer in which to store the 32 byte computed secret key.
- **outlen** Pointer in which to store the length written to the output buffer.

See:

- wc_curve25519_init
- wc_curve25519_make_key
- wc_curve25519_shared_secret_ex

Return:

- 0 Returned on successfully computing a shared secret key.
- BAD_FUNC_ARG Returned if any of the input parameters passed in are NULL.
- ECC_BAD_ARG_E Returned if the first bit of the public key is set, to avoid implementation fingerprinting.

Example

```
int ret;

byte sharedKey[32];
word32 keySz;
curve25519_key privKey, pubKey;
// initialize both keys

ret = wc_curve25519_shared_secret(&privKey, &pubKey, sharedKey, &keySz);
if (ret != 0) {
    // error generating shared key
}
```

C.14.2.3 function wc_curve25519_shared_secret_ex

```
int wc_curve25519_shared_secret_ex(
    curve25519_key * private_key,
    curve25519_key * public_key,
    byte * out,
    word32 * outlen,
    int endian
)
```

This function computes a shared secret key given a secret private key and a received public key. It stores the generated secret key in the buffer out and assigns the variable of the secret key to outlen. Supports both big and little endian.

Parameters:

- **private_key** Pointer to the curve25519_key structure initialized with the user's private key.
- **public_key** Pointer to the curve25519_key structure containing the received public key.
- **out** Pointer to a buffer in which to store the 32 byte computed secret key.
- **pinout]** outlen Pointer in which to store the length written to the output buffer.
- **endian** EC25519_BIG_ENDIAN or EC25519_LITTLE_ENDIAN to set which form to use.

See:

- wc_curve25519_init
- wc_curve25519_make_key
- wc_curve25519_shared_secret

Return:

- 0 Returned on successfully computing a shared secret key.
- BAD_FUNC_ARG Returned if any of the input parameters passed in are NULL.
- ECC_BAD_ARG_E Returned if the first bit of the public key is set, to avoid implementation fingerprinting.

Example

```
int ret;

byte sharedKey[32];
word32 keySz;

curve25519_key privKey, pubKey;
// initialize both keys

ret = wc_curve25519_shared_secret_ex(&privKey, &pubKey, sharedKey, &keySz,
        EC25519_BIG_ENDIAN);
if (ret != 0) {
    // error generating shared key
}
```

C.14.2.4 function wc_curve25519_init

```
int wc_curve25519_init(
    curve25519_key * key
)
```

This function initializes a Curve25519 key. It should be called before generating a key for the structure.

Parameters:

- **key** Pointer to the curve25519_key structure to initialize.

See: wc_curve25519_make_key

Return:

- 0 Returned on successfully initializing the curve25519_key structure.
- BAD_FUNC_ARG Returned when key is NULL.

Example

```
curve25519_key key;
wc_curve25519_init(&key); // initialize key
// make key and proceed to encryption
```

C.14.2.5 function wc_curve25519_free

```
void wc_curve25519_free(
    curve25519_key * key
)
```

This function frees a Curve25519 object.

Parameters:

- **key** Pointer to the key object to free.

See:

- wc_curve25519_init
- wc_curve25519_make_key

Example

```
curve25519_key privKey;
// initialize key, use it to generate shared secret key
wc_curve25519_free(&privKey);
```

C.14.2.6 function wc_curve25519_import_private

```
int wc_curve25519_import_private(
    const byte * priv,
    word32 privSz,
    curve25519_key * key
)
```

This function imports a curve25519 private key only. (Big endian).

Parameters:

- **priv** Pointer to a buffer containing the private key to import.
- **privSz** Length of the private key to import.
- **key** Pointer to the structure in which to store the imported key.

See:

- wc_curve25519_import_private_ex
- wc_curve25519_size

Return:

- 0 Returned on successfully importing private key.
- BAD_FUNC_ARG Returns if key or priv is null.
- ECC_BAD_ARG_E Returns if privSz is not equal to CURVE25519_KEY_SIZE.

Example

```
int ret;

byte priv[] = { Contents of private key };
curve25519_key key;
wc_curve25519_init(&key);

ret = wc_curve25519_import_private(priv, sizeof(priv), &key);
if (ret != 0) {
    // error importing keys
}
```

C.14.2.7 function wc_curve25519_import_private_ex

```
int wc_curve25519_import_private_ex(
    const byte * priv,
    word32 privSz,
    curve25519_key * key,
    int endian
)
```

curve25519 private key import only. (Big or Little endian).

Parameters:

- **priv** Pointer to a buffer containing the private key to import.
- **privSz** Length of the private key to import.
- **key** Pointer to the structure in which to store the imported key.

- **endian** EC25519_BIG_ENDIAN or EC25519_LITTLE_ENDIAN to set which form to use.

See:

- wc_curve25519_import_private
- wc_curve25519_size

Return:

- 0 Returned on successfully importing private key.
- BAD_FUNC_ARG Returns if key or priv is null.
- ECC_BAD_ARG_E Returns if privSz is not equal to CURVE25519_KEY_SIZE.

Example

```
int ret;

byte priv[] = { // Contents of private key };
curve25519_key key;
wc_curve25519_init(&key);

ret = wc_curve25519_import_private_ex(priv, sizeof(priv), &key,
        EC25519_BIG_ENDIAN);
if (ret != 0) {
    // error importing keys
}
```

C.14.2.8 function wc_curve25519_import_private_raw

```
int wc_curve25519_import_private_raw(
    const byte * priv,
    word32 privSz,
    const byte * pub,
    word32 pubSz,
    curve25519_key * key
)
```

This function imports a public-private key pair into a curve25519_key structure. Big endian only.

Parameters:

- **priv** Pointer to a buffer containing the private key to import.
- **privSz** Length of the private key to import.
- **pub** Pointer to a buffer containing the public key to import.
- **pubSz** Length of the public key to import.
- **key** Pointer to the structure in which to store the imported keys.

See:

- wc_curve25519_init
- wc_curve25519_make_key
- wc_curve25519_import_public
- wc_curve25519_export_private_raw

Return:

- 0 Returned on importing into the curve25519_key structure
- BAD_FUNC_ARG Returns if any of the input parameters are null.
- ECC_BAD_ARG_E Returned if the input key's key size does not match the public or private key sizes.

Example

```
int ret;

byte priv[32];
byte pub[32];
// initialize with public and private keys
curve25519_key key;

wc_curve25519_init(&key);
// initialize key

ret = wc_curve25519_import_private_raw(&priv, sizeof(priv), pub,
        sizeof(pub), &key);
if (ret != 0) {
    // error importing keys
}
```

C.14.2.9 function wc_curve25519_import_private_raw_ex

```
int wc_curve25519_import_private_raw_ex(
    const byte * priv,
    word32 privSz,
    const byte * pub,
    word32 pubSz,
    curve25519_key * key,
    int endian
)
```

This function imports a public-private key pair into a curve25519_key structure. Supports both big and little endian.

Parameters:

- **priv** Pointer to a buffer containing the private key to import.
- **privSz** Length of the private key to import.
- **pub** Pointer to a buffer containing the public key to import.
- **pubSz** Length of the public key to import.
- **key** Pointer to the structure in which to store the imported keys.
- **endian** EC25519_BIG_ENDIAN or EC25519_LITTLE_ENDIAN to set which form to use.

See:

- wc_curve25519_init
- wc_curve25519_make_key
- wc_curve25519_import_public
- wc_curve25519_export_private_raw
- wc_curve25519_import_private_raw

Return:

- 0 Returned on importing into the curve25519_key structure
- BAD_FUNC_ARG Returns if any of the input parameters are null.
- ECC_BAD_ARG_E Returned if or the input key's key size does not match the public or private key sizes

Example

```
int ret;
byte priv[32];
byte pub[32];
// initialize with public and private keys
curve25519_key key;

wc_curve25519_init(&key);
// initialize key

ret = wc_curve25519_import_private_raw_ex(&priv, sizeof(priv), pub,
        sizeof(pub), &key, EC25519_BIG_ENDIAN);
if (ret != 0) {
    // error importing keys
}
```

C.14.2.10 function wc_curve25519_export_private_raw

```
int wc_curve25519_export_private_raw(
    curve25519_key * key,
    byte * out,
    word32 * outLen
)
```

This function exports a private key from a curve25519_key structure and stores it in the given out buffer. It also sets outLen to be the size of the exported key. Big Endian only.

Parameters:

- **key** Pointer to the structure from which to export the key.
- **out** Pointer to the buffer in which to store the exported key.
- **outLen** On in, is the size of the out in bytes. On out, will store the bytes written to the output buffer.

See:

- wc_curve25519_init
- wc_curve25519_make_key
- wc_curve25519_import_private_raw
- wc_curve25519_export_private_raw_ex

Return:

- 0 Returned on successfully exporting the private key from the curve25519_key structure.
- BAD_FUNC_ARG Returned if any input parameters are NULL.
- ECC_BAD_ARG_E Returned if wc_curve25519_size() is not equal to key.

Example

```
int ret;
byte priv[32];
int privSz;

curve25519_key key;
// initialize and make key

ret = wc_curve25519_export_private_raw(&key, priv, &privSz);
if (ret != 0) {
    // error exporting key
}
```

C.14.2.11 function wc_curve25519_export_private_raw_ex

```
int wc_curve25519_export_private_raw_ex(
    curve25519_key * key,
    byte * out,
    word32 * outLen,
    int endian
)
```

This function exports a private key from a curve25519_key structure and stores it in the given out buffer. It also sets outLen to be the size of the exported key. Can specify whether it's big or little endian.

Parameters:

- **key** Pointer to the structure from which to export the key.
- **out** Pointer to the buffer in which to store the exported key.
- **outLen** On in, is the size of the out in bytes. On out, will store the bytes written to the output buffer.
- **endian** EC25519_BIG_ENDIAN or EC25519_LITTLE_ENDIAN to set which form to use.

See:

- wc_curve25519_init
- wc_curve25519_make_key
- wc_curve25519_import_private_raw
- wc_curve25519_export_private_raw
- wc_curve25519_size

Return:

- 0 Returned on successfully exporting the private key from the curve25519_key structure.
- BAD_FUNC_ARG Returned if any input parameters are NULL.
- ECC_BAD_ARG_E Returned if wc_curve25519_size() is not equal to key.

Example

```
int ret;

byte priv[32];
int privSz;
curve25519_key key;
// initialize and make key
ret = wc_curve25519_export_private_raw_ex(&key, priv, &privSz,
        EC25519_BIG_ENDIAN);
if (ret != 0) {
    // error exporting key
}
```

C.14.2.12 function wc_curve25519_import_public

```
int wc_curve25519_import_public(
    const byte * in,
    word32 inLen,
    curve25519_key * key
)
```

This function imports a public key from the given in buffer and stores it in the curve25519_key structure.

Parameters:

- **in** Pointer to the buffer containing the public key to import.
- **inLen** Length of the public key to import.
- **key** Pointer to the curve25519_key structure in which to store the key.

See:

- wc_curve25519_init
- wc_curve25519_export_public
- wc_curve25519_import_private_raw
- wc_curve25519_import_public_ex
- wc_curve25519_check_public
- wc_curve25519_size

Return:

- 0 Returned on successfully importing the public key into the curve25519_key structure.
- ECC_BAD_ARG_E Returned if the inLen parameter does not match the key size of the key structure.
- BAD_FUNC_ARG Returned if any of the input parameters are NULL.

Example

```
int ret;

byte pub[32];
// initialize pub with public key

curve25519_key key;
// initialize key

ret = wc_curve25519_import_public(pub,sizeof(pub), &key);
if (ret != 0) {
    // error importing key
}
```

C.14.2.13 function wc_curve25519_import_public_ex

```
int wc_curve25519_import_public_ex(
    const byte * in,
    word32 inLen,
    curve25519_key * key,
    int endian
)
```

This function imports a public key from the given in buffer and stores it in the curve25519_key structure.

Parameters:

- **in** Pointer to the buffer containing the public key to import.
- **inLen** Length of the public key to import.
- **key** Pointer to the curve25519_key structure in which to store the key.
- **endian** EC25519_BIG_ENDIAN or EC25519_LITTLE_ENDIAN to set which form to use.

See:

- wc_curve25519_init
- wc_curve25519_export_public
- wc_curve25519_import_private_raw
- wc_curve25519_import_public

- wc_curve25519_check_public
- wc_curve25519_size

Return:

- 0 Returned on successfully importing the public key into the curve25519_key structure.
- ECC_BAD_ARG_E Returned if the inLen parameter does not match the key size of the key structure.
- BAD_FUNC_ARG Returned if any of the input parameters are NULL.

Example

```
int ret;

byte pub[32];
// initialize pub with public key
curve25519_key key;
// initialize key

ret = wc_curve25519_import_public_ex(pub, sizeof(pub), &key,
        EC25519_BIG_ENDIAN);
if (ret != 0) {
    // error importing key
}
```

C.14.2.14 function wc_curve25519_check_public

```
int wc_curve25519_check_public(
    const byte * pub,
    word32 pubSz,
    int endian
)
```

This function checks that a public key buffer holds a valid Curve25519 key value given the endian ordering.

Parameters:

- **pub** Pointer to the buffer containing the public key to check.
- **pubLen** Length of the public key to check.
- **endian** EC25519_BIG_ENDIAN or EC25519_LITTLE_ENDIAN to set which form to use.

See:

- wc_curve25519_init
- wc_curve25519_import_public
- wc_curve25519_import_public_ex
- wc_curve25519_size

Return:

- 0 Returned when the public key value is valid.
- ECC_BAD_ARG_E Returned if the public key value is not valid.
- BAD_FUNC_ARG Returned if any of the input parameters are NULL.

Example

```
int ret;

byte pub[] = { Contents of public key };
```

```
ret = wc_curve25519_check_public_ex(pub, sizeof(pub), EC25519_BIG_ENDIAN);
if (ret != 0) {
    // error importing key
}
```

C.14.2.15 function wc_curve25519_export_public

```
int wc_curve25519_export_public(
    curve25519_key * key,
    byte * out,
    word32 * outLen
)
```

This function exports a public key from the given key structure and stores the result in the out buffer. Big endian only.

Parameters:

- **key** Pointer to the curve25519_key structure in from which to export the key.
- **out** Pointer to the buffer in which to store the public key.
- **outLen** On in, is the size of the out in bytes. On out, will store the bytes written to the output buffer.

See:

- wc_curve25519_init
- wc_curve25519_export_private_raw
- wc_curve25519_import_public

Return:

- 0 Returned on successfully exporting the public key from the curve25519_key structure.
- ECC_BAD_ARG_E Returned if outLen is less than CURVE25519_PUB_KEY_SIZE.
- BAD_FUNC_ARG Returned if any of the input parameters are NULL.

Example

```
int ret;

byte pub[32];
int pubSz;

curve25519_key key;
// initialize and make key
ret = wc_curve25519_export_public(&key, pub, &pubSz);
if (ret != 0) {
    // error exporting key
}
```

C.14.2.16 function wc_curve25519_export_public_ex

```
int wc_curve25519_export_public_ex(
    curve25519_key * key,
    byte * out,
    word32 * outLen,
    int endian
)
```

This function exports a public key from the given key structure and stores the result in the out buffer. Supports both big and little endian.

Parameters:

- **key** Pointer to the curve25519_key structure in from which to export the key.
- **out** Pointer to the buffer in which to store the public key.
- **outLen** On in, is the size of the out in bytes. On out, will store the bytes written to the output buffer.
- **endian** EC25519_BIG_ENDIAN or EC25519_LITTLE_ENDIAN to set which form to use.

See:

- wc_curve25519_init
- wc_curve25519_export_private_raw
- wc_curve25519_import_public

Return:

- 0 Returned on successfully exporting the public key from the curve25519_key structure.
- ECC_BAD_ARG_E Returned if outLen is less than CURVE25519_PUB_KEY_SIZE.
- BAD_FUNC_ARG Returned if any of the input parameters are NULL.

Example

```
int ret;

byte pub[32];
int pubSz;

curve25519_key key;
// initialize and make key

ret = wc_curve25519_export_public_ex(&key, pub, &pubSz, EC25519_BIG_ENDIAN);
if (ret != 0) {
    // error exporting key
}
```

C.14.2.17 function wc_curve25519_export_key_raw

```
int wc_curve25519_export_key_raw(
    curve25519_key * key,
    byte * priv,
    word32 * privSz,
    byte * pub,
    word32 * pubSz
)
```

Export Curve25519 key pair. Big endian only.

Parameters:

- **key** Pointer to the curve448_key structure in from which to export the key pair.
- **priv** Pointer to the buffer in which to store the private key.
- **privSz** On in, is the size of the priv buffer in bytes. On out, will store the bytes written to the priv buffer.
- **pub** Pointer to the buffer in which to store the public key.
- **pubSz** On in, is the size of the pub buffer in bytes. On out, will store the bytes written to the pub buffer.

See:

- wc_curve25519_export_key_raw_ex
- wc_curve25519_export_private_raw

Return:

- 0 Returned on successfully exporting the key pair from the curve25519_key structure.
- BAD_FUNC_ARG Returned if any input parameters are NULL.
- ECC_BAD_ARG_E Returned if privSz is less than CURVE25519_KEY_SIZE or pubSz is less than CURVE25519_PUB_KEY_SIZE.

Example

```
int ret;

byte pub[32];
byte priv[32];
int pubSz;
int privSz;

curve25519_key key;
// initialize and make key

ret = wc_curve25519_export_key_raw(&key, priv, &privSz, pub, &pubSz);
if (ret != 0) {
    // error exporting key
}
```

C.14.2.18 function wc_curve25519_export_key_raw_ex

```
int wc_curve25519_export_key_raw_ex(
    curve25519_key * key,
    byte * priv,
    word32 * privSz,
    byte * pub,
    word32 * pubSz,
    int endian
)
```

Export curve25519 key pair. Big or little endian.

Parameters:

- **key** Pointer to the curve448_key structure in from which to export the key pair.
- **priv** Pointer to the buffer in which to store the private key.
- **privSz** On in, is the size of the priv buffer in bytes. On out, will store the bytes written to the priv buffer.
- **pub** Pointer to the buffer in which to store the public key.
- **pubSz** On in, is the size of the pub buffer in bytes. On out, will store the bytes written to the pub buffer.
- **endian** EC25519_BIG_ENDIAN or EC25519_LITTLE_ENDIAN to set which form to use.

See:

- wc_curve25519_export_key_raw
- wc_curve25519_export_private_raw_ex
- wc_curve25519_export_public_ex

Return:

- 0 Returned on successfully exporting the key pair from the curve25519_key structure.
- BAD_FUNC_ARG Returned if any input parameters are NULL.
- ECC_BAD_ARG_E Returned if privSz is less than CURVE25519_KEY_SIZE or pubSz is less than CURVE25519_PUB_KEY_SIZE.

Example

```
int ret;

byte pub[32];
byte priv[32];
int pubSz;
int privSz;

curve25519_key key;
// initialize and make key

ret = wc_curve25519_export_key_raw_ex(&key,priv, &privSz, pub, &pubSz,
        EC25519_BIG_ENDIAN);
if (ret != 0) {
    // error exporting key
}
```

C.14.2.19 function wc_curve25519_size

```
int wc_curve25519_size(
    curve25519_key * key
)
```

This function returns the key size of the given key structure.

Parameters:

- **key** Pointer to the curve25519_key structure in for which to determine the key size.

See:

- wc_curve25519_init
- wc_curve25519_make_key

Return:

- Success Given a valid, initialized curve25519_key structure, returns the size of the key.
- 0 Returned if key is NULL

Example

```
int keySz;

curve25519_key key;
// initialize and make key

keySz = wc_curve25519_size(&key);
```

C.14.3 Source code

```
int wc_curve25519_make_key(WC_RNG* rng, int keysize, curve25519_key* key);

int wc_curve25519_shared_secret(curve25519_key* private_key,
```

```
                                curve25519_key* public_key,
                                byte* out, word32* outlen);

int wc_curve25519_shared_secret_ex(curve25519_key* private_key,
                                   curve25519_key* public_key,
                                   byte* out, word32* outlen, int endian);

int wc_curve25519_init(curve25519_key* key);

void wc_curve25519_free(curve25519_key* key);

int wc_curve25519_import_private(const byte* priv, word32 privSz,
                                 curve25519_key* key);

int wc_curve25519_import_private_ex(const byte* priv, word32 privSz,
                                    curve25519_key* key, int endian);

int wc_curve25519_import_private_raw(const byte* priv, word32 privSz,
                            const byte* pub, word32 pubSz, curve25519_key* key);

int wc_curve25519_import_private_raw_ex(const byte* priv, word32 privSz,
                                       const byte* pub, word32 pubSz,
                                       curve25519_key* key, int endian);

int wc_curve25519_export_private_raw(curve25519_key* key, byte* out,
                                     word32* outLen);

int wc_curve25519_export_private_raw_ex(curve25519_key* key, byte* out,
                                       word32* outLen, int endian);

int wc_curve25519_import_public(const byte* in, word32 inLen,
                                curve25519_key* key);

int wc_curve25519_import_public_ex(const byte* in, word32 inLen,
                                   curve25519_key* key, int endian);

int wc_curve25519_check_public(const byte* pub, word32 pubSz, int endian);

int wc_curve25519_export_public(curve25519_key* key, byte* out, word32*
 ↪  outLen);

int wc_curve25519_export_public_ex(curve25519_key* key, byte* out,
                                   word32* outLen, int endian);

int wc_curve25519_export_key_raw(curve25519_key* key,
                                 byte* priv, word32 *privSz,
                                 byte* pub, word32 *pubSz);

int wc_curve25519_export_key_raw_ex(curve25519_key* key,
                                    byte* priv, word32 *privSz,
                                    byte* pub, word32 *pubSz,
                                    int endian);

int wc_curve25519_size(curve25519_key* key);
```

C.15 dox_comments/header_files/curve448.h

C.15.1 Functions

	Name
int	**wc_curve448_make_key.
int	**wc_curve448_shared_secret**(curve448_key * private_key, curve448_key * public_key, byte * out, word32 * outlen)This function computes a shared secret key given a secret private key and a received public key. It stores the generated secret key in the buffer out and assigns the variable of the secret key to outlen. Only supports big endian.
int	**wc_curve448_shared_secret_ex**(curve448_key * private_key, curve448_key * public_key, byte * out, word32 * outlen, int endian)This function computes a shared secret key given a secret private key and a received public key. It stores the generated secret key in the buffer out and assigns the variable of the secret key to outlen. Supports both big and little endian.
int	**wc_curve448_init**(curve448_key * key)This function initializes a Curve448 key. It should be called before generating a key for the structure.
void	**wc_curve448_free**(curve448_key * key)This function frees a Curve448 object.
int	**wc_curve448_import_private**(const byte * priv, word32 privSz, curve448_key * key)This function imports a curve448 private key only. (Big endian).
int	**wc_curve448_import_private_ex**(const byte * priv, word32 privSz, curve448_key * key, int endian)curve448 private key import only. (Big or Little endian).
int	**wc_curve448_import_private_raw**(const byte * priv, word32 privSz, const byte * pub, word32 pubSz, curve448_key * key)This function imports a public-private key pair into a curve448_key structure. Big endian only.
int	**wc_curve448_import_private_raw_ex**(const byte * priv, word32 privSz, const byte * pub, word32 pubSz, curve448_key * key, int endian)This function imports a public-private key pair into a curve448_key structure. Supports both big and little endian.
int	**wc_curve448_export_private_raw**(curve448_key * key, byte * out, word32 * outLen)This function exports a private key from a curve448_key structure and stores it in the given out buffer. It also sets outLen to be the size of the exported key. Big Endian only.

	Name
int	**wc_curve448_export_private_raw_ex**(curve448_key * key, byte * out, word32 * outLen, int endian)This function exports a private key from a curve448_key structure and stores it in the given out buffer. It also sets outLen to be the size of the exported key. Can specify whether it's big or little endian.
int	**wc_curve448_import_public**(const byte * in, word32 inLen, curve448_key * key)This function imports a public key from the given in buffer and stores it in the curve448_key structure.
int	**wc_curve448_import_public_ex**(const byte * in, word32 inLen, curve448_key * key, int endian)This function imports a public key from the given in buffer and stores it in the curve448_key structure.
int	**wc_curve448_check_public**(const byte * pub, word32 pubSz, int endian)This function checks that a public key buffer holds a valid Curve448 key value given the endian ordering.
int	**wc_curve448_export_public**(curve448_key * key, byte * out, word32 * outLen)This function exports a public key from the given key structure and stores the result in the out buffer. Big endian only.
int	**wc_curve448_export_public_ex**(curve448_key * key, byte * out, word32 * outLen, int endian)This function exports a public key from the given key structure and stores the result in the out buffer. Supports both big and little endian.
int	**wc_curve448_export_key_raw**(curve448_key * key, byte * priv, word32 * privSz, byte * pub, word32 * pubSz)This function exports a key pair from the given key structure and stores the result in the out buffer. Big endian only.
int	**wc_curve448_export_key_raw_ex**(curve448_key * key, byte * priv, word32 * privSz, byte * pub, word32 * pubSz, int endian)Export curve448 key pair. Big or little endian.
int	**wc_curve448_size**(curve448_key * key)This function returns the key size of the given key structure.

C.15.2 Functions Documentation

C.15.2.1 function wc_curve448_make_key

```
int wc_curve448_make_key(
    WC_RNG * rng,
    int keysize,
    curve448_key * key
```

)

This function generates a Curve448 key using the given random number generator, rng, of the size given (keysize), and stores it in the given curve448_key structure. It should be called after the key structure has been initialized through wc_curve448_init().

Parameters:

- **rng** Pointer to the RNG object used to generate the ecc key.
- **keysize** Size of the key to generate. Must be 56 bytes for curve448.
- **key** Pointer to the curve448_key structure in which to store the generated key.

See: wc_curve448_init

Return:

- 0 Returned on successfully generating the key and and storing it in the given curve448_key structure.
- ECC_BAD_ARG_E Returned if the input keysize does not correspond to the keysize for a curve448 key (56 bytes).
- RNG_FAILURE_E Returned if the rng internal status is not DRBG_OK or if there is in generating the next random block with rng.
- BAD_FUNC_ARG Returned if any of the input parameters passed in are NULL.

Example

```
int ret;

curve448_key key;
wc_curve448_init(&key); // initialize key
WC_RNG rng;
wc_InitRng(&rng); // initialize random number generator

ret = wc_curve448_make_key(&rng, 56, &key);
if (ret != 0) {
    // error making Curve448 key
}
```

C.15.2.2 function wc_curve448_shared_secret

```
int wc_curve448_shared_secret(
    curve448_key * private_key,
    curve448_key * public_key,
    byte * out,
    word32 * outlen
)
```

This function computes a shared secret key given a secret private key and a received public key. It stores the generated secret key in the buffer out and assigns the variable of the secret key to outlen. Only supports big endian.

Parameters:

- **private_key** Pointer to the curve448_key structure initialized with the user's private key.
- **public_key** Pointer to the curve448_key structure containing the received public key.
- **out** Pointer to a buffer in which to store the 56 byte computed secret key.
- **outlen** Pointer in which to store the length written to the output buffer.

See:

- wc_curve448_init

- wc_curve448_make_key
- wc_curve448_shared_secret_ex

Return:

- 0 Returned on successfully computing a shared secret key
- BAD_FUNC_ARG Returned if any of the input parameters passed in are NULL

Example

```
int ret;

byte sharedKey[56];
word32 keySz;
curve448_key privKey, pubKey;
// initialize both keys

ret = wc_curve448_shared_secret(&privKey, &pubKey, sharedKey, &keySz);
if (ret != 0) {
    // error generating shared key
}
```

C.15.2.3 function wc_curve448_shared_secret_ex

```
int wc_curve448_shared_secret_ex(
    curve448_key * private_key,
    curve448_key * public_key,
    byte * out,
    word32 * outlen,
    int endian
)
```

This function computes a shared secret key given a secret private key and a received public key. It stores the generated secret key in the buffer out and assigns the variable of the secret key to outlen. Supports both big and little endian.

Parameters:

- **private_key** Pointer to the curve448_key structure initialized with the user's private key.
- **public_key** Pointer to the curve448_key structure containing the received public key.
- **out** Pointer to a buffer in which to store the 56 byte computed secret key.
- **outlen** Pointer in which to store the length written to the output buffer.
- **endian** EC448_BIG_ENDIAN or EC448_LITTLE_ENDIAN to set which form to use.

See:

- wc_curve448_init
- wc_curve448_make_key
- wc_curve448_shared_secret

Return:

- 0 Returned on successfully computing a shared secret key.
- BAD_FUNC_ARG Returned if any of the input parameters passed in are NULL.

Example

```
int ret;

byte sharedKey[56];
word32 keySz;
```

```
curve448_key privKey, pubKey;
// initialize both keys

ret = wc_curve448_shared_secret_ex(&privKey, &pubKey, sharedKey, &keySz,
        EC448_BIG_ENDIAN);
if (ret != 0) {
    // error generating shared key
}
```

C.15.2.4 function wc_curve448_init

```
int wc_curve448_init(
    curve448_key * key
)
```

This function initializes a Curve448 key. It should be called before generating a key for the structure.

Parameters:

- **key** Pointer to the curve448_key structure to initialize.

See: wc_curve448_make_key

Return:

- 0 Returned on successfully initializing the curve448_key structure.
- BAD_FUNC_ARG Returned when key is NULL.

Example

```
curve448_key key;
wc_curve448_init(&key); // initialize key
// make key and proceed to encryption
```

C.15.2.5 function wc_curve448_free

```
void wc_curve448_free(
    curve448_key * key
)
```

This function frees a Curve448 object.

Parameters:

- **key** Pointer to the key object to free.

See:

- wc_curve448_init
- wc_curve448_make_key

Example

```
curve448_key privKey;
// initialize key, use it to generate shared secret key
wc_curve448_free(&privKey);
```

C.15.2.6 function wc_curve448_import_private

```
int wc_curve448_import_private(
    const byte * priv,
    word32 privSz,
    curve448_key * key
)
```

This function imports a curve448 private key only. (Big endian).

Parameters:

- **priv** Pointer to a buffer containing the private key to import.
- **privSz** Length of the private key to import.
- **key** Pointer to the structure in which to store the imported key.

See:

- wc_curve448_import_private_ex
- wc_curve448_size

Return:

- 0 Returned on successfully importing private key.
- BAD_FUNC_ARG Returns if key or priv is null.
- ECC_BAD_ARG_E Returns if privSz is not equal to CURVE448_KEY_SIZE.

Example

```
int ret;

byte priv[] = { Contents of private key };
curve448_key key;
wc_curve448_init(&key);

ret = wc_curve448_import_private(priv, sizeof(priv), &key);
if (ret != 0) {
    // error importing key
}
```

C.15.2.7 function wc_curve448_import_private_ex

```
int wc_curve448_import_private_ex(
    const byte * priv,
    word32 privSz,
    curve448_key * key,
    int endian
)
```

curve448 private key import only. (Big or Little endian).

Parameters:

- **priv** Pointer to a buffer containing the private key to import.
- **privSz** Length of the private key to import.
- **key** Pointer to the structure in which to store the imported key.
- **endian** EC448_BIG_ENDIAN or EC448_LITTLE_ENDIAN to set which form to use.

See:

- wc_curve448_import_private
- wc_curve448_size

Return:

- 0 Returned on successfully importing private key.
- BAD_FUNC_ARG Returns if key or priv is null.
- ECC_BAD_ARG_E Returns if privSz is not equal to CURVE448_KEY_SIZE.

Example

```
int ret;

byte priv[] = { // Contents of private key };
curve448_key key;
wc_curve448_init(&key);

ret = wc_curve448_import_private_ex(priv, sizeof(priv), &key,
        EC448_BIG_ENDIAN);
if (ret != 0) {
    // error importing key
}
```

C.15.2.8 function wc_curve448_import_private_raw

```
int wc_curve448_import_private_raw(
    const byte * priv,
    word32 privSz,
    const byte * pub,
    word32 pubSz,
    curve448_key * key
)
```

This function imports a public-private key pair into a curve448_key structure. Big endian only.

Parameters:

- **priv** Pointer to a buffer containing the private key to import.
- **privSz** Length of the private key to import.
- **pub** Pointer to a buffer containing the public key to import.
- **pubSz** Length of the public key to import.
- **key** Pointer to the structure in which to store the imported keys

See:

- wc_curve448_init
- wc_curve448_make_key
- wc_curve448_import_public
- wc_curve448_export_private_raw

Return:

- 0 Returned on importing into the curve448_key structure.
- BAD_FUNC_ARG Returns if any of the input parameters are null.
- ECC_BAD_ARG_E Returned if the input key's key size does not match the public or private key sizes.

Example

```
int ret;

byte priv[56];
byte pub[56];
```

```
// initialize with public and private keys
curve448_key key;

wc_curve448_init(&key);
// initialize key

ret = wc_curve448_import_private_raw(&priv, sizeof(priv), pub, sizeof(pub),
        &key);
if (ret != 0) {
    // error importing keys
}
```

C.15.2.9 function wc_curve448_import_private_raw_ex

```
int wc_curve448_import_private_raw_ex(
    const byte * priv,
    word32 privSz,
    const byte * pub,
    word32 pubSz,
    curve448_key * key,
    int endian
)
```

This function imports a public-private key pair into a curve448_key structure. Supports both big and little endian.

Parameters:

- **priv** Pointer to a buffer containing the private key to import.
- **privSz** Length of the private key to import.
- **pub** Pointer to a buffer containing the public key to import.
- **pubSz** Length of the public key to import.
- **key** Pointer to the structure in which to store the imported keys.
- **endian** EC448_BIG_ENDIAN or EC448_LITTLE_ENDIAN to set which form to use.

See:

- wc_curve448_init
- wc_curve448_make_key
- wc_curve448_import_public
- wc_curve448_export_private_raw
- wc_curve448_import_private_raw

Return:

- 0 Returned on importing into the curve448_key structure.
- BAD_FUNC_ARG Returns if any of the input parameters are null.
- ECC_BAD_ARG_E Returned if the input key's key size does not match the public or private key sizes.

Example

```
int ret;

byte priv[56];
byte pub[56];
// initialize with public and private keys
curve448_key key;
```

```
wc_curve448_init(&key);
// initialize key

ret = wc_curve448_import_private_raw_ex(&priv, sizeof(priv), pub,
        sizeof(pub), &key, EC448_BIG_ENDIAN);
if (ret != 0) {
    // error importing keys
}
```

C.15.2.10 function wc_curve448_export_private_raw

```
int wc_curve448_export_private_raw(
    curve448_key * key,
    byte * out,
    word32 * outLen
)
```

This function exports a private key from a curve448_key structure and stores it in the given out buffer. It also sets outLen to be the size of the exported key. Big Endian only.

Parameters:

- **key** Pointer to the structure from which to export the key.
- **out** Pointer to the buffer in which to store the exported key.
- **outLen** On in, is the size of the out in bytes. On out, will store the bytes written to the output buffer.

See:

- wc_curve448_init
- wc_curve448_make_key
- wc_curve448_import_private_raw
- wc_curve448_export_private_raw_ex

Return:

- 0 Returned on successfully exporting the private key from the curve448_key structure.
- BAD_FUNC_ARG Returned if any input parameters are NULL.
- ECC_BAD_ARG_E Returned if wc_curve448_size() is not equal to key.

Example

```
int ret;
byte priv[56];
int privSz;

curve448_key key;
// initialize and make key

ret = wc_curve448_export_private_raw(&key, priv, &privSz);
if (ret != 0) {
    // error exporting key
}
```

C.15.2.11 function wc_curve448_export_private_raw_ex

```
int wc_curve448_export_private_raw_ex(
    curve448_key * key,
    byte * out,
```

```
    word32 * outLen,
    int endian
)
```

This function exports a private key from a curve448_key structure and stores it in the given out buffer. It also sets outLen to be the size of the exported key. Can specify whether it's big or little endian.

Parameters:

- **key** Pointer to the structure from which to export the key.
- **out** Pointer to the buffer in which to store the exported key.
- **outLen** On in, is the size of the out in bytes. On out, will store the bytes written to the output buffer.
- **endian** EC448_BIG_ENDIAN or EC448_LITTLE_ENDIAN to set which form to use.

See:

- wc_curve448_init
- wc_curve448_make_key
- wc_curve448_import_private_raw
- wc_curve448_export_private_raw
- wc_curve448_size

Return:

- 0 Returned on successfully exporting the private key from the curve448_key structure.
- BAD_FUNC_ARG Returned if any input parameters are NULL.
- ECC_BAD_ARG_E Returned if wc_curve448_size() is not equal to key.

Example

```
int ret;

byte priv[56];
int privSz;
curve448_key key;
// initialize and make key
ret = wc_curve448_export_private_raw_ex(&key, priv, &privSz,
        EC448_BIG_ENDIAN);
if (ret != 0) {
    // error exporting key
}
```

C.15.2.12 function wc_curve448_import_public

```
int wc_curve448_import_public(
    const byte * in,
    word32 inLen,
    curve448_key * key
)
```

This function imports a public key from the given in buffer and stores it in the curve448_key structure.

Parameters:

- **in** Pointer to the buffer containing the public key to import.
- **inLen** Length of the public key to import.
- **key** Pointer to the curve448_key structure in which to store the key.

See:

- wc_curve448_init
- wc_curve448_export_public
- wc_curve448_import_private_raw
- wc_curve448_import_public_ex
- wc_curve448_check_public
- wc_curve448_size

Return:

- 0 Returned on successfully importing the public key into the curve448_key structure.
- ECC_BAD_ARG_E Returned if the inLen parameter does not match the key size of the key structure.
- BAD_FUNC_ARG Returned if any of the input parameters are NULL.

Example

```
int ret;

byte pub[56];
// initialize pub with public key

curve448_key key;
// initialize key

ret = wc_curve448_import_public(pub,sizeof(pub), &key);
if (ret != 0) {
    // error importing key
}
```

C.15.2.13 function wc_curve448_import_public_ex

```
int wc_curve448_import_public_ex(
    const byte * in,
    word32 inLen,
    curve448_key * key,
    int endian
)
```

This function imports a public key from the given in buffer and stores it in the curve448_key structure.

Parameters:

- **in** Pointer to the buffer containing the public key to import.
- **inLen** Length of the public key to import.
- **key** Pointer to the curve448_key structure in which to store the key.
- **endian** EC448_BIG_ENDIAN or EC448_LITTLE_ENDIAN to set which form to use.

See:

- wc_curve448_init
- wc_curve448_export_public
- wc_curve448_import_private_raw
- wc_curve448_import_public
- wc_curve448_check_public
- wc_curve448_size

Return:

- 0 Returned on successfully importing the public key into the curve448_key structure.

- ECC_BAD_ARG_E Returned if the inLen parameter does not match the key size of the key structure.
- BAD_FUNC_ARG Returned if any of the input parameters are NULL.

Example

```
int ret;

byte pub[56];
// initialize pub with public key
curve448_key key;
// initialize key

ret = wc_curve448_import_public_ex(pub, sizeof(pub), &key,
        EC448_BIG_ENDIAN);
if (ret != 0) {
    // error importing key
}
```

C.15.2.14 function wc_curve448_check_public

```
int wc_curve448_check_public(
    const byte * pub,
    word32 pubSz,
    int endian
)
```

This function checks that a public key buffer holds a valid Curve448 key value given the endian ordering.

Parameters:

- **pub** Pointer to the buffer containing the public key to check.
- **pubLen** Length of the public key to check.
- **endian** EC448_BIG_ENDIAN or EC448_LITTLE_ENDIAN to set which form to use.

See:

- wc_curve448_init
- wc_curve448_import_public
- wc_curve448_import_public_ex
- wc_curve448_size

Return:

- 0 Returned when the public key value is valid.
- ECC_BAD_ARG_E Returned if the public key value is not valid.
- BAD_FUNC_ARG Returned if any of the input parameters are NULL.

Example

```
int ret;

byte pub[] = { Contents of public key };

ret = wc_curve448_check_public_ex(pub, sizeof(pub), EC448_BIG_ENDIAN);
if (ret != 0) {
    // error importing key
}
```

C.15.2.15 function wc_curve448_export_public

```
int wc_curve448_export_public(
    curve448_key * key,
    byte * out,
    word32 * outLen
)
```

This function exports a public key from the given key structure and stores the result in the out buffer. Big endian only.

Parameters:

- **key** Pointer to the curve448_key structure in from which to export the key.
- **out** Pointer to the buffer in which to store the public key.
- **outLen** On in, is the size of the out in bytes. On out, will store the bytes written to the output buffer.

See:

- wc_curve448_init
- wc_curve448_export_private_raw
- wc_curve448_import_public

Return:

- 0 Returned on successfully exporting the public key from the curve448_key structure.
- ECC_BAD_ARG_E Returned if outLen is less than CURVE448_PUB_KEY_SIZE.
- BAD_FUNC_ARG Returned if any of the input parameters are NULL.

Example

```
int ret;

byte pub[56];
int pubSz;

curve448_key key;
// initialize and make key

ret = wc_curve448_export_public(&key, pub, &pubSz);
if (ret != 0) {
    // error exporting key
}
```

C.15.2.16 function wc_curve448_export_public_ex

```
int wc_curve448_export_public_ex(
    curve448_key * key,
    byte * out,
    word32 * outLen,
    int endian
)
```

This function exports a public key from the given key structure and stores the result in the out buffer. Supports both big and little endian.

Parameters:

- **key** Pointer to the curve448_key structure in from which to export the key.
- **out** Pointer to the buffer in which to store the public key.

- **outLen** On in, is the size of the out in bytes. On out, will store the bytes written to the output buffer.
- **endian** EC448_BIG_ENDIAN or EC448_LITTLE_ENDIAN to set which form to use.

See:

- wc_curve448_init
- wc_curve448_export_private_raw
- wc_curve448_import_public

Return:

- 0 Returned on successfully exporting the public key from the curve448_key structure.
- ECC_BAD_ARG_E Returned if outLen is less than CURVE448_PUB_KEY_SIZE.
- BAD_FUNC_ARG Returned if any of the input parameters are NULL.

Example

```
int ret;

byte pub[56];
int pubSz;

curve448_key key;
// initialize and make key

ret = wc_curve448_export_public_ex(&key, pub, &pubSz, EC448_BIG_ENDIAN);
if (ret != 0) {
    // error exporting key
}
```

C.15.2.17 function wc_curve448_export_key_raw

```
int wc_curve448_export_key_raw(
    curve448_key * key,
    byte * priv,
    word32 * privSz,
    byte * pub,
    word32 * pubSz
)
```

This function exports a key pair from the given key structure and stores the result in the out buffer. Big endian only.

Parameters:

- **key** Pointer to the curve448_key structure in from which to export the key pair.
- **priv** Pointer to the buffer in which to store the private key.
- **privSz** On in, is the size of the priv buffer in bytes. On out, will store the bytes written to the priv buffer.
- **pub** Pointer to the buffer in which to store the public key.
- **pubSz** On in, is the size of the pub buffer in bytes. On out, will store the bytes written to the pub buffer.

See:

- wc_curve448_export_key_raw_ex
- wc_curve448_export_private_raw

Return:

- 0 Returned on successfully exporting the key pair from the curve448_key structure.
- BAD_FUNC_ARG Returned if any input parameters are NULL.
- ECC_BAD_ARG_E Returned if privSz is less than CURVE448_KEY_SIZE or pubSz is less than CURVE448_PUB_KEY_SIZE.

Example

```
int ret;

byte pub[56];
byte priv[56];
int pubSz;
int privSz;

curve448_key key;
// initialize and make key

ret = wc_curve448_export_key_raw(&key, priv, &privSz, pub, &pubSz);
if (ret != 0) {
    // error exporting key
}
```

C.15.2.18 function wc_curve448_export_key_raw_ex

```
int wc_curve448_export_key_raw_ex(
    curve448_key * key,
    byte * priv,
    word32 * privSz,
    byte * pub,
    word32 * pubSz,
    int endian
)
```

Export curve448 key pair. Big or little endian.

Parameters:

- **key** Pointer to the curve448_key structure in from which to export the key pair.
- **priv** Pointer to the buffer in which to store the private key.
- **privSz** On in, is the size of the priv buffer in bytes. On out, will store the bytes written to the priv buffer.
- **pub** Pointer to the buffer in which to store the public key.
- **pubSz** On in, is the size of the pub buffer in bytes. On out, will store the bytes written to the pub buffer.
- **endian** EC448_BIG_ENDIAN or EC448_LITTLE_ENDIAN to set which form to use.

See:

- wc_curve448_export_key_raw
- wc_curve448_export_private_raw_ex
- wc_curve448_export_public_ex

Return:

- 0 Success
- BAD_FUNC_ARG Returned if any input parameters are NULL.
- ECC_BAD_ARG_E Returned if privSz is less than CURVE448_KEY_SIZE or pubSz is less than CURVE448_PUB_KEY_SIZE.

This function exports a key pair from the given key structure and stores the result in the out buffer. Big or little endian.

Example

```
int ret;

byte pub[56];
byte priv[56];
int pubSz;
int privSz;

curve448_key key;
// initialize and make key

ret = wc_curve448_export_key_raw_ex(&key,priv, &privSz, pub, &pubSz,
        EC448_BIG_ENDIAN);
if (ret != 0) {
    // error exporting key
}
```

C.15.2.19 function wc_curve448_size

```
int wc_curve448_size(
    curve448_key * key
)
```

This function returns the key size of the given key structure.

Parameters:

- **key** Pointer to the curve448_key structure in for which to determine the key size.

See:

- wc_curve448_init
- wc_curve448_make_key

Return:

- Success Given a valid, initialized curve448_key structure, returns the size of the key.
- 0 Returned if key is NULL.

Example

```
int keySz;

curve448_key key;
// initialize and make key

keySz = wc_curve448_size(&key);
```

C.15.3 Source code

```
int wc_curve448_make_key(WC_RNG* rng, int keysize, curve448_key* key);

int wc_curve448_shared_secret(curve448_key* private_key,
                              curve448_key* public_key,
                              byte* out, word32* outlen);
```

```
int wc_curve448_shared_secret_ex(curve448_key* private_key,
                                 curve448_key* public_key,
                                 byte* out, word32* outlen, int endian);

int wc_curve448_init(curve448_key* key);

void wc_curve448_free(curve448_key* key);

int wc_curve448_import_private(const byte* priv, word32 privSz,
                               curve448_key* key);

int wc_curve448_import_private_ex(const byte* priv, word32 privSz,
                                  curve448_key* key, int endian);

int wc_curve448_import_private_raw(const byte* priv, word32 privSz,
                           const byte* pub, word32 pubSz, curve448_key* key);

int wc_curve448_import_private_raw_ex(const byte* priv, word32 privSz,
                                      const byte* pub, word32 pubSz,
                                      curve448_key* key, int endian);

int wc_curve448_export_private_raw(curve448_key* key, byte* out,
                                   word32* outLen);

int wc_curve448_export_private_raw_ex(curve448_key* key, byte* out,
                                      word32* outLen, int endian);

int wc_curve448_import_public(const byte* in, word32 inLen,
                              curve448_key* key);

int wc_curve448_import_public_ex(const byte* in, word32 inLen,
                                 curve448_key* key, int endian);

int wc_curve448_check_public(const byte* pub, word32 pubSz, int endian);

int wc_curve448_export_public(curve448_key* key, byte* out, word32* outLen);

int wc_curve448_export_public_ex(curve448_key* key, byte* out,
                                 word32* outLen, int endian);

int wc_curve448_export_key_raw(curve448_key* key,
                               byte* priv, word32 *privSz,
                               byte* pub, word32 *pubSz);

int wc_curve448_export_key_raw_ex(curve448_key* key,
                                  byte* priv, word32 *privSz,
                                  byte* pub, word32 *pubSz,
                                  int endian);

int wc_curve448_size(curve448_key* key);
```

C.16 dox_comments/header_files/des3.h

C.16.1 Functions

	Name
int	**wc_Des_SetKey**(Des * des, const byte * key, const byte * iv, int dir)This function sets the key and initialization vector (iv) for the Des structure given as argument. It also initializes and allocates space for the buffers needed for encryption and decryption, if these have not yet been initialized. Note: If no iv is provided (i.e. iv == NULL) the initialization vector defaults to an iv of 0.
void	**wc_Des_SetIV**(Des * des, const byte * iv)This function sets the initialization vector (iv) for the Des structure given as argument. When passed a NULL iv, it sets the initialization vector to 0.
int	**wc_Des_CbcEncrypt**(Des * des, byte * out, const byte * in, word32 sz)This function encrypts the input message, in, and stores the result in the output buffer, out. It uses DES encryption with cipher block chaining (CBC) mode.
int	**wc_Des_CbcDecrypt**(Des * des, byte * out, const byte * in, word32 sz)This function decrypts the input ciphertext, in, and stores the resulting plaintext in the output buffer, out. It uses DES encryption with cipher block chaining (CBC) mode.
int	**wc_Des_EcbEncrypt**(Des * des, byte * out, const byte * in, word32 sz)This function encrypts the input message, in, and stores the result in the output buffer, out. It uses Des encryption with Electronic Codebook (ECB) mode.
int	**wc_Des3_EcbEncrypt**(Des3 * des, byte * out, const byte * in, word32 sz)This function encrypts the input message, in, and stores the result in the output buffer, out. It uses Des3 encryption with Electronic Codebook (ECB) mode. Warning: In nearly all use cases ECB mode is considered to be less secure. Please avoid using ECB API's directly whenever possible.

COPYRIGHT ©2024 wolfSSL Inc.

	Name
int	**wc_Des3_SetKey**(Des3 * des, const byte * key, const byte * iv, int dir)This function sets the key and initialization vector (iv) for the Des3 structure given as argument. It also initializes and allocates space for the buffers needed for encryption and decryption, if these have not yet been initialized. Note: If no iv is provided (i.e. iv == NULL) the initialization vector defaults to an iv of 0.
int	**wc_Des3_SetIV**(Des3 * des, const byte * iv)This function sets the initialization vector (iv) for the Des3 structure given as argument. When passed a NULL iv, it sets the initialization vector to 0.
int	**wc_Des3_CbcEncrypt**(Des3 * des, byte * out, const byte * in, word32 sz)This function encrypts the input message, in, and stores the result in the output buffer, out. It uses Triple Des (3DES) encryption with cipher block chaining (CBC) mode.
int	**wc_Des3_CbcDecrypt**(Des3 * des, byte * out, const byte * in, word32 sz)This function decrypts the input ciphertext, in, and stores the resulting plaintext in the output buffer, out. It uses Triple Des (3DES) encryption with cipher block chaining (CBC) mode.

C.16.2 Functions Documentation

C.16.2.1 function wc_Des_SetKey

```
int wc_Des_SetKey(
    Des * des,
    const byte * key,
    const byte * iv,
    int dir
)
```

This function sets the key and initialization vector (iv) for the Des structure given as argument. It also initializes and allocates space for the buffers needed for encryption and decryption, if these have not yet been initialized. Note: If no iv is provided (i.e. iv == NULL) the initialization vector defaults to an iv of 0.

Parameters:

- **des** pointer to the Des structure to initialize
- **key** pointer to the buffer containing the 8 byte key with which to initialize the Des structure
- **iv** pointer to the buffer containing the 8 byte iv with which to initialize the Des structure. If this is not provided, the iv defaults to 0
- **dir** direction of encryption. Valid options are: DES_ENCRYPTION, and DES_DECRYPTION

See:

- wc_Des_SetIV
- wc_Des3_SetKey

Return: 0 On successfully setting the key and initialization vector for the Des structure

3

Example

```
Des enc; // Des structure used for encryption
int ret;
byte key[] = { // initialize with 8 byte key };
byte iv[]  = { // initialize with 8 byte iv };

ret = wc_Des_SetKey(&des, key, iv, DES_ENCRYPTION);
if (ret != 0) {
    // error initializing des structure
}
```

C.16.2.2 function wc_Des_SetIV

```
void wc_Des_SetIV(
    Des * des,
    const byte * iv
)
```

This function sets the initialization vector (iv) for the Des structure given as argument. When passed a NULL iv, it sets the initialization vector to 0.

Parameters:

- **des** pointer to the Des structure for which to set the iv
- **iv** pointer to the buffer containing the 8 byte iv with which to initialize the Des structure. If this is not provided, the iv defaults to 0

See: wc_Des_SetKey

Return: none No returns.

3

Example

```
Des enc; // Des structure used for encryption
// initialize enc with wc_Des_SetKey
byte iv[]  = { // initialize with 8 byte iv };
wc_Des_SetIV(&enc, iv);
}
```

C.16.2.3 function wc_Des_CbcEncrypt

```
int wc_Des_CbcEncrypt(
    Des * des,
    byte * out,
    const byte * in,
    word32 sz
)
```

This function encrypts the input message, in, and stores the result in the output buffer, out. It uses DES encryption with cipher block chaining (CBC) mode.

Parameters:

- **des** pointer to the Des structure to use for encryption

- **out** pointer to the buffer in which to store the encrypted ciphertext
- **in** pointer to the input buffer containing the message to encrypt
- **sz** length of the message to encrypt

See:

- wc_Des_SetKey
- wc_Des_CbcDecrypt

Return: 0 Returned upon successfully encrypting the given input message

3

Example

```
Des enc; // Des structure used for encryption
// initialize enc with wc_Des_SetKey, use mode DES_ENCRYPTION

byte plain[]  = { // initialize with message };
byte cipher[sizeof(plain)];

if ( wc_Des_CbcEncrypt(&enc, cipher, plain, sizeof(plain)) != 0) {
    // error encrypting message
}
```

C.16.2.4 function wc_Des_CbcDecrypt

```
int wc_Des_CbcDecrypt(
    Des * des,
    byte * out,
    const byte * in,
    word32 sz
)
```

This function decrypts the input ciphertext, in, and stores the resulting plaintext in the output buffer, out. It uses DES encryption with cipher block chaining (CBC) mode.

Parameters:

- **des** pointer to the Des structure to use for decryption
- **out** pointer to the buffer in which to store the decrypted plaintext
- **in** pointer to the input buffer containing the encrypted ciphertext
- **sz** length of the ciphertext to decrypt

See:

- wc_Des_SetKey
- wc_Des_CbcEncrypt

Return: 0 Returned upon successfully decrypting the given ciphertext

3

Example

```
Des dec; // Des structure used for decryption
// initialize dec with wc_Des_SetKey, use mode DES_DECRYPTION

byte cipher[]  = { // initialize with ciphertext };
byte decoded[sizeof(cipher)];

if ( wc_Des_CbcDecrypt(&dec, decoded, cipher, sizeof(cipher)) != 0) {
```

```
    // error decrypting message
}
```

C.16.2.5 function wc_Des_EcbEncrypt

```
int wc_Des_EcbEncrypt(
    Des * des,
    byte * out,
    const byte * in,
    word32 sz
)
```

This function encrypts the input message, in, and stores the result in the output buffer, out. It uses Des encryption with Electronic Codebook (ECB) mode.

Parameters:

- **des** pointer to the Des structure to use for encryption
- **out** pointer to the buffer in which to store the encrypted message
- **in** pointer to the input buffer containing the plaintext to encrypt
- **sz** length of the plaintext to encrypt

See: wc_Des_SetKe

Return: 0: Returned upon successfully encrypting the given plaintext.

3

Example

```
Des enc; // Des structure used for encryption
// initialize enc with wc_Des_SetKey, use mode DES_ENCRYPTION

byte plain[]  = { // initialize with message to encrypt };
byte cipher[sizeof(plain)];

if ( wc_Des_EcbEncrypt(&enc,cipher, plain, sizeof(plain)) != 0) {
    // error encrypting message
}
```

C.16.2.6 function wc_Des3_EcbEncrypt

```
int wc_Des3_EcbEncrypt(
    Des3 * des,
    byte * out,
    const byte * in,
    word32 sz
)
```

This function encrypts the input message, in, and stores the result in the output buffer, out. It uses Des3 encryption with Electronic Codebook (ECB) mode. Warning: In nearly all use cases ECB mode is considered to be less secure. Please avoid using ECB API's directly whenever possible.

Parameters:

- **des3** pointer to the Des3 structure to use for encryption
- **out** pointer to the buffer in which to store the encrypted message
- **in** pointer to the input buffer containing the plaintext to encrypt
- **sz** length of the plaintext to encrypt

See: wc_Des3_SetKey

Return: 0 Returned upon successfully encrypting the given plaintext

3

Example

```
Des3 enc; // Des3 structure used for encryption
// initialize enc with wc_Des3_SetKey, use mode DES_ENCRYPTION

byte plain[]  = { // initialize with message to encrypt };
byte cipher[sizeof(plain)];

if ( wc_Des3_EcbEncrypt(&enc,cipher, plain, sizeof(plain)) != 0) {
    // error encrypting message
}
```

C.16.2.7 function wc_Des3_SetKey

```
int wc_Des3_SetKey(
    Des3 * des,
    const byte * key,
    const byte * iv,
    int dir
)
```

This function sets the key and initialization vector (iv) for the Des3 structure given as argument. It also initializes and allocates space for the buffers needed for encryption and decryption, if these have not yet been initialized. Note: If no iv is provided (i.e. iv == NULL) the initialization vector defaults to an iv of 0.

Parameters:

- **des3** pointer to the Des3 structure to initialize
- **key** pointer to the buffer containing the 24 byte key with which to initialize the Des3 structure
- **iv** pointer to the buffer containing the 8 byte iv with which to initialize the Des3 structure. If this is not provided, the iv defaults to 0
- **dir** direction of encryption. Valid options are: DES_ENCRYPTION, and DES_DECRYPTION

See:

- wc_Des3_SetIV
- wc_Des3_CbcEncrypt
- wc_Des3_CbcDecrypt

Return: 0 On successfully setting the key and initialization vector for the Des structure

3

Example

```
Des3 enc; // Des3 structure used for encryption
int ret;
byte key[] = { // initialize with 24 byte key };
byte iv[]  = { // initialize with 8 byte iv };

ret = wc_Des3_SetKey(&des, key, iv, DES_ENCRYPTION);
if (ret != 0) {
    // error initializing des structure
}
```

C.16.2.8 function wc_Des3_SetIV

```
int wc_Des3_SetIV(
    Des3 * des,
    const byte * iv
)
```

This function sets the initialization vector (iv) for the Des3 structure given as argument. When passed a NULL iv, it sets the initialization vector to 0.

Parameters:

- **des** pointer to the Des3 structure for which to set the iv
- **iv** pointer to the buffer containing the 8 byte iv with which to initialize the Des3 structure. If this is not provided, the iv defaults to 0

See: wc_Des3_SetKey

Return: none No returns.

3

Example

```
Des3 enc; // Des3 structure used for encryption
// initialize enc with wc_Des3_SetKey

byte iv[]  = { // initialize with 8 byte iv };

wc_Des3_SetIV(&enc, iv);
}
```

C.16.2.9 function wc_Des3_CbcEncrypt

```
int wc_Des3_CbcEncrypt(
    Des3 * des,
    byte * out,
    const byte * in,
    word32 sz
)
```

This function encrypts the input message, in, and stores the result in the output buffer, out. It uses Triple Des (3DES) encryption with cipher block chaining (CBC) mode.

Parameters:

- **des** pointer to the Des3 structure to use for encryption
- **out** pointer to the buffer in which to store the encrypted ciphertext
- **in** pointer to the input buffer containing the message to encrypt
- **sz** length of the message to encrypt

See:

- wc_Des3_SetKey
- wc_Des3_CbcDecrypt

Return: 0 Returned upon successfully encrypting the given input message

3

Example

```
Des3 enc; // Des3 structure used for encryption
// initialize enc with wc_Des3_SetKey, use mode DES_ENCRYPTION

byte plain[]  = { // initialize with message };
byte cipher[sizeof(plain)];

if ( wc_Des3_CbcEncrypt(&enc, cipher, plain, sizeof(plain)) != 0) {
    // error encrypting message
}
```

C.16.2.10 function wc_Des3_CbcDecrypt

```
int wc_Des3_CbcDecrypt(
    Des3 * des,
    byte * out,
    const byte * in,
    word32 sz
)
```

This function decrypts the input ciphertext, in, and stores the resulting plaintext in the output buffer, out. It uses Triple Des (3DES) encryption with cipher block chaining (CBC) mode.

Parameters:

- **des** pointer to the Des3 structure to use for decryption
- **out** pointer to the buffer in which to store the decrypted plaintext
- **in** pointer to the input buffer containing the encrypted ciphertext
- **sz** length of the ciphertext to decrypt

See:

- wc_Des3_SetKey
- wc_Des3_CbcEncrypt

Return: 0 Returned upon successfully decrypting the given ciphertext

3

Example

```
Des3 dec; // Des structure used for decryption
// initialize dec with wc_Des3_SetKey, use mode DES_DECRYPTION

byte cipher[]  = { // initialize with ciphertext };
byte decoded[sizeof(cipher)];

if ( wc_Des3_CbcDecrypt(&dec, decoded, cipher, sizeof(cipher)) != 0) {
    // error decrypting message
}
```

C.16.3 Source code

```
int  wc_Des_SetKey(Des* des, const byte* key,
                               const byte* iv, int dir);

void wc_Des_SetIV(Des* des, const byte* iv);

int  wc_Des_CbcEncrypt(Des* des, byte* out,
```

```
                                    const byte* in, word32 sz);

int  wc_Des_CbcDecrypt(Des* des, byte* out,
                                    const byte* in, word32 sz);

int  wc_Des_EcbEncrypt(Des* des, byte* out,
                                    const byte* in, word32 sz);

int wc_Des3_EcbEncrypt(Des3* des, byte* out,
                                    const byte* in, word32 sz);

int  wc_Des3_SetKey(Des3* des, const byte* key,
                                 const byte* iv,int dir);

int  wc_Des3_SetIV(Des3* des, const byte* iv);

int  wc_Des3_CbcEncrypt(Des3* des, byte* out,
                                     const byte* in,word32 sz);

int  wc_Des3_CbcDecrypt(Des3* des, byte* out,
                                     const byte* in,word32 sz);
```

C.17 dox_comments/header_files/dh.h

C.17.1 Functions

	Name
int	**wc_InitDhKey**(DhKey * key)This function initializes a Diffie-Hellman key for use in negotiating a secure secret key with the Diffie-Hellman exchange protocol.
void	**wc_FreeDhKey**(DhKey * key)This function frees a Diffie-Hellman key after it has been used to negotiate a secure secret key with the Diffie-Hellman exchange protocol.
int	**wc_DhGenerateKeyPair**(DhKey * key, WC_RNG * rng, byte * priv, word32 * privSz, byte * pub, word32 * pubSz)This function generates a public/private key pair based on the Diffie-Hellman public parameters, storing the private key in priv and the public key in pub. It takes an initialized Diffie-Hellman key and an initialized rng structure.

	Name
int	**wc_DhAgree**(DhKey * key, byte * agree, word32 * agreeSz, const byte * priv, word32 privSz, const byte * otherPub, word32 pubSz)This function generates an agreed upon secret key based on a local private key and a received public key. If completed on both sides of an exchange, this function generates an agreed upon secret key for symmetric communication. On successfully generating a shared secret key, the size of the secret key written will be stored in agreeSz.
int	**wc_DhKeyDecode**(const byte * input, word32 * inOutIdx, DhKey * key, word32)This function decodes a Diffie-Hellman key from the given input buffer containing the key in DER format. It stores the result in the DhKey structure.
int	**wc_DhSetKey**(DhKey * key, const byte * p, word32 pSz, const byte * g, word32 gSz)This function sets the key for a DhKey structure using the input private key parameters. Unlike wc_DhKeyDecode, this function does not require that the input key be formatted in DER format, and instead simply accepts the parsed input parameters p (prime) and g (base).
int	**wc_DhParamsLoad**(const byte * input, word32 inSz, byte * p, word32 * pInOutSz, byte * g, word32 * gInOutSz)This function loads the Diffie_Hellman parameters, p (prime) and g (base) out of the given input buffer, DER formatted.
const DhParams *	**wc_Dh_ffdhe2048_Get**(void)This function returns ... and requires that HAVE_FFDHE_2048 be defined.
const DhParams *	**wc_Dh_ffdhe3072_Get**(void)This function returns ... and requires that HAVE_FFDHE_3072 be defined.
const DhParams *	**wc_Dh_ffdhe4096_Get**(void)This function returns ... and requires that HAVE_FFDHE_4096 be defined.
const DhParams *	**wc_Dh_ffdhe6144_Get**(void)This function returns ... and requires that HAVE_FFDHE_6144 be defined.
const DhParams *	**wc_Dh_ffdhe8192_Get**(void)This function returns ... and requires that HAVE_FFDHE_8192 be defined.
int	**wc_DhCheckKeyPair**(DhKey * key, const byte * pub, word32 pubSz, const byte * priv, word32 privSz)Checks DH keys for pair_wise consistency per process in SP 800_56Ar3, section 5.6.2.1.4, method (b) for FFC.

	Name
int	**wc_DhCheckPrivKey**(DhKey * key, const byte * priv, word32 pubSz)Check DH private key for invalid numbers.
int	**wc_DhCheckPrivKey_ex**(DhKey * key, const byte * priv, word32 pubSz, const byte * prime, word32 primeSz)
int	**wc_DhCheckPubKey**(DhKey * key, const byte * pub, word32 pubSz)
int	**wc_DhCheckPubKey_ex**(DhKey * key, const byte * pub, word32 pubSz, const byte * prime, word32 primeSz)
int	**wc_DhExportParamsRaw**(DhKey * dh, byte * p, word32 * pSz, byte * q, word32 * qSz, byte * g, word32 * gSz)
int	**wc_DhGenerateParams**(WC_RNG * rng, int modSz, DhKey * dh)
int	**wc_DhSetCheckKey**(DhKey * key, const byte * p, word32 pSz, const byte * g, word32 gSz, const byte * q, word32 qSz, int trusted, WC_RNG * rng)
int	**wc_DhSetKey_ex**(DhKey * key, const byte * p, word32 pSz, const byte * g, word32 gSz, const byte * q, word32 qSz)

C.17.2 Functions Documentation

C.17.2.1 function wc_InitDhKey

```
int wc_InitDhKey(
    DhKey * key
)
```

This function initializes a Diffie-Hellman key for use in negotiating a secure secret key with the Diffie-Hellman exchange protocol.

Parameters:

- **key** pointer to the DhKey structure to initialize for use with secure key exchanges

See:

- wc_FreeDhKey
- wc_DhGenerateKeyPair

Return: none No returns.

Example

```
DhKey key;
wc_InitDhKey(&key); // initialize DH key
```

C.17.2.2 function wc_FreeDhKey

```
void wc_FreeDhKey(
    DhKey * key
)
```

This function frees a Diffie-Hellman key after it has been used to negotiate a secure secret key with the Diffie-Hellman exchange protocol.

Parameters:

- **key** pointer to the DhKey structure to free

See: wc_InitDhKey

Return: none No returns.

Example

```
DhKey key;
// initialize key, perform key exchange

wc_FreeDhKey(&key); // free DH key to avoid memory leaks
```

C.17.2.3 function wc_DhGenerateKeyPair

```
int wc_DhGenerateKeyPair(
    DhKey * key,
    WC_RNG * rng,
    byte * priv,
    word32 * privSz,
    byte * pub,
    word32 * pubSz
)
```

This function generates a public/private key pair based on the Diffie-Hellman public parameters, storing the private key in priv and the public key in pub. It takes an initialized Diffie-Hellman key and an initialized rng structure.

Parameters:

- **key** pointer to the DhKey structure from which to generate the key pair
- **rng** pointer to an initialized random number generator (rng) with which to generate the keys
- **priv** pointer to a buffer in which to store the private key
- **privSz** will store the size of the private key written to priv
- **pub** pointer to a buffer in which to store the public key
- **pubSz** will store the size of the private key written to pub

See:

- wc_InitDhKey
- wc_DhSetKey
- wc_DhKeyDecode

Return:

- BAD_FUNC_ARG Returned if there is an error parsing one of the inputs to this function
- RNG_FAILURE_E Returned if there is an error generating a random number using rng
- MP_INIT_E May be returned if there is an error in the math library while generating the public key
- MP_READ_E May be returned if there is an error in the math library while generating the public key
- MP_EXPTMOD_E May be returned if there is an error in the math library while generating the public key
- MP_TO_E May be returned if there is an error in the math library while generating the public key

Example

```
DhKey key;
int ret;
byte priv[256];
byte pub[256];
word32 privSz, pubSz;

wc_InitDhKey(&key); // initialize key
// Set DH parameters using wc_DhSetKey or wc_DhKeyDecode
WC_RNG rng;
wc_InitRng(&rng); // initialize rng
ret = wc_DhGenerateKeyPair(&key, &rng, priv, &privSz, pub, &pubSz);
```

C.17.2.4 function wc_DhAgree

```
int wc_DhAgree(
    DhKey * key,
    byte * agree,
    word32 * agreeSz,
    const byte * priv,
    word32 privSz,
    const byte * otherPub,
    word32 pubSz
)
```

This function generates an agreed upon secret key based on a local private key and a received public key. If completed on both sides of an exchange, this function generates an agreed upon secret key for symmetric communication. On successfully generating a shared secret key, the size of the secret key written will be stored in agreeSz.

Parameters:

- **key** pointer to the DhKey structure to use to compute the shared key
- **agree** pointer to the buffer in which to store the secret key
- **agreeSz** will hold the size of the secret key after successful generation
- **priv** pointer to the buffer containing the local secret key
- **privSz** size of the local secret key
- **otherPub** pointer to a buffer containing the received public key
- **pubSz** size of the received public key

See: wc_DhGenerateKeyPair

Return:

- 0 Returned on successfully generating an agreed upon secret key
- MP_INIT_E May be returned if there is an error while generating the shared secret key
- MP_READ_E May be returned if there is an error while generating the shared secret key
- MP_EXPTMOD_E May be returned if there is an error while generating the shared secret key
- MP_TO_E May be returned if there is an error while generating the shared secret key

Example

```
DhKey key;
int ret;
byte priv[256];
byte agree[256];
word32 agreeSz;

// initialize key, set key prime and base
```

```
// wc_DhGenerateKeyPair -- store private key in priv
byte pub[] = { // initialized with the received public key };
ret = wc_DhAgree(&key, agree, &agreeSz, priv, sizeof(priv), pub,
sizeof(pub));
if ( ret != 0 ) {
    // error generating shared key
}
```

C.17.2.5 function wc_DhKeyDecode

```
int wc_DhKeyDecode(
    const byte * input,
    word32 * inOutIdx,
    DhKey * key,
    word32
)
```

This function decodes a Diffie-Hellman key from the given input buffer containing the key in DER format. It stores the result in the DhKey structure.

Parameters:

- **input** pointer to the buffer containing the DER formatted Diffie-Hellman key
- **inOutIdx** pointer to an integer in which to store the index parsed to while decoding the key
- **key** pointer to the DhKey structure to initialize with the input key
- **inSz** length of the input buffer. Gives the max length that may be read

See: wc_DhSetKey

Return:

- 0 Returned on successfully decoding the input key
- ASN_PARSE_E Returned if there is an error parsing the sequence of the input
- ASN_DH_KEY_E Returned if there is an error reading the private key parameters from the parsed input

Example

```
DhKey key;
word32 idx = 0;

byte keyBuff[1024];
// initialize with DER formatted key
wc_DhKeyInit(&key);
ret = wc_DhKeyDecode(keyBuff, &idx, &key, sizeof(keyBuff));

if ( ret != 0 ) {
    // error decoding key
}
```

C.17.2.6 function wc_DhSetKey

```
int wc_DhSetKey(
    DhKey * key,
    const byte * p,
    word32 pSz,
    const byte * g,
```

```
    word32 gSz
)
```

This function sets the key for a DhKey structure using the input private key parameters. Unlike wc_DhKeyDecode, this function does not require that the input key be formatted in DER format, and instead simply accepts the parsed input parameters p (prime) and g (base).

Parameters:

- **key** pointer to the DhKey structure on which to set the key
- **p** pointer to the buffer containing the prime for use with the key
- **pSz** length of the input prime
- **g** pointer to the buffer containing the base for use with the key
- **gSz** length of the input base

See: wc_DhKeyDecode

Return:

- 0 Returned on successfully setting the key
- BAD_FUNC_ARG Returned if any of the input parameters evaluate to NULL
- MP_INIT_E Returned if there is an error initializing the key parameters for storage
- ASN_DH_KEY_E Returned if there is an error reading in the DH key parameters p and g

Example

```
DhKey key;

byte p[] = { // initialize with prime };
byte g[] = { // initialize with base };
wc_DhKeyInit(&key);
ret = wc_DhSetKey(key, p, sizeof(p), g, sizeof(g));

if ( ret != 0 ) {
    // error setting key
}
```

C.17.2.7 function wc_DhParamsLoad

```
int wc_DhParamsLoad(
    const byte * input,
    word32 inSz,
    byte * p,
    word32 * pInOutSz,
    byte * g,
    word32 * gInOutSz
)
```

This function loads the Diffie-Hellman parameters, p (prime) and g (base) out of the given input buffer, DER formatted.

Parameters:

- **input** pointer to a buffer containing a DER formatted Diffie-Hellman certificate to parse
- **inSz** size of the input buffer
- **p** pointer to a buffer in which to store the parsed prime
- **pInOutSz** pointer to a word32 object containing the available size in the p buffer. Will be overwritten with the number of bytes written to the buffer after completing the function call
- **g** pointer to a buffer in which to store the parsed base

- **gInOutSz** pointer to a word32 object containing the available size in the g buffer. Will be overwritten with the number of bytes written to the buffer after completing the function call

See:

- wc_DhSetKey
- wc_DhKeyDecode

Return:

- 0 Returned on successfully extracting the DH parameters
- ASN_PARSE_E Returned if an error occurs while parsing the DER formatted DH certificate
- BUFFER_E Returned if there is inadequate space in p or g to store the parsed parameters

Example

```
byte dhCert[] = { initialize with DER formatted certificate };
byte p[MAX_DH_SIZE];
byte g[MAX_DH_SIZE];
word32 pSz = MAX_DH_SIZE;
word32 gSz = MAX_DH_SIZE;

ret = wc_DhParamsLoad(dhCert, sizeof(dhCert), p, &pSz, g, &gSz);
if ( ret != 0 ) {
    // error parsing inputs
}
```

C.17.2.8 function wc_Dh_ffdhe2048_Get

```
const DhParams * wc_Dh_ffdhe2048_Get(
    void
)
```

This function returns ... and requires that HAVE_FFDHE_2048 be defined.

See:

- wc_Dh_ffdhe3072_Get
- wc_Dh_ffdhe4096_Get
- wc_Dh_ffdhe6144_Get
- wc_Dh_ffdhe8192_Get

C.17.2.9 function wc_Dh_ffdhe3072_Get

```
const DhParams * wc_Dh_ffdhe3072_Get(
    void
)
```

This function returns ... and requires that HAVE_FFDHE_3072 be defined.

See:

- wc_Dh_ffdhe2048_Get
- wc_Dh_ffdhe4096_Get
- wc_Dh_ffdhe6144_Get
- wc_Dh_ffdhe8192_Get

C.17.2.10 function wc_Dh_ffdhe4096_Get

```
const DhParams * wc_Dh_ffdhe4096_Get(
    void
)
```

This function returns ... and requires that HAVE_FFDHE_4096 be defined.

See:

- wc_Dh_ffdhe2048_Get
- wc_Dh_ffdhe3072_Get
- wc_Dh_ffdhe6144_Get
- wc_Dh_ffdhe8192_Get

C.17.2.11 function wc_Dh_ffdhe6144_Get

```
const DhParams * wc_Dh_ffdhe6144_Get(
    void
)
```

This function returns ... and requires that HAVE_FFDHE_6144 be defined.

See:

- wc_Dh_ffdhe2048_Get
- wc_Dh_ffdhe3072_Get
- wc_Dh_ffdhe4096_Get
- wc_Dh_ffdhe8192_Get

C.17.2.12 function wc_Dh_ffdhe8192_Get

```
const DhParams * wc_Dh_ffdhe8192_Get(
    void
)
```

This function returns ... and requires that HAVE_FFDHE_8192 be defined.

See:

- wc_Dh_ffdhe2048_Get
- wc_Dh_ffdhe3072_Get
- wc_Dh_ffdhe4096_Get
- wc_Dh_ffdhe6144_Get

C.17.2.13 function wc_DhCheckKeyPair

```
int wc_DhCheckKeyPair(
    DhKey * key,
    const byte * pub,
    word32 pubSz,
    const byte * priv,
    word32 privSz
)
```

Checks DH keys for pair-wise consistency per process in SP 800-56Ar3, section 5.6.2.1.4, method (b) for FFC.

C.17.2.14 function wc_DhCheckPrivKey

```
int wc_DhCheckPrivKey(
    DhKey * key,
    const byte * priv,
    word32 pubSz
)
```

Check DH private key for invalid numbers.

C.17.2.15 function wc_DhCheckPrivKey_ex

```
int wc_DhCheckPrivKey_ex(
    DhKey * key,
    const byte * priv,
    word32 pubSz,
    const byte * prime,
    word32 primeSz
)
```

C.17.2.16 function wc_DhCheckPubKey

```
int wc_DhCheckPubKey(
    DhKey * key,
    const byte * pub,
    word32 pubSz
)
```

C.17.2.17 function wc_DhCheckPubKey_ex

```
int wc_DhCheckPubKey_ex(
    DhKey * key,
    const byte * pub,
    word32 pubSz,
    const byte * prime,
    word32 primeSz
)
```

C.17.2.18 function wc_DhExportParamsRaw

```
int wc_DhExportParamsRaw(
    DhKey * dh,
    byte * p,
    word32 * pSz,
    byte * q,
    word32 * qSz,
    byte * g,
    word32 * gSz
)
```

C.17.2.19 function wc_DhGenerateParams

```
int wc_DhGenerateParams(
    WC_RNG * rng,
    int modSz,
```

```
    DhKey * dh
)
```

C.17.2.20 function wc_DhSetCheckKey

```
int wc_DhSetCheckKey(
    DhKey * key,
    const byte * p,
    word32 pSz,
    const byte * g,
    word32 gSz,
    const byte * q,
    word32 qSz,
    int trusted,
    WC_RNG * rng
)
```

C.17.2.21 function wc_DhSetKey_ex

```
int wc_DhSetKey_ex(
    DhKey * key,
    const byte * p,
    word32 pSz,
    const byte * g,
    word32 gSz,
    const byte * q,
    word32 qSz
)
```

C.17.3 Source code

```
int wc_InitDhKey(DhKey* key);

void wc_FreeDhKey(DhKey* key);

int wc_DhGenerateKeyPair(DhKey* key, WC_RNG* rng, byte* priv,
                                 word32* privSz, byte* pub, word32* pubSz);

int wc_DhAgree(DhKey* key, byte* agree, word32* agreeSz,
                       const byte* priv, word32 privSz, const byte* otherPub,
                       word32 pubSz);

int wc_DhKeyDecode(const byte* input, word32* inOutIdx, DhKey* key,
                           word32);

int wc_DhSetKey(DhKey* key, const byte* p, word32 pSz, const byte* g,
                        word32 gSz);

int wc_DhParamsLoad(const byte* input, word32 inSz, byte* p,
                            word32* pInOutSz, byte* g, word32* gInOutSz);

const DhParams* wc_Dh_ffdhe2048_Get(void);
```

```
const DhParams* wc_Dh_ffdhe3072_Get(void);

const DhParams* wc_Dh_ffdhe4096_Get(void);

const DhParams* wc_Dh_ffdhe6144_Get(void);

const DhParams* wc_Dh_ffdhe8192_Get(void);

int wc_DhCheckKeyPair(DhKey* key, const byte* pub, word32 pubSz,
                      const byte* priv, word32 privSz);

int wc_DhCheckPrivKey(DhKey* key, const byte* priv, word32 pubSz);

int wc_DhCheckPrivKey_ex(DhKey* key, const byte* priv, word32 pubSz,
                         const byte* prime, word32 primeSz);

int wc_DhCheckPubKey(DhKey* key, const byte* pub, word32 pubSz);

int wc_DhCheckPubKey_ex(DhKey* key, const byte* pub, word32 pubSz,
                         const byte* prime, word32 primeSz);

int wc_DhExportParamsRaw(DhKey* dh, byte* p, word32* pSz,
                       byte* q, word32* qSz, byte* g, word32* gSz);

int wc_DhGenerateParams(WC_RNG *rng, int modSz, DhKey *dh);

int wc_DhSetCheckKey(DhKey* key, const byte* p, word32 pSz,
                      const byte* g, word32 gSz, const byte* q, word32 qSz,
                      int trusted, WC_RNG* rng);

int wc_DhSetKey_ex(DhKey* key, const byte* p, word32 pSz,
                      const byte* g, word32 gSz, const byte* q, word32 qSz);

int wc_FreeDhKey(DhKey* key);
```

C.18 dox_comments/header_files/doxygen_groups.h

C.19 dox_comments/header_files/doxygen_pages.h

C.20 dox_comments/header_files/dsa.h

C.20.1 Functions

	Name
int	**wc_InitDsaKey**(DsaKey * key)This function initializes a DsaKey object in order to use it for authentication via the Digital Signature Algorithm (DSA).
void	**wc_FreeDsaKey**(DsaKey * key)This function frees a DsaKey object after it has been used.

	Name
int	**wc_DsaSign**(const byte * digest, byte * out, DsaKey * key, WC_RNG * rng)This function signs the input digest and stores the result in the output buffer, out.
int	**wc_DsaVerify**(const byte * digest, const byte * sig, DsaKey * key, int * answer)This function verifies the signature of a digest, given a private key. It stores whether the key properly verifies in the answer parameter, with 1 corresponding to a successful verification, and 0 corresponding to failed verification.
int	**wc_DsaPublicKeyDecode**(const byte * input, word32 * inOutIdx, DsaKey * key, word32 inSz)This function decodes a DER formatted certificate buffer containing a DSA public key, and stores the key in the given DsaKey structure. It also sets the inOutIdx parameter according to the length of the input read.
int	**wc_DsaPrivateKeyDecode**(const byte * input, word32 * inOutIdx, DsaKey * key, word32 inSz)This function decodes a DER formatted certificate buffer containing a DSA private key, and stores the key in the given DsaKey structure. It also sets the inOutIdx parameter according to the length of the input read.
int	**wc_DsaKeyToDer**(DsaKey * key, byte * output, word32 inLen)Convert DsaKey key to DER format, write to output (inLen), return bytes written.
int	**wc_MakeDsaKey**(WC_RNG * rng, DsaKey * dsa)Create a DSA key.
int	**wc_MakeDsaParameters**(WC_RNG * rng, int modulus_size, DsaKey * dsa)FIPS 186_4 defines valid for modulus_size values as (1024, 160) (2048, 256) (3072, 256)

C.20.2 Functions Documentation

C.20.2.1 function wc_InitDsaKey

```
int wc_InitDsaKey(
    DsaKey * key
)
```

This function initializes a DsaKey object in order to use it for authentication via the Digital Signature Algorithm (DSA).

Parameters:

- **key** pointer to the DsaKey structure to initialize

See: wc_FreeDsaKey

Return:

- 0 Returned on success.
- BAD_FUNC_ARG Returned if a NULL key is passed in.

Example

```
DsaKey key;
int ret;
ret = wc_InitDsaKey(&key); // initialize DSA key
```

C.20.2.2 function wc_FreeDsaKey

```
void wc_FreeDsaKey(
    DsaKey * key
)
```

This function frees a DsaKey object after it has been used.

Parameters:

- **key** pointer to the DsaKey structure to free

See: wc_FreeDsaKey

Return: none No returns.

Example

```
DsaKey key;
// initialize key, use for authentication
...
wc_FreeDsaKey(&key); // free DSA key
```

C.20.2.3 function wc_DsaSign

```
int wc_DsaSign(
    const byte * digest,
    byte * out,
    DsaKey * key,
    WC_RNG * rng
)
```

This function signs the input digest and stores the result in the output buffer, out.

Parameters:

- **digest** pointer to the hash to sign
- **out** pointer to the buffer in which to store the signature
- **key** pointer to the initialized DsaKey structure with which to generate the signature
- **rng** pointer to an initialized RNG to use with the signature generation

See: wc_DsaVerify

Return:

- 0 Returned on successfully signing the input digest
- MP_INIT_E may be returned if there is an error in processing the DSA signature.
- MP_READ_E may be returned if there is an error in processing the DSA signature.
- MP_CMP_E may be returned if there is an error in processing the DSA signature.
- MP_INVMOD_E may be returned if there is an error in processing the DSA signature.
- MP_EXPTMOD_E may be returned if there is an error in processing the DSA signature.
- MP_MOD_E may be returned if there is an error in processing the DSA signature.
- MP_MUL_E may be returned if there is an error in processing the DSA signature.

- MP_ADD_E may be returned if there is an error in processing the DSA signature.
- MP_MULMOD_E may be returned if there is an error in processing the DSA signature.
- MP_TO_E may be returned if there is an error in processing the DSA signature.
- MP_MEM may be returned if there is an error in processing the DSA signature.

Example

```
DsaKey key;
// initialize DSA key, load private Key
int ret;
WC_RNG rng;
wc_InitRng(&rng);
byte hash[] = { // initialize with hash digest };
byte signature[40]; // signature will be 40 bytes (320 bits)

ret = wc_DsaSign(hash, signature, &key, &rng);
if (ret != 0) {
    // error generating DSA signature
}
```

C.20.2.4 function wc_DsaVerify

```
int wc_DsaVerify(
    const byte * digest,
    const byte * sig,
    DsaKey * key,
    int * answer
)
```

This function verifies the signature of a digest, given a private key. It stores whether the key properly verifies in the answer parameter, with 1 corresponding to a successful verification, and 0 corresponding to failed verification.

Parameters:

- **digest** pointer to the digest containing the subject of the signature
- **sig** pointer to the buffer containing the signature to verify
- **key** pointer to the initialized DsaKey structure with which to verify the signature
- **answer** pointer to an integer which will store whether the verification was successful

See: wc_DsaSign

Return:

- 0 Returned on successfully processing the verify request. Note: this does not mean that the signature is verified, only that the function succeeded
- MP_INIT_E may be returned if there is an error in processing the DSA signature.
- MP_READ_E may be returned if there is an error in processing the DSA signature.
- MP_CMP_E may be returned if there is an error in processing the DSA signature.
- MP_INVMOD_E may be returned if there is an error in processing the DSA signature.
- MP_EXPTMOD_E may be returned if there is an error in processing the DSA signature.
- MP_MOD_E may be returned if there is an error in processing the DSA signature.
- MP_MUL_E may be returned if there is an error in processing the DSA signature.
- MP_ADD_E may be returned if there is an error in processing the DSA signature.
- MP_MULMOD_E may be returned if there is an error in processing the DSA signature.
- MP_TO_E may be returned if there is an error in processing the DSA signature.
- MP_MEM may be returned if there is an error in processing the DSA signature.

Example

```
DsaKey key;
// initialize DSA key, load public Key

int ret;
int verified;
byte hash[] = { // initialize with hash digest };
byte signature[] = { // initialize with signature to verify };
ret = wc_DsaVerify(hash, signature, &key, &verified);
if (ret != 0) {
    // error processing verify request
} else if (answer == 0) {
    // invalid signature
}
```

C.20.2.5 function wc_DsaPublicKeyDecode

```
int wc_DsaPublicKeyDecode(
    const byte * input,
    word32 * inOutIdx,
    DsaKey * key,
    word32 inSz
)
```

This function decodes a DER formatted certificate buffer containing a DSA public key, and stores the key in the given DsaKey structure. It also sets the inOutIdx parameter according to the length of the input read.

Parameters:

- **input** pointer to the buffer containing the DER formatted DSA public key
- **inOutIdx** pointer to an integer in which to store the final index of the certificate read
- **key** pointer to the DsaKey structure in which to store the public key
- **inSz** size of the input buffer

See:

- wc_InitDsaKey
- wc_DsaPrivateKeyDecode

Return:

- 0 Returned on successfully setting the public key for the DsaKey object
- ASN_PARSE_E Returned if there is an error in the encoding while reading the certificate buffer
- ASN_DH_KEY_E Returned if one of the DSA parameters is incorrectly formatted

Example

```
int ret, idx=0;

DsaKey key;
wc_InitDsaKey(&key);
byte derBuff[] = { // DSA public key};
ret = wc_DsaPublicKeyDecode(derBuff, &idx, &key, inSz);
if (ret != 0) {
    // error reading public key
}
```

C.20.2.6 function wc_DsaPrivateKeyDecode

```
int wc_DsaPrivateKeyDecode(
    const byte * input,
    word32 * inOutIdx,
    DsaKey * key,
    word32 inSz
)
```

This function decodes a DER formatted certificate buffer containing a DSA private key, and stores the key in the given DsaKey structure. It also sets the inOutIdx parameter according to the length of the input read.

Parameters:

- **input** pointer to the buffer containing the DER formatted DSA private key
- **inOutIdx** pointer to an integer in which to store the final index of the certificate read
- **key** pointer to the DsaKey structure in which to store the private key
- **inSz** size of the input buffer

See:

- wc_InitDsaKey
- wc_DsaPublicKeyDecode

Return:

- 0 Returned on successfully setting the private key for the DsaKey object
- ASN_PARSE_E Returned if there is an error in the encoding while reading the certificate buffer
- ASN_DH_KEY_E Returned if one of the DSA parameters is incorrectly formatted

Example

```
int ret, idx=0;

DsaKey key;
wc_InitDsaKey(&key);
byte derBuff[] = { // DSA private key };
ret = wc_DsaPrivateKeyDecode(derBuff, &idx, &key, inSz);
if (ret != 0) {
    // error reading private key
}
```

C.20.2.7 function wc_DsaKeyToDer

```
int wc_DsaKeyToDer(
    DsaKey * key,
    byte * output,
    word32 inLen
)
```

Convert DsaKey key to DER format, write to output (inLen), return bytes written.

Parameters:

- **key** Pointer to DsaKey structure to convert.
- **output** Pointer to output buffer for converted key.
- **inLen** Length of key input.

See:

- wc_InitDsaKey

- wc_FreeDsaKey
- wc_MakeDsaKey

Return:

- outLen Success, number of bytes written
- BAD_FUNC_ARG key or output are null or key->type is not DSA_PRIVATE.
- MEMORY_E Error allocating memory.

Example

```
DsaKey key;
WC_RNG rng;
int derSz;
int bufferSize = // Sufficient buffer size;
byte der[bufferSize];

wc_InitDsaKey(&key);
wc_InitRng(&rng);
wc_MakeDsaKey(&rng, &key);
derSz = wc_DsaKeyToDer(&key, der, bufferSize);
```

C.20.2.8 function wc_MakeDsaKey

```
int wc_MakeDsaKey(
    WC_RNG * rng,
    DsaKey * dsa
)
```

Create a DSA key.

Parameters:

- **rng** Pointer to WC_RNG structure.
- **dsa** Pointer to DsaKey structure.

See:

- wc_InitDsaKey
- wc_FreeDsaKey
- wc_DsaSign

Return:

- MP_OKAY Success
- BAD_FUNC_ARG Either rng or dsa is null.
- MEMORY_E Couldn't allocate memory for buffer.
- MP_INIT_E Error initializing mp_int

Example

```
WC_RNG rng;
DsaKey dsa;
wc_InitRng(&rng);
wc_InitDsa(&dsa);
if(wc_MakeDsaKey(&rng, &dsa) != 0)
{
    // Error creating key
}
```

C.20.2.9 function wc_MakeDsaParameters

```
int wc_MakeDsaParameters(
    WC_RNG * rng,
    int modulus_size,
    DsaKey * dsa
)
```

FIPS 186-4 defines valid for modulus_size values as (1024, 160) (2048, 256) (3072, 256)

Parameters:

- **rng** pointer to wolfCrypt rng.
- **modulus_size** 1024, 2048, or 3072 are valid values.
- **dsa** Pointer to a DsaKey structure.

See:

- wc_MakeDsaKey
- wc_DsaKeyToDer
- wc_InitDsaKey

Return:

- 0 Success
- BAD_FUNC_ARG rng or dsa is null or modulus_size is invalid.
- MEMORY_E Error attempting to allocate memory.

Example

```
DsaKey key;
WC_RNG rng;
wc_InitDsaKey(&key);
wc_InitRng(&rng);
if(wc_MakeDsaParameters(&rng, 1024, &genKey) != 0)
{
    // Handle error
}
```

C.20.3 Source code

```
int wc_InitDsaKey(DsaKey* key);

void wc_FreeDsaKey(DsaKey* key);

int wc_DsaSign(const byte* digest, byte* out,
                           DsaKey* key, WC_RNG* rng);

int wc_DsaVerify(const byte* digest, const byte* sig,
                             DsaKey* key, int* answer);

int wc_DsaPublicKeyDecode(const byte* input, word32* inOutIdx,
                                      DsaKey* key, word32 inSz);

int wc_DsaPrivateKeyDecode(const byte* input, word32* inOutIdx,
                                       DsaKey* key, word32 inSz);

int wc_DsaKeyToDer(DsaKey* key, byte* output, word32 inLen);
```

```
int wc_MakeDsaKey(WC_RNG *rng, DsaKey *dsa);

int wc_MakeDsaParameters(WC_RNG *rng, int modulus_size, DsaKey *dsa);
```

C.21 dox_comments/header_files/ecc.h

C.21.1 Functions

	Name
int	**wc_ecc_make_key**(WC_RNG * rng, int keysize, ecc_key * key)This function generates a new ecc_key and stores it in key.
int	**wc_ecc_make_key_ex**(WC_RNG * rng, int keysize, ecc_key * key, int curve_id)This function generates a new ecc_key and stores it in key.
int	**wc_ecc_check_key**(ecc_key * key)Perform sanity checks on ecc key validity.
void	**wc_ecc_key_free**(ecc_key * key)This function frees an ecc_key key after it has been used.
int	**wc_ecc_shared_secret**(ecc_key * private_key, ecc_key * public_key, byte * out, word32 * outlen)This function generates a new secret key using a local private key and a received public key. It stores this shared secret key in the buffer out and updates outlen to hold the number of bytes written to the output buffer.
int	**wc_ecc_shared_secret_ex**(ecc_key * private_key, ecc_point * point, byte * out, word32 * outlen)Create an ECC shared secret between private key and public point.
int	**wc_ecc_sign_hash**(const byte * in, word32 inlen, byte * out, word32 * outlen, WC_RNG * rng, ecc_key * key)This function signs a message digest using an ecc_key object to guarantee authenticity.
int	**wc_ecc_sign_hash_ex**(const byte * in, word32 inlen, WC_RNG * rng, ecc_key * key, mp_int * r, mp_int * s)Sign a message digest.
int	**wc_ecc_verify_hash**(const byte * sig, word32 siglen, const byte * hash, word32 hashlen, int * stat, ecc_key * key)This function verifies the ECC signature of a hash to ensure authenticity. It returns the answer through stat, with 1 corresponding to a valid signature, and 0 corresponding to an invalid signature.
int	**wc_ecc_verify_hash_ex**(mp_int * r, mp_int * s, const byte * hash, word32 hashlen, int * stat, ecc_key * key)Verify an ECC signature. Result is written to stat. 1 is valid, 0 is invalid. Note: Do not use the return value to test for valid. Only use stat.

	Name
int	**wc_ecc_init**(ecc_key * key)This function initializes an ecc_key object for future use with message verification or key negotiation.
int	**wc_ecc_init_ex**(ecc_key * key, void * heap, int devId)This function initializes an ecc_key object for future use with message verification or key negotiation.
ecc_key *	**wc_ecc_key_new**(void * heap)This function uses a user defined heap and allocates space for the key structure.
int	**wc_ecc_free**(ecc_key * key)This function frees an ecc_key object after it has been used.
void	**wc_ecc_fp_free**(void)This function frees the fixed_point cache, which can be used with ecc to speed up computation times. To use this functionality, FP_ECC (fixed_point ecc), should be defined. Threaded applications should call this function before exiting the thread.
int	**wc_ecc_is_valid_idx**(int n)Checks if an ECC idx is valid.
ecc_point *	**wc_ecc_new_point**(void)Allocate a new ECC point.
void	**wc_ecc_del_point**(ecc_point * p)Free an ECC point from memory.
int	**wc_ecc_copy_point**(ecc_point * p, ecc_point * r)Copy the value of one point to another one.
int	**wc_ecc_cmp_point**(ecc_point * a, ecc_point * b)Compare the value of a point with another one.
int	**wc_ecc_point_is_at_infinity**(ecc_point * p)Checks if a point is at infinity. Returns 1 if point is at infinity, 0 if not, < 0 on error.
int	**wc_ecc_mulmod**(mp_int * k, ecc_point * G, ecc_point * R, mp_int * a, mp_int * modulus, int map)Perform ECC Fixed Point multiplication.
int	**wc_ecc_export_x963**(ecc_key * key, byte * out, word32 * outLen)This function exports the ECC key from the ecc_key structure, storing the result in out. The key will be stored in ANSI X9.63 format. It stores the bytes written to the output buffer in outLen.
int	**wc_ecc_export_x963_ex**(ecc_key * key, byte * out, word32 * outLen, int compressed)This function exports the ECC key from the ecc_key structure, storing the result in out. The key will be stored in ANSI X9.63 format. It stores the bytes written to the output buffer in outLen. This function allows the additional option of compressing the certificate through the compressed parameter. When this parameter is true, the key will be stored in ANSI X9.63 compressed format.

	Name
int	**wc_ecc_import_x963**(const byte * in, word32 inLen, ecc_key * key)This function imports a public ECC key from a buffer containing the key stored in ANSI X9.63 format. This function will handle both compressed and uncompressed keys, as long as compressed keys are enabled at compile time through the HAVE_COMP_KEY option.
int	**wc_ecc_import_private_key**(const byte * priv, word32 privSz, const byte * pub, word32 pubSz, ecc_key * key)This function imports a public/private ECC key pair from a buffer containing the raw private key, and a second buffer containing the ANSI X9.63 formatted public key. This function will handle both compressed and uncompressed keys, as long as compressed keys are enabled at compile time through the HAVE_COMP_KEY option.
int	**wc_ecc_rs_to_sig**(const char * r, const char * s, byte * out, word32 * outlen)This function converts the R and S portions of an ECC signature into a DER-encoded ECDSA signature. This function also stores the length written to the output buffer, out, in outlen.
int	**wc_ecc_import_raw**(ecc_key * key, const char * qx, const char * qy, const char * d, const char * curveName)This function fills an ecc_key structure with the raw components of an ECC signature.
int	**wc_ecc_export_private_only**(ecc_key * key, byte * out, word32 * outLen)This function exports only the private key from an ecc_key structure. It stores the private key in the buffer out, and sets the bytes written to this buffer in outLen.
int	**wc_ecc_export_point_der**(const int curve_idx, ecc_point * point, byte * out, word32 * outLen)Export point to der.
int	**wc_ecc_import_point_der**(byte * in, word32 inLen, const int curve_idx, ecc_point * point)Import point from der format.
int	**wc_ecc_size**(ecc_key * key)This function returns the key size of an ecc_key structure in octets.
int	**wc_ecc_sig_size_calc**(int sz)This function returns the worst case size for an ECC signature, given by: (keySz * 2) + SIG_HEADER_SZ + ECC_MAX_PAD_SZ. The actual signature size can be computed with wc_ecc_sign_hash.

	Name
int	**wc_ecc_sig_size**(ecc_key * key)This function returns the worst case size for an ECC signature, given by: (keySz * 2) + SIG_HEADER_SZ + ECC_MAX_PAD_SZ. The actual signature size can be computed with wc_ecc_sign_hash.
ecEncCtx *	**wc_ecc_ctx_new**(int flags, WC_RNG * rng)This function allocates and initializes space for a new ECC context object to allow secure message exchange with ECC.
void	**wc_ecc_ctx_free**(ecEncCtx *)This function frees the ecEncCtx object used for encrypting and decrypting messages.
int	**wc_ecc_ctx_reset**(ecEncCtx * ctx, WC_RNG * rng)This function resets an ecEncCtx structure to avoid having to free and allocate a new context object.
int	**wc_ecc_ctx_set_algo**(ecEncCtx * ctx, byte encAlgo, byte kdfAlgo, byte macAlgo)This function can optionally be called after wc_ecc_ctx_new. It sets the encryption, KDF, and MAC algorithms into an ecEncCtx object.
const byte *	**wc_ecc_ctx_get_own_salt**(ecEncCtx *)This function returns the salt of an ecEncCtx object. This function should only be called when the ecEncCtx's state is ecSRV_INIT or ecCLI_INIT.
int	**wc_ecc_ctx_set_peer_salt**(ecEncCtx * ctx, const byte * salt)This function sets the peer salt of an ecEncCtx object.
int	**wc_ecc_ctx_set_kdf_salt**(ecEncCtx * ctx, const byte * salt, word32 sz)This function sets the salt pointer and length to use with KDF into the ecEncCtx object.
int	**wc_ecc_ctx_set_info**(ecEncCtx * ctx, const byte * info, int sz)This function can optionally be called before or after wc_ecc_ctx_set_peer_salt. It sets optional information for an ecEncCtx object.
int	**wc_ecc_encrypt**(ecc_key * privKey, ecc_key * pubKey, const byte * msg, word32 msgSz, byte * out, word32 * outSz, ecEncCtx * ctx)This function encrypts the given input message from msg to out. This function takes an optional ctx object as parameter. When supplied, encryption proceeds based on the ecEncCtx's encAlgo, kdfAlgo, and macAlgo. If ctx is not supplied, processing completes with the default algorithms, ecAES_128_CBC, ecHKDF_SHA256 and ecHMAC_SHA256. This function requires that the messages are padded according to the encryption type specified by ctx.

	Name
int	**wc_ecc_encrypt_ex**(ecc_key * privKey, ecc_key * pubKey, const byte * msg, word32 msgSz, byte * out, word32 * outSz, ecEncCtx * ctx, int compressed)This function encrypts the given input message from msg to out. This function takes an optional ctx object as parameter. When supplied, encryption proceeds based on the ecEncCtx's encAlgo, kdfAlgo, and macAlgo. If ctx is not supplied, processing completes with the default algorithms, ecAES_128_CBC, ecHKDF_SHA256 and ecHMAC_SHA256. This function requires that the messages are padded according to the encryption type specified by ctx.
int	**wc_ecc_decrypt**(ecc_key * privKey, ecc_key * pubKey, const byte * msg, word32 msgSz, byte * out, word32 * outSz, ecEncCtx * ctx)This function decrypts the ciphertext from msg to out. This function takes an optional ctx object as parameter. When supplied, encryption proceeds based on the ecEncCtx's encAlgo, kdfAlgo, and macAlgo. If ctx is not supplied, processing completes with the default algorithms, ecAES_128_CBC, ecHKDF_SHA256 and ecHMAC_SHA256. This function requires that the messages are padded according to the encryption type specified by ctx.
int	**wc_ecc_set_nonblock**(ecc_key * key, ecc_nb_ctx_t * ctx)Enable ECC support for non_blocking operations. Supported for Single Precision (SP) math with the following build options: WOLFSSL_SP_NONBLOCK WOLFSSL_SP_SMALL WOLFSSL_SP_NO_MALLOC WC_ECC_NONBLOCK.
int	**wc_ecc_set_curve**(ecc_key * key, int keysize, int curve_id)Compare a curve which has larger key than specified size or the curve matched curve ID, set a curve with smaller key size to the key.

C.21.2 Functions Documentation

C.21.2.1 function wc_ecc_make_key

```
int wc_ecc_make_key(
    WC_RNG * rng,
    int keysize,
    ecc_key * key
)
```

This function generates a new ecc_key and stores it in key.

Parameters:

- **rng** pointer to an initialized RNG object with which to generate the key

- **keysize** desired length for the ecc_key
- **key** pointer to the ecc_key for which to generate a key

See:

- wc_ecc_init
- wc_ecc_shared_secret

Return:

- 0 Returned on success.
- ECC_BAD_ARG_E Returned if rng or key evaluate to NULL
- BAD_FUNC_ARG Returned if the specified key size is not in the correct range of supported keys
- MEMORY_E Returned if there is an error allocating memory while computing the ecc key
- MP_INIT_E may be returned if there is an error while computing the ecc key
- MP_READ_E may be returned if there is an error while computing the ecc key
- MP_CMP_E may be returned if there is an error while computing the ecc key
- MP_INVMOD_E may be returned if there is an error while computing the ecc key
- MP_EXPTMOD_E may be returned if there is an error while computing the ecc key
- MP_MOD_E may be returned if there is an error while computing the ecc key
- MP_MUL_E may be returned if there is an error while computing the ecc key
- MP_ADD_E may be returned if there is an error while computing the ecc key
- MP_MULMOD_E may be returned if there is an error while computing the ecc key
- MP_TO_E may be returned if there is an error while computing the ecc key
- MP_MEM may be returned if there is an error while computing the ecc key

Example

```
ecc_key key;
wc_ecc_init(&key);
WC_RNG rng;
wc_InitRng(&rng);
wc_ecc_make_key(&rng, 32, &key); // initialize 32 byte ecc key
```

C.21.2.2 function wc_ecc_make_key_ex

```
int wc_ecc_make_key_ex(
    WC_RNG * rng,
    int keysize,
    ecc_key * key,
    int curve_id
)
```

This function generates a new ecc_key and stores it in key.

Parameters:

- **key** Pointer to store the created key.
- **keysize** size of key to be created in bytes, set based on curveId
- **rng** Rng to be used in key creation
- **curve_id** Curve to use for key

See:

- wc_ecc_make_key
- wc_ecc_get_curve_size_from_id

Return:

- 0 Returned on success.
- ECC_BAD_ARG_E Returned if rng or key evaluate to NULL

- BAD_FUNC_ARG Returned if the specified key size is not in the correct range of supported keys
- MEMORY_E Returned if there is an error allocating memory while computing the ecc key
- MP_INIT_E may be returned if there is an error while computing the ecc key
- MP_READ_E may be returned if there is an error while computing the ecc key
- MP_CMP_E may be returned if there is an error while computing the ecc key
- MP_INVMOD_E may be returned if there is an error while computing the ecc key
- MP_EXPTMOD_E may be returned if there is an error while computing the ecc key
- MP_MOD_E may be returned if there is an error while computing the ecc key
- MP_MUL_E may be returned if there is an error while computing the ecc key
- MP_ADD_E may be returned if there is an error while computing the ecc key
- MP_MULMOD_E may be returned if there is an error while computing the ecc key
- MP_TO_E may be returned if there is an error while computing the ecc key
- MP_MEM may be returned if there is an error while computing the ecc key

Example

```
ecc_key key;
int ret;
WC_RNG rng;
wc_ecc_init(&key);
wc_InitRng(&rng);
int curveId = ECC_SECP521R1;
int keySize = wc_ecc_get_curve_size_from_id(curveId);
ret = wc_ecc_make_key_ex(&rng, keySize, &key, curveId);
if (ret != MP_OKAY) {
    // error handling
}
```

C.21.2.3 function wc_ecc_check_key

```
int wc_ecc_check_key(
    ecc_key * key
)
```

Perform sanity checks on ecc key validity.

Parameters:

- **key** Pointer to key to check.

See: wc_ecc_point_is_at_infinity

Return:

- MP_OKAY Success, key is OK.
- BAD_FUNC_ARG Returns if key is NULL.
- ECC_INF_E Returns if wc_ecc_point_is_at_infinity returns 1.

Example

```
ecc_key key;
WC_RNG rng;
int check_result;
wc_ecc_init(&key);
wc_InitRng(&rng);
wc_ecc_make_key(&rng, 32, &key);
check_result = wc_ecc_check_key(&key);

if (check_result == MP_OKAY)
```

```
{
    // key check succeeded
}
else
{
    // key check failed
}
```

C.21.2.4 function wc_ecc_key_free

```
void wc_ecc_key_free(
    ecc_key * key
)
```

This function frees an ecc_key key after it has been used.

Parameters:

- **key** pointer to the ecc_key structure to free

See:

- wc_ecc_key_new
- wc_ecc_init_ex

Example

```
// initialize key and perform ECC operations
...
wc_ecc_key_free(&key);
```

C.21.2.5 function wc_ecc_shared_secret

```
int wc_ecc_shared_secret(
    ecc_key * private_key,
    ecc_key * public_key,
    byte * out,
    word32 * outlen
)
```

This function generates a new secret key using a local private key and a received public key. It stores this shared secret key in the buffer out and updates outlen to hold the number of bytes written to the output buffer.

Parameters:

- **private_key** pointer to the ecc_key structure containing the local private key
- **public_key** pointer to the ecc_key structure containing the received public key
- **out** pointer to an output buffer in which to store the generated shared secret key
- **outlen** pointer to the word32 object containing the length of the output buffer. Will be overwritten with the length written to the output buffer upon successfully generating a shared secret key

See:

- wc_ecc_init
- wc_ecc_make_key

Return:

- 0 Returned upon successfully generating a shared secret key

- BAD_FUNC_ARG Returned if any of the input parameters evaluate to NULL
- ECC_BAD_ARG_E Returned if the type of the private key given as argument, private_key, is not ECC_PRIVATEKEY, or if the public and private key types (given by ecc->dp) are not equivalent
- MEMORY_E Returned if there is an error generating a new ecc point
- BUFFER_E Returned if the generated shared secret key is too long to store in the provided buffer
- MP_INIT_E may be returned if there is an error while computing the shared key
- MP_READ_E may be returned if there is an error while computing the shared key
- MP_CMP_E may be returned if there is an error while computing the shared key
- MP_INVMOD_E may be returned if there is an error while computing the shared key
- MP_EXPTMOD_E may be returned if there is an error while computing the shared key
- MP_MOD_E may be returned if there is an error while computing the shared key
- MP_MUL_E may be returned if there is an error while computing the shared key
- MP_ADD_E may be returned if there is an error while computing the shared key
- MP_MULMOD_E may be returned if there is an error while computing the shared key
- MP_TO_E may be returned if there is an error while computing the shared key
- MP_MEM may be returned if there is an error while computing the shared key

Example

```
ecc_key priv, pub;
WC_RNG rng;
byte secret[1024]; // can hold 1024 byte shared secret key
word32 secretSz = sizeof(secret);
int ret;

wc_InitRng(&rng); // initialize rng
wc_ecc_init(&priv); // initialize key
wc_ecc_make_key(&rng, 32, &priv); // make public/private key pair
// receive public key, and initialise into pub
ret = wc_ecc_shared_secret(&priv, &pub, secret, &secretSz);
// generate secret key
if ( ret != 0 ) {
    // error generating shared secret key
}
```

C.21.2.6 function wc_ecc_shared_secret_ex

```
int wc_ecc_shared_secret_ex(
    ecc_key * private_key,
    ecc_point * point,
    byte * out,
    word32 * outlen
)
```

Create an ECC shared secret between private key and public point.

Parameters:

- **private_key** The private ECC key.
- **point** The point to use (public key).
- **out** Output destination of the shared secret. Conforms to EC-DH from ANSI X9.63.
- **outlen** Input the max size and output the resulting size of the shared secret.

See: wc_ecc_verify_hash_ex

Return:

- MP_OKAY Indicates success.

- BAD_FUNC_ARG Error returned when any arguments are null.
- ECC_BAD_ARG_E Error returned if private_key->type is not ECC_PRIVATEKEY or private_key->idx fails to validate.
- BUFFER_E Error when outlen is too small.
- MEMORY_E Error to create a new point.
- MP_VAL possible when an initialization failure occurs.
- MP_MEM possible when an initialization failure occurs.

Example

```
ecc_key key;
ecc_point* point;
byte shared_secret[];
int secret_size;
int result;

point = wc_ecc_new_point();

result = wc_ecc_shared_secret_ex(&key, point,
&shared_secret, &secret_size);

if (result != MP_OKAY)
{
    // Handle error
}
```

C.21.2.7 function wc_ecc_sign_hash

```
int wc_ecc_sign_hash(
    const byte * in,
    word32 inlen,
    byte * out,
    word32 * outlen,
    WC_RNG * rng,
    ecc_key * key
)
```

This function signs a message digest using an ecc_key object to guarantee authenticity.

Parameters:

- **in** pointer to the buffer containing the message hash to sign
- **inlen** length of the message hash to sign
- **out** buffer in which to store the generated signature
- **outlen** max length of the output buffer. Will store the bytes written to out upon successfully generating a message signature
- **key** pointer to a private ECC key with which to generate the signature

See: wc_ecc_verify_hash

Return:

- 0 Returned upon successfully generating a signature for the message digest
- BAD_FUNC_ARG Returned if any of the input parameters evaluate to NULL, or if the output buffer is too small to store the generated signature
- ECC_BAD_ARG_E Returned if the input key is not a private key, or if the ECC OID is invalid
- RNG_FAILURE_E Returned if the rng cannot successfully generate a satisfactory key
- MP_INIT_E may be returned if there is an error while computing the message signature

- MP_READ_E may be returned if there is an error while computing the message signature
- MP_CMP_E may be returned if there is an error while computing the message signature
- MP_INVMOD_E may be returned if there is an error while computing the message signature
- MP_EXPTMOD_E may be returned if there is an error while computing the message signature
- MP_MOD_E may be returned if there is an error while computing the message signature
- MP_MUL_E may be returned if there is an error while computing the message signature
- MP_ADD_E may be returned if there is an error while computing the message signature
- MP_MULMOD_E may be returned if there is an error while computing the message signature
- MP_TO_E may be returned if there is an error while computing the message signature
- MP_MEM may be returned if there is an error while computing the message signature

Example

```
ecc_key key;
WC_RNG rng;
int ret, sigSz;

byte sig[512]; // will hold generated signature
sigSz = sizeof(sig);
byte digest[] = { // initialize with message hash };
wc_InitRng(&rng); // initialize rng
wc_ecc_init(&key); // initialize key
wc_ecc_make_key(&rng, 32, &key); // make public/private key pair
ret = wc_ecc_sign_hash(digest, sizeof(digest), sig, &sigSz, &key);
if ( ret != 0 ) {
    // error generating message signature
}
```

C.21.2.8 function wc_ecc_sign_hash_ex

```
int wc_ecc_sign_hash_ex(
    const byte * in,
    word32 inlen,
    WC_RNG * rng,
    ecc_key * key,
    mp_int * r,
    mp_int * s
)
```

Sign a message digest.

Parameters:

- **in** The message digest to sign.
- **inlen** The length of the digest.
- **rng** Pointer to WC_RNG struct.
- **key** A private ECC key.
- **r** The destination for r component of the signature.
- **s** The destination for s component of the signature.

See: wc_ecc_verify_hash_ex

Return:

- MP_OKAY Returned upon successfully generating a signature for the message digest
- ECC_BAD_ARG_E Returned if the input key is not a private key, or if the ECC IDX is invalid, or if any of the input parameters evaluate to NULL, or if the output buffer is too small to store the generated signature

- RNG_FAILURE_E Returned if the rng cannot successfully generate a satisfactory key
- MP_INIT_E may be returned if there is an error while computing the message signature
- MP_READ_E may be returned if there is an error while computing the message signature
- MP_CMP_E may be returned if there is an error while computing the message signature
- MP_INVMOD_E may be returned if there is an error while computing the message signature
- MP_EXPTMOD_E may be returned if there is an error while computing the message signature
- MP_MOD_E may be returned if there is an error while computing the message signature
- MP_MUL_E may be returned if there is an error while computing the message signature
- MP_ADD_E may be returned if there is an error while computing the message signature
- MP_MULMOD_E may be returned if there is an error while computing the message signature
- MP_TO_E may be returned if there is an error while computing the message signature
- MP_MEM may be returned if there is an error while computing the message signature

Example

```
ecc_key key;
WC_RNG rng;
int ret, sigSz;
mp_int r; // destination for r component of signature.
mp_int s; // destination for s component of signature.

byte sig[512]; // will hold generated signature
sigSz = sizeof(sig);
byte digest[] = { initialize with message hash };
wc_InitRng(&rng); // initialize rng
wc_ecc_init(&key); // initialize key
mp_init(&r); // initialize r component
mp_init(&s); // initialize s component
wc_ecc_make_key(&rng, 32, &key); // make public/private key pair
ret = wc_ecc_sign_hash_ex(digest, sizeof(digest), &rng, &key, &r, &s);

if ( ret != MP_OKAY ) {
    // error generating message signature
}
```

C.21.2.9 function wc_ecc_verify_hash

```
int wc_ecc_verify_hash(
    const byte * sig,
    word32 siglen,
    const byte * hash,
    word32 hashlen,
    int * stat,
    ecc_key * key
)
```

This function verifies the ECC signature of a hash to ensure authenticity. It returns the answer through stat, with 1 corresponding to a valid signature, and 0 corresponding to an invalid signature.

Parameters:

- **sig** pointer to the buffer containing the signature to verify
- **siglen** length of the signature to verify
- **hash** pointer to the buffer containing the hash of the message verified
- **hashlen** length of the hash of the message verified
- **stat** pointer to the result of the verification. 1 indicates the message was successfully verified
- **key** pointer to a public ECC key with which to verify the signature

See:

- wc_ecc_sign_hash
- wc_ecc_verify_hash_ex

Return:

- 0 Returned upon successfully performing the signature verification. Note: This does not mean that the signature is verified. The authenticity information is stored instead in stat
- BAD_FUNC_ARG Returned any of the input parameters evaluate to NULL
- MEMORY_E Returned if there is an error allocating memory
- MP_INIT_E may be returned if there is an error while computing the message signature
- MP_READ_E may be returned if there is an error while computing the message signature
- MP_CMP_E may be returned if there is an error while computing the message signature
- MP_INVMOD_E may be returned if there is an error while computing the message signature
- MP_EXPTMOD_E may be returned if there is an error while computing the message signature
- MP_MOD_E may be returned if there is an error while computing the message signature
- MP_MUL_E may be returned if there is an error while computing the message signature
- MP_ADD_E may be returned if there is an error while computing the message signature
- MP_MULMOD_E may be returned if there is an error while computing the message signature
- MP_TO_E may be returned if there is an error while computing the message signature
- MP_MEM may be returned if there is an error while computing the message signature

Example

```
ecc_key key;
int ret, verified = 0;

byte sig[1024] { initialize with received signature };
byte digest[] = { initialize with message hash };
// initialize key with received public key
ret = wc_ecc_verify_hash(sig, sizeof(sig), digest,sizeof(digest),
&verified, &key);
if ( ret != 0 ) {
    // error performing verification
} else if ( verified == 0 ) {
    // the signature is invalid
}
```

C.21.2.10 function wc_ecc_verify_hash_ex

```
int wc_ecc_verify_hash_ex(
    mp_int * r,
    mp_int * s,
    const byte * hash,
    word32 hashlen,
    int * stat,
    ecc_key * key
)
```

Verify an ECC signature. Result is written to stat. 1 is valid, 0 is invalid. Note: Do not use the return value to test for valid. Only use stat.

Parameters:

- **r** The signature R component to verify
- **s** The signature S component to verify
- **hash** The hash (message digest) that was signed

- **hashlen** The length of the hash (octets)
- **stat** Result of signature, 1==valid, 0==invalid
- **key** The corresponding public ECC key

See: wc_ecc_verify_hash

Return:

- MP_OKAY If successful (even if the signature is not valid)
- ECC_BAD_ARG_E Returns if arguments are null or if key-idx is invalid.
- MEMORY_E Error allocating ints or points.

Example

```
mp_int r;
mp_int s;
int stat;
byte hash[] = { Some hash }
ecc_key key;

if(wc_ecc_verify_hash_ex(&r, &s, hash, hashlen, &stat, &key) == MP_OKAY)
{
    // Check stat
}
```

C.21.2.11 function wc_ecc_init

```
int wc_ecc_init(
    ecc_key * key
)
```

This function initializes an ecc_key object for future use with message verification or key negotiation.

Parameters:

- **key** pointer to the ecc_key object to initialize

See:

- wc_ecc_make_key
- wc_ecc_free

Return:

- 0 Returned upon successfully initializing the ecc_key object
- MEMORY_E Returned if there is an error allocating memory

Example

```
ecc_key key;
wc_ecc_init(&key);
```

C.21.2.12 function wc_ecc_init_ex

```
int wc_ecc_init_ex(
    ecc_key * key,
    void * heap,
    int devId
)
```

This function initializes an ecc_key object for future use with message verification or key negotiation.

Parameters:

- **key** pointer to the ecc_key object to initialize
- **heap** pointer to a heap identifier
- **devId** ID to use with crypto callbacks or async hardware. Set to INVALID_DEVID (-2) if not used

See:

- wc_ecc_make_key
- wc_ecc_free
- wc_ecc_init

Return:

- 0 Returned upon successfully initializing the ecc_key object
- MEMORY_E Returned if there is an error allocating memory

Example

```
ecc_key key;
wc_ecc_init_ex(&key, heap, devId);
```

C.21.2.13 function wc_ecc_key_new

```
ecc_key * wc_ecc_key_new(
    void * heap
)
```

This function uses a user defined heap and allocates space for the key structure.

See:

- wc_ecc_make_key
- wc_ecc_key_free
- wc_ecc_init

Return:

- 0 Returned upon successfully initializing the ecc_key object
- MEMORY_E Returned if there is an error allocating memory

Example

```
wc_ecc_key_new(&heap);
```

C.21.2.14 function wc_ecc_free

```
int wc_ecc_free(
    ecc_key * key
)
```

This function frees an ecc_key object after it has been used.

Parameters:

- **key** pointer to the ecc_key object to free

See: wc_ecc_init

Return: int integer returned indicating wolfSSL error or success status.

Example

```
// initialize key and perform secure exchanges
...
wc_ecc_free(&key);
```

C.21.2.15 function wc_ecc_fp_free

```
void wc_ecc_fp_free(
    void
)
```

This function frees the fixed-point cache, which can be used with ecc to speed up computation times. To use this functionality, FP_ECC (fixed-point ecc), should be defined. Threaded applications should call this function before exiting the thread.

Parameters:

- **none** No parameters.

See: wc_ecc_free

Return: none No returns.

Example

```
ecc_key key;
// initialize key and perform secure exchanges
...

wc_ecc_fp_free();
```

C.21.2.16 function wc_ecc_is_valid_idx

```
int wc_ecc_is_valid_idx(
    int n
)
```

Checks if an ECC idx is valid.

Parameters:

- **n** The idx number to check.

See: none

Return:

- 1 Return if valid.
- 0 Return if not valid.

Example

```
ecc_key key;
WC_RNG rng;
int is_valid;
wc_ecc_init(&key);
wc_InitRng(&rng);
wc_ecc_make_key(&rng, 32, &key);
is_valid = wc_ecc_is_valid_idx(key.idx);
if (is_valid == 1)
{
    // idx is valid
}
```

```
else if (is_valid == 0)
{
    // idx is not valid
}
```

C.21.2.17 function wc_ecc_new_point

```
ecc_point * wc_ecc_new_point(
    void
)
```

Allocate a new ECC point.

Parameters:

- **none** No parameters.

See:

- wc_ecc_del_point
- wc_ecc_cmp_point
- wc_ecc_copy_point

Return:

- p A newly allocated point.
- NULL Returns NULL on error.

Example

```
ecc_point* point;
point = wc_ecc_new_point();
if (point == NULL)
{
    // Handle point creation error
}
// Do stuff with point
```

C.21.2.18 function wc_ecc_del_point

```
void wc_ecc_del_point(
    ecc_point * p
)
```

Free an ECC point from memory.

Parameters:

- **p** The point to free.

See:

- wc_ecc_new_point
- wc_ecc_cmp_point
- wc_ecc_copy_point

Return: none No returns.

Example

```
ecc_point* point;
point = wc_ecc_new_point();
if (point == NULL)
```

```
{
    // Handle point creation error
}
// Do stuff with point
wc_ecc_del_point(point);
```

C.21.2.19 function wc_ecc_copy_point

```
int wc_ecc_copy_point(
    ecc_point * p,
    ecc_point * r
)
```

Copy the value of one point to another one.

Parameters:

- **p** The point to copy.
- **r** The created point.

See:

- wc_ecc_new_point
- wc_ecc_cmp_point
- wc_ecc_del_point

Return:

- ECC_BAD_ARG_E Error thrown when p or r is null.
- MP_OKAY Point copied successfully
- ret Error from internal functions. Can be...

Example

```
ecc_point* point;
ecc_point* copied_point;
int copy_return;

point = wc_ecc_new_point();
copy_return = wc_ecc_copy_point(point, copied_point);
if (copy_return != MP_OKAY)
{
    // Handle error
}
```

C.21.2.20 function wc_ecc_cmp_point

```
int wc_ecc_cmp_point(
    ecc_point * a,
    ecc_point * b
)
```

Compare the value of a point with another one.

Parameters:

- **a** First point to compare.
- **b** Second point to compare.

See:

- wc_ecc_new_point
- wc_ecc_del_point
- wc_ecc_copy_point

Return:

- BAD_FUNC_ARG One or both arguments are NULL.
- MP_EQ The points are equal.
- ret Either MP_LT or MP_GT and signifies that the points are not equal.

Example

```
ecc_point* point;
ecc_point* point_to_compare;
int cmp_result;

point = wc_ecc_new_point();
point_to_compare = wc_ecc_new_point();
cmp_result = wc_ecc_cmp_point(point, point_to_compare);
if (cmp_result == BAD_FUNC_ARG)
{
    // arguments are invalid
}
else if (cmp_result == MP_EQ)
{
    // Points are equal
}
else
{
    // Points are not equal
}
```

C.21.2.21 function wc_ecc_point_is_at_infinity

```
int wc_ecc_point_is_at_infinity(
    ecc_point * p
)
```

Checks if a point is at infinity. Returns 1 if point is at infinity, 0 if not, < 0 on error.

Parameters:

- **p** The point to check.

See:

- wc_ecc_new_point
- wc_ecc_del_point
- wc_ecc_cmp_point
- wc_ecc_copy_point

Return:

- 1 p is at infinity.
- 0 p is not at infinity.
- <0 Error.

Example

```
ecc_point* point;
int is_infinity;
```

```
point = wc_ecc_new_point();

is_infinity = wc_ecc_point_is_at_infinity(point);
if (is_infinity < 0)
{
    // Handle error
}
else if (is_infinity == 0)
{
    // Point is not at infinity
}
else if (is_infinity == 1)
{
    // Point is at infinity
}
```

C.21.2.22 function wc_ecc_mulmod

```
int wc_ecc_mulmod(
    mp_int * k,
    ecc_point * G,
    ecc_point * R,
    mp_int * a,
    mp_int * modulus,
    int map
)
```

Perform ECC Fixed Point multiplication.

Parameters:

- **k** The multiplicand.
- **G** Base point to multiply.
- **R** Destination of product.
- **modulus** The modulus for the curve.
- **map** If non-zero maps the point back to affine coordinates, otherwise it's left in jacobian-montgomery form.

See: none

Return:

- MP_OKAY Returns on successful operation.
- MP_INIT_E Returned if there is an error initializing an integer for use with the multiple precision integer (mp_int) library.

Example

```
ecc_point* base;
ecc_point* destination;
// Initialize points
base = wc_ecc_new_point();
destination = wc_ecc_new_point();
// Setup other arguments
mp_int multiplicand;
mp_int modulus;
int map;
```

C.21.2.23 function wc_ecc_export_x963

```
int wc_ecc_export_x963(
    ecc_key * key,
    byte * out,
    word32 * outLen
)
```

This function exports the ECC key from the ecc_key structure, storing the result in out. The key will be stored in ANSI X9.63 format. It stores the bytes written to the output buffer in outLen.

Parameters:

- **key** pointer to the ecc_key object to export
- **out** pointer to the buffer in which to store the ANSI X9.63 formatted key
- **outLen** size of the output buffer. On successfully storing the key, will hold the bytes written to the output buffer

See:

- wc_ecc_export_x963_ex
- wc_ecc_import_x963

Return:

- 0 Returned on successfully exporting the ecc_key
- LENGTH_ONLY_E Returned if the output buffer evaluates to NULL, but the other two input parameters are valid. Indicates that the function is only returning the length required to store the key
- ECC_BAD_ARG_E Returned if any of the input parameters are NULL, or the key is unsupported (has an invalid index)
- BUFFER_E Returned if the output buffer is too small to store the ecc key. If the output buffer is too small, the size needed will be returned in outLen
- MEMORY_E Returned if there is an error allocating memory with XMALLOC
- MP_INIT_E may be returned if there is an error processing the ecc_key
- MP_READ_E may be returned if there is an error processing the ecc_key
- MP_CMP_E may be returned if there is an error processing the ecc_key
- MP_INVMOD_E may be returned if there is an error processing the ecc_key
- MP_EXPTMOD_E may be returned if there is an error processing the ecc_key
- MP_MOD_E may be returned if there is an error processing the ecc_key
- MP_MUL_E may be returned if there is an error processing the ecc_key
- MP_ADD_E may be returned if there is an error processing the ecc_key
- MP_MULMOD_E may be returned if there is an error processing the ecc_key
- MP_TO_E may be returned if there is an error processing the ecc_key
- MP_MEM may be returned if there is an error processing the ecc_key

Example

```
int ret;
byte buff[1024];
word32 buffSz = sizeof(buff);

ecc_key key;
// initialize key, make key
ret = wc_ecc_export_x963(&key, buff, &buffSz);
if ( ret != 0) {
    // error exporting key
}
```

C.21.2.24 function wc_ecc_export_x963_ex

```
int wc_ecc_export_x963_ex(
    ecc_key * key,
    byte * out,
    word32 * outLen,
    int compressed
)
```

This function exports the ECC key from the ecc_key structure, storing the result in out. The key will be stored in ANSI X9.63 format. It stores the bytes written to the output buffer in outLen. This function allows the additional option of compressing the certificate through the compressed parameter. When this parameter is true, the key will be stored in ANSI X9.63 compressed format.

Parameters:

- **key** pointer to the ecc_key object to export
- **out** pointer to the buffer in which to store the ANSI X9.63 formatted key
- **outLen** size of the output buffer. On successfully storing the key, will hold the bytes written to the output buffer
- **compressed** indicator of whether to store the key in compressed format. 1==compressed, 0==uncompressed

See:

- wc_ecc_export_x963
- wc_ecc_import_x963

Return:

- 0 Returned on successfully exporting the ecc_key
- NOT_COMPILED_IN Returned if the HAVE_COMP_KEY was not enabled at compile time, but the key was requested in compressed format
- LENGTH_ONLY_E Returned if the output buffer evaluates to NULL, but the other two input parameters are valid. Indicates that the function is only returning the length required to store the key
- ECC_BAD_ARG_E Returned if any of the input parameters are NULL, or the key is unsupported (has an invalid index)
- BUFFER_E Returned if the output buffer is too small to store the ecc key. If the output buffer is too small, the size needed will be returned in outLen
- MEMORY_E Returned if there is an error allocating memory with XMALLOC
- MP_INIT_E may be returned if there is an error processing the ecc_key
- MP_READ_E may be returned if there is an error processing the ecc_key
- MP_CMP_E may be returned if there is an error processing the ecc_key
- MP_INVMOD_E may be returned if there is an error processing the ecc_key
- MP_EXPTMOD_E may be returned if there is an error processing the ecc_key
- MP_MOD_E may be returned if there is an error processing the ecc_key
- MP_MUL_E may be returned if there is an error processing the ecc_key
- MP_ADD_E may be returned if there is an error processing the ecc_key
- MP_MULMOD_E may be returned if there is an error processing the ecc_key
- MP_TO_E may be returned if there is an error processing the ecc_key
- MP_MEM may be returned if there is an error processing the ecc_key

Example

```
int ret;
byte buff[1024];
word32 buffSz = sizeof(buff);
ecc_key key;
```

```
// initialize key, make key
ret = wc_ecc_export_x963_ex(&key, buff, &buffSz, 1);
if ( ret != 0) {
    // error exporting key
}
```

C.21.2.25 function wc_ecc_import_x963

```
int wc_ecc_import_x963(
    const byte * in,
    word32 inLen,
    ecc_key * key
)
```

This function imports a public ECC key from a buffer containing the key stored in ANSI X9.63 format. This function will handle both compressed and uncompressed keys, as long as compressed keys are enabled at compile time through the HAVE_COMP_KEY option.

Parameters:

- **in** pointer to the buffer containing the ANSI x9.63 formatted ECC key
- **inLen** length of the input buffer
- **key** pointer to the ecc_key object in which to store the imported key

See:

- wc_ecc_export_x963
- wc_ecc_import_private_key

Return:

- 0 Returned on successfully importing the ecc_key
- NOT_COMPILED_IN Returned if the HAVE_COMP_KEY was not enabled at compile time, but the key is stored in compressed format
- ECC_BAD_ARG_E Returned if in or key evaluate to NULL, or the inLen is even (according to the x9.63 standard, the key must be odd)
- MEMORY_E Returned if there is an error allocating memory
- ASN_PARSE_E Returned if there is an error parsing the ECC key; may indicate that the ECC key is not stored in valid ANSI X9.63 format
- IS_POINT_E Returned if the public key exported is not a point on the ECC curve
- MP_INIT_E may be returned if there is an error processing the ecc_key
- MP_READ_E may be returned if there is an error processing the ecc_key
- MP_CMP_E may be returned if there is an error processing the ecc_key
- MP_INVMOD_E may be returned if there is an error processing the ecc_key
- MP_EXPTMOD_E may be returned if there is an error processing the ecc_key
- MP_MOD_E may be returned if there is an error processing the ecc_key
- MP_MUL_E may be returned if there is an error processing the ecc_key
- MP_ADD_E may be returned if there is an error processing the ecc_key
- MP_MULMOD_E may be returned if there is an error processing the ecc_key
- MP_TO_E may be returned if there is an error processing the ecc_key
- MP_MEM may be returned if there is an error processing the ecc_key

Example

```
int ret;
byte buff[] = { initialize with ANSI X9.63 formatted key };

ecc_key pubKey;
wc_ecc_init(&pubKey);
```

```
ret = wc_ecc_import_x963(buff, sizeof(buff), &pubKey);
if ( ret != 0) {
    // error importing key
}
```

C.21.2.26 function wc_ecc_import_private_key

```
int wc_ecc_import_private_key(
    const byte * priv,
    word32 privSz,
    const byte * pub,
    word32 pubSz,
    ecc_key * key
)
```

This function imports a public/private ECC key pair from a buffer containing the raw private key, and a second buffer containing the ANSI X9.63 formatted public key. This function will handle both compressed and uncompressed keys, as long as compressed keys are enabled at compile time through the HAVE_COMP_KEY option.

Parameters:

- **priv** pointer to the buffer containing the raw private key
- **privSz** size of the private key buffer
- **pub** pointer to the buffer containing the ANSI x9.63 formatted ECC public key
- **pubSz** length of the public key input buffer
- **key** pointer to the ecc_key object in which to store the imported private/public key pair

See:

- wc_ecc_export_x963
- wc_ecc_import_private_key

Return:

- 0 Returned on successfully importing the ecc_key NOT_COMPILED_IN Returned if the HAVE_COMP_KEY was not enabled at compile time, but the key is stored in compressed format
- ECC_BAD_ARG_E Returned if in or key evaluate to NULL, or the inLen is even (according to the x9.63 standard, the key must be odd)
- MEMORY_E Returned if there is an error allocating memory
- ASN_PARSE_E Returned if there is an error parsing the ECC key; may indicate that the ECC key is not stored in valid ANSI X9.63 format
- IS_POINT_E Returned if the public key exported is not a point on the ECC curve
- MP_INIT_E may be returned if there is an error processing the ecc_key
- MP_READ_E may be returned if there is an error processing the ecc_key
- MP_CMP_E may be returned if there is an error processing the ecc_key
- MP_INVMOD_E may be returned if there is an error processing the ecc_key
- MP_EXPTMOD_E may be returned if there is an error processing the ecc_key
- MP_MOD_E may be returned if there is an error processing the ecc_key
- MP_MUL_E may be returned if there is an error processing the ecc_key
- MP_ADD_E may be returned if there is an error processing the ecc_key
- MP_MULMOD_E may be returned if there is an error processing the ecc_key
- MP_TO_E may be returned if there is an error processing the ecc_key
- MP_MEM may be returned if there is an error processing the ecc_key

Example

```
int ret;
byte pub[] = { initialize with ANSI X9.63 formatted key };
byte priv[] = { initialize with the raw private key };

ecc_key key;
wc_ecc_init(&key);
ret = wc_ecc_import_private_key(priv, sizeof(priv), pub, sizeof(pub),
&key);
if ( ret != 0) {
    // error importing key
}
```

C.21.2.27 function wc_ecc_rs_to_sig

```
int wc_ecc_rs_to_sig(
    const char * r,
    const char * s,
    byte * out,
    word32 * outlen
)
```

This function converts the R and S portions of an ECC signature into a DER-encoded ECDSA signature. This function also stores the length written to the output buffer, out, in outlen.

Parameters:

- **r** pointer to the buffer containing the R portion of the signature as a string
- **s** pointer to the buffer containing the S portion of the signature as a string
- **out** pointer to the buffer in which to store the DER-encoded ECDSA signature
- **outlen** length of the output buffer available. Will store the bytes written to the buffer after successfully converting the signature to ECDSA format

See:

- wc_ecc_sign_hash
- wc_ecc_sig_size

Return:

- 0 Returned on successfully converting the signature
- ECC_BAD_ARG_E Returned if any of the input parameters evaluate to NULL, or if the input buffer is not large enough to hold the DER-encoded ECDSA signature
- MP_INIT_E may be returned if there is an error processing the ecc_key
- MP_READ_E may be returned if there is an error processing the ecc_key
- MP_CMP_E may be returned if there is an error processing the ecc_key
- MP_INVMOD_E may be returned if there is an error processing the ecc_key
- MP_EXPTMOD_E may be returned if there is an error processing the ecc_key
- MP_MOD_E may be returned if there is an error processing the ecc_key
- MP_MUL_E may be returned if there is an error processing the ecc_key
- MP_ADD_E may be returned if there is an error processing the ecc_key
- MP_MULMOD_E may be returned if there is an error processing the ecc_key
- MP_TO_E may be returned if there is an error processing the ecc_key
- MP_MEM may be returned if there is an error processing the ecc_key

Example

```
int ret;
ecc_key key;
// initialize key, generate R and S
```

```
char r[] = { initialize with R };
char s[] = { initialize with S };
byte sig[wc_ecc_sig_size(key)];
// signature size will be 2 * ECC key size + ~10 bytes for ASN.1 overhead
word32 sigSz = sizeof(sig);
ret = wc_ecc_rs_to_sig(r, s, sig, &sigSz);
if ( ret != 0) {
    // error converting parameters to signature
}
```

C.21.2.28 function wc_ecc_import_raw

```
int wc_ecc_import_raw(
    ecc_key * key,
    const char * qx,
    const char * qy,
    const char * d,
    const char * curveName
)
```

This function fills an ecc_key structure with the raw components of an ECC signature.

Parameters:

- **key** pointer to an ecc_key structure to fill
- **qx** pointer to a buffer containing the x component of the base point as an ASCII hex string
- **qy** pointer to a buffer containing the y component of the base point as an ASCII hex string
- **d** pointer to a buffer containing the private key as an ASCII hex string
- **curveName** pointer to a string containing the ECC curve name, as found in ecc_sets

See: wc_ecc_import_private_key

Return:

- 0 Returned upon successfully importing into the ecc_key structure
- ECC_BAD_ARG_E Returned if any of the input values evaluate to NULL
- MEMORY_E Returned if there is an error initializing space to store the parameters of the ecc_key
- ASN_PARSE_E Returned if the input curveName is not defined in ecc_sets
- MP_INIT_E may be returned if there is an error processing the input parameters
- MP_READ_E may be returned if there is an error processing the input parameters
- MP_CMP_E may be returned if there is an error processing the input parameters
- MP_INVMOD_E may be returned if there is an error processing the input parameters
- MP_EXPTMOD_E may be returned if there is an error processing the input parameters
- MP_MOD_E may be returned if there is an error processing the input parameters
- MP_MUL_E may be returned if there is an error processing the input parameters
- MP_ADD_E may be returned if there is an error processing the input parameters
- MP_MULMOD_E may be returned if there is an error processing the input parameters
- MP_TO_E may be returned if there is an error processing the input parameters
- MP_MEM may be returned if there is an error processing the input parameters

Example

```
int ret;
ecc_key key;
wc_ecc_init(&key);

char qx[] = { initialize with x component of base point };
```

```
char qy[] = { initialize with y component of base point };
char d[]  = { initialize with private key };
ret = wc_ecc_import_raw(&key,qx, qy, d, "ECC-256");
if ( ret != 0) {
    // error initializing key with given inputs
}
```

C.21.2.29 function wc_ecc_export_private_only

```
int wc_ecc_export_private_only(
    ecc_key * key,
    byte * out,
    word32 * outLen
)
```

This function exports only the private key from an ecc_key structure. It stores the private key in the buffer out, and sets the bytes written to this buffer in outLen.

Parameters:

- **key** pointer to an ecc_key structure from which to export the private key
- **out** pointer to the buffer in which to store the private key
- **outLen** pointer to a word32 object with the size available in out. Set with the number of bytes written to out after successfully exporting the private key

See: wc_ecc_import_private_key

Return:

- 0 Returned upon successfully exporting the private key
- ECC_BAD_ARG_E Returned if any of the input values evaluate to NULL
- MEMORY_E Returned if there is an error initializing space to store the parameters of the ecc_key
- ASN_PARSE_E Returned if the input curveName is not defined in ecc_sets
- MP_INIT_E may be returned if there is an error processing the input parameters
- MP_READ_E may be returned if there is an error processing the input parameters
- MP_CMP_E may be returned if there is an error processing the input parameters
- MP_INVMOD_E may be returned if there is an error processing the input parameters
- MP_EXPTMOD_E may be returned if there is an error processing the input parameters
- MP_MOD_E may be returned if there is an error processing the input parameters
- MP_MUL_E may be returned if there is an error processing the input parameters
- MP_ADD_E may be returned if there is an error processing the input parameters
- MP_MULMOD_E may be returned if there is an error processing the input parameters
- MP_TO_E may be returned if there is an error processing the input parameters
- MP_MEM may be returned if there is an error processing the input parameters

Example

```
int ret;
ecc_key key;
// initialize key, make key

char priv[ECC_KEY_SIZE];
word32 privSz = sizeof(priv);
ret = wc_ecc_export_private_only(&key, priv, &privSz);
if ( ret != 0) {
    // error exporting private key
}
```

C.21.2.30 function wc_ecc_export_point_der

```
int wc_ecc_export_point_der(
    const int curve_idx,
    ecc_point * point,
    byte * out,
    word32 * outLen
)
```

Export point to der.

Parameters:

- **curve_idx** Index of the curve used from ecc_sets.
- **point** Point to export to der.
- **out** Destination for the output.
- **outLen** Maxsize allowed for output, destination for final size of output

See: wc_ecc_import_point_der

Return:

- 0 Returned on success.
- ECC_BAD_ARG_E Returns if curve_idx is less than 0 or invalid. Also returns when
- LENGTH_ONLY_E outLen is set but nothing else.
- BUFFER_E Returns if outLen is less than 1 + 2 * the curve size.
- MEMORY_E Returns if there is a problem allocating memory.

Example

```
int curve_idx;
ecc_point* point;
byte out[];
word32 outLen;
wc_ecc_export_point_der(curve_idx, point, out, &outLen);
```

C.21.2.31 function wc_ecc_import_point_der

```
int wc_ecc_import_point_der(
    byte * in,
    word32 inLen,
    const int curve_idx,
    ecc_point * point
)
```

Import point from der format.

Parameters:

- **in** der buffer to import point from.
- **inLen** Length of der buffer.
- **curve_idx** Index of curve.
- **point** Destination for point.

See: wc_ecc_export_point_der

Return:

- ECC_BAD_ARG_E Returns if any arguments are null or if inLen is even.
- MEMORY_E Returns if there is an error initializing
- NOT_COMPILED_IN Returned if HAVE_COMP_KEY is not true and in is a compressed cert
- MP_OKAY Successful operation.

Example

```
byte in[];
word32 inLen;
int curve_idx;
ecc_point* point;
wc_ecc_import_point_der(in, inLen, curve_idx, point);
```

C.21.2.32 function wc_ecc_size

```
int wc_ecc_size(
    ecc_key * key
)
```

This function returns the key size of an ecc_key structure in octets.

Parameters:

- **key** pointer to an ecc_key structure for which to get the key size

See: wc_ecc_make_key

Return:

- Given a valid key, returns the key size in octets
- 0 Returned if the given key is NULL

Example

```
int keySz;
ecc_key key;
// initialize key, make key
keySz = wc_ecc_size(&key);
if ( keySz == 0) {
    // error determining key size
}
```

C.21.2.33 function wc_ecc_sig_size_calc

```
int wc_ecc_sig_size_calc(
    int sz
)
```

This function returns the worst case size for an ECC signature, given by: (keySz * 2) + SIG_HEADER_SZ + ECC_MAX_PAD_SZ. The actual signature size can be computed with wc_ecc_sign_hash.

Parameters:

- **key** size

See:

- wc_ecc_sign_hash
- wc_ecc_sig_size

Return: returns the maximum signature size, in octets

Example

```
int sigSz = wc_ecc_sig_size_calc(32);
if ( sigSz == 0) {
    // error determining sig size
}
```

C.21.2.34 function wc_ecc_sig_size

```
int wc_ecc_sig_size(
    ecc_key * key
)
```

This function returns the worst case size for an ECC signature, given by: (keySz * 2) + SIG_HEADER_SZ + ECC_MAX_PAD_SZ. The actual signature size can be computed with wc_ecc_sign_hash.

Parameters:

- **key** pointer to an ecc_key structure for which to get the signature size

See:

- wc_ecc_sign_hash
- wc_ecc_sig_size_calc

Return:

- Success Given a valid key, returns the maximum signature size, in octets
- 0 Returned if the given key is NULL

Example

```
int sigSz;
ecc_key key;
// initialize key, make key

sigSz = wc_ecc_sig_size(&key);
if ( sigSz == 0) {
    // error determining sig size
}
```

C.21.2.35 function wc_ecc_ctx_new

```
ecEncCtx * wc_ecc_ctx_new(
    int flags,
    WC_RNG * rng
)
```

This function allocates and initializes space for a new ECC context object to allow secure message exchange with ECC.

Parameters:

- **flags** indicate whether this is a server or client context Options are: REQ_RESP_CLIENT, and REQ_RESP_SERVER
- **rng** pointer to a RNG object with which to generate a salt

See:

- wc_ecc_encrypt
- wc_ecc_encrypt_ex
- wc_ecc_decrypt

Return:

- Success On successfully generating a new ecEncCtx object, returns a pointer to that object
- NULL Returned if the function fails to generate a new ecEncCtx object

Example

```
ecEncCtx* ctx;
WC_RNG rng;
wc_InitRng(&rng);
ctx = wc_ecc_ctx_new(REQ_RESP_CLIENT, &rng);
if(ctx == NULL) {
    // error generating new ecEncCtx object
}
```

C.21.2.36 function wc_ecc_ctx_free

```
void wc_ecc_ctx_free(
    ecEncCtx *
)
```

This function frees the ecEncCtx object used for encrypting and decrypting messages.

Parameters:

- **ctx** pointer to the ecEncCtx object to free

See: wc_ecc_ctx_new

Return: none Returns.

Example

```
ecEncCtx* ctx;
WC_RNG rng;
wc_InitRng(&rng);
ctx = wc_ecc_ctx_new(REQ_RESP_CLIENT, &rng);
// do secure communication
...
wc_ecc_ctx_free(&ctx);
```

C.21.2.37 function wc_ecc_ctx_reset

```
int wc_ecc_ctx_reset(
    ecEncCtx * ctx,
    WC_RNG * rng
)
```

This function resets an ecEncCtx structure to avoid having to free and allocate a new context object.

Parameters:

- **ctx** pointer to the ecEncCtx object to reset
- **rng** pointer to an RNG object with which to generate a new salt

See: wc_ecc_ctx_new

Return:

- 0 Returned if the ecEncCtx structure is successfully reset
- BAD_FUNC_ARG Returned if either rng or ctx is NULL
- RNG_FAILURE_E Returned if there is an error generating a new salt for the ECC object

Example

```
ecEncCtx* ctx;
WC_RNG rng;
wc_InitRng(&rng);
ctx = wc_ecc_ctx_new(REQ_RESP_CLIENT, &rng);
```

```
// do secure communication
...
wc_ecc_ctx_reset(&ctx, &rng);
// do more secure communication
```

C.21.2.38 function wc_ecc_ctx_set_algo

```
int wc_ecc_ctx_set_algo(
    ecEncCtx * ctx,
    byte encAlgo,
    byte kdfAlgo,
    byte macAlgo
)
```

This function can optionally be called after wc_ecc_ctx_new. It sets the encryption, KDF, and MAC algorithms into an ecEncCtx object.

Parameters:

- **ctx** pointer to the ecEncCtx for which to set the info
- **encAlgo** encryption algorithm to use.
- **kdfAlgo** KDF algorithm to use.
- **macAlgo** MAC algorithm to use.

See: wc_ecc_ctx_new

Return:

- 0 Returned upon successfully setting the information for the ecEncCtx object.
- BAD_FUNC_ARG Returned if the given ecEncCtx object is NULL.

Example

```
ecEncCtx* ctx;
// initialize ctx
if(wc_ecc_ctx_set_algo(&ctx, ecAES_128_CTR, ecHKDF_SHA256, ecHMAC_SHA256))) {
    // error setting info
}
```

C.21.2.39 function wc_ecc_ctx_get_own_salt

```
const byte * wc_ecc_ctx_get_own_salt(
    ecEncCtx *
)
```

This function returns the salt of an ecEncCtx object. This function should only be called when the ecEncCtx's state is ecSRV_INIT or ecCLI_INIT.

Parameters:

- **ctx** pointer to the ecEncCtx object from which to get the salt

See:

- wc_ecc_ctx_new
- wc_ecc_ctx_set_peer_salt
- wc_ecc_ctx_set_kdf_salt

Return:

- Success On success, returns the ecEncCtx salt

- NULL Returned if the ecEncCtx object is NULL, or the ecEncCtx's state is not ecSRV_INIT or ecCLI_INIT. In the latter two cases, this function also sets the ecEncCtx's state to ecSRV_BAD_STATE or ecCLI_BAD_STATE, respectively

Example

```
ecEncCtx* ctx;
WC_RNG rng;
const byte* salt;
wc_InitRng(&rng);
ctx = wc_ecc_ctx_new(REQ_RESP_CLIENT, &rng);
salt = wc_ecc_ctx_get_own_salt(&ctx);
if(salt == NULL) {
    // error getting salt
}
```

C.21.2.40 function wc_ecc_ctx_set_peer_salt

```
int wc_ecc_ctx_set_peer_salt(
    ecEncCtx * ctx,
    const byte * salt
)
```

This function sets the peer salt of an ecEncCtx object.

Parameters:

- **ctx** pointer to the ecEncCtx for which to set the salt
- **salt** pointer to the peer's salt

See:

- wc_ecc_ctx_get_own_salt
- wc_ecc_ctx_set_kdf_salt

Return:

- 0 Returned upon successfully setting the peer salt for the ecEncCtx object.
- BAD_FUNC_ARG Returned if the given ecEncCtx object is NULL or has an invalid protocol, or if the given salt is NULL
- BAD_ENC_STATE_E Returned if the ecEncCtx's state is ecSRV_SALT_GET or ecCLI_SALT_GET. In the latter two cases, this function also sets the ecEncCtx's state to ecSRV_BAD_STATE or ecCLI_BAD_STATE, respectively

Example

```
ecEncCtx* cliCtx, srvCtx;
WC_RNG rng;
const byte* cliSalt, srvSalt;
int ret;

wc_InitRng(&rng);
cliCtx = wc_ecc_ctx_new(REQ_RESP_CLIENT, &rng);
srvCtx = wc_ecc_ctx_new(REQ_RESP_SERVER, &rng);

cliSalt = wc_ecc_ctx_get_own_salt(&cliCtx);
srvSalt = wc_ecc_ctx_get_own_salt(&srvCtx);
ret = wc_ecc_ctx_set_peer_salt(&cliCtx, srvSalt);
```

C.21.2.41 function wc_ecc_ctx_set_kdf_salt

```
int wc_ecc_ctx_set_kdf_salt(
    ecEncCtx * ctx,
    const byte * salt,
    word32 sz
)
```

This function sets the salt pointer and length to use with KDF into the ecEncCtx object.

Parameters:

- **ctx** pointer to the ecEncCtx for which to set the salt
- **salt** pointer to salt buffer
- **sz** length salt in bytes

See:

- wc_ecc_ctx_get_own_salt
- wc_ecc_ctx_get_peer_salt

Return:

- 0 Returned upon successfully setting the salt for the ecEncCtx object.
- BAD_FUNC_ARG Returned if the given ecEncCtx object is NULL or if the given salt is NULL and length is not NULL.

Example

```
ecEncCtx* srvCtx;
WC_RNG rng;
byte cliSalt[] = { fixed salt data };
word32 cliSaltLen = (word32)sizeof(cliSalt);
int ret;

wc_InitRng(&rng);
cliCtx = wc_ecc_ctx_new(REQ_RESP_SERVER, &rng);

ret = wc_ecc_ctx_set_kdf_salt(&cliCtx, cliSalt, cliSaltLen);
```

C.21.2.42 function wc_ecc_ctx_set_info

```
int wc_ecc_ctx_set_info(
    ecEncCtx * ctx,
    const byte * info,
    int sz
)
```

This function can optionally be called before or after wc_ecc_ctx_set_peer_salt. It sets optional information for an ecEncCtx object.

Parameters:

- **ctx** pointer to the ecEncCtx for which to set the info
- **info** pointer to a buffer containing the info to set
- **sz** size of the info buffer

See: wc_ecc_ctx_new

Return:

- 0 Returned upon successfully setting the information for the ecEncCtx object.

- BAD_FUNC_ARG Returned if the given ecEncCtx object is NULL, the input info is NULL or it's size is invalid

Example

```
ecEncCtx* ctx;
byte info[] = { initialize with information };
// initialize ctx, get salt,
if(wc_ecc_ctx_set_info(&ctx, info, sizeof(info))) {
    // error setting info
}
```

C.21.2.43 function wc_ecc_encrypt

```
int wc_ecc_encrypt(
    ecc_key * privKey,
    ecc_key * pubKey,
    const byte * msg,
    word32 msgSz,
    byte * out,
    word32 * outSz,
    ecEncCtx * ctx
)
```

This function encrypts the given input message from msg to out. This function takes an optional ctx object as parameter. When supplied, encryption proceeds based on the ecEncCtx's encAlgo, kdfAlgo, and macAlgo. If ctx is not supplied, processing completes with the default algorithms, ecAES_128_CBC, ecHKDF_SHA256 and ecHMAC_SHA256. This function requires that the messages are padded according to the encryption type specified by ctx.

Parameters:

- **privKey** pointer to the ecc_key object containing the private key to use for encryption
- **pubKey** pointer to the ecc_key object containing the public key of the peer with whom one wishes to communicate
- **msg** pointer to the buffer holding the message to encrypt
- **msgSz** size of the buffer to encrypt
- **out** pointer to the buffer in which to store the encrypted ciphertext
- **outSz** pointer to a word32 object containing the available size in the out buffer. Upon successfully encrypting the message, holds the number of bytes written to the output buffer
- **ctx** Optional: pointer to an ecEncCtx object specifying different encryption algorithms to use

See:

- wc_ecc_encrypt_ex
- wc_ecc_decrypt

Return:

- 0 Returned upon successfully encrypting the input message
- BAD_FUNC_ARG Returned if privKey, pubKey, msg, msgSz, out, or outSz are NULL, or the ctx object specifies an unsupported encryption type
- BAD_ENC_STATE_E Returned if the ctx object given is in a state that is not appropriate for encryption
- BUFFER_E Returned if the supplied output buffer is too small to store the encrypted ciphertext
- MEMORY_E Returned if there is an error allocating memory for the shared secret key

Example

```
byte msg[] = { initialize with msg to encrypt. Ensure padded to block size };
byte out[sizeof(msg)];
word32 outSz = sizeof(out);
int ret;
ecc_key cli, serv;
// initialize cli with private key
// initialize serv with received public key

ecEncCtx* cliCtx, servCtx;
// initialize cliCtx and servCtx
// exchange salts
ret = wc_ecc_encrypt(&cli, &serv, msg, sizeof(msg), out, &outSz, cliCtx);
if(ret != 0) {
    // error encrypting message
}
```

C.21.2.44 function wc_ecc_encrypt_ex

```
int wc_ecc_encrypt_ex(
    ecc_key * privKey,
    ecc_key * pubKey,
    const byte * msg,
    word32 msgSz,
    byte * out,
    word32 * outSz,
    ecEncCtx * ctx,
    int compressed
)
```

This function encrypts the given input message from msg to out. This function takes an optional ctx object as parameter. When supplied, encryption proceeds based on the ecEncCtx's encAlgo, kdfAlgo, and macAlgo. If ctx is not supplied, processing completes with the default algorithms, ecAES_128_CBC, ecHKDF_SHA256 and ecHMAC_SHA256. This function requires that the messages are padded according to the encryption type specified by ctx.

Parameters:

- **privKey** pointer to the ecc_key object containing the private key to use for encryption
- **pubKey** pointer to the ecc_key object containing the public key of the peer with whom one wishes to communicate
- **msg** pointer to the buffer holding the message to encrypt
- **msgSz** size of the buffer to encrypt
- **out** pointer to the buffer in which to store the encrypted ciphertext
- **outSz** pointer to a word32 object containing the available size in the out buffer. Upon successfully encrypting the message, holds the number of bytes written to the output buffer
- **ctx** Optional: pointer to an ecEncCtx object specifying different encryption algorithms to use
- **compressed** Public key field is to be output in compressed format.

See:

- wc_ecc_encrypt
- wc_ecc_decrypt

Return:

- 0 Returned upon successfully encrypting the input message
- BAD_FUNC_ARG Returned if privKey, pubKey, msg, msgSz, out, or outSz are NULL, or the ctx object specifies an unsupported encryption type

- BAD_ENC_STATE_E Returned if the ctx object given is in a state that is not appropriate for encryption
- BUFFER_E Returned if the supplied output buffer is too small to store the encrypted ciphertext
- MEMORY_E Returned if there is an error allocating memory for the shared secret key

Example

```
byte msg[] = { initialize with msg to encrypt. Ensure padded to block size };
byte out[sizeof(msg)];
word32 outSz = sizeof(out);
int ret;
ecc_key cli, serv;
// initialize cli with private key
// initialize serv with received public key

ecEncCtx* cliCtx, servCtx;
// initialize cliCtx and servCtx
// exchange salts
ret = wc_ecc_encrypt_ex(&cli, &serv, msg, sizeof(msg), out, &outSz, cliCtx,
    1);
if(ret != 0) {
    // error encrypting message
}
```

C.21.2.45 function wc_ecc_decrypt

```
int wc_ecc_decrypt(
    ecc_key * privKey,
    ecc_key * pubKey,
    const byte * msg,
    word32 msgSz,
    byte * out,
    word32 * outSz,
    ecEncCtx * ctx
)
```

This function decrypts the ciphertext from msg to out. This function takes an optional ctx object as parameter. When supplied, encryption proceeds based on the ecEncCtx's encAlgo, kdfAlgo, and macAlgo. If ctx is not supplied, processing completes with the default algorithms, ecAES_128_CBC, ecHKDF_SHA256 and ecHMAC_SHA256. This function requires that the messages are padded according to the encryption type specified by ctx.

Parameters:

- **privKey** pointer to the ecc_key object containing the private key to use for decryption
- **pubKey** pointer to the ecc_key object containing the public key of the peer with whom one wishes to communicate
- **msg** pointer to the buffer holding the ciphertext to decrypt
- **msgSz** size of the buffer to decrypt
- **out** pointer to the buffer in which to store the decrypted plaintext
- **outSz** pointer to a word32 object containing the available size in the out buffer. Upon successfully decrypting the ciphertext, holds the number of bytes written to the output buffer
- **ctx** Optional: pointer to an ecEncCtx object specifying different decryption algorithms to use

See:

- wc_ecc_encrypt
- wc_ecc_encrypt_ex

Return:

- 0 Returned upon successfully decrypting the input message
- BAD_FUNC_ARG Returned if privKey, pubKey, msg, msgSz, out, or outSz are NULL, or the ctx object specifies an unsupported encryption type
- BAD_ENC_STATE_E Returned if the ctx object given is in a state that is not appropriate for decryption
- BUFFER_E Returned if the supplied output buffer is too small to store the decrypted plaintext
- MEMORY_E Returned if there is an error allocating memory for the shared secret key

Example

```
byte cipher[] = { initialize with
ciphertext to decrypt. Ensure padded to block size };
byte plain[sizeof(cipher)];
word32 plainSz = sizeof(plain);
int ret;
ecc_key cli, serv;
// initialize cli with private key
// initialize serv with received public key
ecEncCtx* cliCtx, servCtx;
// initialize cliCtx and servCtx
// exchange salts
ret = wc_ecc_decrypt(&cli, &serv, cipher, sizeof(cipher),
plain, &plainSz, cliCtx);

if(ret != 0) {
    // error decrypting message
}
```

C.21.2.46 function wc_ecc_set_nonblock

```
int wc_ecc_set_nonblock(
    ecc_key * key,
    ecc_nb_ctx_t * ctx
)
```

Enable ECC support for non-blocking operations. Supported for Single Precision (SP) math with the following build options: WOLFSSL_SP_NONBLOCK WOLFSSL_SP_SMALL WOLFSSL_SP_NO_MALLOC WC_ECC_NONBLOCK.

Parameters:

- **key** pointer to the ecc_key object
- **ctx** pointer to ecc_nb_ctx_t structure with stack data cache for SP

Return: 0 Returned upon successfully setting the callback context the input message

Example

```
int ret;
ecc_key ecc;
ecc_nb_ctx_t nb_ctx;

ret = wc_ecc_init(&ecc);
if (ret == 0) {
    ret = wc_ecc_set_nonblock(&ecc, &nb_ctx);
    if (ret == 0) {
        do {
```

```
            ret = wc_ecc_verify_hash_ex(
                &r, &s,       // r/s as mp_int
                hash, hashSz, // computed hash digest
                &verify_res,  // verification result 1=success
                &key
            );

            // TODO: Real-time work can be called here
        } while (ret == FP_WOULDBLOCK);
    }
    wc_ecc_free(&key);
}
```

C.21.2.47 function wc_ecc_set_curve

```
int wc_ecc_set_curve(
    ecc_key * key,
    int keysize,
    int curve_id
)
```

Compare a curve which has larger key than specified size or the curve matched curve ID, set a curve with smaller key size to the key.

Parameters:

- **keysize** Key size in bytes
- **curve_id** Curve ID

```
 int ret;
ecc_key ecc;

ret = wc_ecc_init(&ecc);
if (ret != 0)
    return ret;
    ret = wc_ecc_set_curve(&ecc, 32, ECC_SECP256R1));
    if (ret != 0)
        return ret;
```

Return: 0 Returned upon successfully setting the key

C.21.3 Source code

```
int wc_ecc_make_key(WC_RNG* rng, int keysize, ecc_key* key);

int wc_ecc_make_key_ex(WC_RNG* rng, int keysize, ecc_key* key, int curve_id);

int wc_ecc_check_key(ecc_key* key);

void wc_ecc_key_free(ecc_key* key);

int wc_ecc_shared_secret(ecc_key* private_key, ecc_key* public_key, byte* out,
                         word32* outlen);
```

```
int wc_ecc_shared_secret_ex(ecc_key* private_key, ecc_point* point,
                            byte* out, word32 *outlen);

int wc_ecc_sign_hash(const byte* in, word32 inlen, byte* out, word32 *outlen,
                     WC_RNG* rng, ecc_key* key);

int wc_ecc_sign_hash_ex(const byte* in, word32 inlen, WC_RNG* rng,
                        ecc_key* key, mp_int *r, mp_int *s);

int wc_ecc_verify_hash(const byte* sig, word32 siglen, const byte* hash,
                   word32 hashlen, int* stat, ecc_key* key);

int wc_ecc_verify_hash_ex(mp_int *r, mp_int *s, const byte* hash,
                          word32 hashlen, int* stat, ecc_key* key);

int wc_ecc_init(ecc_key* key);

int wc_ecc_init_ex(ecc_key* key, void* heap, int devId);

ecc_key* wc_ecc_key_new(void* heap);

int wc_ecc_free(ecc_key* key);

void wc_ecc_fp_free(void);

int wc_ecc_is_valid_idx(int n);

ecc_point* wc_ecc_new_point(void);

void wc_ecc_del_point(ecc_point* p);

int wc_ecc_copy_point(ecc_point* p, ecc_point *r);

int wc_ecc_cmp_point(ecc_point* a, ecc_point *b);

int wc_ecc_point_is_at_infinity(ecc_point *p);

int wc_ecc_mulmod(mp_int* k, ecc_point *G, ecc_point *R,
                  mp_int* a, mp_int* modulus, int map);

int wc_ecc_export_x963(ecc_key* key, byte* out, word32* outLen);

int wc_ecc_export_x963_ex(ecc_key* key, byte* out, word32* outLen, int
 ↪  compressed);

int wc_ecc_import_x963(const byte* in, word32 inLen, ecc_key* key);

int wc_ecc_import_private_key(const byte* priv, word32 privSz, const byte* pub,
                          word32 pubSz, ecc_key* key);

int wc_ecc_rs_to_sig(const char* r, const char* s, byte* out, word32* outlen);

int wc_ecc_import_raw(ecc_key* key, const char* qx, const char* qy,
```

```
                    const char* d, const char* curveName);

int wc_ecc_export_private_only(ecc_key* key, byte* out, word32* outLen);

int wc_ecc_export_point_der(const int curve_idx, ecc_point* point,
                            byte* out, word32* outLen);

int wc_ecc_import_point_der(byte* in, word32 inLen, const int curve_idx,
                            ecc_point* point);

int wc_ecc_size(ecc_key* key);

int wc_ecc_sig_size_calc(int sz);


int wc_ecc_sig_size(ecc_key* key);


ecEncCtx* wc_ecc_ctx_new(int flags, WC_RNG* rng);

void wc_ecc_ctx_free(ecEncCtx*);

int wc_ecc_ctx_reset(ecEncCtx* ctx, WC_RNG* rng);  /* reset for use again w/o
 ↪  alloc/free */

int wc_ecc_ctx_set_algo(ecEncCtx* ctx, byte encAlgo, byte kdfAlgo,
    byte macAlgo);

const byte* wc_ecc_ctx_get_own_salt(ecEncCtx*);

int wc_ecc_ctx_set_peer_salt(ecEncCtx* ctx, const byte* salt);

int wc_ecc_ctx_set_kdf_salt(ecEncCtx* ctx, const byte* salt, word32 sz);

int wc_ecc_ctx_set_info(ecEncCtx* ctx, const byte* info, int sz);

int wc_ecc_encrypt(ecc_key* privKey, ecc_key* pubKey, const byte* msg,
                word32 msgSz, byte* out, word32* outSz, ecEncCtx* ctx);

int wc_ecc_encrypt_ex(ecc_key* privKey, ecc_key* pubKey, const byte* msg,
    word32 msgSz, byte* out, word32* outSz, ecEncCtx* ctx, int compressed);

int wc_ecc_decrypt(ecc_key* privKey, ecc_key* pubKey, const byte* msg,
                word32 msgSz, byte* out, word32* outSz, ecEncCtx* ctx);


int wc_ecc_set_nonblock(ecc_key *key, ecc_nb_ctx_t* ctx);

int wc_ecc_set_curve(ecc_key *key, int keysize, int curve_id);
```

C.22 dox_comments/header_files/eccsi.h

C.22.1 Functions

	Name
int	**wc_InitEccsiKey**(EccsiKey * key, void * heap, int devId)
int	**wc_InitEccsiKey_ex**(EccsiKey * key, int keySz, int curveId, void * heap, int devId)
void	**wc_FreeEccsiKey**(EccsiKey * key)
int	**wc_MakeEccsiKey**(EccsiKey * key, WC_RNG * rng)
int	**wc_MakeEccsiPair**(EccsiKey * key, WC_RNG * rng, enum wc_HashType hashType, const byte * id, word32 idSz, mp_int * ssk, ecc_point * pvt)
int	**wc_ValidateEccsiPair**(EccsiKey * key, enum wc_HashType hashType, const byte * id, word32 idSz, const mp_int * ssk, ecc_point * pvt, int * valid)
int	**wc_ValidateEccsiPvt**(EccsiKey * key, const ecc_point * pvt, int * valid)
int	**wc_EncodeEccsiPair**(const EccsiKey * key, mp_int * ssk, ecc_point * pvt, byte * data, word32 * sz)
int	**wc_EncodeEccsiSsk**(const EccsiKey * key, mp_int * ssk, byte * data, word32 * sz)
int	**wc_EncodeEccsiPvt**(const EccsiKey * key, ecc_point * pvt, byte * data, word32 * sz, int raw)
int	**wc_DecodeEccsiPair**(const EccsiKey * key, const byte * data, word32 sz, mp_int * ssk, ecc_point * pvt)
int	**wc_DecodeEccsiSsk**(const EccsiKey * key, const byte * data, word32 sz, mp_int * ssk)
int	**wc_DecodeEccsiPvt**(const EccsiKey * key, const byte * data, word32 sz, ecc_point * pvt)
int	**wc_DecodeEccsiPvtFromSig**(const EccsiKey * key, const byte * sig, word32 sz, ecc_point * pvt)
int	**wc_ExportEccsiKey**(EccsiKey * key, byte * data, word32 * sz)
int	**wc_ImportEccsiKey**(EccsiKey * key, const byte * data, word32 sz)
int	**wc_ExportEccsiPrivateKey**(EccsiKey * key, byte * data, word32 * sz)
int	**wc_ImportEccsiPrivateKey**(EccsiKey * key, const byte * data, word32 sz)
int	**wc_ExportEccsiPublicKey**(EccsiKey * key, byte * data, word32 * sz, int raw)
int	**wc_ImportEccsiPublicKey**(EccsiKey * key, const byte * data, word32 sz, int trusted)
int	**wc_HashEccsiId**(EccsiKey * key, enum wc_HashType hashType, const byte * id, word32 idSz, ecc_point * pvt, byte * hash, byte * hashSz)
int	**wc_SetEccsiHash**(EccsiKey * key, const byte * hash, byte hashSz)

COPYRIGHT ©2024 wolfSSL Inc.

	Name
int	**wc_SetEccsiPair**(EccsiKey * key, const mp_int * ssk, const ecc_point * pvt)
int	**wc_SignEccsiHash**(EccsiKey * key, WC_RNG * rng, enum wc_HashType hashType, const byte * msg, word32 msgSz, byte * sig, word32 * sigSz)
int	**wc_VerifyEccsiHash**(EccsiKey * key, enum wc_HashType hashType, const byte * msg, word32 msgSz, const byte * sig, word32 sigSz, int * verified)

C.22.2 Functions Documentation

C.22.2.1 function wc_InitEccsiKey

```
int wc_InitEccsiKey(
    EccsiKey * key,
    void * heap,
    int devId
)
```

C.22.2.2 function wc_InitEccsiKey_ex

```
int wc_InitEccsiKey_ex(
    EccsiKey * key,
    int keySz,
    int curveId,
    void * heap,
    int devId
)
```

C.22.2.3 function wc_FreeEccsiKey

```
void wc_FreeEccsiKey(
    EccsiKey * key
)
```

C.22.2.4 function wc_MakeEccsiKey

```
int wc_MakeEccsiKey(
    EccsiKey * key,
    WC_RNG * rng
)
```

C.22.2.5 function wc_MakeEccsiPair

```
int wc_MakeEccsiPair(
    EccsiKey * key,
    WC_RNG * rng,
    enum wc_HashType hashType,
    const byte * id,
    word32 idSz,
    mp_int * ssk,
```

```
    ecc_point * pvt
)
```

C.22.2.6 function wc_ValidateEccsiPair

```
int wc_ValidateEccsiPair(
    EccsiKey * key,
    enum wc_HashType hashType,
    const byte * id,
    word32 idSz,
    const mp_int * ssk,
    ecc_point * pvt,
    int * valid
)
```

C.22.2.7 function wc_ValidateEccsiPvt

```
int wc_ValidateEccsiPvt(
    EccsiKey * key,
    const ecc_point * pvt,
    int * valid
)
```

C.22.2.8 function wc_EncodeEccsiPair

```
int wc_EncodeEccsiPair(
    const EccsiKey * key,
    mp_int * ssk,
    ecc_point * pvt,
    byte * data,
    word32 * sz
)
```

C.22.2.9 function wc_EncodeEccsiSsk

```
int wc_EncodeEccsiSsk(
    const EccsiKey * key,
    mp_int * ssk,
    byte * data,
    word32 * sz
)
```

C.22.2.10 function wc_EncodeEccsiPvt

```
int wc_EncodeEccsiPvt(
    const EccsiKey * key,
    ecc_point * pvt,
    byte * data,
    word32 * sz,
    int raw
)
```

C.22.2.11 function wc_DecodeEccsiPair

```
int wc_DecodeEccsiPair(
    const EccsiKey * key,
    const byte * data,
    word32 sz,
    mp_int * ssk,
    ecc_point * pvt
)
```

C.22.2.12 function wc_DecodeEccsiSsk

```
int wc_DecodeEccsiSsk(
    const EccsiKey * key,
    const byte * data,
    word32 sz,
    mp_int * ssk
)
```

C.22.2.13 function wc_DecodeEccsiPvt

```
int wc_DecodeEccsiPvt(
    const EccsiKey * key,
    const byte * data,
    word32 sz,
    ecc_point * pvt
)
```

C.22.2.14 function wc_DecodeEccsiPvtFromSig

```
int wc_DecodeEccsiPvtFromSig(
    const EccsiKey * key,
    const byte * sig,
    word32 sz,
    ecc_point * pvt
)
```

C.22.2.15 function wc_ExportEccsiKey

```
int wc_ExportEccsiKey(
    EccsiKey * key,
    byte * data,
    word32 * sz
)
```

C.22.2.16 function wc_ImportEccsiKey

```
int wc_ImportEccsiKey(
    EccsiKey * key,
    const byte * data,
    word32 sz
)
```

C.22.2.17 function wc_ExportEccsiPrivateKey

```
int wc_ExportEccsiPrivateKey(
    EccsiKey * key,
    byte * data,
    word32 * sz
)
```

C.22.2.18 function wc_ImportEccsiPrivateKey

```
int wc_ImportEccsiPrivateKey(
    EccsiKey * key,
    const byte * data,
    word32 sz
)
```

C.22.2.19 function wc_ExportEccsiPublicKey

```
int wc_ExportEccsiPublicKey(
    EccsiKey * key,
    byte * data,
    word32 * sz,
    int raw
)
```

C.22.2.20 function wc_ImportEccsiPublicKey

```
int wc_ImportEccsiPublicKey(
    EccsiKey * key,
    const byte * data,
    word32 sz,
    int trusted
)
```

C.22.2.21 function wc_HashEccsiId

```
int wc_HashEccsiId(
    EccsiKey * key,
    enum wc_HashType hashType,
    const byte * id,
    word32 idSz,
    ecc_point * pvt,
    byte * hash,
    byte * hashSz
)
```

C.22.2.22 function wc_SetEccsiHash

```
int wc_SetEccsiHash(
    EccsiKey * key,
    const byte * hash,
    byte hashSz
)
```

C.22.2.23 function wc_SetEccsiPair

```
int wc_SetEccsiPair(
    EccsiKey * key,
    const mp_int * ssk,
    const ecc_point * pvt
)
```

C.22.2.24 function wc_SignEccsiHash

```
int wc_SignEccsiHash(
    EccsiKey * key,
    WC_RNG * rng,
    enum wc_HashType hashType,
    const byte * msg,
    word32 msgSz,
    byte * sig,
    word32 * sigSz
)
```

C.22.2.25 function wc_VerifyEccsiHash

```
int wc_VerifyEccsiHash(
    EccsiKey * key,
    enum wc_HashType hashType,
    const byte * msg,
    word32 msgSz,
    const byte * sig,
    word32 sigSz,
    int * verified
)
```

C.22.3 Source code

```
int wc_InitEccsiKey(EccsiKey* key, void* heap, int devId);
int wc_InitEccsiKey_ex(EccsiKey* key, int keySz, int curveId,
        void* heap, int devId);
void wc_FreeEccsiKey(EccsiKey* key);

int wc_MakeEccsiKey(EccsiKey* key, WC_RNG* rng);

int wc_MakeEccsiPair(EccsiKey* key, WC_RNG* rng,
        enum wc_HashType hashType, const byte* id, word32 idSz, mp_int* ssk,
        ecc_point* pvt);
int wc_ValidateEccsiPair(EccsiKey* key, enum wc_HashType hashType,
        const byte* id, word32 idSz, const mp_int* ssk, ecc_point* pvt,
        int* valid);
int wc_ValidateEccsiPvt(EccsiKey* key, const ecc_point* pvt,
        int* valid);
int wc_EncodeEccsiPair(const EccsiKey* key, mp_int* ssk,
        ecc_point* pvt, byte* data, word32* sz);
int wc_EncodeEccsiSsk(const EccsiKey* key, mp_int* ssk, byte* data,
        word32* sz);
int wc_EncodeEccsiPvt(const EccsiKey* key, ecc_point* pvt,
```

```
        byte* data, word32* sz, int raw);
int wc_DecodeEccsiPair(const EccsiKey* key, const byte* data,
        word32 sz, mp_int* ssk, ecc_point* pvt);
int wc_DecodeEccsiSsk(const EccsiKey* key, const byte* data,
        word32 sz, mp_int* ssk);
int wc_DecodeEccsiPvt(const EccsiKey* key, const byte* data,
        word32 sz, ecc_point* pvt);
int wc_DecodeEccsiPvtFromSig(const EccsiKey* key, const byte* sig,
        word32 sz, ecc_point* pvt);

int wc_ExportEccsiKey(EccsiKey* key, byte* data, word32* sz);
int wc_ImportEccsiKey(EccsiKey* key, const byte* data, word32 sz);

int wc_ExportEccsiPrivateKey(EccsiKey* key, byte* data, word32* sz);
int wc_ImportEccsiPrivateKey(EccsiKey* key, const byte* data,
        word32 sz);

int wc_ExportEccsiPublicKey(EccsiKey* key, byte* data, word32* sz,
        int raw);
int wc_ImportEccsiPublicKey(EccsiKey* key, const byte* data,
        word32 sz, int trusted);

int wc_HashEccsiId(EccsiKey* key, enum wc_HashType hashType,
        const byte* id, word32 idSz, ecc_point* pvt, byte* hash, byte* hashSz);
int wc_SetEccsiHash(EccsiKey* key, const byte* hash, byte hashSz);
int wc_SetEccsiPair(EccsiKey* key, const mp_int* ssk,
        const ecc_point* pvt);

int wc_SignEccsiHash(EccsiKey* key, WC_RNG* rng,
        enum wc_HashType hashType, const byte* msg, word32 msgSz, byte* sig,
        word32* sigSz);
int wc_VerifyEccsiHash(EccsiKey* key, enum wc_HashType hashType,
        const byte* msg, word32 msgSz, const byte* sig, word32 sigSz,
        int* verified);
```

C.23 dox_comments/header_files/ed25519.h

C.23.1 Functions

	Name
int	**wc_ed25519_make_public**(ed25519_key * key, unsigned char * pubKey, word32 pubKeySz)This function generates the Ed25519 public key from the private key, stored in the ed25519_key object. It stores the public key in the buffer pubKey.
int	**wc_ed25519_make_key**(WC_RNG * rng, int keysize, ed25519_key * key)This function generates a new Ed25519 key and stores it in key.

	Name
int	**wc_ed25519_sign_msg**(const byte * in, word32 inlen, byte * out, word32 * outlen, ed25519_key * key)This function signs a message using an ed25519_key object to guarantee authenticity.
int	**wc_ed25519ctx_sign_msg**(const byte * in, word32 inlen, byte * out, word32 * outlen, ed25519_key * key, const byte * context, byte contextLen)This function signs a message using an ed25519_key object to guarantee authenticity. The context is part of the data signed.
int	**wc_ed25519ph_sign_hash**(const byte * hash, word32 hashLen, byte * out, word32 * outLen, ed25519_key * key, const byte * context, byte contextLen)This function signs a message digest using an ed25519_key object to guarantee authenticity. The context is included as part of the data signed. The message is pre-hashed before signature calculation.
int	**wc_ed25519ph_sign_msg**(const byte * in, word32 inlen, byte * out, word32 * outlen, ed25519_key * key, const byte * context, byte contextLen)This function signs a message using an ed25519_key object to guarantee authenticity. The context is included as part of the data signed. The message is pre-hashed before signature calculation.
int	**wc_ed25519_verify_msg**(const byte * sig, word32 siglen, const byte * msg, word32 msgLen, int * ret, ed25519_key * key)This function verifies the Ed25519 signature of a message to ensure authenticity. It returns the answer through ret, with 1 corresponding to a valid signature, and 0 corresponding to an invalid signature.
int	**wc_ed25519ctx_verify_msg**(const byte * sig, word32 siglen, const byte * msg, word32 msgLen, int * ret, ed25519_key * key, const byte * context, byte contextLen)This function verifies the Ed25519 signature of a message to ensure authenticity. The context is included as part of the data verified. It returns the answer through ret, with 1 corresponding to a valid signature, and 0 corresponding to an invalid signature.

	Name
int	**wc_ed25519ph_verify_hash**(const byte * sig, word32 siglen, const byte * hash, word32 hashLen, int * ret, ed25519_key * key, const byte * context, byte contextLen)This function verifies the Ed25519 signature of the digest of a message to ensure authenticity. The context is included as part of the data verified. The hash is the pre-hashed message before signature calculation. The hash algorithm used to create message digest must be SHA-512. The answer is returned through ret, with 1 corresponding to a valid signature, and 0 corresponding to an invalid signature.
int	**wc_ed25519ph_verify_msg**(const byte * sig, word32 siglen, const byte * msg, word32 msgLen, int * ret, ed25519_key * key, const byte * context, byte contextLen)This function verifies the Ed25519 signature of a message to ensure authenticity. The context is included as part of the data verified. The message is pre-hashed before verification. It returns the answer through ret, with 1 corresponding to a valid signature, and 0 corresponding to an invalid signature.
int	**wc_ed25519_init**(ed25519_key * key)This function initializes an ed25519_key object for future use with message verification.
void	**wc_ed25519_free**(ed25519_key * key)This function frees an Ed25519 object after it has been used.
int	**wc_ed25519_import_public**(const byte * in, word32 inLen, ed25519_key * key)This function imports a public ed25519_key from a buffer containing the public key. This function will handle both compressed and uncompressed keys. The public key is checked that it matches the private key when one is present.
int	**wc_ed25519_import_public_ex**(const byte * in, word32 inLen, ed25519_key * key, int trusted)This function imports a public ed25519_key from a buffer containing the public key. This function will handle both compressed and uncompressed keys. Check public key matches private key, when present, when not trusted.
int	**wc_ed25519_import_private_only**(const byte * priv, word32 privSz, ed25519_key * key)This function imports an Ed25519 private key only from a buffer.

	Name
int	**wc_ed25519_import_private_key**(const byte * priv, word32 privSz, const byte * pub, word32 pubSz, ed25519_key * key)This function imports a public/private Ed25519 key pair from a pair of buffers. This function will handle both compressed and uncompressed keys. The public key is assumed to be untrusted and is checked against the private key.
int	**wc_ed25519_import_private_key_ex**(const byte * priv, word32 privSz, const byte * pub, word32 pubSz, ed25519_key * key, int trusted)This function imports a public/private Ed25519 key pair from a pair of buffers. This function will handle both compressed and uncompressed keys. The public is checked against private key if not trusted.
int	**wc_ed25519_export_public**(ed25519_key * key, byte * out, word32 * outLen)This function exports the private key from an ed25519_key structure. It stores the public key in the buffer out, and sets the bytes written to this buffer in outLen.
int	**wc_ed25519_export_private_only**(ed25519_key * key, byte * out, word32 * outLen)This function exports only the private key from an ed25519_key structure. It stores the private key in the buffer out, and sets the bytes written to this buffer in outLen.
int	**wc_ed25519_export_private**(ed25519_key * key, byte * out, word32 * outLen)This function exports the key pair from an ed25519_key structure. It stores the key pair in the buffer out, and sets the bytes written to this buffer in outLen.
int	**wc_ed25519_export_key**(ed25519_key * key, byte * priv, word32 * privSz, byte * pub, word32 * pubSz)This function exports the private and public key separately from an ed25519_key structure. It stores the private key in the buffer priv, and sets the bytes written to this buffer in privSz. It stores the public key in the buffer pub, and sets the bytes written to this buffer in pubSz.
int	**wc_ed25519_check_key**(ed25519_key * key)This function checks the public key in ed25519_key structure matches the private key.
int	**wc_ed25519_size**(ed25519_key * key)This function returns the size of an Ed25519 - 32 bytes.
int	**wc_ed25519_priv_size**(ed25519_key * key)This function returns the private key size (secret + public) in bytes.

	Name
int	**wc_ed25519_pub_size**(ed25519_key * key)This function returns the compressed key size in bytes (public key).
int	**wc_ed25519_sig_size**(ed25519_key * key)This function returns the size of an Ed25519 signature (64 in bytes).

C.23.2 Functions Documentation

C.23.2.1 function wc_ed25519_make_public

```
int wc_ed25519_make_public(
    ed25519_key * key,
    unsigned char * pubKey,
    word32 pubKeySz
)
```

This function generates the Ed25519 public key from the private key, stored in the ed25519_key object. It stores the public key in the buffer pubKey.

Parameters:

- **key** Pointer to the ed25519_key for which to generate a key.
- **pubKey** Pointer to the buffer in which to store the public key.
- **pubKeySz** Size of the public key. Should be ED25519_PUB_KEY_SIZE.

See:

- wc_ed25519_init
- wc_ed25519_import_private_only
- wc_ed25519_make_key

Return:

- 0 Returned upon successfully making the public key.
- BAD_FUNC_ARG Returned if key or pubKey evaluate to NULL, or if the specified key size is not 32 bytes (Ed25519 has 32 byte keys).
- ECC_PRIV_KEY_E returned if the ed25519_key object does not have the private key in it.
- MEMORY_E Returned if there is an error allocating memory during function execution.

Example

```
int ret;

ed25519_key key;
byte priv[] = { initialize with 32 byte private key };
byte pub[32];
word32 pubSz = sizeof(pub);

wc_ed25519_init(&key);
wc_ed25519_import_private_only(priv, sizeof(priv), &key);
ret = wc_ed25519_make_public(&key, pub, &pubSz);
if (ret != 0) {
    // error making public key
}
```

C.23.2.2 function wc_ed25519_make_key

```
int wc_ed25519_make_key(
    WC_RNG * rng,
    int keysize,
    ed25519_key * key
)
```

This function generates a new Ed25519 key and stores it in key.

Parameters:

- **rng** Pointer to an initialized RNG object with which to generate the key.
- **keysize** Length of key to generate. Should always be 32 for Ed25519.
- **key** Pointer to the ed25519_key for which to generate a key.

See: wc_ed25519_init

Return:

- 0 Returned upon successfully making an ed25519_key.
- BAD_FUNC_ARG Returned if rng or key evaluate to NULL, or if the specified key size is not 32 bytes (Ed25519 has 32 byte keys).
- MEMORY_E Returned if there is an error allocating memory during function execution.

Example

```
int ret;

WC_RNG rng;
ed25519_key key;

wc_InitRng(&rng);
wc_ed25519_init(&key);
wc_ed25519_make_key(&rng, 32, &key);
if (ret != 0) {
    // error making key
}
```

C.23.2.3 function wc_ed25519_sign_msg

```
int wc_ed25519_sign_msg(
    const byte * in,
    word32 inlen,
    byte * out,
    word32 * outlen,
    ed25519_key * key
)
```

This function signs a message using an ed25519_key object to guarantee authenticity.

Parameters:

- **in** Pointer to the buffer containing the message to sign.
- **inlen** Length of the message to sign.
- **out** Buffer in which to store the generated signature.
- **outlen** Maximum length of the output buffer. Will store the bytes written to out upon successfully generating a message signature.
- **key** Pointer to a private ed25519_key with which to generate the signature.

See:

- wc_ed25519ctx_sign_msg
- wc_ed25519ph_sign_hash
- wc_ed25519ph_sign_msg
- wc_ed25519_verify_msg

Return:

- 0 Returned upon successfully generating a signature for the message.
- BAD_FUNC_ARG Returned if any of the input parameters evaluate to NULL, or if the output buffer is too small to store the generated signature.
- MEMORY_E Returned if there is an error allocating memory during function execution.

Example

```
ed25519_key key;
WC_RNG rng;
int ret, sigSz;

byte sig[64]; // will hold generated signature
sigSz = sizeof(sig);
byte message[] = { initialize with message };

wc_InitRng(&rng); // initialize rng
wc_ed25519_init(&key); // initialize key
wc_ed25519_make_key(&rng, 32, &key); // make public/private key pair
ret = wc_ed25519_sign_msg(message, sizeof(message), sig, &sigSz, &key);
if (ret != 0) {
    // error generating message signature
}
```

C.23.2.4 function wc_ed25519ctx_sign_msg

```
int wc_ed25519ctx_sign_msg(
    const byte * in,
    word32 inlen,
    byte * out,
    word32 * outlen,
    ed25519_key * key,
    const byte * context,
    byte contextLen
)
```

This function signs a message using an ed25519_key object to guarantee authenticity. The context is part of the data signed.

Parameters:

- **in** Pointer to the buffer containing the message to sign.
- **inlen** Length of the message to sign.
- **out** Buffer in which to store the generated signature.
- **outlen** Maximum length of the output buffer. Will store the bytes written to out upon successfully generating a message signature.
- **key** Pointer to a private ed25519_key with which to generate the signature.
- **context** Pointer to the buffer containing the context for which message is being signed.
- **contextLen** Length of the context buffer.

See:

- wc_ed25519_sign_msg

- wc_ed25519ph_sign_hash
- wc_ed25519ph_sign_msg
- wc_ed25519_verify_msg

Return:

- 0 Returned upon successfully generating a signature for the message.
- BAD_FUNC_ARG Returned any of the input parameters evaluate to NULL, or if the output buffer is too small to store the generated signature.
- MEMORY_E Returned if there is an error allocating memory during function execution.

Example

```
ed25519_key key;
WC_RNG rng;
int ret, sigSz;

byte sig[64]; // will hold generated signature
sigSz = sizeof(sig);
byte message[] = { initialize with message };
byte context[] = { initialize with context of signing };

wc_InitRng(&rng); // initialize rng
wc_ed25519_init(&key); // initialize key
wc_ed25519_make_key(&rng, 32, &key); // make public/private key pair
ret = wc_ed25519ctx_sign_msg(message, sizeof(message), sig, &sigSz, &key,
        context, sizeof(context));
if (ret != 0) {
    // error generating message signature
}
```

C.23.2.5 function wc_ed25519ph_sign_hash

```
int wc_ed25519ph_sign_hash(
    const byte * hash,
    word32 hashLen,
    byte * out,
    word32 * outLen,
    ed25519_key * key,
    const byte * context,
    byte contextLen
)
```

This function signs a message digest using an ed25519_key object to guarantee authenticity. The context is included as part of the data signed. The message is pre-hashed before signature calculation.

Parameters:

- **hash** Pointer to the buffer containing the hash of the message to sign.
- **hashLen** Length of the hash of the message to sign.
- **out** Buffer in which to store the generated signature.
- **outlen** Maximum length of the output buffer. Will store the bytes written to out upon successfully generating a message signature.
- **key** Pointer to a private ed25519_key with which to generate the signature.
- **context** Pointer to the buffer containing the context for which message is being signed.
- **contextLen** Length of the context buffer.

See:

- wc_ed25519_sign_msg
- wc_ed25519ctx_sign_msg
- wc_ed25519ph_sign_msg
- wc_ed25519_verify_msg

Return:

- 0 Returned upon successfully generating a signature for the message digest.
- BAD_FUNC_ARG Returned any of the input parameters evaluate to NULL, or if the output buffer is too small to store the generated signature.
- MEMORY_E Returned if there is an error allocating memory during function execution.

Example

```
ed25519_key key;
WC_RNG rng;
int ret, sigSz;

byte sig[64]; // will hold generated signature
sigSz = sizeof(sig);
byte hash[] = { initialize with SHA-512 hash of message };
byte context[] = { initialize with context of signing };

wc_InitRng(&rng); // initialize rng
wc_ed25519_init(&key); // initialize key
wc_ed25519_make_key(&rng, 32, &key); // make public/private key pair
ret = wc_ed25519ph_sign_hash(hash, sizeof(hash), sig, &sigSz, &key,
        context, sizeof(context));
if (ret != 0) {
    // error generating message signature
}
```

C.23.2.6 function wc_ed25519ph_sign_msg

```
int wc_ed25519ph_sign_msg(
    const byte * in,
    word32 inlen,
    byte * out,
    word32 * outlen,
    ed25519_key * key,
    const byte * context,
    byte contextLen
)
```

This function signs a message using an ed25519_key object to guarantee authenticity. The context is included as part of the data signed. The message is pre-hashed before signature calculation.

Parameters:

- **in** Pointer to the buffer containing the message to sign.
- **inlen** Length of the message to sign.
- **out** Buffer in which to store the generated signature.
- **outlen** Maximum length of the output buffer. Will store the bytes written to out upon successfully generating a message signature.
- **key** Pointer to a private ed25519_key with which to generate the signature.
- **context** Pointer to the buffer containing the context for which message is being signed.
- **contextLen** Length of the context buffer.

See:

- wc_ed25519_sign_msg
- wc_ed25519ctx_sign_msg
- wc_ed25519ph_sign_hash
- wc_ed25519_verify_msg

Return:

- 0 Returned upon successfully generating a signature for the message.
- BAD_FUNC_ARG Returned any of the input parameters evaluate to NULL, or if the output buffer is too small to store the generated signature.
- MEMORY_E Returned if there is an error allocating memory during function execution.

Example

```
ed25519_key key;
WC_RNG rng;
int ret, sigSz;

byte sig[64]; // will hold generated signature
sigSz = sizeof(sig);
byte message[] = { initialize with message };
byte context[] = { initialize with context of signing };

wc_InitRng(&rng); // initialize rng
wc_ed25519_init(&key); // initialize key
wc_ed25519_make_key(&rng, 32, &key); // make public/private key pair
ret = wc_ed25519ph_sign_msg(message, sizeof(message), sig, &sigSz, &key,
        context, sizeof(context));
if (ret != 0) {
    // error generating message signature
}
```

C.23.2.7 function wc_ed25519_verify_msg

```
int wc_ed25519_verify_msg(
    const byte * sig,
    word32 siglen,
    const byte * msg,
    word32 msgLen,
    int * ret,
    ed25519_key * key
)
```

This function verifies the Ed25519 signature of a message to ensure authenticity. It returns the answer through ret, with 1 corresponding to a valid signature, and 0 corresponding to an invalid signature.

Parameters:

- **sig** Pointer to the buffer containing the signature to verify.
- **siglen** Length of the signature to verify.
- **msg** Pointer to the buffer containing the message to verify.
- **msgLen** Length of the message to verify.
- **ret** Pointer to the result of the verification. 1 indicates the message was successfully verified.
- **key** Pointer to a public Ed25519 key with which to verify the signature.

See:

- wc_ed25519ctx_verify_msg
- wc_ed25519ph_verify_hash
- wc_ed25519ph_verify_msg
- wc_ed25519_sign_msg

Return:

- 0 Returned upon successfully performing the signature verification and authentication.
- BAD_FUNC_ARG Returned if any of the input parameters evaluate to NULL, or if the siglen does not match the actual length of a signature.
- SIG_VERIFY_E Returned if verification completes, but the signature generated does not match the signature provided.

Example

```
ed25519_key key;
int ret, verified = 0;

byte sig[] { initialize with received signature };
byte msg[] = { initialize with message };
// initialize key with received public key
ret = wc_ed25519_verify_msg(sig, sizeof(sig), msg, sizeof(msg), &verified,
        &key);
if (ret < 0) {
    // error performing verification
} else if (verified == 0)
    // the signature is invalid
}
```

C.23.2.8 function wc_ed25519ctx_verify_msg

```
int wc_ed25519ctx_verify_msg(
    const byte * sig,
    word32 siglen,
    const byte * msg,
    word32 msgLen,
    int * ret,
    ed25519_key * key,
    const byte * context,
    byte contextLen
)
```

This function verifies the Ed25519 signature of a message to ensure authenticity. The context is included as part of the data verified. It returns the answer through ret, with 1 corresponding to a valid signature, and 0 corresponding to an invalid signature.

Parameters:

- **sig** Pointer to the buffer containing the signature to verify.
- **siglen** Length of the signature to verify.
- **msg** Pointer to the buffer containing the message to verify.
- **msgLen** Length of the message to verify.
- **ret** Pointer to the result of the verification. 1 indicates the message was successfully verified.
- **key** Pointer to a public Ed25519 key with which to verify the signature.
- **context** Pointer to the buffer containing the context for which the message was signed.
- **contextLen** Length of the context buffer.

See:

- wc_ed25519_verify_msg
- wc_ed25519ph_verify_hash
- wc_ed25519ph_verify_msg
- wc_ed25519_sign_msg

Return:

- 0 Returned upon successfully performing the signature verification and authentication.
- BAD_FUNC_ARG Returned if any of the input parameters evaluate to NULL, or if the siglen does not match the actual length of a signature.
- SIG_VERIFY_E Returned if verification completes, but the signature generated does not match the signature provided.

Example

```
ed25519_key key;
int ret, verified = 0;

byte sig[] { initialize with received signature };
byte msg[] = { initialize with message };
byte context[] = { initialize with context of signature };
// initialize key with received public key
ret = wc_ed25519ctx_verify_msg(sig, sizeof(sig), msg, sizeof(msg),
        &verified, &key, );
if (ret < 0) {
    // error performing verification
} else if (verified == 0)
    // the signature is invalid
}
```

C.23.2.9 function wc_ed25519ph_verify_hash

```
int wc_ed25519ph_verify_hash(
    const byte * sig,
    word32 siglen,
    const byte * hash,
    word32 hashLen,
    int * ret,
    ed25519_key * key,
    const byte * context,
    byte contextLen
)
```

This function verifies the Ed25519 signature of the digest of a message to ensure authenticity. The context is included as part of the data verified. The hash is the pre-hashed message before signature calculation. The hash algorithm used to create message digest must be SHA-512. The answer is returned through ret, with 1 corresponding to a valid signature, and 0 corresponding to an invalid signature.

Parameters:

- **sig** Pointer to the buffer containing the signature to verify.
- **siglen** Length of the signature to verify.
- **hash** Pointer to the buffer containing the hash of the message to verify.
- **hashLen** Length of the hash to verify.
- **ret** Pointer to the result of the verification. 1 indicates the message was successfully verified.
- **key** Pointer to a public Ed25519 key with which to verify the signature.
- **context** Pointer to the buffer containing the context for which the message was signed.

- **contextLen** Length of the context buffer.

See:

- wc_ed25519_verify_msg
- wc_ed25519ctx_verify_msg
- wc_ed25519ph_verify_msg
- wc_ed25519_sign_msg

Return:

- 0 Returned upon successfully performing the signature verification and authentication.
- BAD_FUNC_ARG Returned if any of the input parameters evaluate to NULL, or if the siglen does not match the actual length of a signature.
- SIG_VERIFY_E Returned if verification completes, but the signature generated does not match the signature provided.

Example

```
ed25519_key key;
int ret, verified = 0;

byte sig[] { initialize with received signature };
byte hash[] = { initialize with SHA-512 hash of message };
byte context[] = { initialize with context of signature };
// initialize key with received public key
ret = wc_ed25519ph_verify_hash(sig, sizeof(sig), msg, sizeof(msg),
        &verified, &key, );
if (ret < 0) {
    // error performing verification
} else if (verified == 0)
    // the signature is invalid
}
```

C.23.2.10 function wc_ed25519ph_verify_msg

```
int wc_ed25519ph_verify_msg(
    const byte * sig,
    word32 siglen,
    const byte * msg,
    word32 msgLen,
    int * ret,
    ed25519_key * key,
    const byte * context,
    byte contextLen
)
```

This function verifies the Ed25519 signature of a message to ensure authenticity. The context is included as part of the data verified. The message is pre-hashed before verification. It returns the answer through ret, with 1 corresponding to a valid signature, and 0 corresponding to an invalid signature.

Parameters:

- **sig** Pointer to the buffer containing the signature to verify.
- **siglen** Length of the signature to verify.
- **msg** Pointer to the buffer containing the message to verify.
- **msgLen** Length of the message to verify.
- **ret** Pointer to the result of the verification. 1 indicates the message was successfully verified.

- **key** Pointer to a public Ed25519 key with which to verify the signature.
- **context** Pointer to the buffer containing the context for which the message was signed.
- **contextLen** Length of the context buffer.

See:

- wc_ed25519_verify_msg
- wc_ed25519ph_verify_hash
- wc_ed25519ph_verify_msg
- wc_ed25519_sign_msg

Return:

- 0 Returned upon successfully performing the signature verification and authentication.
- BAD_FUNC_ARG Returned if any of the input parameters evaluate to NULL, or if the siglen does not match the actual length of a signature.
- SIG_VERIFY_E Returned if verification completes, but the signature generated does not match the signature provided.

Example

```
ed25519_key key;
int ret, verified = 0;

byte sig[] { initialize with received signature };
byte msg[] = { initialize with message };
byte context[] = { initialize with context of signature };
// initialize key with received public key
ret = wc_ed25519ctx_verify_msg(sig, sizeof(sig), msg, sizeof(msg),
        &verified, &key, );
if (ret < 0) {
    // error performing verification
} else if (verified == 0)
    // the signature is invalid
}
```

C.23.2.11 function wc_ed25519_init

```
int wc_ed25519_init(
    ed25519_key * key
)
```

This function initializes an ed25519_key object for future use with message verification.

Parameters:

- **key** Pointer to the ed25519_key object to initialize.

See:

- wc_ed25519_make_key
- wc_ed25519_free

Return:

- 0 Returned upon successfully initializing the ed25519_key object.
- BAD_FUNC_ARG Returned if key is NULL.

Example

```
ed25519_key key;
wc_ed25519_init(&key);
```

C.23.2.12 function wc_ed25519_free

```
void wc_ed25519_free(
    ed25519_key * key
)
```

This function frees an Ed25519 object after it has been used.

Parameters:

- **key** Pointer to the ed25519_key object to free

See: wc_ed25519_init

Example

```
ed25519_key key;
// initialize key and perform secure exchanges
...
wc_ed25519_free(&key);
```

C.23.2.13 function wc_ed25519_import_public

```
int wc_ed25519_import_public(
    const byte * in,
    word32 inLen,
    ed25519_key * key
)
```

This function imports a public ed25519_key from a buffer containing the public key. This function will handle both compressed and uncompressed keys. The public key is checked that it matches the private key when one is present.

Parameters:

- **in** Pointer to the buffer containing the public key.
- **inLen** Length of the buffer containing the public key.
- **key** Pointer to the ed25519_key object in which to store the public key.

See:

- wc_ed25519_import_public_ex
- wc_ed25519_import_private_key
- wc_ed25519_import_private_key_ex
- wc_ed25519_export_public

Return:

- 0 Returned on successfully importing the ed25519_key.
- BAD_FUNC_ARG Returned if in or key evaluate to NULL, or inLen is less than the size of an Ed25519 key.

Example

```
int ret;
byte pub[] = { initialize Ed25519 public key };

ed_25519 key;
wc_ed25519_init_key(&key);
ret = wc_ed25519_import_public(pub, sizeof(pub), &key);
if (ret != 0) {
```

```
    // error importing key
}
```

C.23.2.14 function wc_ed25519_import_public_ex

```
int wc_ed25519_import_public_ex(
    const byte * in,
    word32 inLen,
    ed25519_key * key,
    int trusted
)
```

This function imports a public ed25519_key from a buffer containing the public key. This function will handle both compressed and uncompressed keys. Check public key matches private key, when present, when not trusted.

Parameters:

- **in** Pointer to the buffer containing the public key.
- **inLen** Length of the buffer containing the public key.
- **key** Pointer to the ed25519_key object in which to store the public key.
- **trusted** Public key data is trusted or not.

See:

- wc_ed25519_import_public
- wc_ed25519_import_private_key
- wc_ed25519_import_private_key_ex
- wc_ed25519_export_public

Return:

- 0 Returned on successfully importing the ed25519_key.
- BAD_FUNC_ARG Returned if in or key evaluate to NULL, or inLen is less than the size of an Ed25519 key.

Example

```
int ret;
byte pub[] = { initialize Ed25519 public key };

ed_25519 key;
wc_ed25519_init_key(&key);
ret = wc_ed25519_import_public_ex(pub, sizeof(pub), &key, 1);
if (ret != 0) {
    // error importing key
}
```

C.23.2.15 function wc_ed25519_import_private_only

```
int wc_ed25519_import_private_only(
    const byte * priv,
    word32 privSz,
    ed25519_key * key
)
```

This function imports an Ed25519 private key only from a buffer.

Parameters:

- **priv** Pointer to the buffer containing the private key.
- **privSz** Length of the private key.
- **key** Pointer to the ed25519_key object in which to store the imported private key.

See:

- wc_ed25519_import_public
- wc_ed25519_import_public_ex
- wc_ed25519_import_private_key
- wc_ed25519_import_private_key_ex
- wc_ed25519_export_private_only

Return:

- 0 Returned on successfully importing the Ed25519 key.
- BAD_FUNC_ARG Returned if priv or key evaluate to NULL, or if privSz is not equal to ED25519_KEY_SIZE.

Example

```
int ret;
byte priv[] = { initialize with 32 byte private key };

ed25519_key key;
wc_ed25519_init_key(&key);
ret = wc_ed25519_import_private_only(priv, sizeof(priv), &key);
if (ret != 0) {
    // error importing private key
}
```

C.23.2.16 function wc_ed25519_import_private_key

```
int wc_ed25519_import_private_key(
    const byte * priv,
    word32 privSz,
    const byte * pub,
    word32 pubSz,
    ed25519_key * key
)
```

This function imports a public/private Ed25519 key pair from a pair of buffers. This function will handle both compressed and uncompressed keys. The public key is assumed to be untrusted and is checked against the private key.

Parameters:

- **priv** Pointer to the buffer containing the private key.
- **privSz** Length of the private key.
- **pub** Pointer to the buffer containing the public key.
- **pubSz** Length of the public key.
- **key** Pointer to the ed25519_key object in which to store the imported private/public key pair.

See:

- wc_ed25519_import_public
- wc_ed25519_import_public_ex
- wc_ed25519_import_private_only
- wc_ed25519_import_private_key_ex
- wc_ed25519_export_private

Return:

- 0 Returned on successfully importing the ed25519_key.
- BAD_FUNC_ARG Returned if priv or key evaluate to NULL; or if either privSz is not equal to ED25519_KEY_SIZE nor ED25519_PRV_KEY_SIZE, or pubSz is less than ED25519_PUB_KEY_SIZE.

Example

```
int ret;
byte priv[] = { initialize with 32 byte private key };
byte pub[]  = { initialize with the corresponding public key };

ed25519_key key;
wc_ed25519_init_key(&key);
ret = wc_ed25519_import_private_key(priv, sizeof(priv), pub, sizeof(pub),
        &key);
if (ret != 0) {
    // error importing key
}
```

C.23.2.17 function wc_ed25519_import_private_key_ex

```
int wc_ed25519_import_private_key_ex(
    const byte * priv,
    word32 privSz,
    const byte * pub,
    word32 pubSz,
    ed25519_key * key,
    int trusted
)
```

This function imports a public/private Ed25519 key pair from a pair of buffers. This function will handle both compressed and uncompressed keys. The public is checked against private key if not trusted.

Parameters:

- **priv** Pointer to the buffer containing the private key.
- **privSz** Length of the private key.
- **pub** Pointer to the buffer containing the public key.
- **pubSz** Length of the public key.
- **key** Pointer to the ed25519_key object in which to store the imported private/public key pair.
- **trusted** Public key data is trusted or not.

See:

- wc_ed25519_import_public
- wc_ed25519_import_public_ex
- wc_ed25519_import_private_only
- wc_ed25519_import_private_key
- wc_ed25519_export_private

Return:

- 0 Returned on successfully importing the ed25519_key.
- BAD_FUNC_ARG Returned if priv or key evaluate to NULL; or if either privSz is not equal to ED25519_KEY_SIZE nor ED25519_PRV_KEY_SIZE, or pubSz is less than ED25519_PUB_KEY_SIZE.

Example

```
int ret;
byte priv[] = { initialize with 32 byte private key };
byte pub[]  = { initialize with the corresponding public key };

ed25519_key key;
wc_ed25519_init_key(&key);
ret = wc_ed25519_import_private_key(priv, sizeof(priv), pub, sizeof(pub),
        &key, 1);
if (ret != 0) {
    // error importing key
}
```

C.23.2.18 function wc_ed25519_export_public

```
int wc_ed25519_export_public(
    ed25519_key * key,
    byte * out,
    word32 * outLen
)
```

This function exports the private key from an ed25519_key structure. It stores the public key in the buffer out, and sets the bytes written to this buffer in outLen.

Parameters:

- **key** Pointer to an ed25519_key structure from which to export the public key.
- **out** Pointer to the buffer in which to store the public key.
- **outLen** Pointer to a word32 object with the size available in out. Set with the number of bytes written to out after successfully exporting the public key.

See:

- wc_ed25519_import_public
- wc_ed25519_import_public_ex
- wc_ed25519_export_private_only

Return:

- 0 Returned upon successfully exporting the public key.
- BAD_FUNC_ARG Returned if any of the input values evaluate to NULL.
- BUFFER_E Returned if the buffer provided is not large enough to store the private key. Upon returning this error, the function sets the size required in outLen.

Example

```
int ret;
ed25519_key key;
// initialize key, make key

char pub[32];
word32 pubSz = sizeof(pub);

ret = wc_ed25519_export_public(&key, pub, &pubSz);
if (ret != 0) {
    // error exporting public key
}
```

C.23.2.19 function wc_ed25519_export_private_only

```
int wc_ed25519_export_private_only(
    ed25519_key * key,
    byte * out,
    word32 * outLen
)
```

This function exports only the private key from an ed25519_key structure. It stores the private key in the buffer out, and sets the bytes written to this buffer in outLen.

Parameters:

- **key** Pointer to an ed25519_key structure from which to export the private key.
- **out** Pointer to the buffer in which to store the private key.
- **outLen** Pointer to a word32 object with the size available in out. Set with the number of bytes written to out after successfully exporting the private key.

See:

- wc_ed25519_export_public
- wc_ed25519_import_private_key
- wc_ed25519_import_private_key_ex

Return:

- 0 Returned upon successfully exporting the private key.
- BAD_FUNC_ARG Returned if any of the input values evaluate to NULL.
- BUFFER_E Returned if the buffer provided is not large enough to store the private key.

Example

```
int ret;
ed25519_key key;
// initialize key, make key

char priv[32]; // 32 bytes because only private key
word32 privSz = sizeof(priv);
ret = wc_ed25519_export_private_only(&key, priv, &privSz);
if (ret != 0) {
    // error exporting private key
}
```

C.23.2.20 function wc_ed25519_export_private

```
int wc_ed25519_export_private(
    ed25519_key * key,
    byte * out,
    word32 * outLen
)
```

This function exports the key pair from an ed25519_key structure. It stores the key pair in the buffer out, and sets the bytes written to this buffer in outLen.

Parameters:

- **key** Pointer to an ed25519_key structure from which to export the key pair.
- **out** Pointer to the buffer in which to store the key pair.
- **outLen** Pointer to a word32 object with the size available in out. Set with the number of bytes written to out after successfully exporting the key pair.

See:

- wc_ed25519_import_private_key
- wc_ed25519_import_private_key_ex
- wc_ed25519_export_private_only

Return:

- 0 Returned upon successfully exporting the key pair.
- BAD_FUNC_ARG Returned if any of the input values evaluate to NULL.
- BUFFER_E Returned if the buffer provided is not large enough to store the key pair.

Example

```
ed25519_key key;
wc_ed25519_init(&key);

WC_RNG rng;
wc_InitRng(&rng);

wc_ed25519_make_key(&rng, 32, &key); // initialize 32 byte Ed25519 key

byte out[64]; // out needs to be a sufficient buffer size
word32 outLen = sizeof(out);
int key_size = wc_ed25519_export_private(&key, out, &outLen);
if (key_size == BUFFER_E) {
    // Check size of out compared to outLen to see if function reset outLen
}
```

C.23.2.21 function wc_ed25519_export_key

```
int wc_ed25519_export_key(
    ed25519_key * key,
    byte * priv,
    word32 * privSz,
    byte * pub,
    word32 * pubSz
)
```

This function exports the private and public key separately from an ed25519_key structure. It stores the private key in the buffer priv, and sets the bytes written to this buffer in privSz. It stores the public key in the buffer pub, and sets the bytes written to this buffer in pubSz.

Parameters:

- **key** Pointer to an ed25519_key structure from which to export the key pair.
- **priv** Pointer to the buffer in which to store the private key.
- **privSz** Pointer to a word32 object with the size available in out. Set with the number of bytes written to out after successfully exporting the private key.
- **pub** Pointer to the buffer in which to store the public key.
- **pubSz** Pointer to a word32 object with the size available in out. Set with the number of bytes written to out after successfully exporting the public key.

See:

- wc_ed25519_export_private
- wc_ed25519_export_public

Return:

- 0 Returned upon successfully exporting the key pair.
- BAD_FUNC_ARG Returned if any of the input values evaluate to NULL.
- BUFFER_E Returned if the buffer provided is not large enough to store the key pair.

Example

```
int ret;
ed25519_key key;
// initialize key, make key

char pub[32];
word32 pubSz = sizeof(pub);
char priv[32];
word32 privSz = sizeof(priv);

ret = wc_ed25519_export_key(&key, priv, &pubSz, pub, &pubSz);
if (ret != 0) {
    // error exporting public key
}
```

C.23.2.22 function wc_ed25519_check_key

```
int wc_ed25519_check_key(
    ed25519_key * key
)
```

This function checks the public key in ed25519_key structure matches the private key.

Parameters:

- **key** Pointer to an ed25519_key structure holding a private and public key.

See:

- wc_ed25519_import_private_key
- wc_ed25519_import_private_key_ex

Return:

- 0 Returned if the private and public key matched.
- BAD_FUNC_ARG Returned if the given key is NULL.
- PUBLIC_KEY_E Returned if the no public key available or is invalid.

Example

```
int ret;
byte priv[] = { initialize with 57 byte private key };
byte pub[]  = { initialize with the corresponding public key };

ed25519_key key;
wc_ed25519_init_key(&key);
wc_ed25519_import_private_key_ex(priv, sizeof(priv), pub, sizeof(pub), &key,
    1);
ret = wc_ed25519_check_key(&key);
if (ret != 0) {
    // error checking key
}
```

C.23.2.23 function wc_ed25519_size

```
int wc_ed25519_size(
    ed25519_key * key
)
```

This function returns the size of an Ed25519 - 32 bytes.

Parameters:

- **key** Pointer to an ed25519_key structure for which to get the key size.

See: wc_ed25519_make_key

Return:

- ED25519_KEY_SIZE The size of a valid private key (32 bytes).
- BAD_FUNC_ARG Returned if the given key is NULL.

Example

```
int keySz;
ed25519_key key;
// initialize key, make key
keySz = wc_ed25519_size(&key);
if (keySz == 0) {
    // error determining key size
}
```

C.23.2.24 function wc_ed25519_priv_size

```
int wc_ed25519_priv_size(
    ed25519_key * key
)
```

This function returns the private key size (secret + public) in bytes.

Parameters:

- **key** Pointer to an ed25519_key structure for which to get the key size.

See: wc_ed25519_pub_size

Return:

- ED25519_PRV_KEY_SIZE The size of the private key (64 bytes).
- BAD_FUNC_ARG Returned if key argument is NULL.

Example

```
ed25519_key key;
wc_ed25519_init(&key);

WC_RNG rng;
wc_InitRng(&rng);

wc_ed25519_make_key(&rng, 32, &key); // initialize 32 byte Ed25519 key
int key_size = wc_ed25519_priv_size(&key);
```

C.23.2.25 function wc_ed25519_pub_size

```
int wc_ed25519_pub_size(
    ed25519_key * key
)
```

This function returns the compressed key size in bytes (public key).

Parameters:

- **key** Pointer to an ed25519_key structure for which to get the key size.

See: wc_ed25519_priv_size

Return:

- ED25519_PUB_KEY_SIZE The size of the compressed public key (32 bytes).
- BAD_FUNC_ARG Returns if key argument is NULL.

Example

```
ed25519_key key;
wc_ed25519_init(&key);
WC_RNG rng;
wc_InitRng(&rng);

wc_ed25519_make_key(&rng, 32, &key); // initialize 32 byte Ed25519 key
int key_size = wc_ed25519_pub_size(&key);
```

C.23.2.26 function wc_ed25519_sig_size

```
int wc_ed25519_sig_size(
    ed25519_key * key
)
```

This function returns the size of an Ed25519 signature (64 in bytes).

Parameters:

- **key** Pointer to an ed25519_key structure for which to get the signature size.

See: wc_ed25519_sign_msg

Return:

- ED25519_SIG_SIZE The size of an Ed25519 signature (64 bytes).
- BAD_FUNC_ARG Returns if key argument is NULL.

Example

```
int sigSz;
ed25519_key key;
// initialize key, make key

sigSz = wc_ed25519_sig_size(&key);
if (sigSz == 0) {
    // error determining sig size
}
```

C.23.3 Source code

```
int wc_ed25519_make_public(ed25519_key* key, unsigned char* pubKey,
                           word32 pubKeySz);

int wc_ed25519_make_key(WC_RNG* rng, int keysize, ed25519_key* key);

int wc_ed25519_sign_msg(const byte* in, word32 inlen, byte* out,
                        word32 *outlen, ed25519_key* key);

int wc_ed25519ctx_sign_msg(const byte* in, word32 inlen, byte* out,
                        word32 *outlen, ed25519_key* key,
                        const byte* context, byte contextLen);

int wc_ed25519ph_sign_hash(const byte* hash, word32 hashLen, byte* out,
                           word32 *outLen, ed25519_key* key,
                           const byte* context, byte contextLen);

int wc_ed25519ph_sign_msg(const byte* in, word32 inlen, byte* out,
                        word32 *outlen, ed25519_key* key,
                        const byte* context, byte contextLen);

int wc_ed25519_verify_msg(const byte* sig, word32 siglen, const byte* msg,
                          word32 msgLen, int* ret, ed25519_key* key);

int wc_ed25519ctx_verify_msg(const byte* sig, word32 siglen, const byte* msg,
                             word32 msgLen, int* ret, ed25519_key* key,
                             const byte* context, byte contextLen);

int wc_ed25519ph_verify_hash(const byte* sig, word32 siglen, const byte* hash,
                             word32 hashLen, int* ret, ed25519_key* key,
                             const byte* context, byte contextLen);

int wc_ed25519ph_verify_msg(const byte* sig, word32 siglen, const byte* msg,
                            word32 msgLen, int* ret, ed25519_key* key,
                            const byte* context, byte contextLen);

int wc_ed25519_init(ed25519_key* key);

void wc_ed25519_free(ed25519_key* key);

int wc_ed25519_import_public(const byte* in, word32 inLen, ed25519_key* key);

int wc_ed25519_import_public_ex(const byte* in, word32 inLen, ed25519_key* key,
    int trusted);

int wc_ed25519_import_private_only(const byte* priv, word32 privSz,
                                   ed25519_key* key);

int wc_ed25519_import_private_key(const byte* priv, word32 privSz,
                              const byte* pub, word32 pubSz, ed25519_key* key);

int wc_ed25519_import_private_key_ex(const byte* priv, word32 privSz,
```

```
    const byte* pub, word32 pubSz, ed25519_key* key, int trusted);

int wc_ed25519_export_public(ed25519_key* key, byte* out, word32* outLen);

int wc_ed25519_export_private_only(ed25519_key* key, byte* out, word32*
 ↪  outLen);

int wc_ed25519_export_private(ed25519_key* key, byte* out, word32* outLen);

int wc_ed25519_export_key(ed25519_key* key,
                          byte* priv, word32 *privSz,
                          byte* pub, word32 *pubSz);

int wc_ed25519_check_key(ed25519_key* key);

int wc_ed25519_size(ed25519_key* key);

int wc_ed25519_priv_size(ed25519_key* key);

int wc_ed25519_pub_size(ed25519_key* key);

int wc_ed25519_sig_size(ed25519_key* key);
```

C.24 dox_comments/header_files/ed448.h

C.24.1 Functions

	Name
int	**wc_ed448_make_public**(ed448_key * key, unsigned char * pubKey, word32 pubKeySz)This function generates the Ed448 public key from the private key. It stores the public key in the buffer pubKey, and sets the bytes written to this buffer in pubKeySz.
int	**wc_ed448_make_key**(WC_RNG * rng, int keysize, ed448_key * key)This function generates a new Ed448 key and stores it in key.
int	**wc_ed448_sign_msg**(const byte * in, word32 inlen, byte * out, word32 * outlen, ed448_key * key)This function signs a message using an ed448_key object to guarantee authenticity.
int	**wc_ed448ph_sign_hash**(const byte * hash, word32 hashLen, byte * out, word32 * outLen, ed448_key * key, const byte * context, byte contextLen)This function signs a message digest using an ed448_key object to guarantee authenticity. The context is included as part of the data signed. The hash is the pre-hashed message before signature calculation.

	Name
int	**wc_ed448ph_sign_msg**(const byte * in, word32 inLen, byte * out, word32 * outLen, ed448_key * key, const byte * context, byte contextLen)This function signs a message using an ed448_key object to guarantee authenticity. The context is included as part of the data signed. The message is pre-hashed before signature calculation.
int	**wc_ed448_verify_msg**(const byte * sig, word32 siglen, const byte * msg, word32 msgLen, int * res, ed448_key * key, const byte * context, byte contextLen)This function verifies the Ed448 signature of a message to ensure authenticity. The context is included as part of the data verified. The answer is returned through res, with 1 corresponding to a valid signature, and 0 corresponding to an invalid signature.
int	**wc_ed448ph_verify_hash**(const byte * sig, word32 siglen, const byte * hash, word32 hashlen, int * res, ed448_key * key, const byte * context, byte contextLen)This function verifies the Ed448 signature of the digest of a message to ensure authenticity. The context is included as part of the data verified. The hash is the pre-hashed message before signature calculation. The answer is returned through res, with 1 corresponding to a valid signature, and 0 corresponding to an invalid signature.
int	**wc_ed448ph_verify_msg**(const byte * sig, word32 siglen, const byte * msg, word32 msgLen, int * res, ed448_key * key, const byte * context, byte contextLen)This function verifies the Ed448 signature of a message to ensure authenticity. The context is included as part of the data verified. The message is pre-hashed before verification. The answer is returned through res, with 1 corresponding to a valid signature, and 0 corresponding to an invalid signature.
int	**wc_ed448_init**(ed448_key * key)This function initializes an ed448_key object for future use with message verification.
void	**wc_ed448_free**(ed448_key * key)This function frees an Ed448 object after it has been used.
int	**wc_ed448_import_public**(const byte * in, word32 inLen, ed448_key * key)This function imports a public ed448_key pair from a buffer containing the public key. This function will handle both compressed and uncompressed keys. The public key is checked that it matches the private key when one is present.

	Name
int	**wc_ed448_import_public_ex**(const byte * in, word32 inLen, ed448_key * key, int trusted)This function imports a public ed448_key pair from a buffer containing the public key. This function will handle both compressed and uncompressed keys. Check public key matches private key, when present, when not trusted.
int	**wc_ed448_import_private_only**(const byte * priv, word32 privSz, ed448_key * key)This function imports an Ed448 private key only from a buffer.
int	**wc_ed448_import_private_key**(const byte * priv, word32 privSz, const byte * pub, word32 pubSz, ed448_key * key)This function imports a public/private Ed448 key pair from a pair of buffers. This function will handle both compressed and uncompressed keys.
int	**wc_ed448_import_private_key_ex**(const byte * priv, word32 privSz, const byte * pub, word32 pubSz, ed448_key * key, int trusted)This function imports a public/private Ed448 key pair from a pair of buffers. This function will handle both compressed and uncompressed keys. The public is checked against private key if not trusted.
int	**wc_ed448_export_public**(ed448_key * key, byte * out, word32 * outLen)This function exports the private key from an ed448_key structure. It stores the public key in the buffer out, and sets the bytes written to this buffer in outLen.
int	**wc_ed448_export_private_only**(ed448_key * key, byte * out, word32 * outLen)This function exports only the private key from an ed448_key structure. It stores the private key in the buffer out, and sets the bytes written to this buffer in outLen.
int	**wc_ed448_export_private**(ed448_key * key, byte * out, word32 * outLen)This function exports the key pair from an ed448_key structure. It stores the key pair in the buffer out, and sets the bytes written to this buffer in outLen.
int	**wc_ed448_export_key**(ed448_key * key, byte * priv, word32 * privSz, byte * pub, word32 * pubSz)This function exports the private and public key separately from an ed448_key structure. It stores the private key in the buffer priv, and sets the bytes written to this buffer in privSz. It stores the public key in the buffer pub, and sets the bytes written to this buffer in pubSz.

	Name
int	**wc_ed448_check_key**(ed448_key * key)This function checks the public key in ed448_key structure matches the private key.
int	**wc_ed448_size**(ed448_key * key)This function returns the size of an Ed448 private key - 57 bytes.
int	**wc_ed448_priv_size**(ed448_key * key)This function returns the private key size (secret + public) in bytes.
int	**wc_ed448_pub_size**(ed448_key * key)This function returns the compressed key size in bytes (public key).
int	**wc_ed448_sig_size**(ed448_key * key)This function returns the size of an Ed448 signature (114 in bytes).

C.24.2 Functions Documentation

C.24.2.1 function wc_ed448_make_public

```
int wc_ed448_make_public(
    ed448_key * key,
    unsigned char * pubKey,
    word32 pubKeySz
)
```

This function generates the Ed448 public key from the private key. It stores the public key in the buffer pubKey, and sets the bytes written to this buffer in pubKeySz.

Parameters:

- **key** Pointer to the ed448_key for which to generate a key.
- **out** Pointer to the buffer in which to store the public key.
- **outLen** Pointer to a word32 object with the size available in out. Set with the number of bytes written to out after successfully exporting the public key.

See:

- wc_ed448_init
- wc_ed448_import_private_only
- wc_ed448_make_key

Return:

- 0 Returned upon successfully making the public key.
- BAD_FUNC_ARG Returned ifi key or pubKey evaluate to NULL, or if the specified key size is not 57 bytes (Ed448 has 57 byte keys).
- MEMORY_E Returned if there is an error allocating memory during function execution.

Example

```
int ret;

ed448_key key;
byte priv[] = { initialize with 57 byte private key };
byte pub[57];
word32 pubSz = sizeof(pub);
```

```
wc_ed448_init(&key);
wc_ed448_import_private_only(priv, sizeof(priv), &key);
ret = wc_ed448_make_public(&key, pub, &pubSz);
if (ret != 0) {
    // error making public key
}
```

C.24.2.2 function wc_ed448_make_key

```
int wc_ed448_make_key(
    WC_RNG * rng,
    int keysize,
    ed448_key * key
)
```

This function generates a new Ed448 key and stores it in key.

Parameters:

- **rng** Pointer to an initialized RNG object with which to generate the key.
- **keysize** Length of key to generate. Should always be 57 for Ed448.
- **key** Pointer to the ed448_key for which to generate a key.

See: wc_ed448_init

Return:

- 0 Returned upon successfully making an ed448_key.
- BAD_FUNC_ARG Returned if rng or key evaluate to NULL, or if the specified key size is not 57 bytes (Ed448 has 57 byte keys).
- MEMORY_E Returned if there is an error allocating memory during function execution.

Example

```
int ret;

WC_RNG rng;
ed448_key key;

wc_InitRng(&rng);
wc_ed448_init(&key);
ret = wc_ed448_make_key(&rng, 57, &key);
if (ret != 0) {
    // error making key
}
```

C.24.2.3 function wc_ed448_sign_msg

```
int wc_ed448_sign_msg(
    const byte * in,
    word32 inlen,
    byte * out,
    word32 * outlen,
    ed448_key * key
)
```

This function signs a message using an ed448_key object to guarantee authenticity.

Parameters:

- **in** Pointer to the buffer containing the message to sign.
- **inlen** Length of the message to sign.
- **out** Buffer in which to store the generated signature.
- **outlen** Maximum length of the output buffer. Will store the bytes written to out upon successfully generating a message signature.
- **key** Pointer to a private ed448_key with which to generate the signature.

See:

- wc_ed448ph_sign_hash
- wc_ed448ph_sign_msg
- wc_ed448_verify_msg

Return:

- 0 Returned upon successfully generating a signature for the message.
- BAD_FUNC_ARG Returned if any of the input parameters evaluate to NULL, or if the output buffer is too small to store the generated signature.
- MEMORY_E Returned if there is an error allocating memory during function execution.

Example

```
ed448_key key;
WC_RNG rng;
int ret, sigSz;

byte sig[114]; // will hold generated signature
sigSz = sizeof(sig);
byte message[] = { initialize with message };

wc_InitRng(&rng); // initialize rng
wc_ed448_init(&key); // initialize key
wc_ed448_make_key(&rng, 57, &key); // make public/private key pair
ret = wc_ed448_sign_msg(message, sizeof(message), sig, &sigSz, &key);
if (ret != 0 ) {
    // error generating message signature
}
```

C.24.2.4 function wc_ed448ph_sign_hash

```
int wc_ed448ph_sign_hash(
    const byte * hash,
    word32 hashLen,
    byte * out,
    word32 * outLen,
    ed448_key * key,
    const byte * context,
    byte contextLen
)
```

This function signs a message digest using an ed448_key object to guarantee authenticity. The context is included as part of the data signed. The hash is the pre-hashed message before signature calculation.

Parameters:

- **hash** Pointer to the buffer containing the hash of the message to sign.

- **hashLen** Length of the hash of the message to sign.
- **out** Buffer in which to store the generated signature.
- **outlen** Maximum length of the output buffer. Will store the bytes written to out upon successfully generating a message signature.
- **key** Pointer to a private ed448_key with which to generate the signature.
- **context** Pointer to the buffer containing the context for which message is being signed.
- **contextLen** Length of the context buffer.

See:

- wc_ed448_sign_msg
- wc_ed448ph_sign_msg
- wc_ed448ph_verify_hash

Return:

- 0 Returned upon successfully generating a signature for the message digest.
- BAD_FUNC_ARG Returned any of the input parameters evaluate to NULL, or if the output buffer is too small to store the generated signature.
- MEMORY_E Returned if there is an error allocating memory during function execution.

Example

```
ed448_key key;
WC_RNG rng;
int ret, sigSz;

byte sig[114]; // will hold generated signature
sigSz = sizeof(sig);
byte hash[] = { initialize hash of message };
byte context[] = { initialize with context of signing };

wc_InitRng(&rng); // initialize rng
wc_ed448_init(&key); // initialize key
wc_ed448_make_key(&rng, 57, &key); // make public/private key pair
ret = wc_ed448ph_sign_hash(hash, sizeof(hash), sig, &sigSz, &key,
        context, sizeof(context));
if (ret != 0) {
    // error generating message signature
}
```

C.24.2.5 function wc_ed448ph_sign_msg

```
int wc_ed448ph_sign_msg(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 * outLen,
    ed448_key * key,
    const byte * context,
    byte contextLen
)
```

This function signs a message using an ed448_key object to guarantee authenticity. The context is included as part of the data signed. The message is pre-hashed before signature calculation.

Parameters:

- **in** Pointer to the buffer containing the message to sign.

- **inlen** Length of the message to sign.
- **out** Buffer in which to store the generated signature.
- **outlen** Maximum length of the output buffer. Will store the bytes written to out upon successfully generating a message signature.
- **key** Pointer to a private ed448_key with which to generate the signature.
- **context** Pointer to the buffer containing the context for which message is being signed.
- **contextLen** Length of the context buffer.

See:

- wc_ed448_sign_msg
- wc_ed448ph_sign_hash
- wc_ed448ph_verify_msg

Return:

- 0 Returned upon successfully generating a signature for the message.
- BAD_FUNC_ARG Returned any of the input parameters evaluate to NULL, or if the output buffer is too small to store the generated signature.
- MEMORY_E Returned if there is an error allocating memory during function execution.

Example

```
ed448_key key;
WC_RNG rng;
int ret, sigSz;

byte sig[114]; // will hold generated signature
sigSz = sizeof(sig);
byte message[] = { initialize with message };
byte context[] = { initialize with context of signing };

wc_InitRng(&rng); // initialize rng
wc_ed448_init(&key); // initialize key
wc_ed448_make_key(&rng, 57, &key); // make public/private key pair
ret = wc_ed448ph_sign_msg(message, sizeof(message), sig, &sigSz, &key,
        context, sizeof(context));
if (ret != 0) {
    // error generating message signature
}
```

C.24.2.6 function wc_ed448_verify_msg

```
int wc_ed448_verify_msg(
    const byte * sig,
    word32 siglen,
    const byte * msg,
    word32 msgLen,
    int * res,
    ed448_key * key,
    const byte * context,
    byte contextLen
)
```

This function verifies the Ed448 signature of a message to ensure authenticity. The context is included as part of the data verified. The answer is returned through res, with 1 corresponding to a valid signature, and 0 corresponding to an invalid signature.

Parameters:

- **sig** Pointer to the buffer containing the signature to verify.
- **siglen** Length of the signature to verify.
- **msg** Pointer to the buffer containing the message to verify.
- **msgLen** Length of the message to verify.
- **key** Pointer to a public Ed448 key with which to verify the signature.
- **context** Pointer to the buffer containing the context for which the message was signed.
- **contextLen** Length of the context buffer.

See:

- wc_ed448ph_verify_hash
- wc_ed448ph_verify_msg
- wc_ed448_sign_msg

Return:

- 0 Returned upon successfully performing the signature verification and authentication.
- BAD_FUNC_ARG Returned if any of the input parameters evaluate to NULL, or if the siglen does not match the actual length of a signature.
- SIG_VERIFY_E Returned if verification completes, but the signature generated does not match the signature provided.

Example

```
ed448_key key;
int ret, verified = 0;

byte sig[] { initialize with received signature };
byte msg[] = { initialize with message };
byte context[] = { initialize with context of signature };
// initialize key with received public key
ret = wc_ed448_verify_msg(sig, sizeof(sig), msg, sizeof(msg), &verified,
        &key, context, sizeof(context));
if (ret < 0) {
    // error performing verification
} else if (verified == 0)
    // the signature is invalid
}
```

C.24.2.7 function wc_ed448ph_verify_hash

```
int wc_ed448ph_verify_hash(
    const byte * sig,
    word32 siglen,
    const byte * hash,
    word32 hashlen,
    int * res,
    ed448_key * key,
    const byte * context,
    byte contextLen
)
```

This function verifies the Ed448 signature of the digest of a message to ensure authenticity. The context is included as part of the data verified. The hash is the pre-hashed message before signature calculation. The answer is returned through res, with 1 corresponding to a valid signature, and 0 corresponding to an invalid signature.

Parameters:

- **sig** Pointer to the buffer containing the signature to verify.
- **siglen** Length of the signature to verify.
- **hash** Pointer to the buffer containing the hash of the message to verify.
- **hashLen** Length of the hash to verify.
- **key** Pointer to a public Ed448 key with which to verify the signature.
- **context** Pointer to the buffer containing the context for which the message was signed.
- **contextLen** Length of the context buffer.

See:

- wc_ed448_verify_msg
- wc_ed448ph_verify_msg
- wc_ed448ph_sign_hash

Return:

- 0 Returned upon successfully performing the signature verification and authentication.
- BAD_FUNC_ARG Returned if any of the input parameters evaluate to NULL, or if the siglen does not match the actual length of a signature.
- SIG_VERIFY_E Returned if verification completes, but the signature generated does not match the signature provided.

Example

```
ed448_key key;
int ret, verified = 0;

byte sig[] { initialize with received signature };
byte hash[] = { initialize hash of message };
byte context[] = { initialize with context of signature };
// initialize key with received public key
ret = wc_ed448ph_verify_hash(sig, sizeof(sig), hash, sizeof(hash),
        &verified, &key, context, sizeof(context));
if (ret < 0) {
    // error performing verification
} else if (verified == 0)
    // the signature is invalid
}
```

C.24.2.8 function wc_ed448ph_verify_msg

```
int wc_ed448ph_verify_msg(
    const byte * sig,
    word32 siglen,
    const byte * msg,
    word32 msgLen,
    int * res,
    ed448_key * key,
    const byte * context,
    byte contextLen
)
```

This function verifies the Ed448 signature of a message to ensure authenticity. The context is included as part of the data verified. The message is pre-hashed before verification. The answer is returned through res, with 1 corresponding to a valid signature, and 0 corresponding to an invalid signature.

Parameters:

- **sig** Pointer to the buffer containing the signature to verify.
- **siglen** Length of the signature to verify.
- **msg** Pointer to the buffer containing the message to verify.
- **msgLen** Length of the message to verify.
- **key** Pointer to a public Ed448 key with which to verify the signature.
- **context** Pointer to the buffer containing the context for which the message was signed.
- **contextLen** Length of the context buffer.

See:

- wc_ed448_verify_msg
- wc_ed448ph_verify_hash
- wc_ed448ph_sign_msg

Return:

- 0 Returned upon successfully performing the signature verification and authentication.
- BAD_FUNC_ARG Returned if any of the input parameters evaluate to NULL, or if the siglen does not match the actual length of a signature.
- SIG_VERIFY_E Returned if verification completes, but the signature generated does not match the signature provided.

Example

```
ed448_key key;
int ret, verified = 0;

byte sig[] { initialize with received signature };
byte msg[] = { initialize with message };
byte context[] = { initialize with context of signature };
// initialize key with received public key
ret = wc_ed448ph_verify_msg(sig, sizeof(sig), msg, sizeof(msg), &verified,
        &key, context, sizeof(context));
if (ret < 0) {
    // error performing verification
} else if (verified == 0)
    // the signature is invalid
}
```

C.24.2.9 function wc_ed448_init

```
int wc_ed448_init(
    ed448_key * key
)
```

This function initializes an ed448_key object for future use with message verification.

Parameters:

- **key** Pointer to the ed448_key object to initialize.

See:

- wc_ed448_make_key
- wc_ed448_free

Return:

- 0 Returned upon successfully initializing the ed448_key object.
- BAD_FUNC_ARG Returned if key is NULL.

Example

```
ed448_key key;
wc_ed448_init(&key);
```

C.24.2.10 function wc_ed448_free

```
void wc_ed448_free(
    ed448_key * key
)
```

This function frees an Ed448 object after it has been used.

Parameters:

- **key** Pointer to the ed448_key object to free

See: wc_ed448_init

Example

```
ed448_key key;
// initialize key and perform secure exchanges
...
wc_ed448_free(&key);
```

C.24.2.11 function wc_ed448_import_public

```
int wc_ed448_import_public(
    const byte * in,
    word32 inLen,
    ed448_key * key
)
```

This function imports a public ed448_key pair from a buffer containing the public key. This function will handle both compressed and uncompressed keys. The public key is checked that it matches the private key when one is present.

Parameters:

- **in** Pointer to the buffer containing the public key.
- **inLen** Length of the buffer containing the public key.
- **key** Pointer to the ed448_key object in which to store the public key.

See:

- wc_ed448_import_public_ex
- wc_ed448_import_private_key
- wc_ed448_import_private_key_ex
- wc_ed448_export_public

Return:

- 0 Returned on successfully importing the ed448_key.
- BAD_FUNC_ARG Returned if in or key evaluate to NULL, or inLen is less than the size of an Ed448 key.

Example

```
int ret;
byte pub[] = { initialize Ed448 public key };
```

```
ed_448 key;
wc_ed448_init_key(&key);
ret = wc_ed448_import_public(pub, sizeof(pub), &key);
if (ret != 0) {
    // error importing key
}
```

C.24.2.12 function wc_ed448_import_public_ex

```
int wc_ed448_import_public_ex(
    const byte * in,
    word32 inLen,
    ed448_key * key,
    int trusted
)
```

This function imports a public ed448_key pair from a buffer containing the public key. This function will handle both compressed and uncompressed keys. Check public key matches private key, when present, when not trusted.

Parameters:

- **in** Pointer to the buffer containing the public key.
- **inLen** Length of the buffer containing the public key.
- **key** Pointer to the ed448_key object in which to store the public key.
- **trusted** Public key data is trusted or not.

See:

- wc_ed448_import_public
- wc_ed448_import_private_key
- wc_ed448_import_private_key_ex
- wc_ed448_export_public

Return:

- 0 Returned on successfully importing the ed448_key.
- BAD_FUNC_ARG Returned if in or key evaluate to NULL, or inLen is less than the size of an Ed448 key.

Example

```
int ret;
byte pub[] = { initialize Ed448 public key };

ed_448 key;
wc_ed448_init_key(&key);
ret = wc_ed448_import_public_ex(pub, sizeof(pub), &key, 1);
if (ret != 0) {
    // error importing key
}
```

C.24.2.13 function wc_ed448_import_private_only

```
int wc_ed448_import_private_only(
    const byte * priv,
    word32 privSz,
    ed448_key * key
)
```

This function imports an Ed448 private key only from a buffer.

Parameters:

- **priv** Pointer to the buffer containing the private key.
- **privSz** Length of the private key.
- **key** Pointer to the ed448_key object in which to store the imported private key.

See:

- wc_ed448_import_public
- wc_ed448_import_public_ex
- wc_ed448_import_private_key
- wc_ed448_import_private_key_ex
- wc_ed448_export_private_only

Return:

- 0 Returned on successfully importing the Ed448 private key.
- BAD_FUNC_ARG Returned if in or key evaluate to NULL, or if privSz is less than ED448_KEY_SIZE.

Example

```
int ret;
byte priv[] = { initialize with 57 byte private key };

ed448_key key;
wc_ed448_init_key(&key);
ret = wc_ed448_import_private_only(priv, sizeof(priv), &key);
if (ret != 0) {
    // error importing private key
}
```

C.24.2.14 function wc_ed448_import_private_key

```
int wc_ed448_import_private_key(
    const byte * priv,
    word32 privSz,
    const byte * pub,
    word32 pubSz,
    ed448_key * key
)
```

This function imports a public/private Ed448 key pair from a pair of buffers. This function will handle both compressed and uncompressed keys.

Parameters:

- **priv** Pointer to the buffer containing the private key.
- **privSz** Length of the private key.
- **pub** Pointer to the buffer containing the public key.
- **pubSz** Length of the public key.
- **key** Pointer to the ed448_key object in which to store the imported private/public key pair.

See:

- wc_ed448_import_public
- wc_ed448_import_public_ex
- wc_ed448_import_private_only
- wc_ed448_import_private_key_ex
- wc_ed448_export_private

Return:

- 0 Returned on successfully importing the Ed448 key.
- BAD_FUNC_ARG Returned if in or key evaluate to NULL, or if either privSz is less than ED448_KEY_SIZE or pubSz is less than ED448_PUB_KEY_SIZE.

Example

```
int ret;
byte priv[] = { initialize with 57 byte private key };
byte pub[]  = { initialize with the corresponding public key };

ed448_key key;
wc_ed448_init_key(&key);
ret = wc_ed448_import_private_key(priv, sizeof(priv), pub, sizeof(pub),
        &key);
if (ret != 0) {
    // error importing key
}
```

C.24.2.15 function wc_ed448_import_private_key_ex

```
int wc_ed448_import_private_key_ex(
    const byte * priv,
    word32 privSz,
    const byte * pub,
    word32 pubSz,
    ed448_key * key,
    int trusted
)
```

This function imports a public/private Ed448 key pair from a pair of buffers. This function will handle both compressed and uncompressed keys. The public is checked against private key if not trusted.

Parameters:

- **priv** Pointer to the buffer containing the private key.
- **privSz** Length of the private key.
- **pub** Pointer to the buffer containing the public key.
- **pubSz** Length of the public key.
- **key** Pointer to the ed448_key object in which to store the imported private/public key pair.
- **trusted** Public key data is trusted or not.

See:

- wc_ed448_import_public
- wc_ed448_import_public_ex
- wc_ed448_import_private_only
- wc_ed448_import_private_key
- wc_ed448_export_private

Return:

- 0 Returned on successfully importing the Ed448 key.
- BAD_FUNC_ARG Returned if in or key evaluate to NULL, or if either privSz is less than ED448_KEY_SIZE or pubSz is less than ED448_PUB_KEY_SIZE.

Example

```
int ret;
byte priv[] = { initialize with 57 byte private key };
byte pub[]  = { initialize with the corresponding public key };

ed448_key key;
wc_ed448_init_key(&key);
ret = wc_ed448_import_private_key_ex(priv, sizeof(priv), pub, sizeof(pub),
        &key, 1);
if (ret != 0) {
    // error importing key
}
```

C.24.2.16 function wc_ed448_export_public

```
int wc_ed448_export_public(
    ed448_key * key,
    byte * out,
    word32 * outLen
)
```

This function exports the private key from an ed448_key structure. It stores the public key in the buffer out, and sets the bytes written to this buffer in outLen.

Parameters:

- **key** Pointer to an ed448_key structure from which to export the public key.
- **out** Pointer to the buffer in which to store the public key.
- **outLen** Pointer to a word32 object with the size available in out. Set with the number of bytes written to out after successfully exporting the public key.

See:

- wc_ed448_import_public
- wc_ed448_import_public_ex
- wc_ed448_export_private_only

Return:

- 0 Returned upon successfully exporting the public key.
- BAD_FUNC_ARG Returned if any of the input values evaluate to NULL.
- BUFFER_E Returned if the buffer provided is not large enough to store the private key. Upon returning this error, the function sets the size required in outLen.

Example

```
int ret;
ed448_key key;
// initialize key, make key

char pub[57];
word32 pubSz = sizeof(pub);

ret = wc_ed448_export_public(&key, pub, &pubSz);
if (ret != 0) {
    // error exporting public key
}
```

C.24.2.17 function wc_ed448_export_private_only

```
int wc_ed448_export_private_only(
    ed448_key * key,
    byte * out,
    word32 * outLen
)
```

This function exports only the private key from an ed448_key structure. It stores the private key in the buffer out, and sets the bytes written to this buffer in outLen.

Parameters:

- **key** Pointer to an ed448_key structure from which to export the private key.
- **out** Pointer to the buffer in which to store the private key.
- **outLen** Pointer to a word32 object with the size available in out. Set with the number of bytes written to out after successfully exporting the private key.

See:

- wc_ed448_export_public
- wc_ed448_import_private_key
- wc_ed448_import_private_key_ex

Return:

- 0 Returned upon successfully exporting the private key.
- ECC_BAD_ARG_E Returned if any of the input values evaluate to NULL.
- BUFFER_E Returned if the buffer provided is not large enough to store the private key.

Example

```
int ret;
ed448_key key;
// initialize key, make key

char priv[57]; // 57 bytes because only private key
word32 privSz = sizeof(priv);
ret = wc_ed448_export_private_only(&key, priv, &privSz);
if (ret != 0) {
    // error exporting private key
}
```

C.24.2.18 function wc_ed448_export_private

```
int wc_ed448_export_private(
    ed448_key * key,
    byte * out,
    word32 * outLen
)
```

This function exports the key pair from an ed448_key structure. It stores the key pair in the buffer out, and sets the bytes written to this buffer in outLen.

Parameters:

- **key** Pointer to an ed448_key structure from which to export the key pair.
- **out** Pointer to the buffer in which to store the key pair.
- **outLen** Pointer to a word32 object with the size available in out. Set with the number of bytes written to out after successfully exporting the key pair.

See:

- wc_ed448_import_private
- wc_ed448_export_private_only

Return:

- 0 Returned upon successfully exporting the key pair.
- ECC_BAD_ARG_E Returned if any of the input values evaluate to NULL.
- BUFFER_E Returned if the buffer provided is not large enough to store the key pair.

Example

```
ed448_key key;
wc_ed448_init(&key);

WC_RNG rng;
wc_InitRng(&rng);

wc_ed448_make_key(&rng, 57, &key); // initialize 57 byte Ed448 key

byte out[114]; // out needs to be a sufficient buffer size
word32 outLen = sizeof(out);
int key_size = wc_ed448_export_private(&key, out, &outLen);
if (key_size == BUFFER_E) {
    // Check size of out compared to outLen to see if function reset outLen
}
```

C.24.2.19 function wc_ed448_export_key

```
int wc_ed448_export_key(
    ed448_key * key,
    byte * priv,
    word32 * privSz,
    byte * pub,
    word32 * pubSz
)
```

This function exports the private and public key separately from an ed448_key structure. It stores the private key in the buffer priv, and sets the bytes written to this buffer in privSz. It stores the public key in the buffer pub, and sets the bytes written to this buffer in pubSz.

Parameters:

- **key** Pointer to an ed448_key structure from which to export the key pair.
- **priv** Pointer to the buffer in which to store the private key.
- **privSz** Pointer to a word32 object with the size available in out. Set with the number of bytes written to out after successfully exporting the private key.
- **pub** Pointer to the buffer in which to store the public key.
- **pubSz** Pointer to a word32 object with the size available in out. Set with the number of bytes written to out after successfully exporting the public key.

See:

- wc_ed448_export_private
- wc_ed448_export_public

Return:

- 0 Returned upon successfully exporting the key pair.

- ECC_BAD_ARG_E Returned if any of the input values evaluate to NULL.
- BUFFER_E Returned if the buffer provided is not large enough to store the key pair.

Example

```
int ret;
ed448_key key;
// initialize key, make key

char pub[57];
word32 pubSz = sizeof(pub);
char priv[57];
word32 privSz = sizeof(priv);

ret = wc_ed448_export_key(&key, priv, &pubSz, pub, &pubSz);
if (ret != 0) {
    // error exporting private and public key
}
```

C.24.2.20 function wc_ed448_check_key

```
int wc_ed448_check_key(
    ed448_key * key
)
```

This function checks the public key in ed448_key structure matches the private key.

Parameters:

- **key** Pointer to an ed448_key structure holding a private and public key.

See:

- wc_ed448_import_private_key
- wc_ed448_import_private_key_ex

Return:

- 0 Returned if the private and public key matched.
- BAD_FUNC_ARGS Returned if the given key is NULL.

Example

```
int ret;
byte priv[] = { initialize with 57 byte private key };
byte pub[]  = { initialize with the corresponding public key };

ed448_key key;
wc_ed448_init_key(&key);
wc_ed448_import_private_key_ex(priv, sizeof(priv), pub, sizeof(pub), &key,
    1);
ret = wc_ed448_check_key(&key);
if (ret != 0) {
    // error checking key
}
```

C.24.2.21 function wc_ed448_size

```
int wc_ed448_size(
    ed448_key * key
)
```

This function returns the size of an Ed448 private key - 57 bytes.

Parameters:

- **key** Pointer to an ed448_key structure for which to get the key size.

See: wc_ed448_make_key

Return:

- ED448_KEY_SIZE The size of a valid private key (57 bytes).
- BAD_FUNC_ARGS Returned if the given key is NULL.

Example

```
int keySz;
ed448_key key;
// initialize key, make key
keySz = wc_ed448_size(&key);
if (keySz == 0) {
    // error determining key size
}
```

C.24.2.22 function wc_ed448_priv_size

```
int wc_ed448_priv_size(
    ed448_key * key
)
```

This function returns the private key size (secret + public) in bytes.

Parameters:

- **key** Pointer to an ed448_key structure for which to get the key size.

See: wc_ed448_pub_size

Return:

- ED448_PRV_KEY_SIZE The size of the private key (114 bytes).
- BAD_FUNC_ARG Returns if key argument is NULL.

Example

```
ed448_key key;
wc_ed448_init(&key);

WC_RNG rng;
wc_InitRng(&rng);

wc_ed448_make_key(&rng, 57, &key); // initialize 57 byte Ed448 key
int key_size = wc_ed448_priv_size(&key);
```

C.24.2.23 function wc_ed448_pub_size

```
int wc_ed448_pub_size(
    ed448_key * key
)
```

This function returns the compressed key size in bytes (public key).

Parameters:

- **key** Pointer to an ed448_key structure for which to get the key size.

See: wc_ed448_priv_size

Return:

- ED448_PUB_KEY_SIZE The size of the compressed public key (57 bytes).
- BAD_FUNC_ARG Returns if key argument is NULL.

Example

```
ed448_key key;
wc_ed448_init(&key);
WC_RNG rng;
wc_InitRng(&rng);

wc_ed448_make_key(&rng, 57, &key); // initialize 57 byte Ed448 key
int key_size = wc_ed448_pub_size(&key);
```

C.24.2.24 function wc_ed448_sig_size

```
int wc_ed448_sig_size(
    ed448_key * key
)
```

This function returns the size of an Ed448 signature (114 in bytes).

Parameters:

- **key** Pointer to an ed448_key structure for which to get the signature size.

See: wc_ed448_sign_msg

Return:

- ED448_SIG_SIZE The size of an Ed448 signature (114 bytes).
- BAD_FUNC_ARG Returns if key argument is NULL.

Example

```
int sigSz;
ed448_key key;
// initialize key, make key

sigSz = wc_ed448_sig_size(&key);
if (sigSz == 0) {
    // error determining sig size
}
```

C.24.3 Source code

```
int wc_ed448_make_public(ed448_key* key, unsigned char* pubKey,
                         word32 pubKeySz);

int wc_ed448_make_key(WC_RNG* rng, int keysize, ed448_key* key);

int wc_ed448_sign_msg(const byte* in, word32 inlen, byte* out,
```

```
                        word32 *outlen, ed448_key* key);

int wc_ed448ph_sign_hash(const byte* hash, word32 hashLen, byte* out,
                         word32 *outLen, ed448_key* key,
                         const byte* context, byte contextLen);

int wc_ed448ph_sign_msg(const byte* in, word32 inLen, byte* out,
                        word32 *outLen, ed448_key* key, const byte* context,
                        byte contextLen);

int wc_ed448_verify_msg(const byte* sig, word32 siglen, const byte* msg,
                         word32 msgLen, int* res, ed448_key* key,
                         const byte* context, byte contextLen);

int wc_ed448ph_verify_hash(const byte* sig, word32 siglen, const byte* hash,
                         word32 hashlen, int* res, ed448_key* key,
                         const byte* context, byte contextLen);

int wc_ed448ph_verify_msg(const byte* sig, word32 siglen, const byte* msg,
                         word32 msgLen, int* res, ed448_key* key,
                         const byte* context, byte contextLen);

int wc_ed448_init(ed448_key* key);

void wc_ed448_free(ed448_key* key);

int wc_ed448_import_public(const byte* in, word32 inLen, ed448_key* key);

int wc_ed448_import_public_ex(const byte* in, word32 inLen, ed448_key* key,
    int trusted);

int wc_ed448_import_private_only(const byte* priv, word32 privSz,
                                 ed448_key* key);

int wc_ed448_import_private_key(const byte* priv, word32 privSz,
                                const byte* pub, word32 pubSz, ed448_key* key);

int wc_ed448_import_private_key_ex(const byte* priv, word32 privSz,
    const byte* pub, word32 pubSz, ed448_key* key, int trusted);

int wc_ed448_export_public(ed448_key* key, byte* out, word32* outLen);

int wc_ed448_export_private_only(ed448_key* key, byte* out, word32* outLen);

int wc_ed448_export_private(ed448_key* key, byte* out, word32* outLen);

int wc_ed448_export_key(ed448_key* key,
                         byte* priv, word32 *privSz,
                         byte* pub, word32 *pubSz);

int wc_ed448_check_key(ed448_key* key);


int wc_ed448_size(ed448_key* key);
```

```
int wc_ed448_priv_size(ed448_key* key);

int wc_ed448_pub_size(ed448_key* key);

int wc_ed448_sig_size(ed448_key* key);
```

C.25 dox_comments/header_files/error-crypt.h

C.25.1 Functions

	Name
void	**wc_ErrorString**(int err, char * buff)This function stores the error string for a particular error code in the given buffer.
const char *	**wc_GetErrorString**(int error)This function returns the error string for a particular error code.

C.25.2 Functions Documentation

C.25.2.1 function wc_ErrorString

```
void wc_ErrorString(
    int err,
    char * buff
)
```

This function stores the error string for a particular error code in the given buffer.

Parameters:

- **error** error code for which to get the string
- **buffer** buffer in which to store the error string. Buffer should be at least WOLFSSL_MAX_ERROR_SZ (80 bytes) long

See: wc_GetErrorString

Return: none No returns.

Example

```
char errorMsg[WOLFSSL_MAX_ERROR_SZ];
int err = wc_some_function();

if( err != 0) { // error occurred
    wc_ErrorString(err, errorMsg);
}
```

C.25.2.2 function wc_GetErrorString

```
const char * wc_GetErrorString(
    int error
)
```

This function returns the error string for a particular error code.

Parameters:

- **error** error code for which to get the string

See: wc_ErrorString

Return: string Returns the error string for an error code as a string literal.

Example

```
char * errorMsg;
int err = wc_some_function();

if( err != 0) { // error occurred
    errorMsg = wc_GetErrorString(err);
}
```

C.25.3 Source code

```
void wc_ErrorString(int err, char* buff);

const char* wc_GetErrorString(int error);
```

C.26 dox_comments/header_files/evp.h

C.26.1 Functions

	Name
const WOLFSSL_EVP_CIPHER *	**wolfSSL_EVP_des_ede3_ecb.
const WOLFSSL_EVP_CIPHER *	**wolfSSL_EVP_des_cbc**(void)Getter functions for the respective WOLFSSL_EVP_CIPHER pointers. wolfSSL_EVP_init() must be called once in the program first to populate these cipher strings. WOLFSSL_DES_ECB macro must be defined for wolfSSL_EVP_des_ecb().
int	**wolfSSL_EVP_DigestInit_ex**(WOLFSSL_EVP_MD_CTX * ctx, const WOLFSSL_EVP_MD * type, WOLFSSL_ENGINE * impl)Function for initializing WOLFSSL_EVP_MD_CTX. This function is a wrapper for wolfSSL_EVP_DigestInit() because wolfSSL does not use WOLFSSL_ENGINE.
int	**wolfSSL_EVP_CipherInit_ex**(WOLFSSL_EVP_CIPHER_CTX * ctx, const WOLFSSL_EVP_CIPHER * type, WOLFSSL_ENGINE * impl, const unsigned char * key, const unsigned char * iv, int enc)Function for initializing WOLFSSL_EVP_CIPHER_CTX. This function is a wrapper for wolfSSL_CipherInit() because wolfSSL does not use WOLFSSL_ENGINE.

	Name
int	**wolfSSL_EVP_EncryptInit_ex**(WOLFSSL_EVP_CIPHER_CTX * ctx, const WOLFSSL_EVP_CIPHER * type, WOLFSSL_ENGINE * impl, const unsigned char * key, const unsigned char * iv)Function for initializing WOLFSSL_EVP_CIPHER_CTX. This function is a wrapper for wolfSSL_EVP_CipherInit() because wolfSSL does not use WOLFSSL_ENGINE. Sets encrypt flag to be encrypt.
int	**wolfSSL_EVP_DecryptInit_ex**(WOLFSSL_EVP_CIPHER_CTX * ctx, const WOLFSSL_EVP_CIPHER * type, WOLFSSL_ENGINE * impl, const unsigned char * key, const unsigned char * iv)Function for initializing WOLFSSL_EVP_CIPHER_CTX. This function is a wrapper for wolfSSL_EVP_CipherInit() because wolfSSL does not use WOLFSSL_ENGINE. Sets encrypt flag to be decrypt.
int	**wolfSSL_EVP_CipherUpdate**(WOLFSSL_EVP_CIPHER_CTX * ctx, unsigned char * out, int * outl, const unsigned char * in, int inl)Function for encrypting/decrypting data. In buffer is added to be encrypted or decrypted and out buffer holds the results. outl will be the length of encrypted/decrypted information.
int	**wolfSSL_EVP_CipherFinal**(WOLFSSL_EVP_CIPHER_CTX * ctx, unsigned char * out, int * outl)This function performs the final cipher operations adding in padding. If WOLFSSL_EVP_CIPH_NO_PADDING flag is set in WOLFSSL_EVP_CIPHER_CTX structure then 1 is returned and no encryption/decryption is done. If padding flag is seti padding is added and encrypted when ctx is set to encrypt, padding values are checked when set to decrypt.
int	**wolfSSL_EVP_CIPHER_CTX_set_key_length**(WOLFSSL_EVP_CIPHE * ctx, int keylen)Setter function for WOLFSSL_EVP_CIPHER_CTX structure key length.
int	**wolfSSL_EVP_CIPHER_CTX_block_size**(const WOLFSSL_EVP_CIPHER_CTX * ctx)This is a getter function for the ctx block size.
int	**wolfSSL_EVP_CIPHER_block_size**(const WOLFSSL_EVP_CIPHER * cipher)This is a getter function for the block size of cipher.
void	**wolfSSL_EVP_CIPHER_CTX_set_flags**(WOLFSSL_EVP_CIPHER_CTX * ctx, int flags)Setter function for WOLFSSL_EVP_CIPHER_CTX structure.
void	**wolfSSL_EVP_CIPHER_CTX_clear_flags**(WOLFSSL_EVP_CIPHER_C * ctx, int flags)Clearing function for WOLFSSL_EVP_CIPHER_CTX structure.

	Name
int	**wolfSSL_EVP_CIPHER_CTX_set_padding**(WOLFSSL_EVP_CIPHER_ * c, int pad)Setter function for WOLFSSL_EVP_CIPHER_CTX structure to use padding.
unsigned long	**wolfSSL_EVP_CIPHER_CTX_flags**(const WOLFSSL_EVP_CIPHER_CTX * ctx)Getter function for WOLFSSL_EVP_CIPHER_CTX structure. Deprecated v1.1.0.

C.26.2 Functions Documentation

C.26.2.1 function wolfSSL_EVP_des_ede3_ecb

```
const WOLFSSL_EVP_CIPHER * wolfSSL_EVP_des_ede3_ecb(
    void
)
```

Getter functions for the respective WOLFSSL_EVP_CIPHER pointers. wolfSSL_EVP_init() must be called once in the program first to populate these cipher strings. WOLFSSL_DES_ECB macro must be defined for wolfSSL_EVP_des_ede3_ecb().

Parameters:

- **none** No parameters.

See: wolfSSL_EVP_CIPHER_CTX_init

Return: pointer Returns a WOLFSSL_EVP_CIPHER pointer for DES EDE3 operations.

Example

```
printf("block size des ede3 cbc = %d\n",
wolfSSL_EVP_CIPHER_block_size(wolfSSL_EVP_des_ede3_cbc()));
printf("block size des ede3 ecb = %d\n",
wolfSSL_EVP_CIPHER_block_size(wolfSSL_EVP_des_ede3_ecb()));
```

C.26.2.2 function wolfSSL_EVP_des_cbc

```
const WOLFSSL_EVP_CIPHER * wolfSSL_EVP_des_cbc(
    void
)
```

Getter functions for the respective WOLFSSL_EVP_CIPHER pointers. wolfSSL_EVP_init() must be called once in the program first to populate these cipher strings. WOLFSSL_DES_ECB macro must be defined for wolfSSL_EVP_des_ecb().

Parameters:

- **none** No parameters.

See: wolfSSL_EVP_CIPHER_CTX_init

Return: pointer Returns a WOLFSSL_EVP_CIPHER pointer for DES operations.

Example

```
WOLFSSL_EVP_CIPHER* cipher;
cipher = wolfSSL_EVP_des_cbc();
…
```

C.26.2.3 function wolfSSL_EVP_DigestInit_ex

```
int wolfSSL_EVP_DigestInit_ex(
    WOLFSSL_EVP_MD_CTX * ctx,
    const WOLFSSL_EVP_MD * type,
    WOLFSSL_ENGINE * impl
)
```

Function for initializing WOLFSSL_EVP_MD_CTX. This function is a wrapper for wolfSSL_EVP_DigestInit() because wolfSSL does not use WOLFSSL_ENGINE.

Parameters:

- **ctx** structure to initialize.
- **type** type of hash to do, for example SHA.
- **impl** engine to use. N/A for wolfSSL, can be NULL.

See:

- wolfSSL_EVP_MD_CTX_new
- wolfCrypt_Init
- wolfSSL_EVP_MD_CTX_free

Return:

- SSL_SUCCESS If successfully set.
- SSL_FAILURE If not successful.

Example

```
WOLFSSL_EVP_MD_CTX* md = NULL;
wolfCrypt_Init();
md = wolfSSL_EVP_MD_CTX_new();
if (md == NULL) {
    printf("error setting md\n");
    return -1;
}
printf("cipher md init ret = %d\n", wolfSSL_EVP_DigestInit_ex(md,
wolfSSL_EVP_sha1(), e));
//free resources
```

C.26.2.4 function wolfSSL_EVP_CipherInit_ex

```
int wolfSSL_EVP_CipherInit_ex(
    WOLFSSL_EVP_CIPHER_CTX * ctx,
    const WOLFSSL_EVP_CIPHER * type,
    WOLFSSL_ENGINE * impl,
    const unsigned char * key,
    const unsigned char * iv,
    int enc
)
```

Function for initializing WOLFSSL_EVP_CIPHER_CTX. This function is a wrapper for wolfSSL_CipherInit() because wolfSSL does not use WOLFSSL_ENGINE.

Parameters:

- **ctx** structure to initialize.
- **type** type of encryption/decryption to do, for example AES.
- **impl** engine to use. N/A for wolfSSL, can be NULL.
- **key** key to set .

- **iv** iv if needed by algorithm.
- **enc** encryption (1) or decryption (0) flag.

See:

- wolfSSL_EVP_CIPHER_CTX_new
- wolfCrypt_Init
- wolfSSL_EVP_CIPHER_CTX_free

Return:

- SSL_SUCCESS If successfully set.
- SSL_FAILURE If not successful.

Example

```
WOLFSSL_EVP_CIPHER_CTX* ctx = NULL;
WOLFSSL_ENGINE* e = NULL;
unsigned char key[16];
unsigned char iv[12];
wolfCrypt_Init();
ctx = wolfSSL_EVP_CIPHER_CTX_new();
if (ctx == NULL) {
    printf("issue creating ctx\n");
    return -1;
}

printf("cipher init ex error ret = %d\n", wolfSSL_EVP_CipherInit_ex(NULL,
EVP_aes_128_    cbc(), e, key, iv, 1));
printf("cipher init ex success ret = %d\n", wolfSSL_EVP_CipherInit_ex(ctx,
EVP_aes_128_c    bc(), e, key, iv, 1));
// free resources
```

C.26.2.5 function wolfSSL_EVP_EncryptInit_ex

```
int wolfSSL_EVP_EncryptInit_ex(
    WOLFSSL_EVP_CIPHER_CTX * ctx,
    const WOLFSSL_EVP_CIPHER * type,
    WOLFSSL_ENGINE * impl,
    const unsigned char * key,
    const unsigned char * iv
)
```

Function for initializing WOLFSSL_EVP_CIPHER_CTX. This function is a wrapper for wolfSSL_EVP_CipherInit() because wolfSSL does not use WOLFSSL_ENGINE. Sets encrypt flag to be encrypt.

Parameters:

- **ctx** structure to initialize.
- **type** type of encryption to do, for example AES.
- **impl** engine to use. N/A for wolfSSL, can be NULL.
- **key** key to use.
- **iv** iv to use.

See:

- wolfSSL_EVP_CIPHER_CTX_new
- wolfCrypt_Init
- wolfSSL_EVP_CIPHER_CTX_free

Return:

- SSL_SUCCESS If successfully set.
- SSL_FAILURE If not successful.

Example

```
WOLFSSL_EVP_CIPHER_CTX* ctx = NULL;
wolfCrypt_Init();
ctx = wolfSSL_EVP_CIPHER_CTX_new();
if (ctx == NULL) {
    printf("error setting ctx\n");
    return -1;
}
printf("cipher ctx init ret = %d\n", wolfSSL_EVP_EncryptInit_ex(ctx,
wolfSSL_EVP_aes_128_cbc(), e, key, iv));
//free resources
```

C.26.2.6 function wolfSSL_EVP_DecryptInit_ex

```
int wolfSSL_EVP_DecryptInit_ex(
    WOLFSSL_EVP_CIPHER_CTX * ctx,
    const WOLFSSL_EVP_CIPHER * type,
    WOLFSSL_ENGINE * impl,
    const unsigned char * key,
    const unsigned char * iv
)
```

Function for initializing WOLFSSL_EVP_CIPHER_CTX. This function is a wrapper for wolfSSL_EVP_CipherInit() because wolfSSL does not use WOLFSSL_ENGINE. Sets encrypt flag to be decrypt.

Parameters:

- **ctx** structure to initialize.
- **type** type of encryption/decryption to do, for example AES.
- **impl** engine to use. N/A for wolfSSL, can be NULL.
- **key** key to set .
- **iv** iv if needed by algorithm.
- **enc** encryption (1) or decryption (0) flag.

See:

- wolfSSL_EVP_CIPHER_CTX_new
- wolfCrypt_Init
- wolfSSL_EVP_CIPHER_CTX_free

Return:

- SSL_SUCCESS If successfully set.
- SSL_FAILURE If not successful.

Example

```
WOLFSSL_EVP_CIPHER_CTX* ctx = NULL;
WOLFSSL_ENGINE* e = NULL;
unsigned char key[16];
unsigned char iv[12];

wolfCrypt_Init();
```

```
ctx = wolfSSL_EVP_CIPHER_CTX_new();
if (ctx == NULL) {
    printf("issue creating ctx\n");
    return -1;
}

printf("cipher init ex error ret = %d\n", wolfSSL_EVP_DecryptInit_ex(NULL,
EVP_aes_128_    cbc(), e, key, iv, 1));
printf("cipher init ex success ret = %d\n", wolfSSL_EVP_DecryptInit_ex(ctx,
EVP_aes_128_c    bc(), e, key, iv, 1));
// free resources
```

C.26.2.7 function wolfSSL_EVP_CipherUpdate

```
int wolfSSL_EVP_CipherUpdate(
    WOLFSSL_EVP_CIPHER_CTX * ctx,
    unsigned char * out,
    int * outl,
    const unsigned char * in,
    int inl
)
```

Function for encrypting/decrypting data. In buffer is added to be encrypted or decrypted and out buffer holds the results. outl will be the length of encrypted/decrypted information.

Parameters:

- **ctx** structure to get cipher type from.
- **out** buffer to hold output.
- **outl** adjusted to be size of output.
- **in** buffer to perform operation on.
- **inl** length of input buffer.

See:

- wolfSSL_EVP_CIPHER_CTX_new
- wolfCrypt_Init
- wolfSSL_EVP_CIPHER_CTX_free

Return:

- SSL_SUCCESS If successful.
- SSL_FAILURE If not successful.

Example

```
WOLFSSL_EVP_CIPHER_CTX* ctx = NULL;
unsigned char out[100];
int outl;
unsigned char in[100];
int inl = 100;

ctx = wolfSSL_EVP_CIPHER_CTX_new();
// set up ctx
ret = wolfSSL_EVP_CipherUpdate(ctx, out, outl, in, inl);
// check ret value
// buffer out holds outl bytes of data
// free resources
```

C.26.2.8 function wolfSSL_EVP_CipherFinal

```
int wolfSSL_EVP_CipherFinal(
    WOLFSSL_EVP_CIPHER_CTX * ctx,
    unsigned char * out,
    int * outl
)
```

This function performs the final cipher operations adding in padding. If WOLFSSL_EVP_CIPH_NO_PADDING flag is set in WOLFSSL_EVP_CIPHER_CTX structure then 1 is returned and no encryption/decryption is done. If padding flag is seti padding is added and encrypted when ctx is set to encrypt, padding values are checked when set to decrypt.

Parameters:

- **ctx** structure to decrypt/encrypt with.
- **out** buffer for final decrypt/encrypt.
- **out1** size of out buffer when data has been added by function.

See: wolfSSL_EVP_CIPHER_CTX_new

Return:

- 1 Returned on success.
- 0 If encountering a failure.

Example

```
WOLFSSL_EVP_CIPHER_CTX* ctx;
int out1;
unsigned char out[64];
// create ctx
wolfSSL_EVP_CipherFinal(ctx, out, &out1);
```

C.26.2.9 function wolfSSL_EVP_CIPHER_CTX_set_key_length

```
int wolfSSL_EVP_CIPHER_CTX_set_key_length(
    WOLFSSL_EVP_CIPHER_CTX * ctx,
    int keylen
)
```

Setter function for WOLFSSL_EVP_CIPHER_CTX structure key length.

Parameters:

- **ctx** structure to set key length.
- **keylen** key length.

See: wolfSSL_EVP_CIPHER_flags

Return:

- SSL_SUCCESS If successfully set.
- SSL_FAILURE If failed to set key length.

Example

```
WOLFSSL_EVP_CIPHER_CTX* ctx;
int keylen;
// create ctx
wolfSSL_EVP_CIPHER_CTX_set_key_length(ctx, keylen);
```

C.26.2.10 function wolfSSL_EVP_CIPHER_CTX_block_size

```
int wolfSSL_EVP_CIPHER_CTX_block_size(
    const WOLFSSL_EVP_CIPHER_CTX * ctx
)
```

This is a getter function for the ctx block size.

Parameters:

- **ctx** the cipher ctx to get block size of.

See: wolfSSL_EVP_CIPHER_block_size

Return: size Returns ctx->block_size.

Example

```
const WOLFSSL_CVP_CIPHER_CTX* ctx;
//set up ctx
printf("block size = %d\n", wolfSSL_EVP_CIPHER_CTX_block_size(ctx));
```

C.26.2.11 function wolfSSL_EVP_CIPHER_block_size

```
int wolfSSL_EVP_CIPHER_block_size(
    const WOLFSSL_EVP_CIPHER * cipher
)
```

This is a getter function for the block size of cipher.

Parameters:

- **cipher** cipher to get block size of.

See: wolfSSL_EVP_aes_256_ctr

Return: size returns the block size.

Example

```
printf("block size = %d\n",
wolfSSL_EVP_CIPHER_block_size(wolfSSL_EVP_aes_256_ecb()));
```

C.26.2.12 function wolfSSL_EVP_CIPHER_CTX_set_flags

```
void wolfSSL_EVP_CIPHER_CTX_set_flags(
    WOLFSSL_EVP_CIPHER_CTX * ctx,
    int flags
)
```

Setter function for WOLFSSL_EVP_CIPHER_CTX structure.

Parameters:

- **ctx** structure to set flag.
- **flag** flag to set in structure.

See:

- wolfSSL_EVP_CIPHER_flags
- wolfSSL_EVP_CIPHER_CTX_flags

Return: none No returns.

Example

```
WOLFSSL_EVP_CIPHER_CTX* ctx;
int flag;
// create ctx
wolfSSL_EVP_CIPHER_CTX_set_flags(ctx, flag);
```

C.26.2.13 function wolfSSL_EVP_CIPHER_CTX_clear_flags

```
void wolfSSL_EVP_CIPHER_CTX_clear_flags(
    WOLFSSL_EVP_CIPHER_CTX * ctx,
    int flags
)
```

Clearing function for WOLFSSL_EVP_CIPHER_CTX structure.

Parameters:

- **ctx** structure to clear flag.
- **flag** flag value to clear in structure.

See:

- wolfSSL_EVP_CIPHER_flags
- wolfSSL_EVP_CIPHER_CTX_flags

Return: none No returns.

Example

```
WOLFSSL_EVP_CIPHER_CTX* ctx;
int flag;
// create ctx
wolfSSL_EVP_CIPHER_CTX_clear_flags(ctx, flag);
```

C.26.2.14 function wolfSSL_EVP_CIPHER_CTX_set_padding

```
int wolfSSL_EVP_CIPHER_CTX_set_padding(
    WOLFSSL_EVP_CIPHER_CTX * c,
    int pad
)
```

Setter function for WOLFSSL_EVP_CIPHER_CTX structure to use padding.

Parameters:

- **ctx** structure to set padding flag.
- **padding** 0 for not setting padding, 1 for setting padding.

See: wolfSSL_EVP_CIPHER_CTX_new

Return:

- SSL_SUCCESS If successfully set.
- BAD_FUNC_ARG If null argument passed in.

Example

```
WOLFSSL_EVP_CIPHER_CTX* ctx;
// create ctx
wolfSSL_EVP_CIPHER_CTX_set_padding(ctx, 1);
```

C.26.2.15 function wolfSSL_EVP_CIPHER_CTX_flags

```
unsigned long wolfSSL_EVP_CIPHER_CTX_flags(
    const WOLFSSL_EVP_CIPHER_CTX * ctx
)
```

Getter function for WOLFSSL_EVP_CIPHER_CTX structure. Deprecated v1.1.0.

Parameters:

- **ctx** structure to get flag.

See:

- wolfSSL_EVP_CIPHER_CTX_new
- wolfSSL_EVP_CIPHER_flags

Return: unsigned long of flags/mode.

Example

```
WOLFSSL_EVP_CIPHER_CTX* ctx;
unsigned long flags;
ctx = wolfSSL_EVP_CIPHER_CTX_new()
flags = wolfSSL_EVP_CIPHER_CTX_flags(ctx);
```

C.26.3 Source code

```
const WOLFSSL_EVP_CIPHER* wolfSSL_EVP_des_ede3_ecb(void);

const WOLFSSL_EVP_CIPHER* wolfSSL_EVP_des_cbc(void);

int wolfSSL_EVP_DigestInit_ex(WOLFSSL_EVP_MD_CTX* ctx,
                                    const WOLFSSL_EVP_MD* type,
                                    WOLFSSL_ENGINE *impl);

int  wolfSSL_EVP_CipherInit_ex(WOLFSSL_EVP_CIPHER_CTX* ctx,
                                   const WOLFSSL_EVP_CIPHER* type,
                                   WOLFSSL_ENGINE *impl,
                                   const unsigned char* key,
                                   const unsigned char* iv,
                                   int enc);

int  wolfSSL_EVP_EncryptInit_ex(WOLFSSL_EVP_CIPHER_CTX* ctx,
                                   const WOLFSSL_EVP_CIPHER* type,
                                   WOLFSSL_ENGINE *impl,
                                   const unsigned char* key,
                                   const unsigned char* iv);

int  wolfSSL_EVP_DecryptInit_ex(WOLFSSL_EVP_CIPHER_CTX* ctx,
                                   const WOLFSSL_EVP_CIPHER* type,
                                   WOLFSSL_ENGINE *impl,
                                   const unsigned char* key,
                                   const unsigned char* iv);

int wolfSSL_EVP_CipherUpdate(WOLFSSL_EVP_CIPHER_CTX *ctx,
                                  unsigned char *out, int *outl,
```

```
                                    const unsigned char *in, int inl);

int  wolfSSL_EVP_CipherFinal(WOLFSSL_EVP_CIPHER_CTX *ctx,
                                    unsigned char *out, int *outl);

int  wolfSSL_EVP_CIPHER_CTX_set_key_length(WOLFSSL_EVP_CIPHER_CTX* ctx,
                                                        int keylen);

int wolfSSL_EVP_CIPHER_CTX_block_size(const WOLFSSL_EVP_CIPHER_CTX *ctx);

int wolfSSL_EVP_CIPHER_block_size(const WOLFSSL_EVP_CIPHER *cipher);

void wolfSSL_EVP_CIPHER_CTX_set_flags(WOLFSSL_EVP_CIPHER_CTX *ctx, int flags);

void wolfSSL_EVP_CIPHER_CTX_clear_flags(WOLFSSL_EVP_CIPHER_CTX *ctx, int
 ↪  flags);

int  wolfSSL_EVP_CIPHER_CTX_set_padding(WOLFSSL_EVP_CIPHER_CTX *c, int pad);


unsigned long wolfSSL_EVP_CIPHER_CTX_flags(const WOLFSSL_EVP_CIPHER_CTX *ctx);
```

C.27 dox_comments/header_files/hash.h

C.27.1 Functions

	Name
int	**wc_HashGetOID**(enum wc_HashType hash_type)This function will return the OID for the wc_HashType provided.
int	**wc_HashGetDigestSize**(enum wc_HashType hash_type)This function returns the size of the digest (output) for a hash_type. The returns size is used to make sure the output buffer provided to wc_Hash is large enough.
int	**wc_Hash**(enum wc_HashType hash_type, const byte * data, word32 data_len, byte * hash, word32 hash_len)This function performs a hash on the provided data buffer and returns it in the hash buffer provided.
int	**wc_Md5Hash**(const byte * data, word32 len, byte * hash)Convenience function, handles all the hashing and places the result into hash.
int	**wc_ShaHash**(const byte * data, word32 len, byte * hash)Convenience function, handles all the hashing and places the result into hash.
int	**wc_Sha224Hash**(const byte * data, word32 len, byte * hash)Convenience function, handles all the hashing and places the result into hash.
int	**wc_Sha256Hash**(const byte * data, word32 len, byte * hash)Convenience function, handles all the hashing and places the result into hash.

	Name
int	**wc_Sha384Hash**(const byte * data, word32 len, byte * hash)Convenience function, handles all the hashing and places the result into hash.
int	**wc_Sha512Hash**(const byte * data, word32 len, byte * hash)Convenience function, handles all the hashing and places the result into hash.
int	**wc_Sha3_224Hash**(const byte * data, word32 len, byte * hash)Convenience function, handles all the hashing and places the result into hash.
int	**wc_Sha3_256Hash**(const byte * data, word32 len, byte * hash)Convenience function, handles all the hashing and places the result into hash.
int	**wc_Sha3_384Hash**(const byte * data, word32 len, byte * hash)Convenience function, handles all the hashing and places the result into hash.
int	**wc_Sha3_512Hash**(const byte * data, word32 len, byte * hash)Convenience function, handles all the hashing and places the result into hash.
int	**wc_Shake128Hash**(const byte * data, word32 len, byte * hash)Convenience function, handles all the hashing and places the result into hash.
int	**wc_Shake256Hash**(const byte * data, word32 len, byte * hash)Convenience function, handles all the hashing and places the result into hash.

C.27.2 Functions Documentation

C.27.2.1 function wc_HashGetOID

```
int wc_HashGetOID(
    enum wc_HashType hash_type
)
```

This function will return the OID for the wc_HashType provided.

Parameters:

- **hash_type** A hash type from the “enum wc_HashType” such as “WC_HASH_TYPE_SHA256”.

See:

- wc_HashGetDigestSize
- wc_Hash

Return:

- OID returns value greater than 0
- HASH_TYPE_E hash type not supported.
- BAD_FUNC_ARG one of the provided arguments is incorrect.

Example

```
enum wc_HashType hash_type = WC_HASH_TYPE_SHA256;
int oid = wc_HashGetOID(hash_type);
if (oid > 0) {
    // Success
}
```

C.27.2.2 function wc_HashGetDigestSize

```
int wc_HashGetDigestSize(
    enum wc_HashType hash_type
)
```

This function returns the size of the digest (output) for a hash_type. The returns size is used to make sure the output buffer provided to wc_Hash is large enough.

Parameters:

- **hash_type** A hash type from the "enum wc_HashType" such as "WC_HASH_TYPE_SHA256".

See: wc_Hash

Return:

- Success A positive return value indicates the digest size for the hash.
- Error Returns HASH_TYPE_E if hash_type is not supported.
- Failure Returns BAD_FUNC_ARG if an invalid hash_type was used.

Example

```
int hash_len = wc_HashGetDigestSize(hash_type);
if (hash_len <= 0) {
WOLFSSL_MSG("Invalid hash type/len");
return BAD_FUNC_ARG;
}
```

C.27.2.3 function wc_Hash

```
int wc_Hash(
    enum wc_HashType hash_type,
    const byte * data,
    word32 data_len,
    byte * hash,
    word32 hash_len
)
```

This function performs a hash on the provided data buffer and returns it in the hash buffer provided.

Parameters:

- **hash_type** A hash type from the "enum wc_HashType" such as "WC_HASH_TYPE_SHA256".
- **data** Pointer to buffer containing the data to hash.
- **data_len** Length of the data buffer.
- **hash** Pointer to buffer used to output the final hash to.
- **hash_len** Length of the hash buffer.

See: wc_HashGetDigestSize

Return: 0 Success, else error (such as BAD_FUNC_ARG or BUFFER_E).

Example

```
enum wc_HashType hash_type = WC_HASH_TYPE_SHA256;
int hash_len = wc_HashGetDigestSize(hash_type);
if (hash_len > 0) {
    int ret = wc_Hash(hash_type, data, data_len, hash_data, hash_len);
    if(ret == 0) {
        // Success
```

```
    }
}
```

C.27.2.4 function wc_Md5Hash

```
int wc_Md5Hash(
    const byte * data,
    word32 len,
    byte * hash
)
```

Convenience function, handles all the hashing and places the result into hash.

Parameters:

- **data** the data to hash
- **len** the length of data
- **hash** Byte array to hold hash value.

See:

- wc_Md5Hash
- wc_Md5Final
- wc_InitMd5

Return:

- 0 Returned upon successfully hashing the data.
- Memory_E memory error, unable to allocate memory. This is only possible with the small stack option enabled.

Example

```
const byte* data;
word32 data_len;
byte* hash;
int ret;
...
ret = wc_Md5Hash(data, data_len, hash);
if (ret != 0) {
     // Md5 Hash Failure Case.
}
```

C.27.2.5 function wc_ShaHash

```
int wc_ShaHash(
    const byte * data,
    word32 len,
    byte * hash
)
```

Convenience function, handles all the hashing and places the result into hash.

Parameters:

- **data** the data to hash
- **len** the length of data
- **hash** Byte array to hold hash value.

See:

- wc_ShaHash
- wc_ShaFinal
- wc_InitSha

Return:

- 0 Returned upon successfully ....
- Memory_E memory error, unable to allocate memory. This is only possible with the small stack option enabled.

Example

none

C.27.2.6 function wc_Sha224Hash

```
int wc_Sha224Hash(
    const byte * data,
    word32 len,
    byte * hash
)
```

Convenience function, handles all the hashing and places the result into hash.

Parameters:

- **data** the data to hash
- **len** the length of data
- **hash** Byte array to hold hash value.

See:

- wc_InitSha224
- wc_Sha224Update
- wc_Sha224Final

Return:

- 0 Success
- <0 Error

Example

none

C.27.2.7 function wc_Sha256Hash

```
int wc_Sha256Hash(
    const byte * data,
    word32 len,
    byte * hash
)
```

Convenience function, handles all the hashing and places the result into hash.

Parameters:

- **data** the data to hash
- **len** the length of data
- **hash** Byte array to hold hash value.

See:

- wc_Sha256Hash
- wc_Sha256Final
- wc_InitSha256

Return:

- 0 Returned upon successfully ...
- Memory_E memory error, unable to allocate memory. This is only possible with the small stack option enabled.

Example

none

C.27.2.8 function wc_Sha384Hash

```
int wc_Sha384Hash(
    const byte * data,
    word32 len,
    byte * hash
)
```

Convenience function, handles all the hashing and places the result into hash.

Parameters:

- **data** the data to hash
- **len** the length of data
- **hash** Byte array to hold hash value.

See:

- wc_Sha384Hash
- wc_Sha384Final
- wc_InitSha384

Return:

- 0 Returned upon successfully hashing the data
- Memory_E memory error, unable to allocate memory. This is only possible with the small stack option enabled.

Example

none

C.27.2.9 function wc_Sha512Hash

```
int wc_Sha512Hash(
    const byte * data,
    word32 len,
    byte * hash
)
```

Convenience function, handles all the hashing and places the result into hash.

Parameters:

- **data** the data to hash
- **len** the length of data
- **hash** Byte array to hold hash value.

See:

- wc_Sha512Hash
- wc_Sha512Final
- wc_InitSha512

Return:

- 0 Returned upon successfully hashing the inputted data
- Memory_E memory error, unable to allocate memory. This is only possible with the small stack option enabled.

Example

none

C.27.2.10 function wc_Sha3_224Hash

```
int wc_Sha3_224Hash(
    const byte * data,
    word32 len,
    byte * hash
)
```

Convenience function, handles all the hashing and places the result into hash.

Parameters:

- **data** the data to hash
- **len** the length of data
- **hash** Byte array to hold hash value.

See:

- wc_InitSha3_224
- wc_Sha3_224_Update
- wc_Sha3_224_Final

Return:

- 0 Returned upon successfully hashing the data
- Memory_E memory error, unable to allocate memory. This is only possible with the small stack option enabled.

Example

none

C.27.2.11 function wc_Sha3_256Hash

```
int wc_Sha3_256Hash(
    const byte * data,
    word32 len,
    byte * hash
)
```

Convenience function, handles all the hashing and places the result into hash.

Parameters:

- **data** the data to hash
- **len** the length of data
- **hash** Byte array to hold hash value.

See:

- wc_InitSha3_256
- wc_Sha3_256_Update
- wc_Sha3_256_Final

Return:

- 0 Returned upon successfully hashing the data
- Memory_E memory error, unable to allocate memory. This is only possible with the small stack option enabled.

Example

none

C.27.2.12 function wc_Sha3_384Hash

```
int wc_Sha3_384Hash(
    const byte * data,
    word32 len,
    byte * hash
)
```

Convenience function, handles all the hashing and places the result into hash.

Parameters:

- **data** the data to hash
- **len** the length of data
- **hash** Byte array to hold hash value.

See:

- wc_InitSha3_384
- wc_Sha3_384_Update
- wc_Sha3_384_Final

Return:

- 0 Returned upon successfully hashing the data
- Memory_E memory error, unable to allocate memory. This is only possible with the small stack option enabled.

Example

none

C.27.2.13 function wc_Sha3_512Hash

```
int wc_Sha3_512Hash(
    const byte * data,
    word32 len,
    byte * hash
)
```

Convenience function, handles all the hashing and places the result into hash.

Parameters:

- **data** the data to hash
- **len** the length of data
- **hash** Byte array to hold hash value.

See:

- wc_InitSha3_512
- wc_Sha3_512_Update
- wc_Sha3_512_Final

Return:

- 0 Returned upon successfully hashing the inputted data
- Memory_E memory error, unable to allocate memory. This is only possible with the small stack option enabled.

Example

none

C.27.2.14 function wc_Shake128Hash

```
int wc_Shake128Hash(
    const byte * data,
    word32 len,
    byte * hash
)
```

Convenience function, handles all the hashing and places the result into hash.

Parameters:

- **data** the data to hash
- **len** the length of data
- **hash** Byte array to hold hash value.

See:

- wc_InitShake128
- wc_Shake128_Update
- wc_Shake128_Final

Return:

- 0 Returned upon successfully hashing the inputted data
- Memory_E memory error, unable to allocate memory. This is only possible with the small stack option enabled.

Example

none

C.27.2.15 function wc_Shake256Hash

```
int wc_Shake256Hash(
    const byte * data,
    word32 len,
    byte * hash
)
```

Convenience function, handles all the hashing and places the result into hash.

Parameters:

- **data** the data to hash
- **len** the length of data
- **hash** Byte array to hold hash value.

See:

- wc_InitShake256
- wc_Shake256_Update
- wc_Shake256_Final

Return:

- 0 Returned upon successfully hashing the inputted data
- Memory_E memory error, unable to allocate memory. This is only possible with the small stack option enabled.

Example

none

C.27.3 Source code

```
int wc_HashGetOID(enum wc_HashType hash_type);

int wc_HashGetDigestSize(enum wc_HashType hash_type);

int wc_Hash(enum wc_HashType hash_type,
    const byte* data, word32 data_len,
    byte* hash, word32 hash_len);

int wc_Md5Hash(const byte* data, word32 len, byte* hash);

int wc_ShaHash(const byte* data, word32 len, byte* hash);

int wc_Sha224Hash(const byte* data, word32 len, byte* hash);

int wc_Sha256Hash(const byte* data, word32 len, byte* hash);

int wc_Sha384Hash(const byte* data, word32 len, byte* hash);

int wc_Sha512Hash(const byte* data, word32 len, byte* hash);

int wc_Sha3_224Hash(const byte* data, word32 len, byte* hash);

int wc_Sha3_256Hash(const byte* data, word32 len, byte* hash);

int wc_Sha3_384Hash(const byte* data, word32 len, byte* hash);

int wc_Sha3_512Hash(const byte* data, word32 len, byte* hash);

int wc_Shake128Hash(const byte* data, word32 len, byte* hash);

int wc_Shake256Hash(const byte* data, word32 len, byte* hash);
```

C.28 dox_comments/header_files/hmac.h

C.28.1 Functions

	Name
int	**wc_HmacSetKey**(Hmac * hmac, int type, const byte * key, word32 keySz)This function initializes an Hmac object, setting its encryption type, key and HMAC length.
int	**wc_HmacUpdate**(Hmac * hmac, const byte * in, word32 sz)This function updates the message to authenticate using HMAC. It should be called after the Hmac object has been initialized with wc_HmacSetKey. This function may be called multiple times to update the message to hash. After calling wc_HmacUpdate as desired, one should call wc_HmacFinal to obtain the final authenticated message tag.
int	**wc_HmacFinal**(Hmac * hmac, byte * out)This function computes the final hash of an Hmac object's message.
int	**wolfSSL_GetHmacMaxSize**(void)This function returns the largest HMAC digest size available based on the configured cipher suites.
int	**wc_HKDF**(int type, const byte * inKey, word32 inKeySz, const byte * salt, word32 saltSz, const byte * info, word32 infoSz, byte * out, word32 outSz)This function provides access to a HMAC Key Derivation Function (HKDF). It utilizes HMAC to convert inKey, with an optional salt and optional info into a derived key, which it stores in out. The hash type defaults to MD5 if 0 or NULL is given.
int	**wc_HKDF_Extract**(int type, const byte * salt, word32 saltSz, const byte * inKey, word32 inKeySz, byte * out)This function provides access to a HMAC Key Derivation Function (HKDF). It utilizes HMAC to convert inKey, with an optional salt into a derived key, which it stores in out. The hash type defaults to MD5 if 0 or NULL is given.
int	**wc_HKDF_Extract_ex**(int type, const byte * salt, word32 saltSz, const byte * inKey, word32 inKeySz, byte * out, void * heap, int devId)This function provides access to a HMAC Key Derivation Function (HKDF). It utilizes HMAC to convert inKey, with an optional salt into a derived key, which it stores in out. The hash type defaults to MD5 if 0 or NULL is given. This is the _ex version adding heap hint and device identifier.

	Name
int	**wc_HKDF_Expand**(int type, const byte * inKey, word32 inKeySz, const byte * info, word32 infoSz, byte * out, word32 outSz)This function provides access to a HMAC Key Derivation Function (HKDF). It utilizes HMAC to convert inKey, with optional info into a derived key, which it stores in out. The hash type defaults to MD5 if 0 or NULL is given.
int	**wc_HKDF_Expand_ex**(int type, const byte * inKey, word32 inKeySz, const byte * info, word32 infoSz, byte * out, word32 outSz, void * heap, int devId)This function provides access to a HMAC Key Derivation Function (HKDF). It utilizes HMAC to convert inKey, with optional info into a derived key, which it stores in out. The hash type defaults to MD5 if 0 or NULL is given. This is the _ex version adding heap hint and device identifier.
int	**wc_Tls13_HKDF_Extract**(byte * prk, const byte * salt, word32 saltLen, byte * ikm, word32 ikmLen, int digest)This function provides access to RFC 5869 HMAC_based Extract_and_Expand Key Derivation Function (HKDF) for TLS v1.3 key derivation.
int	**wc_Tls13_HKDF_Extract_ex**(byte * prk, const byte * salt, word32 saltLen, byte * ikm, word32 ikmLen, int digest, void * heap, int devId)This function provides access to RFC 5869 HMAC_based Extract_and_Expand Key Derivation Function (HKDF) for TLS v1.3 key derivation. This is the _ex version adding heap hint and device identifier.
int	**wc_Tls13_HKDF_Expand_Label_ex**(byte * okm, word32 okmLen, const byte * prk, word32 prkLen, const byte * protocol, word32 protocolLen, const byte * label, word32 labelLen, const byte * info, word32 infoLen, int digest, void * heap, int devId)Expand data using HMAC, salt and label and info. TLS v1.3 defines this function for key derivation. This is the _ex version adding heap hint and device identifier.
int	**wc_Tls13_HKDF_Expand_Label**(byte * okm, word32 okmLen, const byte * prk, word32 prkLen, const byte * protocol, word32 protocolLen, const byte * label, word32 labelLen, const byte * info, word32 infoLen, int digest)Expand data using HMAC, salt and label and info. TLS v1.3 defines this function for key derivation. This is the _ex version adding heap hint and device identifier.

	Name
int	**wc_Tls13_HKDF_Expand_Label_Alloc, but it allocates memory if the stack space usually used isn't enough. Expand data using HMAC, salt and label and info. TLS v1.3 defines this function for key derivation. This is the _ex version adding heap hint and device identifier.

C.28.2 Functions Documentation

C.28.2.1 function wc_HmacSetKey

```
int wc_HmacSetKey(
    Hmac * hmac,
    int type,
    const byte * key,
    word32 keySz
)
```

This function initializes an Hmac object, setting its encryption type, key and HMAC length.

Parameters:

- **hmac** pointer to the Hmac object to initialize
- **type** type specifying which encryption method the Hmac object should use. Valid options are: WC_MD5, WC_SHA, WC_SHA256, WC_SHA384, WC_SHA512, WC_SHA3_224, WC_SHA3_256, WC_SHA3_384 or WC_SHA3_512
- **key** pointer to a buffer containing the key with which to initialize the Hmac object
- **length** length of the key

See:

- wc_HmacUpdate
- wc_HmacFinal

Return:

- 0 Returned on successfully initializing the Hmac object
- BAD_FUNC_ARG Returned if the input type is invalid (see type param)
- MEMORY_E Returned if there is an error allocating memory for the structure to use for hashing
- HMAC_MIN_KEYLEN_E Returned when using a FIPS implementation and the key length specified is shorter than the minimum acceptable FIPS standard of 14 bytes

Example

```
Hmac hmac;
byte key[] = { // initialize with key to use for encryption };
if (wc_HmacSetKey(&hmac, WC_MD5, key, sizeof(key)) != 0) {
    // error initializing Hmac object
}
```

C.28.2.2 function wc_HmacUpdate

```
int wc_HmacUpdate(
    Hmac * hmac,
    const byte * in,
    word32 sz
)
```

This function updates the message to authenticate using HMAC. It should be called after the Hmac object has been initialized with wc_HmacSetKey. This function may be called multiple times to update the message to hash. After calling wc_HmacUpdate as desired, one should call wc_HmacFinal to obtain the final authenticated message tag.

Parameters:

- **hmac** pointer to the Hmac object for which to update the message
- **msg** pointer to the buffer containing the message to append
- **length** length of the message to append

See:

- wc_HmacSetKey
- wc_HmacFinal

Return:

- 0 Returned on successfully updating the message to authenticate
- MEMORY_E Returned if there is an error allocating memory for use with a hashing algorithm

Example

```
Hmac hmac;
byte msg[] = { // initialize with message to authenticate };
byte msg2[] = { // initialize with second half of message };
// initialize hmac
if( wc_HmacUpdate(&hmac, msg, sizeof(msg)) != 0) {
    // error updating message
}
if( wc_HmacUpdate(&hmac, msg2, sizeof(msg)) != 0) {
    // error updating with second message
}
```

C.28.2.3 function wc_HmacFinal

```
int wc_HmacFinal(
    Hmac * hmac,
    byte * out
)
```

This function computes the final hash of an Hmac object's message.

Parameters:

- **hmac** pointer to the Hmac object for which to calculate the final hash
- **hash** pointer to the buffer in which to store the final hash. Should have room available as required by the hashing algorithm chosen

See:

- wc_HmacSetKey
- wc_HmacUpdate

Return:

- 0 Returned on successfully computing the final hash
- MEMORY_E Returned if there is an error allocating memory for use with a hashing algorithm

Example

```
Hmac hmac;
byte hash[MD5_DIGEST_SIZE];
// initialize hmac with MD5 as type
// wc_HmacUpdate() with messages

if (wc_HmacFinal(&hmac, hash) != 0) {
    // error computing hash
}
```

C.28.2.4 function wolfSSL_GetHmacMaxSize

```
int wolfSSL_GetHmacMaxSize(
    void
)
```

This function returns the largest HMAC digest size available based on the configured cipher suites.

Parameters:

- **none** No parameters.

See: none

Return: Success Returns the largest HMAC digest size available based on the configured cipher suites

Example

```
int maxDigestSz = wolfSSL_GetHmacMaxSize();
```

C.28.2.5 function wc_HKDF

```
int wc_HKDF(
    int type,
    const byte * inKey,
    word32 inKeySz,
    const byte * salt,
    word32 saltSz,
    const byte * info,
    word32 infoSz,
    byte * out,
    word32 outSz
)
```

This function provides access to a HMAC Key Derivation Function (HKDF). It utilizes HMAC to convert inKey, with an optional salt and optional info into a derived key, which it stores in out. The hash type defaults to MD5 if 0 or NULL is given.

Parameters:

- **type** hash type to use for the HKDF. Valid types are: WC_MD5, WC_SHA, WC_SHA256, WC_SHA384, WC_SHA512, WC_SHA3_224, WC_SHA3_256, WC_SHA3_384 or WC_SHA3_512
- **inKey** pointer to the buffer containing the key to use for KDF
- **inKeySz** length of the input key
- **salt** pointer to a buffer containing an optional salt. Use NULL instead if not using a salt
- **saltSz** length of the salt. Use 0 if not using a salt
- **info** pointer to a buffer containing optional additional info. Use NULL if not appending extra info
- **infoSz** length of additional info. Use 0 if not using additional info
- **out** pointer to the buffer in which to store the derived key
- **outSz** space available in the output buffer to store the generated key

See: wc_HmacSetKey

Return:

- 0 Returned upon successfully generating a key with the given inputs
- BAD_FUNC_ARG Returned if an invalid hash type is given (see type param)
- MEMORY_E Returned if there is an error allocating memory
- HMAC_MIN_KEYLEN_E May be returned when using a FIPS implementation and the key length specified is shorter than the minimum acceptable FIPS standard

The HMAC configure option is –enable-hmac (on by default) or if building sources directly HAVE_HKDF

Example

```
byte key[] = { // initialize with key };
byte salt[] = { // initialize with salt };
byte derivedKey[MAX_DIGEST_SIZE];

int ret = wc_HKDF(WC_SHA512, key, sizeof(key), salt, sizeof(salt),
NULL, 0, derivedKey, sizeof(derivedKey));
if ( ret != 0 ) {
    // error generating derived key
}
```

C.28.2.6 function wc_HKDF_Extract

```
int wc_HKDF_Extract(
    int type,
    const byte * salt,
    word32 saltSz,
    const byte * inKey,
    word32 inKeySz,
    byte * out
)
```

This function provides access to a HMAC Key Derivation Function (HKDF). It utilizes HMAC to convert inKey, with an optional salt into a derived key, which it stores in out. The hash type defaults to MD5 if 0 or NULL is given.

Parameters:

- **type** hash type to use for the HKDF. Valid types are: WC_MD5, WC_SHA, WC_SHA256, WC_SHA384, WC_SHA512, WC_SHA3_224, WC_SHA3_256, WC_SHA3_384 or WC_SHA3_512
- **salt** pointer to a buffer containing an optional salt. Use NULL instead if not using a salt
- **saltSz** length of the salt. Use 0 if not using a salt
- **inKey** pointer to the buffer containing the key to use for KDF
- **inKeySz** length of the input key
- **out** pointer to the buffer in which to store the derived key

See:

- wc_HKDF
- wc_HKDF_Extract_ex
- wc_HKDF_Expand
- wc_HKDF_Expand_ex

Return:

- 0 Returned upon successfully generating a key with the given inputs
- BAD_FUNC_ARG Returned if an invalid hash type is given (see type param)

- MEMORY_E Returned if there is an error allocating memory
- HMAC_MIN_KEYLEN_E May be returned when using a FIPS implementation and the key length specified is shorter than the minimum acceptable FIPS standard

The HMAC configure option is –enable-hmac (on by default) or if building sources directly HAVE_HKDF

Example

```
byte key[] = { // initialize with key };
byte salt[] = { // initialize with salt };
byte derivedKey[MAX_DIGEST_SIZE];

int ret = wc_HKDF_Extract(WC_SHA512, salt, sizeof(salt), key, sizeof(key),
    derivedKey);
if ( ret != 0 ) {
    // error generating derived key
}
```

C.28.2.7 function wc_HKDF_Extract_ex

```
int wc_HKDF_Extract_ex(
    int type,
    const byte * salt,
    word32 saltSz,
    const byte * inKey,
    word32 inKeySz,
    byte * out,
    void * heap,
    int devId
)
```

This function provides access to a HMAC Key Derivation Function (HKDF). It utilizes HMAC to convert inKey, with an optional salt into a derived key, which it stores in out. The hash type defaults to MD5 if 0 or NULL is given. This is the _ex version adding heap hint and device identifier.

Parameters:

- **type** hash type to use for the HKDF. Valid types are: WC_MD5, WC_SHA, WC_SHA256, WC_SHA384, WC_SHA512, WC_SHA3_224, WC_SHA3_256, WC_SHA3_384 or WC_SHA3_512
- **salt** pointer to a buffer containing an optional salt. Use NULL instead if not using a salt
- **saltSz** length of the salt. Use 0 if not using a salt
- **inKey** pointer to the buffer containing the key to use for KDF
- **inKeySz** length of the input key
- **out** pointer to the buffer in which to store the derived key
- **heap** heap hint to use for memory. Can be NULL
- **devId** ID to use with crypto callbacks or async hardware. Set to INVALID_DEVID (-2) if not used

See:

- wc_HKDF
- wc_HKDF_Extract
- wc_HKDF_Expand
- wc_HKDF_Expand_ex

Return:

- 0 Returned upon successfully generating a key with the given inputs
- BAD_FUNC_ARG Returned if an invalid hash type is given (see type param)
- MEMORY_E Returned if there is an error allocating memory

- HMAC_MIN_KEYLEN_E May be returned when using a FIPS implementation and the key length specified is shorter than the minimum acceptable FIPS standard

The HMAC configure option is –enable-hmac (on by default) or if building sources directly HAVE_HKDF

Example

```
byte key[] = { // initialize with key };
byte salt[] = { // initialize with salt };
byte derivedKey[MAX_DIGEST_SIZE];

int ret = wc_HKDF_Extract_ex(WC_SHA512, salt, sizeof(salt), key, sizeof(key),
    derivedKey, NULL, INVALID_DEVID);
if ( ret != 0 ) {
    // error generating derived key
}
```

C.28.2.8 function wc_HKDF_Expand

```
int wc_HKDF_Expand(
    int type,
    const byte * inKey,
    word32 inKeySz,
    const byte * info,
    word32 infoSz,
    byte * out,
    word32 outSz
)
```

This function provides access to a HMAC Key Derivation Function (HKDF). It utilizes HMAC to convert inKey, with optional info into a derived key, which it stores in out. The hash type defaults to MD5 if 0 or NULL is given.

Parameters:

- **type** hash type to use for the HKDF. Valid types are: WC_MD5, WC_SHA, WC_SHA256, WC_SHA384, WC_SHA512, WC_SHA3_224, WC_SHA3_256, WC_SHA3_384 or WC_SHA3_512
- **inKey** pointer to the buffer containing the key to use for KDF
- **inKeySz** length of the input key
- **info** pointer to a buffer containing optional additional info. Use NULL if not appending extra info
- **infoSz** length of additional info. Use 0 if not using additional info
- **out** pointer to the buffer in which to store the derived key
- **outSz** space available in the output buffer to store the generated key

See:

- wc_HKDF
- wc_HKDF_Extract
- wc_HKDF_Extract_ex
- wc_HKDF_Expand_ex

Return:

- 0 Returned upon successfully generating a key with the given inputs
- BAD_FUNC_ARG Returned if an invalid hash type is given (see type param)
- MEMORY_E Returned if there is an error allocating memory
- HMAC_MIN_KEYLEN_E May be returned when using a FIPS implementation and the key length specified is shorter than the minimum acceptable FIPS standard

The HMAC configure option is –enable-hmac (on by default) or if building sources directly HAVE_HKDF

Example

```
byte key[] = { // initialize with key };
byte salt[] = { // initialize with salt };
byte derivedKey[MAX_DIGEST_SIZE];

int ret = wc_HKDF_Expand(WC_SHA512, key, sizeof(key), NULL, 0,
    derivedKey, sizeof(derivedKey));
if ( ret != 0 ) {
    // error generating derived key
}
```

C.28.2.9 function wc_HKDF_Expand_ex

```
int wc_HKDF_Expand_ex(
    int type,
    const byte * inKey,
    word32 inKeySz,
    const byte * info,
    word32 infoSz,
    byte * out,
    word32 outSz,
    void * heap,
    int devId
)
```

This function provides access to a HMAC Key Derivation Function (HKDF). It utilizes HMAC to convert inKey, with optional info into a derived key, which it stores in out. The hash type defaults to MD5 if 0 or NULL is given. This is the _ex version adding heap hint and device identifier.

Parameters:

- **type** hash type to use for the HKDF. Valid types are: WC_MD5, WC_SHA, WC_SHA256, WC_SHA384, WC_SHA512, WC_SHA3_224, WC_SHA3_256, WC_SHA3_384 or WC_SHA3_512
- **inKey** pointer to the buffer containing the key to use for KDF
- **inKeySz** length of the input key
- **info** pointer to a buffer containing optional additional info. Use NULL if not appending extra info
- **infoSz** length of additional info. Use 0 if not using additional info
- **out** pointer to the buffer in which to store the derived key
- **outSz** space available in the output buffer to store the generated key
- **heap** heap hint to use for memory. Can be NULL
- **devId** ID to use with crypto callbacks or async hardware. Set to INVALID_DEVID (-2) if not used

See:

- wc_HKDF
- wc_HKDF_Extract
- wc_HKDF_Extract_ex
- wc_HKDF_Expand

Return:

- 0 Returned upon successfully generating a key with the given inputs
- BAD_FUNC_ARG Returned if an invalid hash type is given (see type param)
- MEMORY_E Returned if there is an error allocating memory
- HMAC_MIN_KEYLEN_E May be returned when using a FIPS implementation and the key length specified is shorter than the minimum acceptable FIPS standard

The HMAC configure option is –enable-hmac (on by default) or if building sources directly HAVE_HKDF

Example

```
byte key[] = { // initialize with key };
byte salt[] = { // initialize with salt };
byte derivedKey[MAX_DIGEST_SIZE];

int ret = wc_HKDF_Expand_ex(WC_SHA512, key, sizeof(key), NULL, 0,
    derivedKey, sizeof(derivedKey), NULL, INVALID_DEVID);
if ( ret != 0 ) {
    // error generating derived key
}
```

C.28.2.10 function wc_Tls13_HKDF_Extract

```
int wc_Tls13_HKDF_Extract(
    byte * prk,
    const byte * salt,
    word32 saltLen,
    byte * ikm,
    word32 ikmLen,
    int digest
)
```

This function provides access to RFC 5869 HMAC-based Extract-and-Expand Key Derivation Function (HKDF) for TLS v1.3 key derivation.

Parameters:

- **prk** Generated pseudorandom key
- **salt** salt.
- **saltLen** length of the salt
- **ikm** pointer to putput for keying material
- **ikmLen** length of the input keying material buffer
- **digest** hash type to use for the HKDF. Valid types are: WC_SHA256, WC_SHA384 or WC_SHA512

See:

- wc_HKDF
- wc_HKDF_Extract
- wc_HKDF_Extract_ex
- wc_HKDF_Expand
- wc_Tls13_HKDF_Extract_ex

Return:

- 0 Returned upon successfully generating a key with the given inputs
- BAD_FUNC_ARG Returned if an invalid hash type is given (see type param)
- MEMORY_E Returned if there is an error allocating memory
- HMAC_MIN_KEYLEN_E May be returned when using a FIPS implementation and the key length specified is shorter than the minimum acceptable FIPS standard

Example

```
byte secret[] = { // initialize with random key };
byte salt[] = { // initialize with optional salt };
byte masterSecret[MAX_DIGEST_SIZE];

int ret = wc_Tls13_HKDF_Extract(secret, salt, sizeof(salt), 0,
```

```
    masterSecret, sizeof(masterSecret), WC_SHA512);
if ( ret != 0 ) {
    // error generating derived key
}
```

C.28.2.11 function wc_Tls13_HKDF_Extract_ex

```
int wc_Tls13_HKDF_Extract_ex(
    byte * prk,
    const byte * salt,
    word32 saltLen,
    byte * ikm,
    word32 ikmLen,
    int digest,
    void * heap,
    int devId
)
```

This function provides access to RFC 5869 HMAC-based Extract-and-Expand Key Derivation Function (HKDF) for TLS v1.3 key derivation. This is the _ex version adding heap hint and device identifier.

Parameters:

- **prk** Generated pseudorandom key
- **salt** Salt.
- **saltLen** Length of the salt
- **ikm** Pointer to output for keying material
- **ikmLen** Length of the input keying material buffer
- **digest** Hash type to use for the HKDF. Valid types are: WC_SHA256, WC_SHA384 or WC_SHA512
- **heap** Heap hint to use for memory. Can be NULL
- **devId** ID to use with crypto callbacks or async hardware. Set to INVALID_DEVID (-2) if not used

See:

- wc_HKDF
- wc_HKDF_Extract
- wc_HKDF_Extract_ex
- wc_HKDF_Expand
- wc_Tls13_HKDF_Extract

Return:

- 0 Returned upon successfully generating a key with the given inputs
- BAD_FUNC_ARG Returned if an invalid hash type is given (see type param)
- MEMORY_E Returned if there is an error allocating memory
- HMAC_MIN_KEYLEN_E May be returned when using a FIPS implementation and the key length specified is shorter than the minimum acceptable FIPS standard

Example

```
byte secret[] = { // initialize with random key };
byte salt[] = { // initialize with optional salt };
byte masterSecret[MAX_DIGEST_SIZE];

int ret = wc_Tls13_HKDF_Extract_ex(secret, salt, sizeof(salt), 0,
    masterSecret, sizeof(masterSecret), WC_SHA512, NULL, INVALID_DEVID);
if ( ret != 0 ) {
    // error generating derived key
}
```

C.28.2.12 function wc_Tls13_HKDF_Expand_Label_ex

```
int wc_Tls13_HKDF_Expand_Label_ex(
    byte * okm,
    word32 okmLen,
    const byte * prk,
    word32 prkLen,
    const byte * protocol,
    word32 protocolLen,
    const byte * label,
    word32 labelLen,
    const byte * info,
    word32 infoLen,
    int digest,
    void * heap,
    int devId
)
```

Expand data using HMAC, salt and label and info. TLS v1.3 defines this function for key derivation. This is the _ex version adding heap hint and device identifier.

Parameters:

- **okm** Generated pseudorandom key - output key material.
- **okmLen** Length of generated pseudorandom key - output key material.
- **prk** Salt - pseudo-random key.
- **prkLen** Length of the salt - pseudo-random key.
- **protocol** TLS protocol label.
- **protocolLen** Length of the TLS protocol label.
- **info** Information to expand.
- **infoLen** Length of the information.
- **digest** Hash type to use for the HKDF. Valid types are: WC_SHA256, WC_SHA384 or WC_SHA512
- **heap** Heap hint to use for memory. Can be NULL
- **devId** ID to use with crypto callbacks or async hardware. Set to INVALID_DEVID (-2) if not used

See:

- wc_HKDF
- wc_HKDF_Extract
- wc_HKDF_Extract_ex
- wc_HKDF_Expand
- wc_Tls13_HKDF_Expand_Label
- wc_Tls13_HKDF_Expand_Label_Alloc

Return:

- 0 Returned upon successfully generating a key with the given inputs
- BAD_FUNC_ARG Returned if an invalid hash type is given (see type param)
- MEMORY_E Returned if there is an error allocating memory
- HMAC_MIN_KEYLEN_E May be returned when using a FIPS implementation and the key length specified is shorter than the minimum acceptable FIPS standard

C.28.2.13 function wc_Tls13_HKDF_Expand_Label

```
int wc_Tls13_HKDF_Expand_Label(
    byte * okm,
    word32 okmLen,
    const byte * prk,
```

```
    word32 prkLen,
    const byte * protocol,
    word32 protocolLen,
    const byte * label,
    word32 labelLen,
    const byte * info,
    word32 infoLen,
    int digest
)
```

Expand data using HMAC, salt and label and info. TLS v1.3 defines this function for key derivation. This is the _ex version adding heap hint and device identifier.

Parameters:

- **okm** Generated pseudorandom key - output key material.
- **okmLen** Length of generated pseudorandom key - output key material.
- **prk** Salt - pseudo-random key.
- **prkLen** Length of the salt - pseudo-random key.
- **protocol** TLS protocol label.
- **protocolLen** Length of the TLS protocol label.
- **info** Information to expand.
- **infoLen** Length of the information.
- **digest** Hash type to use for the HKDF. Valid types are: WC_SHA256, WC_SHA384 or WC_SHA512

See:

- wc_HKDF
- wc_HKDF_Extract
- wc_HKDF_Extract_ex
- wc_HKDF_Expand
- wc_Tls13_HKDF_Expand_Label_ex
- wc_Tls13_HKDF_Expand_Label_Alloc

Return:

- 0 Returned upon successfully generating a key with the given inputs
- BAD_FUNC_ARG Returned if an invalid hash type is given (see type param)
- MEMORY_E Returned if there is an error allocating memory
- HMAC_MIN_KEYLEN_E May be returned when using a FIPS implementation and the key length specified is shorter than the minimum acceptable FIPS standard

C.28.2.14 function wc_Tls13_HKDF_Expand_Label_Alloc

```
int wc_Tls13_HKDF_Expand_Label_Alloc(
    byte * okm,
    word32 okmLen,
    const byte * prk,
    word32 prkLen,
    const byte * protocol,
    word32 protocolLen,
    const byte * label,
    word32 labelLen,
    const byte * info,
    word32 infoLen,
    int digest,
    void * heap
)
```

This functions is very similar to wc_Tls13_HKDF_Expand_Label(), but it allocates memory if the stack space usually used isn't enough. Expand data using HMAC, salt and label and info. TLS v1.3 defines this function for key derivation. This is the _ex version adding heap hint and device identifier.

Parameters:

- **okm** Generated pseudorandom key - output key material.
- **okmLen** Length of generated pseudorandom key - output key material.
- **prk** Salt - pseudo-random key.
- **prkLen** Length of the salt - pseudo-random key.
- **protocol** TLS protocol label.
- **protocolLen** Length of the TLS protocol label.
- **info** Information to expand.
- **infoLen** Length of the information.
- **digest** Hash type to use for the HKDF. Valid types are: WC_SHA256, WC_SHA384 or WC_SHA512
- **heap** Heap hint to use for memory. Can be NULL

See:

- wc_HKDF
- wc_HKDF_Extract
- wc_HKDF_Extract_ex
- wc_HKDF_Expand
- wc_Tls13_HKDF_Expand_Label
- wc_Tls13_HKDF_Expand_Label_ex

Return:

- 0 Returned upon successfully generating a key with the given inputs
- BAD_FUNC_ARG Returned if an invalid hash type is given (see type param)
- MEMORY_E Returned if there is an error allocating memory
- HMAC_MIN_KEYLEN_E May be returned when using a FIPS implementation and the key length specified is shorter than the minimum acceptable FIPS standard

C.28.3 Source code

```
int wc_HmacSetKey(Hmac* hmac, int type, const byte* key, word32 keySz);

int wc_HmacUpdate(Hmac* hmac, const byte* in, word32 sz);

int wc_HmacFinal(Hmac* hmac, byte* out);

int wolfSSL_GetHmacMaxSize(void);

int wc_HKDF(int type, const byte* inKey, word32 inKeySz,
                    const byte* salt, word32 saltSz,
                    const byte* info, word32 infoSz,
                    byte* out, word32 outSz);


int wc_HKDF_Extract(
    int type,
    const byte* salt, word32 saltSz,
    const byte* inKey, word32 inKeySz,
    byte* out);
```

```
int wc_HKDF_Extract_ex(
    int type,
    const byte* salt, word32 saltSz,
    const byte* inKey, word32 inKeySz,
    byte* out,
    void* heap, int devId);

int wc_HKDF_Expand(
    int type,
    const byte* inKey, word32 inKeySz,
    const byte* info, word32 infoSz,
    byte* out, word32 outSz);

int wc_HKDF_Expand_ex(
    int type,
    const byte* inKey, word32 inKeySz,
    const byte* info, word32 infoSz,
    byte* out, word32 outSz,
    void* heap, int devId);

int wc_Tls13_HKDF_Extract(
    byte* prk,
    const byte* salt, word32 saltLen,
    byte* ikm, word32 ikmLen, int digest);

int wc_Tls13_HKDF_Extract_ex(
    byte* prk,
    const byte* salt, word32 saltLen,
    byte* ikm, word32 ikmLen, int digest,
    void* heap, int devId);

int wc_Tls13_HKDF_Expand_Label_ex(
    byte* okm, word32 okmLen,
    const byte* prk, word32 prkLen,
    const byte* protocol, word32 protocolLen,
    const byte* label, word32 labelLen,
    const byte* info, word32 infoLen,
    int digest,
    void* heap, int devId);

int wc_Tls13_HKDF_Expand_Label(
    byte* okm, word32 okmLen,
    const byte* prk, word32 prkLen,
    const byte* protocol, word32 protocolLen,
    const byte* label, word32 labelLen,
    const byte* info, word32 infoLen,
    int digest);

int wc_Tls13_HKDF_Expand_Label_Alloc(
    byte* okm, word32 okmLen,
    const byte* prk, word32 prkLen,
    const byte* protocol, word32 protocolLen,
    const byte* label, word32 labelLen,
    const byte* info, word32 infoLen,
```

```
    int digest, void* heap);
```

C.29 dox_comments/header_files/iotsafe.h

C.29.1 Functions

	Name
int	**wolfSSL_CTX_iotsafe_enable**(WOLFSSL_CTX * ctx)This function enables the IoT-Safe support on the given context.
int	**wolfSSL_iotsafe_on**(WOLFSSL * ssl, byte privkey_id, byte ecdh_keypair_slot, byte peer_pubkey_slot, byte peer_cert_slot)This function connects the IoT-Safe TLS callbacks to the given SSL session.
int	**wolfSSL_iotsafe_on_ex except that the IDs for the IoT-SAFE slots can be passed by reference, and the length of the ID fields can be specified via the parameter “id_size”.
void	**wolfIoTSafe_SetCSIM_read_cb**(wolfSSL_IOTSafe_CSIM_read_cb rf)Associates a read callback for the AT+CSIM commands. This input function is usually associated to a read event of a UART channel communicating with the modem. The read callback associated is global and changes for all the contexts that use IoT-safe support at the same time.
void	**wolfIoTSafe_SetCSIM_write_cb**(wolfSSL_IOTSafe_CSIM_write_cb wf)Associates a write callback for the AT+CSIM commands. This output function is usually associated to a write event on a UART channel communicating with the modem. The write callback associated is global and changes for all the contexts that use IoT-safe support at the same time.
int	**wolfIoTSafe_GetRandom**(unsigned char * out, word32 sz)Generate a random buffer of given size, using the IoT-Safe function GetRandom. This function is automatically used by the wolfCrypt RNG object.
int	**wolfIoTSafe_GetCert**(uint8_t id, unsigned char * output, unsigned long sz)Import a certificate stored in a file on IoT-Safe applet, and store it locally in memory. Works with one-byte file ID field.
int	**wolfIoTSafe_GetCert_ex, except that it can be invoked with a file ID of two or more bytes.
int	**wc_iotsafe_ecc_import_public**(ecc_key * key, byte key_id)Import an ECC 256-bit public key, stored in the IoT-Safe applet, into an ecc_key object.

	Name
int	**wc_iotsafe_ecc_export_public**(ecc_key * key, byte key_id)Export an ECC 256-bit public key, from ecc_key object to a writable public-key slot into the IoT-Safe applet.
int	**wc_iotsafe_ecc_import_public_ex, except that it can be invoked with a key ID of two or more bytes.
int	**wc_iotsafe_ecc_export_private**(ecc_key * key, byte key_id)Export an ECC 256-bit key, from ecc_key object to a writable private-key slot into the IoT-Safe applet.
int	**wc_iotsafe_ecc_export_private_ex, except that it can be invoked with a key ID of two or more bytes.
int	**wc_iotsafe_ecc_sign_hash**(byte * in, word32 inlen, byte * out, word32 * outlen, byte key_id)Sign a pre-computed HASH, using a private key previously stored, or pre-provisioned, in the IoT-Safe applet.
int	**wc_iotsafe_ecc_sign_hash_ex, except that it can be invoked with a key ID of two or more bytes.
int	**wc_iotsafe_ecc_verify_hash**(byte * sig, word32 siglen, byte * hash, word32 hashlen, int * res, byte key_id)Verify an ECC signature against a pre-computed HASH, using a public key previously stored, or pre-provisioned, in the IoT-Safe applet. Result is written to res. 1 is valid, 0 is invalid. Note: Do not use the return value to test for valid. Only use res.
int	**wc_iotsafe_ecc_verify_hash_ex, except that it can be invoked with a key ID of two or more bytes.
int	**wc_iotsafe_ecc_gen_k**(byte key_id)Generate an ECC 256_bit keypair and store it in a (writable) slot into the IoT-Safe applet.

C.29.2 Functions Documentation

C.29.2.1 function wolfSSL_CTX_iotsafe_enable

```
int wolfSSL_CTX_iotsafe_enable(
    WOLFSSL_CTX * ctx
)
```

This function enables the IoT-Safe support on the given context.

Parameters:

- **ctx** pointer to the WOLFSSL_CTX object on which the IoT-safe support must be enabled

See:

- wolfSSL_iotsafe_on
- wolfIoTSafe_SetCSIM_read_cb

- wolfIoTSafe_SetCSIM_write_cb

Return:

- 0 on success
- WC_HW_E on hardware error

Example

```
WOLFSSL_CTX *ctx;
ctx = wolfSSL_CTX_new(wolfTLSv1_2_client_method());
if (!ctx)
    return NULL;
wolfSSL_CTX_iotsafe_enable(ctx);
```

C.29.2.2 function wolfSSL_iotsafe_on

```
int wolfSSL_iotsafe_on(
    WOLFSSL * ssl,
    byte privkey_id,
    byte ecdh_keypair_slot,
    byte peer_pubkey_slot,
    byte peer_cert_slot
)
```

This function connects the IoT-Safe TLS callbacks to the given SSL session.

Parameters:

- **ssl** pointer to the WOLFSSL object where the callbacks will be enabled
- **privkey_id** id of the iot-safe applet slot containing the private key for the host
- **ecdh_keypair_slot** id of the iot-safe applet slot to store the ECDH keypair
- **peer_pubkey_slot** id of the iot-safe applet slot to store the other endpoint's public key for ECDH
- **peer_cert_slot** id of the iot-safe applet slot to store the other endpoint's public key for verification

See:

- wolfSSL_iotsafe_on_ex
- wolfSSL_CTX_iotsafe_enable

Return:

- 0 upon success
- NOT_COMPILED_IN if HAVE_PK_CALLBACKS is disabled
- BAD_FUNC_ARG if the ssl pointer is invalid

This should be called to connect a SSL session to IoT-Safe applet when the ID of the slots are one-byte long. If IoT-SAFE slots have an ID of two or more bytes, wolfSSL_iotsafe_on_ex() should be used instead.

Example

```
// Define key ids for IoT-Safe
#define PRIVKEY_ID 0x02
#define ECDH_KEYPAIR_ID 0x03
#define PEER_PUBKEY_ID 0x04
#define PEER_CERT_ID 0x05
// Create new ssl session
WOLFSSL *ssl;
ssl = wolfSSL_new(ctx);
if (!ssl)
```

```
    return NULL;
// Enable IoT-Safe and associate key slots
ret = wolfSSL_CTX_iotsafe_enable(ctx);
if (ret == 0) {
    ret = wolfSSL_iotsafe_on(ssl, PRIVKEY_ID, ECDH_KEYPAIR_ID, PEER_PUBKEY_ID,
 ↪  PEER_CERT_ID);
}
```

C.29.2.3 function wolfSSL_iotsafe_on_ex

```
int wolfSSL_iotsafe_on_ex(
    WOLFSSL * ssl,
    byte * privkey_id,
    byte * ecdh_keypair_slot,
    byte * peer_pubkey_slot,
    byte * peer_cert_slot,
    word16 id_size
)
```

This function connects the IoT-Safe TLS callbacks to the given SSL session. This is equivalent to wolfSSL_iotsafe_on except that the IDs for the IoT-SAFE slots can be passed by reference, and the length of the ID fields can be specified via the parameter "id_size".

Parameters:

- **ssl** pointer to the WOLFSSL object where the callbacks will be enabled
- **privkey_id** pointer to the id of the iot-safe applet slot containing the private key for the host
- **ecdh_keypair_slot** pointer to the id of the iot-safe applet slot to store the ECDH keypair
- **peer_pubkey_slot** pointer to the of id the iot-safe applet slot to store the other endpoint's public key for ECDH
- **peer_cert_slot** pointer to the id of the iot-safe applet slot to store the other endpoint's public key for verification
- **id_size** size of each slot ID

See:

- wolfSSL_iotsafe_on
- wolfSSL_CTX_iotsafe_enable

Return:

- 0 upon success
- NOT_COMPILED_IN if HAVE_PK_CALLBACKS is disabled
- BAD_FUNC_ARG if the ssl pointer is invalid

Example

```
// Define key ids for IoT-Safe (16 bit, little endian)
#define PRIVKEY_ID 0x0201
#define ECDH_KEYPAIR_ID 0x0301
#define PEER_PUBKEY_ID 0x0401
#define PEER_CERT_ID 0x0501
#define ID_SIZE (sizeof(word16))

word16 privkey = PRIVKEY_ID,
         ecdh_keypair = ECDH_KEYPAIR_ID,
         peer_pubkey = PEER_PUBKEY_ID,
         peer_cert = PEER_CERT_ID;
```

```
// Create new ssl session
WOLFSSL *ssl;
ssl = wolfSSL_new(ctx);
if (!ssl)
    return NULL;
// Enable IoT-Safe and associate key slots
ret = wolfSSL_CTX_iotsafe_enable(ctx);
if (ret == 0) {
    ret = wolfSSL_CTX_iotsafe_on_ex(ssl, &privkey, &ecdh_keypair, &peer_pubkey,
 ↪  &peer_cert, ID_SIZE);
}
```

C.29.2.4 function wolfIoTSafe_SetCSIM_read_cb

```
void wolfIoTSafe_SetCSIM_read_cb(
    wolfSSL_IOTSafe_CSIM_read_cb rf
)
```

Associates a read callback for the AT+CSIM commands. This input function is usually associated to a read event of a UART channel communicating with the modem. The read callback associated is global and changes for all the contexts that use IoT-safe support at the same time.

Parameters:

- **rf** Read callback associated to a UART read event. The callback function takes two arguments (buf, len) and return the number of characters read, up to len. When a newline is encountered, the callback should return the number of characters received so far, including the newline character.

See: wolfIoTSafe_SetCSIM_write_cb

Example

```
// USART read function, defined elsewhere
int usart_read(char *buf, int len);

wolfIoTSafe_SetCSIM_read_cb(usart_read);
```

C.29.2.5 function wolfIoTSafe_SetCSIM_write_cb

```
void wolfIoTSafe_SetCSIM_write_cb(
    wolfSSL_IOTSafe_CSIM_write_cb wf
)
```

Associates a write callback for the AT+CSIM commands. This output function is usually associated to a write event on a UART channel communicating with the modem. The write callback associated is global and changes for all the contexts that use IoT-safe support at the same time.

Parameters:

- **rf** Write callback associated to a UART write event. The callback function takes two arguments (buf, len) and return the number of characters written, up to len.

See: wolfIoTSafe_SetCSIM_read_cb

Example

```
// USART write function, defined elsewhere
int usart_write(const char *buf, int len);
wolfIoTSafe_SetCSIM_write_cb(usart_write);
```

C.29.2.6 function wolfIoTSafe_GetRandom

```
int wolfIoTSafe_GetRandom(
    unsigned char * out,
    word32 sz
)
```

Generate a random buffer of given size, using the IoT-Safe function GetRandom. This function is automatically used by the wolfCrypt RNG object.

Parameters:

- **out** the buffer where the random sequence of bytes is stored.
- **sz** the size of the random sequence to generate, in bytes

Return: 0 upon success

C.29.2.7 function wolfIoTSafe_GetCert

```
int wolfIoTSafe_GetCert(
    uint8_t id,
    unsigned char * output,
    unsigned long sz
)
```

Import a certificate stored in a file on IoT-Safe applet, and store it locally in memory. Works with one-byte file ID field.

Parameters:

- **id** The file id in the IoT-Safe applet where the certificate is stored
- **output** the buffer where the certificate will be imported
- **sz** the maximum size available in the buffer output

Return:

- the length of the certificate imported
- < 0 in case of failure

Example

```
#define CRT_CLIENT_FILE_ID 0x03
unsigned char cert_buffer[2048];
// Get the certificate into the buffer
cert_buffer_size = wolfIoTSafe_GetCert(CRT_CLIENT_FILE_ID, cert_buffer, 2048);
if (cert_buffer_size < 1) {
    printf("Bad cli cert\n");
    return -1;
}
printf("Loaded Client certificate from IoT-Safe, size = %lu\n",
 ↪  cert_buffer_size);

// Use the certificate buffer as identity for the TLS client context
if (wolfSSL_CTX_use_certificate_buffer(cli_ctx, cert_buffer,
            cert_buffer_size, SSL_FILETYPE_ASN1) != SSL_SUCCESS) {
    printf("Cannot load client cert\n");
```

```
    return -1;
}
printf("Client certificate successfully imported.\n");
```

C.29.2.8 function wolfIoTSafe_GetCert_ex

```
int wolfIoTSafe_GetCert_ex(
    uint8_t * id,
    uint16_t id_sz,
    unsigned char * output,
    unsigned long sz
)
```

Import a certificate stored in a file on IoT-Safe applet, and store it locally in memory. Equivalent to wolfIoTSafe_GetCert, except that it can be invoked with a file ID of two or more bytes.

Parameters:

- **id** Pointer to the file id in the IoT-Safe applet where the certificate is stored
- **id_sz** Size of the file id in bytes
- **output** the buffer where the certificate will be imported
- **sz** the maximum size available in the buffer output

Return:

- the length of the certificate imported
- < 0 in case of failure

Example

```
#define CRT_CLIENT_FILE_ID 0x0302
#define ID_SIZE (sizeof(word16))
unsigned char cert_buffer[2048];
word16 client_file_id = CRT_CLIENT_FILE_ID;



// Get the certificate into the buffer
cert_buffer_size = wolfIoTSafe_GetCert_ex(&client_file_id, ID_SIZE,
 ↪  cert_buffer, 2048);
if (cert_buffer_size < 1) {
    printf("Bad cli cert\n");
    return -1;
}
printf("Loaded Client certificate from IoT-Safe, size = %lu\n",
 ↪  cert_buffer_size);

// Use the certificate buffer as identity for the TLS client context
if (wolfSSL_CTX_use_certificate_buffer(cli_ctx, cert_buffer,
            cert_buffer_size, SSL_FILETYPE_ASN1) != SSL_SUCCESS) {
    printf("Cannot load client cert\n");
    return -1;
}
printf("Client certificate successfully imported.\n");
```

C.29.2.9 function wc_iotsafe_ecc_import_public

```
int wc_iotsafe_ecc_import_public(
    ecc_key * key,
    byte key_id
)
```

Import an ECC 256-bit public key, stored in the IoT-Safe applet, into an ecc_key object.

Parameters:

- **key** the ecc_key object that will contain the key imported from the IoT-Safe applet
- **id** The key id in the IoT-Safe applet where the public key is stored

See:

- wc_iotsafe_ecc_export_public
- wc_iotsafe_ecc_export_private

Return:

- 0 upon success
- < 0 in case of failure

C.29.2.10 function wc_iotsafe_ecc_export_public

```
int wc_iotsafe_ecc_export_public(
    ecc_key * key,
    byte key_id
)
```

Export an ECC 256-bit public key, from ecc_key object to a writable public-key slot into the IoT-Safe applet.

Parameters:

- **key** the ecc_key object containing the key to be exported
- **id** The key id in the IoT-Safe applet where the public key will be stored

See:

- wc_iotsafe_ecc_import_public_ex
- wc_iotsafe_ecc_export_private

Return:

- 0 upon success
- < 0 in case of failure

C.29.2.11 function wc_iotsafe_ecc_import_public_ex

```
int wc_iotsafe_ecc_import_public_ex(
    ecc_key * key,
    byte * key_id,
    word16 id_size
)
```

Export an ECC 256-bit public key, from ecc_key object to a writable public-key slot into the IoT-Safe applet. Equivalent to wc_iotsafe_ecc_import_public, except that it can be invoked with a key ID of two or more bytes.

Parameters:

- **key** the ecc_key object containing the key to be exported
- **id** The pointer to the key id in the IoT-Safe applet where the public key will be stored

- **id_size** The key id size

See:

- wc_iotsafe_ecc_import_public
- wc_iotsafe_ecc_export_private

Return:

- 0 upon success
- < 0 in case of failure

C.29.2.12 function wc_iotsafe_ecc_export_private

```
int wc_iotsafe_ecc_export_private(
    ecc_key * key,
    byte key_id
)
```

Export an ECC 256-bit key, from ecc_key object to a writable private-key slot into the IoT-Safe applet.

Parameters:

- **key** the ecc_key object containing the key to be exported
- **id** The key id in the IoT-Safe applet where the private key will be stored

See:

- wc_iotsafe_ecc_export_private_ex
- wc_iotsafe_ecc_import_public
- wc_iotsafe_ecc_export_public

Return:

- 0 upon success
- < 0 in case of failure

C.29.2.13 function wc_iotsafe_ecc_export_private_ex

```
int wc_iotsafe_ecc_export_private_ex(
    ecc_key * key,
    byte * key_id,
    word16 id_size
)
```

Export an ECC 256-bit key, from ecc_key object to a writable private-key slot into the IoT-Safe applet. Equivalent to wc_iotsafe_ecc_export_private, except that it can be invoked with a key ID of two or more bytes.

Parameters:

- **key** the ecc_key object containing the key to be exported
- **id** The pointer to the key id in the IoT-Safe applet where the private key will be stored
- **id_size** The key id size

See:

- wc_iotsafe_ecc_export_private
- wc_iotsafe_ecc_import_public
- wc_iotsafe_ecc_export_public

Return:

- 0 upon success
- < 0 in case of failure

C.29.2.14 function wc_iotsafe_ecc_sign_hash

```
int wc_iotsafe_ecc_sign_hash(
    byte * in,
    word32 inlen,
    byte * out,
    word32 * outlen,
    byte key_id
)
```

Sign a pre-computed HASH, using a private key previously stored, or pre-provisioned, in the IoT-Safe applet.

Parameters:

- **in** pointer to the buffer containing the message hash to sign
- **inlen** length of the message hash to sign
- **out** buffer in which to store the generated signature
- **outlen** max length of the output buffer. Will store the bytes
- **id** key id in the IoT-Safe applet for the slot containing the private key to sign the payload written to out upon successfully generating a message signature

See:

- wc_iotsafe_ecc_sign_hash_ex
- wc_iotsafe_ecc_verify_hash
- wc_iotsafe_ecc_gen_k

Return:

- 0 upon success
- < 0 in case of failure

C.29.2.15 function wc_iotsafe_ecc_sign_hash_ex

```
int wc_iotsafe_ecc_sign_hash_ex(
    byte * in,
    word32 inlen,
    byte * out,
    word32 * outlen,
    byte * key_id,
    word16 id_size
)
```

Sign a pre-computed HASH, using a private key previously stored, or pre-provisioned, in the IoT-Safe applet. Equivalent to wc_iotsafe_ecc_sign_hash, except that it can be invoked with a key ID of two or more bytes.

Parameters:

- **in** pointer to the buffer containing the message hash to sign
- **inlen** length of the message hash to sign
- **out** buffer in which to store the generated signature
- **outlen** max length of the output buffer. Will store the bytes
- **id** pointer to a key id in the IoT-Safe applet for the slot containing the private key to sign the payload written to out upon successfully generating a message signature

- **id_size** The key id size

See:

- wc_iotsafe_ecc_sign_hash
- wc_iotsafe_ecc_verify_hash
- wc_iotsafe_ecc_gen_k

Return:

- 0 upon success
- < 0 in case of failure

C.29.2.16 function wc_iotsafe_ecc_verify_hash

```
int wc_iotsafe_ecc_verify_hash(
    byte * sig,
    word32 siglen,
    byte * hash,
    word32 hashlen,
    int * res,
    byte key_id
)
```

Verify an ECC signature against a pre-computed HASH, using a public key previously stored, or pre-provisioned, in the IoT-Safe applet. Result is written to res. 1 is valid, 0 is invalid. Note: Do not use the return value to test for valid. Only use res.

Parameters:

- **sig** buffer containing the signature to verify
- **hash** The hash (message digest) that was signed
- **hashlen** The length of the hash (octets)
- **res** Result of signature, 1==valid, 0==invalid
- **key_id** The id of the slot where the public ECC key is stored in the IoT-Safe applet

See:

- wc_iotsafe_ecc_verify_hash_ex
- wc_iotsafe_ecc_sign_hash
- wc_iotsafe_ecc_gen_k

Return:

- 0 upon success (even if the signature is not valid)
- < 0 in case of failure.

C.29.2.17 function wc_iotsafe_ecc_verify_hash_ex

```
int wc_iotsafe_ecc_verify_hash_ex(
    byte * sig,
    word32 siglen,
    byte * hash,
    word32 hashlen,
    int * res,
    byte * key_id,
    word16 id_size
)
```

Verify an ECC signature against a pre-computed HASH, using a public key previously stored, or pre-provisioned, in the IoT-Safe applet. Result is written to res. 1 is valid, 0 is invalid. Note: Do not use the return value to test for valid. Only use res. Equivalent to wc_iotsafe_ecc_verify_hash, except that it can be invoked with a key ID of two or more bytes.

Parameters:

- **sig** buffer containing the signature to verify
- **hash** The hash (message digest) that was signed
- **hashlen** The length of the hash (octets)
- **res** Result of signature, 1==valid, 0==invalid
- **key_id** The id of the slot where the public ECC key is stored in the IoT-Safe applet
- **id_size** The key id size

See:

- wc_iotsafe_ecc_verify_hash
- wc_iotsafe_ecc_sign_hash
- wc_iotsafe_ecc_gen_k

Return:

- 0 upon success (even if the signature is not valid)
- < 0 in case of failure.

C.29.2.18 function wc_iotsafe_ecc_gen_k

```
int wc_iotsafe_ecc_gen_k(
    byte key_id
)
```

Generate an ECC 256-bit keypair and store it in a (writable) slot into the IoT-Safe applet.

Parameters:

- **key_id** The id of the slot where the ECC key pair is stored in the IoT-Safe applet.
- **key_id** The id of the slot where the ECC key pair is stored in the IoT-Safe applet.
- **id_size** The key id size

See:

- wc_iotsafe_ecc_gen_k_ex
- wc_iotsafe_ecc_sign_hash
- wc_iotsafe_ecc_verify_hash
- wc_iotsafe_ecc_gen_k
- wc_iotsafe_ecc_sign_hash_ex
- wc_iotsafe_ecc_verify_hash_ex

Return:

- 0 upon success
- < 0 in case of failure.
- 0 upon success
- < 0 in case of failure.

Generate an ECC 256-bit keypair and store it in a (writable) slot into the IoT-Safe applet. Equivalent to wc_iotsafe_ecc_gen_k, except that it can be invoked with a key ID of two or more bytes.

C.29.3 Source code

```
int wolfSSL_CTX_iotsafe_enable(WOLFSSL_CTX *ctx);


int wolfSSL_iotsafe_on(WOLFSSL *ssl, byte privkey_id,
       byte ecdh_keypair_slot, byte peer_pubkey_slot, byte peer_cert_slot);


int wolfSSL_iotsafe_on_ex(WOLFSSL *ssl, byte *privkey_id,
       byte *ecdh_keypair_slot, byte *peer_pubkey_slot, byte *peer_cert_slot,
 ↪  word16 id_size);


void wolfIoTSafe_SetCSIM_read_cb(wolfSSL_IOTSafe_CSIM_read_cb rf);

void wolfIoTSafe_SetCSIM_write_cb(wolfSSL_IOTSafe_CSIM_write_cb wf);



int wolfIoTSafe_GetRandom(unsigned char* out, word32 sz);


int wolfIoTSafe_GetCert(uint8_t id, unsigned char *output, unsigned long sz);


int wolfIoTSafe_GetCert_ex(uint8_t *id, uint16_t id_sz, unsigned char *output,
 ↪  unsigned long sz);

int wc_iotsafe_ecc_import_public(ecc_key *key, byte key_id);

int wc_iotsafe_ecc_export_public(ecc_key *key, byte key_id);


int wc_iotsafe_ecc_import_public_ex(ecc_key *key, byte *key_id, word16
 ↪  id_size);

int wc_iotsafe_ecc_export_private(ecc_key *key, byte key_id);

int wc_iotsafe_ecc_export_private_ex(ecc_key *key, byte *key_id, word16
 ↪  id_size);

int wc_iotsafe_ecc_sign_hash(byte *in, word32 inlen, byte *out, word32 *outlen,
 ↪  byte key_id);

int wc_iotsafe_ecc_sign_hash_ex(byte *in, word32 inlen, byte *out, word32
 ↪  *outlen, byte *key_id, word16 id_size);

int wc_iotsafe_ecc_verify_hash(byte *sig, word32 siglen, byte *hash, word32
 ↪  hashlen, int *res, byte key_id);

int wc_iotsafe_ecc_verify_hash_ex(byte *sig, word32 siglen, byte *hash, word32
 ↪  hashlen, int *res, byte *key_id, word16 id_size);
```

```
int wc_iotsafe_ecc_gen_k(byte key_id);
```


```
int wc_iotsafe_ecc_gen_k(byte key_id);
```


C.30 dox_comments/header_files/logging.h

C.30.1 Functions

	Name
int	**wolfSSL_SetLoggingCb**(wolfSSL_Logging_cb log_function)This function registers a logging callback that will be used to handle the wolfSSL log message. By default, if the system supports it fprintf() to stderr is used but by using this function anything can be done by the user.
int	**wolfSSL_Debugging_ON**(void)If logging has been enabled at build time this function turns on logging at runtime. To enable logging at build time use –enable-debug or define DEBUG_WOLFSSL.
void	**wolfSSL_Debugging_OFF**(void)This function turns off runtime logging messages. If they're already off, no action is taken.

C.30.2 Functions Documentation

C.30.2.1 function wolfSSL_SetLoggingCb

```
int wolfSSL_SetLoggingCb(
    wolfSSL_Logging_cb log_function
)
```

This function registers a logging callback that will be used to handle the wolfSSL log message. By default, if the system supports it fprintf() to stderr is used but by using this function anything can be done by the user.

Parameters:

- **log_function** function to register as a logging callback. Function signature must follow the above prototype.

See:

- wolfSSL_Debugging_ON
- wolfSSL_Debugging_OFF

Return:

- Success If successful this function will return 0.
- BAD_FUNC_ARG is the error that will be returned if a function pointer is not provided.

Example

```
int ret = 0;
// Logging callback prototype
void MyLoggingCallback(const int logLevel, const char* const logMessage);
```

```
// Register the custom logging callback with wolfSSL
ret = wolfSSL_SetLoggingCb(MyLoggingCallback);
if (ret != 0) {
    // failed to set logging callback
}
void MyLoggingCallback(const int logLevel, const char* const logMessage)
{
// custom logging function
}
```

C.30.2.2 function wolfSSL_Debugging_ON

```
int wolfSSL_Debugging_ON(
    void
)
```

If logging has been enabled at build time this function turns on logging at runtime. To enable logging at build time use –enable-debug or define DEBUG_WOLFSSL.

Parameters:

- **none** No parameters.

See:

- wolfSSL_Debugging_OFF
- wolfSSL_SetLoggingCb

Return:

- 0 upon success.
- NOT_COMPILED_IN is the error that will be returned if logging isn't enabled for this build.

Example

```
wolfSSL_Debugging_ON();
```

C.30.2.3 function wolfSSL_Debugging_OFF

```
void wolfSSL_Debugging_OFF(
    void
)
```

This function turns off runtime logging messages. If they're already off, no action is taken.

Parameters:

- **none** No parameters.

See:

- wolfSSL_Debugging_ON
- wolfSSL_SetLoggingCb

Return: none No returns.

Example

```
wolfSSL_Debugging_OFF();
```

C.30.3 Source code

```
int wolfSSL_SetLoggingCb(wolfSSL_Logging_cb log_function);

int  wolfSSL_Debugging_ON(void);

void wolfSSL_Debugging_OFF(void);
```

C.31 dox_comments/header_files/md2.h

C.31.1 Functions

	Name
void	**wc_InitMd2**(Md2 *)This function initializes md2. This is automatically called by wc_Md2Hash.
void	**wc_Md2Update**(Md2 * md2, const byte * data, word32 len)Can be called to continually hash the provided byte array of length len.
void	**wc_Md2Final**(Md2 * md2, byte * hash)Finalizes hashing of data. Result is placed into hash.
int	**wc_Md2Hash**(const byte * data, word32 len, byte * hash)Convenience function, handles all the hashing and places the result into hash.

C.31.2 Functions Documentation

C.31.2.1 function wc_InitMd2

```
void wc_InitMd2(
    Md2 *
)
```

This function initializes md2. This is automatically called by wc_Md2Hash.

Parameters:

- **md2** pointer to the md2 structure to use for encryption

See:

- wc_Md2Hash
- wc_Md2Update
- wc_Md2Final

Return: 0 Returned upon successfully initializing

Example

```
md2 md2[1];
if ((ret = wc_InitMd2(md2)) != 0) {
   WOLFSSL_MSG("wc_Initmd2 failed");
}
else {
   wc_Md2Update(md2, data, len);
```

```
   wc_Md2Final(md2, hash);
}
```

C.31.2.2 function wc_Md2Update

```
void wc_Md2Update(
    Md2 * md2,
    const byte * data,
    word32 len
)
```

Can be called to continually hash the provided byte array of length len.

Parameters:

- **md2** pointer to the md2 structure to use for encryption
- **data** the data to be hashed
- **len** length of data to be hashed

See:

- wc_Md2Hash
- wc_Md2Final
- wc_InitMd2

Return: 0 Returned upon successfully adding the data to the digest.

Example

```
md2 md2[1];
byte data[] = { }; // Data to be hashed
word32 len = sizeof(data);

if ((ret = wc_InitMd2(md2)) != 0) {
   WOLFSSL_MSG("wc_Initmd2 failed");
}
else {
   wc_Md2Update(md2, data, len);
   wc_Md2Final(md2, hash);
}
```

C.31.2.3 function wc_Md2Final

```
void wc_Md2Final(
    Md2 * md2,
    byte * hash
)
```

Finalizes hashing of data. Result is placed into hash.

Parameters:

- **md2** pointer to the md2 structure to use for encryption
- **hash** Byte array to hold hash value.

See:

- wc_Md2Hash
- wc_Md2Final
- wc_InitMd2

Return: 0 Returned upon successfully finalizing.

Example

```
md2 md2[1];
byte data[] = { }; // Data to be hashed
word32 len = sizeof(data);

if ((ret = wc_InitMd2(md2)) != 0) {
   WOLFSSL_MSG("wc_Initmd2 failed");
}
else {
   wc_Md2Update(md2, data, len);
   wc_Md2Final(md2, hash);
}
```

C.31.2.4 function wc_Md2Hash

```
int wc_Md2Hash(
    const byte * data,
    word32 len,
    byte * hash
)
```

Convenience function, handles all the hashing and places the result into hash.

Parameters:

- **data** the data to hash
- **len** the length of data
- **hash** Byte array to hold hash value.

See:

- wc_Md2Hash
- wc_Md2Final
- wc_InitMd2

Return:

- 0 Returned upon successfully hashing the data.
- Memory_E memory error, unable to allocate memory. This is only possible with the small stack option enabled.

Example

none

C.31.3 Source code

```
void wc_InitMd2(Md2*);

void wc_Md2Update(Md2* md2, const byte* data, word32 len);

void wc_Md2Final(Md2* md2, byte* hash);

int  wc_Md2Hash(const byte* data, word32 len, byte* hash);
```

C.32 dox_comments/header_files/md4.h

C.32.1 Functions

	Name
void	**wc_InitMd4**(Md4 *)This function initializes md4. This is automatically called by wc_Md4Hash.
void	**wc_Md4Update**(Md4 * md4, const byte * data, word32 len)Can be called to continually hash the provided byte array of length len.
void	**wc_Md4Final**(Md4 * md4, byte * hash)Finalizes hashing of data. Result is placed into hash.

C.32.2 Functions Documentation

C.32.2.1 function wc_InitMd4

```
void wc_InitMd4(
    Md4 *
)
```

This function initializes md4. This is automatically called by wc_Md4Hash.

Parameters:

- **md4** pointer to the md4 structure to use for encryption

See:

- wc_Md4Hash
- wc_Md4Update
- wc_Md4Final

Return: 0 Returned upon successfully initializing

Example

```
md4 md4[1];
if ((ret = wc_InitMd4(md4)) != 0) {
   WOLFSSL_MSG("wc_Initmd4 failed");
}
else {
   wc_Md4Update(md4, data, len);
   wc_Md4Final(md4, hash);
}
```

C.32.2.2 function wc_Md4Update

```
void wc_Md4Update(
    Md4 * md4,
    const byte * data,
    word32 len
)
```

Can be called to continually hash the provided byte array of length len.

Parameters:

- **md4** pointer to the md4 structure to use for encryption
- **data** the data to be hashed
- **len** length of data to be hashed

See:

- wc_Md4Hash
- wc_Md4Final
- wc_InitMd4

Return: 0 Returned upon successfully adding the data to the digest.

Example

```
md4 md4[1];
byte data[] = { }; // Data to be hashed
word32 len = sizeof(data);

if ((ret = wc_InitMd4(md4)) != 0) {
   WOLFSSL_MSG("wc_Initmd4 failed");
}
else {
   wc_Md4Update(md4, data, len);
   wc_Md4Final(md4, hash);
}
```

C.32.2.3 function wc_Md4Final

```
void wc_Md4Final(
    Md4 * md4,
    byte * hash
)
```

Finalizes hashing of data. Result is placed into hash.

Parameters:

- **md4** pointer to the md4 structure to use for encryption
- **hash** Byte array to hold hash value.

See:

- wc_Md4Hash
- wc_Md4Final
- wc_InitMd4

Return: 0 Returned upon successfully finalizing.

Example

```
md4 md4[1];
if ((ret = wc_InitMd4(md4)) != 0) {
    WOLFSSL_MSG("wc_Initmd4 failed");
}
else {
    wc_Md4Update(md4, data, len);
    wc_Md4Final(md4, hash);
}
```

C.32.3 Source code

```
void wc_InitMd4(Md4*);

void wc_Md4Update(Md4* md4, const byte* data, word32 len);

void wc_Md4Final(Md4* md4, byte* hash);
```

C.33 dox_comments/header_files/md5.h

C.33.1 Functions

	Name
int	**wc_InitMd5**(wc_Md5 *)This function initializes md5. This is automatically called by wc_Md5Hash.
int	**wc_Md5Update**(wc_Md5 * md5, const byte * data, word32 len)Can be called to continually hash the provided byte array of length len.
int	**wc_Md5Final**(wc_Md5 * md5, byte * hash)Finalizes hashing of data. Result is placed into hash. Md5 Struct is reset. Note: This function will also return the result of calling IntelQaSymMd5() in the case that HAVE_INTEL_QA is defined.
void	**wc_Md5Free**(wc_Md5 *)Resets the Md5 structure. Note: this is only supported if you have WOLFSSL_TI_HASH defined.
int	**wc_Md5GetHash**(wc_Md5 * md5, byte * hash)Gets hash data. Result is placed into hash. Md5 struct is not reset.

C.33.2 Functions Documentation

C.33.2.1 function wc_InitMd5

```
int wc_InitMd5(
    wc_Md5 *
)
```

This function initializes md5. This is automatically called by wc_Md5Hash.

Parameters:

- **md5** pointer to the md5 structure to use for encryption

See:

- wc_Md5Hash
- wc_Md5Update
- wc_Md5Final

Return:

- 0 Returned upon successfully initializing.

- BAD_FUNC_ARG Returned if the Md5 structure is passed as a NULL value.

Example

```
Md5 md5;
byte* hash;
if ((ret = wc_InitMd5(&md5)) != 0) {
   WOLFSSL_MSG("wc_Initmd5 failed");
}
else {
   ret = wc_Md5Update(&md5, data, len);
   if (ret != 0) {
     // Md5 Update Failure Case.
   }
   ret = wc_Md5Final(&md5, hash);
  if (ret != 0) {
    // Md5 Final Failure Case.
  }
}
```

C.33.2.2 function wc_Md5Update

```
int wc_Md5Update(
    wc_Md5 * md5,
    const byte * data,
    word32 len
)
```

Can be called to continually hash the provided byte array of length len.

Parameters:

- **md5** pointer to the md5 structure to use for encryption
- **data** the data to be hashed
- **len** length of data to be hashed

See:

- wc_Md5Hash
- wc_Md5Final
- wc_InitMd5

Return:

- 0 Returned upon successfully adding the data to the digest.
- BAD_FUNC_ARG Returned if the Md5 structure is NULL or if data is NULL and len is greater than zero. The function should not return an error if the data parameter is NULL and len is zero.

Example

```
Md5 md5;
byte data[] = { Data to be hashed };
word32 len = sizeof(data);

if ((ret = wc_InitMd5(&md5)) != 0) {
   WOLFSSL_MSG("wc_Initmd5 failed");
}
else {
   ret = wc_Md5Update(&md5, data, len);
   if (ret != 0) {
```

```
     // Md5 Update Error Case.
   }
   ret = wc_Md5Final(&md5, hash);
   if (ret != 0) {
    // Md5 Final Error Case.
   }
}
```

C.33.2.3 function wc_Md5Final

```
int wc_Md5Final(
    wc_Md5 * md5,
    byte * hash
)
```

Finalizes hashing of data. Result is placed into hash. Md5 Struct is reset. Note: This function will also return the result of calling IntelQaSymMd5() in the case that HAVE_INTEL_QA is defined.

Parameters:

- **md5** pointer to the md5 structure to use for encryption
- **hash** Byte array to hold hash value.

See:

- wc_Md5Hash
- wc_InitMd5
- wc_Md5GetHash

Return:

- 0 Returned upon successfully finalizing.
- BAD_FUNC_ARG Returned if the Md5 structure or hash pointer is passed in NULL.

Example

```
md5 md5[1];
byte data[] = { Data to be hashed };
word32 len = sizeof(data);

if ((ret = wc_InitMd5(md5)) != 0) {
   WOLFSSL_MSG("wc_Initmd5 failed");
}
else {
   ret = wc_Md5Update(md5, data, len);
   if (ret != 0) {
    // Md5 Update Failure Case.
   }
  ret = wc_Md5Final(md5, hash);
   if (ret != 0) {
    // Md5 Final Failure Case.
   }
}
```

C.33.2.4 function wc_Md5Free

```
void wc_Md5Free(
    wc_Md5 *
)
```

Resets the Md5 structure. Note: this is only supported if you have WOLFSSL_TI_HASH defined.

Parameters:

- **md5** Pointer to the Md5 structure to be reset.

See:

- wc_InitMd5
- wc_Md5Update
- wc_Md5Final

Return: none No returns.

Example

```
Md5 md5;
byte data[] = { Data to be hashed };
word32 len = sizeof(data);

if ((ret = wc_InitMd5(&md5)) != 0) {
    WOLFSSL_MSG("wc_InitMd5 failed");
}
else {
    wc_Md5Update(&md5, data, len);
    wc_Md5Final(&md5, hash);
    wc_Md5Free(&md5);
}
```

C.33.2.5 function wc_Md5GetHash

```
int wc_Md5GetHash(
    wc_Md5 * md5,
    byte * hash
)
```

Gets hash data. Result is placed into hash. Md5 struct is not reset.

Parameters:

- **md5** pointer to the md5 structure to use for encryption.
- **hash** Byte array to hold hash value.

See:

- wc_Md5Hash
- wc_Md5Final
- wc_InitMd5

Return: none No returns

Example

```
md5 md5[1];
if ((ret = wc_InitMd5(md5)) != 0) {
   WOLFSSL_MSG("wc_Initmd5 failed");
}
else {
   wc_Md5Update(md5, data, len);
   wc_Md5GetHash(md5, hash);
}
```

C.33.3 Source code

```
int wc_InitMd5(wc_Md5*);

int wc_Md5Update(wc_Md5* md5, const byte* data, word32 len);

int wc_Md5Final(wc_Md5* md5, byte* hash);

void wc_Md5Free(wc_Md5*);

int  wc_Md5GetHash(wc_Md5* md5, byte* hash);
```

C.34 dox_comments/header_files/memory.h

C.34.1 Functions

	Name
void *	**wolfSSL_Malloc. Note wolfSSL_Malloc is not called directly by wolfSSL, but instead called by macro XMALLOC. For the default build only the size argument exists. If using WOLFSSL_STATIC_MEMORY build then heap and type arguments are included.
void	**wolfSSL_Free. Note wolfSSL_Free is not called directly by wolfSSL, but instead called by macro XFREE. For the default build only the ptr argument exists. If using WOLFSSL_STATIC_MEMORY build then heap and type arguments are included.
void *	**wolfSSL_Realloc. Note wolfSSL_Realloc is not called directly by wolfSSL, but instead called by macro XREALLOC. For the default build only the size argument exists. If using WOLFSSL_STATIC_MEMORY build then heap and type arguments are included.
int	**wolfSSL_SetAllocators**(wolfSSL_Malloc_cb , wolfSSL_Free_cb , wolfSSL_Realloc_cb)This function registers the allocation functions used by wolfSSL. By default, if the system supports it, malloc/free and realloc are used. Using this function allows the user at runtime to install their own memory handlers.
int	**wolfSSL_StaticBufferSz**(byte * buffer, word32 sz, int flag)This function is available when static memory feature is used (–enable_staticmemory). It gives the optimum buffer size for memory “buckets”. This allows for a way to compute buffer size so that no extra unused memory is left at the end after it has been partitioned. The returned value, if positive, is the computed buffer size to use.

	Name
int	**wolfSSL_MemoryPaddingSz**(void)This function is available when static memory feature is used (–enable_staticmemory). It gives the size of padding needed for each partition of memory. This padding size will be the size needed to contain a memory management structure along with any extra for memory alignment.
int	**wolfSSL_CTX_load_static_memory**(WOLFSSL_CTX ** ctx, wolfSSL_method_func method, unsigned char * buf, unsigned int sz, int flag, int max)This function is used to set aside static memory for a CTX. Memory set aside is then used for the CTX's lifetime and for any SSL objects created from the CTX. By passing in a NULL ctx pointer and a wolfSSL_method_func function the creation of the CTX itself will also use static memory. wolfSSL_method_func has the function signature of WOLFSSL_METHOD* (*wolfSSL_method_func)(void* heap);. Passing in 0 for max makes it behave as if not set and no max concurrent use restrictions is in place. The flag value passed in determines how the memory is used and behavior while operating. Available flags are the following.
int	**wolfSSL_CTX_is_static_memory**(WOLFSSL_CTX * ctx, WOLFSSL_MEM_STATS * mem_stats)This function does not change any of the connections behavior and is used only for gathering information about the static memory usage.
int	**wolfSSL_is_static_memory**(WOLFSSL * ssl, WOLFSSL_MEM_CONN_STATS * mem_stats)wolfSSL_is_static_memory is used to gather information about a SSL's static memory usage. The return value indicates if static memory is being used and WOLFSSL_MEM_CONN_STATS will be filled out if and only if the flag WOLFMEM_TRACK_STATS was passed to the parent CTX when loading in static memory.

	Name
int	**wc_LoadStaticMemory**(WOLFSSL_HEAP_HINT * hint, unsigned char * buf, unsigned int sz, int flag, int max)This function is used to set aside static memory for wolfCrypt use. Memory can be used by passing the created heap hint into functions. An example of this is when calling wc_InitRng_ex. The flag value passed in determines how the memory is used and behavior while operating, in general wolfCrypt operations will use memory from a WOLFMEM_GENERAL pool. Available flags are the following.

C.34.2 Functions Documentation

C.34.2.1 function wolfSSL_Malloc

```
void * wolfSSL_Malloc(
    size_t size,
    void * heap,
    int type
)
```

This function is similar to malloc(), but calls the memory allocation function which wolfSSL has been configured to use. By default, wolfSSL uses malloc(). This can be changed using the wolfSSL memory abstraction layer - see wolfSSL_SetAllocators(). Note wolfSSL_Malloc is not called directly by wolfSSL, but instead called by macro XMALLOC. For the default build only the size argument exists. If using WOLFSSL_STATIC_MEMORY build then heap and type arguments are included.

Parameters:

- **size** size, in bytes, of the memory to allocate
- **heap** heap hint to use for memory. Can be NULL
- **type** dynamic type (see DYNAMIC_TYPE_ list in types.h)

See:

- wolfSSL_Free
- wolfSSL_Realloc
- wolfSSL_SetAllocators
- XMALLOC
- XFREE
- XREALLOC

Return:

- pointer If successful, this function returns a pointer to allocated memory.
- error If there is an error, NULL will be returned.

Example

```
int* tenInts = (int*)wolfSSL_Malloc(sizeof(int)*10);
```

C.34.2.2 function wolfSSL_Free

```
void wolfSSL_Free(
    void * ptr,
```

```
    void * heap,
    int type
)
```

This function is similar to free(), but calls the memory free function which wolfSSL has been configured to use. By default, wolfSSL uses free(). This can be changed using the wolfSSL memory abstraction layer - see wolfSSL_SetAllocators(). Note wolfSSL_Free is not called directly by wolfSSL, but instead called by macro XFREE. For the default build only the ptr argument exists. If using WOLFSSL_STATIC_MEMORY build then heap and type arguments are included.

Parameters:

- **ptr** pointer to the memory to be freed.
- **heap** heap hint to use for memory. Can be NULL
- **type** dynamic type (see DYNAMIC_TYPE_ list in types.h)

See:

- wolfSSL_Alloc
- wolfSSL_Realloc
- wolfSSL_SetAllocators
- XMALLOC
- XFREE
- XREALLOC

Return: none No returns.

Example

```
int* tenInts = (int*)wolfSSL_Malloc(sizeof(int)*10);
// process data as desired
...
if(tenInts) {
    wolfSSL_Free(tenInts);
}
```

C.34.2.3 function wolfSSL_Realloc

```
void * wolfSSL_Realloc(
    void * ptr,
    size_t size,
    void * heap,
    int type
)
```

This function is similar to realloc(), but calls the memory re-allocation function which wolfSSL has been configured to use. By default, wolfSSL uses realloc(). This can be changed using the wolfSSL memory abstraction layer - see wolfSSL_SetAllocators(). Note wolfSSL_Realloc is not called directly by wolfSSL, but instead called by macro XREALLOC. For the default build only the size argument exists. If using WOLFSSL_STATIC_MEMORY build then heap and type arguments are included.

Parameters:

- **ptr** pointer to the previously-allocated memory, to be reallocated.
- **size** number of bytes to allocate.
- **heap** heap hint to use for memory. Can be NULL
- **type** dynamic type (see DYNAMIC_TYPE_ list in types.h)

See:

- wolfSSL_Free

- wolfSSL_Malloc
- wolfSSL_SetAllocators
- XMALLOC
- XFREE
- XREALLOC

Return:

- pointer If successful, this function returns a pointer to re-allocated memory. This may be the same pointer as ptr, or a new pointer location.
- Null If there is an error, NULL will be returned.

Example

```
int* tenInts = (int*)wolfSSL_Malloc(sizeof(int)*10);
int* twentyInts = (int*)wolfSSL_Realloc(tenInts, sizeof(int)*20);
```

C.34.2.4 function wolfSSL_SetAllocators

```
int wolfSSL_SetAllocators(
    wolfSSL_Malloc_cb ,
    wolfSSL_Free_cb ,
    wolfSSL_Realloc_cb
)
```

This function registers the allocation functions used by wolfSSL. By default, if the system supports it, malloc/free and realloc are used. Using this function allows the user at runtime to install their own memory handlers.

Parameters:

- **malloc_function** memory allocation function for wolfSSL to use. Function signature must match wolfSSL_Malloc_cb prototype, above.
- **free_function** memory free function for wolfSSL to use. Function signature must match wolfSSL_Free_cb prototype, above.
- **realloc_function** memory re-allocation function for wolfSSL to use. Function signature must match wolfSSL_Realloc_cb prototype, above.

See: none

Return:

- Success If successful this function will return 0.
- BAD_FUNC_ARG is the error that will be returned if a function pointer is not provided.

Example

```
static void* MyMalloc(size_t size)
{
    // custom malloc function
}

static void MyFree(void* ptr)
{
    // custom free function
}

static void* MyRealloc(void* ptr, size_t size)
{
    // custom realloc function
```

```
}

// Register custom memory functions with wolfSSL
int ret = wolfSSL_SetAllocators(MyMalloc, MyFree, MyRealloc);
if (ret != 0) {
    // failed to set memory functions
}
```

C.34.2.5 function wolfSSL_StaticBufferSz

```
int wolfSSL_StaticBufferSz(
    byte * buffer,
    word32 sz,
    int flag
)
```

This function is available when static memory feature is used (–enable-staticmemory). It gives the optimum buffer size for memory “buckets”. This allows for a way to compute buffer size so that no extra unused memory is left at the end after it has been partitioned. The returned value, if positive, is the computed buffer size to use.

Parameters:

- **buffer** pointer to buffer
- **size** size of buffer
- **type** desired type of memory ie WOLFMEM_GENERAL or WOLFMEM_IO_POOL

See:

- wolfSSL_Malloc
- wolfSSL_Free

Return:

- Success On successfully completing buffer size calculations a positive value is returned. This returned value is for optimum buffer size.
- Failure All negative values are considered to be error cases.

Example

```
byte buffer[1000];
word32 size = sizeof(buffer);
int optimum;
optimum = wolfSSL_StaticBufferSz(buffer, size, WOLFMEM_GENERAL);
if (optimum < 0) { //handle error case }
printf(“The optimum buffer size to make use of all memory is %d\n”,
optimum);
...
```

C.34.2.6 function wolfSSL_MemoryPaddingSz

```
int wolfSSL_MemoryPaddingSz(
    void
)
```

This function is available when static memory feature is used (–enable-staticmemory). It gives the size of padding needed for each partition of memory. This padding size will be the size needed to contain a memory management structure along with any extra for memory alignment.

Parameters:

- **none** No parameters.

See:

- wolfSSL_Malloc
- wolfSSL_Free

Return:

- On successfully memory padding calculation the return value will be a positive value
- All negative values are considered error cases.

Example

```
int padding;
padding = wolfSSL_MemoryPaddingSz();
if (padding < 0) { //handle error case }
printf("The padding size needed for each \"bucket\" of memory is %d\n",
padding);
// calculation of buffer for IO POOL size is number of buckets
// times (padding + WOLFMEM_IO_SZ)
...
```

C.34.2.7 function wolfSSL_CTX_load_static_memory

```
int wolfSSL_CTX_load_static_memory(
    WOLFSSL_CTX ** ctx,
    wolfSSL_method_func method,
    unsigned char * buf,
    unsigned int sz,
    int flag,
    int max
)
```

This function is used to set aside static memory for a CTX. Memory set aside is then used for the CTX's lifetime and for any SSL objects created from the CTX. By passing in a NULL ctx pointer and a wolfSSL_method_func function the creation of the CTX itself will also use static memory. wolfSSL_method_func has the function signature of WOLFSSL_METHOD* (*wolfSSL_method_func)(void* heap);. Passing in 0 for max makes it behave as if not set and no max concurrent use restrictions is in place. The flag value passed in determines how the memory is used and behavior while operating. Available flags are the following.

Parameters:

- **ctx** address of pointer to a WOLFSSL_CTX structure.
- **method** function to create protocol. (should be NULL if ctx is not also NULL)
- **buf** memory to use for all operations.
- **sz** size of memory buffer being passed in.
- **flag** type of memory.
- **max** max concurrent operations.

See:

- wolfSSL_CTX_new
- wolfSSL_CTX_is_static_memory
- wolfSSL_is_static_memory

Return:

- If successful, SSL_SUCCESS will be returned.
- All unsuccessful return values will be less than 0 or equal to SSL_FAILURE.

0 - default general memory

WOLFMEM_IO_POOL - used for input/output buffer when sending receiving messages. Overrides general memory, so all memory in buffer passed in is used for IO. WOLFMEM_IO_FIXED - same as WOLFMEM_IO_POOL but each SSL now keeps two buffers to themselves for their lifetime. WOLFMEM_TRACK_STATS - each SSL keeps track of memory stats while running.

Example

```
WOLFSSL_CTX* ctx;
WOLFSSL* ssl;
int ret;
unsigned char memory[MAX];
int memorySz = MAX;
unsigned char IO[MAX];
int IOSz = MAX;
int flag = WOLFMEM_IO_FIXED | WOLFMEM_TRACK_STATS;
...
// create ctx also using static memory, start with general memory to use
ctx = NULL:
ret = wolfSSL_CTX_load_static_memory(&ctx, wolfSSLv23_server_method_ex,
 ↪  memory, memorySz, 0,
MAX_CONCURRENT_HANDSHAKES);
if (ret != SSL_SUCCESS) {
// handle error case
}
// load in memory for use with IO
ret = wolfSSL_CTX_load_static_memory(&ctx, NULL, IO, IOSz, flag,
 ↪  MAX_CONCURRENT_IO);
if (ret != SSL_SUCCESS) {
// handle error case
}
...
```

C.34.2.8 function wolfSSL_CTX_is_static_memory

```
int wolfSSL_CTX_is_static_memory(
    WOLFSSL_CTX * ctx,
    WOLFSSL_MEM_STATS * mem_stats
)
```

This function does not change any of the connections behavior and is used only for gathering information about the static memory usage.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().
- **mem_stats** structure to hold information about staic memory usage.

See:

- wolfSSL_CTX_new
- wolfSSL_CTX_load_static_memory
- wolfSSL_is_static_memory

Return:

- A value of 1 is returned if using static memory for the CTX is true.
- 0 is returned if not using static memory.

Example

```
WOLFSSL_CTX* ctx;
int ret;
WOLFSSL_MEM_STATS mem_stats;
...
//get information about static memory with CTX

ret = wolfSSL_CTX_is_static_memory(ctx, &mem_stats);

if (ret == 1) {
    // handle case of is using static memory
    // print out or inspect elements of mem_stats
}

if (ret == 0) {
    //handle case of ctx not using static memory
}
...
```

C.34.2.9 function wolfSSL_is_static_memory

```
int wolfSSL_is_static_memory(
    WOLFSSL * ssl,
    WOLFSSL_MEM_CONN_STATS * mem_stats
)
```

wolfSSL_is_static_memory is used to gather information about a SSL's static memory usage. The return value indicates if static memory is being used and WOLFSSL_MEM_CONN_STATS will be filled out if and only if the flag WOLFMEM_TRACK_STATS was passed to the parent CTX when loading in static memory.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **mem_stats** structure to contain static memory usage

See:

- wolfSSL_new
- wolfSSL_CTX_is_static_memory

Return:

- A value of 1 is returned if using static memory for the CTX is true.
- 0 is returned if not using static memory.

Example

```
WOLFSSL* ssl;
int ret;
WOLFSSL_MEM_CONN_STATS mem_stats;

...

ret = wolfSSL_is_static_memory(ssl, mem_stats);

if (ret == 1) {
    // handle case when is static memory
    // investigate elements in mem_stats if WOLFMEM_TRACK_STATS flag
```

```
}
...
```

C.34.2.10 function wc_LoadStaticMemory

```
int wc_LoadStaticMemory(
    WOLFSSL_HEAP_HINT * hint,
    unsigned char * buf,
    unsigned int sz,
    int flag,
    int max
)
```

This function is used to set aside static memory for wolfCrypt use. Memory can be used by passing the created heap hint into functions. An example of this is when calling wc_InitRng_ex. The flag value passed in determines how the memory is used and behavior while operating, in general wolfCrypt operations will use memory from a WOLFMEM_GENERAL pool. Available flags are the following.

Parameters:

- **hint** WOLFSSL_HEAP_HINT structure to use
- **buf** memory to use for all operations.
- **sz** size of memory buffer being passed in.
- **flag** type of memory.
- **max** max concurrent operations (handshakes, IO).

See: none

Return:

- If successful, 0 will be returned.
- All unsuccessful return values will be less than 0.

WOLFMEM_GENERAL - default general memory

WOLFMEM_IO_POOL - used for input/output buffer when sending receiving messages. Overrides general memory, so all memory in buffer passed in is used for IO. WOLFMEM_IO_FIXED - same as WOLFMEM_IO_POOL but each SSL now keeps two buffers to themselves for their lifetime. WOLFMEM_TRACK_STATS - each SSL keeps track of memory stats while running

Example

```
WOLFSSL_HEAP_HINT hint;
int ret;
unsigned char memory[MAX];
int memorySz = MAX;
int flag = WOLFMEM_GENERAL | WOLFMEM_TRACK_STATS;
...

// load in memory for use

ret = wc_LoadStaticMemory(&hint, memory, memorySz, flag, 0);
if (ret != SSL_SUCCESS) {
    // handle error case
}
...

ret = wc_InitRng_ex(&rng, hint, 0);
```

```
// check ret value
```

C.34.3 Source code

```
void* wolfSSL_Malloc(size_t size, void* heap, int type);

void  wolfSSL_Free(void *ptr, void* heap, int type);

void* wolfSSL_Realloc(void *ptr, size_t size, void* heap, int type);

int wolfSSL_SetAllocators(wolfSSL_Malloc_cb,
                                    wolfSSL_Free_cb,
                                    wolfSSL_Realloc_cb);

int wolfSSL_StaticBufferSz(byte* buffer, word32 sz, int flag);

int wolfSSL_MemoryPaddingSz(void);

int wolfSSL_CTX_load_static_memory(WOLFSSL_CTX** ctx, wolfSSL_method_func
 ↪  method,
        unsigned char* buf, unsigned int sz, int flag, int max);

int wolfSSL_CTX_is_static_memory(WOLFSSL_CTX* ctx, WOLFSSL_MEM_STATS*
 ↪  mem_stats);

int wolfSSL_is_static_memory(WOLFSSL* ssl, WOLFSSL_MEM_CONN_STATS* mem_stats);

int wc_LoadStaticMemory(WOLFSSL_HEAP_HINT* hint, unsigned char* buf, unsigned
 ↪  int sz,
                int flag, int max);
```

C.35 dox_comments/header_files/pem.h

C.35.1 Functions

	Name
int	**wolfSSL_PEM_write_bio_PrivateKey**(WOLFSSL_BIO * bio, WOLFSSL_EVP_PKEY * key, const WOLFSSL_EVP_CIPHER * cipher, unsigned char * passwd, int len, wc_pem_password_cb * cb, void * arg)This function writes a key into a WOLFSSL_BIO structure in PEM format.

C.35.2 Functions Documentation

C.35.2.1 function wolfSSL_PEM_write_bio_PrivateKey

```
int wolfSSL_PEM_write_bio_PrivateKey(
    WOLFSSL_BIO * bio,
    WOLFSSL_EVP_PKEY * key,
    const WOLFSSL_EVP_CIPHER * cipher,
```

```
    unsigned char * passwd,
    int len,
    wc_pem_password_cb * cb,
    void * arg
)
```

This function writes a key into a WOLFSSL_BIO structure in PEM format.

Parameters:

- **bio** WOLFSSL_BIO structure to get PEM buffer from.
- **key** key to convert to PEM format.
- **cipher** EVP cipher structure.
- **passwd** password.
- **len** length of password.
- **cb** password callback.
- **arg** optional argument.

See: wolfSSL_PEM_read_bio_X509_AUX

Return:

- SSL_SUCCESS upon success.
- SSL_FAILURE upon failure.

Example

```
WOLFSSL_BIO* bio;
WOLFSSL_EVP_PKEY* key;
int ret;
// create bio and setup key
ret = wolfSSL_PEM_write_bio_PrivateKey(bio, key, NULL, NULL, 0, NULL, NULL);
//check ret value
```

C.35.3 Source code

```
int wolfSSL_PEM_write_bio_PrivateKey(WOLFSSL_BIO* bio, WOLFSSL_EVP_PKEY* key,
                                     const WOLFSSL_EVP_CIPHER* cipher,
                                     unsigned char* passwd, int len,
                                     wc_pem_password_cb* cb, void* arg);
```

C.36 dox_comments/header_files/pkcs11.h

C.36.1 Functions

	Name
int	**wc_Pkcs11_Initialize**(Pkcs11Dev * dev, const char * library, void * heap)
void	**wc_Pkcs11_Finalize**(Pkcs11Dev * dev)
int	**wc_Pkcs11Token_Init**(Pkcs11Token * token, Pkcs11Dev * dev, int slotId, const char * tokenName, const unsigned char * userPin, int userPinSz)
void	**wc_Pkcs11Token_Final**(Pkcs11Token * token)
int	**wc_Pkcs11Token_Open**(Pkcs11Token * token, int readWrite)

	Name
void	**wc_Pkcs11Token_Close**(Pkcs11Token * token)

C.36.2 Functions Documentation

C.36.2.1 function wc_Pkcs11_Initialize

```
int wc_Pkcs11_Initialize(
    Pkcs11Dev * dev,
    const char * library,
    void * heap
)
```

C.36.2.2 function wc_Pkcs11_Finalize

```
void wc_Pkcs11_Finalize(
    Pkcs11Dev * dev
)
```

C.36.2.3 function wc_Pkcs11Token_Init

```
int wc_Pkcs11Token_Init(
    Pkcs11Token * token,
    Pkcs11Dev * dev,
    int slotId,
    const char * tokenName,
    const unsigned char * userPin,
    int userPinSz
)
```

C.36.2.4 function wc_Pkcs11Token_Final

```
void wc_Pkcs11Token_Final(
    Pkcs11Token * token
)
```

C.36.2.5 function wc_Pkcs11Token_Open

```
int wc_Pkcs11Token_Open(
    Pkcs11Token * token,
    int readWrite
)
```

C.36.2.6 function wc_Pkcs11Token_Close

```
void wc_Pkcs11Token_Close(
    Pkcs11Token * token
)
```

C.36.3 Source code

```
int wc_Pkcs11_Initialize(Pkcs11Dev* dev, const char* library,
```

```
                                    void* heap);

void wc_Pkcs11_Finalize(Pkcs11Dev* dev);

int wc_Pkcs11Token_Init(Pkcs11Token* token, Pkcs11Dev* dev,
    int slotId, const char* tokenName, const unsigned char *userPin,
    int userPinSz);

void wc_Pkcs11Token_Final(Pkcs11Token* token);

int wc_Pkcs11Token_Open(Pkcs11Token* token, int readWrite);

void wc_Pkcs11Token_Close(Pkcs11Token* token);

int wc_Pkcs11StoreKey(Pkcs11Token* token, int type, int clear,

int wc_Pkcs11_CryptoDevCb(int devId, wc_CryptoInfo* info,
    void* ctx);
```

C.37 dox_comments/header_files/pkcs7.h

C.37.1 Functions

	Name
int	**wc_PKCS7_InitWithCert**(PKCS7 * pkcs7, byte * cert, word32 certSz)This function initializes a PKCS7 structure with a DER-formatted certificate. To initialize an empty PKCS7 structure, one can pass in a NULL cert and 0 for certSz.
void	**wc_PKCS7_Free**(PKCS7 * pkcs7)This function releases any memory allocated by a PKCS7 initializer.
int	**wc_PKCS7_EncodeData**(PKCS7 * pkcs7, byte * output, word32 outputSz)This function builds the PKCS7 data content type, encoding the PKCS7 structure into a buffer containing a parsable PKCS7 data packet.
int	**wc_PKCS7_EncodeSignedData**(PKCS7 * pkcs7, byte * output, word32 outputSz)This function builds the PKCS7 signed data content type, encoding the PKCS7 structure into a buffer containing a parsable PKCS7 signed data packet.

	Name
int	**wc_PKCS7_EncodeSignedData_ex**(PKCS7 * pkcs7, const byte * hashBuf, word32 hashSz, byte * outputHead, word32 * outputHeadSz, byte * outputFoot, word32 * outputFootSz)This function builds the PKCS7 signed data content type, encoding the PKCS7 structure into a header and footer buffer containing a parsable PKCS7 signed data packet. This does not include the content. A hash must be computed and provided for the data.
int	**wc_PKCS7_VerifySignedData**(PKCS7 * pkcs7, byte * pkiMsg, word32 pkiMsgSz)This function takes in a transmitted PKCS7 signed data message, extracts the certificate list and certificate revocation list, and then verifies the signature. It stores the extracted content in the given PKCS7 structure.
int	**wc_PKCS7_VerifySignedData_ex**(PKCS7 * pkcs7, const byte * hashBuf, word32 hashSz, byte * pkiMsgHead, word32 pkiMsgHeadSz, byte * pkiMsgFoot, word32 pkiMsgFootSz)This function takes in a transmitted PKCS7 signed data message as hash/header/footer, then extracts the certificate list and certificate revocation list, and then verifies the signature. It stores the extracted content in the given PKCS7 structure.
int	**wc_PKCS7_EncodeEnvelopedData**(PKCS7 * pkcs7, byte * output, word32 outputSz)This function builds the PKCS7 enveloped data content type, encoding the PKCS7 structure into a buffer containing a parsable PKCS7 enveloped data packet.
int	**wc_PKCS7_DecodeEnvelopedData**(PKCS7 * pkcs7, byte * pkiMsg, word32 pkiMsgSz, byte * output, word32 outputSz)This function unwraps and decrypts a PKCS7 enveloped data content type, decoding the message into output. It uses the private key of the PKCS7 object passed in to decrypt the message.

C.37.2 Functions Documentation

C.37.2.1 function wc_PKCS7_InitWithCert

```
int wc_PKCS7_InitWithCert(
    PKCS7 * pkcs7,
    byte * cert,
    word32 certSz
)
```

This function initializes a PKCS7 structure with a DER-formatted certificate. To initialize an empty PKCS7 structure, one can pass in a NULL cert and 0 for certSz.

Parameters:

- **pkcs7** pointer to the PKCS7 structure in which to store the decoded cert
- **cert** pointer to a buffer containing a DER formatted ASN.1 certificate with which to initialize the PKCS7 structure
- **certSz** size of the certificate buffer

See: wc_PKCS7_Free

Return:

- 0 Returned on successfully initializing the PKCS7 structure
- MEMORY_E Returned if there is an error allocating memory with XMALLOC
- ASN_PARSE_E Returned if there is an error parsing the cert header
- ASN_OBJECT_ID_E Returned if there is an error parsing the encryption type from the cert
- ASN_EXPECT_0_E Returned if there is a formatting error in the encryption specification of the cert file
- ASN_BEFORE_DATE_E Returned if the date is before the certificate start date
- ASN_AFTER_DATE_E Returned if the date is after the certificate expiration date
- ASN_BITSTR_E Returned if there is an error parsing a bit string from the certificate
- ECC_CURVE_OID_E Returned if there is an error parsing the ECC key from the certificate
- ASN_UNKNOWN_OID_E Returned if the certificate is using an unknown key object id
- ASN_VERSION_E Returned if the ALLOW_V1_EXTENSIONS option is not defined and the certificate is a V1 or V2 certificate
- BAD_FUNC_ARG Returned if there is an error processing the certificate extension
- ASN_CRIT_EXT_E Returned if an unfamiliar critical extension is encountered in processing the certificate
- ASN_SIG_OID_E Returned if the signature encryption type is not the same as the encryption type of the certificate in the provided file
- ASN_SIG_CONFIRM_E Returned if confirming the certification signature fails
- ASN_NAME_INVALID_E Returned if the certificate's name is not permitted by the CA name constraints
- ASN_NO_SIGNER_E Returned if there is no CA signer to verify the certificate's authenticity

Example

```
PKCS7 pkcs7;
byte derBuff[] = { }; // initialize with DER-encoded certificate
if ( wc_PKCS7_InitWithCert(&pkcs7, derBuff, sizeof(derBuff)) != 0 ) {
    // error parsing certificate into pkcs7 format
}
```

C.37.2.2 function wc_PKCS7_Free

```
void wc_PKCS7_Free(
    PKCS7 * pkcs7
)
```

This function releases any memory allocated by a PKCS7 initializer.

Parameters:

- **pkcs7** pointer to the PKCS7 structure to free

See: wc_PKCS7_InitWithCert

Return: none No returns.

Example

```
PKCS7 pkcs7;
// initialize and use PKCS7 object

wc_PKCS7_Free(pkcs7);
```

C.37.2.3 function wc_PKCS7_EncodeData

```
int wc_PKCS7_EncodeData(
    PKCS7 * pkcs7,
    byte * output,
    word32 outputSz
)
```

This function builds the PKCS7 data content type, encoding the PKCS7 structure into a buffer containing a parsable PKCS7 data packet.

Parameters:

- **pkcs7** pointer to the PKCS7 structure to encode
- **output** pointer to the buffer in which to store the encoded certificate
- **outputSz** size available in the output buffer

See: wc_PKCS7_InitWithCert

Return:

- Success On successfully encoding the PKCS7 data into the buffer, returns the index parsed up to in the PKCS7 structure. This index also corresponds to the bytes written to the output buffer.
- BUFFER_E Returned if the given buffer is not large enough to hold the encoded certificate

Example

```
PKCS7 pkcs7;
int ret;

byte derBuff[] = { }; // initialize with DER-encoded certificate
byte pkcs7Buff[FOURK_BUF];

wc_PKCS7_InitWithCert(&pkcs7, derBuff, sizeof(derBuff));
// update message and data to encode
pkcs7.privateKey = key;
pkcs7.privateKeySz = keySz;
pkcs7.content = data;
pkcs7.contentSz = dataSz;
... etc.

ret = wc_PKCS7_EncodeData(&pkcs7, pkcs7Buff, sizeof(pkcs7Buff));
if ( ret != 0 ) {
    // error encoding into output buffer
}
```

C.37.2.4 function wc_PKCS7_EncodeSignedData

```
int wc_PKCS7_EncodeSignedData(
    PKCS7 * pkcs7,
    byte * output,
    word32 outputSz
)
```

This function builds the PKCS7 signed data content type, encoding the PKCS7 structure into a buffer containing a parsable PKCS7 signed data packet.

Parameters:

- **pkcs7** pointer to the PKCS7 structure to encode
- **output** pointer to the buffer in which to store the encoded certificate
- **outputSz** size available in the output buffer

See:

- wc_PKCS7_InitWithCert
- wc_PKCS7_VerifySignedData

Return:

- Success On successfully encoding the PKCS7 data into the buffer, returns the index parsed up to in the PKCS7 structure. This index also corresponds to the bytes written to the output buffer.
- BAD_FUNC_ARG Returned if the PKCS7 structure is missing one or more required elements to generate a signed data packet
- MEMORY_E Returned if there is an error allocating memory
- PUBLIC_KEY_E Returned if there is an error parsing the public key
- RSA_BUFFER_E Returned if buffer error, output too small or input too large
- BUFFER_E Returned if the given buffer is not large enough to hold the encoded certificate
- MP_INIT_E may be returned if there is an error generating the signature
- MP_READ_E may be returned if there is an error generating the signature
- MP_CMP_E may be returned if there is an error generating the signature
- MP_INVMOD_E may be returned if there is an error generating the signature
- MP_EXPTMOD_E may be returned if there is an error generating the signature
- MP_MOD_E may be returned if there is an error generating the signature
- MP_MUL_E may be returned if there is an error generating the signature
- MP_ADD_E may be returned if there is an error generating the signature
- MP_MULMOD_E may be returned if there is an error generating the signature
- MP_TO_E may be returned if there is an error generating the signature
- MP_MEM may be returned if there is an error generating the signature

Example

```
PKCS7 pkcs7;
int ret;

byte data[] = {}; // initialize with data to sign
byte derBuff[] = { }; // initialize with DER-encoded certificate
byte pkcs7Buff[FOURK_BUF];

wc_PKCS7_InitWithCert(&pkcs7, derBuff, sizeof(derBuff));
// update message and data to encode
pkcs7.privateKey = key;
pkcs7.privateKeySz = keySz;
pkcs7.content = data;
pkcs7.contentSz = dataSz;
pkcs7.hashOID = SHAh;
pkcs7.rng = &rng;
... etc.

ret = wc_PKCS7_EncodeSignedData(&pkcs7, pkcs7Buff, sizeof(pkcs7Buff));
if ( ret != 0 ) {
    // error encoding into output buffer
```

```
}

wc_PKCS7_Free(&pkcs7);
```

C.37.2.5 function wc_PKCS7_EncodeSignedData_ex

```
int wc_PKCS7_EncodeSignedData_ex(
    PKCS7 * pkcs7,
    const byte * hashBuf,
    word32 hashSz,
    byte * outputHead,
    word32 * outputHeadSz,
    byte * outputFoot,
    word32 * outputFootSz
)
```

This function builds the PKCS7 signed data content type, encoding the PKCS7 structure into a header and footer buffer containing a parsable PKCS7 signed data packet. This does not include the content. A hash must be computed and provided for the data.

Parameters:

- **pkcs7** pointer to the PKCS7 structure to encode
- **hashBuf** pointer to computed hash for the content data
- **hashSz** size of the digest
- **outputHead** pointer to the buffer in which to store the encoded certificate header
- **outputHeadSz** pointer populated with size of output header buffer and returns actual size
- **outputFoot** pointer to the buffer in which to store the encoded certificate footer
- **outputFootSz** pointer populated with size of output footer buffer and returns actual size

See:

- wc_PKCS7_InitWithCert
- wc_PKCS7_VerifySignedData_ex

Return:

- 0=Success
- BAD_FUNC_ARG Returned if the PKCS7 structure is missing one or more required elements to generate a signed data packet
- MEMORY_E Returned if there is an error allocating memory
- PUBLIC_KEY_E Returned if there is an error parsing the public key
- RSA_BUFFER_E Returned if buffer error, output too small or input too large
- BUFFER_E Returned if the given buffer is not large enough to hold the encoded certificate
- MP_INIT_E may be returned if there is an error generating the signature
- MP_READ_E may be returned if there is an error generating the signature
- MP_CMP_E may be returned if there is an error generating the signature
- MP_INVMOD_E may be returned if there is an error generating the signature
- MP_EXPTMOD_E may be returned if there is an error generating the signature
- MP_MOD_E may be returned if there is an error generating the signature
- MP_MUL_E may be returned if there is an error generating the signature
- MP_ADD_E may be returned if there is an error generating the signature
- MP_MULMOD_E may be returned if there is an error generating the signature
- MP_TO_E may be returned if there is an error generating the signature
- MP_MEM may be returned if there is an error generating the signature

Example

```
PKCS7 pkcs7;
int ret;
byte derBuff[] = { }; // initialize with DER-encoded certificate
byte data[] = {}; // initialize with data to sign
byte pkcs7HeadBuff[FOURK_BUF/2];
byte pkcs7FootBuff[FOURK_BUF/2];
word32 pkcs7HeadSz = (word32)sizeof(pkcs7HeadBuff);
word32 pkcs7FootSz = (word32)sizeof(pkcs7HeadBuff);
enum wc_HashType hashType = WC_HASH_TYPE_SHA;
byte   hashBuf[WC_MAX_DIGEST_SIZE];
word32 hashSz = wc_HashGetDigestSize(hashType);

wc_PKCS7_InitWithCert(&pkcs7, derBuff, sizeof(derBuff));
// update message and data to encode
pkcs7.privateKey = key;
pkcs7.privateKeySz = keySz;
pkcs7.content = NULL;
pkcs7.contentSz = dataSz;
pkcs7.hashOID = SHAh;
pkcs7.rng = &rng;
... etc.

// calculate hash for content
ret = wc_HashInit(&hash, hashType);
if (ret == 0) {
    ret = wc_HashUpdate(&hash, hashType, data, sizeof(data));
    if (ret == 0) {
        ret = wc_HashFinal(&hash, hashType, hashBuf);
    }
    wc_HashFree(&hash, hashType);
}

ret = wc_PKCS7_EncodeSignedData_ex(&pkcs7, hashBuf, hashSz, pkcs7HeadBuff,
    &pkcs7HeadSz, pkcs7FootBuff, &pkcs7FootSz);
if ( ret != 0 ) {
    // error encoding into output buffer
}

wc_PKCS7_Free(&pkcs7);
```

C.37.2.6 function wc_PKCS7_VerifySignedData

```
int wc_PKCS7_VerifySignedData(
    PKCS7 * pkcs7,
    byte * pkiMsg,
    word32 pkiMsgSz
)
```

This function takes in a transmitted PKCS7 signed data message, extracts the certificate list and certificate revocation list, and then verifies the signature. It stores the extracted content in the given PKCS7 structure.

Parameters:

- **pkcs7** pointer to the PKCS7 structure in which to store the parsed certificates
- **pkiMsg** pointer to the buffer containing the signed message to verify and decode

- **pkiMsgSz** size of the signed message

See:

- wc_PKCS7_InitWithCert
- wc_PKCS7_EncodeSignedData

Return:

- 0 Returned on successfully extracting the information from the message
- BAD_FUNC_ARG Returned if one of the input parameters is invalid
- ASN_PARSE_E Returned if there is an error parsing from the given pkiMsg
- PKCS7_OID_E Returned if the given pkiMsg is not a signed data type
- ASN_VERSION_E Returned if the PKCS7 signer info is not version 1
- MEMORY_E Returned if there is an error allocating memory
- PUBLIC_KEY_E Returned if there is an error parsing the public key
- RSA_BUFFER_E Returned if buffer error, output too small or input too large
- BUFFER_E Returned if the given buffer is not large enough to hold the encoded certificate
- MP_INIT_E may be returned if there is an error generating the signature
- MP_READ_E may be returned if there is an error generating the signature
- MP_CMP_E may be returned if there is an error generating the signature
- MP_INVMOD_E may be returned if there is an error generating the signature
- MP_EXPTMOD_E may be returned if there is an error generating the signature
- MP_MOD_E may be returned if there is an error generating the signature
- MP_MUL_E may be returned if there is an error generating the signature
- MP_ADD_E may be returned if there is an error generating the signature
- MP_MULMOD_E may be returned if there is an error generating the signature
- MP_TO_E may be returned if there is an error generating the signature
- MP_MEM may be returned if there is an error generating the signature

Example

```
PKCS7 pkcs7;
int ret;
byte pkcs7Buff[] = {}; // the PKCS7 signature

wc_PKCS7_InitWithCert(&pkcs7, NULL, 0);
// update message and data to encode
pkcs7.privateKey = key;
pkcs7.privateKeySz = keySz;
pkcs7.content = data;
pkcs7.contentSz = dataSz;
... etc.

ret = wc_PKCS7_VerifySignedData(&pkcs7, pkcs7Buff, sizeof(pkcs7Buff));
if ( ret != 0 ) {
    // error encoding into output buffer
}

wc_PKCS7_Free(&pkcs7);
```

C.37.2.7 function wc_PKCS7_VerifySignedData_ex

```
int wc_PKCS7_VerifySignedData_ex(
    PKCS7 * pkcs7,
    const byte * hashBuf,
    word32 hashSz,
```

```
    byte * pkiMsgHead,
    word32 pkiMsgHeadSz,
    byte * pkiMsgFoot,
    word32 pkiMsgFootSz
)
```

This function takes in a transmitted PKCS7 signed data message as hash/header/footer, then extracts the certificate list and certificate revocation list, and then verifies the signature. It stores the extracted content in the given PKCS7 structure.

Parameters:

- **pkcs7** pointer to the PKCS7 structure in which to store the parsed certificates
- **hashBuf** pointer to computed hash for the content data
- **hashSz** size of the digest
- **pkiMsgHead** pointer to the buffer containing the signed message header to verify and decode
- **pkiMsgHeadSz** size of the signed message header
- **pkiMsgFoot** pointer to the buffer containing the signed message footer to verify and decode
- **pkiMsgFootSz** size of the signed message footer

See:

- wc_PKCS7_InitWithCert
- wc_PKCS7_EncodeSignedData_ex

Return:

- 0 Returned on successfully extracting the information from the message
- BAD_FUNC_ARG Returned if one of the input parameters is invalid
- ASN_PARSE_E Returned if there is an error parsing from the given pkiMsg
- PKCS7_OID_E Returned if the given pkiMsg is not a signed data type
- ASN_VERSION_E Returned if the PKCS7 signer info is not version 1
- MEMORY_E Returned if there is an error allocating memory
- PUBLIC_KEY_E Returned if there is an error parsing the public key
- RSA_BUFFER_E Returned if buffer error, output too small or input too large
- BUFFER_E Returned if the given buffer is not large enough to hold the encoded certificate
- MP_INIT_E may be returned if there is an error generating the signature
- MP_READ_E may be returned if there is an error generating the signature
- MP_CMP_E may be returned if there is an error generating the signature
- MP_INVMOD_E may be returned if there is an error generating the signature
- MP_EXPTMOD_E may be returned if there is an error generating the signature
- MP_MOD_E may be returned if there is an error generating the signature
- MP_MUL_E may be returned if there is an error generating the signature
- MP_ADD_E may be returned if there is an error generating the signature
- MP_MULMOD_E may be returned if there is an error generating the signature
- MP_TO_E may be returned if there is an error generating the signature
- MP_MEM may be returned if there is an error generating the signature

Example

```
PKCS7 pkcs7;
int ret;
byte data[] = {}; // initialize with data to sign
byte pkcs7HeadBuff[] = {}; // initialize with PKCS7 header
byte pkcs7FootBuff[] = {}; // initialize with PKCS7 footer
enum wc_HashType hashType = WC_HASH_TYPE_SHA;
byte   hashBuf[WC_MAX_DIGEST_SIZE];
word32 hashSz = wc_HashGetDigestSize(hashType);
```

```
wc_PKCS7_InitWithCert(&pkcs7, NULL, 0);
// update message and data to encode
pkcs7.privateKey = key;
pkcs7.privateKeySz = keySz;
pkcs7.content = NULL;
pkcs7.contentSz = dataSz;
pkcs7.rng = &rng;
... etc.

// calculate hash for content
ret = wc_HashInit(&hash, hashType);
if (ret == 0) {
    ret = wc_HashUpdate(&hash, hashType, data, sizeof(data));
    if (ret == 0) {
        ret = wc_HashFinal(&hash, hashType, hashBuf);
    }
    wc_HashFree(&hash, hashType);
}

ret = wc_PKCS7_VerifySignedData_ex(&pkcs7, hashBuf, hashSz, pkcs7HeadBuff,
    sizeof(pkcs7HeadBuff), pkcs7FootBuff, sizeof(pkcs7FootBuff));
if ( ret != 0 ) {
    // error encoding into output buffer
}

wc_PKCS7_Free(&pkcs7);
```

C.37.2.8 function wc_PKCS7_EncodeEnvelopedData

```
int wc_PKCS7_EncodeEnvelopedData(
    PKCS7 * pkcs7,
    byte * output,
    word32 outputSz
)
```

This function builds the PKCS7 enveloped data content type, encoding the PKCS7 structure into a buffer containing a parsable PKCS7 enveloped data packet.

Parameters:

- **pkcs7** pointer to the PKCS7 structure to encode
- **output** pointer to the buffer in which to store the encoded certificate
- **outputSz** size available in the output buffer

See:

- wc_PKCS7_InitWithCert
- wc_PKCS7_DecodeEnvelopedData

Return:

- Success Returned on successfully encoding the message in enveloped data format, returns the size written to the output buffer
- BAD_FUNC_ARG: Returned if one of the input parameters is invalid, or if the PKCS7 structure is missing required elements
- ALGO_ID_E Returned if the PKCS7 structure is using an unsupported algorithm type. Currently, only DESb and DES3b are supported
- BUFFER_E Returned if the given output buffer is too small to store the output data

- MEMORY_E Returned if there is an error allocating memory
- RNG_FAILURE_E Returned if there is an error initializing the random number generator for encryption
- DRBG_FAILED Returned if there is an error generating numbers with the random number generator used for encryption

Example

```
PKCS7 pkcs7;
int ret;

byte derBuff[] = { }; // initialize with DER-encoded certificate
byte pkcs7Buff[FOURK_BUF];

wc_PKCS7_InitWithCert(&pkcs7, derBuff, sizeof(derBuff));
// update message and data to encode
pkcs7.privateKey = key;
pkcs7.privateKeySz = keySz;
pkcs7.content = data;
pkcs7.contentSz = dataSz;
... etc.

ret = wc_PKCS7_EncodeEnvelopedData(&pkcs7, pkcs7Buff, sizeof(pkcs7Buff));
if ( ret != 0 ) {
    // error encoding into output buffer
}
```

C.37.2.9 function wc_PKCS7_DecodeEnvelopedData

```
int wc_PKCS7_DecodeEnvelopedData(
    PKCS7 * pkcs7,
    byte * pkiMsg,
    word32 pkiMsgSz,
    byte * output,
    word32 outputSz
)
```

This function unwraps and decrypts a PKCS7 enveloped data content type, decoding the message into output. It uses the private key of the PKCS7 object passed in to decrypt the message.

Parameters:

- **pkcs7** pointer to the PKCS7 structure containing the private key with which to decode the enveloped data package
- **pkiMsg** pointer to the buffer containing the enveloped data package
- **pkiMsgSz** size of the enveloped data package
- **output** pointer to the buffer in which to store the decoded message
- **outputSz** size available in the output buffer

See:

- wc_PKCS7_InitWithCert
- wc_PKCS7_EncodeEnvelopedData

Return:

- On successfully extracting the information from the message, returns the bytes written to output
- BAD_FUNC_ARG Returned if one of the input parameters is invalid
- ASN_PARSE_E Returned if there is an error parsing from the given pkiMsg

- PKCS7_OID_E Returned if the given pkiMsg is not an enveloped data type
- ASN_VERSION_E Returned if the PKCS7 signer info is not version 0
- MEMORY_E Returned if there is an error allocating memory
- ALGO_ID_E Returned if the PKCS7 structure is using an unsupported algorithm type. Currently, only DESb and DES3b are supported for encryption, with RSAk for signature generation
- PKCS7_RECIP_E Returned if there is no recipient found in the enveloped data that matches the recipient provided
- RSA_BUFFER_E Returned if there is an error during RSA signature verification due to buffer error, output too small or input too large.
- MP_INIT_E may be returned if there is an error during signature verification
- MP_READ_E may be returned if there is an error during signature verification
- MP_CMP_E may be returned if there is an error during signature verification
- MP_INVMOD_E may be returned if there is an error during signature verification
- MP_EXPTMOD_E may be returned if there is an error during signature verification
- MP_MOD_E may be returned if there is an error during signature verification
- MP_MUL_E may be returned if there is an error during signature verification
- MP_ADD_E may be returned if there is an error during signature verification
- MP_MULMOD_E may be returned if there is an error during signature verification
- MP_TO_E may be returned if there is an error during signature verification
- MP_MEM may be returned if there is an error during signature verification

Example

```
PKCS7 pkcs7;
byte received[] = { }; // initialize with received enveloped message
byte decoded[FOURK_BUF];
int decodedSz;

// initialize pkcs7 with certificate
// update key
pkcs7.privateKey = key;
pkcs7.privateKeySz = keySz;

decodedSz = wc_PKCS7_DecodeEnvelopedData(&pkcs7, received,
sizeof(received),decoded, sizeof(decoded));
if ( decodedSz != 0 ) {
    // error decoding message
}
```

C.37.3 Source code

```
int  wc_PKCS7_InitWithCert(PKCS7* pkcs7, byte* cert, word32 certSz);

void wc_PKCS7_Free(PKCS7* pkcs7);

int  wc_PKCS7_EncodeData(PKCS7* pkcs7, byte* output,
                                       word32 outputSz);

int  wc_PKCS7_EncodeSignedData(PKCS7* pkcs7,
                                       byte* output, word32 outputSz);

int wc_PKCS7_EncodeSignedData_ex(PKCS7* pkcs7, const byte* hashBuf,
    word32 hashSz, byte* outputHead, word32* outputHeadSz, byte* outputFoot,
    word32* outputFootSz);
```

```
int  wc_PKCS7_VerifySignedData(PKCS7* pkcs7,
                                     byte* pkiMsg, word32 pkiMsgSz);


int wc_PKCS7_VerifySignedData_ex(PKCS7* pkcs7, const byte* hashBuf,
    word32 hashSz, byte* pkiMsgHead, word32 pkiMsgHeadSz, byte* pkiMsgFoot,
    word32 pkiMsgFootSz);

int  wc_PKCS7_EncodeEnvelopedData(PKCS7* pkcs7,
                                      byte* output, word32 outputSz);

int  wc_PKCS7_DecodeEnvelopedData(PKCS7* pkcs7, byte* pkiMsg,
                                      word32 pkiMsgSz, byte* output,
                                      word32 outputSz);
```

C.38 dox_comments/header_files/poly1305.h

C.38.1 Functions

	Name
int	**wc_Poly1305SetKey**(Poly1305 * poly1305, const byte * key, word32 kySz)This function sets the key for a Poly1305 context structure, initializing it for hashing. Note: A new key should be set after generating a message hash with wc_Poly1305Final to ensure security.
int	**wc_Poly1305Update**(Poly1305 * poly1305, const byte * m, word32 bytes)This function updates the message to hash with the Poly1305 structure.
int	**wc_Poly1305Final**(Poly1305 * poly1305, byte * tag)This function calculates the hash of the input messages and stores the result in mac. After this is called, the key should be reset.
int	**wc_Poly1305_MAC**(Poly1305 * ctx, byte * additional, word32 addSz, byte * input, word32 sz, byte * tag, word32 tagSz)Takes in an initialized Poly1305 struct that has a key loaded and creates a MAC (tag) using recent TLS AEAD padding scheme.

C.38.2 Functions Documentation

C.38.2.1 function wc_Poly1305SetKey

```
int wc_Poly1305SetKey(
    Poly1305 * poly1305,
    const byte * key,
    word32 kySz
)
```

This function sets the key for a Poly1305 context structure, initializing it for hashing. Note: A new key should be set after generating a message hash with wc_Poly1305Final to ensure security.

Parameters:

- **ctx** pointer to a Poly1305 structure to initialize
- **key** pointer to the buffer containing the key to use for hashing
- **keySz** size of the key in the buffer. Should be 32 bytes

See:

- wc_Poly1305Update
- wc_Poly1305Final

Return:

- 0 Returned on successfully setting the key and initializing the Poly1305 structure
- BAD_FUNC_ARG Returned if the given key is not 32 bytes long, or the Poly1305 context is NULL

Example

```
Poly1305 enc;
byte key[] = { initialize with 32 byte key to use for hashing };
wc_Poly1305SetKey(&enc, key, sizeof(key));
```

C.38.2.2 function wc_Poly1305Update

```
int wc_Poly1305Update(
    Poly1305 * poly1305,
    const byte * m,
    word32 bytes
)
```

This function updates the message to hash with the Poly1305 structure.

Parameters:

- **ctx** pointer to a Poly1305 structure for which to update the message to hash
- **m** pointer to the buffer containing the message which should be added to the hash
- **bytes** size of the message to hash

See:

- wc_Poly1305SetKey
- wc_Poly1305Final

Return:

- 0 Returned on successfully updating the message to hash
- BAD_FUNC_ARG Returned if the Poly1305 structure is NULL

Example

```
Poly1305 enc;
byte key[] = { }; // initialize with 32 byte key to use for encryption

byte msg[] = { }; // initialize with message to hash
wc_Poly1305SetKey(&enc, key, sizeof(key));

if( wc_Poly1305Update(key, msg, sizeof(msg)) != 0 ) {
    // error updating message to hash
}
```

C.38.2.3 function wc_Poly1305Final

```
int wc_Poly1305Final(
    Poly1305 * poly1305,
    byte * tag
)
```

This function calculates the hash of the input messages and stores the result in mac. After this is called, the key should be reset.

Parameters:

- **ctx** pointer to a Poly1305 structure with which to generate the MAC
- **mac** pointer to the buffer in which to store the MAC. Should be POLY1305_DIGEST_SIZE (16 bytes) wide

See:

- wc_Poly1305SetKey
- wc_Poly1305Update

Return:

- 0 Returned on successfully computing the final MAC
- BAD_FUNC_ARG Returned if the Poly1305 structure is NULL

Example

```
Poly1305 enc;
byte mac[POLY1305_DIGEST_SIZE]; // space for a 16 byte mac

byte key[] = { }; // initialize with 32 byte key to use for encryption

byte msg[] = { }; // initialize with message to hash
wc_Poly1305SetKey(&enc, key, sizeof(key));
wc_Poly1305Update(key, msg, sizeof(msg));

if ( wc_Poly1305Final(&enc, mac) != 0 ) {
    // error computing final MAC
}
```

C.38.2.4 function wc_Poly1305_MAC

```
int wc_Poly1305_MAC(
    Poly1305 * ctx,
    byte * additional,
    word32 addSz,
    byte * input,
    word32 sz,
    byte * tag,
    word32 tagSz
)
```

Takes in an initialized Poly1305 struct that has a key loaded and creates a MAC (tag) using recent TLS AEAD padding scheme.

Parameters:

- **ctx** Initialized Poly1305 struct to use
- **additional** Additional data to use
- **addSz** Size of additional buffer

- **input** Input buffer to create tag from
- **sz** Size of input buffer
- **tag** Buffer to hold created tag
- **tagSz** Size of input tag buffer (must be at least WC_POLY1305_MAC_SZ(16))

See:

- wc_Poly1305SetKey
- wc_Poly1305Update
- wcPoly1305Final

Return:

- 0 Success
- BAD_FUNC_ARG Returned if ctx, input, or tag is null or if additional is null and addSz is greater than 0 or if tagSz is less than WC_POLY1305_MAC_SZ.

Example

```
Poly1305 ctx;
byte key[] = { }; // initialize with 32 byte key to use for hashing
byte additional[] = { }; // initialize with additional data
byte msg[] = { }; // initialize with message
byte tag[16];

wc_Poly1305SetKey(&ctx, key, sizeof(key));
if(wc_Poly1305_MAC(&ctx, additional, sizeof(additional), (byte*)msg,
sizeof(msg), tag, sizeof(tag)) != 0)
{
    // Handle the error
}
```

C.38.3 Source code

```
int wc_Poly1305SetKey(Poly1305* poly1305, const byte* key,
                                  word32 kySz);

int wc_Poly1305Update(Poly1305* poly1305, const byte* m, word32 bytes);

int wc_Poly1305Final(Poly1305* poly1305, byte* tag);

int wc_Poly1305_MAC(Poly1305* ctx, byte* additional, word32 addSz,
                             byte* input, word32 sz, byte* tag, word32 tagSz);
```

C.39 dox_comments/header_files/psa.h

C.39.1 Functions

	Name
int	**wolfSSL_CTX_psa_enable**(WOLFSSL_CTX * ctx)This function enables PSA support on the given context.
int	**wolfSSL_set_psa_ctx**(WOLFSSL * ssl, struct psa_ssl_ctx * ctx)This function setup the PSA context for the given SSL session.

	Name
void	**wolfSSL_free_psa_ctx**(struct psa_ssl_ctx * ctx)This function releases the resources used by a PSA context.
int	**wolfSSL_psa_set_private_key_id**(struct psa_ssl_ctx * ctx, psa_key_id_t id)This function set the private key used by an SSL session.

C.39.2 Functions Documentation

C.39.2.1 function wolfSSL_CTX_psa_enable

```
int wolfSSL_CTX_psa_enable(
    WOLFSSL_CTX * ctx
)
```

This function enables PSA support on the given context.

Parameters:

- **ctx** pointer to the WOLFSSL_CTX object on which the PSA support must be enabled

See: wolfSSL_set_psa_ctx

Return:

- WOLFSSL_SUCCESS on success
- BAD_FUNC_ARG if ctx == NULL

Example

```
WOLFSSL_CTX *ctx;
ctx = wolfSSL_CTX_new(wolfTLSv1_2_client_method());
if (!ctx)
    return NULL;
ret = wolfSSL_CTX_psa_enable(ctx);
if (ret != WOLFSSL_SUCCESS)
    printf("can't enable PSA on ctx");
```

C.39.2.2 function wolfSSL_set_psa_ctx

```
int wolfSSL_set_psa_ctx(
    WOLFSSL * ssl,
    struct psa_ssl_ctx * ctx
)
```

This function setup the PSA context for the given SSL session.

Parameters:

- **ssl** pointer to the WOLFSSL where the ctx will be enabled
- **ctx** pointer to a struct psa_ssl_ctx (must be unique for a ssl session)

See:

- wolfSSL_psa_set_private_key_id
- wolfSSL_psa_free_psa_ctx

Return:

- WOLFSSL_SUCCESS on success

- BAD_FUNC_ARG if ssl or ctx are NULL

This function setup the PSA context for the TLS callbacks to the given SSL session. At the end of the session, the resources used by the context should be freed using wolfSSL_free_psa_ctx().

Example

```
// Create new ssl session
WOLFSSL *ssl;
struct psa_ssl_ctx psa_ctx = { 0 };
ssl = wolfSSL_new(ctx);
if (!ssl)
    return NULL;
// setup PSA context
ret = wolfSSL_set_psa_ctx(ssl, ctx);
```

C.39.2.3 function wolfSSL_free_psa_ctx

```
void wolfSSL_free_psa_ctx(
    struct psa_ssl_ctx * ctx
)
```

This function releases the resources used by a PSA context.

Parameters:

- **ctx** pointer to a struct psa_ssl_ctx

See: wolfSSL_set_psa_ctx

C.39.2.4 function wolfSSL_psa_set_private_key_id

```
int wolfSSL_psa_set_private_key_id(
    struct psa_ssl_ctx * ctx,
    psa_key_id_t id
)
```

This function set the private key used by an SSL session.

Parameters:

- **ctx** pointer to a struct psa_ssl_ctx
- **id** PSA id of the key to be used as private key

See: wolfSSL_set_psa_ctx

Example

```
// Create new ssl session
WOLFSSL *ssl;
struct psa_ssl_ctx psa_ctx = { 0 };
psa_key_id_t key_id;

// key provisioning already done
get_private_key_id(&key_id);

ssl = wolfSSL_new(ctx);
if (!ssl)
    return NULL;
```

```
wolfSSL_psa_set_private_key_id(&psa_ctx, key_id);
wolfSSL_set_psa_ctx(ssl, ctx);
```

C.39.3 Source code

```
int wolfSSL_CTX_psa_enable(WOLFSSL_CTX *ctx);

int wolfSSL_set_psa_ctx(WOLFSSL *ssl, struct psa_ssl_ctx *ctx);

void wolfSSL_free_psa_ctx(struct psa_ssl_ctx *ctx);

int wolfSSL_psa_set_private_key_id(struct psa_ssl_ctx *ctx,
                                                psa_key_id_t id);
```

C.40 dox_comments/header_files/pwdbased.h

C.40.1 Functions

	Name
int	**wc_PBKDF1**(byte * output, const byte * passwd, int pLen, const byte * salt, int sLen, int iterations, int kLen, int typeH)This function implements the Password Based Key Derivation Function 1 (PBKDF1), converting an input password with a concatenated salt into a more secure key, which it stores in output. It allows the user to select between SHA and MD5 as hash functions.
int	**wc_PBKDF2**(byte * output, const byte * passwd, int pLen, const byte * salt, int sLen, int iterations, int kLen, int typeH)This function implements the Password Based Key Derivation Function 2 (PBKDF2), converting an input password with a concatenated salt into a more secure key, which it stores in output. It allows the user to select any of the supported HMAC hash functions, including: WC_MD5, WC_SHA, WC_SHA256, WC_SHA384, WC_SHA512, WC_SHA3_224, WC_SHA3_256, WC_SHA3_384 or WC_SHA3_512.

	Name
int	**wc_PKCS12_PBKDF**(byte * output, const byte * passwd, int pLen, const byte * salt, int sLen, int iterations, int kLen, int typeH, int purpose)This function implements the Password Based Key Derivation Function (PBKDF) described in RFC 7292 Appendix B. This function converts an input password with a concatenated salt into a more secure key, which it stores in output. It allows the user to select any of the supported HMAC hash functions, including: WC_MD5, WC_SHA, WC_SHA256, WC_SHA384, WC_SHA512, WC_SHA3_224, WC_SHA3_256, WC_SHA3_384 or WC_SHA3_512.

C.40.2 Functions Documentation

C.40.2.1 function wc_PBKDF1

```
int wc_PBKDF1(
    byte * output,
    const byte * passwd,
    int pLen,
    const byte * salt,
    int sLen,
    int iterations,
    int kLen,
    int typeH
)
```

This function implements the Password Based Key Derivation Function 1 (PBKDF1), converting an input password with a concatenated salt into a more secure key, which it stores in output. It allows the user to select between SHA and MD5 as hash functions.

Parameters:

- **output** pointer to the buffer in which to store the generated key. Should be at least kLen long
- **passwd** pointer to the buffer containing the password to use for the key derivation
- **pLen** length of the password to use for key derivation
- **salt** pointer to the buffer containing the salt to use for key derivation
- **sLen** length of the salt
- **iterations** number of times to process the hash
- **kLen** desired length of the derived key. Should not be longer than the digest size of the hash chosen
- **hashType** the hashing algorithm to use. Valid choices are WC_MD5 and WC_SHA

See:

- wc_PBKDF2
- wc_PKCS12_PBKDF

Return:

- 0 Returned on successfully deriving a key from the input password
- BAD_FUNC_ARG Returned if there is an invalid hash type given (valid type are: MD5 and SHA), iterations is less than 1, or the key length (kLen) requested is greater than the hash length of the provided hash

- MEMORY_E Returned if there is an error allocating memory for a SHA or MD5 object

Example

```
int ret;
byte key[WC_MD5_DIGEST_SIZE];
byte pass[] = { }; // initialize with password
byte salt[] = { }; // initialize with salt

ret = wc_PBKDF1(key, pass, sizeof(pass), salt, sizeof(salt), 1000,
sizeof(key), WC_MD5);
if ( ret != 0 ) {
    // error deriving key from password
}
```

C.40.2.2 function wc_PBKDF2

```
int wc_PBKDF2(
    byte * output,
    const byte * passwd,
    int pLen,
    const byte * salt,
    int sLen,
    int iterations,
    int kLen,
    int typeH
)
```

This function implements the Password Based Key Derivation Function 2 (PBKDF2), converting an input password with a concatenated salt into a more secure key, which it stores in output. It allows the user to select any of the supported HMAC hash functions, including: WC_MD5, WC_SHA, WC_SHA256, WC_SHA384, WC_SHA512, WC_SHA3_224, WC_SHA3_256, WC_SHA3_384 or WC_SHA3_512.

Parameters:

- **output** pointer to the buffer in which to store the generated key. Should be kLen long
- **passwd** pointer to the buffer containing the password to use for the key derivation
- **pLen** length of the password to use for key derivation
- **salt** pointer to the buffer containing the salt to use for key derivation
- **sLen** length of the salt
- **iterations** number of times to process the hash
- **kLen** desired length of the derived key
- **hashType** the hashing algorithm to use. Valid choices are: WC_MD5, WC_SHA, WC_SHA256, WC_SHA384, WC_SHA512, WC_SHA3_224, WC_SHA3_256, WC_SHA3_384 or WC_SHA3_512

See:

- wc_PBKDF1
- wc_PKCS12_PBKDF

Return:

- 0 Returned on successfully deriving a key from the input password
- BAD_FUNC_ARG Returned if there is an invalid hash type given or iterations is less than 1
- MEMORY_E Returned if there is an allocating memory for the HMAC object

Example

```
int ret;
byte key[64];
```

```
byte pass[] = { }; // initialize with password
byte salt[] = { }; // initialize with salt

ret = wc_PBKDF2(key, pass, sizeof(pass), salt, sizeof(salt), 2048, sizeof(key),
WC_SHA512);
if ( ret != 0 ) {
    // error deriving key from password
}
```

C.40.2.3 function wc_PKCS12_PBKDF

```
int wc_PKCS12_PBKDF(
    byte * output,
    const byte * passwd,
    int pLen,
    const byte * salt,
    int sLen,
    int iterations,
    int kLen,
    int typeH,
    int purpose
)
```

This function implements the Password Based Key Derivation Function (PBKDF) described in RFC 7292 Appendix B. This function converts an input password with a concatenated salt into a more secure key, which it stores in output. It allows the user to select any of the supported HMAC hash functions, including: WC_MD5, WC_SHA, WC_SHA256, WC_SHA384, WC_SHA512, WC_SHA3_224, WC_SHA3_256, WC_SHA3_384 or WC_SHA3_512.

Parameters:

- **output** pointer to the buffer in which to store the generated key. Should be kLen long
- **passwd** pointer to the buffer containing the password to use for the key derivation
- **pLen** length of the password to use for key derivation
- **salt** pointer to the buffer containing the salt to use for key derivation
- **sLen** length of the salt
- **iterations** number of times to process the hash
- **kLen** desired length of the derived key
- **hashType** the hashing algorithm to use. Valid choices are: WC_MD5, WC_SHA, WC_SHA256, WC_SHA384, WC_SHA512, WC_SHA3_224, WC_SHA3_256, WC_SHA3_384 or WC_SHA3_512
- **id** this is a byte identifier indicating the purpose of key generation. It is used to diversify the key output, and should be assigned as follows: ID=1: pseudorandom bits are to be used as key material for performing encryption or decryption. ID=2: pseudorandom bits are to be used an IV (Initial Value) for encryption or decryption. ID=3: pseudorandom bits are to be used as an integrity key for MACing.

See:

- wc_PBKDF1
- wc_PBKDF2

Return:

- 0 Returned on successfully deriving a key from the input password
- BAD_FUNC_ARG Returned if there is an invalid hash type given, iterations is less than 1, or the key length (kLen) requested is greater than the hash length of the provided hash
- MEMORY_E Returned if there is an allocating memory
- MP_INIT_E may be returned if there is an error during key generation

- MP_READ_E may be returned if there is an error during key generation
- MP_CMP_E may be returned if there is an error during key generation
- MP_INVMOD_E may be returned if there is an error during key generation
- MP_EXPTMOD_E may be returned if there is an error during key generation
- MP_MOD_E may be returned if there is an error during key generation
- MP_MUL_E may be returned if there is an error during key generation
- MP_ADD_E may be returned if there is an error during key generation
- MP_MULMOD_E may be returned if there is an error during key generation
- MP_TO_E may be returned if there is an error during key generation
- MP_MEM may be returned if there is an error during key generation

Example

```
int ret;
byte key[64];
byte pass[] = { }; // initialize with password
byte salt[] = { }; // initialize with salt

ret = wc_PKCS512_PBKDF(key, pass, sizeof(pass), salt, sizeof(salt), 2048,
sizeof(key), WC_SHA512, 1);
if ( ret != 0 ) {
    // error deriving key from password
}
```

C.40.3 Source code

```
int wc_PBKDF1(byte* output, const byte* passwd, int pLen,
                      const byte* salt, int sLen, int iterations, int kLen,
                      int typeH);

int wc_PBKDF2(byte* output, const byte* passwd, int pLen,
                      const byte* salt, int sLen, int iterations, int kLen,
                      int typeH);

int wc_PKCS12_PBKDF(byte* output, const byte* passwd, int pLen,
                            const byte* salt, int sLen, int iterations,
                            int kLen, int typeH, int purpose);
```

C.41 dox_comments/header_files/quic.h

C.41.1 Functions

	Name
int	**wolfSSL_CTX_set_quic_method**(WOLFSSL_CTX * ctx, const WOLFSSL_QUIC_METHOD * quic_method)Activate QUIC protocol for a WOLFSSL_CTX and all derived WOLFSSL instances by providing the four callbacks required. The CTX needs to be a TLSv1.3 one.

	Name
int	**wolfSSL_set_quic_method**(WOLFSSL * ssl, const WOLFSSL_QUIC_METHOD * quic_method)Activate QUIC protocol for a WOLFSSL instance by providing the four callbacks required. The WOLFSSL needs to be a TLSv1.3 one.
int	**wolfSSL_is_quic**(WOLFSSL * ssl)Check if QUIC has been activated in a WOLFSSL instance.
WOLFSSL_ENCRYPTION_LEVEL	**wolfSSL_quic_read_level**(const WOLFSSL * ssl)Determine the encryption level for reads currently in use. Meaningful only when the WOLFSSL instance is using QUIC.
WOLFSSL_ENCRYPTION_LEVEL	**wolfSSL_quic_write_level**(const WOLFSSL * ssl)Determine the encryption level for writes currently in use. Meaningful only when the WOLFSSL instance is using QUIC.
void	**wolfSSL_set_quic_use_legacy_codepoint**(WOLFSSL * ssl, int use_legacy)Configure which QUIC version shall be used. Without calling this, the WOLFSSL will offer both (draft_27 and v1) to a server, resp. accept both from a client and negotiate the most recent one.
void	**wolfSSL_set_quic_transport_version**(WOLFSSL * ssl, int version)Configure which QUIC version shall be used.
int	**wolfSSL_get_quic_transport_version**(const WOLFSSL * ssl)Get the configured QUIC version.
int	**wolfSSL_set_quic_transport_params**(WOLFSSL * ssl, const uint8_t * params, size_t params_len)Set the QUIC transport parameters to use.
int	**wolfSSL_get_peer_quic_transport_version**(const WOLFSSL * ssl)Get the negotiated QUIC transport version. This will only give meaningful results when called after the respective TLS extensions have been seen from the peer.
void	**wolfSSL_get_peer_quic_transport_params**(const WOLFSSL * ssl, const uint8_t ** out_params, size_t * out_params_len)Get the negotiated QUIC transport parameters. This will only give meaningful results when called after the respective TLS extensions have been seen from the peer.
void	**wolfSSL_set_quic_early_data_enabled**(WOLFSSL * ssl, int enabled)Configure if Early Data is enabled. Intended for servers to signal this to clients.

	Name
size_t	**wolfSSL_quic_max_handshake_flight_len**(const WOLFSSL * ssl, WOLFSSL_ENCRYPTION_LEVEL level)Get advice on the amount of data that shall be “in flight”, e.g. unacknowledged at the given encryption level. This is the amount of data the WOLFSSL instance is prepared to buffer.
int	**wolfSSL_provide_quic_data**(WOLFSSL * ssl, WOLFSSL_ENCRYPTION_LEVEL level, const uint8_t * data, size_t len)Pass decrypted CRYPTO data to the WOLFSSL instance for further processing. The encryption level between calls is only every allowed to increase and it is also checked that data records are complete before a change in encryption level is accepted.
WOLFSSL_API int	**wolfSSL_process_quic_post_handshake**(WOLFSSL * ssl)Process any CRYPTO records that have been provided after the handshake has completed. Will fail if called before that.
int	**wolfSSL_quic_read_write**(WOLFSSL * ssl)Process any CRYPTO records that have been provided during or after the handshake. Will progress the handshake if not already complete and otherwise work like wolfSSL_process_quic_post_handshake().
const WOLFSSL_EVP_CIPHER *	**wolfSSL_quic_get_aead**(WOLFSSL * ssl)Get the AEAD cipher negotiated in the TLS handshake.
int	**wolfSSL_quic_aead_is_gcm**(const WOLFSSL_EVP_CIPHER * aead_cipher)Check if the AEAD cipher is GCM.
int	**wolfSSL_quic_aead_is_ccm**(const WOLFSSL_EVP_CIPHER * aead_cipher)Check if the AEAD cipher is CCM.
int	**wolfSSL_quic_aead_is_chacha20**(const WOLFSSL_EVP_CIPHER * aead_cipher)Check if the AEAD cipher is CHACHA20.
WOLFSSL_API size_t	**wolfSSL_quic_get_aead_tag_len**(const WOLFSSL_EVP_CIPHER * aead_cipher)Determine the tag length for the AEAD cipher.
WOLFSSL_API const WOLFSSL_EVP_MD *	**wolfSSL_quic_get_md**(WOLFSSL * ssl)Determine the message digest negotiated in the TLS handshake.
const WOLFSSL_EVP_CIPHER *	**wolfSSL_quic_get_hp**(WOLFSSL * ssl)Determine the header protection cipher negotiated in the TLS handshake.
WOLFSSL_EVP_CIPHER_CTX *	**wolfSSL_quic_crypt_new**(const WOLFSSL_EVP_CIPHER * cipher, const uint8_t * key, const uint8_t * iv, int encrypt)Create a cipher context for en-/decryption.

	Name
int	**wolfSSL_quic_aead_encrypt**(uint8_t * dest, WOLFSSL_EVP_CIPHER_CTX * aead_ctx, const uint8_t * plain, size_t plainlen, const uint8_t * iv, const uint8_t * aad, size_t aadlen)Encrypt the plain text in the given context.
int	**wolfSSL_quic_aead_decrypt**(uint8_t * dest, WOLFSSL_EVP_CIPHER_CTX * ctx, const uint8_t * enc, size_t enclen, const uint8_t * iv, const uint8_t * aad, size_t aadlen)Decrypt the cipher text in the given context.
int	**wolfSSL_quic_hkdf_extract**(uint8_t * dest, const WOLFSSL_EVP_MD * md, const uint8_t * secret, size_t secretlen, const uint8_t * salt, size_t saltlen)Extract a pseudo random key.
int	**wolfSSL_quic_hkdf_expand**(uint8_t * dest, size_t destlen, const WOLFSSL_EVP_MD * md, const uint8_t * secret, size_t secretlen, const uint8_t * info, size_t infolen)Expand a pseudo random key into a new key.
int	**wolfSSL_quic_hkdf**(uint8_t * dest, size_t destlen, const WOLFSSL_EVP_MD * md, const uint8_t * secret, size_t secretlen, const uint8_t * salt, size_t saltlen, const uint8_t * info, size_t infolen)Expand and Extract a pseudo random key.

C.41.2 Attributes

	Name
int()(*WOLFSSL* ssl, WOLFSSL_ENCRYPTION_LEVEL level, const uint8_t *read_secret, const uint8_t* write_secret, size_t secret_len)	**set_encryption_secrets** Callback invoked when secrets are generated during a handshake. Since QUIC protocol handlers perform the en-/decryption of packets, they need the negotiated secrets for the levels early_data/handshake/application.
int()(*WOLFSSL* ssl, WOLFSSL_ENCRYPTION_LEVEL level, const uint8_t *data, size_t len)	**add_handshake_data** Callback invoked for forwarding handshake CRYPTO data to peer. The data forwarded this way is not encrypted. It is the job of the QUIC protocol implementation to do this. Which secrets are to be used is determined by the encryption level specified.
int()(*WOLFSSL* ssl)	**flush_flight** Callback invoked for advisory flushing of the data to send.
int()(*WOLFSSL* ssl, WOLFSSL_ENCRYPTION_LEVEL level, uint8_t alert)	**send_alert** Callback invoked when an SSL alert happened during processing.

C.41.3 Functions Documentation

C.41.3.1 function wolfSSL_CTX_set_quic_method

```
int wolfSSL_CTX_set_quic_method(
    WOLFSSL_CTX * ctx,
    const WOLFSSL_QUIC_METHOD * quic_method
)
```

Activate QUIC protocol for a WOLFSSL_CTX and all derived WOLFSSL instances by providing the four callbacks required. The CTX needs to be a TLSv1.3 one.

Parameters:

- **ctx** - a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().
- **quic_method** - the callback structure

See:

- wolfSSL_is_quic
- wolfSSL_set_quic_method

Return: WOLFSSL_SUCCESS If successful.

The passed quic_method needs to have a lifetime outlasting the SSL instances. It is not copied. All callbacks need to be provided.

C.41.3.2 function wolfSSL_set_quic_method

```
int wolfSSL_set_quic_method(
    WOLFSSL * ssl,
    const WOLFSSL_QUIC_METHOD * quic_method
)
```

Activate QUIC protocol for a WOLFSSL instance by providing the four callbacks required. The WOLFSSL needs to be a TLSv1.3 one.

Parameters:

- **ssl** - a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **quic_method** - the callback structure

See:

- wolfSSL_is_quic
- wolfSSL_CTX_set_quic_method

Return: WOLFSSL_SUCCESS If successful.

The passed quic_method needs to have a lifetime outlasting the SSL instance. It is not copied. All callbacks need to be provided.

C.41.3.3 function wolfSSL_is_quic

```
int wolfSSL_is_quic(
    WOLFSSL * ssl
)
```

Check if QUIC has been activated in a WOLFSSL instance.

Parameters:

- **ssl** - a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_CTX_quic_method
- wolfSSL_CTX_set_quic_method

Return: 1 if WOLFSSL is using QUIC.

C.41.3.4 function wolfSSL_quic_read_level

```
WOLFSSL_ENCRYPTION_LEVEL wolfSSL_quic_read_level(
    const WOLFSSL * ssl
)
```

Determine the encryption level for reads currently in use. Meaningful only when the WOLFSSL instance is using QUIC.

Parameters:

- **ssl** - a pointer to a WOLFSSL structure, created using wolfSSL_new().

See: wolfSSL_quic_write_level

Return: encryption level.

Note that the effective level is always a parameter when passing data back and forth. Data from a peer might arrive at other levels than reported via this function.

C.41.3.5 function wolfSSL_quic_write_level

```
WOLFSSL_ENCRYPTION_LEVEL wolfSSL_quic_write_level(
    const WOLFSSL * ssl
)
```

Determine the encryption level for writes currently in use. Meaningful only when the WOLFSSL instance is using QUIC.

Parameters:

- **ssl** - a pointer to a WOLFSSL structure, created using wolfSSL_new().

See: wolfSSL_quic_read_level

Return: encryption level.

Note that the effective level is always a parameter when passing data back and forth. Data from a peer might arrive at other levels than reported via this function.

C.41.3.6 function wolfSSL_set_quic_use_legacy_codepoint

```
void wolfSSL_set_quic_use_legacy_codepoint(
    WOLFSSL * ssl,
    int use_legacy
)
```

Configure which QUIC version shall be used. Without calling this, the WOLFSSL will offer both (draft-27 and v1) to a server, resp. accept both from a client and negotiate the most recent one.

Parameters:

- **ssl** - a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **use_legacy** - true if draft-27 shall be used, 0 if only QUICv1 is used.

See: wolfSSL_set_quic_transport_version

Return: WOLFSSL_SUCCESS If successful.

C.41.3.7 function wolfSSL_set_quic_transport_version

```
void wolfSSL_set_quic_transport_version(
    WOLFSSL * ssl,
    int version
)
```

Configure which QUIC version shall be used.

Parameters:

- **ssl** - a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **version** - the TLS Extension defined for the QUIC version.

See: wolfSSL_set_quic_use_legacy_codepoint

Return: WOLFSSL_SUCCESS If successful.

C.41.3.8 function wolfSSL_get_quic_transport_version

```
int wolfSSL_get_quic_transport_version(
    const WOLFSSL * ssl
)
```

Get the configured QUIC version.

Parameters:

- **ssl** - a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_set_quic_use_legacy_codepoint
- wolfSSL_set_quic_transport_version

Return: TLS Extension of configured version.

C.41.3.9 function wolfSSL_set_quic_transport_params

```
int wolfSSL_set_quic_transport_params(
    WOLFSSL * ssl,
    const uint8_t * params,
    size_t params_len
)
```

Set the QUIC transport parameters to use.

Parameters:

- **ssl** - a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **params** - the parameter bytes to use ·param params_len - the length of the parameters

See:

- wolfSSL_set_quic_use_legacy_codepoint
- wolfSSL_set_quic_transport_version

Return: WOLFSSL_SUCCESS If successful.

C.41.3.10 function wolfSSL_get_peer_quic_transport_version

```
int wolfSSL_get_peer_quic_transport_version(
    const WOLFSSL * ssl
)
```

Get the negotiated QUIC transport version. This will only give meaningful results when called after the respective TLS extensions have been seen from the peer.

Parameters:

- **ssl** - a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_set_quic_use_legacy_codepoint
- wolfSSL_set_quic_transport_version

Return: the negotiated version or -1.

C.41.3.11 function wolfSSL_get_peer_quic_transport_params

```
void wolfSSL_get_peer_quic_transport_params(
    const WOLFSSL * ssl,
    const uint8_t ** out_params,
    size_t * out_params_len
)
```

Get the negotiated QUIC transport parameters. This will only give meaningful results when called after the respective TLS extensions have been seen from the peer.

Parameters:

- **ssl** - a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **out_params** - the parameters sent be the peer, set to NULL if not available.
- **out_params_len** - the length of the parameters sent be the peer, set to 0 if not available

See: wolfSSL_get_peer_quic_transport_version

C.41.3.12 function wolfSSL_set_quic_early_data_enabled

```
void wolfSSL_set_quic_early_data_enabled(
    WOLFSSL * ssl,
    int enabled
)
```

Configure if Early Data is enabled. Intended for servers to signal this to clients.

Parameters:

- **ssl** - a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **enabled** - != 0 iff early data is enabled

C.41.3.13 function wolfSSL_quic_max_handshake_flight_len

```
size_t wolfSSL_quic_max_handshake_flight_len(
    const WOLFSSL * ssl,
    WOLFSSL_ENCRYPTION_LEVEL level
)
```

Get advice on the amount of data that shall be “in flight”, e.g. unacknowledged at the given encryption level. This is the amount of data the WOLFSSL instance is prepared to buffer.

Parameters:

- **ssl** - a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **level** - the encryption level to inquire about

Return: the recommend max data in flight

C.41.3.14 function wolfSSL_provide_quic_data

```
int wolfSSL_provide_quic_data(
    WOLFSSL * ssl,
    WOLFSSL_ENCRYPTION_LEVEL level,
    const uint8_t * data,
    size_t len
)
```

Pass decrypted CRYPTO data to the WOLFSSL instance for further processing. The encryption level between calls is only every allowed to increase and it is also checked that data records are complete before a change in encryption level is accepted.

Parameters:

- **ssl** - a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **level** - the level the data was encrypted at
- **data** - the data itself
- **len** - the length of the data

See:

- wolfSSL_process_quic_post_handshake
- wolfSSL_quic_read_write
- wolfSSL_accept
- wolfSSL_connect

Return: WOLFSSL_SUCCESS If successful.

C.41.3.15 function wolfSSL_process_quic_post_handshake

```
WOLFSSL_API int wolfSSL_process_quic_post_handshake(
    WOLFSSL * ssl
)
```

Process any CRYPTO records that have been provided after the handshake has completed. Will fail if called before that.

Parameters:

- **ssl** - a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_provide_quic_data
- wolfSSL_quic_read_write
- wolfSSL_accept
- wolfSSL_connect

Return: WOLFSSL_SUCCESS If successful.

C.41.3.16 function wolfSSL_quic_read_write

```
int wolfSSL_quic_read_write(
    WOLFSSL * ssl
)
```

Process any CRYPTO records that have been provided during or after the handshake. Will progress the handshake if not already complete and otherwise work like wolfSSL_process_quic_post_handshake().

Parameters:

- **ssl** - a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_provide_quic_data
- wolfSSL_quic_read_write
- wolfSSL_accept
- wolfSSL_connect

Return: WOLFSSL_SUCCESS If successful.

C.41.3.17 function wolfSSL_quic_get_aead

```
const WOLFSSL_EVP_CIPHER * wolfSSL_quic_get_aead(
    WOLFSSL * ssl
)
```

Get the AEAD cipher negotiated in the TLS handshake.

Parameters:

- **ssl** - a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_quic_aead_is_gcm
- wolfSSL_quic_aead_is_ccm
- wolfSSL_quic_aead_is_chacha20
- wolfSSL_quic_get_aead_tag_len
- wolfSSL_quic_get_md
- wolfSSL_quic_get_hp
- wolfSSL_quic_crypt_new
- wolfSSL_quic_aead_encrypt
- wolfSSL_quic_aead_decrypt

Return: negotiated cipher or NULL if not determined.

C.41.3.18 function wolfSSL_quic_aead_is_gcm

```
int wolfSSL_quic_aead_is_gcm(
    const WOLFSSL_EVP_CIPHER * aead_cipher
)
```

Check if the AEAD cipher is GCM.

Parameters:

- **cipher** - the cipher

See:

- wolfSSL_quic_get_aead

- wolfSSL_quic_aead_is_ccm
- wolfSSL_quic_aead_is_chacha20
- wolfSSL_quic_get_aead_tag_len
- wolfSSL_quic_get_md
- wolfSSL_quic_get_hp
- wolfSSL_quic_crypt_new
- wolfSSL_quic_aead_encrypt
- wolfSSL_quic_aead_decrypt

Return: != 0 iff the AEAD cipher is GCM.

C.41.3.19 function wolfSSL_quic_aead_is_ccm

```
int wolfSSL_quic_aead_is_ccm(
    const WOLFSSL_EVP_CIPHER * aead_cipher
)
```

Check if the AEAD cipher is CCM.

Parameters:

- **cipher** - the cipher

See:

- wolfSSL_quic_get_aead
- wolfSSL_quic_aead_is_gcm
- wolfSSL_quic_aead_is_chacha20
- wolfSSL_quic_get_aead_tag_len
- wolfSSL_quic_get_md
- wolfSSL_quic_get_hp
- wolfSSL_quic_crypt_new
- wolfSSL_quic_aead_encrypt
- wolfSSL_quic_aead_decrypt

Return: != 0 iff the AEAD cipher is CCM.

C.41.3.20 function wolfSSL_quic_aead_is_chacha20

```
int wolfSSL_quic_aead_is_chacha20(
    const WOLFSSL_EVP_CIPHER * aead_cipher
)
```

Check if the AEAD cipher is CHACHA20.

Parameters:

- **cipher** - the cipher

See:

- wolfSSL_quic_get_aead
- wolfSSL_quic_aead_is_ccm
- wolfSSL_quic_aead_is_gcm
- wolfSSL_quic_get_aead_tag_len
- wolfSSL_quic_get_md
- wolfSSL_quic_get_hp
- wolfSSL_quic_crypt_new
- wolfSSL_quic_aead_encrypt
- wolfSSL_quic_aead_decrypt

Return: != 0 iff the AEAD cipher is CHACHA20.

C.41.3.21 function wolfSSL_quic_get_aead_tag_len

```
WOLFSSL_API size_t wolfSSL_quic_get_aead_tag_len(
    const WOLFSSL_EVP_CIPHER * aead_cipher
)
```

Determine the tag length for the AEAD cipher.

Parameters:

- **cipher** - the cipher

See: wolfSSL_quic_get_aead

Return: tag length of AEAD cipher.

C.41.3.22 function wolfSSL_quic_get_md

```
WOLFSSL_API const WOLFSSL_EVP_MD * wolfSSL_quic_get_md(
    WOLFSSL * ssl
)
```

Determine the message digest negotiated in the TLS handshake.

Parameters:

- **ssl** - a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_quic_get_aead
- wolfSSL_quic_get_hp

Return: the message digest negotiated in the TLS handshake

C.41.3.23 function wolfSSL_quic_get_hp

```
const WOLFSSL_EVP_CIPHER * wolfSSL_quic_get_hp(
    WOLFSSL * ssl
)
```

Determine the header protection cipher negotiated in the TLS handshake.

Parameters:

- **ssl** - a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_quic_get_aead
- wolfSSL_quic_get_md

Return: the header protection cipher negotiated in the TLS handshake

C.41.3.24 function wolfSSL_quic_crypt_new

```
WOLFSSL_EVP_CIPHER_CTX * wolfSSL_quic_crypt_new(
    const WOLFSSL_EVP_CIPHER * cipher,
    const uint8_t * key,
    const uint8_t * iv,
```

```
    int encrypt
)
```

Create a cipher context for en-/decryption.

Parameters:

- **cipher** - the cipher to use in the context.
- **key** - the key to use in the context.
- **iv** - the iv to use in the context.
- **encrypt** - != 0 if for encryption, otherwise decryption

See:

- wolfSSL_quic_get_aead
- wolfSSL_quic_get_hp
- wolfSSL_quic_aead_encrypt
- wolfSSL_quic_aead_decrypt

Return: the created context or NULL in case of errors.

C.41.3.25 function wolfSSL_quic_aead_encrypt

```
int wolfSSL_quic_aead_encrypt(
    uint8_t * dest,
    WOLFSSL_EVP_CIPHER_CTX * aead_ctx,
    const uint8_t * plain,
    size_t plainlen,
    const uint8_t * iv,
    const uint8_t * aad,
    size_t aadlen
)
```

Encrypt the plain text in the given context.

Parameters:

- **dest** - destination where encrypted data is to be written
- **aead_ctx** - the cipher context to use
- **plain** - the plain data to encrypt
- **plainlen** - the length of the plain data
- **iv** - the iv to use
- **aad** - the add to use
- **aadlen** - the length of the aad

See:

- wolfSSL_quic_get_aead
- wolfSSL_quic_get_hp
- wolfSSL_quic_crypt_new
- wolfSSL_quic_aead_decrypt

Return: WOLFSSL_SUCCESS If successful.

C.41.3.26 function wolfSSL_quic_aead_decrypt

```
int wolfSSL_quic_aead_decrypt(
    uint8_t * dest,
    WOLFSSL_EVP_CIPHER_CTX * ctx,
    const uint8_t * enc,
```

```
    size_t enclen,
    const uint8_t * iv,
    const uint8_t * aad,
    size_t aadlen
)
```

Decrypt the cipher text in the given context.

Parameters:

- **dest** - destination where plain text is to be written
- **ctx** - the cipher context to use
- **enc** - the encrypted data to decrypt
- **envlen** - the length of the encrypted data
- **iv** - the iv to use
- **aad** - the add to use
- **aadlen** - the length of the aad

See:

- wolfSSL_quic_get_aead
- wolfSSL_quic_get_hp
- wolfSSL_quic_crypt_new
- wolfSSL_quic_aead_encrypt

Return: WOLFSSL_SUCCESS If successful.

C.41.3.27 function wolfSSL_quic_hkdf_extract

```
int wolfSSL_quic_hkdf_extract(
    uint8_t * dest,
    const WOLFSSL_EVP_MD * md,
    const uint8_t * secret,
    size_t secretlen,
    const uint8_t * salt,
    size_t saltlen
)
```

Extract a pseudo random key.

Parameters:

- **dest** - destination where key is to be written
- **md** - message digest to use
- **secret** - the secret to use
- **secretlen** - the length of the secret
- **salt** - the salt to use
- **saltlen** - the length of the salt

See:

- wolfSSL_quic_hkdf_expand
- wolfSSL_quic_hkdf

Return: WOLFSSL_SUCCESS If successful.

C.41.3.28 function wolfSSL_quic_hkdf_expand

```
int wolfSSL_quic_hkdf_expand(
    uint8_t * dest,
```

```
    size_t destlen,
    const WOLFSSL_EVP_MD * md,
    const uint8_t * secret,
    size_t secretlen,
    const uint8_t * info,
    size_t infolen
)
```

Expand a pseudo random key into a new key.

Parameters:

- **dest** - destination where key is to be written
- **destlen** - length of the key to expand
- **md** - message digest to use
- **secret** - the secret to use
- **secretlen** - the length of the secret
- **info** - the info to use
- **infolen** - the length of the info

See:

- wolfSSL_quic_hkdf_extract
- wolfSSL_quic_hkdf

Return: WOLFSSL_SUCCESS If successful.

C.41.3.29 function wolfSSL_quic_hkdf

```
int wolfSSL_quic_hkdf(
    uint8_t * dest,
    size_t destlen,
    const WOLFSSL_EVP_MD * md,
    const uint8_t * secret,
    size_t secretlen,
    const uint8_t * salt,
    size_t saltlen,
    const uint8_t * info,
    size_t infolen
)
```

Expand and Extract a pseudo random key.

Parameters:

- **dest** - destination where key is to be written
- **destlen** - length of the key
- **md** - message digest to use
- **secret** - the secret to use
- **secretlen** - the length of the secret
- **salt** - the salt to use
- **saltlen** - the length of the salt
- **info** - the info to use
- **infolen** - the length of the info

See:

- wolfSSL_quic_hkdf_extract
- wolfSSL_quic_hkdf_expand

Return: WOLFSSL_SUCCESS If successful.

C.41.4 Attributes Documentation

C.41.4.1 variable set_encryption_secrets

```
int(*)(WOLFSSL *ssl, WOLFSSL_ENCRYPTION_LEVEL level, const uint8_t
 ↪  *read_secret, const uint8_t *write_secret, size_t secret_len)
 ↪  set_encryption_secrets;
```

Callback invoked when secrets are generated during a handshake. Since QUIC protocol handlers perform the en-/decryption of packets, they need the negotiated secrets for the levels early_data/handshake/application.

Parameters:

- **ssl** - a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **level** - the encryption level the secrets are for
- **read_secret** - the secret used in decryption at the given level, may be NULL.
- **write_secret** - the secret used in encryption at the given level, may be NULL.
- **secret_len** - the length of the secret

See: wolfSSL_set_quic_method

Return: 1 on success, 0 on failure.

The callback will be invoked several times during a handshake. Either both or only the read or write secret might be provided. This does not mean the given encryption level is already in effect.

C.41.4.2 variable add_handshake_data

```
int(*)(WOLFSSL *ssl, WOLFSSL_ENCRYPTION_LEVEL level, const uint8_t *data,
 ↪  size_t len) add_handshake_data;
```

Callback invoked for forwarding handshake CRYPTO data to peer. The data forwarded this way is not encrypted. It is the job of the QUIC protocol implementation to do this. Which secrets are to be used is determined by the encryption level specified.

Parameters:

- **ssl** - a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **level** - the encryption level to use for encrypting the data
- **data** - the data itself
- **len** - the length of the data

See: wolfSSL_set_quic_method

Return: 1 on success, 0 on failure.

This callback may be invoked several times during handshake or post handshake processing. The data may cover a complete CRYPTO record, but may also be partial. However, the callback will have received all records data before using another encryption level.

C.41.4.3 variable flush_flight

```
int(*)(WOLFSSL *ssl) flush_flight;
```

Callback invoked for advisory flushing of the data to send.

Parameters:

- **ssl** - a pointer to a WOLFSSL structure, created using wolfSSL_new().

See: wolfSSL_set_quic_method

Return: 1 on success, 0 on failure.

C.41.4.4 variable send_alert

```
int(*)(WOLFSSL *ssl, WOLFSSL_ENCRYPTION_LEVEL level, uint8_t alert) send_alert;
```

Callback invoked when an SSL alert happened during processing.

Parameters:

- **ssl** - a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **level** - the encryption level in effect when the alert happened
- **alert** - the error

See: wolfSSL_set_quic_method

Return: 1 on success, 0 on failure.

C.41.5 Source code

```
int (*set_encryption_secrets)(WOLFSSL *ssl, WOLFSSL_ENCRYPTION_LEVEL level,
                              const uint8_t *read_secret,
                              const uint8_t *write_secret, size_t secret_len);

int (*add_handshake_data)(WOLFSSL *ssl, WOLFSSL_ENCRYPTION_LEVEL level,
                          const uint8_t *data, size_t len);

int (*flush_flight)(WOLFSSL *ssl);

int (*send_alert)(WOLFSSL *ssl, WOLFSSL_ENCRYPTION_LEVEL level, uint8_t alert);

int wolfSSL_CTX_set_quic_method(WOLFSSL_CTX *ctx, const WOLFSSL_QUIC_METHOD
 ↪  *quic_method);

int wolfSSL_set_quic_method(WOLFSSL *ssl, const WOLFSSL_QUIC_METHOD
 ↪  *quic_method);

int wolfSSL_is_quic(WOLFSSL *ssl);

WOLFSSL_ENCRYPTION_LEVEL wolfSSL_quic_read_level(const WOLFSSL *ssl);

WOLFSSL_ENCRYPTION_LEVEL wolfSSL_quic_write_level(const WOLFSSL *ssl);


void wolfSSL_set_quic_use_legacy_codepoint(WOLFSSL *ssl, int use_legacy);

void wolfSSL_set_quic_transport_version(WOLFSSL *ssl, int version);

int wolfSSL_get_quic_transport_version(const WOLFSSL *ssl);

int wolfSSL_set_quic_transport_params(WOLFSSL *ssl, const uint8_t *params,
 ↪  size_t params_len);

int wolfSSL_get_peer_quic_transport_version(const WOLFSSL *ssl);
```

```
void wolfSSL_get_peer_quic_transport_params(const WOLFSSL *ssl, const uint8_t
 ↪  **out_params, size_t *out_params_len);


void wolfSSL_set_quic_early_data_enabled(WOLFSSL *ssl, int enabled);

size_t wolfSSL_quic_max_handshake_flight_len(const WOLFSSL *ssl,
 ↪  WOLFSSL_ENCRYPTION_LEVEL level);


int wolfSSL_provide_quic_data(WOLFSSL *ssl, WOLFSSL_ENCRYPTION_LEVEL level,
 ↪  const uint8_t *data, size_t len);

WOLFSSL_API int wolfSSL_process_quic_post_handshake(WOLFSSL *ssl);

int wolfSSL_quic_read_write(WOLFSSL *ssl);

const WOLFSSL_EVP_CIPHER *wolfSSL_quic_get_aead(WOLFSSL *ssl);

int wolfSSL_quic_aead_is_gcm(const WOLFSSL_EVP_CIPHER *aead_cipher);

int wolfSSL_quic_aead_is_ccm(const WOLFSSL_EVP_CIPHER *aead_cipher);

int wolfSSL_quic_aead_is_chacha20(const WOLFSSL_EVP_CIPHER *aead_cipher);

WOLFSSL_API size_t wolfSSL_quic_get_aead_tag_len(const WOLFSSL_EVP_CIPHER
 ↪  *aead_cipher);

WOLFSSL_API const WOLFSSL_EVP_MD *wolfSSL_quic_get_md(WOLFSSL *ssl);

const WOLFSSL_EVP_CIPHER *wolfSSL_quic_get_hp(WOLFSSL *ssl);

WOLFSSL_EVP_CIPHER_CTX *wolfSSL_quic_crypt_new(const WOLFSSL_EVP_CIPHER
 ↪  *cipher,
                                               const uint8_t *key, const uint8_t
                                                 ↪  *iv, int encrypt);

int wolfSSL_quic_aead_encrypt(uint8_t *dest, WOLFSSL_EVP_CIPHER_CTX *aead_ctx,
                              const uint8_t *plain, size_t plainlen,
                              const uint8_t *iv, const uint8_t *aad, size_t
                               ↪  aadlen);

int wolfSSL_quic_aead_decrypt(uint8_t *dest, WOLFSSL_EVP_CIPHER_CTX *ctx,
                              const uint8_t *enc, size_t enclen,
                              const uint8_t *iv, const uint8_t *aad, size_t
                               ↪  aadlen);

int wolfSSL_quic_hkdf_extract(uint8_t *dest, const WOLFSSL_EVP_MD *md,
                              const uint8_t *secret, size_t secretlen,
                              const uint8_t *salt, size_t saltlen);

int wolfSSL_quic_hkdf_expand(uint8_t *dest, size_t destlen,
                             const WOLFSSL_EVP_MD *md,
```

```
                            const uint8_t *secret, size_t secretlen,
                            const uint8_t *info, size_t infolen);

int wolfSSL_quic_hkdf(uint8_t *dest, size_t destlen,
                      const WOLFSSL_EVP_MD *md,
                      const uint8_t *secret, size_t secretlen,
                      const uint8_t *salt, size_t saltlen,
                      const uint8_t *info, size_t infolen);
```

C.42 dox_comments/header_files/random.h

C.42.1 Functions

	Name
int	**wc_InitNetRandom**(const char * configFile, wnr_hmac_key hmac_cb, int timeout)Init global Whitewood netRandom context.
int	**wc_FreeNetRandom**(void)Free global Whitewood netRandom context.
int	**wc_InitRng**(WC_RNG *)Gets the seed (from OS) and key cipher for rng. rng_>drbg (deterministic random bit generator) allocated (should be deallocated with wc_FreeRng). This is a blocking operation.
int	**wc_RNG_GenerateBlock**(WC_RNG * rng, byte * b, word32 sz)Copies a sz bytes of pseudorandom data to output. Will reseed rng if needed (blocking).
WC_RNG *	**wc_rng_new**(byte * nonce, word32 nonceSz, void * heap)Creates a new WC_RNG structure.
int	**wc_FreeRng**(WC_RNG *)Should be called when RNG no longer needed in order to securely free drgb. Zeros and XFREEs rng-drbg.
WC_RNG *	**wc_rng_free**(WC_RNG * rng)Should be called when RNG no longer needed in order to securely free rng.
int	**wc_RNG_HealthTest**(int reseed, const byte * entropyA, word32 entropyASz, const byte * entropyB, word32 entropyBSz, byte * output, word32 outputSz)Creates and tests functionality of drbg.

C.42.2 Attributes

	Name
WC_RNG byte *	**b**

C.42.3 Functions Documentation

C.42.3.1 function wc_InitNetRandom

```
int wc_InitNetRandom(
    const char * configFile,
    wnr_hmac_key hmac_cb,
    int timeout
)
```

Init global Whitewood netRandom context.

Parameters:

- **configFile** Path to configuration file
- **hmac_cb** Optional to create HMAC callback.
- **timeout** A timeout duration.

See: wc_FreeNetRandom

Return:

- 0 Success
- BAD_FUNC_ARG Either configFile is null or timeout is negative.
- RNG_FAILURE_E There was a failure initializing the rng.

Example

```
char* config = "path/to/config/example.conf";
int time = // Some sufficient timeout value;

if (wc_InitNetRandom(config, NULL, time) != 0)
{
    // Some error occurred
}
```

C.42.3.2 function wc_FreeNetRandom

```
int wc_FreeNetRandom(
    void
)
```

Free global Whitewood netRandom context.

Parameters:

- **none** No returns.

See: wc_InitNetRandom

Return:

- 0 Success
- BAD_MUTEX_E Error locking mutex on wnr_mutex

Example

```
int ret = wc_FreeNetRandom();
if(ret != 0)
{
    // Handle the error
}
```

C.42.3.3 function wc_InitRng

```
int wc_InitRng(
    WC_RNG *
)
```

Gets the seed (from OS) and key cipher for rng. rng->drbg (deterministic random bit generator) allocated (should be deallocated with wc_FreeRng). This is a blocking operation.

Parameters:

- **rng** random number generator to be initialized for use with a seed and key cipher

See:

- wc_InitRngCavium
- wc_RNG_GenerateBlock
- wc_RNG_GenerateByte
- wc_FreeRng
- wc_RNG_HealthTest

Return:

- 0 on success.
- MEMORY_E XMALLOC failed
- WINCRYPT_E wc_GenerateSeed: failed to acquire context
- CRYPTGEN_E wc_GenerateSeed: failed to get random
- BAD_FUNC_ARG wc_RNG_GenerateBlock input is null or sz exceeds MAX_REQUEST_LEN
- DRBG_CONT_FIPS_E wc_RNG_GenerateBlock: Hash_gen returned DRBG_CONT_FAILURE
- RNG_FAILURE_E wc_RNG_GenerateBlock: Default error. rng's status originally not ok, or set to DRBG_FAILED

Example

```
RNG  rng;
int ret;

#ifdef HAVE_CAVIUM
ret = wc_InitRngCavium(&rng, CAVIUM_DEV_ID);
if (ret != 0){
    printf("RNG Nitrox init for device: %d failed", CAVIUM_DEV_ID);
    return -1;
}
#endif
ret = wc_InitRng(&rng);
if (ret != 0){
    printf("RNG init failed");
    return -1;
}
```

C.42.3.4 function wc_RNG_GenerateBlock

```
int wc_RNG_GenerateBlock(
    WC_RNG * rng,
    byte * b,
    word32 sz
)
```

Copies a sz bytes of pseudorandom data to output. Will reseed rng if needed (blocking).

Parameters:

- **rng** random number generator initialized with wc_InitRng
- **output** buffer to which the block is copied
- **sz** size of output in bytes

See:

- wc_InitRngCavium, wc_InitRng
- wc_RNG_GenerateByte
- wc_FreeRng
- wc_RNG_HealthTest

Return:

- 0 on success
- BAD_FUNC_ARG an input is null or sz exceeds MAX_REQUEST_LEN
- DRBG_CONT_FIPS_E Hash_gen returned DRBG_CONT_FAILURE
- RNG_FAILURE_E Default error. rng's status originally not ok, or set to DRBG_FAILED

Example

```
RNG  rng;
int  sz = 32;
byte block[sz];

int ret = wc_InitRng(&rng);
if (ret != 0) {
    return -1; //init of rng failed!
}

ret = wc_RNG_GenerateBlock(&rng, block, sz);
if (ret != 0) {
    return -1; //generating block failed!
}
```

C.42.3.5 function wc_rng_new

```
WC_RNG * wc_rng_new(
    byte * nonce,
    word32 nonceSz,
    void * heap
)
```

Creates a new WC_RNG structure.

Parameters:

- **heap** pointer to a heap identifier
- **nonce** pointer to the buffer containing the nonce
- **nonceSz** length of the nonce
- **rng** random number generator initialized with wc_InitRng
- **b** one byte buffer to which the block is copied

See:

- wc_InitRng
- wc_rng_free
- wc_FreeRng
- wc_RNG_HealthTest

- wc_InitRngCavium
- wc_InitRng
- wc_RNG_GenerateBlock
- wc_FreeRng
- wc_RNG_HealthTest

Return:

- WC_RNG structure on success
- NULL on error
- 0 on success
- BAD_FUNC_ARG an input is null or sz exceeds MAX_REQUEST_LEN
- DRBG_CONT_FIPS_E Hash_gen returned DRBG_CONT_FAILURE
- RNG_FAILURE_E Default error. rng's status originally not ok, or set to DRBG_FAILED

Example

```
RNG  rng;
byte nonce[] = { initialize nonce };
word32 nonceSz = sizeof(nonce);

wc_rng_new(&nonce, nonceSz, &heap);
```

Calls wc_RNG_GenerateBlock to copy a byte of pseudorandom data to b. Will reseed rng if needed.

Example

```
RNG  rng;
int  sz = 32;
byte b[1];

int ret = wc_InitRng(&rng);
if (ret != 0) {
    return -1; //init of rng failed!
}

ret = wc_RNG_GenerateByte(&rng, b);
if (ret != 0) {
    return -1; //generating block failed!
}
```

C.42.3.6 function wc_FreeRng

```
int wc_FreeRng(
    WC_RNG *
)
```

Should be called when RNG no longer needed in order to securely free drgb. Zeros and XFREEs rng-drbg.

Parameters:

- **rng** random number generator initialized with wc_InitRng

See:

- wc_InitRngCavium
- wc_InitRng
- wc_RNG_GenerateBlock
- wc_RNG_GenerateByte,

- wc_RNG_HealthTest

Return:

- 0 on success
- BAD_FUNC_ARG rng or rng->drgb null
- RNG_FAILURE_E Failed to deallocated drbg

Example

```
RNG  rng;
int ret = wc_InitRng(&rng);
if (ret != 0) {
    return -1; //init of rng failed!
}

int ret = wc_FreeRng(&rng);
if (ret != 0) {
    return -1; //free of rng failed!
}
```

C.42.3.7 function wc_rng_free

```
WC_RNG * wc_rng_free(
    WC_RNG * rng
)
```

Should be called when RNG no longer needed in order to securely free rng.

Parameters:

- **rng** random number generator initialized with wc_InitRng

See:

- wc_InitRng
- wc_rng_new
- wc_FreeRng
- wc_RNG_HealthTest

Example

```
RNG  rng;
byte nonce[] = { initialize nonce };
word32 nonceSz = sizeof(nonce);

rng = wc_rng_new(&nonce, nonceSz, &heap);

// use rng

wc_rng_free(&rng);
```

C.42.3.8 function wc_RNG_HealthTest

```
int wc_RNG_HealthTest(
    int reseed,
    const byte * entropyA,
    word32 entropyASz,
    const byte * entropyB,
    word32 entropyBSz,
```

```
    byte * output,
    word32 outputSz
)
```

Creates and tests functionality of drbg.

Parameters:

- **int** reseed: if set, will test reseed functionality
- **entropyA** entropy to instantiate drgb with
- **entropyASz** size of entropyA in bytes
- **entropyB** If reseed set, drbg will be reseeded with entropyB
- **entropyBSz** size of entropyB in bytes
- **output** initialized to random data seeded with entropyB if seedrandom is set, and entropyA otherwise
- **outputSz** length of output in bytes

See:

- wc_InitRngCavium
- wc_InitRng
- wc_RNG_GenerateBlock
- wc_RNG_GenerateByte
- wc_FreeRng

Return:

- 0 on success
- BAD_FUNC_ARG entropyA and output must not be null. If reseed set entropyB must not be null
- -1 test failed

Example

```
byte output[SHA256_DIGEST_SIZE * 4];
const byte test1EntropyB[] = ....; // test input for reseed false
const byte test1Output[] = ....;   // testvector: expected output of
                                   // reseed false
ret = wc_RNG_HealthTest(0, test1Entropy, sizeof(test1Entropy), NULL, 0,
                    output, sizeof(output));
if (ret != 0)
    return -1;//healthtest without reseed failed

if (XMEMCMP(test1Output, output, sizeof(output)) != 0)
    return -1; //compare to testvector failed: unexpected output

const byte test2EntropyB[] = ....; // test input for reseed
const byte test2Output[] = ....;   // testvector expected output of reseed
ret = wc_RNG_HealthTest(1, test2EntropyA, sizeof(test2EntropyA),
                    test2EntropyB, sizeof(test2EntropyB),
                    output, sizeof(output));

if (XMEMCMP(test2Output, output, sizeof(output)) != 0)
    return -1; //compare to testvector failed
```

C.42.4 Attributes Documentation

C.42.4.1 variable b

```
WC_RNG byte * b;
```

C.42.5 Source code

```
int  wc_InitNetRandom(const char* configFile, wnr_hmac_key hmac_cb, int
 ↪  timeout);

int  wc_FreeNetRandom(void);

int  wc_InitRng(WC_RNG*);

int  wc_RNG_GenerateBlock(WC_RNG* rng, byte* b, word32 sz);

WC_RNG* wc_rng_new(byte* nonce, word32 nonceSz, void* heap)


int  wc_RNG_GenerateByte(WC_RNG* rng, byte* b);

int  wc_FreeRng(WC_RNG*);

WC_RNG* wc_rng_free(WC_RNG* rng);

int wc_RNG_HealthTest(int reseed,
                                        const byte* entropyA, word32 entropyASz,
                                        const byte* entropyB, word32 entropyBSz,
                                         byte* output, word32 outputSz);
```

C.43 dox_comments/header_files/ripemd.h

C.43.1 Functions

	Name
int	**wc_InitRipeMd**(RipeMd *)This function initializes a ripemd structure by initializing ripemd's digest, buffer, loLen and hiLen.
int	**wc_RipeMdUpdate**(RipeMd * ripemd, const byte * data, word32 len)This function generates the RipeMd digest of the data input and stores the result in the ripemd->digest buffer. After running wc_RipeMdUpdate, one should compare the generated ripemd->digest to a known authentication tag to verify the authenticity of a message.
int	**wc_RipeMdFinal**(RipeMd * ripemd, byte * hash)This function copies the computed digest into hash. If there is a partial unhashed block, this method will pad the block with 0s, and include that block's round in the digest before copying to hash. State of ripemd is reset.

C.43.2 Functions Documentation

C.43.2.1 function wc_InitRipeMd

```
int wc_InitRipeMd(
    RipeMd *
)
```

This function initializes a ripemd structure by initializing ripemd's digest, buffer, loLen and hiLen.

Parameters:

- **ripemd** pointer to the ripemd structure to initialize

See:

- wc_RipeMdUpdate
- wc_RipeMdFinal

Return:

- 0 returned on successful execution of the function. The RipeMd structure is initialized.
- BAD_FUNC_ARG returned if the RipeMd structure is NULL.

Example

```
RipeMd md;
int ret;
ret = wc_InitRipeMd(&md);
if (ret != 0) {
    // Failure case.
}
```

C.43.2.2 function wc_RipeMdUpdate

```
int wc_RipeMdUpdate(
    RipeMd * ripemd,
    const byte * data,
    word32 len
)
```

This function generates the RipeMd digest of the data input and stores the result in the ripemd->digest buffer. After running wc_RipeMdUpdate, one should compare the generated ripemd->digest to a known authentication tag to verify the authenticity of a message.

Parameters:

- **ripemd** pointer to the ripemd structure to be initialized with wc_InitRipeMd
- **data** data to be hashed
- **len** sizeof data in bytes

See:

- wc_InitRipeMd
- wc_RipeMdFinal

Return:

- 0 Returned on successful execution of the function.
- BAD_FUNC_ARG Returned if the RipeMd structure is NULL or if data is NULL and len is not zero. This function should execute if data is NULL and len is 0.

Example

```
const byte* data; // The data to be hashed
....
RipeMd md;
```

```
int ret;
ret = wc_InitRipeMd(&md);
if (ret == 0) {
ret = wc_RipeMdUpdate(&md, plain, sizeof(plain));
if (ret != 0) {
// Failure case …
```

C.43.2.3 function wc_RipeMdFinal

```
int wc_RipeMdFinal(
    RipeMd * ripemd,
    byte * hash
)
```

This function copies the computed digest into hash. If there is a partial unhashed block, this method will pad the block with 0s, and include that block's round in the digest before copying to hash. State of ripemd is reset.

Parameters:

- **ripemd** pointer to the ripemd structure to be initialized with wc_InitRipeMd, and containing hashes from wc_RipeMdUpdate. State will be reset
- **hash** buffer to copy digest to. Should be RIPEMD_DIGEST_SIZE bytes

See: none

Return:

- 0 Returned on successful execution of the function. The state of the RipeMd structure has been reset.
- BAD_FUNC_ARG Returned if the RipeMd structure or hash parameters are NULL.

Example

```
RipeMd md;
int ret;
byte   digest[RIPEMD_DIGEST_SIZE];
const byte* data; // The data to be hashed
...
ret = wc_InitRipeMd(&md);
if (ret == 0) {
ret = wc_RipeMdUpdate(&md, plain, sizeof(plain));
    if (ret != 0) {
        // RipeMd Update Failure Case.
}
ret = wc_RipeMdFinal(&md, digest);
if (ret != 0) {
    // RipeMd Final Failure Case.
}...
```

C.43.3 Source code

```
int wc_InitRipeMd(RipeMd*);

int wc_RipeMdUpdate(RipeMd* ripemd, const byte* data, word32 len);

int wc_RipeMdFinal(RipeMd* ripemd, byte* hash);
```

C.44 dox_comments/header_files/rsa.h

C.44.1 Functions

	Name
int	**wc_InitRsaKey**(RsaKey * key, void * heap)This function initializes a provided RsaKey struct. It also takes in a heap identifier, for use with user defined memory overrides (see XMALLOC, XFREE, XREALLOC).
int	**wc_InitRsaKey_Id**(RsaKey * key, unsigned char * id, int len, void * heap, int devId)This function initializes a provided RsaKey struct. The id and len are used to identify the key on the device while the devId identifies the device. It also takes in a heap identifier, for use with user defined memory overrides (see XMALLOC, XFREE, XREALLOC).
int	**wc_RsaSetRNG**(RsaKey * key, WC_RNG * rng)This function associates RNG with Key. It is needed when WC_RSA_BLINDING is enabled.
int	**wc_FreeRsaKey**(RsaKey * key)This function frees a provided RsaKey struct using mp_clear.
int	**wc_RsaDirect**(byte * in, word32 inLen, byte * out, word32 * outSz, RsaKey * key, int type, WC_RNG * rng)Function that does the RSA operation directly with no padding. The input size must match key size. Typically this is used when padding is already done on the RSA input.
int	**wc_RsaPublicEncrypt**(const byte * in, word32 inLen, byte * out, word32 outLen, RsaKey * key, WC_RNG * rng)This function encrypts a message from in and stores the result in out. It requires an initialized public key and a random number generator. As a side effect, this function will return the bytes written to out in outLen.
int	**wc_RsaPrivateDecryptInline**(byte * in, word32 inLen, byte ** out, RsaKey * key)This functions is utilized by the wc_RsaPrivateDecrypt function for decrypting.
int	**wc_RsaPrivateDecrypt**(const byte * in, word32 inLen, byte * out, word32 outLen, RsaKey * key)This functions provides private RSA decryption.
int	**wc_RsaSSL_Sign**(const byte * in, word32 inLen, byte * out, word32 outLen, RsaKey * key, WC_RNG * rng)Signs the provided array with the private key.

	Name
int	**wc_RsaSSL_VerifyInline**(byte * in, word32 inLen, byte ** out, RsaKey * key)Used to verify that the message was signed by RSA key. The output uses the same byte array as the input.
int	**wc_RsaSSL_Verify**(const byte * in, word32 inLen, byte * out, word32 outLen, RsaKey * key)Used to verify that the message was signed by key.
int	**wc_RsaPSS_Sign**(const byte * in, word32 inLen, byte * out, word32 outLen, enum wc_HashType hash, int mgf, RsaKey * key, WC_RNG * rng)Signs the provided array with the private key.
int	**wc_RsaPSS_Verify**(byte * in, word32 inLen, byte * out, word32 outLen, enum wc_HashType hash, int mgf, RsaKey * key)Decrypt input signature to verify that the message was signed by key. The key has to be associated with RNG by wc_RsaSetRNG when WC_RSA_BLINDING is enabled.
int	**wc_RsaPSS_VerifyInline**(byte * in, word32 inLen, byte ** out, enum wc_HashType hash, int mgf, RsaKey * key)Decrypt input signature to verify that the message was signed by RSA key. The output uses the same byte array as the input. The key has to be associated with RNG by wc_RsaSetRNG when WC_RSA_BLINDING is enabled.
int	**wc_RsaPSS_VerifyCheck**(byte * in, word32 inLen, byte * out, word32 outLen, const byte * digest, word32 digestLen, enum wc_HashType hash, int mgf, RsaKey * key)Verify the message signed with RSA-PSS. Salt length is equal to hash length. The key has to be associated with RNG by wc_RsaSetRNG when WC_RSA_BLINDING is enabled.
int	**wc_RsaPSS_VerifyCheck_ex**(byte * in, word32 inLen, byte * out, word32 outLen, const byte * digest, word32 digestLen, enum wc_HashType hash, int mgf, int saltLen, RsaKey * key)Verify the message signed with RSA-PSS. The key has to be associated with RNG by wc_RsaSetRNG when WC_RSA_BLINDING is enabled.
int	**wc_RsaPSS_VerifyCheckInline**(byte * in, word32 inLen, byte ** out, const byte * digest, word32 digentLen, enum wc_HashType hash, int mgf, RsaKey * key)Verify the message signed with RSA-PSS. The input buffer is reused for the output buffer. Salt length is equal to hash length.

	Name
int	**wc_RsaPSS_VerifyCheckInline_ex**(byte * in, word32 inLen, byte ** out, const byte * digest, word32 digentLen, enum wc_HashType hash, int mgf, int saltLen, RsaKey * key)Verify the message signed with RSA-PSS. The input buffer is reused for the output buffer. The key has to be associated with RNG by wc_RsaSetRNG when WC_RSA_BLINDING is enabled.
int	**wc_RsaPSS_CheckPadding**(const byte * in, word32 inLen, byte * sig, word32 sigSz, enum wc_HashType hashType)Checks the PSS data to ensure that the signature matches. Salt length is equal to hash length. The key has to be associated with RNG by wc_RsaSetRNG when WC_RSA_BLINDING is enabled.
int	**wc_RsaPSS_CheckPadding_ex**(const byte * in, word32 inLen, byte * sig, word32 sigSz, enum wc_HashType hashType, int saltLen, int bits)Checks the PSS data to ensure that the signature matches. Salt length is equal to hash length.
int	**wc_RsaEncryptSize**(RsaKey * key)Returns the encryption size for the provided key structure.
int	**wc_RsaPrivateKeyDecode**(const byte * input, word32 * inOutIdx, RsaKey * key, word32 inSz)This function parses a DER-formatted RSA private key, extracts the private key and stores it in the given RsaKey structure. It also sets the distance parsed in idx.
int	**wc_RsaPublicKeyDecode**(const byte * input, word32 * inOutIdx, RsaKey * key, word32 inSz)This function parses a DER-formatted RSA public key, extracts the public key and stores it in the given RsaKey structure. It also sets the distance parsed in idx.
int	**wc_RsaPublicKeyDecodeRaw**(const byte * n, word32 nSz, const byte * e, word32 eSz, RsaKey * key)This function decodes the raw elements of an RSA public key, taking in the public modulus (n) and exponent (e). It stores these raw elements in the provided RsaKey structure, allowing one to use them in the encryption/decryption process.
int	**wc_RsaKeyToDer**(RsaKey * key, byte * output, word32 inLen)This function converts an RsaKey key to DER format. The result is written to output and it returns the number of bytes written.

	Name
int	**wc_RsaPublicEncrypt_ex**(const byte * in, word32 inLen, byte * out, word32 outLen, RsaKey * key, WC_RNG * rng, int type, enum wc_HashType hash, int mgf, byte * label, word32 labelSz)This function performs RSA encrypt while allowing the choice of which padding to use.
int	**wc_RsaPrivateDecrypt_ex**(const byte * in, word32 inLen, byte * out, word32 outLen, RsaKey * key, int type, enum wc_HashType hash, int mgf, byte * label, word32 labelSz)This function uses RSA to decrypt a message and gives the option of what padding type.
int	**wc_RsaPrivateDecryptInline_ex**(byte * in, word32 inLen, byte ** out, RsaKey * key, int type, enum wc_HashType hash, int mgf, byte * label, word32 labelSz)This function uses RSA to decrypt a message inline and gives the option of what padding type. The in buffer will contain the decrypted message after being called and the out byte pointer will point to the location in the “in” buffer where the plain text is.
int	**wc_RsaFlattenPublicKey**(RsaKey * key, byte * e, word32 * eSz, byte * n, word32 * nSz)Flattens the RsaKey structure into individual elements (e, n) used for the RSA algorithm.
int	**wc_RsaKeyToPublicDer**(RsaKey * key, byte * output, word32 inLen)Convert Rsa Public key to DER format. Writes to output, and returns count of bytes written.
int	**wc_RsaKeyToPublicDer_ex**(RsaKey * key, byte * output, word32 inLen, int with_header)Convert RSA Public key to DER format. Writes to output, and returns count of bytes written. If with_header is 0 then only the (seq + n + e) is returned in ASN.1 DER format and will exclude the header.
int	**wc_MakeRsaKey**(RsaKey * key, int size, long e, WC_RNG * rng)This function generates a RSA private key of length size (in bits) and given exponent (e). It then stores this key in the provided RsaKey structure, so that it may be used for encryption/decryption. A secure number to use for e is 65537. size is required to be greater than RSA_MIN_SIZE and less than RSA_MAX_SIZE. For this function to be available, the option WOLFSSL_KEY_GEN must be enabled at compile time. This can be accomplished with –enable-keygen if using ./configure.

	Name
int	**wc_RsaSetNonBlock**(RsaKey * key, RsaNb * nb)This function sets the non-blocking RSA context. When a RsaNb context is set it enables fast math based non-blocking exptmod, which splits the RSA function into many smaller operations. Enabled when WC_RSA_NONBLOCK is defined.
int	**wc_RsaSetNonBlockTime**(RsaKey * key, word32 maxBlockUs, word32 cpuMHz)This function configures the maximum amount of blocking time in microseconds. It uses a pre_computed table (see tfm.c exptModNbInst) along with the CPU speed in megahertz to determine if the next operation can be completed within the maximum blocking time provided. Enabled when WC_RSA_NONBLOCK_TIME is defined.

C.44.2 Functions Documentation

C.44.2.1 function wc_InitRsaKey

```
int wc_InitRsaKey(
    RsaKey * key,
    void * heap
)
```

This function initializes a provided RsaKey struct. It also takes in a heap identifier, for use with user defined memory overrides (see XMALLOC, XFREE, XREALLOC).

Parameters:

- **key** pointer to the RsaKey structure to initialize
- **heap** pointer to a heap identifier, for use with memory overrides, allowing custom handling of memory allocation. This heap will be the default used when allocating memory for use with this RSA object

See:

- wc_FreeRsaKey
- wc_RsaSetRNG

Return:

- 0 Returned upon successfully initializing the RSA structure for use with encryption and decryption
- BAD_FUNC_ARGS Returned if the RSA key pointer evaluates to NULL

The key has to be associated with RNG by wc_RsaSetRNG when WC_RSA_BLINDING is enabled.

Example

```
RsaKey enc;
int ret;
ret = wc_InitRsaKey(&enc, NULL); // not using heap hint. No custom memory
if ( ret != 0 ) {
    // error initializing RSA key
}
```

C.44.2.2 function wc_InitRsaKey_Id

```
int wc_InitRsaKey_Id(
    RsaKey * key,
    unsigned char * id,
    int len,
    void * heap,
    int devId
)
```

This function initializes a provided RsaKey struct. The id and len are used to identify the key on the device while the devId identifies the device. It also takes in a heap identifier, for use with user defined memory overrides (see XMALLOC, XFREE, XREALLOC).

Parameters:

- **key** pointer to the RsaKey structure to initialize
- **id** identifier of key on device
- **len** length of identifier in bytes
- **heap** pointer to a heap identifier, for use with memory overrides, allowing custom handling of memory allocation. This heap will be the default used when allocating memory for use with this RSA object
- **devId** ID to use with crypto callbacks or async hardware. Set to INVALID_DEVID (-2) if not used

See:

- wc_InitRsaKey
- wc_FreeRsaKey
- wc_RsaSetRNG

Return:

- 0 Returned upon successfully initializing the RSA structure for use with encryption and decryption
- BAD_FUNC_ARGS Returned if the RSA key pointer evaluates to NULL
- BUFFER_E Returned if len is less than 0 or greater than RSA_MAX_ID_LEN.

The key has to be associated with RNG by wc_RsaSetRNG when WC_RSA_BLINDING is enabled.

Example

```
RsaKey enc;
unsigned char* id = (unsigned char*)"RSA2048";
int len = 6;
int devId = 1;
int ret;
ret = wc_CryptoDev_RegisterDevice(devId, wc_Pkcs11_CryptoDevCb,
                                  &token);
if ( ret != 0) {
    // error associating callback and token with device id
}
ret = wc_InitRsaKey_Id(&enc, id, len, NULL, devId); // not using heap hint
if ( ret != 0 ) {
    // error initializing RSA key
}
```

C.44.2.3 function wc_RsaSetRNG

```
int wc_RsaSetRNG(
    RsaKey * key,
```

```
    WC_RNG * rng
)
```

This function associates RNG with Key. It is needed when WC_RSA_BLINDING is enabled.

Parameters:

- **key** pointer to the RsaKey structure to be associated
- **rng** pointer to the WC_RNG structure to associate with

See:

- wc_InitRsaKey
- wc_RsaSetRNG

Return:

- 0 Returned upon success
- BAD_FUNC_ARGS Returned if the RSA key, rng pointer evaluates to NULL

Example

```
ret = wc_InitRsaKey(&key, NULL);
if (ret == 0) {
    ret = wc_InitRng(&rng);
} else return -1;
if (ret == 0) {
    ret = wc_RsaSetRNG(&key, &rng);
```

C.44.2.4 function wc_FreeRsaKey

```
int wc_FreeRsaKey(
    RsaKey * key
)
```

This function frees a provided RsaKey struct using mp_clear.

Parameters:

- **key** pointer to the RsaKey structure to free

See: wc_InitRsaKey

Return: 0 Returned upon successfully freeing the key

Example

```
RsaKey enc;
wc_InitRsaKey(&enc, NULL); // not using heap hint. No custom memory
... set key, do encryption

wc_FreeRsaKey(&enc);
```

C.44.2.5 function wc_RsaDirect

```
int wc_RsaDirect(
    byte * in,
    word32 inLen,
    byte * out,
    word32 * outSz,
    RsaKey * key,
    int type,
```

```
    WC_RNG * rng
)
```

Function that does the RSA operation directly with no padding. The input size must match key size. Typically this is used when padding is already done on the RSA input.

Parameters:

- **in** buffer to do operation on
- **inLen** length of input buffer
- **out** buffer to hold results
- **outSz** gets set to size of result buffer. Should be passed in as length of out buffer. If the pointer "out" is null then outSz gets set to the expected buffer size needed and LENGTH_ONLY_E gets returned.
- **key** initialized RSA key to use for encrypt/decrypt
- **type** if using private or public key (RSA_PUBLIC_ENCRYPT, RSA_PUBLIC_DECRYPT, RSA_PRIVATE_ENCRYPT, RSA_PRIVATE_DECRYPT)
- **rng** initialized WC_RNG struct

See:

- wc_RsaPublicEncrypt
- wc_RsaPrivateDecrypt

Return:

- size On successfully encryption the size of the encrypted buffer is returned
- RSA_BUFFER_E RSA buffer error, output too small or input too large

Example

```
int ret;
WC_RNG rng;
RsaKey key;
byte  in[256];
byte out[256];
word32 outSz = (word32)sizeof(out);
…

ret = wc_RsaDirect(in, (word32)sizeof(in), out, &outSz, &key,
    RSA_PRIVATE_ENCRYPT, &rng);
if (ret < 0) {
    //handle error
}
```

C.44.2.6 function wc_RsaPublicEncrypt

```
int wc_RsaPublicEncrypt(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 outLen,
    RsaKey * key,
    WC_RNG * rng
)
```

This function encrypts a message from in and stores the result in out. It requires an initialized public key and a random number generator. As a side effect, this function will return the bytes written to out in outLen.

Parameters:

- **in** pointer to a buffer containing the input message to encrypt
- **inLen** the length of the message to encrypt
- **out** pointer to the buffer in which to store the output ciphertext
- **outLen** the length of the output buffer
- **key** pointer to the RsaKey structure containing the public key to use for encryption
- **rng** The RNG structure with which to generate random block padding

See: wc_RsaPrivateDecrypt

Return:

- Success Upon successfully encrypting the input message, returns 0 for success and less than zero for failure. Also returns the number bytes written to out by storing the value in outLen
- BAD_FUNC_ARG Returned if any of the input parameters are invalid
- RSA_BUFFER_E Returned if the output buffer is too small to store the ciphertext
- RNG_FAILURE_E Returned if there is an error generating a random block using the provided RNG structure
- MP_INIT_E May be returned if there is an error in the math library used while encrypting the message
- MP_READ_E May be returned if there is an error in the math library used while encrypting the message
- MP_CMP_E May be returned if there is an error in the math library used while encrypting the message
- MP_INVMOD_E May be returned if there is an error in the math library used while encrypting the message
- MP_EXPTMOD_E May be returned if there is an error in the math library used while encrypting the message
- MP_MOD_E May be returned if there is an error in the math library used while encrypting the message
- MP_MUL_E May be returned if there is an error in the math library used while encrypting the message
- MP_ADD_E May be returned if there is an error in the math library used while encrypting the message
- MP_MULMOD_E May be returned if there is an error in the math library used while encrypting the message
- MP_TO_E May be returned if there is an error in the math library used while encrypting the message
- MP_MEM May be returned if there is an error in the math library used while encrypting the message
- MP_ZERO_E May be returned if there is an error in the math library used while encrypting the message

Example

```
RsaKey pub;
int ret = 0;
byte n[] = { // initialize with received n component of public key };
byte e[] = { // initialize with received e component of public key };
byte msg[] = { // initialize with plaintext of message to encrypt };
byte cipher[256]; // 256 bytes is large enough to store 2048 bit RSA
ciphertext

wc_InitRsaKey(&pub, NULL); // not using heap hint. No custom memory
wc_RsaPublicKeyDecodeRaw(n, sizeof(n), e, sizeof(e), &pub);
// initialize with received public key parameters
```

```
ret = wc_RsaPublicEncrypt(msg, sizeof(msg), out, sizeof(out), &pub, &rng);
if ( ret != 0 ) {
    // error encrypting message
}
```

C.44.2.7 function wc_RsaPrivateDecryptInline

```
int wc_RsaPrivateDecryptInline(
    byte * in,
    word32 inLen,
    byte ** out,
    RsaKey * key
)
```

This functions is utilized by the wc_RsaPrivateDecrypt function for decrypting.

Parameters:

- **in** The byte array to be decrypted.
- **inLen** The length of in.
- **out** The byte array for the decrypted data to be stored.
- **key** The key to use for decryption.

See: wc_RsaPrivateDecrypt

Return:

- Success Length of decrypted data.
- RSA_PAD_E RsaUnPad error, bad formatting

Example

none

C.44.2.8 function wc_RsaPrivateDecrypt

```
int wc_RsaPrivateDecrypt(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 outLen,
    RsaKey * key
)
```

This functions provides private RSA decryption.

Parameters:

- **in** The byte array to be decrypted.
- **inLen** The length of in.
- **out** The byte array for the decrypted data to be stored.
- **outLen** The length of out.
- **key** The key to use for decryption.

See:

- RsaUnPad
- wc_RsaFunction
- wc_RsaPrivateDecryptInline

Return:

- Success length of decrypted data.
- MEMORY_E -125, out of memory error
- BAD_FUNC_ARG -173, Bad function argument provided

Example

```
ret = wc_RsaPublicEncrypt(in, inLen, out, sizeof(out), &key, &rng);
if (ret < 0) {
    return -1;
}
ret = wc_RsaPrivateDecrypt(out, ret, plain, sizeof(plain), &key);
if (ret < 0) {
    return -1;
}
```

C.44.2.9 function wc_RsaSSL_Sign

```
int wc_RsaSSL_Sign(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 outLen,
    RsaKey * key,
    WC_RNG * rng
)
```

Signs the provided array with the private key.

Parameters:

- **in** The byte array to be encrypted.
- **inLen** The length of in.
- **out** The byte array for the encrypted data to be stored.
- **outLen** The length of out.
- **key** The key to use for encryption.
- **RNG** The RNG struct to use for random number purposes.

See: wc_RsaPad

Return: RSA_BUFFER_E: -131, RSA buffer error, output too small or input too large

Example

```
ret = wc_RsaSSL_Sign(in, inLen, out, sizeof(out), &key, &rng);
if (ret < 0) {
    return -1;
}
memset(plain, 0, sizeof(plain));
ret = wc_RsaSSL_Verify(out, ret, plain, sizeof(plain), &key);
if (ret < 0) {
    return -1;
}
if (ret != inLen) {
    return -1;
}
if (XMEMCMP(in, plain, ret) != 0) {
    return -1;
}
```

C.44.2.10 function wc_RsaSSL_VerifyInline

```
int wc_RsaSSL_VerifyInline(
    byte * in,
    word32 inLen,
    byte ** out,
    RsaKey * key
)
```

Used to verify that the message was signed by RSA key. The output uses the same byte array as the input.

Parameters:

- **in** Byte array to be decrypted.
- **inLen** Length of the buffer input.
- **out** Pointer to a pointer for decrypted information.
- **key** RsaKey to use.

See:

- wc_RsaSSL_Verify
- wc_RsaSSL_Sign

Return:

- 0 Length of text.
- <0 An error occurred.

Example

```
RsaKey key;
WC_RNG rng;
int ret = 0;
long e = 65537; // standard value to use for exponent
wc_InitRsaKey(&key, NULL); // not using heap hint. No custom memory
wc_InitRng(&rng);
wc_MakeRsaKey(&key, 2048, e, &rng);

byte in[] = { // Initialize with some RSA encrypted information }
byte* out;
if(wc_RsaSSL_VerifyInline(in, sizeof(in), &out, &key) < 0)
{
    // handle error
}
```

C.44.2.11 function wc_RsaSSL_Verify

```
int wc_RsaSSL_Verify(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 outLen,
    RsaKey * key
)
```

Used to verify that the message was signed by key.

Parameters:

- **in** The byte array to be decrypted.
- **inLen** The length of in.
- **out** The byte array for the decrypted data to be stored.
- **outLen** The length of out.
- **key** The key to use for verification.

See: wc_RsaSSL_Sign

Return:

- Success Length of text on no error.
- MEMORY_E memory exception.

Example

```
ret = wc_RsaSSL_Sign(in, inLen, out, sizeof(out), &key, &rng);
if (ret < 0) {
    return -1;
}
memset(plain, 0, sizeof(plain));
ret = wc_RsaSSL_Verify(out, ret, plain, sizeof(plain), &key);
if (ret < 0) {
    return -1;
}
if (ret != inLen) {
    return -1;
}
if (XMEMCMP(in, plain, ret) != 0) {
    return -1;
}
```

C.44.2.12 function wc_RsaPSS_Sign

```
int wc_RsaPSS_Sign(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 outLen,
    enum wc_HashType hash,
    int mgf,
    RsaKey * key,
    WC_RNG * rng
)
```

Signs the provided array with the private key.

Parameters:

- **in** The byte array to be encrypted.
- **inLen** The length of in.
- **out** The byte array for the encrypted data to be stored.
- **outLen** The length of out.
- **hash** The hash type to be in message
- **mgf** Mask Generation Function Identifiers
- **key** The key to use for verification.

See:

- wc_RsaPSS_Verify

- wc_RsaSetRNG

Return: RSA_BUFFER_E: -131, RSA buffer error, output too small or input too large

Example

```
ret = wc_InitRsaKey(&key, NULL);
if (ret == 0) {
    ret = wc_InitRng(&rng);
} else return -1;
if (ret == 0) {
    ret = wc_RsaSetRNG(&key, &rng);
} else return -1;
if (ret == 0) {
        ret = wc_MakeRsaKey(&key, 2048, WC_RSA_EXPONENT, &rng);
} else return -1;

ret = wc_RsaPSS_Sign((byte*)szMessage, (word32)XSTRLEN(szMessage)+1,
        pSignature, sizeof(pSignature),
        WC_HASH_TYPE_SHA256, WC_MGF1SHA256, &key, &rng);
if (ret > 0 ){
    sz = ret;
} else return -1;

ret = wc_RsaPSS_Verify(pSignature, sz, pt, outLen,
        WC_HASH_TYPE_SHA256, WC_MGF1SHA256, &key);
if (ret <= 0)return -1;

wc_FreeRsaKey(&key);
wc_FreeRng(&rng);
```

C.44.2.13 function wc_RsaPSS_Verify

```
int wc_RsaPSS_Verify(
    byte * in,
    word32 inLen,
    byte * out,
    word32 outLen,
    enum wc_HashType hash,
    int mgf,
    RsaKey * key
)
```

Decrypt input signature to verify that the message was signed by key. The key has to be associated with RNG by wc_RsaSetRNG when WC_RSA_BLINDING is enabled.

Parameters:

- **in** The byte array to be decrypted.
- **inLen** The length of in.
- **out** The byte array for the decrypted data to be stored.
- **outLen** The length of out.
- **hash** The hash type to be in message
- **mgf** Mask Generation Function Identifiers
- **key** The key to use for verification.

See:

- wc_RsaPSS_Sign
- wc_RsaPSS_VerifyInline
- wc_RsaPSS_CheckPadding
- wc_RsaSetRNG

Return:

- Success Length of text on no error.
- MEMORY_E memory exception.
- MP_EXPTMOD_E - When using fastmath and FP_MAX_BITS not set to at least 2 times the keySize (Example when using 4096-bit key set FP_MAX_BITS to 8192 or greater value)

Example

```
ret = wc_InitRsaKey(&key, NULL);
if (ret == 0) {
    ret = wc_InitRng(&rng);
} else return -1;
if (ret == 0) {
    ret = wc_RsaSetRNG(&key, &rng);
} else return -1;
if (ret == 0) {
        ret = wc_MakeRsaKey(&key, 2048, WC_RSA_EXPONENT, &rng);
} else return -1;
ret = wc_RsaPSS_Sign((byte*)szMessage, (word32)XSTRLEN(szMessage)+1,
        pSignature, sizeof(pSignature),
        WC_HASH_TYPE_SHA256, WC_MGF1SHA256, &key, &rng);
if (ret > 0 ){
    sz = ret;
} else return -1;

ret = wc_RsaPSS_Verify(pSignature, sz, pt, outLen,
        WC_HASH_TYPE_SHA256, WC_MGF1SHA256, &key);
if (ret <= 0)return -1;

wc_FreeRsaKey(&key);
wc_FreeRng(&rng);
```

C.44.2.14 function wc_RsaPSS_VerifyInline

```
int wc_RsaPSS_VerifyInline(
    byte * in,
    word32 inLen,
    byte ** out,
    enum wc_HashType hash,
    int mgf,
    RsaKey * key
)
```

Decrypt input signature to verify that the message was signed by RSA key. The output uses the same byte array as the input. The key has to be associated with RNG by wc_RsaSetRNG when WC_RSA_BLINDING is enabled.

Parameters:

- **in** Byte array to be decrypted.
- **inLen** Length of the buffer input.
- **out** Pointer to address containing the PSS data.

- **hash** The hash type to be in message
- **mgf** Mask Generation Function Identifiers
- **key** RsaKey to use.

See:

- wc_RsaPSS_Verify
- wc_RsaPSS_Sign
- wc_RsaPSS_VerifyCheck
- wc_RsaPSS_VerifyCheck_ex
- wc_RsaPSS_VerifyCheckInline
- wc_RsaPSS_VerifyCheckInline_ex
- wc_RsaPSS_CheckPadding
- wc_RsaPSS_CheckPadding_ex
- wc_RsaSetRNG

Return:

- 0 Length of text.
- <0 An error occurred.

Example

```
ret = wc_InitRsaKey(&key, NULL);
if (ret == 0) {
    ret = wc_InitRng(&rng);
} else return -1;
if (ret == 0) {
    ret = wc_RsaSetRNG(&key, &rng);
} else return -1;
if (ret == 0) {
        ret = wc_MakeRsaKey(&key, 2048, WC_RSA_EXPONENT, &rng);
} else return -1;
ret = wc_RsaPSS_Sign(digest, digestSz, pSignature, pSignatureSz,
        WC_HASH_TYPE_SHA256, WC_MGF1SHA256, &key, &rng);
if (ret > 0 ){
    sz = ret;
} else return -1;

ret = wc_RsaPSS_VerifyInline(pSignature, sz, pt,
        WC_HASH_TYPE_SHA256, WC_MGF1SHA256, &key);
if (ret <= 0)return -1;

wc_FreeRsaKey(&key);
wc_FreeRng(&rng);
```

C.44.2.15 function wc_RsaPSS_VerifyCheck

```
int wc_RsaPSS_VerifyCheck(
    byte * in,
    word32 inLen,
    byte * out,
    word32 outLen,
    const byte * digest,
    word32 digestLen,
    enum wc_HashType hash,
    int mgf,
```

```
    RsaKey * key
)
```

Verify the message signed with RSA-PSS. Salt length is equal to hash length. The key has to be associated with RNG by wc_RsaSetRNG when WC_RSA_BLINDING is enabled.

Parameters:

- **in** The byte array to be decrypted.
- **inLen** The length of in.
- **out** Pointer to address containing the PSS data.
- **outLen** The length of out.
- **digest** Hash of the data that is being verified.
- **digestLen** Length of hash.
- **hash** Hash algorithm.
- **mgf** Mask generation function.
- **key** Public RSA key.

See:

- wc_RsaPSS_Sign
- wc_RsaPSS_Verify
- wc_RsaPSS_VerifyCheck_ex
- wc_RsaPSS_VerifyCheckInline
- wc_RsaPSS_VerifyCheckInline_ex
- wc_RsaPSS_CheckPadding
- wc_RsaPSS_CheckPadding_ex
- wc_RsaSetRNG

Return:

- the length of the PSS data on success and negative indicates failure.
- MEMORY_E memory exception.

Example

```
ret = wc_InitRsaKey(&key, NULL);
if (ret == 0) {
    ret = wc_InitRng(&rng);
} else return -1;
if (ret == 0) {
    ret = wc_RsaSetRNG(&key, &rng);
} else return -1;
if (ret == 0) {
        ret = wc_MakeRsaKey(&key, 2048, WC_RSA_EXPONENT, &rng);
} else return -1;

if (ret == 0) {
    digestSz = wc_HashGetDigestSize(WC_HASH_TYPE_SHA256);
    ret = wc_Hash(WC_HASH_TYPE_SHA256, message, sz, digest, digestSz);
} else return -1;

if (ret == 0) {
    ret = wc_RsaPSS_Sign(digest, digestSz, pSignature, pSignatureSz,
            WC_HASH_TYPE_SHA256, WC_MGF1SHA256, &key, &rng);
    if (ret > 0 ){
        sz = ret;
    } else return -1;
} else return -1;
```

```
if (ret == 0) {
    ret = wc_RsaPSS_VerifyCheck(pSignature, sz, pt, outLen,
            digest, digestSz, WC_HASH_TYPE_SHA256, WC_MGF1SHA256, &key);
        if (ret <= 0) return -1;
} else return -1;

wc_FreeRsaKey(&key);
wc_FreeRng(&rng);
```

C.44.2.16 function wc_RsaPSS_VerifyCheck_ex

```
int wc_RsaPSS_VerifyCheck_ex(
    byte * in,
    word32 inLen,
    byte * out,
    word32 outLen,
    const byte * digest,
    word32 digestLen,
    enum wc_HashType hash,
    int mgf,
    int saltLen,
    RsaKey * key
)
```

Verify the message signed with RSA-PSS. The key has to be associated with RNG by wc_RsaSetRNG when WC_RSA_BLINDING is enabled.

Parameters:

- **in** The byte array to be decrypted.
- **inLen** The length of in.
- **out** Pointer to address containing the PSS data.
- **outLen** The length of out.
- **digest** Hash of the data that is being verified.
- **digestLen** Length of hash.
- **hash** Hash algorithm.
- **mgf** Mask generation function.
- **saltLen** Length of salt used. RSA_PSS_SALT_LEN_DEFAULT (-1) indicates salt length is the same as the hash length. RSA_PSS_SALT_LEN_DISCOVER indicates salt length is determined from the data.
- **key** Public RSA key.

See:

- wc_RsaPSS_Sign
- wc_RsaPSS_Verify
- wc_RsaPSS_VerifyCheck
- wc_RsaPSS_VerifyCheckInline
- wc_RsaPSS_VerifyCheckInline_ex
- wc_RsaPSS_CheckPadding
- wc_RsaPSS_CheckPadding_ex
- wc_RsaSetRNG

Return:

- the length of the PSS data on success and negative indicates failure.
- MEMORY_E memory exception.

Example

```
ret = wc_InitRsaKey(&key, NULL);
if (ret == 0) {
    ret = wc_InitRng(&rng);
} else return -1;
if (ret == 0) {
    ret = wc_RsaSetRNG(&key, &rng);
} else return -1;
if (ret == 0) {
        ret = wc_MakeRsaKey(&key, 2048, WC_RSA_EXPONENT, &rng);
} else return -1;

if (ret == 0) {
    digestSz = wc_HashGetDigestSize(WC_HASH_TYPE_SHA256);
    ret = wc_Hash(WC_HASH_TYPE_SHA256, message, sz, digest, digestSz);
} else return -1;

if (ret == 0) {
    ret = wc_RsaPSS_Sign(digest, digestSz, pSignature, pSignatureSz,
            WC_HASH_TYPE_SHA256, WC_MGF1SHA256, &key, &rng);
    if (ret > 0 ){
        sz = ret;
    } else return -1;
} else return -1;
if (ret == 0) {
    ret = wc_RsaPSS_VerifyCheck_ex(pSignature, sz, pt, outLen,
            digest, digestSz, WC_HASH_TYPE_SHA256, WC_MGF1SHA256, saltLen,
 ↪  &key);
        if (ret <= 0) return -1;
} else return -1;

wc_FreeRsaKey(&key);
wc_FreeRng(&rng);
```

C.44.2.17 function wc_RsaPSS_VerifyCheckInline

```
int wc_RsaPSS_VerifyCheckInline(
    byte * in,
    word32 inLen,
    byte ** out,
    const byte * digest,
    word32 digentLen,
    enum wc_HashType hash,
    int mgf,
    RsaKey * key
)
```

Verify the message signed with RSA-PSS. The input buffer is reused for the output buffer. Salt length is equal to hash length.

Parameters:

- **in** The byte array to be decrypted.
- **inLen** The length of in.
- **out** The byte array for the decrypted data to be stored.

- **digest** Hash of the data that is being verified.
- **digestLen** Length of hash.
- **hash** The hash type to be in message
- **mgf** Mask Generation Function Identifiers
- **key** The key to use for verification.

See:

- wc_RsaPSS_Sign
- wc_RsaPSS_Verify
- wc_RsaPSS_VerifyCheck
- wc_RsaPSS_VerifyCheck_ex
- wc_RsaPSS_VerifyCheckInline_ex
- wc_RsaPSS_CheckPadding
- wc_RsaPSS_CheckPadding_ex
- wc_RsaSetRNG

Return: the length of the PSS data on success and negative indicates failure.

The key has to be associated with RNG by wc_RsaSetRNG when WC_RSA_BLINDING is enabled.

Example

```
ret = wc_InitRsaKey(&key, NULL);
if (ret == 0) {
    ret = wc_InitRng(&rng);
} else return -1;
if (ret == 0) {
    ret = wc_RsaSetRNG(&key, &rng);
} else return -1;
if (ret == 0) {
        ret = wc_MakeRsaKey(&key, 2048, WC_RSA_EXPONENT, &rng);
} else return -1;

if (ret == 0) {
    digestSz = wc_HashGetDigestSize(WC_HASH_TYPE_SHA256);
    ret = wc_Hash(WC_HASH_TYPE_SHA256, message, sz, digest, digestSz);
} else return -1;

if (ret == 0) {
    ret = wc_RsaPSS_Sign(digest, digestSz, pSignature, pSignatureSz,
            WC_HASH_TYPE_SHA256, WC_MGF1SHA256, &key, &rng);
    if (ret > 0 ){
        sz = ret;
    } else return -1;
} else return -1;
if (ret == 0) {
    ret = wc_RsaPSS_VerifyCheckInline(pSignature, sz, pt,
            digest, digestSz, WC_HASH_TYPE_SHA256, WC_MGF1SHA256, &key);
        if (ret <= 0) return -1;
} else return -1;

wc_FreeRsaKey(&key);
wc_FreeRng(&rng);
```

C.44.2.18 function wc_RsaPSS_VerifyCheckInline_ex

```
int wc_RsaPSS_VerifyCheckInline_ex(
    byte * in,
    word32 inLen,
    byte ** out,
    const byte * digest,
    word32 digentLen,
    enum wc_HashType hash,
    int mgf,
    int saltLen,
    RsaKey * key
)
```

Verify the message signed with RSA-PSS. The input buffer is reused for the output buffer. The key has to be associated with RNG by wc_RsaSetRNG when WC_RSA_BLINDING is enabled.

Parameters:

- **in** The byte array to be decrypted.
- **inLen** The length of in.
- **out** The byte array for the decrypted data to be stored.
- **digest** Hash of the data that is being verified.
- **digestLen** Length of hash.
- **hash** The hash type to be in message
- **mgf** Mask Generation Function Identifiers
- **saltLen** Length of salt used. RSA_PSS_SALT_LEN_DEFAULT (-1) indicates salt length is the same as the hash length. RSA_PSS_SALT_LEN_DISCOVER indicates salt length is determined from the data.
- **key** The key to use for verification.

See:

- wc_RsaPSS_Sign
- wc_RsaPSS_Verify
- wc_RsaPSS_VerifyCheck
- wc_RsaPSS_VerifyCheck_ex
- wc_RsaPSS_VerifyCheckInline
- wc_RsaPSS_CheckPadding
- wc_RsaPSS_CheckPadding_ex
- wc_RsaSetRNG

Return: the length of the PSS data on success and negative indicates failure.

Example

```
ret = wc_InitRsaKey(&key, NULL);
if (ret == 0) {
    ret = wc_InitRng(&rng);
} else return -1;
if (ret == 0) {
    ret = wc_RsaSetRNG(&key, &rng);
} else return -1;
if (ret == 0) {
        ret = wc_MakeRsaKey(&key, 2048, WC_RSA_EXPONENT, &rng);
} else return -1;

if (ret == 0) {
    digestSz = wc_HashGetDigestSize(WC_HASH_TYPE_SHA256);
    ret = wc_Hash(WC_HASH_TYPE_SHA256, message, sz, digest, digestSz);
```

```
} else return -1;

if (ret == 0) {
    ret = wc_RsaPSS_Sign(digest, digestSz, pSignature, pSignatureSz,
            WC_HASH_TYPE_SHA256, WC_MGF1SHA256, &key, &rng);
    if (ret > 0 ){
        sz = ret;
    } else return -1;
} else return -1;
if (ret == 0) {
    ret = wc_RsaPSS_VerifyCheckInline_ex(pSignature, sz, pt,
            digest, digestSz, WC_HASH_TYPE_SHA256, WC_MGF1SHA256, saltLen,
 ↪  &key);
        if (ret <= 0) return -1;
} else return -1;

wc_FreeRsaKey(&key);
wc_FreeRng(&rng);
```

C.44.2.19 function wc_RsaPSS_CheckPadding

```
int wc_RsaPSS_CheckPadding(
    const byte * in,
    word32 inLen,
    byte * sig,
    word32 sigSz,
    enum wc_HashType hashType
)
```

Checks the PSS data to ensure that the signature matches. Salt length is equal to hash length. The key has to be associated with RNG by wc_RsaSetRNG when WC_RSA_BLINDING is enabled.

Parameters:

- **in** Hash of the data that is being verified.
- **inSz** Length of hash.
- **sig** Buffer holding PSS data.
- **sigSz** Size of PSS data.
- **hashType** Hash algorithm.

See:

- wc_RsaPSS_Sign
- wc_RsaPSS_Verify
- wc_RsaPSS_VerifyInline
- wc_RsaPSS_VerifyCheck
- wc_RsaPSS_VerifyCheck_ex
- wc_RsaPSS_VerifyCheckInline
- wc_RsaPSS_VerifyCheckInline_ex
- wc_RsaPSS_CheckPadding_ex
- wc_RsaSetRNG

Return:

- BAD_PADDING_E when the PSS data is invalid, BAD_FUNC_ARG when NULL is passed in to in or sig or inSz is not the same as the hash algorithm length and 0 on success.
- MEMORY_E memory exception.

Example

```
ret = wc_InitRsaKey(&key, NULL);
if (ret == 0) {
    ret = wc_InitRng(&rng);
} else return -1;
if (ret == 0) {
    ret = wc_RsaSetRNG(&key, &rng);
} else return -1;
if (ret == 0) {
        ret = wc_MakeRsaKey(&key, 2048, WC_RSA_EXPONENT, &rng);
} else return -1;
if (ret == 0) {
    digestSz = wc_HashGetDigestSize(WC_HASH_TYPE_SHA256);
    ret = wc_Hash(WC_HASH_TYPE_SHA256, message, sz, digest, digestSz);
} else return -1;
ret = wc_RsaPSS_Sign(digest, digestSz, pSignature, sizeof(pSignature),
        WC_HASH_TYPE_SHA256, WC_MGF1SHA256, &key, &rng);
if (ret > 0 ){
    sz = ret;
} else return -1;

verify = wc_RsaPSS_Verify(pSignature, sz, out, outLen,
        WC_HASH_TYPE_SHA256, WC_MGF1SHA256, &key);
if (verify <= 0)return -1;

ret = wc_RsaPSS_CheckPadding(digest, digestSz, out, verify, hash);

wc_FreeRsaKey(&key);
wc_FreeRng(&rng);
```

C.44.2.20 function wc_RsaPSS_CheckPadding_ex

```
int wc_RsaPSS_CheckPadding_ex(
    const byte * in,
    word32 inLen,
    byte * sig,
    word32 sigSz,
    enum wc_HashType hashType,
    int saltLen,
    int bits
)
```

Checks the PSS data to ensure that the signature matches. Salt length is equal to hash length.

Parameters:

- **in** Hash of the data that is being verified.
- **inSz** Length of hash.
- **sig** Buffer holding PSS data.
- **sigSz** Size of PSS data.
- **hashType** Hash algorithm.
- **saltLen** Length of salt used. RSA_PSS_SALT_LEN_DEFAULT (-1) indicates salt length is the same as the hash length. RSA_PSS_SALT_LEN_DISCOVER indicates salt length is determined from the data.
- **bits** Can be used to calculate salt size in FIPS case

See:

- wc_RsaPSS_Sign
- wc_RsaPSS_Verify
- wc_RsaPSS_VerifyInline
- wc_RsaPSS_VerifyCheck
- wc_RsaPSS_VerifyCheck_ex
- wc_RsaPSS_VerifyCheckInline
- wc_RsaPSS_VerifyCheckInline_ex
- wc_RsaPSS_CheckPadding

Return:

- BAD_PADDING_E when the PSS data is invalid, BAD_FUNC_ARG when NULL is passed in to in or sig or inSz is not the same as the hash algorithm length and 0 on success.
- MEMORY_E memory exception.

Example

```
ret = wc_InitRsaKey(&key, NULL);
if (ret == 0) {
    ret = wc_InitRng(&rng);
} else return -1;
if (ret == 0) {
    ret = wc_RsaSetRNG(&key, &rng);
} else return -1;
if (ret == 0) {
        ret = wc_MakeRsaKey(&key, 2048, WC_RSA_EXPONENT, &rng);
} else return -1;
if (ret == 0) {
    digestSz = wc_HashGetDigestSize(WC_HASH_TYPE_SHA256);
    ret = wc_Hash(WC_HASH_TYPE_SHA256, message, sz, digest, digestSz);
} else return -1;
ret = wc_RsaPSS_Sign(digest, digestSz, pSignature, sizeof(pSignature),
        WC_HASH_TYPE_SHA256, WC_MGF1SHA256, &key, &rng);
if (ret > 0 ){
    sz = ret;
} else return -1;

verify = wc_RsaPSS_Verify(pSignature, sz, out, outLen,
        WC_HASH_TYPE_SHA256, WC_MGF1SHA256, &key);
if (verify <= 0)return -1;

ret = wc_RsaPSS_CheckPadding_ex(digest, digestSz, out, verify, hash, saltLen,
 ↪  0);

wc_FreeRsaKey(&key);
wc_FreeRng(&rng);
```

C.44.2.21 function wc_RsaEncryptSize

```
int wc_RsaEncryptSize(
    RsaKey * key
)
```

Returns the encryption size for the provided key structure.

Parameters:

- **key** The key to use for verification.

See:

- wc_InitRsaKey
- wc_InitRsaKey_ex
- wc_MakeRsaKey

Return: Success Encryption size for the provided key structure.

Example

```
int sz = wc_RsaEncryptSize(&key);
```

C.44.2.22 function wc_RsaPrivateKeyDecode

```
int wc_RsaPrivateKeyDecode(
    const byte * input,
    word32 * inOutIdx,
    RsaKey * key,
    word32 inSz
)
```

This function parses a DER-formatted RSA private key, extracts the private key and stores it in the given RsaKey structure. It also sets the distance parsed in idx.

Parameters:

- **input** pointer to the buffer containing the DER formatted private key to decode
- **inOutIdx** pointer to the index in the buffer at which the key begins (usually 0). As a side effect of this function, inOutIdx will store the distance parsed through the input buffer
- **key** pointer to the RsaKey structure in which to store the decoded private key
- **inSz** size of the input buffer

See:

- wc_RsaPublicKeyDecode
- wc_MakeRsaKey

Return:

- 0 Returned upon successfully parsing the private key from the DER encoded input
- ASN_PARSE_E Returned if there is an error parsing the private key from the input buffer. This may happen if the input private key is not properly formatted according to ASN.1 standards
- ASN_RSA_KEY_E Returned if there is an error reading the private key elements of the RSA key input

Example

```
RsaKey enc;
word32 idx = 0;
int ret = 0;
byte der[] = { // initialize with DER-encoded RSA private key };

wc_InitRsaKey(&enc, NULL); // not using heap hint. No custom memory
ret = wc_RsaPrivateKeyDecode(der, &idx, &enc, sizeof(der));
if( ret != 0 ) {
    // error parsing private key
}
```

C.44.2.23 function wc_RsaPublicKeyDecode

```
int wc_RsaPublicKeyDecode(
    const byte * input,
    word32 * inOutIdx,
    RsaKey * key,
    word32 inSz
)
```

This function parses a DER-formatted RSA public key, extracts the public key and stores it in the given RsaKey structure. It also sets the distance parsed in idx.

Parameters:

- **input** pointer to the buffer containing the input DER-encoded RSA public key to decode
- **inOutIdx** pointer to the index in the buffer at which the key begins (usually 0). As a side effect of this function, inOutIdx will store the distance parsed through the input buffer
- **key** pointer to the RsaKey structure in which to store the decoded public key
- **inSz** size of the input buffer

See: wc_RsaPublicKeyDecodeRaw

Return:

- 0 Returned upon successfully parsing the public key from the DER encoded input
- ASN_PARSE_E Returned if there is an error parsing the public key from the input buffer. This may happen if the input public key is not properly formatted according to ASN.1 standards
- ASN_OBJECT_ID_E Returned if the ASN.1 Object ID does not match that of a RSA public key
- ASN_EXPECT_0_E Returned if the input key is not correctly formatted according to ASN.1 standards
- ASN_BITSTR_E Returned if the input key is not correctly formatted according to ASN.1 standards
- ASN_RSA_KEY_E Returned if there is an error reading the public key elements of the RSA key input

Example

```
RsaKey pub;
word32 idx = 0;
int ret = 0;
byte der[] = { // initialize with DER-encoded RSA public key };

wc_InitRsaKey(&pub, NULL); // not using heap hint. No custom memory
ret = wc_RsaPublicKeyDecode(der, &idx, &pub, sizeof(der));
if( ret != 0 ) {
    // error parsing public key
}
```

C.44.2.24 function wc_RsaPublicKeyDecodeRaw

```
int wc_RsaPublicKeyDecodeRaw(
    const byte * n,
    word32 nSz,
    const byte * e,
    word32 eSz,
    RsaKey * key
)
```

This function decodes the raw elements of an RSA public key, taking in the public modulus (n) and exponent (e). It stores these raw elements in the provided RsaKey structure, allowing one to use them in the encryption/decryption process.

Parameters:

- **n** pointer to a buffer containing the raw modulus parameter of the public RSA key
- **nSz** size of the buffer containing n
- **e** pointer to a buffer containing the raw exponent parameter of the public RSA key
- **eSz** size of the buffer containing e
- **key** pointer to the RsaKey struct to initialize with the provided public key elements

See: wc_RsaPublicKeyDecode

Return:

- 0 Returned upon successfully decoding the raw elements of the public key into the RsaKey structure
- BAD_FUNC_ARG Returned if any of the input arguments evaluates to NULL
- MP_INIT_E Returned if there is an error initializing an integer for use with the multiple precision integer (mp_int) library
- ASN_GETINT_E Returned if there is an error reading one of the provided RSA key elements, n or e

Example

```
RsaKey pub;
int ret = 0;
byte n[] = { // initialize with received n component of public key };
byte e[] = { // initialize with received e component of public key };

wc_InitRsaKey(&pub, NULL); // not using heap hint. No custom memory
ret = wc_RsaPublicKeyDecodeRaw(n, sizeof(n), e, sizeof(e), &pub);
if( ret != 0 ) {
    // error parsing public key elements
}
```

C.44.2.25 function wc_RsaKeyToDer

```
int wc_RsaKeyToDer(
    RsaKey * key,
    byte * output,
    word32 inLen
)
```

This function converts an RsaKey key to DER format. The result is written to output and it returns the number of bytes written.

Parameters:

- **key** Initialized RsaKey structure.
- **output** Pointer to output buffer.
- **inLen** Size of output buffer.

See:

- wc_RsaKeyToPublicDer
- wc_InitRsaKey
- wc_MakeRsaKey
- wc_InitRng

Return:

- 0 Success, number of bytes written.

- BAD_FUNC_ARG Returned if key or output is null, or if key->type is not RSA_PRIVATE, or if inLen isn't large enough for output buffer.
- MEMORY_E Returned if there is an error allocating memory.

Example

```
byte* der;
// Allocate memory for der
int derSz = // Amount of memory allocated for der;
RsaKey key;
WC_RNG rng;
long e = 65537; // standard value to use for exponent
ret = wc_MakeRsaKey(&key, 2048, e, &rng); // generate 2048 bit long
private key
wc_InitRsaKey(&key, NULL);
wc_InitRng(&rng);
if(wc_RsaKeyToDer(&key, der, derSz) != 0)
{
    // Handle the error thrown
}
```

C.44.2.26 function wc_RsaPublicEncrypt_ex

```
int wc_RsaPublicEncrypt_ex(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 outLen,
    RsaKey * key,
    WC_RNG * rng,
    int type,
    enum wc_HashType hash,
    int mgf,
    byte * label,
    word32 labelSz
)
```

This function performs RSA encrypt while allowing the choice of which padding to use.

Parameters:

- **in** pointer to the buffer for encryption
- **inLen** length of the buffer to encrypt
- **out** encrypted msg created
- **outLen** length of buffer available to hold encrypted msg
- **key** initialized RSA key struct
- **rng** initialized WC_RNG struct
- **type** type of padding to use (WC_RSA_OAEP_PAD or WC_RSA_PKCSV15_PAD)
- **hash** type of hash to use (choices can be found in hash.h)
- **mgf** type of mask generation function to use
- **label** an optional label to associate with encrypted message
- **labelSz** size of the optional label used

See:

- wc_RsaPublicEncrypt
- wc_RsaPrivateDecrypt_ex

Return:

- size On successfully encryption the size of the encrypted buffer is returned
- RSA_BUFFER_E RSA buffer error, output too small or input too large

Example

```
WC_RNG rng;
RsaKey key;
byte in[] = “I use Turing Machines to ask questions”
byte out[256];
int ret;
…

ret = wc_RsaPublicEncrypt_ex(in, sizeof(in), out, sizeof(out), &key, &rng,
WC_RSA_OAEP_PAD, WC_HASH_TYPE_SHA, WC_MGF1SHA1, NULL, 0);
if (ret < 0) {
    //handle error
}
```

C.44.2.27 function wc_RsaPrivateDecrypt_ex

```
int wc_RsaPrivateDecrypt_ex(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 outLen,
    RsaKey * key,
    int type,
    enum wc_HashType hash,
    int mgf,
    byte * label,
    word32 labelSz
)
```

This function uses RSA to decrypt a message and gives the option of what padding type.

Parameters:

- **in** pointer to the buffer for decryption
- **inLen** length of the buffer to decrypt
- **out** decrypted msg created
- **outLen** length of buffer available to hold decrypted msg
- **key** initialized RSA key struct
- **type** type of padding to use (WC_RSA_OAEP_PAD or WC_RSA_PKCSV15_PAD)
- **hash** type of hash to use (choices can be found in hash.h)
- **mgf** type of mask generation function to use
- **label** an optional label to associate with encrypted message
- **labelSz** size of the optional label used

See: none

Return:

- size On successful decryption, the size of the decrypted message is returned.
- MEMORY_E Returned if not enough memory on system to malloc a needed array.
- BAD_FUNC_ARG Returned if a bad argument was passed into the function.

Example

```
WC_RNG rng;
RsaKey key;
byte in[] = “I use Turing Machines to ask questions”
byte out[256];
byte plain[256];
int ret;
…
ret = wc_RsaPublicEncrypt_ex(in, sizeof(in), out, sizeof(out), &key,
&rng, WC_RSA_OAEP_PAD, WC_HASH_TYPE_SHA, WC_MGF1SHA1, NULL, 0);
if (ret < 0) {
    //handle error
}
…
ret = wc_RsaPrivateDecrypt_ex(out, ret, plain, sizeof(plain), &key,
WC_RSA_OAEP_PAD, WC_HASH_TYPE_SHA, WC_MGF1SHA1, NULL, 0);

if (ret < 0) {
    //handle error
}
```

C.44.2.28 function wc_RsaPrivateDecryptInline_ex

```
int wc_RsaPrivateDecryptInline_ex(
    byte * in,
    word32 inLen,
    byte ** out,
    RsaKey * key,
    int type,
    enum wc_HashType hash,
    int mgf,
    byte * label,
    word32 labelSz
)
```

This function uses RSA to decrypt a message inline and gives the option of what padding type. The in buffer will contain the decrypted message after being called and the out byte pointer will point to the location in the “in” buffer where the plain text is.

Parameters:

- **in** pointer to the buffer for decryption
- **inLen** length of the buffer to decrypt
- **out** pointer to location of decrypted message in “in” buffer
- **key** initialized RSA key struct
- **type** type of padding to use (WC_RSA_OAEP_PAD or WC_RSA_PKCSV15_PAD)
- **hash** type of hash to use (choices can be found in hash.h)
- **mgf** type of mask generation function to use
- **label** an optional label to associate with encrypted message
- **labelSz** size of the optional label used

See: none

Return:

- size On successful decryption, the size of the decrypted message is returned.
- MEMORY_E: Returned if not enough memory on system to malloc a needed array.
- RSA_PAD_E: Returned if an error in the padding was encountered.

- BAD_PADDING_E: Returned if an error happened during parsing past padding.
- BAD_FUNC_ARG: Returned if a bad argument was passed into the function.

Example

```
WC_RNG rng;
RsaKey key;
byte in[] = “I use Turing Machines to ask questions”
byte out[256];
byte* plain;
int ret;
…
ret = wc_RsaPublicEncrypt_ex(in, sizeof(in), out, sizeof(out), &key,
&rng, WC_RSA_OAEP_PAD, WC_HASH_TYPE_SHA, WC_MGF1SHA1, NULL, 0);

if (ret < 0) {
    //handle error
}
…
ret = wc_RsaPrivateDecryptInline_ex(out, ret, &plain, &key,
WC_RSA_OAEP_PAD, WC_HASH_TYPE_SHA, WC_MGF1SHA1, NULL, 0);

if (ret < 0) {
    //handle error
}
```

C.44.2.29 function wc_RsaFlattenPublicKey

```
int wc_RsaFlattenPublicKey(
    RsaKey * key,
    byte * e,
    word32 * eSz,
    byte * n,
    word32 * nSz
)
```

Flattens the RsaKey structure into individual elements (e, n) used for the RSA algorithm.

Parameters:

- **key** The key to use for verification.
- **e** a buffer for the value of e. e is a large positive integer in the RSA modular arithmetic operation.
- **eSz** the size of the e buffer.
- **n** a buffer for the value of n. n is a large positive integer in the RSA modular arithmetic operation.
- **nSz** the size of the n buffer.

See:

- wc_InitRsaKey
- wc_InitRsaKey_ex
- wc_MakeRsaKey

Return:

- 0 Returned if the function executed normally, without error.
- BAD_FUNC_ARG: Returned if any of the parameters are passed in with a null value.
- RSA_BUFFER_E: Returned if the e or n buffers passed in are not the correct size.
- MP_MEM: Returned if an internal function has memory errors.
- MP_VAL: Returned if an internal function argument is not valid.

Example

```
Rsa key; // A valid RSA key.
byte e[ buffer sz E.g. 256 ];
byte n[256];
int ret;
word32 eSz = sizeof(e);
word32 nSz = sizeof(n);
...
ret = wc_RsaFlattenPublicKey(&key, e, &eSz, n, &nSz);
if (ret != 0) {
    // Failure case.
}
```

C.44.2.30 function wc_RsaKeyToPublicDer

```
int wc_RsaKeyToPublicDer(
    RsaKey * key,
    byte * output,
    word32 inLen
)
```

Convert Rsa Public key to DER format. Writes to output, and returns count of bytes written.

Parameters:

- **key** The RSA key structure to convert.
- **output** Output buffer to hold DER. (if NULL will return length only)
- **inLen** Length of buffer.

See:

- wc_RsaPublicKeyDerSize
- wc_RsaKeyToPublicDer_ex
- wc_InitRsaKey

Return:

- 0 Success, number of bytes written.
- BAD_FUNC_ARG Returned if key or output is null.
- MEMORY_E Returned when an error allocating memory occurs.
- <0 Error

Example

```
RsaKey key;

wc_InitRsaKey(&key, NULL);
// Use key

const int BUFFER_SIZE = 1024; // Some adequate size for the buffer
byte output[BUFFER_SIZE];
if (wc_RsaKeyToPublicDer(&key, output, sizeof(output)) != 0) {
    // Handle Error
}
```

C.44.2.31 function wc_RsaKeyToPublicDer_ex

```
int wc_RsaKeyToPublicDer_ex(
    RsaKey * key,
    byte * output,
    word32 inLen,
    int with_header
)
```

Convert RSA Public key to DER format. Writes to output, and returns count of bytes written. If with_header is 0 then only the (seq + n + e) is returned in ASN.1 DER format and will exclude the header.

Parameters:

- **key** The RSA key structure to convert.
- **output** Output buffer to hold DER. (if NULL will return length only)
- **inLen** Length of buffer.

See:

- wc_RsaPublicKeyDerSize
- wc_RsaKeyToPublicDer
- wc_InitRsaKey

Return:

- 0 Success, number of bytes written.
- BAD_FUNC_ARG Returned if key or output is null.
- MEMORY_E Returned when an error allocating memory occurs.
- <0 Error

Example

```
RsaKey key;

wc_InitRsaKey(&key, NULL);
// Use key

const int BUFFER_SIZE = 1024; // Some adequate size for the buffer
byte output[BUFFER_SIZE];
if (wc_RsaKeyToPublicDer_ex(&key, output, sizeof(output), 0) != 0) {
    // Handle Error
}
```

C.44.2.32 function wc_MakeRsaKey

```
int wc_MakeRsaKey(
    RsaKey * key,
    int size,
    long e,
    WC_RNG * rng
)
```

This function generates a RSA private key of length size (in bits) and given exponent (e). It then stores this key in the provided RsaKey structure, so that it may be used for encryption/decryption. A secure number to use for e is 65537. size is required to be greater than RSA_MIN_SIZE and less than

RSA_MAX_SIZE. For this function to be available, the option WOLFSSL_KEY_GEN must be enabled at compile time. This can be accomplished with –enable-keygen if using ./configure.

Parameters:

- **key** pointer to the RsaKey structure in which to store the generated private key
- **size** desired key length, in bits. Required to be greater than RSA_MIN_SIZE and less than RSA_MAX_SIZE
- **e** exponent parameter to use for generating the key. A secure choice is 65537
- **rng** pointer to an RNG structure to use for random number generation while making the ke

See: none

Return:

- 0 Returned upon successfully generating a RSA private key
- BAD_FUNC_ARG Returned if any of the input arguments are NULL, the size parameter falls outside of the necessary bounds, or e is incorrectly chosen
- RNG_FAILURE_E Returned if there is an error generating a random block using the provided RNG structure
- MP_INIT_E
- MP_READ_E May be May be returned if there is an error in the math library used while generating the RSA key returned if there is an error in the math library used while generating the RSA key
- MP_CMP_E May be returned if there is an error in the math library used while generating the RSA key
- MP_INVMOD_E May be returned if there is an error in the math library used while generating the RSA key
- MP_EXPTMOD_E May be returned if there is an error in the math library used while generating the RSA key
- MP_MOD_E May be returned if there is an error in the math library used while generating the RSA key
- MP_MUL_E May be returned if there is an error in the math library used while generating the RSA key
- MP_ADD_E May be returned if there is an error in the math library used while generating the RSA key
- MP_MULMOD_E May be returned if there is an error in the math library used while generating the RSA key
- MP_TO_E May be returned if there is an error in the math library used while generating the RSA key
- MP_MEM May be returned if there is an error in the math library used while generating the RSA key
- MP_ZERO_E May be returned if there is an error in the math library used while generating the RSA key

Example

```
RsaKey priv;
WC_RNG rng;
int ret = 0;
long e = 65537; // standard value to use for exponent

wc_InitRsaKey(&priv, NULL); // not using heap hint. No custom memory
wc_InitRng(&rng);
// generate 2048 bit long private key
ret = wc_MakeRsaKey(&priv, 2048, e, &rng);
if( ret != 0 ) {
    // error generating private key
}
```

C.44.2.33 function wc_RsaSetNonBlock

```
int wc_RsaSetNonBlock(
    RsaKey * key,
    RsaNb * nb
)
```

This function sets the non-blocking RSA context. When a RsaNb context is set it enables fast math based non-blocking exptmod, which splits the RSA function into many smaller operations. Enabled when WC_RSA_NONBLOCK is defined.

Parameters:

- **key** The RSA key structure
- **nb** The RSA non-blocking structure for this RSA key to use.

See: wc_RsaSetNonBlockTime

Return:

- 0 Success
- BAD_FUNC_ARG Returned if key or nb is null.

Example

```
int ret, count = 0;
RsaKey key;
RsaNb  nb;

wc_InitRsaKey(&key, NULL);

// Enable non-blocking RSA mode - provide context
ret = wc_RsaSetNonBlock(key, &nb);
if (ret != 0)
    return ret;

do {
    ret = wc_RsaSSL_Sign(in, inLen, out, outSz, key, rng);
    count++; // track number of would blocks
    if (ret == FP_WOULDBLOCK) {
        // do "other" work here
    }
} while (ret == FP_WOULDBLOCK);
if (ret < 0) {
    return ret;
}

printf("RSA non-block sign: size %d, %d times\n", ret, count);
```

C.44.2.34 function wc_RsaSetNonBlockTime

```
int wc_RsaSetNonBlockTime(
    RsaKey * key,
    word32 maxBlockUs,
    word32 cpuMHz
)
```

This function configures the maximum amount of blocking time in microseconds. It uses a pre-computed table (see tfm.c exptModNbInst) along with the CPU speed in megahertz to determine if

the next operation can be completed within the maximum blocking time provided. Enabled when WC_RSA_NONBLOCK_TIME is defined.

Parameters:

- **key** The RSA key structure.
- **maxBlockUs** Maximum time to block microseconds.
- **cpuMHz** CPU speed in megahertz.

See: wc_RsaSetNonBlock

Return:

- 0 Success
- BAD_FUNC_ARG Returned if key is null or wc_RsaSetNonBlock was not previously called and key->nb is null.

Example

```
RsaKey key;
RsaNb  nb;

wc_InitRsaKey(&key, NULL);
wc_RsaSetNonBlock(key, &nb);
wc_RsaSetNonBlockTime(&key, 4000, 160); // Block Max = 4 ms, CPU = 160MHz
```

C.44.3 Source code

```
int  wc_InitRsaKey(RsaKey* key, void* heap);

int  wc_InitRsaKey_Id(RsaKey* key, unsigned char* id, int len,
        void* heap, int devId);

int wc_RsaSetRNG(RsaKey* key, WC_RNG* rng);

int  wc_FreeRsaKey(RsaKey* key);

int wc_RsaDirect(byte* in, word32 inLen, byte* out, word32* outSz,
        RsaKey* key, int type, WC_RNG* rng);

int  wc_RsaPublicEncrypt(const byte* in, word32 inLen, byte* out,
                                 word32 outLen, RsaKey* key, WC_RNG* rng);

int  wc_RsaPrivateDecryptInline(byte* in, word32 inLen, byte** out,
                                       RsaKey* key);

int  wc_RsaPrivateDecrypt(const byte* in, word32 inLen, byte* out,
                                  word32 outLen, RsaKey* key);

int  wc_RsaSSL_Sign(const byte* in, word32 inLen, byte* out,
                            word32 outLen, RsaKey* key, WC_RNG* rng);

int  wc_RsaSSL_VerifyInline(byte* in, word32 inLen, byte** out,
                                    RsaKey* key);

int  wc_RsaSSL_Verify(const byte* in, word32 inLen, byte* out,
```

```
                              word32 outLen, RsaKey* key);

int  wc_RsaPSS_Sign(const byte* in, word32 inLen, byte* out,
                              word32 outLen, enum wc_HashType hash, int mgf,
                              RsaKey* key, WC_RNG* rng);

int  wc_RsaPSS_Verify(byte* in, word32 inLen, byte* out,
                               word32 outLen, enum wc_HashType hash, int mgf,
                                RsaKey* key);

int  wc_RsaPSS_VerifyInline(byte* in, word32 inLen, byte** out,
                                      enum wc_HashType hash, int mgf,
                                      RsaKey* key);
int  wc_RsaPSS_VerifyCheck(byte* in, word32 inLen,
                             byte* out, word32 outLen,
                             const byte* digest, word32 digestLen,
                             enum wc_HashType hash, int mgf,
                             RsaKey* key);
int  wc_RsaPSS_VerifyCheck_ex(byte* in, word32 inLen,
                             byte* out, word32 outLen,
                             const byte* digest, word32 digestLen,
                             enum wc_HashType hash, int mgf, int saltLen,
                             RsaKey* key);

int  wc_RsaPSS_VerifyCheckInline(byte* in, word32 inLen, byte** out,
                             const byte* digest, word32 digentLen,
                             enum wc_HashType hash, int mgf,
                             RsaKey* key);
int  wc_RsaPSS_VerifyCheckInline_ex(byte* in, word32 inLen, byte** out,
                             const byte* digest, word32 digentLen,
                             enum wc_HashType hash, int mgf, int saltLen,
                             RsaKey* key);

int  wc_RsaPSS_CheckPadding(const byte* in, word32 inLen, byte* sig,
                                      word32 sigSz,
                                      enum wc_HashType hashType);
int  wc_RsaPSS_CheckPadding_ex(const byte* in, word32 inLen, byte* sig,
               word32 sigSz, enum wc_HashType hashType, int saltLen, int bits);
int  wc_RsaEncryptSize(RsaKey* key);

int  wc_RsaPrivateKeyDecode(const byte* input, word32* inOutIdx,
                            RsaKey* key, word32 inSz);

int  wc_RsaPublicKeyDecode(const byte* input, word32* inOutIdx,
                           RsaKey* key, word32 inSz);

int  wc_RsaPublicKeyDecodeRaw(const byte* n, word32 nSz,
                                       const byte* e, word32 eSz, RsaKey* key);

int wc_RsaKeyToDer(RsaKey* key, byte* output, word32 inLen);

int  wc_RsaPublicEncrypt_ex(const byte* in, word32 inLen, byte* out,
                  word32 outLen, RsaKey* key, WC_RNG* rng, int type,
                 enum wc_HashType hash, int mgf, byte* label, word32 labelSz);
```

```
int  wc_RsaPrivateDecrypt_ex(const byte* in, word32 inLen,
                  byte* out, word32 outLen, RsaKey* key, int type,
                 enum wc_HashType hash, int mgf, byte* label, word32 labelSz);

int  wc_RsaPrivateDecryptInline_ex(byte* in, word32 inLen,
                    byte** out, RsaKey* key, int type, enum wc_HashType hash,
                     int mgf, byte* label, word32 labelSz);

int  wc_RsaFlattenPublicKey(RsaKey* key, byte* e, word32* eSz, byte* n,
                            word32* nSz);

int wc_RsaKeyToPublicDer(RsaKey* key, byte* output, word32 inLen);

int wc_RsaKeyToPublicDer_ex(RsaKey* key, byte* output, word32 inLen,
    int with_header);

int wc_MakeRsaKey(RsaKey* key, int size, long e, WC_RNG* rng);

int wc_RsaSetNonBlock(RsaKey* key, RsaNb* nb);

int wc_RsaSetNonBlockTime(RsaKey* key, word32 maxBlockUs,
    word32 cpuMHz);
```

C.45 dox_comments/header_files/sakke.h

C.45.1 Functions

	Name
int	**wc_InitSakkeKey**(SakkeKey * key, void * heap, int devId)
int	**wc_InitSakkeKey_ex**(SakkeKey * key, int keySize, int curveId, void * heap, int devId)
void	**wc_FreeSakkeKey**(SakkeKey * key)
int	**wc_MakeSakkeKey**(SakkeKey * key, WC_RNG * rng)
int	**wc_MakeSakkePublicKey**(SakkeKey * key, ecc_point * pub)
int	**wc_MakeSakkeRsk**(SakkeKey * key, const byte * id, word16 idSz, ecc_point * rsk)
int	**wc_ValidateSakkeRsk**(SakkeKey * key, const byte * id, word16 idSz, ecc_point * rsk, int * valid)
int	**wc_GenerateSakkeRskTable**(const SakkeKey * key, const ecc_point * rsk, byte * table, word32 * len)
int	**wc_ExportSakkeKey**(SakkeKey * key, byte * data, word32 * sz)
int	**wc_ImportSakkeKey**(SakkeKey * key, const byte * data, word32 sz)
int	**wc_ExportSakkePrivateKey**(SakkeKey * key, byte * data, word32 * sz)

	Name
int	**wc_ImportSakkePrivateKey**(SakkeKey * key, const byte * data, word32 sz)
int	**wc_EncodeSakkeRsk**(const SakkeKey * key, ecc_point * rsk, byte * out, word32 * sz, int raw)
int	**wc_DecodeSakkeRsk**(const SakkeKey * key, const byte * data, word32 sz, ecc_point * rsk)
int	**wc_ImportSakkeRsk**(SakkeKey * key, const byte * data, word32 sz)
int	**wc_ExportSakkePublicKey**(SakkeKey * key, byte * data, word32 * sz, int raw)
int	**wc_ImportSakkePublicKey**(SakkeKey * key, const byte * data, word32 sz, int trusted)
int	**wc_GetSakkeAuthSize**(SakkeKey * key, word16 * authSz)
int	**wc_SetSakkeIdentity**(SakkeKey * key, const byte * id, word16 idSz)
int	**wc_MakeSakkePointI**(SakkeKey * key, const byte * id, word16 idSz)
int	**wc_GetSakkePointI**(SakkeKey * key, byte * data, word32 * sz)
int	**wc_SetSakkePointI**(SakkeKey * key, const byte * id, word16 idSz, const byte * data, word32 sz)
int	**wc_GenerateSakkePointITable**(SakkeKey * key, byte * table, word32 * len)
int	**wc_SetSakkePointITable**(SakkeKey * key, byte * table, word32 len)
int	**wc_ClearSakkePointITable**(SakkeKey * key)
int	**wc_MakeSakkeEncapsulatedSSV**(SakkeKey * key, enum wc_HashType hashType, byte * ssv, word16 ssvSz, byte * auth, word16 * authSz)
int	**wc_GenerateSakkeSSV**(SakkeKey * key, WC_RNG * rng, byte * ssv, word16 * ssvSz)
int	**wc_SetSakkeRsk**(SakkeKey * key, const ecc_point * rsk, byte * table, word32 len)
int	**wc_DeriveSakkeSSV**(SakkeKey * key, enum wc_HashType hashType, byte * ssv, word16 ssvSz, const byte * auth, word16 authSz)

C.45.2 Functions Documentation

C.45.2.1 function wc_InitSakkeKey

```
int wc_InitSakkeKey(
    SakkeKey * key,
    void * heap,
    int devId
)
```

C.45.2.2 function wc_InitSakkeKey_ex

```
int wc_InitSakkeKey_ex(
    SakkeKey * key,
```

```
    int keySize,
    int curveId,
    void * heap,
    int devId
)
```

C.45.2.3 function wc_FreeSakkeKey

```
void wc_FreeSakkeKey(
    SakkeKey * key
)
```

C.45.2.4 function wc_MakeSakkeKey

```
int wc_MakeSakkeKey(
    SakkeKey * key,
    WC_RNG * rng
)
```

C.45.2.5 function wc_MakeSakkePublicKey

```
int wc_MakeSakkePublicKey(
    SakkeKey * key,
    ecc_point * pub
)
```

C.45.2.6 function wc_MakeSakkeRsk

```
int wc_MakeSakkeRsk(
    SakkeKey * key,
    const byte * id,
    word16 idSz,
    ecc_point * rsk
)
```

C.45.2.7 function wc_ValidateSakkeRsk

```
int wc_ValidateSakkeRsk(
    SakkeKey * key,
    const byte * id,
    word16 idSz,
    ecc_point * rsk,
    int * valid
)
```

C.45.2.8 function wc_GenerateSakkeRskTable

```
int wc_GenerateSakkeRskTable(
    const SakkeKey * key,
    const ecc_point * rsk,
    byte * table,
    word32 * len
)
```

C.45.2.9 function wc_ExportSakkeKey

```
int wc_ExportSakkeKey(
    SakkeKey * key,
    byte * data,
    word32 * sz
)
```

C.45.2.10 function wc_ImportSakkeKey

```
int wc_ImportSakkeKey(
    SakkeKey * key,
    const byte * data,
    word32 sz
)
```

C.45.2.11 function wc_ExportSakkePrivateKey

```
int wc_ExportSakkePrivateKey(
    SakkeKey * key,
    byte * data,
    word32 * sz
)
```

C.45.2.12 function wc_ImportSakkePrivateKey

```
int wc_ImportSakkePrivateKey(
    SakkeKey * key,
    const byte * data,
    word32 sz
)
```

C.45.2.13 function wc_EncodeSakkeRsk

```
int wc_EncodeSakkeRsk(
    const SakkeKey * key,
    ecc_point * rsk,
    byte * out,
    word32 * sz,
    int raw
)
```

C.45.2.14 function wc_DecodeSakkeRsk

```
int wc_DecodeSakkeRsk(
    const SakkeKey * key,
    const byte * data,
    word32 sz,
    ecc_point * rsk
)
```

C.45.2.15 function wc_ImportSakkeRsk

```
int wc_ImportSakkeRsk(
    SakkeKey * key,
    const byte * data,
    word32 sz
)
```

C.45.2.16 function wc_ExportSakkePublicKey

```
int wc_ExportSakkePublicKey(
    SakkeKey * key,
    byte * data,
    word32 * sz,
    int raw
)
```

C.45.2.17 function wc_ImportSakkePublicKey

```
int wc_ImportSakkePublicKey(
    SakkeKey * key,
    const byte * data,
    word32 sz,
    int trusted
)
```

C.45.2.18 function wc_GetSakkeAuthSize

```
int wc_GetSakkeAuthSize(
    SakkeKey * key,
    word16 * authSz
)
```

C.45.2.19 function wc_SetSakkeIdentity

```
int wc_SetSakkeIdentity(
    SakkeKey * key,
    const byte * id,
    word16 idSz
)
```

C.45.2.20 function wc_MakeSakkePointI

```
int wc_MakeSakkePointI(
    SakkeKey * key,
    const byte * id,
    word16 idSz
)
```

C.45.2.21 function wc_GetSakkePointI

```
int wc_GetSakkePointI(
    SakkeKey * key,
    byte * data,
    word32 * sz
)
```

C.45.2.22 function wc_SetSakkePointI

```
int wc_SetSakkePointI(
    SakkeKey * key,
    const byte * id,
    word16 idSz,
    const byte * data,
    word32 sz
)
```

C.45.2.23 function wc_GenerateSakkePointITable

```
int wc_GenerateSakkePointITable(
    SakkeKey * key,
    byte * table,
    word32 * len
)
```

C.45.2.24 function wc_SetSakkePointITable

```
int wc_SetSakkePointITable(
    SakkeKey * key,
    byte * table,
    word32 len
)
```

C.45.2.25 function wc_ClearSakkePointITable

```
int wc_ClearSakkePointITable(
    SakkeKey * key
)
```

C.45.2.26 function wc_MakeSakkeEncapsulatedSSV

```
int wc_MakeSakkeEncapsulatedSSV(
    SakkeKey * key,
    enum wc_HashType hashType,
    byte * ssv,
    word16 ssvSz,
    byte * auth,
    word16 * authSz
)
```

C.45.2.27 function wc_GenerateSakkeSSV

```
int wc_GenerateSakkeSSV(
    SakkeKey * key,
    WC_RNG * rng,
    byte * ssv,
    word16 * ssvSz
)
```

C.45.2.28 function wc_SetSakkeRsk

```
int wc_SetSakkeRsk(
    SakkeKey * key,
    const ecc_point * rsk,
    byte * table,
    word32 len
)
```

C.45.2.29 function wc_DeriveSakkeSSV

```
int wc_DeriveSakkeSSV(
    SakkeKey * key,
    enum wc_HashType hashType,
    byte * ssv,
    word16 ssvSz,
    const byte * auth,
    word16 authSz
)
```

C.45.3 Source code

```
int wc_InitSakkeKey(SakkeKey* key, void* heap, int devId);
int wc_InitSakkeKey_ex(SakkeKey* key, int keySize, int curveId,
        void* heap, int devId);
void wc_FreeSakkeKey(SakkeKey* key);

int wc_MakeSakkeKey(SakkeKey* key, WC_RNG* rng);
int wc_MakeSakkePublicKey(SakkeKey* key, ecc_point* pub);

int wc_MakeSakkeRsk(SakkeKey* key, const byte* id, word16 idSz,
        ecc_point* rsk);
int wc_ValidateSakkeRsk(SakkeKey* key, const byte* id, word16 idSz,
        ecc_point* rsk, int* valid);
int wc_GenerateSakkeRskTable(const SakkeKey* key,
        const ecc_point* rsk, byte* table, word32* len);


int wc_ExportSakkeKey(SakkeKey* key, byte* data, word32* sz);
int wc_ImportSakkeKey(SakkeKey* key, const byte* data, word32 sz);
int wc_ExportSakkePrivateKey(SakkeKey* key, byte* data, word32* sz);
int wc_ImportSakkePrivateKey(SakkeKey* key, const byte* data,
        word32 sz);

int wc_EncodeSakkeRsk(const SakkeKey* key, ecc_point* rsk,
        byte* out, word32* sz, int raw);
int wc_DecodeSakkeRsk(const SakkeKey* key, const byte* data,
        word32 sz, ecc_point* rsk);

int wc_ImportSakkeRsk(SakkeKey* key, const byte* data, word32 sz);

int wc_ExportSakkePublicKey(SakkeKey* key, byte* data,
        word32* sz, int raw);
int wc_ImportSakkePublicKey(SakkeKey* key, const byte* data,
```

```
        word32 sz, int trusted);

int wc_GetSakkeAuthSize(SakkeKey* key, word16* authSz);
int wc_SetSakkeIdentity(SakkeKey* key, const byte* id, word16 idSz);
int wc_MakeSakkePointI(SakkeKey* key, const byte* id, word16 idSz);
int wc_GetSakkePointI(SakkeKey* key, byte* data, word32* sz);
int wc_SetSakkePointI(SakkeKey* key, const byte* id, word16 idSz,
        const byte* data, word32 sz);
int wc_GenerateSakkePointITable(SakkeKey* key, byte* table,
         word32* len);
int wc_SetSakkePointITable(SakkeKey* key, byte* table, word32 len);
int wc_ClearSakkePointITable(SakkeKey* key);
int wc_MakeSakkeEncapsulatedSSV(SakkeKey* key,
        enum wc_HashType hashType, byte* ssv, word16 ssvSz, byte* auth,
        word16* authSz);
int wc_GenerateSakkeSSV(SakkeKey* key, WC_RNG* rng, byte* ssv,
        word16* ssvSz);
int wc_SetSakkeRsk(SakkeKey* key, const ecc_point* rsk, byte* table,
        word32 len);
int wc_DeriveSakkeSSV(SakkeKey* key, enum wc_HashType hashType,
        byte* ssv, word16 ssvSz, const byte* auth,
        word16 authSz);
```

C.46 dox_comments/header_files/sha256.h

C.46.1 Functions

	Name
int	**wc_InitSha256**(wc_Sha256 *)This function initializes SHA256. This is automatically called by wc_Sha256Hash.
int	**wc_Sha256Update**(wc_Sha256 * sha, const byte * data, word32 len)Can be called to continually hash the provided byte array of length len.
int	**wc_Sha256Final**(wc_Sha256 * sha256, byte * hash)Finalizes hashing of data. Result is placed into hash. Resets state of sha256 struct.
void	**wc_Sha256Free**(wc_Sha256 *)Resets the Sha256 structure. Note: this is only supported if you have WOLFSSL_TI_HASH defined.
int	**wc_Sha256GetHash**(wc_Sha256 * sha256, byte * hash)Gets hash data. Result is placed into hash. Does not reset state of sha256 struct.
int	**wc_InitSha224**(wc_Sha224 *)Used to initialize a Sha224 struct.
int	**wc_Sha224Update**(wc_Sha224 * sha224, const byte * data, word32 len)Can be called to continually hash the provided byte array of length len.

	Name
int	**wc_Sha224Final**(wc_Sha224 * sha224, byte * hash)Finalizes hashing of data. Result is placed into hash. Resets state of sha224 struct.

C.46.2 Functions Documentation

C.46.2.1 function wc_InitSha256

```
int wc_InitSha256(
    wc_Sha256 *
)
```

This function initializes SHA256. This is automatically called by wc_Sha256Hash.

Parameters:

- **sha256** pointer to the sha256 structure to use for encryption

See:

- wc_Sha256Hash
- wc_Sha256Update
- wc_Sha256Final

Return: 0 Returned upon successfully initializing

Example

```
Sha256 sha256[1];
if ((ret = wc_InitSha256(sha256)) != 0) {
    WOLFSSL_MSG("wc_InitSha256 failed");
}
else {
    wc_Sha256Update(sha256, data, len);
    wc_Sha256Final(sha256, hash);
}
```

C.46.2.2 function wc_Sha256Update

```
int wc_Sha256Update(
    wc_Sha256 * sha,
    const byte * data,
    word32 len
)
```

Can be called to continually hash the provided byte array of length len.

Parameters:

- **sha256** pointer to the sha256 structure to use for encryption
- **data** the data to be hashed
- **len** length of data to be hashed

See:

- wc_Sha256Hash
- wc_Sha256Final
- wc_InitSha256

Return: 0 Returned upon successfully adding the data to the digest.

Example

```
Sha256 sha256[1];
byte data[] = { Data to be hashed };
word32 len = sizeof(data);

if ((ret = wc_InitSha256(sha256)) != 0) {
   WOLFSSL_MSG("wc_InitSha256 failed");
}
else {
    wc_Sha256Update(sha256, data, len);
    wc_Sha256Final(sha256, hash);
}
```

C.46.2.3 function wc_Sha256Final

```
int wc_Sha256Final(
    wc_Sha256 * sha256,
    byte * hash
)
```

Finalizes hashing of data. Result is placed into hash. Resets state of sha256 struct.

Parameters:

- **sha256** pointer to the sha256 structure to use for encryption
- **hash** Byte array to hold hash value.

See:

- wc_Sha256Hash
- wc_Sha256GetHash
- wc_InitSha256

Return: 0 Returned upon successfully finalizing.

Example

```
Sha256 sha256[1];
byte data[] = { Data to be hashed };
word32 len = sizeof(data);

if ((ret = wc_InitSha256(sha256)) != 0) {
   WOLFSSL_MSG("wc_InitSha256 failed");
}
else {
   wc_Sha256Update(sha256, data, len);
   wc_Sha256Final(sha256, hash);
}
```

C.46.2.4 function wc_Sha256Free

```
void wc_Sha256Free(
    wc_Sha256 *
)
```

Resets the Sha256 structure. Note: this is only supported if you have WOLFSSL_TI_HASH defined.

Parameters:

- **sha256** Pointer to the sha256 structure to be freed.

See:

- wc_InitSha256
- wc_Sha256Update
- wc_Sha256Final

Return: none No returns.

Example

```
Sha256 sha256;
byte data[] = { Data to be hashed };
word32 len = sizeof(data);

if ((ret = wc_InitSha256(&sha256)) != 0) {
    WOLFSSL_MSG("wc_InitSha256 failed");
}
else {
    wc_Sha256Update(&sha256, data, len);
    wc_Sha256Final(&sha256, hash);
    wc_Sha256Free(&sha256);
}
```

C.46.2.5 function wc_Sha256GetHash

```
int wc_Sha256GetHash(
    wc_Sha256 * sha256,
    byte * hash
)
```

Gets hash data. Result is placed into hash. Does not reset state of sha256 struct.

Parameters:

- **sha256** pointer to the sha256 structure to use for encryption
- **hash** Byte array to hold hash value.

See:

- wc_Sha256Hash
- wc_Sha256Final
- wc_InitSha256

Return: 0 Returned upon successfully finalizing.

Example

```
Sha256 sha256[1];
if ((ret = wc_InitSha256(sha256)) != 0) {
   WOLFSSL_MSG("wc_InitSha256 failed");
}
else {
   wc_Sha256Update(sha256, data, len);
   wc_Sha256GetHash(sha256, hash);
}
```

C.46.2.6 function wc_InitSha224

```
int wc_InitSha224(
    wc_Sha224 *
)
```

Used to initialize a Sha224 struct.

Parameters:

- **sha224** Pointer to a Sha224 struct to initialize.

See:

- wc_Sha224Hash
- wc_Sha224Update
- wc_Sha224Final

Return:

- 0 Success
- 1 Error returned because sha224 is null.

Example

```
Sha224 sha224;
if(wc_InitSha224(&sha224) != 0)
{
    // Handle error
}
```

C.46.2.7 function wc_Sha224Update

```
int wc_Sha224Update(
    wc_Sha224 * sha224,
    const byte * data,
    word32 len
)
```

Can be called to continually hash the provided byte array of length len.

Parameters:

- **sha224** Pointer to the Sha224 structure to use for encryption.
- **data** Data to be hashed.
- **len** Length of data to be hashed.

See:

- wc_InitSha224
- wc_Sha224Final
- wc_Sha224Hash

Return:

- 0 Success
- 1 Error returned if function fails.
- BAD_FUNC_ARG Error returned if sha224 or data is null.

Example

```
Sha224 sha224;
byte data[] = { /* Data to be hashed };
word32 len = sizeof(data);
```

```
if ((ret = wc_InitSha224(&sha224)) != 0) {
   WOLFSSL_MSG("wc_InitSha224 failed");
}
else {
  wc_Sha224Update(&sha224, data, len);
  wc_Sha224Final(&sha224, hash);
}
```

C.46.2.8 function wc_Sha224Final

```
int wc_Sha224Final(
    wc_Sha224 * sha224,
    byte * hash
)
```

Finalizes hashing of data. Result is placed into hash. Resets state of sha224 struct.

Parameters:

- **sha224** pointer to the sha224 structure to use for encryption
- **hash** Byte array to hold hash value.

See:

- wc_InitSha224
- wc_Sha224Hash
- wc_Sha224Update

Return:

- 0 Success
- <0 Error

Example

```
Sha224 sha224;
byte data[] = { /* Data to be hashed };
word32 len = sizeof(data);

if ((ret = wc_InitSha224(&sha224)) != 0) {
    WOLFSSL_MSG("wc_InitSha224 failed");
}
else {
    wc_Sha224Update(&sha224, data, len);
    wc_Sha224Final(&sha224, hash);
}
```

C.46.3 Source code

```
int wc_InitSha256(wc_Sha256*);

int wc_Sha256Update(wc_Sha256* sha, const byte* data, word32 len);

int wc_Sha256Final(wc_Sha256* sha256, byte* hash);

void wc_Sha256Free(wc_Sha256*);
```

```
int wc_Sha256GetHash(wc_Sha256* sha256, byte* hash);

int wc_InitSha224(wc_Sha224*);

int wc_Sha224Update(wc_Sha224* sha224, const byte* data, word32 len);

int wc_Sha224Final(wc_Sha224* sha224, byte* hash);
```

C.47 dox_comments/header_files/sha512.h

C.47.1 Functions

	Name
int	**wc_InitSha512**(wc_Sha512 *)This function initializes SHA512. This is automatically called by wc_Sha512Hash.
int	**wc_Sha512Update**(wc_Sha512 * sha, const byte * data, word32 len)Can be called to continually hash the provided byte array of length len.
int	**wc_Sha512Final**(wc_Sha512 * sha512, byte * hash)Finalizes hashing of data. Result is placed into hash.
int	**wc_InitSha384**(wc_Sha384 *)This function initializes SHA384. This is automatically called by wc_Sha384Hash.
int	**wc_Sha384Update**(wc_Sha384 * sha, const byte * data, word32 len)Can be called to continually hash the provided byte array of length len.
int	**wc_Sha384Final**(wc_Sha384 * sha384, byte * hash)Finalizes hashing of data. Result is placed into hash.

C.47.2 Functions Documentation

C.47.2.1 function wc_InitSha512

```
int wc_InitSha512(
    wc_Sha512 *
)
```

This function initializes SHA512. This is automatically called by wc_Sha512Hash.

Parameters:

- **sha512** pointer to the sha512 structure to use for encryption

See:

- wc_Sha512Hash
- wc_Sha512Update
- wc_Sha512Final

Return: 0 Returned upon successfully initializing

Example

```
Sha512 sha512[1];
if ((ret = wc_InitSha512(sha512)) != 0) {
   WOLFSSL_MSG("wc_InitSha512 failed");
}
else {
   wc_Sha512Update(sha512, data, len);
   wc_Sha512Final(sha512, hash);
}
```

C.47.2.2 function wc_Sha512Update

```
int wc_Sha512Update(
    wc_Sha512 * sha,
    const byte * data,
    word32 len
)
```

Can be called to continually hash the provided byte array of length len.

Parameters:

- **sha512** pointer to the sha512 structure to use for encryption
- **data** the data to be hashed
- **len** length of data to be hashed

See:

- wc_Sha512Hash
- wc_Sha512Final
- wc_InitSha512

Return: 0 Returned upon successfully adding the data to the digest.

Example

```
Sha512 sha512[1];
byte data[] = { Data to be hashed };
word32 len = sizeof(data);

if ((ret = wc_InitSha512(sha512)) != 0) {
   WOLFSSL_MSG("wc_InitSha512 failed");
}
else {
   wc_Sha512Update(sha512, data, len);
   wc_Sha512Final(sha512, hash);
}
```

C.47.2.3 function wc_Sha512Final

```
int wc_Sha512Final(
    wc_Sha512 * sha512,
    byte * hash
)
```

Finalizes hashing of data. Result is placed into hash.

Parameters:

- **sha512** pointer to the sha512 structure to use for encryption
- **hash** Byte array to hold hash value.

See:

- wc_Sha512Hash
- wc_Sha512Final
- wc_InitSha512

Return: 0 Returned upon successfully finalizing the hash.

Example

```
Sha512 sha512[1];
byte data[] = { Data to be hashed };
word32 len = sizeof(data);

if ((ret = wc_InitSha512(sha512)) != 0) {
    WOLFSSL_MSG("wc_InitSha512 failed");
}
else {
   wc_Sha512Update(sha512, data, len);
   wc_Sha512Final(sha512, hash);
}
```

C.47.2.4 function wc_InitSha384

```
int wc_InitSha384(
    wc_Sha384 *
)
```

This function initializes SHA384. This is automatically called by wc_Sha384Hash.

Parameters:

- **sha384** pointer to the sha384 structure to use for encryption

See:

- wc_Sha384Hash
- wc_Sha384Update
- wc_Sha384Final

Return: 0 Returned upon successfully initializing

Example

```
Sha384 sha384[1];
if ((ret = wc_InitSha384(sha384)) != 0) {
   WOLFSSL_MSG("wc_InitSha384 failed");
}
else {
   wc_Sha384Update(sha384, data, len);
   wc_Sha384Final(sha384, hash);
}
```

C.47.2.5 function wc_Sha384Update

```
int wc_Sha384Update(
    wc_Sha384 * sha,
    const byte * data,
    word32 len
)
```

Can be called to continually hash the provided byte array of length len.

Parameters:

- **sha384** pointer to the sha384 structure to use for encryption
- **data** the data to be hashed
- **len** length of data to be hashed

See:

- wc_Sha384Hash
- wc_Sha384Final
- wc_InitSha384

Return: 0 Returned upon successfully adding the data to the digest.

Example

```
Sha384 sha384[1];
byte data[] = { Data to be hashed };
word32 len = sizeof(data);

if ((ret = wc_InitSha384(sha384)) != 0) {
   WOLFSSL_MSG("wc_InitSha384 failed");
}
else {
   wc_Sha384Update(sha384, data, len);
   wc_Sha384Final(sha384, hash);
}
```

C.47.2.6 function wc_Sha384Final

```
int wc_Sha384Final(
    wc_Sha384 * sha384,
    byte * hash
)
```

Finalizes hashing of data. Result is placed into hash.

Parameters:

- **sha384** pointer to the sha384 structure to use for encryption
- **hash** Byte array to hold hash value.

See:

- wc_Sha384Hash
- wc_Sha384Final
- wc_InitSha384

Return: 0 Returned upon successfully finalizing.

Example

```
Sha384 sha384[1];
byte data[] = { Data to be hashed };
word32 len = sizeof(data);

if ((ret = wc_InitSha384(sha384)) != 0) {
   WOLFSSL_MSG("wc_InitSha384 failed");
}
else {
```

```
   wc_Sha384Update(sha384, data, len);
   wc_Sha384Final(sha384, hash);
}
```

C.47.3 Source code

```
int wc_InitSha512(wc_Sha512*);

int wc_Sha512Update(wc_Sha512* sha, const byte* data, word32 len);

int wc_Sha512Final(wc_Sha512* sha512, byte* hash);

int wc_InitSha384(wc_Sha384*);

int wc_Sha384Update(wc_Sha384* sha, const byte* data, word32 len);

int wc_Sha384Final(wc_Sha384* sha384, byte* hash);
```

C.48 dox_comments/header_files/sha.h

C.48.1 Functions

	Name
int	**wc_InitSha**(wc_Sha *)This function initializes SHA. This is automatically called by wc_ShaHash.
int	**wc_ShaUpdate**(wc_Sha * sha, const byte * data, word32 len)Can be called to continually hash the provided byte array of length len.
int	**wc_ShaFinal**(wc_Sha * sha, byte * hash)Finalizes hashing of data. Result is placed into hash. Resets state of sha struct.
void	**wc_ShaFree**(wc_Sha *)Used to clean up memory used by an initialized Sha struct. Note: this is only supported if you have WOLFSSL_TI_HASH defined.
int	**wc_ShaGetHash**(wc_Sha * sha, byte * hash)Gets hash data. Result is placed into hash. Does not reset state of sha struct.

C.48.2 Functions Documentation

C.48.2.1 function wc_InitSha

```
int wc_InitSha(
    wc_Sha *
)
```

This function initializes SHA. This is automatically called by wc_ShaHash.

Parameters:

- **sha** pointer to the sha structure to use for encryption

See:

- wc_ShaHash
- wc_ShaUpdate
- wc_ShaFinal

Return: 0 Returned upon successfully initializing

Example

```
Sha sha[1];
if ((ret = wc_InitSha(sha)) != 0) {
   WOLFSSL_MSG("wc_InitSha failed");
}
else {
   wc_ShaUpdate(sha, data, len);
   wc_ShaFinal(sha, hash);
}
```

C.48.2.2 function wc_ShaUpdate

```
int wc_ShaUpdate(
    wc_Sha * sha,
    const byte * data,
    word32 len
)
```

Can be called to continually hash the provided byte array of length len.

Parameters:

- **sha** pointer to the sha structure to use for encryption
- **data** the data to be hashed
- **len** length of data to be hashed

See:

- wc_ShaHash
- wc_ShaFinal
- wc_InitSha

Return: 0 Returned upon successfully adding the data to the digest.

Example

```
Sha sha[1];
byte data[] = { // Data to be hashed };
word32 len = sizeof(data);

if ((ret = wc_InitSha(sha)) != 0) {
   WOLFSSL_MSG("wc_InitSha failed");
}
else {
   wc_ShaUpdate(sha, data, len);
   wc_ShaFinal(sha, hash);
}
```

C.48.2.3 function wc_ShaFinal

```
int wc_ShaFinal(
    wc_Sha * sha,
    byte * hash
)
```

Finalizes hashing of data. Result is placed into hash. Resets state of sha struct.

Parameters:

- **sha** pointer to the sha structure to use for encryption
- **hash** Byte array to hold hash value.

See:

- wc_ShaHash
- wc_InitSha
- wc_ShaGetHash

Return: 0 Returned upon successfully finalizing.

Example

```
Sha sha[1];
byte data[] = { Data to be hashed };
word32 len = sizeof(data);

if ((ret = wc_InitSha(sha)) != 0) {
   WOLFSSL_MSG("wc_InitSha failed");
}
else {
   wc_ShaUpdate(sha, data, len);
   wc_ShaFinal(sha, hash);
}
```

C.48.2.4 function wc_ShaFree

```
void wc_ShaFree(
    wc_Sha *
)
```

Used to clean up memory used by an initialized Sha struct. Note: this is only supported if you have WOLFSSL_TI_HASH defined.

Parameters:

- **sha** Pointer to the Sha struct to free.

See:

- wc_InitSha
- wc_ShaUpdate
- wc_ShaFinal

Return: No returns.

Example

```
Sha sha;
wc_InitSha(&sha);
// Use sha
wc_ShaFree(&sha);
```

C.48.2.5 function wc_ShaGetHash

```
int wc_ShaGetHash(
    wc_Sha * sha,
    byte * hash
)
```

Gets hash data. Result is placed into hash. Does not reset state of sha struct.

Parameters:

- **sha** pointer to the sha structure to use for encryption
- **hash** Byte array to hold hash value.

See:

- wc_ShaHash
- wc_ShaFinal
- wc_InitSha

Return: 0 Returned upon successfully finalizing.

Example

```
Sha sha[1];
if ((ret = wc_InitSha(sha)) != 0) {
WOLFSSL_MSG("wc_InitSha failed");
}
else {
    wc_ShaUpdate(sha, data, len);
    wc_ShaGetHash(sha, hash);
}
```

C.48.3 Source code

```
int wc_InitSha(wc_Sha*);

int wc_ShaUpdate(wc_Sha* sha, const byte* data, word32 len);

int wc_ShaFinal(wc_Sha* sha, byte* hash);

void wc_ShaFree(wc_Sha*);

int wc_ShaGetHash(wc_Sha* sha, byte* hash);
```

C.49 dox_comments/header_files/signature.h

C.49.1 Functions

	Name
int	**wc_SignatureGetSize**(enum wc_SignatureType sig_type, const void * key, word32 key_len)This function returns the maximum size of the resulting signature.

	Name
int	**wc_SignatureVerify**(enum wc_HashType hash_type, enum wc_SignatureType sig_type, const byte * data, word32 data_len, const byte * sig, word32 sig_len, const void * key, word32 key_len)This function validates a signature by hashing the data and using the resulting hash and key to verify the signature.
int	**wc_SignatureGenerate**(enum wc_HashType hash_type, enum wc_SignatureType sig_type, const byte * data, word32 data_len, byte * sig, word32 * sig_len, const void * key, word32 key_len, WC_RNG * rng)This function generates a signature from the data using a key. It first creates a hash of the data then signs the hash using the key.

C.49.2 Functions Documentation

C.49.2.1 function wc_SignatureGetSize

```
int wc_SignatureGetSize(
    enum wc_SignatureType sig_type,
    const void * key,
    word32 key_len
)
```

This function returns the maximum size of the resulting signature.

Parameters:

- **sig_type** A signature type enum value such as WC_SIGNATURE_TYPE_ECC or WC_SIGNATURE_TYPE_RSA.
- **key** Pointer to a key structure such as ecc_key or RsaKey.
- **key_len** Size of the key structure.

See:

- wc_HashGetDigestSize
- wc_SignatureGenerate
- wc_SignatureVerify

Return: Returns SIG_TYPE_E if sig_type is not supported. Returns BAD_FUNC_ARG if sig_type was invalid. A positive return value indicates the maximum size of a signature.

Example

```
// Get signature length
enum wc_SignatureType sig_type = WC_SIGNATURE_TYPE_ECC;
ecc_key eccKey;
word32 sigLen;
wc_ecc_init(&eccKey);
sigLen = wc_SignatureGetSize(sig_type, &eccKey, sizeof(eccKey));
if (sigLen > 0) {
    // Success
}
```

C.49.2.2 function wc_SignatureVerify

```
int wc_SignatureVerify(
    enum wc_HashType hash_type,
    enum wc_SignatureType sig_type,
    const byte * data,
    word32 data_len,
    const byte * sig,
    word32 sig_len,
    const void * key,
    word32 key_len
)
```

This function validates a signature by hashing the data and using the resulting hash and key to verify the signature.

Parameters:

- **hash_type** A hash type from the “enum wc_HashType” such as “WC_HASH_TYPE_SHA256”.
- **sig_type** A signature type enum value such as WC_SIGNATURE_TYPE_ECC or WC_SIGNATURE_TYPE_RSA.
- **data** Pointer to buffer containing the data to hash.
- **data_len** Length of the data buffer.
- **sig** Pointer to buffer to output signature.
- **sig_len** Length of the signature output buffer.
- **key** Pointer to a key structure such as ecc_key or RsaKey.
- **key_len** Size of the key structure.

See:

- wc_SignatureGetSize
- wc_SignatureGenerate

Return:

- 0 Success
- SIG_TYPE_E -231, signature type not enabled/ available
- BAD_FUNC_ARG -173, bad function argument provided
- BUFFER_E -132, output buffer too small or input too large.

Example

```
int ret;
ecc_key eccKey;

// Import the public key
wc_ecc_init(&eccKey);
ret = wc_ecc_import_x963(eccPubKeyBuf, eccPubKeyLen, &eccKey);
// Perform signature verification using public key
ret = wc_SignatureVerify(
WC_HASH_TYPE_SHA256, WC_SIGNATURE_TYPE_ECC,
fileBuf, fileLen,
sigBuf, sigLen,
&eccKey, sizeof(eccKey));
printf("Signature Verification: %s
(%d)\n", (ret == 0) ? "Pass" : "Fail", ret);
wc_ecc_free(&eccKey);
```

C.49.2.3 function wc_SignatureGenerate

```
int wc_SignatureGenerate(
    enum wc_HashType hash_type,
    enum wc_SignatureType sig_type,
    const byte * data,
    word32 data_len,
    byte * sig,
    word32 * sig_len,
    const void * key,
    word32 key_len,
    WC_RNG * rng
)
```

This function generates a signature from the data using a key. It first creates a hash of the data then signs the hash using the key.

Parameters:

- **hash_type** A hash type from the "enum wc_HashType" such as "WC_HASH_TYPE_SHA256".
- **sig_type** A signature type enum value such as WC_SIGNATURE_TYPE_ECC or WC_SIGNATURE_TYPE_RSA.
- **data** Pointer to buffer containing the data to hash.
- **data_len** Length of the data buffer.
- **sig** Pointer to buffer to output signature.
- **sig_len** Length of the signature output buffer.
- **key** Pointer to a key structure such as ecc_key or RsaKey.
- **key_len** Size of the key structure.
- **rng** Pointer to an initialized RNG structure.

See:

- wc_SignatureGetSize
- wc_SignatureVerify

Return:

- 0 Success
- SIG_TYPE_E -231, signature type not enabled/ available
- BAD_FUNC_ARG -173, bad function argument provided
- BUFFER_E -132, output buffer too small or input too large.

Example

```
int ret;
WC_RNG rng;
ecc_key eccKey;

wc_InitRng(&rng);
wc_ecc_init(&eccKey);

// Generate key
ret = wc_ecc_make_key(&rng, 32, &eccKey);

// Get signature length and allocate buffer
sigLen = wc_SignatureGetSize(sig_type, &eccKey, sizeof(eccKey));
sigBuf = malloc(sigLen);

// Perform signature verification using public key
ret = wc_SignatureGenerate(
    WC_HASH_TYPE_SHA256, WC_SIGNATURE_TYPE_ECC,
    fileBuf, fileLen,
```

```
    sigBuf, &sigLen,
    &eccKey, sizeof(eccKey),
    &rng);
printf("Signature Generation: %s
(%d)\n", (ret == 0) ? "Pass" : "Fail", ret);

free(sigBuf);
wc_ecc_free(&eccKey);
wc_FreeRng(&rng);
```

C.49.3 Source code

```
int wc_SignatureGetSize(enum wc_SignatureType sig_type,
    const void* key, word32 key_len);

int wc_SignatureVerify(
    enum wc_HashType hash_type, enum wc_SignatureType sig_type,
    const byte* data, word32 data_len,
    const byte* sig, word32 sig_len,
    const void* key, word32 key_len);

int wc_SignatureGenerate(
    enum wc_HashType hash_type, enum wc_SignatureType sig_type,
    const byte* data, word32 data_len,
    byte* sig, word32 *sig_len,
    const void* key, word32 key_len,
    WC_RNG* rng);
```

C.50 dox_comments/header_files/siphash.h

C.50.1 Functions

	Name
int	**wc_InitSipHash**(SipHash * siphash, const unsigned char * key, unsigned char outSz)This function initializes SipHash with a key for a MAC size.
int	**wc_SipHashUpdate**(SipHash * siphash, const unsigned char * in, word32 inSz)Can be called to continually hash the provided byte array of length len.
int	**wc_SipHashFinal**(SipHash * siphash, unsigned char * out, unsigned char outSz)Finalizes MACing of data. Result is placed into out.
int	**wc_SipHash**(const unsigned char * key, const unsigned char * in, word32 inSz, unsigned char * out, unsigned char outSz)This function one-shots the data using SipHash to calculate a MAC based on the key.

C.50.2 Functions Documentation

C.50.2.1 function wc_InitSipHash

```
int wc_InitSipHash(
    SipHash * siphash,
    const unsigned char * key,
    unsigned char outSz
)
```

This function initializes SipHash with a key for a MAC size.

Parameters:

- **siphash** pointer to the SipHash structure to use for MACing
- **key** pointer to the 16-byte array
- **outSz** number of bytes to output as MAC

See:

- wc_SipHash
- wc_SipHashUpdate
- wc_SipHashFinal

Return:

- 0 Returned upon successfully initializing
- BAD_FUNC_ARG Returned when siphash or key is NULL
- BAD_FUNC_ARG Returned when outSz is neither 8 nor 16

Example

```
SipHash siphash[1];
unsigned char key[16] = { ... };
byte macSz = 8; // 8 or 16

if ((ret = wc_InitSipHash(siphash, key, macSz)) != 0) {
    WOLFSSL_MSG("wc_InitSipHash failed");
}
else if ((ret = wc_SipHashUpdate(siphash, data, len)) != 0) {
    WOLFSSL_MSG("wc_SipHashUpdate failed");
}
else if ((ret = wc_SipHashFinal(siphash, mac, macSz)) != 0) {
    WOLFSSL_MSG("wc_SipHashFinal failed");
}
```

C.50.2.2 function wc_SipHashUpdate

```
int wc_SipHashUpdate(
    SipHash * siphash,
    const unsigned char * in,
    word32 inSz
)
```

Can be called to continually hash the provided byte array of length len.

Parameters:

- **siphash** pointer to the SipHash structure to use for MACing
- **in** the data to be MACed
- **inSz** size of data to be MACed

See:

- wc_SipHash
- wc_InitSipHash
- wc_SipHashFinal

Return:

- 0 Returned upon successfully adding the data to the MAC
- BAD_FUNC_ARG Returned when siphash is NULL
- BAD_FUNC_ARG Returned when in is NULL and inSz is not zero

Example

```
SipHash siphash[1];
byte data[] = { Data to be MACed };
word32 len = sizeof(data);

if ((ret = wc_InitSipHash(siphash, key, macSz)) != 0) {
    WOLFSSL_MSG("wc_InitSipHash failed");
}
else if ((ret = wc_SipHashUpdate(siphash, data, len)) != 0) {
    WOLFSSL_MSG("wc_SipHashUpdate failed");
}
else if ((ret = wc_SipHashFinal(siphash, mac, macSz)) != 0) {
    WOLFSSL_MSG("wc_SipHashFinal failed");
}
```

C.50.2.3 function wc_SipHashFinal

```
int wc_SipHashFinal(
    SipHash * siphash,
    unsigned char * out,
    unsigned char outSz
)
```

Finalizes MACing of data. Result is placed into out.

Parameters:

- **siphash** pointer to the SipHash structure to use for MACing
- **out** Byte array to hold MAC value
- **outSz** number of bytes to output as MAC

See:

- wc_SipHash
- wc_InitSipHash
- wc_SipHashUpdate

Return:

- 0 Returned upon successfully finalizing.
- BAD_FUNC_ARG Returned when siphash of out is NULL
- BAD_FUNC_ARG Returned when outSz is not the same as the initialized value

Example

```
SipHash siphash[1];
byte mac[8] = { ... }; // 8 or 16 bytes
byte macSz = sizeof(mac);
```

```
if ((ret = wc_InitSipHash(siphash, key, macSz)) != 0) {
    WOLFSSL_MSG("wc_InitSipHash failed");
}
else if ((ret = wc_SipHashUpdate(siphash, data, len)) != 0) {
    WOLFSSL_MSG("wc_SipHashUpdate failed");
}
else if ((ret = wc_SipHashFinal(siphash, mac, macSz)) != 0) {
    WOLFSSL_MSG("wc_SipHashFinal failed");
}
```

C.50.2.4 function wc_SipHash

```
int wc_SipHash(
    const unsigned char * key,
    const unsigned char * in,
    word32 inSz,
    unsigned char * out,
    unsigned char outSz
)
```

This function one-shots the data using SipHash to calculate a MAC based on the key.

Parameters:

- **key** pointer to the 16-byte array
- **in** the data to be MACed
- **inSz** size of data to be MACed
- **out** Byte array to hold MAC value
- **outSz** number of bytes to output as MAC

See:

- wc_InitSipHash
- wc_SipHashUpdate
- wc_SipHashFinal

Return:

- 0 Returned upon successfully MACing
- BAD_FUNC_ARG Returned when key or out is NULL
- BAD_FUNC_ARG Returned when in is NULL and inSz is not zero
- BAD_FUNC_ARG Returned when outSz is neither 8 nor 16

Example

```
unsigned char key[16] = { ... };
byte data[] = { Data to be MACed };
word32 len = sizeof(data);
byte mac[8] = { ... }; // 8 or 16 bytes
byte macSz = sizeof(mac);

if ((ret = wc_SipHash(key, data, len, mac, macSz)) != 0) {
    WOLFSSL_MSG("wc_SipHash failed");
}
```

C.50.3 Source code

```
int wc_InitSipHash(SipHash* siphash, const unsigned char* key,
    unsigned char outSz);

int wc_SipHashUpdate(SipHash* siphash, const unsigned char* in,
    word32 inSz);

int wc_SipHashFinal(SipHash* siphash, unsigned char* out,
    unsigned char outSz);

int wc_SipHash(const unsigned char* key, const unsigned char* in,
    word32 inSz, unsigned char* out, unsigned char outSz);
```

C.51 dox_comments/header_files/srp.h

C.51.1 Functions

	Name
int	**wc_SrpInit**(Srp * srp, SrpType type, SrpSide side)Initializes the Srp struct for usage.
void	**wc_SrpTerm**(Srp * srp)Releases the Srp struct resources after usage.
int	**wc_SrpSetUsername**(Srp * srp, const byte * username, word32 size)Sets the username. This function MUST be called after wc_SrpInit.
int	**wc_SrpSetParams**(Srp * srp, const byte * N, word32 nSz, const byte * g, word32 gSz, const byte * salt, word32 saltSz)Sets the srp parameters based on the username.. Must be called after wc_SrpSetUsername.
int	**wc_SrpSetPassword**(Srp * srp, const byte * password, word32 size)Sets the password. Setting the password does not persists the clear password data in the srp structure. The client calculates x = H(salt + H(user:pswd)) and stores it in the auth field. This function MUST be called after wc_SrpSetParams and is CLIENT SIDE ONLY.
int	**wc_SrpSetVerifier**(Srp * srp, const byte * verifier, word32 size)Sets the verifier. This function MUST be called after wc_SrpSetParams and is SERVER SIDE ONLY.
int	**wc_SrpGetVerifier**(Srp * srp, byte * verifier, word32 * size)Gets the verifier. The client calculates the verifier with v = g ^ x % N. This function MAY be called after wc_SrpSetPassword and is CLIENT SIDE ONLY.

	Name
int	**wc_SrpSetPrivate**(Srp * srp, const byte * priv, word32 size)Sets the private ephemeral value. The private ephemeral value is known as: a at the client side. a = random() b at the server side. b = random() This function is handy for unit test cases or if the developer wants to use an external random source to set the ephemeral value. This function MAY be called before wc_SrpGetPublic.
int	**wc_SrpGetPublic**(Srp * srp, byte * pub, word32 * size)Gets the public ephemeral value. The public ephemeral value is known as: A at the client side. A = g ^ a % N B at the server side. B = (k * v + (g ^ b % N)) % N This function MUST be called after wc_SrpSetPassword or wc_SrpSetVerifier. The function wc_SrpSetPrivate may be called before wc_SrpGetPublic.
int	**wc_SrpComputeKey**(Srp * srp, byte * clientPubKey, word32 clientPubKeySz, byte * serverPubKey, word32 serverPubKeySz)Computes the session key. The key can be accessed at srp->key after success.
int	**wc_SrpGetProof**(Srp * srp, byte * proof, word32 * size)Gets the proof. This function MUST be called after wc_SrpComputeKey.
int	**wc_SrpVerifyPeersProof**(Srp * srp, byte * proof, word32 size)Verifies the peers proof. This function MUST be called before wc_SrpGetSessionKey.

C.51.2 Functions Documentation

C.51.2.1 function wc_SrpInit

```
int wc_SrpInit(
    Srp * srp,
    SrpType type,
    SrpSide side
)
```

Initializes the Srp struct for usage.

Parameters:

- **srp** the Srp structure to be initialized.
- **type** the hash type to be used.
- **side** the side of the communication.

See:

- wc_SrpTerm
- wc_SrpSetUsername

Return:

- 0 on success.
- BAD_FUNC_ARG Returns when there's an issue with the arguments such as srp being null or SrpSide not being SRP_CLIENT_SIDE or SRP_SERVER_SIDE.
- NOT_COMPILED_IN Returns when a type is passed as an argument but hasn't been configured in the wolfCrypt build.
- <0 on error.

Example

```
Srp srp;
if (wc_SrpInit(&srp, SRP_TYPE_SHA, SRP_CLIENT_SIDE) != 0)
{
    // Initialization error
}
else
{
    wc_SrpTerm(&srp);
}
```

C.51.2.2 function wc_SrpTerm

```
void wc_SrpTerm(
    Srp * srp
)
```

Releases the Srp struct resources after usage.

Parameters:

- **srp** Pointer to the Srp structure to be terminated.

See: wc_SrpInit

Return: none No returns.

Example

```
Srp srp;
wc_SrpInit(&srp, SRP_TYPE_SHA, SRP_CLIENT_SIDE);
// Use srp
wc_SrpTerm(&srp)
```

C.51.2.3 function wc_SrpSetUsername

```
int wc_SrpSetUsername(
    Srp * srp,
    const byte * username,
    word32 size
)
```

Sets the username. This function MUST be called after wc_SrpInit.

Parameters:

- **srp** the Srp structure.
- **username** the buffer containing the username.
- **size** the username size in bytes

See:

- wc_SrpInit

- wc_SrpSetParams
- wc_SrpTerm

Return:

- 0 Username set successfully.
- BAD_FUNC_ARG: Return if srp or username is null.
- MEMORY_E: Returns if there is an issue allocating memory for srp->user
- < 0: Error.

Example

```
Srp srp;
byte username[] = "user";
word32 usernameSize = 4;

wc_SrpInit(&srp, SRP_TYPE_SHA, SRP_CLIENT_SIDE);
if(wc_SrpSetUsername(&srp, username, usernameSize) != 0)
{
    // Error occurred setting username.
}
wc_SrpTerm(&srp);
```

C.51.2.4 function wc_SrpSetParams

```
int wc_SrpSetParams(
    Srp * srp,
    const byte * N,
    word32 nSz,
    const byte * g,
    word32 gSz,
    const byte * salt,
    word32 saltSz
)
```

Sets the srp parameters based on the username.. Must be called after wc_SrpSetUsername.

Parameters:

- **srp** the Srp structure.
- **N** the Modulus. N = 2q+1, [q, N] are primes.
- **nSz** the N size in bytes.
- **g** the Generator modulo N.
- **gSz** the g size in bytes
- **salt** a small random salt. Specific for each username.
- **saltSz** the salt size in bytes

See:

- wc_SrpInit
- wc_SrpSetUsername
- wc_SrpTerm

Return:

- 0 Success
- BAD_FUNC_ARG Returns if srp, N, g, or salt is null or if nSz < gSz.
- SRP_CALL_ORDER_E Returns if wc_SrpSetParams is called before wc_SrpSetUsername.
- <0 Error

Example

```
Srp srp;
byte username[] = "user";
word32 usernameSize = 4;

byte N[] = { }; // Contents of byte array N
byte g[] = { }; // Contents of byte array g
byte salt[] = { }; // Contents of byte array salt

wc_SrpInit(&srp, SRP_TYPE_SHA, SRP_CLIENT_SIDE);
wc_SrpSetUsername(&srp, username, usernameSize);

if(wc_SrpSetParams(&srp, N, sizeof(N), g, sizeof(g), salt,
sizeof(salt)) != 0)
{
    // Error setting params
}
wc_SrpTerm(&srp);
```

C.51.2.5 function wc_SrpSetPassword

```
int wc_SrpSetPassword(
    Srp * srp,
    const byte * password,
    word32 size
)
```

Sets the password. Setting the password does not persists the clear password data in the srp structure. The client calculates x = H(salt + H(user:pswd)) and stores it in the auth field. This function MUST be called after wc_SrpSetParams and is CLIENT SIDE ONLY.

Parameters:

- **srp** The Srp structure.
- **password** The buffer containing the password.
- **size** The size of the password in bytes.

See:

- wc_SrpInit
- wc_SrpSetUsername
- wc_SrpSetParams

Return:

- 0 Success
- BAD_FUNC_ARG Returns if srp or password is null or if srp->side is not set to SRP_CLIENT_SIDE.
- SRP_CALL_ORDER_E Returns when wc_SrpSetPassword is called out of order.
- <0 Error

Example

```
Srp srp;
byte username[] = "user";
word32 usernameSize = 4;
byte password[] = "password";
word32 passwordSize = 8;
```

```
byte N[] = { }; // Contents of byte array N
byte g[] = { }; // Contents of byte array g
byte salt[] = { }; // Contents of byte array salt

wc_SrpInit(&srp, SRP_TYPE_SHA, SRP_CLIENT_SIDE);
wc_SrpSetUsername(&srp, username, usernameSize);
wc_SrpSetParams(&srp, N, sizeof(N), g, sizeof(g), salt, sizeof(salt));

if(wc_SrpSetPassword(&srp, password, passwordSize) != 0)
{
    // Error setting password
}

wc_SrpTerm(&srp);
```

C.51.2.6 function wc_SrpSetVerifier

```
int wc_SrpSetVerifier(
    Srp * srp,
    const byte * verifier,
    word32 size
)
```

Sets the verifier. This function MUST be called after wc_SrpSetParams and is SERVER SIDE ONLY.

Parameters:

- **srp** The Srp structure.
- **verifier** The structure containing the verifier.
- **size** The verifier size in bytes.

See:

- wc_SrpInit
- wc_SrpSetParams
- wc_SrpGetVerifier

Return:

- 0 Success
- BAD_FUNC_ARG Returned if srp or verifier is null or srp->side is not SRP_SERVER_SIDE.
- <0 Error

Example

```
Srp srp;
byte username[] = "user";
word32 usernameSize = 4;

byte N[] = { }; // Contents of byte array N
byte g[] = { }; // Contents of byte array g
byte salt[] = { }; // Contents of byte array salt
wc_SrpInit(&srp, SRP_TYPE_SHA, SRP_SERVER_SIDE);
wc_SrpSetUsername(&srp, username, usernameSize);
wc_SrpSetParams(&srp, N, sizeof(N), g, sizeof(g), salt, sizeof(salt))
byte verifier[] = { }; // Contents of some verifier

if(wc_SrpSetVerifier(&srp, verifier, sizeof(verifier)) != 0)
{
```

```
    // Error setting verifier
}

wc_SrpTerm(&srp);
```

C.51.2.7 function wc_SrpGetVerifier

```
int wc_SrpGetVerifier(
    Srp * srp,
    byte * verifier,
    word32 * size
)
```

Gets the verifier. The client calculates the verifier with v = g ^ x % N. This function MAY be called after wc_SrpSetPassword and is CLIENT SIDE ONLY.

Parameters:

- **srp** The Srp structure.
- **verifier** The buffer to write the verifier.
- **size** Buffer size in bytes. Updated with the verifier size.

See:

- wc_SrpSetVerifier
- wc_SrpSetPassword

Return:

- 0 Success
- BAD_FUNC_ARG Returned if srp, verifier or size is null or if srp->side is not SRP_CLIENT_SIDE.
- SRP_CALL_ORDER_E Returned if wc_SrpGetVerifier is called out of order.
- <0 Error

Example

```
Srp srp;
byte username[] = "user";
word32 usernameSize = 4;
byte password[] = "password";
word32 passwordSize = 8;

byte N[] = { }; // Contents of byte array N
byte g[] = { }; // Contents of byte array g
byte salt[] = { }; // Contents of byte array salt
byte v[64];
word32 vSz = 0;
vSz = sizeof(v);

wc_SrpInit(&srp, SRP_TYPE_SHA, SRP_CLIENT_SIDE);
wc_SrpSetUsername(&srp, username, usernameSize);
wc_SrpSetParams(&srp, N, sizeof(N), g, sizeof(g), salt, sizeof(salt))
wc_SrpSetPassword(&srp, password, passwordSize)

if( wc_SrpGetVerifier(&srp, v, &vSz ) != 0)
{
    // Error getting verifier
}
wc_SrpTerm(&srp);
```

C.51.2.8 function wc_SrpSetPrivate

```
int wc_SrpSetPrivate(
    Srp * srp,
    const byte * priv,
    word32 size
)
```

Sets the private ephemeral value. The private ephemeral value is known as: a at the client side. a = random() b at the server side. b = random() This function is handy for unit test cases or if the developer wants to use an external random source to set the ephemeral value. This function MAY be called before wc_SrpGetPublic.

Parameters:

- **srp** the Srp structure.
- **priv** the ephemeral value.
- **size** the private size in bytes.

See: wc_SrpGetPublic

Return:

- 0 Success
- BAD_FUNC_ARG Returned if srp, private, or size is null.
- SRP_CALL_ORDER_E Returned if wc_SrpSetPrivate is called out of order.
- <0 Error

Example

```
Srp srp;
byte username[] = "user";
word32 usernameSize = 4;

byte N[] = { }; // Contents of byte array N
byte g[] = { }; // Contents of byte array g
byte salt[] = { }; // Contents of byte array salt
byte verifier = { }; // Contents of some verifier
wc_SrpInit(&srp, SRP_TYPE_SHA, SRP_SERVER_SIDE);
wc_SrpSetUsername(&srp, username, usernameSize);
wc_SrpSetParams(&srp, N, sizeof(N), g, sizeof(g), salt, sizeof(salt))
wc_SrpSetVerifier(&srp, verifier, sizeof(verifier))

byte b[] = { }; // Some ephemeral value
if( wc_SrpSetPrivate(&srp, b, sizeof(b)) != 0)
{
    // Error setting private ephemeral
}

wc_SrpTerm(&srp);
```

C.51.2.9 function wc_SrpGetPublic

```
int wc_SrpGetPublic(
    Srp * srp,
    byte * pub,
    word32 * size
)
```

Gets the public ephemeral value. The public ephemeral value is known as: A at the client side. A = g ^ a % N B at the server side. B = (k * v + (g ^ b % N)) % N This function MUST be called after wc_SrpSetPassword or wc_SrpSetVerifier. The function wc_SrpSetPrivate may be called before wc_SrpGetPublic.

Parameters:

- **srp** the Srp structure.
- **pub** the buffer to write the public ephemeral value.
- **size** the the buffer size in bytes. Will be updated with the ephemeral value size.

See:

- wc_SrpSetPrivate
- wc_SrpSetPassword
- wc_SrpSetVerifier

Return:

- 0 Success
- BAD_FUNC_ARG Returned if srp, pub, or size is null.
- SRP_CALL_ORDER_E Returned if wc_SrpGetPublic is called out of order.
- BUFFER_E Returned if size < srp.N.
- <0 Error

Example

```
Srp srp;
byte username[] = "user";
word32 usernameSize = 4;
byte password[] = "password";
word32 passwordSize = 8;

byte N[] = { }; // Contents of byte array N
byte g[] = { }; // Contents of byte array g
byte salt[] = { }; // Contents of byte array salt
wc_SrpInit(&srp, SRP_TYPE_SHA, SRP_CLIENT_SIDE);
wc_SrpSetUsername(&srp, username, usernameSize);
wc_SrpSetParams(&srp, N, sizeof(N), g, sizeof(g), salt, sizeof(salt));
wc_SrpSetPassword(&srp, password, passwordSize)

byte public[64];
word32 publicSz = 0;

if( wc_SrpGetPublic(&srp, public, &publicSz) != 0)
{
    // Error getting public ephemeral
}

wc_SrpTerm(&srp);
```

C.51.2.10 function wc_SrpComputeKey

```
int wc_SrpComputeKey(
    Srp * srp,
    byte * clientPubKey,
    word32 clientPubKeySz,
    byte * serverPubKey,
```

```
    word32 serverPubKeySz
)
```

Computes the session key. The key can be accessed at srp->key after success.

Parameters:

- **srp** the Srp structure.
- **clientPubKey** the client's public ephemeral value.
- **clientPubKeySz** the client's public ephemeral value size.
- **serverPubKey** the server's public ephemeral value.
- **serverPubKeySz** the server's public ephemeral value size.

See: wc_SrpGetPublic

Return:

- 0 Success
- BAD_FUNC_ARG Returned if srp, clientPubKey, or serverPubKey or if clientPubKeySz or serverPubKeySz is 0.
- SRP_CALL_ORDER_E Returned if wc_SrpComputeKey is called out of order.
- <0 Error

Example

```
Srp server;

byte username[] = "user";
    word32 usernameSize = 4;
byte password[] = "password";
word32 passwordSize = 8;
byte N[] = { }; // Contents of byte array N
byte g[] = { }; // Contents of byte array g
byte salt[] = { }; // Contents of byte array salt
byte verifier[] = { }; // Contents of some verifier
byte serverPubKey[] = { }; // Contents of server pub key
word32 serverPubKeySize = sizeof(serverPubKey);
byte clientPubKey[64];
word32 clientPubKeySize = 64;

wc_SrpInit(&server, SRP_TYPE_SHA, SRP_SERVER_SIDE);
wc_SrpSetUsername(&server, username, usernameSize);
wc_SrpSetParams(&server, N, sizeof(N), g, sizeof(g), salt, sizeof(salt));
wc_SrpSetVerifier(&server, verifier, sizeof(verifier));
wc_SrpGetPublic(&server, serverPubKey, &serverPubKeySize);

wc_SrpComputeKey(&server, clientPubKey, clientPubKeySz,
                                        serverPubKey, serverPubKeySize)
wc_SrpTerm(&server);
```

C.51.2.11 function wc_SrpGetProof

```
int wc_SrpGetProof(
    Srp * srp,
    byte * proof,
    word32 * size
)
```

Gets the proof. This function MUST be called after wc_SrpComputeKey.

Parameters:

- **srp** the Srp structure.
- **proof** the peers proof.
- **size** the proof size in bytes.

See: wc_SrpComputeKey

Return:

- 0 Success
- BAD_FUNC_ARG Returns if srp, proof, or size is null.
- BUFFER_E Returns if size is less than the hash size of srp->type.
- <0 Error

Example

```
Srp cli;
byte clientProof[SRP_MAX_DIGEST_SIZE];
word32 clientProofSz = SRP_MAX_DIGEST_SIZE;

// Initialize Srp following steps from previous examples

if (wc_SrpGetProof(&cli, clientProof, &clientProofSz) != 0)
{
    // Error getting proof
}
```

C.51.2.12 function wc_SrpVerifyPeersProof

```
int wc_SrpVerifyPeersProof(
    Srp * srp,
    byte * proof,
    word32 size
)
```

Verifies the peers proof. This function MUST be called before wc_SrpGetSessionKey.

Parameters:

- **srp** the Srp structure.
- **proof** the peers proof.
- **size** the proof size in bytes.

See:

- wc_SrpGetSessionKey
- wc_SrpGetProof
- wc_SrpTerm

Return:

- 0 Success
- <0 Error

Example

```
Srp cli;
Srp srv;
byte clientProof[SRP_MAX_DIGEST_SIZE];
word32 clientProofSz = SRP_MAX_DIGEST_SIZE;
```

```
// Initialize Srp following steps from previous examples
// First get the proof
wc_SrpGetProof(&cli, clientProof, &clientProofSz)

if (wc_SrpVerifyPeersProof(&srv, clientProof, clientProofSz) != 0)
{
    // Error verifying proof
}
```

C.51.3 Source code

```
int wc_SrpInit(Srp* srp, SrpType type, SrpSide side);

void wc_SrpTerm(Srp* srp);

int wc_SrpSetUsername(Srp* srp, const byte* username, word32 size);

int wc_SrpSetParams(Srp* srp, const byte* N,    word32 nSz,
                                      const byte* g,    word32 gSz,
                                      const byte* salt, word32 saltSz);

int wc_SrpSetPassword(Srp* srp, const byte* password, word32 size);

int wc_SrpSetVerifier(Srp* srp, const byte* verifier, word32 size);

int wc_SrpGetVerifier(Srp* srp, byte* verifier, word32* size);

int wc_SrpSetPrivate(Srp* srp, const byte* priv, word32 size);

int wc_SrpGetPublic(Srp* srp, byte* pub, word32* size);

int wc_SrpComputeKey(Srp* srp,
                                 byte* clientPubKey, word32 clientPubKeySz,
                                 byte* serverPubKey, word32 serverPubKeySz);

int wc_SrpGetProof(Srp* srp, byte* proof, word32* size);

int wc_SrpVerifyPeersProof(Srp* srp, byte* proof, word32 size);
```

C.52 dox_comments/header_files/ssl.h

C.52.1 Functions

	Name
WOLFSSL_METHOD *	**wolfDTLSv1_2_client_method_ex**(void * heap)This function initializes the DTLS v1.2 client method.
WOLFSSL_METHOD *	**wolfSSLv23_method**(void)This function returns a WOLFSSL_METHOD similar to wolfSSLv23_client_method except that it is not determined which side yet (server/client).

	Name
WOLFSSL_METHOD *	**wolfSSLv3_server_method.
WOLFSSL_METHOD *	**wolfSSLv3_client_method.
WOLFSSL_METHOD *	**wolfTLSv1_server_method.
WOLFSSL_METHOD *	**wolfTLSv1_client_method.
WOLFSSL_METHOD *	**wolfTLSv1_1_server_method.
WOLFSSL_METHOD *	**wolfTLSv1_1_client_method.
WOLFSSL_METHOD *	**wolfTLSv1_2_server_method.
WOLFSSL_METHOD *	**wolfTLSv1_2_client_method.
WOLFSSL_METHOD *	**wolfDTLSv1_client_method. This function is only available when wolfSSL has been compiled with DTLS support (–enable_dtls, or by defining wolfSSL_DTLS).
WOLFSSL_METHOD *	**wolfDTLSv1_server_method. This function is only available when wolfSSL has been compiled with DTLS support (–enable_dtls, or by defining wolfSSL_DTLS).
WOLFSSL_METHOD *	**wolfDTLSv1_3_server_method. This function is only available when wolfSSL has been compiled with DTLSv1.3 support (–enable_dtls13, or by defining wolfSSL_DTLS13).
WOLFSSL_METHOD *	**wolfDTLSv1_3_client_method. This function is only available when wolfSSL has been compiled with DTLSv1.3 support (–enable_dtls13, or by defining wolfSSL_DTLS13).
WOLFSSL_METHOD *	**wolfDTLS_server_method. This function is only available when wolfSSL has been compiled with DTLS support (–enable_dtls, or by defining wolfSSL_DTLS).
WOLFSSL_METHOD *	**wolfDTLS_client_method. This function is only available when wolfSSL has been compiled with DTLS support (–enable_dtls, or by defining wolfSSL_DTLS).
WOLFSSL_METHOD *	**wolfDTLSv1_2_server_method**(void)This function creates and initializes a WOLFSSL_METHOD for the server side.
int	**wolfSSL_use_old_poly**(WOLFSSL * ssl, int value)Since there is some differences between the first release and newer versions of chacha-poly AEAD construction we have added an option to communicate with servers/clients using the older version. By default wolfSSL uses the new version.
int	**wolfSSL_dtls_import**(WOLFSSL * ssl, unsigned char * buf, unsigned int sz)The wolfSSL_dtls_import() function is used to parse in a serialized session state. This allows for picking up the connection after the handshake has been completed.

	Name
int	**wolfSSL_tls_import**(WOLFSSL * ssl, const unsigned char * buf, unsigned int sz)Used to import a serialized TLS session. This function is for importing the state of the connection. WARNING: buf contains sensitive information about the state and is best to be encrypted before storing if stored. Additional debug info can be displayed with the macro WOLFSSL_SESSION_EXPORT_DEBUG defined.
int	**wolfSSL_CTX_dtls_set_export**(WOLFSSL_CTX * ctx, wc_dtls_export func)The wolfSSL_CTX_dtls_set_export() function is used to set the callback function for exporting a session. It is allowed to pass in NULL as the parameter func to clear the export function previously stored. Used on the server side and is called immediately after handshake is completed.
int	**wolfSSL_dtls_set_export**(WOLFSSL * ssl, wc_dtls_export func)The wolfSSL_dtls_set_export() function is used to set the callback function for exporting a session. It is allowed to pass in NULL as the parameter func to clear the export function previously stored. Used on the server side and is called immediately after handshake is completed.
int	**wolfSSL_dtls_export**(WOLFSSL * ssl, unsigned char * buf, unsigned int * sz)The wolfSSL_dtls_export() function is used to serialize a WOLFSSL session into the provided buffer. Allows for less memory overhead than using a function callback for sending a session and choice over when the session is serialized. If buffer is NULL when passed to function then sz will be set to the size of buffer needed for serializing the WOLFSSL session.
int	**wolfSSL_tls_export**(WOLFSSL * ssl, unsigned char * buf, unsigned int * sz)Used to export a serialized TLS session. This function is for exporting a serialized state of the connection. In most cases wolfSSL_get1_session should be used instead of wolfSSL_tls_export. Additional debug info can be displayed with the macro WOLFSSL_SESSION_EXPORT_DEBUG defined. WARNING: buf contains sensitive information about the state and is best to be encrypted before storing if stored.

COPYRIGHT ©2024 wolfSSL Inc.

	Name
int	**wolfSSL_CTX_load_static_memory**(WOLFSSL_CTX ** ctx, wolfSSL_method_func method, unsigned char * buf, unsigned int sz, int flag, int max)This function is used to set aside static memory for a CTX. Memory set aside is then used for the CTX's lifetime and for any SSL objects created from the CTX. By passing in a NULL ctx pointer and a wolfSSL_method_func function the creation of the CTX itself will also use static memory. wolfSSL_method_func has the function signature of WOLFSSL_METHOD* (*wolfSSL_method_func)(void* heap);. Passing in 0 for max makes it behave as if not set and no max concurrent use restrictions is in place. The flag value passed in determines how the memory is used and behavior while operating. Available flags are the following: 0 - default general memory, WOLFMEM_IO_POOL - used for input/output buffer when sending receiving messages and overrides general memory, so all memory in buffer passed in is used for IO, WOLFMEM_IO_FIXED - same as WOLFMEM_IO_POOL but each SSL now keeps two buffers to themselves for their lifetime, WOLFMEM_TRACK_STATS - each SSL keeps track of memory stats while running.
int	**wolfSSL_CTX_is_static_memory**(WOLFSSL_CTX * ctx, WOLFSSL_MEM_STATS * mem_stats)This function does not change any of the connections behavior and is used only for gathering information about the static memory usage.
int	**wolfSSL_is_static_memory**(WOLFSSL * ssl, WOLFSSL_MEM_CONN_STATS * mem_stats)wolfSSL_is_static_memory is used to gather information about a SSL's static memory usage. The return value indicates if static memory is being used and WOLFSSL_MEM_CONN_STATS will be filled out if and only if the flag WOLFMEM_TRACK_STATS was passed to the parent CTX when loading in static memory.
int	**wolfSSL_CTX_use_certificate_file**(WOLFSSL_CTX * ctx, const char * file, int format)This function loads a certificate file into the SSL context (WOLFSSL_CTX). The file is provided by the file argument. The format argument specifies the format type of the file, either SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM. Please see the examples for proper usage.

	Name
int	**wolfSSL_CTX_use_PrivateKey_file**(WOLFSSL_CTX * ctx, const char * file, int format)This function loads a private key file into the SSL context (WOLFSSL_CTX). The file is provided by the file argument. The format argument specifies the format type of the file - SSL_FILETYPE_ASN1or SSL_FILETYPE_PEM. Please see the examples for proper usage.
int	**wolfSSL_CTX_load_verify_locations**(WOLFSSL_CTX * ctx, const char * file, const char * path)This function loads PEM_formatted CA certificate files into the SSL context (WOLFSSL_CTX). These certificates will be treated as trusted root certificates and used to verify certs received from peers during the SSL handshake. The root certificate file, provided by the file argument, may be a single certificate or a file containing multiple certificates. If multiple CA certs are included in the same file, wolfSSL will load them in the same order they are presented in the file. The path argument is a pointer to the name of a directory that contains certificates of trusted root CAs. If the value of file is not NULL, path may be specified as NULL if not needed. If path is specified and NO_WOLFSSL_DIR was not defined when building the library, wolfSSL will load all CA certificates located in the given directory. This function will attempt to load all files in the directory. This function expects PEM formatted CERT_TYPE file with header “—–BEGIN CERTIFICATE—–”.

	Name
int	**wolfSSL_CTX_load_verify_locations_ex**(WOLFSSL_CTX * ctx, const char * file, const char * path, unsigned int flags)This function loads PEM_formatted CA certificate files into the SSL context (WOLFSSL_CTX). These certificates will be treated as trusted root certificates and used to verify certs received from peers during the SSL handshake. The root certificate file, provided by the file argument, may be a single certificate or a file containing multiple certificates. If multiple CA certs are included in the same file, wolfSSL will load them in the same order they are presented in the file. The path argument is a pointer to the name of a directory that contains certificates of trusted root CAs. If the value of file is not NULL, path may be specified as NULL if not needed. If path is specified and NO_WOLFSSL_DIR was not defined when building the library, wolfSSL will load all CA certificates located in the given directory. This function will attempt to load all files in the directory based on flags specified. This function expects PEM formatted CERT_TYPE files with header "—–BEGIN CERTIFICATE—–".
const char **	**wolfSSL_get_system_CA_dirs**(word32 * num)This function returns a pointer to an array of strings representing directories wolfSSL will search for system CA certs when wolfSSL_CTX_load_system_CA_certs is called. On systems that don't store certificates in an accessible system directory (such as Apple platforms), this function will always return NULL.
int	**wolfSSL_CTX_load_system_CA_certs**(WOLFSSL_CTX * ctx)On most platforms (including Linux and Windows), this function attempts to load CA certificates into a WOLFSSL_CTX from an OS-dependent CA certificate store. Loaded certificates will be trusted.
int	**wolfSSL_CTX_trust_peer_cert**(WOLFSSL_CTX * ctx, const char * file, int type)This function loads a certificate to use for verifying a peer when performing a TLS/SSL handshake. The peer certificate sent during the handshake is compared by using the SKID when available and the signature. If these two things do not match then any loaded CAs are used. Feature is enabled by defining the macro WOLFSSL_TRUST_PEER_CERT. Please see the examples for proper usage.

	Name
int	**wolfSSL_CTX_use_certificate_chain_file**(WOLFSSL_CTX * ctx, const char * file)This function loads a chain of certificates into the SSL context (WOLFSSL_CTX). The file containing the certificate chain is provided by the file argument, and must contain PEM_formatted certificates. This function will process up to MAX_CHAIN_DEPTH (default = 9, defined in internal.h) certificates, plus the subject cert.
int	**wolfSSL_CTX_use_RSAPrivateKey_file function. The file argument contains a pointer to the RSA private key file, in the format specified by format.
long	**wolfSSL_get_verify_depth**(WOLFSSL * ssl)This function returns the maximum chain depth allowed, which is 9 by default, for a valid session i.e. there is a non_null session object (ssl).
long	**wolfSSL_CTX_get_verify_depth**(WOLFSSL_CTX * ctx)This function gets the certificate chaining depth using the CTX structure.
int	**wolfSSL_use_certificate_file**(WOLFSSL * ssl, const char * file, int format)This function loads a certificate file into the SSL session (WOLFSSL structure). The certificate file is provided by the file argument. The format argument specifies the format type of the file - either SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM.
int	**wolfSSL_use_PrivateKey_file**(WOLFSSL * ssl, const char * file, int format)This function loads a private key file into the SSL session (WOLFSSL structure). The key file is provided by the file argument. The format argument specifies the format type of the file - SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM.
int	**wolfSSL_use_certificate_chain_file**(WOLFSSL * ssl, const char * file)This function loads a chain of certificates into the SSL session (WOLFSSL structure). The file containing the certificate chain is provided by the file argument, and must contain PEM_formatted certificates. This function will process up to MAX_CHAIN_DEPTH (default = 9, defined in internal.h) certificates, plus the subject certificate.
int	**wolfSSL_use_RSAPrivateKey_file function. The file argument contains a pointer to the RSA private key file, in the format specified by format.

	Name
int	**wolfSSL_CTX_der_load_verify_locations**(WOLFSSL_CTX * ctx, const char * file, int format)This function is similar to wolfSSL_CTX_load_verify_locations, but allows the loading of DER_formatted CA files into the SSL context (WOLFSSL_CTX). It may still be used to load PEM_formatted CA files as well. These certificates will be treated as trusted root certificates and used to verify certs received from peers during the SSL handshake. The root certificate file, provided by the file argument, may be a single certificate or a file containing multiple certificates. If multiple CA certs are included in the same file, wolfSSL will load them in the same order they are presented in the file. The format argument specifies the format which the certificates are in either, SSL_FILETYPE_PEM or SSL_FILETYPE_ASN1 (DER). Unlike wolfSSL_CTX_load_verify_locations, this function does not allow the loading of CA certificates from a given directory path. Note that this function is only available when the wolfSSL library was compiled with WOLFSSL_DER_LOAD defined.
WOLFSSL_CTX *	**wolfSSL_CTX_new**(WOLFSSL_METHOD *)This function creates a new SSL context, taking a desired SSL/TLS protocol method for input.
WOLFSSL *	**wolfSSL_new**(WOLFSSL_CTX *)This function creates a new SSL session, taking an already created SSL context as input.
int	**wolfSSL_set_fd**(WOLFSSL * ssl, int fd)This function assigns a file descriptor (fd) as the input/output facility for the SSL connection. Typically this will be a socket file descriptor.
int	**wolfSSL_set_dtls_fd_connected**(WOLFSSL * ssl, int fd)This function assigns a file descriptor (fd) as the input/output facility for the SSL connection. Typically this will be a socket file descriptor. This is a DTLS specific API because it marks that the socket is connected. recvfrom and sendto calls on this fd will have the addr and addr_len parameters set to NULL.

	Name
int	**wolfDTLS_SetChGoodCb**(WOLFSSL * ssl, ClientHelloGoodCb cb, void * user_ctx)Allows setting a callback for a correctly processed and verified DTLS client hello. When using a cookie exchange mechanism (either the HelloVerifyRequest in DTLS 1.2 or the HelloRetryRequest with a cookie extension in DTLS 1.3) this callback is called after the cookie exchange has succeeded. This is useful to use one WOLFSSL object as the listener for new connections and being able to isolate the WOLFSSL object once the ClientHello is verified (either through a cookie exchange or just checking if the ClientHello had the correct format). DTLS 1.2: https://datatracker.ietf.org/doc/html/rfc6347#section_4.2.1 DTLS 1.3: https://www.rfc_editor.org/rfc/rfc8446#section_4.2.2.
char *	**wolfSSL_get_cipher_list**(int priority)Get the name of cipher at priority level passed in.
int	**wolfSSL_get_ciphers**(char * buf, int len)This function gets the ciphers enabled in wolfSSL.
const char *	**wolfSSL_get_cipher_name**(WOLFSSL * ssl)This function gets the cipher name in the format DHE-RSA by passing through argument to wolfSSL_get_cipher_name_internal.
int	**wolfSSL_get_fd**(const WOLFSSL *)This function returns the read file descriptor (fd) used as the input facility for the SSL connection. Typically this will be a socket file descriptor.
int	**wolfSSL_get_wfd**(const WOLFSSL *)This function returns the write file descriptor (fd) used as the output facility for the SSL connection. Typically this will be a socket file descriptor.
void	**wolfSSL_set_using_nonblock on it. This lets the WOLFSSL object know that receiving EWOULDBLOCK means that the recvfrom call would block rather than that it timed out.
int	**wolfSSL_get_using_nonblock on it. This lets the WOLFSSL object know that receiving EWOULDBLOCK means that the recvfrom call would block rather than that it timed out.
int	**wolfSSL_write will only return once the buffer data of size sz has been completely written or an error occurred.
int	**wolfSSL_read will trigger processing of the next record.
int	**wolfSSL_peek will trigger processing of the next record.

	Name
int	**wolfSSL_accept will only return once the handshake has been finished or an error occurred.
int	**wolfDTLS_accept_stateless**(WOLFSSL * ssl)This function is called on the server side and statelessly listens for an SSL client to initiate the DTLS handshake.
void	**wolfSSL_CTX_free**(WOLFSSL_CTX *)This function frees an allocated WOLFSSL_CTX object. This function decrements the CTX reference count and only frees the context when the reference count has reached 0.
void	**wolfSSL_free**(WOLFSSL *)This function frees an allocated wolfSSL object.
int	**wolfSSL_shutdown when the underlying I/O is ready.
int	**wolfSSL_send will only return once the buffer data of size sz has been completely written or an error occurred.
int	**wolfSSL_recv will trigger processing of the next record.
int	**wolfSSL_get_error for more information.
int	**wolfSSL_get_alert_history**(WOLFSSL * ssl, WOLFSSL_ALERT_HISTORY * h)This function gets the alert history.
int	**wolfSSL_set_session needs to be freed after the application is done with it by calling wolfSSL_SESSION_free() on it.
WOLFSSL_SESSION *	**wolfSSL_get_session for session resumption.
void	**wolfSSL_flush_sessions**(WOLFSSL_CTX * ctx, long tm)This function flushes session from the session cache which have expired. The time, tm, is used for the time comparison. Note that wolfSSL currently uses a static table for sessions, so no flushing is needed. As such, this function is currently just a stub. This function provides OpenSSL compatibility (SSL_flush_sessions) when wolfSSL is compiled with the OpenSSL compatibility layer.
int	**wolfSSL_SetServerID**(WOLFSSL * ssl, const unsigned char * id, int len, int newSession)This function associates the client session with the server id. If the newSession flag is on, an existing session won't be reused.
int	**wolfSSL_GetSessionIndex**(WOLFSSL * ssl)This function gets the session index of the WOLFSSL structure.

	Name
int	**wolfSSL_GetSessionAtIndex**(int idx, WOLFSSL_SESSION * session)This function gets the session at specified index of the session cache and copies it into memory. The WOLFSSL_SESSION structure holds the session information.
WOLFSSL_X509_CHAIN *	**wolfSSL_SESSION_get_peer_chain**(WOLFSSL_SESSION * session)Returns the peer certificate chain from the WOLFSSL_SESSION struct.
void	**wolfSSL_CTX_set_verify**(WOLFSSL_CTX * ctx, int mode, VerifyCallback verify_callback)This function sets the verification method for remote peers and also allows a verify callback to be registered with the SSL context. The verify callback will be called only when a verification failure has occurred. If no verify callback is desired, the NULL pointer can be used for verify_callback. The verification mode of peer certificates is a logically OR'd list of flags. The possible flag values include: SSL_VERIFY_NONE Client mode: the client will not verify the certificate received from the server and the handshake will continue as normal. Server mode: the server will not send a certificate request to the client. As such, client verification will not be enabled. SSL_VERIFY_PEER Client mode: the client will verify the certificate received from the server during the handshake. This is turned on by default in wolfSSL, therefore, using this option has no effect. Server mode: the server will send a certificate request to the client and verify the client certificate received. SSL_VERIFY_FAIL_IF_NO_PEER_CERT Client mode: no effect when used on the client side. Server mode: the verification will fail on the server side if the client fails to send a certificate when requested to do so (when using SSL_VERIFY_PEER on the SSL server). SSL_VERIFY_FAIL_EXCEPT_PSK Client mode: no effect when used on the client side. Server mode: the verification is the same as SSL_VERIFY_FAIL_IF_NO_PEER_CERT except in the case of a PSK connection. If a PSK connection is being made then the connection will go through without a peer cert.

	Name
void	**wolfSSL_set_verify**(WOLFSSL * ssl, int mode, VerifyCallback verify_callback)This function sets the verification method for remote peers and also allows a verify callback to be registered with the SSL session. The verify callback will be called only when a verification failure has occurred. If no verify callback is desired, the NULL pointer can be used for verify_callback. The verification mode of peer certificates is a logically OR'd list of flags. The possible flag values include: SSL_VERIFY_NONE Client mode: the client will not verify the certificate received from the server and the handshake will continue as normal. Server mode: the server will not send a certificate request to the client. As such, client verification will not be enabled. SSL_VERIFY_PEER Client mode: the client will verify the certificate received from the server during the handshake. This is turned on by default in wolfSSL, therefore, using this option has no effect. Server mode: the server will send a certificate request to the client and verify the client certificate received. SSL_VERIFY_FAIL_IF_NO_PEER_CERT Client mode: no effect when used on the client side. Server mode: the verification will fail on the server side if the client fails to send a certificate when requested to do so (when using SSL_VERIFY_PEER on the SSL server). SSL_VERIFY_FAIL_EXCEPT_PSK Client mode: no effect when used on the client side. Server mode: the verification is the same as SSL_VERIFY_FAIL_IF_NO_PEER_CERT except in the case of a PSK connection. If a PSK connection is being made then the connection will go through without a peer cert.
void	**wolfSSL_SetCertCbCtx**(WOLFSSL * ssl, void * ctx)This function stores user CTX object information for verify callback.
void	**wolfSSL_CTX_SetCertCbCtx**(WOLFSSL_CTX * ctx, void * userCtx)This function stores user CTX object information for verify callback.
int	**wolfSSL_pending.
void	**wolfSSL_load_error_strings**(void)This function is for OpenSSL compatibility (SSL_load_error_string) only and takes no action.
int	**wolfSSL_library_init is the more typically-used wolfSSL initialization function.
int	**wolfSSL_SetDevId**(WOLFSSL * ssl, int devId)This function sets the Device Id at the WOLFSSL session level.

	Name
int	**wolfSSL_CTX_SetDevId**(WOLFSSL_CTX * ctx, int devId)This function sets the Device Id at the WOLFSSL_CTX context level.
int	**wolfSSL_CTX_GetDevId**(WOLFSSL_CTX * ctx, WOLFSSL * ssl)This function retrieves the Device Id.
long	**wolfSSL_CTX_set_session_cache_mode**(WOLFSSL_CTX * ctx, long mode)This function enables or disables SSL session caching. Behavior depends on the value used for mode. The following values for mode are available: SSL_SESS_CACHE_OFF- disable session caching. Session caching is turned on by default. SSL_SESS_CACHE_NO_AUTO_CLEAR - Disable auto-flushing of the session cache. Auto-flushing is turned on by default.
int	**wolfSSL_set_session_secret_cb**(WOLFSSL * ssl, SessionSecretCb cb, void * ctx)This function sets the session secret callback function. The SessionSecretCb type has the signature: int (*SessionSecretCb)(WOLFSSL* ssl, void* secret, int* secretSz, void* ctx). The sessionSecretCb member of the WOLFSSL struct is set to the parameter cb.
int	**wolfSSL_save_session_cache**(const char * fname)This function persists the session cache to file. It doesn't use memsave because of additional memory use.
int	**wolfSSL_restore_session_cache**(const char * fname)This function restores the persistent session cache from file. It does not use memstore because of additional memory use.
int	**wolfSSL_memsave_session_cache**(void * mem, int sz)This function persists session cache to memory.
int	**wolfSSL_memrestore_session_cache**(const void * mem, int sz)This function restores the persistent session cache from memory.
int	**wolfSSL_get_session_cache_memsize**(void)This function returns how large the session cache save buffer should be.
int	**wolfSSL_CTX_save_cert_cache**(WOLFSSL_CTX * ctx, const char * fname)This function writes the cert cache from memory to file.
int	**wolfSSL_CTX_restore_cert_cache**(WOLFSSL_CTX * ctx, const char * fname)This function persistes certificate cache from a file.
int	**wolfSSL_CTX_memsave_cert_cache**(WOLFSSL_CTX * ctx, void * mem, int sz, int * used)This function persists the certificate cache to memory.

	Name
int	**wolfSSL_CTX_memrestore_cert_cache**(WOLFSSL_CTX * ctx, const void * mem, int sz)This function restores the certificate cache from memory.
int	**wolfSSL_CTX_get_cert_cache_memsize**(WOLFSSL_CTX * ctx)Returns the size the certificate cache save buffer needs to be.
int	**wolfSSL_CTX_set_cipher_list resets the cipher suite list for the specific SSL context to the provided list each time the function is called. The cipher suite list, list, is a null_terminated text string, and a colon_delimited list. For example, one value for list may be “DHE_RSA_AES256_SHA256:DHE_RSA_AES128_SHA256:AES256_SH Valid cipher values are the full name values from the cipher_names[] array in src/internal.c (for a definite list of valid cipher values check src/internal.c)
int	**wolfSSL_set_cipher_list resets the cipher suite list for the specific SSL session to the provided list each time the function is called. The cipher suite list, list, is a null_terminated text string, and a colon_delimited list. For example, one value for list may be “DHE_RSA_AES256_SHA256:DHE_RSA_AES128_SHA256:AES256_SH Valid cipher values are the full name values from the cipher_names[] array in src/internal.c (for a definite list of valid cipher values check src/internal.c)
void	**wolfSSL_dtls_set_using_nonblock**(WOLFSSL * ssl, int nonblock)This function informs the WOLFSSL DTLS object that the underlying UDP I/O is non_blocking. After an application creates a WOLFSSL object, if it will be used with a non_blocking UDP socket, call wolfSSL_dtls_set_using_nonblock() on it. This lets the WOLFSSL object know that receiving EWOULDBLOCK means that the recvfrom call would block rather than that it timed out.
int	**wolfSSL_dtls_get_using_nonblock**(WOLFSSL * ssl)This function allows the application to determine if wolfSSL is using non_blocking I/O with UDP. If wolfSSL is using non_blocking I/O, this function will return 1, otherwise 0. After an application creates a WOLFSSL object, if it will be used with a non_blocking UDP socket, call wolfSSL_dtls_set_using_nonblock() on it. This lets the WOLFSSL object know that receiving EWOULDBLOCK means that the recvfrom call would block rather than that it timed out. This function is only meaningful to DTLS sessions.

	Name
int	**wolfSSL_dtls_get_current_timeout**(WOLFSSL * ssl)This function returns the current timeout value in seconds for the WOLFSSL object. When using non-blocking sockets, something in the user code needs to decide when to check for available recv data and how long it has been waiting. The value returned by this function indicates how long the application should wait.
int	**wolfSSL_dtls13_use_quick_timeout**(WOLFSSL * ssl)This function returns true if the application should setup a quicker timeout. When using non_blocking sockets, something in the user code needs to decide when to check for available data and how long it needs to wait. If this function returns true, it means that the library already detected some disruption in the communication, but it wants to wait for a little longer in case some messages from the other peers are still in flight. Is up to the application to fine tune the value of this timer, a good one may be dtls_get_current_timeout() / 4.
void	**wolfSSL_dtls13_set_send_more_acks to determine if it should setup a quicker timeout to send those delayed ACKs.
int	**wolfSSL_dtls_set_timeout_init**(WOLFSSL * ssl, int)This function sets the dtls timeout.
int	**wolfSSL_dtls_set_timeout_max**(WOLFSSL * ssl, int)This function sets the maximum dtls timeout.
int	**wolfSSL_dtls_got_timeout**(WOLFSSL * ssl)When using non-blocking sockets with DTLS, this function should be called on the WOLFSSL object when the controlling code thinks the transmission has timed out. It performs the actions needed to retry the last transmit, including adjusting the timeout value. If it has been too long, this will return a failure.
int	**wolfSSL_dtls_retransmit**(WOLFSSL * ssl)When using non-blocking sockets with DTLS, this function retransmits the last handshake flight ignoring the expected timeout value and retransmit count. It is useful for applications that are using DTLS and need to manage even the timeout and retry count.
int	**wolfSSL_dtls**(WOLFSSL * ssl)This function is used to determine if the SSL session has been configured to use DTLS.
int	**wolfSSL_dtls_set_peer**(WOLFSSL * ssl, void * peer, unsigned int peerSz)This function sets the DTLS peer, peer (sockaddr_in) with size of peerSz.

	Name
int	**wolfSSL_dtls_set_pending_peer**(WOLFSSL * ssl, void * peer, unsigned int peerSz)This function sets the pending DTLS peer, peer (sockaddr_in) with size of peerSz. This sets the pending peer that will be upgraded to a regular peer when we successfully de-protect the next record. This is useful in scenarios where the peer's address can change to avoid off-path attackers from changing the peer address. This should be used with Connection ID's to allow seamless and safe transition to a new peer address.
int	**wolfSSL_dtls_get_peer**(WOLFSSL * ssl, void * peer, unsigned int * peerSz)This function gets the sockaddr_in (of size peerSz) of the current DTLS peer. The function will compare peerSz to the actual DTLS peer size stored in the SSL session. If the peer will fit into peer, the peer's sockaddr_in will be copied into peer, with peerSz set to the size of peer.
int	**wolfSSL_dtls_get0_peer**(WOLFSSL * ssl, const void ** peer, unsigned int * peerSz)This function gets the sockaddr_in (of size peerSz) of the current DTLS peer. This is a zero_copy alternative to wolfSSL_dtls_get_peer().
char *	**wolfSSL_ERR_error_string and data is the storage buffer which the error string will be placed in. The maximum length of data is 80 characters by default, as defined by MAX_ERROR_SZ is wolfssl/wolfcrypt/error.h.
void	**wolfSSL_ERR_error_string_n into a more human-readable error string. The human-readable string is placed in buf.
int	**wolfSSL_get_shutdown**(const WOLFSSL * ssl)This function checks the shutdown conditions in closeNotify or connReset or sentNotify members of the Options structure. The Options structure is within the WOLFSSL structure.
int	**wolfSSL_session_reused**(WOLFSSL * ssl)This function returns the resuming member of the options struct. The flag indicates whether or not to reuse a session. If not, a new session must be established.
int	**wolfSSL_is_init_finished**(WOLFSSL * ssl)This function checks to see if the connection is established.
const char *	**wolfSSL_get_version**(WOLFSSL * ssl)Returns the SSL version being used as a string.
int	**wolfSSL_get_current_cipher_suite**(WOLFSSL * ssl)Returns the current cipher suit an ssl session is using.

	Name
WOLFSSL_CIPHER *	**wolfSSL_get_current_cipher**(WOLFSSL * ssl)This function returns a pointer to the current cipher in the ssl session.
const char *	**wolfSSL_CIPHER_get_name**(const WOLFSSL_CIPHER * cipher)This function matches the cipher suite in the SSL object with the available suites and returns the string representation.
const char *	**wolfSSL_get_cipher**(WOLFSSL *)This function matches the cipher suite in the SSL object with the available suites.
WOLFSSL_SESSION *	**wolfSSL_get1_session needs to be freed after the application is done with it by calling wolfSSL_SESSION_free() on it.
WOLFSSL_METHOD *	**wolfSSLv23_client_method function will use the highest protocol version supported by the server and downgrade to SSLv3 if needed. In this case, the client will be able to connect to a server running SSLv3 - TLSv1.3.
int	**wolfSSL_BIO_get_mem_data**(WOLFSSL_BIO * bio, void * p)This is used to set a byte pointer to the start of the internal memory buffer.
long	**wolfSSL_BIO_set_fd**(WOLFSSL_BIO * b, int fd, int flag)Sets the file descriptor for bio to use.
int	**wolfSSL_BIO_set_close**(WOLFSSL_BIO * b, long flag)Sets the close flag, used to indicate that the i/o stream should be closed when the BIO is freed.
WOLFSSL_BIO_METHOD *	**wolfSSL_BIO_s_socket**(void)This is used to get a BIO_SOCKET type WOLFSSL_BIO_METHOD.
int	**wolfSSL_BIO_set_write_buf_size**(WOLFSSL_BIO * b, long size)This is used to set the size of write buffer for a WOLFSSL_BIO. If write buffer has been previously set this function will free it when resetting the size. It is similar to wolfSSL_BIO_reset in that it resets read and write indexes to 0.
int	**wolfSSL_BIO_make_bio_pair**(WOLFSSL_BIO * b1, WOLFSSL_BIO * b2)This is used to pair two bios together. A pair of bios acts similar to a two way pipe writing to one can be read by the other and vice versa. It is expected that both bios be in the same thread, this function is not thread safe. Freeing one of the two bios removes both from being paired. If a write buffer size was not previously set for either of the bios it is set to a default size of 17000 (WOLFSSL_BIO_SIZE) before being paired.
int	**wolfSSL_BIO_ctrl_reset_read_request**(WOLFSSL_BIO * bio)This is used to set the read request flag back to 0.

	Name
int	**wolfSSL_BIO_nread0**(WOLFSSL_BIO * bio, char ** buf)This is used to get a buffer pointer for reading from. Unlike wolfSSL_BIO_nread the internal read index is not advanced by the number returned from the function call. Reading past the value returned can result in reading out of array bounds.
int	**wolfSSL_BIO_nread**(WOLFSSL_BIO * bio, char ** buf, int num)This is used to get a buffer pointer for reading from. The internal read index is advanced by the number returned from the function call with buf being pointed to the beginning of the buffer to read from. In the case that less bytes are in the read buffer than the value requested with num the lesser value is returned. Reading past the value returned can result in reading out of array bounds.
int	**wolfSSL_BIO_nwrite**(WOLFSSL_BIO * bio, char ** buf, int num)Gets a pointer to the buffer for writing as many bytes as returned by the function. Writing more bytes to the pointer returned then the value returned can result in writing out of bounds.
int	**wolfSSL_BIO_reset**(WOLFSSL_BIO * bio)Resets bio to an initial state. As an example for type BIO_BIO this resets the read and write index.
int	**wolfSSL_BIO_seek**(WOLFSSL_BIO * bio, int ofs)This function adjusts the file pointer to the offset given. This is the offset from the head of the file.
int	**wolfSSL_BIO_write_filename**(WOLFSSL_BIO * bio, char * name)This is used to set and write to a file. WIll overwrite any data currently in the file and is set to close the file when the bio is freed.
long	**wolfSSL_BIO_set_mem_eof_return**(WOLFSSL_BIO * bio, int v)This is used to set the end of file value. Common value is -1 so as not to get confused with expected positive values.
long	**wolfSSL_BIO_get_mem_ptr**(WOLFSSL_BIO * bio, WOLFSSL_BUF_MEM ** m)This is a getter function for WOLFSSL_BIO memory pointer.
char *	**wolfSSL_X509_NAME_oneline**(WOLFSSL_X509_NAME * name, char * in, int sz)This function copies the name of the x509 into a buffer.
WOLFSSL_X509_NAME *	**wolfSSL_X509_get_issuer_name**(WOLFSSL_X509 * cert)This function returns the name of the certificate issuer.
WOLFSSL_X509_NAME *	**wolfSSL_X509_get_subject_name**(WOLFSSL_X509 * cert)This function returns the subject member of the WOLFSSL_X509 structure.

	Name
int	**wolfSSL_X509_get_isCA**(WOLFSSL_X509 * cert)Checks the isCa member of the WOLFSSL_X509 structure and returns the value.
int	**wolfSSL_X509_NAME_get_text_by_NID**(WOLFSSL_X509_NAME * name, int nid, char * buf, int len)This function gets the text related to the passed in NID value.
int	**wolfSSL_X509_get_signature_type**(WOLFSSL_X509 * cert)This function returns the value stored in the sigOID member of the WOLFSSL_X509 structure.
void	**wolfSSL_X509_free**(WOLFSSL_X509 * x509)This function frees a WOLFSSL_X509 structure.
int	**wolfSSL_X509_get_signature**(WOLFSSL_X509 * x509, unsigned char * buf, int * bufSz)Gets the X509 signature and stores it in the buffer.
int	**wolfSSL_X509_STORE_add_cert**(WOLFSSL_X509_STORE * store, WOLFSSL_X509 * x509)This function adds a certificate to the WOLFSSL_X509_STRE structure.
WOLFSSL_STACK *	**wolfSSL_X509_STORE_CTX_get_chain**(WOLFSSL_X509_STORE_CT * ctx)This function is a getter function for chain variable in WOLFSSL_X509_STORE_CTX structure. Currently chain is not populated.
int	**wolfSSL_X509_STORE_set_flags**(WOLFSSL_X509_STORE * store, unsigned long flag)This function takes in a flag to change the behavior of the WOLFSSL_X509_STORE structure passed in. An example of a flag used is WOLFSSL_CRL_CHECK.
const byte *	**wolfSSL_X509_notBefore**(WOLFSSL_X509 * x509)This function the certificate “not before” validity encoded as a byte array.
const byte *	**wolfSSL_X509_notAfter**(WOLFSSL_X509 * x509)This function the certificate “not after” validity encoded as a byte array.
WOLFSSL_BIGNUM *	**wolfSSL_ASN1_INTEGER_to_BN**(const WOLFSSL_ASN1_INTEGER * ai, WOLFSSL_BIGNUM * bn)This function is used to copy a WOLFSSL_ASN1_INTEGER value to a WOLFSSL_BIGNUM structure.
long	**wolfSSL_CTX_add_extra_chain_cert**(WOLFSSL_CTX * ctx, WOLFSSL_X509 * x509)This function adds the certificate to the internal chain being built in the WOLFSSL_CTX structure.
int	**wolfSSL_CTX_get_read_ahead**(WOLFSSL_CTX * ctx)This function returns the get read ahead flag from a WOLFSSL_CTX structure.
int	**wolfSSL_CTX_set_read_ahead**(WOLFSSL_CTX * ctx, int v)This function sets the read ahead flag in the WOLFSSL_CTX structure.
long	**wolfSSL_CTX_set_tlsext_status_arg**(WOLFSSL_CTX * ctx, void * arg)This function sets the options argument to use with OCSP.

	Name
long	**wolfSSL_CTX_set_tlsext_opaque_prf_input_callback_arg**(WOLF * ctx, void * arg)This function sets the optional argument to be passed to the PRF callback.
long	**wolfSSL_set_options**(WOLFSSL * s, long op)This function sets the options mask in the ssl. Some valid options are, SSL_OP_ALL, SSL_OP_COOKIE_EXCHANGE, SSL_OP_NO_SSLv2, SSL_OP_NO_SSLv3, SSL_OP_NO_TLSv1, SSL_OP_NO_TLSv1_1, SSL_OP_NO_TLSv1_2, SSL_OP_NO_COMPRESSION.
long	**wolfSSL_get_options**(const WOLFSSL * ssl)This function returns the current options mask.
long	**wolfSSL_set_tlsext_debug_arg**(WOLFSSL * ssl, void * arg)This is used to set the debug argument passed around.
long	**wolfSSL_set_tlsext_status_type**(WOLFSSL * s, int type)This function is called when the client application request that a server send back an OCSP status response (also known as OCSP stapling).Currently, the only supported type is TLSEXT_STATUSTYPE_ocsp.
long	**wolfSSL_get_verify_result**(const WOLFSSL * ssl)This is used to get the results after trying to verify the peer's certificate.
void	**wolfSSL_ERR_print_errors_fp and fp is the file which the error string will be placed in.
void	**wolfSSL_ERR_print_errors_cb**(int*)(const char* str, size_t len, void *u) cb, void* u)This function uses the provided callback to handle error reporting. The callback function is executed for each error line. The string, length, and userdata are passed into the callback parameters.
void	**wolfSSL_CTX_set_psk_client_callback**(WOLFSSL_CTX * ctx, wc_psk_client_callback cb)The function sets the client_psk_cb member of the WOLFSSL_CTX structure.
void	**wolfSSL_set_psk_client_callback**(WOLFSSL * ssl, wc_psk_client_callback)Sets the PSK client side callback.
const char *	**wolfSSL_get_psk_identity_hint**(const WOLFSSL *)This function returns the psk identity hint.
const char *	**wolfSSL_get_psk_identity**(const WOLFSSL *)The function returns a constant pointer to the client_identity member of the Arrays structure.
int	**wolfSSL_CTX_use_psk_identity_hint**(WOLFSSL_CTX * ctx, const char * hint)This function stores the hint argument in the server_hint member of the WOLFSSL_CTX structure.

	Name
int	**wolfSSL_use_psk_identity_hint**(WOLFSSL * ssl, const char * hint)This function stores the hint argument in the server_hint member of the Arrays structure within the WOLFSSL structure.
void	**wolfSSL_CTX_set_psk_server_callback**(WOLFSSL_CTX * ctx, wc_psk_server_callback cb)This function sets the psk callback for the server side in the WOLFSSL_CTX structure.
void	**wolfSSL_set_psk_server_callback**(WOLFSSL * ssl, wc_psk_server_callback cb)Sets the psk callback for the server side by setting the WOLFSSL structure options members.
int	**wolfSSL_set_psk_callback_ctx**(WOLFSSL * ssl, void * psk_ctx)Sets a PSK user context in the WOLFSSL structure options member.
int	**wolfSSL_CTX_set_psk_callback_ctx**(WOLFSSL_CTX * ctx, void * psk_ctx)Sets a PSK user context in the WOLFSSL_CTX structure.
void *	**wolfSSL_get_psk_callback_ctx**(WOLFSSL * ssl)Get a PSK user context in the WOLFSSL structure options member.
void *	**wolfSSL_CTX_get_psk_callback_ctx**(WOLFSSL_CTX * ctx)Get a PSK user context in the WOLFSSL_CTX structure.
int	**wolfSSL_CTX_allow_anon_cipher**(WOLFSSL_CTX *)This function enables the havAnon member of the CTX structure if HAVE_ANON is defined during compilation.
WOLFSSL_METHOD *	**wolfSSLv23_server_method.
int	**wolfSSL_state**(WOLFSSL * ssl)This is used to get the internal error state of the WOLFSSL structure.
WOLFSSL_X509 *	**wolfSSL_get_peer_certificate**(WOLFSSL * ssl)This function gets the peer's certificate.
int	**wolfSSL_want_read and getting SSL_ERROR_WANT_READ in return. If the underlying error state is SSL_ERROR_WANT_READ, this function will return 1, otherwise, 0.
int	**wolfSSL_want_write and getting SSL_ERROR_WANT_WRITE in return. If the underlying error state is SSL_ERROR_WANT_WRITE, this function will return 1, otherwise, 0.
int	**wolfSSL_check_domain_name will add a domain name check to the list of checks to perform. dn holds the domain name to check against the peer certificate when it's received.
int	**wolfSSL_Init**(void)Initializes the wolfSSL library for use. Must be called once per application and before any other call to the library.

	Name
int	**wolfSSL_Cleanup**(void)Un-initializes the wolfSSL library from further use. Doesn't have to be called, though it will free any resources used by the library.
const char *	**wolfSSL_lib_version**(void)This function returns the current library version.
word32	**wolfSSL_lib_version_hex**(void)This function returns the current library version in hexadecimal notation.
int	**wolfSSL_negotiate is performed if called from the server side.
int	**wolfSSL_set_compression**(WOLFSSL * ssl)Turns on the ability to use compression for the SSL connection. Both sides must have compression turned on otherwise compression will not be used. The zlib library performs the actual data compression. To compile into the library use –with-libz for the configure system and define HAVE_LIBZ otherwise. Keep in mind that while compressing data before sending decreases the actual size of the messages being sent and received, the amount of data saved by compression usually takes longer in time to analyze than it does to send it raw on all but the slowest of networks.
int	**wolfSSL_set_timeout**(WOLFSSL * ssl, unsigned int to)This function sets the SSL session timeout value in seconds.
int	**wolfSSL_CTX_set_timeout**(WOLFSSL_CTX * ctx, unsigned int to)This function sets the timeout value for SSL sessions, in seconds, for the specified SSL context.
WOLFSSL_X509_CHAIN *	**wolfSSL_get_peer_chain**(WOLFSSL * ssl)Retrieves the peer's certificate chain.
int	**wolfSSL_get_chain_count**(WOLFSSL_X509_CHAIN * chain)Retrieve's the peers certificate chain count.
int	**wolfSSL_get_chain_length**(WOLFSSL_X509_CHAIN * chain, int idx)Retrieves the peer's ASN1.DER certificate length in bytes at index (idx).
unsigned char *	**wolfSSL_get_chain_cert**(WOLFSSL_X509_CHAIN * chain, int idx)Retrieves the peer's ASN1.DER certificate at index (idx).
WOLFSSL_X509 *	**wolfSSL_get_chain_X509**(WOLFSSL_X509_CHAIN * chain, int idx)This function gets the peer's wolfSSL_X509_certificate at index (idx) from the chain of certificates.
int	**wolfSSL_get_chain_cert_pem**(WOLFSSL_X509_CHAIN * chain, int idx, unsigned char * buf, int inLen, int * outLen)Retrieves the peer's PEM certificate at index (idx).

	Name
const unsigned char *	**wolfSSL_get_sessionID**(const WOLFSSL_SESSION * s)Retrieves the session's ID. The session ID is always 32 bytes long.
int	**wolfSSL_X509_get_serial_number**(WOLFSSL_X509 * x509, unsigned char * in, int * inOutSz)Retrieves the peer's certificate serial number. The serial number buffer (in) should be at least 32 bytes long and be provided as the *inOutSz argument as input. After calling the function* inOutSz will hold the actual length in bytes written to the in buffer.
char *	**wolfSSL_X509_get_subjectCN**(WOLFSSL_X509 *)Returns the common name of the subject from the certificate.
const unsigned char *	**wolfSSL_X509_get_der**(WOLFSSL_X509 * x509, int * outSz)This function gets the DER encoded certificate in the WOLFSSL_X509 struct.
WOLFSSL_ASN1_TIME *	**wolfSSL_X509_get_notAfter**(WOLFSSL_X509 *)This function checks to see if x509 is NULL and if it's not, it returns the notAfter member of the x509 struct.
int	**wolfSSL_X509_version**(WOLFSSL_X509 *)This function retrieves the version of the X509 certificate.
WOLFSSL_X509 *	**wolfSSL_X509_d2i_fp**(WOLFSSL_X509 ** x509, FILE * file)If NO_STDIO_FILESYSTEM is defined this function will allocate heap memory, initialize a WOLFSSL_X509 structure and return a pointer to it.
WOLFSSL_X509 *	**wolfSSL_X509_load_certificate_file**(const char * fname, int format)The function loads the x509 certificate into memory.
unsigned char *	**wolfSSL_X509_get_device_type**(WOLFSSL_X509 * x509, unsigned char * in, int * inOutSz)This function copies the device type from the x509 structure to the buffer.
unsigned char *	**wolfSSL_X509_get_hw_type**(WOLFSSL_X509 * x509, unsigned char * in, int * inOutSz)The function copies the hwType member of the WOLFSSL_X509 structure to the buffer.
unsigned char *	**wolfSSL_X509_get_hw_serial_number**(WOLFSSL_X509 * x509, unsigned char * in, int * inOutSz)This function returns the hwSerialNum member of the x509 object.
int	**wolfSSL_connect_cert will only return once the peer's certificate chain has been received.

	Name
WC_PKCS12 *	**wolfSSL_d2i_PKCS12_bio**(WOLFSSL_BIO * bio, WC_PKCS12 ** pkcs12)wolfSSL_d2i_PKCS12_bio (d2i_PKCS12_bio) copies in the PKCS12 information from WOLFSSL_BIO to the structure WC_PKCS12. The information is divided up in the structure as a list of Content Infos along with a structure to hold optional MAC information. After the information has been divided into chunks (but not decrypted) in the structure WC_PKCS12, it can then be parsed and decrypted by calling.
WC_PKCS12 *	**wolfSSL_i2d_PKCS12_bio**(WOLFSSL_BIO * bio, WC_PKCS12 * pkcs12)wolfSSL_i2d_PKCS12_bio (i2d_PKCS12_bio) copies in the cert information from the structure WC_PKCS12 to WOLFSSL_BIO.
int	**wolfSSL_PKCS12_parse(WOLFSSL_X509)** ca)PKCS12 can be enabled with adding –enable_opensslextra to the configure command. It can use triple DES and RC4 for decryption so would recommend also enabling these features when enabling opensslextra (–enable_des3 –enable_arc4). wolfSSL does not currently support RC2 so decryption with RC2 is currently not available. This may be noticeable with default encryption schemes used by OpenSSL command line to create .p12 files. wolfSSL_PKCS12_parse (PKCS12_parse). The first thing this function does is check the MAC is correct if present. If the MAC fails then the function returns and does not try to decrypt any of the stored Content Infos. This function then parses through each Content Info looking for a bag type, if the bag type is known it is decrypted as needed and either stored in the list of certificates being built or as a key found. After parsing through all bags the key found is then compared with the certificate list until a matching pair is found. This matching pair is then returned as the key and certificate, optionally the certificate list found is returned as a STACK_OF certificates. At the moment a CRL, Secret or SafeContents bag will be skipped over and not parsed. It can be seen if these or other “Unknown” bags are skipped over by viewing the debug print out. Additional attributes such as friendly name are skipped over when parsing a PKCS12 file.

	Name
int	**wolfSSL_SetTmpDH**(WOLFSSL * ssl, const unsigned char * p, int pSz, const unsigned char * g, int gSz)Server Diffie-Hellman Ephemeral parameters setting. This function sets up the group parameters to be used if the server negotiates a cipher suite that uses DHE.
int	**wolfSSL_SetTmpDH_buffer**(WOLFSSL * ssl, const unsigned char * b, long sz, int format)The function calls the wolfSSL_SetTMpDH_buffer_wrapper, which is a wrapper for Diffie-Hellman parameters.
int	**wolfSSL_SetTmpDH_file**(WOLFSSL * ssl, const char * f, int format)This function calls wolfSSL_SetTmpDH_file_wrapper to set server Diffie-Hellman parameters.
int	**wolfSSL_CTX_SetTmpDH**(WOLFSSL_CTX * ctx, const unsigned char * p, int pSz, const unsigned char * g, int gSz)Sets the parameters for the server CTX Diffie-Hellman.
int	**wolfSSL_CTX_SetTmpDH_buffer**(WOLFSSL_CTX * ctx, const unsigned char * b, long sz, int format)A wrapper function that calls wolfSSL_SetTmpDH_buffer_wrapper.
int	**wolfSSL_CTX_SetTmpDH_file**(WOLFSSL_CTX * ctx, const char * f, int format)The function calls wolfSSL_SetTmpDH_file_wrapper to set the server Diffie-Hellman parameters.
int	**wolfSSL_CTX_SetMinDhKey_Sz**(WOLFSSL_CTX * ctx, word16)This function sets the minimum size (in bits) of the Diffie Hellman key size by accessing the minDhKeySz member in the WOLFSSL_CTX structure.
int	**wolfSSL_SetMinDhKey_Sz**(WOLFSSL * ssl, word16 keySz_bits)Sets the minimum size (in bits) for a Diffie-Hellman key in the WOLFSSL structure.
int	**wolfSSL_CTX_SetMaxDhKey_Sz**(WOLFSSL_CTX * ctx, word16 keySz_bits)This function sets the maximum size (in bits) of the Diffie Hellman key size by accessing the maxDhKeySz member in the WOLFSSL_CTX structure.
int	**wolfSSL_SetMaxDhKey_Sz**(WOLFSSL * ssl, word16 keySz_bits)Sets the maximum size (in bits) for a Diffie-Hellman key in the WOLFSSL structure.
int	**wolfSSL_GetDhKey_Sz**(WOLFSSL *)Returns the value of dhKeySz (in bits) that is a member of the options structure. This value represents the Diffie-Hellman key size in bytes.

	Name
int	**wolfSSL_CTX_SetMinRsaKey_Sz**(WOLFSSL_CTX * ctx, short keySz)Sets the minimum RSA key size in both the WOLFSSL_CTX structure and the WOLFSSL_CERT_MANAGER structure.
int	**wolfSSL_SetMinRsaKey_Sz**(WOLFSSL * ssl, short keySz)Sets the minimum allowable key size in bits for RSA located in the WOLFSSL structure.
int	**wolfSSL_CTX_SetMinEccKey_Sz**(WOLFSSL_CTX * ssl, short keySz)Sets the minimum size in bits for the ECC key in the WOLF_CTX structure and the WOLFSSL_CERT_MANAGER structure.
int	**wolfSSL_SetMinEccKey_Sz**(WOLFSSL * ssl, short keySz)Sets the value of the minEccKeySz member of the options structure. The options struct is a member of the WOLFSSL structure and is accessed through the ssl parameter.
int	**wolfSSL_make_eap_keys**(WOLFSSL * ssl, void * key, unsigned int len, const char * label)This function is used by EAP_TLS and EAP-TTLS to derive keying material from the master secret.
int	**wolfSSL_writev**(WOLFSSL * ssl, const struct iovec * iov, int iovcnt)Simulates writev semantics but doesn't actually do block at a time because of SSL_write() behavior and because front adds may be small. Makes porting into software that uses writev easier.
int	**wolfSSL_CTX_UnloadCAs**(WOLFSSL_CTX *)This function unloads the CA signer list and frees the whole signer table.
int	**wolfSSL_CTX_UnloadIntermediateCerts**(WOLFSSL_CTX * ctx)This function unloads intermediate certificates added to the CA signer list and frees them.
int	**wolfSSL_CTX_Unload_trust_peers**(WOLFSSL_CTX *)This function is used to unload all previously loaded trusted peer certificates. Feature is enabled by defining the macro WOLFSSL_TRUST_PEER_CERT.

	Name
int	**wolfSSL_CTX_trust_peer_buffer**(WOLFSSL_CTX * ctx, const unsigned char * in, long sz, int format)This function loads a certificate to use for verifying a peer when performing a TLS/SSL handshake. The peer certificate sent during the handshake is compared by using the SKID when available and the signature. If these two things do not match then any loaded CAs are used. Is the same functionality as wolfSSL_CTX_trust_peer_cert except is from a buffer instead of a file. Feature is enabled by defining the macro WOLFSSL_TRUST_PEER_CERT Please see the examples for proper usage.
int	**wolfSSL_CTX_load_verify_buffer**(WOLFSSL_CTX * ctx, const unsigned char * in, long sz, int format)This function loads a CA certificate buffer into the WOLFSSL Context. It behaves like the non-buffered version, only differing in its ability to be called with a buffer as input instead of a file. The buffer is provided by the in argument of size sz. format specifies the format type of the buffer; SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM. More than one CA certificate may be loaded per buffer as long as the format is in PEM. Please see the examples for proper usage.
int	**wolfSSL_CTX_load_verify_buffer_ex**(WOLFSSL_CTX * ctx, const unsigned char * in, long sz, int format, int userChain, word32 flags)This function loads a CA certificate buffer into the WOLFSSL Context. It behaves like the non-buffered version, only differing in its ability to be called with a buffer as input instead of a file. The buffer is provided by the in argument of size sz. format specifies the format type of the buffer; SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM. More than one CA certificate may be loaded per buffer as long as the format is in PEM. The _ex version was added in PR 2413 and supports additional arguments for userChain and flags.

	Name
int	**wolfSSL_CTX_load_verify_chain_buffer_format**(WOLFSSL_CTX * ctx, const unsigned char * in, long sz, int format)This function loads a CA certificate chain buffer into the WOLFSSL Context. It behaves like the non-buffered version, only differing in its ability to be called with a buffer as input instead of a file. The buffer is provided by the in argument of size sz. format specifies the format type of the buffer; SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM. More than one CA certificate may be loaded per buffer as long as the format is in PEM. Please see the examples for proper usage.
int	**wolfSSL_CTX_use_certificate_buffer**(WOLFSSL_CTX * ctx, const unsigned char * in, long sz, int format)This function loads a certificate buffer into the WOLFSSL Context. It behaves like the non-buffered version, only differing in its ability to be called with a buffer as input instead of a file. The buffer is provided by the in argument of size sz. format specifies the format type of the buffer; SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM. Please see the examples for proper usage.
int	**wolfSSL_CTX_use_PrivateKey_buffer**(WOLFSSL_CTX * ctx, const unsigned char * in, long sz, int format)This function loads a private key buffer into the SSL Context. It behaves like the non-buffered version, only differing in its ability to be called with a buffer as input instead of a file. The buffer is provided by the in argument of size sz. format specifies the format type of the buffer; SSL_FILETYPE_ASN1or SSL_FILETYPE_PEM. Please see the examples for proper usage.
int	**wolfSSL_CTX_use_certificate_chain_buffer**(WOLFSSL_CTX * ctx, const unsigned char * in, long sz)This function loads a certificate chain buffer into the WOLFSSL Context. It behaves like the non-buffered version, only differing in its ability to be called with a buffer as input instead of a file. The buffer is provided by the in argument of size sz. The buffer must be in PEM format and start with the subject's certificate, ending with the root certificate. Please see the examples for proper usage.

	Name
int	**wolfSSL_use_certificate_buffer**(WOLFSSL * ssl, const unsigned char * in, long sz, int format)This function loads a certificate buffer into the WOLFSSL object. It behaves like the non-buffered version, only differing in its ability to be called with a buffer as input instead of a file. The buffer is provided by the in argument of size sz. format specifies the format type of the buffer; SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM. Please see the examples for proper usage.
int	**wolfSSL_use_PrivateKey_buffer**(WOLFSSL * ssl, const unsigned char * in, long sz, int format)This function loads a private key buffer into the WOLFSSL object. It behaves like the non-buffered version, only differing in its ability to be called with a buffer as input instead of a file. The buffer is provided by the in argument of size sz. format specifies the format type of the buffer; SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM. Please see the examples for proper usage.
int	**wolfSSL_use_certificate_chain_buffer**(WOLFSSL * ssl, const unsigned char * in, long sz)This function loads a certificate chain buffer into the WOLFSSL object. It behaves like the non-buffered version, only differing in its ability to be called with a buffer as input instead of a file. The buffer is provided by the in argument of size sz. The buffer must be in PEM format and start with the subject's certificate, ending with the root certificate. Please see the examples for proper usage.
int	**wolfSSL_UnloadCertsKeys**(WOLFSSL *)This function unloads any certificates or keys that SSL owns.
int	**wolfSSL_CTX_set_group_messages**(WOLFSSL_CTX *)This function turns on grouping of handshake messages where possible.
int	**wolfSSL_set_group_messages**(WOLFSSL *)This function turns on grouping of handshake messages where possible.
void	**wolfSSL_SetFuzzerCb**(WOLFSSL * ssl, CallbackFuzzer cbf, void * fCtx)This function sets the fuzzer callback.
int	**wolfSSL_DTLS_SetCookieSecret**(WOLFSSL * ssl, const unsigned char * secret, unsigned int secretSz)This function sets a new dtls cookie secret.
WC_RNG *	**wolfSSL_GetRNG**(WOLFSSL * ssl)This function retrieves the random number.

	Name
int	**wolfSSL_CTX_SetMinVersion**(WOLFSSL_CTX * ctx, int version)This function sets the minimum downgrade version allowed. Applicable only when the connection allows downgrade using (wolfSSLv23_client_method or wolfSSLv23_server_method).
int	**wolfSSL_SetMinVersion**(WOLFSSL * ssl, int version)This function sets the minimum downgrade version allowed. Applicable only when the connection allows downgrade using (wolfSSLv23_client_method or wolfSSLv23_server_method).
int	**wolfSSL_GetObjectSize**(void)This function returns the size of the WOLFSSL object and will be dependent on build options and settings. If SHOW_SIZES has been defined when building wolfSSL, this function will also print the sizes of individual objects within the WOLFSSL object (Suites, Ciphers, etc.) to stdout.
int	**wolfSSL_GetOutputSize**(WOLFSSL * ssl, int inSz)Returns the record layer size of the plaintext input. This is helpful when an application wants to know how many bytes will be sent across the Transport layer, given a specified plaintext input size. This function must be called after the SSL/TLS handshake has been completed.
int	**wolfSSL_GetMaxOutputSize**(WOLFSSL *)Returns the maximum record layer size for plaintext data. This will correspond to either the maximum SSL/TLS record size as specified by the protocol standard, the maximum TLS fragment size as set by the TLS Max Fragment Length extension. This function is helpful when the application has called wolfSSL_GetOutputSize() and received a INPUT_SIZE_E error. This function must be called after the SSL/TLS handshake has been completed.
int	**wolfSSL_SetVersion) method type.

	Name
void	**wolfSSL_CTX_SetMacEncryptCb**(WOLFSSL_CTX * ctx, CallbackMacEncrypti cb)Allows caller to set the Atomic User Record Processing Mac/Encrypt Callback. The callback should return 0 for success or < 0 for an error. The ssl and ctx pointers are available for the user's convenience. macOut is the output buffer where the result of the mac should be stored. macIn is the mac input buffer and macInSz notes the size of the buffer. macContent and macVerify are needed for wolfSSL_SetTlsHmacInner() and be passed along as is. encOut is the output buffer where the result on the encryption should be stored. encIn is the input buffer to encrypt while encSz is the size of the input. An example callback can be found wolfssl/test.h myMacEncryptCb().
void	**wolfSSL_SetMacEncryptCtx**(WOLFSSL * ssl, void * ctx)Allows caller to set the Atomic User Record Processing Mac/Encrypt Callback Context to ctx.
void *	**wolfSSL_GetMacEncryptCtx**(WOLFSSL * ssl)Allows caller to retrieve the Atomic User Record Processing Mac/Encrypt Callback Context previously stored with wolfSSL_SetMacEncryptCtx().
void	**wolfSSL_CTX_SetDecryptVerifyCb**(WOLFSSL_CTX * ctx, CallbackDecryptVerify cb)Allows caller to set the Atomic User Record Processing Decrypt/Verify Callback. The callback should return 0 for success or < 0 for an error. The ssl and ctx pointers are available for the user's convenience. decOut is the output buffer where the result of the decryption should be stored. decIn is the encrypted input buffer and decInSz notes the size of the buffer. content and verify are needed for wolfSSL_SetTlsHmacInner() and be passed along as is. padSz is an output variable that should be set with the total value of the padding. That is, the mac size plus any padding and pad bytes. An example callback can be found wolfssl/test.h myDecryptVerifyCb().
void	**wolfSSL_SetDecryptVerifyCtx**(WOLFSSL * ssl, void * ctx)Allows caller to set the Atomic User Record Processing Decrypt/Verify Callback Context to ctx.
void *	**wolfSSL_GetDecryptVerifyCtx**(WOLFSSL * ssl)Allows caller to retrieve the Atomic User Record Processing Decrypt/Verify Callback Context previously stored with wolfSSL_SetDecryptVerifyCtx().

	Name
const unsigned char *	**wolfSSL_GetMacSecret**(WOLFSSL * ssl, int verify)Allows retrieval of the Hmac/Mac secret from the handshake process. The verify parameter specifies whether this is for verification of a peer message.
const unsigned char *	**wolfSSL_GetClientWriteKey**(WOLFSSL *)Allows retrieval of the client write key from the handshake process.
const unsigned char *	**wolfSSL_GetClientWriteIV**(WOLFSSL *)Allows retrieval of the client write IV (initialization vector) from the handshake process.
const unsigned char *	**wolfSSL_GetServerWriteKey**(WOLFSSL *)Allows retrieval of the server write key from the handshake process.
const unsigned char *	**wolfSSL_GetServerWriteIV**(WOLFSSL *)Allows retrieval of the server write IV (initialization vector) from the handshake process.
int	**wolfSSL_GetKeySize**(WOLFSSL *)Allows retrieval of the key size from the handshake process.
int	**wolfSSL_GetIVSize**(WOLFSSL *)Returns the iv_size member of the specs structure held in the WOLFSSL struct.
int	**wolfSSL_GetSide**(WOLFSSL *)Allows retrieval of the side of this WOLFSSL connection.
int	**wolfSSL_IsTLSv1_1**(WOLFSSL *)Allows caller to determine if the negotiated protocol version is at least TLS version 1.1 or greater.
int	**wolfSSL_GetBulkCipher**(WOLFSSL *)Allows caller to determine the negotiated bulk cipher algorithm from the handshake.
int	**wolfSSL_GetCipherBlockSize**(WOLFSSL *)Allows caller to determine the negotiated cipher block size from the handshake.
int	**wolfSSL_GetAeadMacSize**(WOLFSSL *)Allows caller to determine the negotiated aead mac size from the handshake. For cipher type WOLFSSL_AEAD_TYPE.
int	**wolfSSL_GetHmacSize**(WOLFSSL *)Allows caller to determine the negotiated (h)mac size from the handshake. For cipher types except WOLFSSL_AEAD_TYPE.
int	**wolfSSL_GetHmacType**(WOLFSSL *)Allows caller to determine the negotiated (h)mac type from the handshake. For cipher types except WOLFSSL_AEAD_TYPE.
int	**wolfSSL_GetCipherType**(WOLFSSL *)Allows caller to determine the negotiated cipher type from the handshake.

	Name
int	**wolfSSL_SetTlsHmacInner**(WOLFSSL * ssl, byte * inner, word32 sz, int content, int verify)Allows caller to set the Hmac Inner vector for message sending/receiving. The result is written to inner which should be at least wolfSSL_GetHmacSize() bytes. The size of the message is specified by sz, content is the type of message, and verify specifies whether this is a verification of a peer message. Valid for cipher types excluding WOLFSSL_AEAD_TYPE.
void	**wolfSSL_CTX_SetEccSignCb**(WOLFSSL_CTX * ctx, CallbackEccSign cb)Allows caller to set the Public Key Callback for ECC Signing. The callback should return 0 for success or < 0 for an error. The ssl and ctx pointers are available for the user's convenience. in is the input buffer to sign while inSz denotes the length of the input. out is the output buffer where the result of the signature should be stored. outSz is an input/output variable that specifies the size of the output buffer upon invocation and the actual size of the signature should be stored there before returning. keyDer is the ECC Private key in ASN1 format and keySz is the length of the key in bytes. An example callback can be found wolfssl/test.h myEccSign().
void	**wolfSSL_SetEccSignCtx**(WOLFSSL * ssl, void * ctx)Allows caller to set the Public Key Ecc Signing Callback Context to ctx.
void *	**wolfSSL_GetEccSignCtx**(WOLFSSL * ssl)Allows caller to retrieve the Public Key Ecc Signing Callback Context previously stored with wolfSSL_SetEccSignCtx().
void	**wolfSSL_CTX_SetEccSignCtx**(WOLFSSL_CTX * ctx, void * userCtx)Allows caller to set the Public Key Ecc Signing Callback Context to ctx.
void *	**wolfSSL_CTX_GetEccSignCtx**(WOLFSSL_CTX * ctx)Allows caller to retrieve the Public Key Ecc Signing Callback Context previously stored with wolfSSL_SetEccSignCtx().

	Name
void	**wolfSSL_CTX_SetEccVerifyCb**(WOLFSSL_CTX * ctx, CallbackEccVerify cb)Allows caller to set the Public Key Callback for ECC Verification. The callback should return 0 for success or < 0 for an error. The ssl and ctx pointers are available for the user's convenience. sig is the signature to verify and sigSz denotes the length of the signature. hash is an input buffer containing the digest of the message and hashSz denotes the length in bytes of the hash. result is an output variable where the result of the verification should be stored, 1 for success and 0 for failure. keyDer is the ECC Private key in ASN1 format and keySz is the length of the key in bytes. An example callback can be found wolfssl/test.h myEccVerify().
void	**wolfSSL_SetEccVerifyCtx**(WOLFSSL * ssl, void * ctx)Allows caller to set the Public Key Ecc Verification Callback Context to ctx.
void *	**wolfSSL_GetEccVerifyCtx**(WOLFSSL * ssl)Allows caller to retrieve the Public Key Ecc Verification Callback Context previously stored with wolfSSL_SetEccVerifyCtx().
void	**wolfSSL_CTX_SetRsaSignCb**(WOLFSSL_CTX * ctx, CallbackRsaSign cb)Allows caller to set the Public Key Callback for RSA Signing. The callback should return 0 for success or < 0 for an error. The ssl and ctx pointers are available for the user's convenience. in is the input buffer to sign while inSz denotes the length of the input. out is the output buffer where the result of the signature should be stored. outSz is an input/output variable that specifies the size of the output buffer upon invocation and the actual size of the signature should be stored there before returning. keyDer is the RSA Private key in ASN1 format and keySz is the length of the key in bytes. An example callback can be found wolfssl/test.h myRsaSign().
void	**wolfSSL_SetRsaSignCtx**(WOLFSSL * ssl, void * ctx)Allows caller to set the Public Key RSA Signing Callback Context to ctx.
void *	**wolfSSL_GetRsaSignCtx**(WOLFSSL * ssl)Allows caller to retrieve the Public Key RSA Signing Callback Context previously stored with wolfSSL_SetRsaSignCtx().

	Name
void	**wolfSSL_CTX_SetRsaVerifyCb**(WOLFSSL_CTX * ctx, CallbackRsaVerify cb)Allows caller to set the Public Key Callback for RSA Verification. The callback should return the number of plaintext bytes for success or < 0 for an error. The ssl and ctx pointers are available for the user's convenience. sig is the signature to verify and sigSz denotes the length of the signature. out should be set to the beginning of the verification buffer after the decryption process and any padding. keyDer is the RSA Public key in ASN1 format and keySz is the length of the key in bytes. An example callback can be found wolfssl/test.h myRsaVerify().
void	**wolfSSL_SetRsaVerifyCtx**(WOLFSSL * ssl, void * ctx)Allows caller to set the Public Key RSA Verification Callback Context to ctx.
void *	**wolfSSL_GetRsaVerifyCtx**(WOLFSSL * ssl)Allows caller to retrieve the Public Key RSA Verification Callback Context previously stored with wolfSSL_SetRsaVerifyCtx().
void	**wolfSSL_CTX_SetRsaEncCb**(WOLFSSL_CTX * ctx, CallbackRsaEnc cb)Allows caller to set the Public Key Callback for RSA Public Encrypt. The callback should return 0 for success or < 0 for an error. The ssl and ctx pointers are available for the user's convenience. in is the input buffer to encrypt while inSz denotes the length of the input. out is the output buffer where the result of the encryption should be stored. outSz is an input/output variable that specifies the size of the output buffer upon invocation and the actual size of the encryption should be stored there before returning. keyDer is the RSA Public key in ASN1 format and keySz is the length of the key in bytes. An example callback can be found wolfssl/test.h myRsaEnc().
void	**wolfSSL_SetRsaEncCtx**(WOLFSSL * ssl, void * ctx)Allows caller to set the Public Key RSA Public Encrypt Callback Context to ctx.
void *	**wolfSSL_GetRsaEncCtx**(WOLFSSL * ssl)Allows caller to retrieve the Public Key RSA Public Encrypt Callback Context previously stored with wolfSSL_SetRsaEncCtx().

COPYRIGHT ©2024 wolfSSL Inc.

	Name
void	**wolfSSL_CTX_SetRsaDecCb**(WOLFSSL_CTX * ctx, CallbackRsaDec cb)Allows caller to set the Public Key Callback for RSA Private Decrypt. The callback should return the number of plaintext bytes for success or < 0 for an error. The ssl and ctx pointers are available for the user's convenience. in is the input buffer to decrypt and inSz denotes the length of the input. out should be set to the beginning of the decryption buffer after the decryption process and any padding. keyDer is the RSA Private key in ASN1 format and keySz is the length of the key in bytes. An example callback can be found wolfssl/test.h myRsaDec().
void	**wolfSSL_SetRsaDecCtx**(WOLFSSL * ssl, void * ctx)Allows caller to set the Public Key RSA Private Decrypt Callback Context to ctx.
void *	**wolfSSL_GetRsaDecCtx**(WOLFSSL * ssl)Allows caller to retrieve the Public Key RSA Private Decrypt Callback Context previously stored with wolfSSL_SetRsaDecCtx().
void	**wolfSSL_CTX_SetCACb**(WOLFSSL_CTX * ctx, CallbackCACache cb)This function registers a callback with the SSL context (WOLFSSL_CTX) to be called when a new CA certificate is loaded into wolfSSL. The callback is given a buffer with the DER-encoded certificate.
WOLFSSL_CERT_MANAGER *	**wolfSSL_CertManagerNew_ex**(void * heap)Allocates and initializes a new Certificate Manager context. This context may be used independent of SSL needs. It may be used to load certificates, verify certificates, and check the revocation status.
WOLFSSL_CERT_MANAGER *	**wolfSSL_CertManagerNew**(void)Allocates and initializes a new Certificate Manager context. This context may be used independent of SSL needs. It may be used to load certificates, verify certificates, and check the revocation status.
void	**wolfSSL_CertManagerFree**(WOLFSSL_CERT_MANAGER *)Frees all resources associated with the Certificate Manager context. Call this when you no longer need to use the Certificate Manager.
int	**wolfSSL_CertManagerLoadCA**(WOLFSSL_CERT_MANAGER * cm, const char * f, const char * d)Specifies the locations for CA certificate loading into the manager context. The PEM certificate CAfile may contain several trusted CA certificates. If CApath is not NULL it specifies a directory containing CA certificates in PEM format.

	Name
int	**wolfSSL_CertManagerLoadCABuffer**(WOLFSSL_CERT_MANAGER * cm, const unsigned char * in, long sz, int format)Loads the CA Buffer by calling wolfSSL_CTX_load_verify_buffer and returning that result using a temporary cm so as not to lose the information in the cm passed into the function.
int	**wolfSSL_CertManagerUnloadCAs**(WOLFSSL_CERT_MANAGER * cm)This function unloads the CA signer list.
int	**wolfSSL_CertManagerUnloadIntermediateCerts**(WOLFSSL_CE * cm)This function unloads intermediate certificates add to the CA signer list.
int	**wolfSSL_CertManagerUnload_trust_peers**(WOLFSSL_CERT_MA * cm)The function will free the Trusted Peer linked list and unlocks the trusted peer list.
int	**wolfSSL_CertManagerVerify**(WOLFSSL_CERT_MANAGER * cm, const char * f, int format)Specifies the certificate to verify with the Certificate Manager context. The format can be SSL_FILETYPE_PEM or SSL_FILETYPE_ASN1.
int	**wolfSSL_CertManagerVerifyBuffer**(WOLFSSL_CERT_MANAGER * cm, const unsigned char * buff, long sz, int format)Specifies the certificate buffer to verify with the Certificate Manager context. The format can be SSL_FILETYPE_PEM or SSL_FILETYPE_ASN1.
void	**wolfSSL_CertManagerSetVerify**(WOLFSSL_CERT_MANAGER * cm, VerifyCallback vc)The function sets the verifyCallback function in the Certificate Manager. If present, it will be called for each cert loaded. If there is a verification error, the verify callback can be used to over-ride the error.
int	**wolfSSL_CertManagerCheckCRL**(WOLFSSL_CERT_MANAGER * cm, unsigned char * der, int sz)Check CRL if the option is enabled and compares the cert to the CRL list.
int	**wolfSSL_CertManagerEnableCRL**(WOLFSSL_CERT_MANAGER * cm, int options)Turns on Certificate Revocation List checking when verifying certificates with the Certificate Manager. By default, CRL checking is off. options include WOLFSSL_CRL_CHECKALL which performs CRL checking on each certificate in the chain versus the Leaf certificate only which is the default.

	Name
int	**wolfSSL_CertManagerDisableCRL**(WOLFSSL_CERT_MANAGER *)Turns off Certificate Revocation List checking when verifying certificates with the Certificate Manager. By default, CRL checking is off. You can use this function to temporarily or permanently disable CRL checking with this Certificate Manager context that previously had CRL checking enabled.
int	**wolfSSL_CertManagerLoadCRL**(WOLFSSL_CERT_MANAGER * cm, const char * path, int type, int monitor)Error checks and passes through to LoadCRL() in order to load the cert into the CRL for revocation checking. An updated CRL can be loaded by first calling wolfSSL_CertManagerFreeCRL, then loading the new CRL.
int	**wolfSSL_CertManagerLoadCRLBuffer**(WOLFSSL_CERT_MANAGE * cm, const unsigned char * buff, long sz, int type)The function loads the CRL file by calling BufferLoadCRL.
int	**wolfSSL_CertManagerSetCRL_Cb**(WOLFSSL_CERT_MANAGER * cm, CbMissingCRL cb)This function sets the CRL Certificate Manager callback. If HAVE_CRL is defined and a matching CRL record is not found then the cbMissingCRL is called (set via wolfSSL_CertManagerSetCRL_Cb). This allows you to externally retrieve the CRL and load it.
int	**wolfSSL_CertManagerSetCRLUpdate_Cb**(WOLFSSL_CERT_MANA * cm, CbUpdateCRL cb)This function sets the CRL Update callback. If HAVE_CRL and HAVE_CRL_UPDATE_CB is defined , and an entry with the same issuer and a lower CRL number exists when a CRL is added, then the CbUpdateCRL is called with the details of the existing entry and the new one replacing it.
int	**wolfSSL_CertManagerGetCRLInfo**(WOLFSSL_CERT_MANAGER * cm, CrlInfo * info, const byte * buff, long sz, int type)This function yields a structure with parsed CRL information from an encoded CRL buffer.
int	**wolfSSL_CertManagerCheckOCSP**(WOLFSSL_CERT_MANAGER * cm, unsigned char * der, int sz)The function enables the WOLFSSL_CERT_MANAGER's member, ocspEnabled to signify that the OCSP check option is enabled.
int	**wolfSSL_CertManagerEnableOCSP**(WOLFSSL_CERT_MANAGER * cm, int options)Turns on OCSP if it's turned off and if compiled with the set option available.
int	**wolfSSL_CertManagerDisableOCSP**(WOLFSSL_CERT_MANAGER *)Disables OCSP certificate revocation.

	Name
int	**wolfSSL_CertManagerSetOCSPOverrideURL**(WOLFSSL_CERT_M * cm, const char * url)The function copies the url to the ocspOverrideURL member of the WOLFSSL_CERT_MANAGER structure.
int	**wolfSSL_CertManagerSetOCSP_Cb**(WOLFSSL_CERT_MANAGER * cm, CbOCSPIO ioCb, CbOCSPRespFree respFreeCb, void * ioCbCtx)The function sets the OCSP callback in the WOLFSSL_CERT_MANAGER.
int	**wolfSSL_CertManagerEnableOCSPStapling**(WOLFSSL_CERT_MA * cm)This function turns on OCSP stapling if it is not turned on as well as set the options.
int	**wolfSSL_EnableCRL**(WOLFSSL * ssl, int options)Enables CRL certificate revocation.
int	**wolfSSL_DisableCRL**(WOLFSSL * ssl)Disables CRL certificate revocation.
int	**wolfSSL_LoadCRL**(WOLFSSL * ssl, const char * path, int type, int monitor)A wrapper function that ends up calling LoadCRL to load the certificate for revocation checking.
int	**wolfSSL_SetCRL_Cb**(WOLFSSL * ssl, CbMissingCRL cb)Sets the CRL callback in the WOLFSSL_CERT_MANAGER structure.
int	**wolfSSL_EnableOCSP**(WOLFSSL * ssl, int options)This function enables OCSP certificate verification. The value of options if formed by or'ing one or more of the following options: WOLFSSL_OCSP_URL_OVERRIDE _ use the override URL instead of the URL in certificates. The override URL is specified using the wolfSSL_CTX_SetOCSP_OverrideURL() function. WOLFSSL_OCSP_CHECKALL - Set all OCSP checks on WOLFSSL_OCSP_NO_NONCE - Set nonce option for creating OCSP requests.
int	**wolfSSL_DisableOCSP**(WOLFSSL *)Disables the OCSP certificate revocation option.
int	**wolfSSL_SetOCSP_OverrideURL**(WOLFSSL * ssl, const char * url)This function sets the ocspOverrideURL member in the WOLFSSL_CERT_MANAGER structure.
int	**wolfSSL_SetOCSP_Cb**(WOLFSSL * ssl, CbOCSPIO ioCb, CbOCSPRespFree respFreeCb, void * ioCbCtx)This function sets the OCSP callback in the WOLFSSL_CERT_MANAGER structure.
int	**wolfSSL_CTX_EnableCRL**(WOLFSSL_CTX * ctx, int options)Enables CRL certificate verification through the CTX.
int	**wolfSSL_CTX_DisableCRL**(WOLFSSL_CTX * ctx)This function disables CRL verification in the CTX structure.

	Name
int	**wolfSSL_CTX_LoadCRL**(WOLFSSL_CTX * ctx, const char * path, int type, int monitor)This function loads CRL into the WOLFSSL_CTX structure through wolfSSL_CertManagerLoadCRL().
int	**wolfSSL_CTX_SetCRL_Cb**(WOLFSSL_CTX * ctx, CbMissingCRL cb)This function will set the callback argument to the cbMissingCRL member of the WOLFSSL_CERT_MANAGER structure by calling wolfSSL_CertManagerSetCRL_Cb.
int	**wolfSSL_CTX_EnableOCSP**(WOLFSSL_CTX * ctx, int options)This function sets options to configure behavior of OCSP functionality in wolfSSL. The value of options if formed by or'ing one or more of the following options: WOLFSSL_OCSP_URL_OVERRIDE _ use the override URL instead of the URL in certificates. The override URL is specified using the wolfSSL_CTX_SetOCSP_OverrideURL() function. WOLFSSL_OCSP_CHECKALL - Set all OCSP checks on WOLFSSL_OCSP_NO_NONCE - Set nonce option for creating OCSP requests.
int	**wolfSSL_CTX_DisableOCSP**(WOLFSSL_CTX *)This function disables OCSP certificate revocation checking by affecting the ocspEnabled member of the WOLFSSL_CERT_MANAGER structure.
int	**wolfSSL_CTX_SetOCSP_OverrideURL**(WOLFSSL_CTX * ctx, const char * url)This function manually sets the URL for OCSP to use. By default, OCSP will use the URL found in the individual certificate unless the WOLFSSL_OCSP_URL_OVERRIDE option is set using the wolfSSL_CTX_EnableOCSP.
int	**wolfSSL_CTX_SetOCSP_Cb**(WOLFSSL_CTX * ctx, CbOCSPIO ioCb, CbOCSPRespFree respFreeCb, void * ioCbCtx)Sets the callback for the OCSP in the WOLFSSL_CTX structure.
int	**wolfSSL_CTX_EnableOCSPStapling**(WOLFSSL_CTX *)This function enables OCSP stapling by calling wolfSSL_CertManagerEnableOCSPStapling().
void	**wolfSSL_KeepArrays may be called to free the resources immediately, or alternatively the resources will be freed when the associated SSL object is freed.

	Name
void	**wolfSSL_FreeArrays has been called before the handshake, wolfSSL will not free temporary arrays. This function explicitly frees temporary arrays and should be called when the user is done with temporary arrays and does not want to wait for the SSL object to be freed to free these resources.
int	**wolfSSL_UseSNI**(WOLFSSL * ssl, unsigned char type, const void * data, unsigned short size)This function enables the use of Server Name Indication in the SSL object passed in the 'ssl' parameter. It means that the SNI extension will be sent on ClientHello by wolfSSL client and wolfSSL server will respond ClientHello + SNI with either ServerHello + blank SNI or alert fatal in case of SNI mismatch.
int	**wolfSSL_CTX_UseSNI**(WOLFSSL_CTX * ctx, unsigned char type, const void * data, unsigned short size)This function enables the use of Server Name Indication for SSL objects created from the SSL context passed in the 'ctx' parameter. It means that the SNI extension will be sent on ClientHello by wolfSSL clients and wolfSSL servers will respond ClientHello + SNI with either ServerHello + blank SNI or alert fatal in case of SNI mismatch.
void	**wolfSSL_SNI_SetOptions**(WOLFSSL * ssl, unsigned char type, unsigned char options)This function is called on the server side to configure the behavior of the SSL session using Server Name Indication in the SSL object passed in the 'ssl' parameter. The options are explained below.
void	**wolfSSL_CTX_SNI_SetOptions**(WOLFSSL_CTX * ctx, unsigned char type, unsigned char options)This function is called on the server side to configure the behavior of the SSL sessions using Server Name Indication for SSL objects created from the SSL context passed in the 'ctx' parameter. The options are explained below.
int	**wolfSSL_SNI_GetFromBuffer**(const unsigned char * clientHello, unsigned int helloSz, unsigned char type, unsigned char * sni, unsigned int * inOutSz)This function is called on the server side to retrieve the Server Name Indication provided by the client from the Client Hello message sent by the client to start a session. It does not requires context or session setup to retrieve the SNI.

	Name
unsigned char	**wolfSSL_SNI_Status**(WOLFSSL * ssl, unsigned char type)This function gets the status of an SNI object.
unsigned short	**wolfSSL_SNI_GetRequest**(WOLFSSL * ssl, unsigned char type, void ** data)This function is called on the server side to retrieve the Server Name Indication provided by the client in a SSL session.
int	**wolfSSL_UseALPN**(WOLFSSL * ssl, char * protocol_name_list, unsigned int protocol_name_listSz, unsigned char options)Setup ALPN use for a wolfSSL session.
int	**wolfSSL_ALPN_GetProtocol**(WOLFSSL * ssl, char ** protocol_name, unsigned short * size)This function gets the protocol name set by the server.
int	**wolfSSL_ALPN_GetPeerProtocol**(WOLFSSL * ssl, char ** list, unsigned short * listSz)This function copies the alpn_client_list data from the SSL object to the buffer.
int	**wolfSSL_UseMaxFragment**(WOLFSSL * ssl, unsigned char mfl)This function is called on the client side to enable the use of Maximum Fragment Length in the SSL object passed in the 'ssl' parameter. It means that the Maximum Fragment Length extension will be sent on ClientHello by wolfSSL clients.
int	**wolfSSL_CTX_UseMaxFragment**(WOLFSSL_CTX * ctx, unsigned char mfl)This function is called on the client side to enable the use of Maximum Fragment Length for SSL objects created from the SSL context passed in the 'ctx' parameter. It means that the Maximum Fragment Length extension will be sent on ClientHello by wolfSSL clients.
int	**wolfSSL_UseTruncatedHMAC**(WOLFSSL * ssl)This function is called on the client side to enable the use of Truncated HMAC in the SSL object passed in the 'ssl' parameter. It means that the Truncated HMAC extension will be sent on ClientHello by wolfSSL clients.
int	**wolfSSL_CTX_UseTruncatedHMAC**(WOLFSSL_CTX * ctx)This function is called on the client side to enable the use of Truncated HMAC for SSL objects created from the SSL context passed in the 'ctx' parameter. It means that the Truncated HMAC extension will be sent on ClientHello by wolfSSL clients.

	Name
int	**wolfSSL_UseOCSPStapling**(WOLFSSL * ssl, unsigned char status_type, unsigned char options)Stapling eliminates the need to contact the CA. Stapling lowers the cost of certificate revocation check presented in OCSP.
int	**wolfSSL_CTX_UseOCSPStapling**(WOLFSSL_CTX * ctx, unsigned char status_type, unsigned char options)This function requests the certificate status during the handshake.
int	**wolfSSL_UseOCSPStaplingV2**(WOLFSSL * ssl, unsigned char status_type, unsigned char options)The function sets the status type and options for OCSP.
int	**wolfSSL_CTX_UseOCSPStaplingV2**(WOLFSSL_CTX * ctx, unsigned char status_type, unsigned char options)Creates and initializes the certificate status request for OCSP Stapling.
int	**wolfSSL_UseSupportedCurve**(WOLFSSL * ssl, word16 name)This function is called on the client side to enable the use of Supported Elliptic Curves Extension in the SSL object passed in the 'ssl' parameter. It means that the supported curves enabled will be sent on ClientHello by wolfSSL clients. This function can be called more than one time to enable multiple curves.
int	**wolfSSL_CTX_UseSupportedCurve**(WOLFSSL_CTX * ctx, word16 name)This function is called on the client side to enable the use of Supported Elliptic Curves Extension for SSL objects created from the SSL context passed in the 'ctx' parameter. It means that the supported curves enabled will be sent on ClientHello by wolfSSL clients. This function can be called more than one time to enable multiple curves.
int	**wolfSSL_UseSecureRenegotiation**(WOLFSSL * ssl)This function forces secure renegotiation for the supplied WOLFSSL structure. This is not recommended.
int	**wolfSSL_Rehandshake**(WOLFSSL * ssl)This function executes a secure renegotiation handshake; this is user forced as wolfSSL discourages this functionality.
int	**wolfSSL_UseSessionTicket**(WOLFSSL * ssl)Force provided WOLFSSL structure to use session ticket. The constant HAVE_SESSION_TICKET should be defined and the constant NO_WOLFSSL_CLIENT should not be defined to use this function.
int	**wolfSSL_CTX_UseSessionTicket**(WOLFSSL_CTX * ctx)This function sets wolfSSL context to use a session ticket.

	Name
int	**wolfSSL_get_SessionTicket**(WOLFSSL * ssl, unsigned char * buf, word32 * bufSz)This function copies the ticket member of the Session structure to the buffer.
int	**wolfSSL_set_SessionTicket**(WOLFSSL * ssl, const unsigned char * buf, word32 bufSz)This function sets the ticket member of the WOLFSSL_SESSION structure within the WOLFSSL struct. The buffer passed into the function is copied to memory.
int	**wolfSSL_set_SessionTicket_cb**(WOLFSSL * ssl, CallbackSessionTicket cb, void * ctx)This function sets the session ticket callback. The type CallbackSessionTicket is a function pointer with the signature of: int (*CallbackSessionTicket)(WOLFSSL*, const unsigned char*, int, void*)
int	**wolfSSL_send_SessionTicket**(WOLFSSL * ssl)This function sends a session ticket to the client after a TLS v1.3 handhsake has been established.
int	**wolfSSL_CTX_set_TicketEncCb**(WOLFSSL_CTX * ctx, SessionTicketEncCb)This function sets the session ticket key encrypt callback function for a server to support session tickets as specified in RFC 5077.
int	**wolfSSL_CTX_set_TicketHint**(WOLFSSL_CTX * ctx, int)This function sets the session ticket hint relayed to the client. For server side use.
int	**wolfSSL_CTX_set_TicketEncCtx**(WOLFSSL_CTX * ctx, void *)This function sets the session ticket encrypt user context for the callback. For server side use.
void *	**wolfSSL_CTX_get_TicketEncCtx**(WOLFSSL_CTX * ctx)This function gets the session ticket encrypt user context for the callback. For server side use.
int	**wolfSSL_SetHsDoneCb**(WOLFSSL * ssl, HandShakeDoneCb cb, void * user_ctx)This function sets the handshake done callback. The hsDoneCb and hsDoneCtx members of the WOLFSSL structure are set in this function.
int	**wolfSSL_PrintSessionStats**(void)This function prints the statistics from the session.
int	**wolfSSL_get_session_stats**(unsigned int * active, unsigned int * total, unsigned int * peak, unsigned int * maxSessions)This function gets the statistics for the session.

COPYRIGHT ©2024 wolfSSL Inc.

	Name
int	**wolfSSL_MakeTlsMasterSecret**(unsigned char * ms, word32 msLen, const unsigned char * pms, word32 pmsLen, const unsigned char * cr, const unsigned char * sr, int tls1_2, int hash_type)This function copies the values of cr and sr then passes through to wc_PRF (pseudo random function) and returns that value.
int	**wolfSSL_DeriveTlsKeys**(unsigned char * key_data, word32 keyLen, const unsigned char * ms, word32 msLen, const unsigned char * sr, const unsigned char * cr, int tls1_2, int hash_type)An external facing wrapper to derive TLS Keys.
int	**wolfSSL_connect_ex**(WOLFSSL * ssl, HandShakeCallBack hsCb, TimeoutCallBack toCb, WOLFSSL_TIMEVAL timeout)wolfSSL_connect_ex() is an extension that allows a HandShake Callback to be set. This can be useful in embedded systems for debugging support when a debugger isn't available and sniffing is impractical. The HandShake Callback will be called whether or not a handshake error occurred. No dynamic memory is used since the maximum number of SSL packets is known. Packet names can be accessed through packetNames[]. The connect extension also allows a Timeout Callback to be set along with a timeout value. This is useful if the user doesn't want to wait for the TCP stack to timeout. This extension can be called with either, both, or neither callbacks.
int	**wolfSSL_accept_ex**(WOLFSSL * ssl, HandShakeCallBacki hsCb, TimeoutCallBack toCb, WOLFSSL_TIMEVAL timeout)wolfSSL_accept_ex() is an extension that allows a HandShake Callback to be set. This can be useful in embedded systems for debugging support when a debugger isn't available and sniffing is impractical. The HandShake Callback will be called whether or not a handshake error occurred. No dynamic memory is used since the maximum number of SSL packets is known. Packet names can be accessed through packetNames[]. The connect extension also allows a Timeout Callback to be set along with a timeout value. This is useful if the user doesn't want to wait for the TCP stack to timeout. This extension can be called with either, both, or neither callbacks.
long	**wolfSSL_BIO_set_fp**(WOLFSSL_BIO * bio, XFILE fp, int c)This is used to set the internal file pointer for a BIO.

	Name
long	**wolfSSL_BIO_get_fp**(WOLFSSL_BIO * bio, XFILE * fp)This is used to get the internal file pointer for a BIO.
int	**wolfSSL_check_private_key**(const WOLFSSL * ssl)This function checks that the private key is a match with the certificate being used.
int	**wolfSSL_X509_get_ext_by_NID**(const WOLFSSL_X509 * x509, int nid, int lastPos)This function looks for and returns the extension index matching the passed in NID value.
void *	**wolfSSL_X509_get_ext_d2i**(const WOLFSSL_X509 * x509, int nid, int * c, int * idx)This function looks for and returns the extension matching the passed in NID value.
int	**wolfSSL_X509_digest**(const WOLFSSL_X509 * x509, const WOLFSSL_EVP_MD * digest, unsigned char * buf, unsigned int * len)This function returns the hash of the DER certificate.
int	**wolfSSL_use_certificate**(WOLFSSL * ssl, WOLFSSL_X509 * x509)his is used to set the certificate for WOLFSSL structure to use during a handshake.
int	**wolfSSL_use_certificate_ASN1**(WOLFSSL * ssl, unsigned char * der, int derSz)This is used to set the certificate for WOLFSSL structure to use during a handshake. A DER formatted buffer is expected.
int	**wolfSSL_use_PrivateKey**(WOLFSSL * ssl, WOLFSSL_EVP_PKEY * pkey)This is used to set the private key for the WOLFSSL structure.
int	**wolfSSL_use_PrivateKey_ASN1**(int pri, WOLFSSL * ssl, unsigned char * der, long derSz)This is used to set the private key for the WOLFSSL structure. A DER formatted key buffer is expected.
int	**wolfSSL_use_RSAPrivateKey_ASN1**(WOLFSSL * ssl, unsigned char * der, long derSz)This is used to set the private key for the WOLFSSL structure. A DER formatted RSA key buffer is expected.
WOLFSSL_DH *	**wolfSSL_DSA_dup_DH**(const WOLFSSL_DSA * r)This function duplicates the parameters in dsa to a newly created WOLFSSL_DH structure.
int	**wolfSSL_SESSION_get_master_key**(const WOLFSSL_SESSION * ses, unsigned char * out, int outSz)This is used to get the master key after completing a handshake.
int	**wolfSSL_SESSION_get_master_key_length**(const WOLFSSL_SESSION * ses)This is used to get the master secret key length.

	Name
void	**wolfSSL_CTX_set_cert_store**(WOLFSSL_CTX * ctx, WOLFSSL_X509_STORE * str)This is a setter function for the WOLFSSL_X509_STORE structure in ctx.
WOLFSSL_X509 *	**wolfSSL_d2i_X509_bio**(WOLFSSL_BIO * bio, WOLFSSL_X509 ** x509)This function get the DER buffer from bio and converts it to a WOLFSSL_X509 structure.
WOLFSSL_X509_STORE *	**wolfSSL_CTX_get_cert_store**(WOLFSSL_CTX * ctx)This is a getter function for the WOLFSSL_X509_STORE structure in ctx.
size_t	**wolfSSL_BIO_ctrl_pending**(WOLFSSL_BIO * b)Gets the number of pending bytes to read. If BIO type is BIO_BIO then is the number to read from pair. If BIO contains an SSL object then is pending data from SSL object (wolfSSL_pending(ssl)). If is BIO_MEMORY type then returns the size of memory buffer.
size_t	**wolfSSL_get_server_random**(const WOLFSSL * ssl, unsigned char * out, size_t outlen)This is used to get the random data sent by the server during the handshake.
size_t	**wolfSSL_get_client_random**(const WOLFSSL * ssl, unsigned char * out, size_t outSz)This is used to get the random data sent by the client during the handshake.
wc_pem_password_cb *	**wolfSSL_CTX_get_default_passwd_cb**(WOLFSSL_CTX * ctx)This is a getter function for the password callback set in ctx.
void *	**wolfSSL_CTX_get_default_passwd_cb_userdata**(WOLFSSL_CTX * ctx)This is a getter function for the password callback user data set in ctx.
WOLFSSL_X509 *	**wolfSSL_PEM_read_bio_X509_AUX**(WOLFSSL_BIO * bp, WOLFSSL_X509 ** x, wc_pem_password_cb * cb, void * u)This function behaves the same as wolfSSL_PEM_read_bio_X509. AUX signifies containing extra information such as trusted/rejected use cases and friendly name for human readability.
long	**wolfSSL_CTX_set_tmp_dh**(WOLFSSL_CTX * ctx, WOLFSSL_DH * dh)Initializes the WOLFSSL_CTX structure's dh member with the Diffie-Hellman parameters.
WOLFSSL_DSA *	**wolfSSL_PEM_read_bio_DSAparams**(WOLFSSL_BIO * bp, WOLFSSL_DSA ** x, wc_pem_password_cb * cb, void * u)This function get the DSA parameters from a PEM buffer in bio.
unsigned long	**wolfSSL_ERR_peek_last_error**(void)This function returns the absolute value of the last error from WOLFSSL_ERROR encountered.

	Name
	WOLF_STACK_OF(WOLFSSL_X509) constThis function gets the peer's certificate chain.
long	**wolfSSL_CTX_clear_options**(WOLFSSL_CTX * ctx, long opt)This function resets option bits of WOLFSSL_CTX object.
int	**wolfSSL_set_jobject**(WOLFSSL * ssl, void * objPtr)This function sets the jObjectRef member of the WOLFSSL structure.
void *	**wolfSSL_get_jobject**(WOLFSSL * ssl)This function returns the jObjectRef member of the WOLFSSL structure.
int	**wolfSSL_set_msg_callback**(WOLFSSL * ssl, SSL_Msg_Cb cb)This function sets a callback in the ssl. The callback is to observe handshake messages. NULL value of cb resets the callback.
int	**wolfSSL_set_msg_callback_arg**(WOLFSSL * ssl, void * arg)This function sets associated callback context value in the ssl. The value is handed over to the callback argument.
char *	**wolfSSL_X509_get_next_altname**(WOLFSSL_X509 *)This function returns the next, if any, altname from the peer certificate.
WOLFSSL_ASN1_TIME *	**wolfSSL_X509_get_notBefore**(WOLFSSL_X509 *)The function checks to see if x509 is NULL and if it's not, it returns the notBefore member of the x509 struct.
int	**wolfSSL_connect will only return once the handshake has been finished or an error occurred. wolfSSL takes a different approach to certificate verification than OpenSSL does. The default policy for the client is to verify the server, this means that if you don't load CAs to verify the server you'll get a connect error, unable to verify (_155). It you want to mimic OpenSSL behavior of having SSL_connect succeed even if verifying the server fails and reducing security you can do this by calling: SSL_CTX_set_verify(ctx, SSL_VERIFY_NONE, 0); before calling SSL_new(); Though it's not recommended.

	Name
int	**wolfSSL_send_hrr_cookie**(WOLFSSL * ssl, const unsigned char * secret, unsigned int secretSz)This function is called on the server side to indicate that a HelloRetryRequest message must contain a Cookie and, in case of using protocol DTLS v1.3, that the handshake will always include a cookie exchange. Please note that when using protocol DTLS v1.3, the cookie exchange is enabled by default. The Cookie holds a hash of the current transcript so that another server process can handle the ClientHello in reply. The secret is used when generating the integrity check on the Cookie data.
int	**wolfSSL_disable_hrr_cookie**(WOLFSSL * ssl)This function is called on the server side to indicate that a HelloRetryRequest message must NOT contain a Cookie and that, if using protocol DTLS v1.3, a cookie exchange will not be included in the handshake. Please note that not doing a cookie exchange when using protocol DTLS v1.3 can make the server susceptible to DoS/Amplification attacks.
int	**wolfSSL_CTX_no_ticket_TLSv13**(WOLFSSL_CTX * ctx)This function is called on the server to stop it from sending a resumption session ticket once the handshake is complete.
int	**wolfSSL_no_ticket_TLSv13**(WOLFSSL * ssl)This function is called on the server to stop it from sending a resumption session ticket once the handshake is complete.
int	**wolfSSL_CTX_no_dhe_psk**(WOLFSSL_CTX * ctx)This function is called on a TLS v1.3 wolfSSL context to disallow Diffie_Hellman (DH) style key exchanges when handshakes are using pre-shared keys for authentication.
int	**wolfSSL_no_dhe_psk**(WOLFSSL * ssl)This function is called on a TLS v1.3 client or server wolfSSL to disallow Diffie_Hellman (DH) style key exchanges when handshakes are using pre-shared keys for authentication.
int	**wolfSSL_update_keys**(WOLFSSL * ssl)This function is called on a TLS v1.3 client or server wolfSSL to force the rollover of keys. A KeyUpdate message is sent to the peer and new keys are calculated for encryption. The peer will send back a KeyUpdate message and the new decryption keys will then be calculated. This function can only be called after a handshake has been completed.

	Name
int	**wolfSSL_key_update_response is called, a KeyUpdate message is sent and the encryption key is updated. The decryption key is updated when the response is received.
int	**wolfSSL_CTX_allow_post_handshake_auth**(WOLFSSL_CTX * ctx)This function is called on a TLS v1.3 client wolfSSL context to allow a client certificate to be sent post handshake upon request from server. This is useful when connecting to a web server that has some pages that require client authentication and others that don't.
int	**wolfSSL_allow_post_handshake_auth**(WOLFSSL * ssl)This function is called on a TLS v1.3 client wolfSSL to allow a client certificate to be sent post handshake upon request from server. A Post-Handshake Client Authentication extension is sent in the ClientHello. This is useful when connecting to a web server that has some pages that require client authentication and others that don't.
int	**wolfSSL_request_certificate**(WOLFSSL * ssl)This function requests a client certificate from the TLS v1.3 client. This is useful when a web server is serving some pages that require client authentication and others that don't. A maximum of 256 requests can be sent on a connection.
int	**wolfSSL_CTX_set1_groups_list**(WOLFSSL_CTX * ctx, char * list)This function sets the list of elliptic curve groups to allow on a wolfSSL context in order of preference. The list is a null-terminated text string, and a colon-delimited list. Call this function to set the key exchange elliptic curve parameters to use with the TLS v1.3 connections.
int	**wolfSSL_set1_groups_list**(WOLFSSL * ssl, char * list)This function sets the list of elliptic curve groups to allow on a wolfSSL in order of preference. The list is a null-terminated text string, and a colon-delimited list. Call this function to set the key exchange elliptic curve parameters to use with the TLS v1.3 connections.
int	**wolfSSL_preferred_group**(WOLFSSL * ssl)This function returns the key exchange group the client prefers to use in the TLS v1.3 handshake. Call this function to after a handshake is complete to determine which group the server prefers so that this information can be used in future connections to pre-generate a key pair for key exchange.

	Name
int	**wolfSSL_CTX_set_groups**(WOLFSSL_CTX * ctx, int * groups, int count)This function sets the list of elliptic curve groups to allow on a wolfSSL context in order of preference. The list is an array of group identifiers with the number of identifiers specified in count. Call this function to set the key exchange elliptic curve parameters to use with the TLS v1.3 connections.
int	**wolfSSL_set_groups**(WOLFSSL * ssl, int * groups, int count)This function sets the list of elliptic curve groups to allow on a wolfSSL. The list is an array of group identifiers with the number of identifiers specified in count. Call this function to set the key exchange elliptic curve parameters to use with the TLS v1.3 connections.
int	**wolfSSL_connect_TLSv13 will only return once the handshake has been finished or an error occurred. wolfSSL takes a different approach to certificate verification than OpenSSL does. The default policy for the client is to verify the server, this means that if you don't load CAs to verify the server you'll get a connect error, unable to verify (_155). It you want to mimic OpenSSL behavior of having SSL_connect succeed even if verifying the server fails and reducing security you can do this by calling: SSL_CTX_set_verify(ctx, SSL_VERIFY_NONE, 0); before calling SSL_new(); Though it's not recommended. **wolfSSL_accept_TLSv13 will only return once the handshake has been finished or an error occurred. Call this function when expecting a TLS v1.3 connection though older version ClientHello messages are supported.

	Name
int	**wolfSSL_CTX_set_max_early_data**(WOLFSSL_CTX * ctx, unsigned int sz)This function sets the maximum amount of early data that a TLS v1.3 client or server is willing to exchange using the wolfSSL context. Call this function to limit the amount of early data to process to mitigate replay attacks. Early data is protected by keys derived from those of the connection that the session ticket was sent and therefore will be the same every time a session ticket is used in resumption. The value is included in the session ticket for resumption. A server value of zero indicates no early data is to be sent by client using session tickets. A client value of zero indicates that the client will not send any early data. It is recommended that the number of early data bytes be kept as low as practically possible in the application.
int	**wolfSSL_set_max_early_data**(WOLFSSL * ssl, unsigned int sz)This function sets the maximum amount of early data that a TLS v1.3 client or server is willing to exchange. Call this function to limit the amount of early data to process to mitigate replay attacks. Early data is protected by keys derived from those of the connection that the session ticket was sent and therefore will be the same every time a session ticket is used in resumption. The value is included in the session ticket for resumption. A server value of zero indicates no early data is to be sent by client using session tickets. A client value of zero indicates that the client will not send any early data. It is recommended that the number of early data bytes be kept as low as practically possible in the application.
int	**wolfSSL_write_early_data to connect to the server and send the data in the handshake. This function is only used with clients.
int	**wolfSSL_read_early_data returns true. Early data may be sent by the client in multiple messages. If there is no early data then the handshake will be processed as normal. This function is only used with servers.
int	**wolfSSL_inject to extract the plaintext data from the WOLFSSL object.

	Name
void	**wolfSSL_CTX_set_psk_client_tls13_callback**(WOLFSSL_CTX * ctx, wc_psk_client_tls13_callback cb)This function sets the Pre_Shared Key (PSK) client side callback for TLS v1.3 connections. The callback is used to find a PSK identity and return its key and the name of the cipher to use for the handshake. The function sets the client_psk_tls13_cb member of the WOLFSSL_CTX structure.
void	**wolfSSL_set_psk_client_tls13_callback**(WOLFSSL * ssl, wc_psk_client_tls13_callback cb)This function sets the Pre_Shared Key (PSK) client side callback for TLS v1.3 connections. The callback is used to find a PSK identity and return its key and the name of the cipher to use for the handshake. The function sets the client_psk_tls13_cb member of the options field in WOLFSSL structure.
void	**wolfSSL_CTX_set_psk_server_tls13_callback**(WOLFSSL_CTX * ctx, wc_psk_server_tls13_callback cb)This function sets the Pre_Shared Key (PSK) server side callback for TLS v1.3 connections. The callback is used to find a PSK identity and return its key and the name of the cipher to use for the handshake. The function sets the server_psk_tls13_cb member of the WOLFSSL_CTX structure.
void	**wolfSSL_set_psk_server_tls13_callback**(WOLFSSL * ssl, wc_psk_server_tls13_callback cb)This function sets the Pre_Shared Key (PSK) server side callback for TLS v1.3 connections. The callback is used to find a PSK identity and return its key and the name of the cipher to use for the handshake. The function sets the server_psk_tls13_cb member of the options field in WOLFSSL structure.
int	**wolfSSL_UseKeyShare**(WOLFSSL * ssl, word16 group)This function creates a key share entry from the group including generating a key pair. The KeyShare extension contains all the generated public keys for key exchange. If this function is called, then only the groups specified will be included. Call this function when a preferred group has been previously established for the server.

	Name
int	**wolfSSL_NoKeyShares**(WOLFSSL * ssl)This function is called to ensure no key shares are sent in the ClientHello. This will force the server to respond with a HelloRetryRequest if a key exchange is required in the handshake. Call this function when the expected key exchange group is not known and to avoid the generation of keys unnecessarily. Note that an extra round-trip will be required to complete the handshake when a key exchange is required.
WOLFSSL_METHOD *	**wolfTLSv1_3_server_method_ex.
WOLFSSL_METHOD *	**wolfTLSv1_3_client_method_ex.
WOLFSSL_METHOD *	**wolfTLSv1_3_server_method.
WOLFSSL_METHOD *	**wolfTLSv1_3_client_method.
WOLFSSL_METHOD *	**wolfTLSv1_3_method_ex**(void * heap)This function returns a WOLFSSL_METHOD similar to wolfTLSv1_3_client_method except that it is not determined which side yet (server/client).
WOLFSSL_METHOD *	**wolfTLSv1_3_method**(void)This function returns a WOLFSSL_METHOD similar to wolfTLSv1_3_client_method except that it is not determined which side yet (server/client).
int	**wolfSSL_CTX_set_ephemeral_key**(WOLFSSL_CTX * ctx, int keyAlgo, const char * key, unsigned int keySz, int format)This function sets a fixed / static ephemeral key for testing only.
int	**wolfSSL_set_ephemeral_key**(WOLFSSL * ssl, int keyAlgo, const char * key, unsigned int keySz, int format)This function sets a fixed / static ephemeral key for testing only.
int	**wolfSSL_CTX_get_ephemeral_key**(WOLFSSL_CTX * ctx, int keyAlgo, const unsigned char ** key, unsigned int * keySz)This function returns pointer to loaded key as ASN.1/DER.
int	**wolfSSL_get_ephemeral_key**(WOLFSSL * ssl, int keyAlgo, const unsigned char ** key, unsigned int * keySz)This function returns pointer to loaded key as ASN.1/DER.
int	**wolfSSL_RSA_sign_generic_padding**(int type, const unsigned char * m, unsigned int mLen, unsigned char * sigRet, unsigned int * sigLen, WOLFSSL_RSA * rsa, int flag, int padding)Sign a message with the chosen message digest, padding, and RSA key.
int	**wolfSSL_dtls13_has_pending_msg**(WOLFSSL * ssl)checks if DTLSv1.3 stack has some messages sent but not yet acknowledged by the other peer
unsigned int	**wolfSSL_SESSION_get_max_early_data**(const WOLFSSL_SESSION * s)Get the maximum size of Early Data from a session.

	Name
int	**wolfSSL_CRYPTO_get_ex_new_index**(int , void * , void * , void * , void *)Get a new index for external data. This entry applies also for the following API:
int	**wolfSSL_CTX_set_client_cert_type**(WOLFSSL_CTX * ctx, const char * buf, int len)In case this function is called in a client side, set certificate types that can be sent to its peer. In case called in a server side, set certificate types that can be acceptable from its peer. Put cert types in the buffer with prioritised order. To reset the settings to default, pass NULL for the buffer or pass zero for len. By default, certificate type is only X509. In case both side intend to send or accept “Raw public key” cert, WOLFSSL_CERT_TYPE_RPK should be included in the buffer to set.
int	**wolfSSL_CTX_set_server_cert_type**(WOLFSSL_CTX * ctx, const char * buf, int len)In case this function is called in a server side, set certificate types that can be sent to its peer. In case called in a client side, set certificate types that can be acceptable from its peer. Put cert types in the buffer with prioritised order. To reset the settings to default, pass NULL for the buffer or pass zero for len. By default, certificate type is only X509. In case both side intend to send or accept “Raw public key” cert, WOLFSSL_CERT_TYPE_RPK should be included in the buffer to set.
int	**wolfSSL_set_client_cert_type**(WOLFSSL * ssl, const char * buf, int len)In case this function is called in a client side, set certificate types that can be sent to its peer. In case called in a server side, set certificate types that can be acceptable from its peer. Put cert types in the buffer with prioritised order. To reset the settings to default, pass NULL for the buffer or pass zero for len. By default, certificate type is only X509. In case both side intend to send or accept “Raw public key” cert, WOLFSSL_CERT_TYPE_RPK should be included in the buffer to set.

	Name
int	**wolfSSL_set_server_cert_type**(WOLFSSL * ssl, const char * buf, int len)In case this function is called in a server side, set certificate types that can be sent to its peer. In case called in a client side, set certificate types that can be acceptable from its peer. Put cert types in the buffer with prioritised order. To reset the settings to default, pass NULL for the buffer or pass zero for len. By default, certificate type is only X509. In case both side intend to send or accept “Raw public key” cert, WOLFSSL_CERT_TYPE_RPK should be included in the buffer to set.
int	**wolfSSL_get_negotiated_client_cert_type**(WOLFSSL * ssl, int * tp)This function returns the result of the client certificate type negotiation done in ClientHello and ServerHello. WOLFSSL_SUCCESS is returned as a return value if no negotiation occurs and WOLFSSL_CERT_TYPE_UNKNOWN is returned as the certificate type.
int	**wolfSSL_get_negotiated_server_cert_type**(WOLFSSL * ssl, int * tp)This function returns the result of the server certificate type negotiation done in ClientHello and ServerHello. WOLFSSL_SUCCESS is returned as a return value if no negotiation occurs and WOLFSSL_CERT_TYPE_UNKNOWN is returned as the certificate type.
int	**wolfSSL_dtls_cid_use**(WOLFSSL * ssl)Enable use of ConnectionID extensions for the SSL object. See RFC 9146 and RFC 9147.
int	**wolfSSL_dtls_cid_is_enabled**(WOLFSSL * ssl)If invoked after the handshake is complete it checks if ConnectionID was successfully negotiated for the SSL object. See RFC 9146 and RFC 9147.
int	**wolfSSL_dtls_cid_set**(WOLFSSL * ssl, unsigned char * cid, unsigned int size)Set the ConnectionID used by the other peer to send records in this connection. See RFC 9146 and RFC 9147. The ConnectionID must be at maximum DTLS_CID_MAX_SIZE, that is an tunable compile time define, and it can’t never be bigger than 255 bytes.
int	**wolfSSL_dtls_cid_get_rx_size**(WOLFSSL * ssl, unsigned int * size)Get the size of the ConnectionID used by the other peer to send records in this connection. See RFC 9146 and RFC 9147. The size is stored in the parameter size.

	Name
int	**wolfSSL_dtls_cid_get_rx**(WOLFSSL * ssl, unsigned char * buffer, unsigned int bufferSz)Copy the ConnectionID used by the other peer to send records in this connection into the buffer pointed by the parameter buffer. See RFC 9146 and RFC.
int	**wolfSSL_dtls_cid_get0_rx**(WOLFSSL * ssl, unsigned char ** cid)Get the ConnectionID used by the other peer. See RFC 9146 and RFC 9147.
int	**wolfSSL_dtls_cid_get_tx_size**(WOLFSSL * ssl, unsigned int * size)Get the size of the ConnectionID used to send records in this connection. See RFC 9146 and RFC 9147. The size is stored in the parameter size.
int	**wolfSSL_dtls_cid_get_tx**(WOLFSSL * ssl, unsigned char * buffer, unsigned int bufferSz)Copy the ConnectionID used when sending records in this connection into the buffer pointer by the parameter buffer. See RFC 9146 and RFC 9147. The available size need to be provided in bufferSz.
int	**wolfSSL_dtls_cid_get0_tx**(WOLFSSL * ssl, unsigned char ** cid)Get the ConnectionID used when sending records in this connection. See RFC 9146 and RFC 9147.
const unsigned char *	**wolfSSL_dtls_cid_parse**(const unsigned char * msg, unsigned int msgSz, unsigned int cidSz)Extract the ConnectionID from a record datagram/message. See RFC 9146 and RFC 9147.
int	**wolfSSL_get_client_suites_sigalgs**(const WOLFSSL * ssl, const byte ** suites, word16 * suiteSz, const byte ** hashSigAlgo, word16 * hashSigAlgoSz)This function returns the raw list of ciphersuites and signature algorithms offered by the client. The lists are only stored and returned inside a callback setup with wolfSSL_CTX_set_cert_cb(). This is useful to be able to dynamically load certificates and keys based on the available ciphersuites and signature algorithms.
WOLFSSL_CIPHERSUITE_INFO	**wolfSSL_get_ciphersuite_info**(byte first, byte second)This returns information about the ciphersuite directly from the raw ciphersuite bytes.
int	**wolfSSL_get_sigalg_info**(byte first, byte second, int * hashAlgo, int * sigAlgo)This returns information about the hash and signature algorithm directly from the raw ciphersuite bytes.

C.52.2 Functions Documentation

C.52.2.1 function wolfDTLSv1_2_client_method_ex

```
WOLFSSL_METHOD * wolfDTLSv1_2_client_method_ex(
    void * heap
)
```

This function initializes the DTLS v1.2 client method.

Parameters:

- **none** No parameters.

See:

- wolfSSL_Init
- wolfSSL_CTX_new

Return: pointer This function returns a pointer to a new WOLFSSL_METHOD structure.

Example

```
wolfSSL_Init();
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(wolfDTLSv1_2_client_method());
…
WOLFSSL* ssl = wolfSSL_new(ctx);
…
```

C.52.2.2 function wolfSSLv23_method

```
WOLFSSL_METHOD * wolfSSLv23_method(
    void
)
```

This function returns a WOLFSSL_METHOD similar to wolfSSLv23_client_method except that it is not determined which side yet (server/client).

Parameters:

- **none** No parameters.

See:

- wolfSSL_new
- wolfSSL_free

Return:

- WOLFSSL_METHOD* On successful creations returns a WOLFSSL_METHOD pointer
- NULL Null if memory allocation error or failure to create method

Example

```
WOLFSSL* ctx;
ctx  = wolfSSL_CTX_new(wolfSSLv23_method());
// check ret value
```

C.52.2.3 function wolfSSLv3_server_method

```
WOLFSSL_METHOD * wolfSSLv3_server_method(
    void
)
```

The wolfSSLv3_server_method().

Parameters:

- **none** No parameters.

See:

- wolfTLSv1_server_method
- wolfTLSv1_1_server_method
- wolfTLSv1_2_server_method
- wolfTLSv1_3_server_method
- wolfDTLSv1_server_method
- wolfSSLv23_server_method
- wolfSSL_CTX_new

Return:

- – If successful, the call will return a pointer to the newly created WOLFSSL_METHOD structure.
- FAIL If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfSSLv3_server_method();
if (method == NULL) {
    unable to get method
}

ctx = wolfSSL_CTX_new(method);
...
```

C.52.2.4 function wolfSSLv3_client_method

```
WOLFSSL_METHOD * wolfSSLv3_client_method(
    void
)
```

The wolfSSLv3_client_method().

Parameters:

- **none** No parameters.

See:

- wolfTLSv1_client_method
- wolfTLSv1_1_client_method
- wolfTLSv1_2_client_method
- wolfTLSv1_3_client_method
- wolfDTLSv1_client_method
- wolfSSLv23_client_method
- wolfSSL_CTX_new

Return:

- – If successful, the call will return a pointer to the newly created WOLFSSL_METHOD structure.
- FAIL If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfSSLv3_client_method();
if (method == NULL) {
    unable to get method
}

ctx = wolfSSL_CTX_new(method);
...
```

C.52.2.5 function wolfTLSv1_server_method

```
WOLFSSL_METHOD * wolfTLSv1_server_method(
    void
)
```

The wolfTLSv1_server_method().

Parameters:

- **none** No parameters.

See:

- wolfSSLv3_server_method
- wolfTLSv1_1_server_method
- wolfTLSv1_2_server_method
- wolfTLSv1_3_server_method
- wolfDTLSv1_server_method
- wolfSSLv23_server_method
- wolfSSL_CTX_new

Return:

- – If successful, the call will return a pointer to the newly created WOLFSSL_METHOD structure.
- FAIL If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfTLSv1_server_method();
if (method == NULL) {
    unable to get method
}
```

```
ctx = wolfSSL_CTX_new(method);
...
```

C.52.2.6 function wolfTLSv1_client_method

```
WOLFSSL_METHOD * wolfTLSv1_client_method(
    void
)
```

The wolfTLSv1_client_method().

Parameters:

- **none** No parameters.

See:

- wolfSSLv3_client_method
- wolfTLSv1_1_client_method
- wolfTLSv1_2_client_method
- wolfTLSv1_3_client_method
- wolfDTLSv1_client_method
- wolfSSLv23_client_method
- wolfSSL_CTX_new

Return:

- – If successful, the call will return a pointer to the newly created WOLFSSL_METHOD structure.
- FAIL If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfTLSv1_client_method();
if (method == NULL) {
    unable to get method
}

ctx = wolfSSL_CTX_new(method);
...
```

C.52.2.7 function wolfTLSv1_1_server_method

```
WOLFSSL_METHOD * wolfTLSv1_1_server_method(
    void
)
```

The wolfTLSv1_1_server_method().

Parameters:

- **none** No parameters.

See:

- wolfSSLv3_server_method

- wolfTLSv1_server_method
- wolfTLSv1_2_server_method
- wolfTLSv1_3_server_method
- wolfDTLSv1_server_method
- wolfSSLv23_server_method
- wolfSSL_CTX_new

Return:

- – If successful, the call will return a pointer to the newly created WOLFSSL_METHOD structure.
- FAIL If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfTLSv1_1_server_method();
if (method == NULL) {
    // unable to get method
}

ctx = wolfSSL_CTX_new(method);
...
```

C.52.2.8 function wolfTLSv1_1_client_method

```
WOLFSSL_METHOD * wolfTLSv1_1_client_method(
    void
)
```

The wolfTLSv1_1_client_method().

Parameters:

- **none** No parameters.

See:

- wolfSSLv3_client_method
- wolfTLSv1_client_method
- wolfTLSv1_2_client_method
- wolfTLSv1_3_client_method
- wolfDTLSv1_client_method
- wolfSSLv23_client_method
- wolfSSL_CTX_new

Return:

- – If successful, the call will return a pointer to the newly created WOLFSSL_METHOD structure.
- FAIL If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Example

```
#include <wolfssl/ssl.h>
```

```
WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfTLSv1_1_client_method();
if (method == NULL) {
    // unable to get method
}

ctx = wolfSSL_CTX_new(method);
...
```

C.52.2.9 function wolfTLSv1_2_server_method

```
WOLFSSL_METHOD * wolfTLSv1_2_server_method(
    void
)
```

The wolfTLSv1_2_server_method().

Parameters:

- **none** No parameters.

See:

- wolfSSLv3_server_method
- wolfTLSv1_server_method
- wolfTLSv1_1_server_method
- wolfTLSv1_3_server_method
- wolfDTLSv1_server_method
- wolfSSLv23_server_method
- wolfSSL_CTX_new

Return:

- – If successful, the call will return a pointer to the newly created WOLFSSL_METHOD structure.
- FAIL If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfTLSv1_2_server_method();
if (method == NULL) {
    // unable to get method
}

ctx = wolfSSL_CTX_new(method);
...
```

C.52.2.10 function wolfTLSv1_2_client_method

```
WOLFSSL_METHOD * wolfTLSv1_2_client_method(
    void
)
```

The wolfTLSv1_2_client_method().

Parameters:

- **none** No parameters.

See:

- wolfSSLv3_client_method
- wolfTLSv1_client_method
- wolfTLSv1_1_client_method
- wolfTLSv1_3_client_method
- wolfDTLSv1_client_method
- wolfSSLv23_client_method
- wolfSSL_CTX_new

Return:

- – If successful, the call will return a pointer to the newly created WOLFSSL_METHOD structure.
- FAIL If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfTLSv1_2_client_method();
if (method == NULL) {
    // unable to get method
}

ctx = wolfSSL_CTX_new(method);
...
```

C.52.2.11 function wolfDTLSv1_client_method

```
WOLFSSL_METHOD * wolfDTLSv1_client_method(
    void
)
```

The wolfDTLSv1_client_method(). This function is only available when wolfSSL has been compiled with DTLS support (–enable_dtls, or by defining wolfSSL_DTLS).

Parameters:

- **none** No parameters.

See:

- wolfSSLv3_client_method
- wolfTLSv1_client_method
- wolfTLSv1_1_client_method
- wolfTLSv1_2_client_method
- wolfTLSv1_3_client_method

- wolfSSLv23_client_method
- wolfSSL_CTX_new

Return:

- – If successful, the call will return a pointer to the newly created WOLFSSL_METHOD structure.
- FAIL If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Example

```
WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfDTLSv1_client_method();
if (method == NULL) {
    // unable to get method
}

ctx = wolfSSL_CTX_new(method);
...
```

C.52.2.12 function wolfDTLSv1_server_method

```
WOLFSSL_METHOD * wolfDTLSv1_server_method(
    void
)
```

The wolfDTLSv1_server_method(). This function is only available when wolfSSL has been compiled with DTLS support (–enable_dtls, or by defining wolfSSL_DTLS).

Parameters:

- **none** No parameters.

See:

- wolfSSLv3_server_method
- wolfTLSv1_server_method
- wolfTLSv1_1_server_method
- wolfTLSv1_2_server_method
- wolfTLSv1_3_server_method
- wolfSSLv23_server_method
- wolfSSL_CTX_new

Return:

- – If successful, the call will return a pointer to the newly created WOLFSSL_METHOD structure.
- FAIL If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Example

```
WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfDTLSv1_server_method();
if (method == NULL) {
    // unable to get method
}
```

```
ctx = wolfSSL_CTX_new(method);
...
```

C.52.2.13 function wolfDTLSv1_3_server_method

```
WOLFSSL_METHOD * wolfDTLSv1_3_server_method(
    void
)
```

The wolfDTLSv1_3_server_method(). This function is only available when wolfSSL has been compiled with DTLSv1.3 support (–enable_dtls13, or by defining wolfSSL_DTLS13).

Parameters:

- **none** No parameters.

See: wolfDTLSv1_3_client_method

Return:

- – If successful, the call will return a pointer to the newly created WOLFSSL_METHOD structure.
- FAIL If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Example

```
WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfDTLSv1_3_server_method();
if (method == NULL) {
    // unable to get method
}

ctx = wolfSSL_CTX_new(method);
...
```

C.52.2.14 function wolfDTLSv1_3_client_method

```
WOLFSSL_METHOD * wolfDTLSv1_3_client_method(
    void
)
```

The wolfDTLSv1_3_client_method(). This function is only available when wolfSSL has been compiled with DTLSv1.3 support (–enable_dtls13, or by defining wolfSSL_DTLS13).

Parameters:

- **none** No parameters.

See: wolfDTLSv1_3_server_method

Return:

- – If successful, the call will return a pointer to the newly created WOLFSSL_METHOD structure.
- FAIL If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Example

```
WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfDTLSv1_3_client_method();
if (method == NULL) {
    // unable to get method
}

ctx = wolfSSL_CTX_new(method);
...
```

C.52.2.15 function wolfDTLS_server_method

```
WOLFSSL_METHOD * wolfDTLS_server_method(
    void
)
```

The wolfDTLS_server_method(). This function is only available when wolfSSL has been compiled with DTLS support (–enable_dtls, or by defining wolfSSL_DTLS).

Parameters:

- **none** No parameters.

See:

- wolfDTLS_client_method
- wolfSSL_SetMinVersion

Return:

- – If successful, the call will return a pointer to the newly created WOLFSSL_METHOD structure.
- FAIL If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Example

```
WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfDTLS_server_method();
if (method == NULL) {
    // unable to get method
}

ctx = wolfSSL_CTX_new(method);
...
```

C.52.2.16 function wolfDTLS_client_method

```
WOLFSSL_METHOD * wolfDTLS_client_method(
    void
)
```

The wolfDTLS_client_method(). This function is only available when wolfSSL has been compiled with DTLS support (–enable_dtls, or by defining wolfSSL_DTLS).

Parameters:

- **none** No parameters.

See:

- wolfDTLS_server_method
- wolfSSL_SetMinVersion

Return:

- – If successful, the call will return a pointer to the newly created WOLFSSL_METHOD structure.
- FAIL If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Example

```
WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfDTLS_client_method();
if (method == NULL) {
    // unable to get method
}

ctx = wolfSSL_CTX_new(method);
...
```

C.52.2.17 function wolfDTLSv1_2_server_method

```
WOLFSSL_METHOD * wolfDTLSv1_2_server_method(
    void
)
```

This function creates and initializes a WOLFSSL_METHOD for the server side.

Parameters:

- **none** No parameters.

See: wolfSSL_CTX_new

Return: This function returns a WOLFSSL_METHOD pointer.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(wolfDTLSv1_2_server_method());
WOLFSSL* ssl = WOLFSSL_new(ctx);
…
```

C.52.2.18 function wolfSSL_use_old_poly

```
int wolfSSL_use_old_poly(
    WOLFSSL * ssl,
    int value
)
```

Since there is some differences between the first release and newer versions of chacha-poly AEAD construction we have added an option to communicate with servers/clients using the older version. By default wolfSSL uses the new version.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

- **value** whether or not to use the older version of setting up the information for poly1305. Passing a flag value of 1 indicates yes use the old poly AEAD, to switch back to using the new version pass a flag value of 0.

See: none

Return: 0 upon success

Example

```
int ret = 0;
WOLFSSL* ssl;
...

ret = wolfSSL_use_old_poly(ssl, 1);
if (ret != 0) {
    // failed to set poly1305 AEAD version
}
```

C.52.2.19 function wolfSSL_dtls_import

```
int wolfSSL_dtls_import(
    WOLFSSL * ssl,
    unsigned char * buf,
    unsigned int sz
)
```

The wolfSSL_dtls_import() function is used to parse in a serialized session state. This allows for picking up the connection after the handshake has been completed.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **buf** serialized session to import.
- **sz** size of serialized session buffer.

See:

- wolfSSL_new
- wolfSSL_CTX_new
- wolfSSL_CTX_dtls_set_export

Return:

- Success If successful, the amount of the buffer read will be returned.
- Failure All unsuccessful return values will be less than 0.
- VERSION_ERROR If a version mismatch is found ie DTLS v1 and ctx was set up for DTLS v1.2 then VERSION_ERROR is returned.

Example

```
WOLFSSL* ssl;
int ret;
unsigned char buf[MAX];
bufSz = MAX;
...
//get information sent from wc_dtls_export function and place it in buf
fread(buf, 1, bufSz, input);
ret = wolfSSL_dtls_import(ssl, buf, bufSz);
if (ret < 0) {
// handle error case
```

```
}
// no wolfSSL_accept needed since handshake was already done
...
ret = wolfSSL_write(ssl) and wolfSSL_read(ssl);
...
```

C.52.2.20 function wolfSSL_tls_import

```
int wolfSSL_tls_import(
    WOLFSSL * ssl,
    const unsigned char * buf,
    unsigned int sz
)
```

Used to import a serialized TLS session. This function is for importing the state of the connection. WARNING: buf contains sensitive information about the state and is best to be encrypted before storing if stored. Additional debug info can be displayed with the macro WOLFSSL_SESSION_EXPORT_DEBUG defined.

Parameters:

- **ssl** WOLFSSL structure to import the session into
- **buf** serialized session
- **sz** size of buffer 'buf'

See:

- wolfSSL_dtls_import
- wolfSSL_tls_export

Return: the number of bytes read from buffer 'buf'

C.52.2.21 function wolfSSL_CTX_dtls_set_export

```
int wolfSSL_CTX_dtls_set_export(
    WOLFSSL_CTX * ctx,
    wc_dtls_export func
)
```

The wolfSSL_CTX_dtls_set_export() function is used to set the callback function for exporting a session. It is allowed to pass in NULL as the parameter func to clear the export function previously stored. Used on the server side and is called immediately after handshake is completed.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created with wolfSSL_CTX_new().
- **func** wc_dtls_export function to use when exporting a session.

See:

- wolfSSL_new
- wolfSSL_CTX_new
- wolfSSL_dtls_set_export
- Static buffer use

Return:

- SSL_SUCCESS upon success.
- BAD_FUNC_ARG If null or not expected arguments are passed in

Example

```
int send_session(WOLFSSL* ssl, byte* buf, word32 sz, void* userCtx);
// body of send session (wc_dtls_export) that passes
// buf (serialized session) to destination
WOLFSSL_CTX* ctx;
int ret;
...
ret = wolfSSL_CTX_dtls_set_export(ctx, send_session);
if (ret != SSL_SUCCESS) {
    // handle error case
}
...
ret = wolfSSL_accept(ssl);
...
```

C.52.2.22 function wolfSSL_dtls_set_export

```
int wolfSSL_dtls_set_export(
    WOLFSSL * ssl,
    wc_dtls_export func
)
```

The wolfSSL_dtls_set_export() function is used to set the callback function for exporting a session. It is allowed to pass in NULL as the parameter func to clear the export function previously stored. Used on the server side and is called immediately after handshake is completed.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **func** wc_dtls_export function to use when exporting a session.

See:

- wolfSSL_new
- wolfSSL_CTX_new
- wolfSSL_CTX_dtls_set_export

Return:

- SSL_SUCCESS upon success.
- BAD_FUNC_ARG If null or not expected arguments are passed in

Example

```
int send_session(WOLFSSL* ssl, byte* buf, word32 sz, void* userCtx);
// body of send session (wc_dtls_export) that passes
// buf (serialized session) to destination
WOLFSSL* ssl;
int ret;
...
ret = wolfSSL_dtls_set_export(ssl, send_session);
if (ret != SSL_SUCCESS) {
    // handle error case
}
...
ret = wolfSSL_accept(ssl);
...
```

C.52.2.23 function wolfSSL_dtls_export

```
int wolfSSL_dtls_export(
    WOLFSSL * ssl,
    unsigned char * buf,
    unsigned int * sz
)
```

The wolfSSL_dtls_export() function is used to serialize a WOLFSSL session into the provided buffer. Allows for less memory overhead than using a function callback for sending a session and choice over when the session is serialized. If buffer is NULL when passed to function then sz will be set to the size of buffer needed for serializing the WOLFSSL session.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **buf** buffer to hold serialized session.
- **sz** size of buffer.

See:

- wolfSSL_new
- wolfSSL_CTX_new
- wolfSSL_CTX_dtls_set_export
- wolfSSL_dtls_import

Return:

- Success If successful, the amount of the buffer used will be returned.
- Failure All unsuccessful return values will be less than 0.

Example

```
WOLFSSL* ssl;
int ret;
unsigned char buf[MAX];
bufSz = MAX;
...
ret = wolfSSL_dtls_export(ssl, buf, bufSz);
if (ret < 0) {
    // handle error case
}
...
```

C.52.2.24 function wolfSSL_tls_export

```
int wolfSSL_tls_export(
    WOLFSSL * ssl,
    unsigned char * buf,
    unsigned int * sz
)
```

Used to export a serialized TLS session. This function is for exporting a serialized state of the connection. In most cases wolfSSL_get1_session should be used instead of wolfSSL_tls_export. Additional debug info can be displayed with the macro WOLFSSL_SESSION_EXPORT_DEBUG defined. WARNING: buf contains sensitive information about the state and is best to be encrypted before storing if stored.

Parameters:

- **ssl** WOLFSSL structure to export the session from
- **buf** output of serialized session

- **sz** size in bytes set in 'buf'

See:

- wolfSSL_dtls_import
- wolfSSL_tls_import

Return: the number of bytes written into buffer 'buf'

C.52.2.25 function wolfSSL_CTX_load_static_memory

```
int wolfSSL_CTX_load_static_memory(
    WOLFSSL_CTX ** ctx,
    wolfSSL_method_func method,
    unsigned char * buf,
    unsigned int sz,
    int flag,
    int max
)
```

This function is used to set aside static memory for a CTX. Memory set aside is then used for the CTX's lifetime and for any SSL objects created from the CTX. By passing in a NULL ctx pointer and a wolfSSL_method_func function the creation of the CTX itself will also use static memory. wolfSSL_method_func has the function signature of WOLFSSL_METHOD* (*wolfSSL_method_func)(void* heap);. Passing in 0 for max makes it behave as if not set and no max concurrent use restrictions is in place. The flag value passed in determines how the memory is used and behavior while operating. Available flags are the following: 0 - default general memory, WOLFMEM_IO_POOL - used for input/output buffer when sending receiving messages and overrides general memory, so all memory in buffer passed in is used for IO, WOLFMEM_IO_FIXED - same as WOLFMEM_IO_POOL but each SSL now keeps two buffers to themselves for their lifetime, WOLFMEM_TRACK_STATS - each SSL keeps track of memory stats while running.

Parameters:

- **ctx** address of pointer to a WOLFSSL_CTX structure.
- **method** function to create protocol. (should be NULL if ctx is not also NULL)
- **buf** memory to use for all operations.
- **sz** size of memory buffer being passed in.
- **flag** type of memory.
- **max** max concurrent operations.

See:

- wolfSSL_CTX_new
- wolfSSL_CTX_is_static_memory
- wolfSSL_is_static_memory

Return:

- SSL_SUCCESS upon success.
- SSL_FAILURE upon failure.

Example

```
WOLFSSL_CTX* ctx;
WOLFSSL* ssl;
int ret;
unsigned char memory[MAX];
int memorySz = MAX;
unsigned char IO[MAX];
```

```
int IOSz = MAX;
int flag = WOLFMEM_IO_FIXED | WOLFMEM_TRACK_STATS;
...
// create ctx also using static memory, start with general memory to use
ctx = NULL:
ret = wolfSSL_CTX_load_static_memory(&ctx, wolfSSLv23_server_method_ex,
memory, memorySz, 0,    MAX_CONCURRENT_HANDSHAKES);
if (ret != SSL_SUCCESS) {
// handle error case
}
// load in memory for use with IO
ret = wolfSSL_CTX_load_static_memory(&ctx, NULL, IO, IOSz, flag,
MAX_CONCURRENT_IO);
if (ret != SSL_SUCCESS) {
// handle error case
}
...
```

C.52.2.26 function wolfSSL_CTX_is_static_memory

```
int wolfSSL_CTX_is_static_memory(
    WOLFSSL_CTX * ctx,
    WOLFSSL_MEM_STATS * mem_stats
)
```

This function does not change any of the connections behavior and is used only for gathering information about the static memory usage.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().
- **mem_stats** structure to hold information about static memory usage.

See:

- wolfSSL_CTX_new
- wolfSSL_CTX_load_static_memory
- wolfSSL_is_static_memory

Return:

- 1 is returned if using static memory for the CTX is true.
- 0 is returned if not using static memory.

Example

```
WOLFSSL_CTX* ctx;
int ret;
WOLFSSL_MEM_STATS mem_stats;
...
//get information about static memory with CTX
ret = wolfSSL_CTX_is_static_memory(ctx, &mem_stats);
if (ret == 1) {
    // handle case of is using static memory
    // print out or inspect elements of mem_stats
}
if (ret == 0) {
    //handle case of ctx not using static memory
```

```
}
...
```

C.52.2.27 function wolfSSL_is_static_memory

```
int wolfSSL_is_static_memory(
    WOLFSSL * ssl,
    WOLFSSL_MEM_CONN_STATS * mem_stats
)
```

wolfSSL_is_static_memory is used to gather information about a SSL's static memory usage. The return value indicates if static memory is being used and WOLFSSL_MEM_CONN_STATS will be filled out if and only if the flag WOLFMEM_TRACK_STATS was passed to the parent CTX when loading in static memory.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **mem_stats** structure to contain static memory usage.

See:

- wolfSSL_new
- wolfSSL_CTX_is_static_memory

Return:

- 1 is returned if using static memory for the CTX is true.
- 0 is returned if not using static memory.

Example

```
WOLFSSL* ssl;
int ret;
WOLFSSL_MEM_CONN_STATS mem_stats;
...
ret = wolfSSL_is_static_memory(ssl, mem_stats);
if (ret == 1) {
    // handle case when is static memory
    // investigate elements in mem_stats if WOLFMEM_TRACK_STATS flag
}
...
```

C.52.2.28 function wolfSSL_CTX_use_certificate_file

```
int wolfSSL_CTX_use_certificate_file(
    WOLFSSL_CTX * ctx,
    const char * file,
    int format
)
```

This function loads a certificate file into the SSL context (WOLFSSL_CTX). The file is provided by the file argument. The format argument specifies the format type of the file, either SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM. Please see the examples for proper usage.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new()
- **file** a pointer to the name of the file containing the certificate to be loaded into the wolfSSL SSL context.

- **format** - format of the certificates pointed to by file. Possible options are SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM.

See:

- wolfSSL_CTX_use_certificate_buffer
- wolfSSL_use_certificate_file
- wolfSSL_use_certificate_buffer

Return:

- SSL_SUCCESS upon success.
- SSL_FAILURE If the function call fails, possible causes might include the file is in the wrong format, or the wrong format has been given using the "format" argument, file doesn't exist, can't be read, or is corrupted, an out of memory condition occurs, Base16 decoding fails on the file.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx;
...
ret = wolfSSL_CTX_use_certificate_file(ctx, "./client-cert.pem",
                                       SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
    // error loading cert file
}
...
```

C.52.2.29 function wolfSSL_CTX_use_PrivateKey_file

```
int wolfSSL_CTX_use_PrivateKey_file(
    WOLFSSL_CTX * ctx,
    const char * file,
    int format
)
```

This function loads a private key file into the SSL context (WOLFSSL_CTX). The file is provided by the file argument. The format argument specifies the format type of the file - SSL_FILETYPE_ASN1or SSL_FILETYPE_PEM. Please see the examples for proper usage.

Parameters:

- **none** No parameters.

See:

- wolfSSL_CTX_use_PrivateKey_buffer
- wolfSSL_use_PrivateKey_file
- wolfSSL_use_PrivateKey_buffer
- wc_CryptoCb_RegisterDevice
- wolfSSL_CTX_SetDevId

Return:

- SSL_SUCCESS upon success.
- SSL_FAILURE The file is in the wrong format, or the wrong format has been given using the "format" argument. The file doesn't exist, can't be read, or is corrupted. An out of memory condition occurs. Base16 decoding fails on the file. The key file is encrypted but no password is provided.

If using an external key store and do not have the private key you can instead provide the public key and register the crypro callback to handle the signing. For this you can build with either build with

crypto callbacks or PK callbacks. To enable crypto callbacks use –enable-cryptocb or WOLF_CRYPTO_CB and register a crypto callback using wc_CryptoCb_RegisterDevice and set the associated devId using wolfSSL_CTX_SetDevId.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx;
...
ret = wolfSSL_CTX_use_PrivateKey_file(ctx, "./server-key.pem",
                                      SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
    // error loading key file
}
...
```

C.52.2.30 function wolfSSL_CTX_load_verify_locations

```
int wolfSSL_CTX_load_verify_locations(
    WOLFSSL_CTX * ctx,
    const char * file,
    const char * path
)
```

This function loads PEM-formatted CA certificate files into the SSL context (WOLFSSL_CTX). These certificates will be treated as trusted root certificates and used to verify certs received from peers during the SSL handshake. The root certificate file, provided by the file argument, may be a single certificate or a file containing multiple certificates. If multiple CA certs are included in the same file, wolfSSL will load them in the same order they are presented in the file. The path argument is a pointer to the name of a directory that contains certificates of trusted root CAs. If the value of file is not NULL, path may be specified as NULL if not needed. If path is specified and NO_WOLFSSL_DIR was not defined when building the library, wolfSSL will load all CA certificates located in the given directory. This function will attempt to load all files in the directory. This function expects PEM formatted CERT_TYPE file with header "—–BEGIN CERTIFICATE—–".

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().
- **file** pointer to name of the file containing PEM-formatted CA certificates.
- **path** pointer to the name of a directory to load PEM-formatted certificates from.

See:

- wolfSSL_CTX_load_verify_locations_ex
- wolfSSL_CTX_load_verify_buffer
- wolfSSL_CTX_use_certificate_file
- wolfSSL_CTX_use_PrivateKey_file
- wolfSSL_CTX_use_certificate_chain_file
- wolfSSL_use_certificate_file
- wolfSSL_use_PrivateKey_file
- wolfSSL_use_certificate_chain_file

Return:

- SSL_SUCCESS up success.
- SSL_FAILURE will be returned if ctx is NULL, or if both file and path are NULL.
- SSL_BAD_FILETYPE will be returned if the file is the wrong format.
- SSL_BAD_FILE will be returned if the file doesn't exist, can't be read, or is corrupted.
- MEMORY_E will be returned if an out of memory condition occurs.

- ASN_INPUT_E will be returned if Base16 decoding fails on the file.
- ASN_BEFORE_DATE_E will be returned if the current date is before the before date.
- ASN_AFTER_DATE_E will be returned if the current date is after the after date.
- BUFFER_E will be returned if a chain buffer is bigger than the receiving buffer.
- BAD_PATH_ERROR will be returned if opendir() fails when trying to open path.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx;
...
ret = wolfSSL_CTX_load_verify_locations(ctx, "./ca-cert.pem", NULL);
if (ret != WOLFSSL_SUCCESS) {
    // error loading CA certs
}
...
```

C.52.2.31 function wolfSSL_CTX_load_verify_locations_ex

```
int wolfSSL_CTX_load_verify_locations_ex(
    WOLFSSL_CTX * ctx,
    const char * file,
    const char * path,
    unsigned int flags
)
```

This function loads PEM-formatted CA certificate files into the SSL context (WOLFSSL_CTX). These certificates will be treated as trusted root certificates and used to verify certs received from peers during the SSL handshake. The root certificate file, provided by the file argument, may be a single certificate or a file containing multiple certificates. If multiple CA certs are included in the same file, wolfSSL will load them in the same order they are presented in the file. The path argument is a pointer to the name of a directory that contains certificates of trusted root CAs. If the value of file is not NULL, path may be specified as NULL if not needed. If path is specified and NO_WOLFSSL_DIR was not defined when building the library, wolfSSL will load all CA certificates located in the given directory. This function will attempt to load all files in the directory based on flags specified. This function expects PEM formatted CERT_TYPE files with header "—–BEGIN CERTIFICATE—–".

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().
- **file** pointer to name of the file containing PEM-formatted CA certificates.
- **path** pointer to the name of a directory to load PEM-formatted certificates from.
- **flags** possible mask values are: WOLFSSL_LOAD_FLAG_IGNORE_ERR, WOLFSSL_LOAD_FLAG_DATE_ERR_OKAY and WOLFSSL_LOAD_FLAG_PEM_CA_ONLY

See:

- wolfSSL_CTX_load_verify_locations
- wolfSSL_CTX_load_verify_buffer
- wolfSSL_CTX_use_certificate_file
- wolfSSL_CTX_use_PrivateKey_file
- wolfSSL_CTX_use_certificate_chain_file
- wolfSSL_use_certificate_file
- wolfSSL_use_PrivateKey_file
- wolfSSL_use_certificate_chain_file

Return:

- SSL_SUCCESS up success.

- SSL_FAILURE will be returned if ctx is NULL, or if both file and path are NULL. This will also be returned if at least one cert is loaded successfully but there is one or more that failed. Check error stack for reason.
- SSL_BAD_FILETYPE will be returned if the file is the wrong format.
- SSL_BAD_FILE will be returned if the file doesn't exist, can't be read, or is corrupted.
- MEMORY_E will be returned if an out of memory condition occurs.
- ASN_INPUT_E will be returned if Base16 decoding fails on the file.
- BUFFER_E will be returned if a chain buffer is bigger than the receiving buffer.
- BAD_PATH_ERROR will be returned if opendir() fails when trying to open path.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx;
...
ret = wolfSSL_CTX_load_verify_locations_ex(ctx, NULL, "./certs/external",
    WOLFSSL_LOAD_FLAG_PEM_CA_ONLY);
if (ret != WOLFSSL_SUCCESS) {
    // error loading CA certs
}
...
```

C.52.2.32 function wolfSSL_get_system_CA_dirs

```
const char ** wolfSSL_get_system_CA_dirs(
    word32 * num
)
```

This function returns a pointer to an array of strings representing directories wolfSSL will search for system CA certs when wolfSSL_CTX_load_system_CA_certs is called. On systems that don't store certificates in an accessible system directory (such as Apple platforms), this function will always return NULL.

Parameters:

- **num** pointer to a word32 that will be populated with the length of the array of strings.

See:

- wolfSSL_CTX_load_system_CA_certs
- wolfSSL_CTX_load_verify_locations
- wolfSSL_CTX_load_verify_locations_ex

Return:

- Valid pointer on success.
- NULL pointer on failure.

Example

```
WOLFSSL_CTX* ctx;
const char** dirs;
word32 numDirs;

dirs = wolfSSL_get_system_CA_dirs(&numDirs);
for (int i = 0; i < numDirs; ++i) {
    printf("Potential system CA dir: %s\n", dirs[i]);
}
...
```

C.52.2.33 function wolfSSL_CTX_load_system_CA_certs

```
int wolfSSL_CTX_load_system_CA_certs(
    WOLFSSL_CTX * ctx
)
```

On most platforms (including Linux and Windows), this function attempts to load CA certificates into a WOLFSSL_CTX from an OS-dependent CA certificate store. Loaded certificates will be trusted.

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().

See:

- wolfSSL_get_system_CA_dirs
- wolfSSL_CTX_load_verify_locations
- wolfSSL_CTX_load_verify_locations_ex

Return:

- WOLFSSL_SUCCESS on success.
- WOLFSSL_BAD_PATH if no system CA certs were loaded.
- WOLFSSL_FAILURE for other failure types (e.g. Windows cert store wasn't properly closed).

On Apple platforms (excluding macOS), certificates can't be obtained from the system, and therefore cannot be loaded into the wolfSSL certificate manager. For these platforms, this function enables TLS connections bound to the WOLFSSL_CTX to use the native system trust APIs to verify authenticity of the peer certificate chain if the authenticity of the peer cannot first be authenticated against certificates loaded by the user.

The platforms supported and tested are: Linux (Debian, Ubuntu, Gentoo, Fedora, RHEL), Windows 10/11, Android, macOS, and iOS.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx;
...
ret = wolfSSL_CTX_load_system_CA_certs(ctx,);
if (ret != WOLFSSL_SUCCESS) {
    // error loading system CA certs
}
...
```

C.52.2.34 function wolfSSL_CTX_trust_peer_cert

```
int wolfSSL_CTX_trust_peer_cert(
    WOLFSSL_CTX * ctx,
    const char * file,
    int type
)
```

This function loads a certificate to use for verifying a peer when performing a TLS/SSL handshake. The peer certificate sent during the handshake is compared by using the SKID when available and the signature. If these two things do not match then any loaded CAs are used. Feature is enabled by defining the macro WOLFSSL_TRUST_PEER_CERT. Please see the examples for proper usage.

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().
- **file** pointer to name of the file containing certificates

- **type** type of certificate being loaded ie SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM.

See:

- wolfSSL_CTX_load_verify_buffer
- wolfSSL_CTX_use_certificate_file
- wolfSSL_CTX_use_PrivateKey_file
- wolfSSL_CTX_use_certificate_chain_file
- wolfSSL_CTX_trust_peer_buffer
- wolfSSL_CTX_Unload_trust_peers
- wolfSSL_use_certificate_file
- wolfSSL_use_PrivateKey_file
- wolfSSL_use_certificate_chain_file

Return:

- SSL_SUCCES upon success.
- SSL_FAILURE will be returned if ctx is NULL, or if both file and type are invalid.
- SSL_BAD_FILETYPE will be returned if the file is the wrong format.
- SSL_BAD_FILE will be returned if the file doesn't exist, can't be read, or is corrupted.
- MEMORY_E will be returned if an out of memory condition occurs.
- ASN_INPUT_E will be returned if Base16 decoding fails on the file.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
...

ret = wolfSSL_CTX_trust_peer_cert(ctx, "./peer-cert.pem",
SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
    // error loading trusted peer cert
}
...
```

C.52.2.35 function wolfSSL_CTX_use_certificate_chain_file

```
int wolfSSL_CTX_use_certificate_chain_file(
    WOLFSSL_CTX * ctx,
    const char * file
)
```

This function loads a chain of certificates into the SSL context (WOLFSSL_CTX). The file containing the certificate chain is provided by the file argument, and must contain PEM-formatted certificates. This function will process up to MAX_CHAIN_DEPTH (default = 9, defined in internal.h) certificates, plus the subject cert.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new()
- **file** a pointer to the name of the file containing the chain of certificates to be loaded into the wolfSSL SSL context. Certificates must be in PEM format.

See:

- wolfSSL_CTX_use_certificate_file
- wolfSSL_CTX_use_certificate_buffer
- wolfSSL_use_certificate_file
- wolfSSL_use_certificate_buffer

Return:

- SSL_SUCCESS upon success
- SSL_FAILURE If the function call fails, possible causes might include the file is in the wrong format, or the wrong format has been given using the “format” argument, file doesn’t exist, can’t be read, or is corrupted, an out of memory condition occurs.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx;
...
ret = wolfSSL_CTX_use_certificate_chain_file(ctx, “./cert-chain.pem”);
if (ret != SSL_SUCCESS) {
    // error loading cert file
}
...
```

C.52.2.36 function wolfSSL_CTX_use_RSAPrivateKey_file

```
int wolfSSL_CTX_use_RSAPrivateKey_file(
    WOLFSSL_CTX * ctx,
    const char * file,
    int format
)
```

This function loads the private RSA key used in the SSL connection into the SSL context (WOLFSSL_CTX). This function is only available when wolfSSL has been compiled with the OpenSSL compatibility layer enabled (–enable-opensslExtra, #define OPENSSL_EXTRA), and is identical to the more-typically used wolfSSL_CTX_use_PrivateKey_file() function. The file argument contains a pointer to the RSA private key file, in the format specified by format.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new()
- **file** a pointer to the name of the file containing the RSA private key to be loaded into the wolfSSL SSL context, with format as specified by format.
- **format** the encoding type of the RSA private key specified by file. Possible values include SSL_FILETYPE_PEM and SSL_FILETYPE_ASN1.

See:

- wolfSSL_CTX_use_PrivateKey_buffer
- wolfSSL_CTX_use_PrivateKey_file
- wolfSSL_use_RSAPrivateKey_file
- wolfSSL_use_PrivateKey_buffer
- wolfSSL_use_PrivateKey_file

Return:

- SSL_SUCCESS upon success.
- SSL_FAILURE If the function call fails, possible causes might include: The input key file is in the wrong format, or the wrong format has been given using the “format” argument, file doesn’t exist, can’t be read, or is corrupted, an out of memory condition occurs.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx;
...
```

```
ret = wolfSSL_CTX_use_RSAPrivateKey_file(ctx, "./server-key.pem",
                                         SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
    // error loading private key file
}
...
```

C.52.2.37 function wolfSSL_get_verify_depth

```
long wolfSSL_get_verify_depth(
    WOLFSSL * ssl
)
```

This function returns the maximum chain depth allowed, which is 9 by default, for a valid session i.e. there is a non-null session object (ssl).

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See: wolfSSL_CTX_get_verify_depth

Return:

- MAX_CHAIN_DEPTH returned if the WOLFSSL structure is not NULL. By default the value is 9.
- BAD_FUNC_ARG returned if the WOLFSSL structure is NULL.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
WOLFSSL* ssl = wolfSSL_new(ctx);
...
long sslDep = wolfSSL_get_verify_depth(ssl);

if(sslDep > EXPECTED){
    // The verified depth is greater than what was expected
} else {
    // The verified depth is smaller or equal to the expected value
}
```

C.52.2.38 function wolfSSL_CTX_get_verify_depth

```
long wolfSSL_CTX_get_verify_depth(
    WOLFSSL_CTX * ctx
)
```

This function gets the certificate chaining depth using the CTX structure.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().

See:

- wolfSSL_CTX_use_certificate_chain_file
- wolfSSL_get_verify_depth

Return:

- MAX_CHAIN_DEPTH returned if the CTX struct is not NULL. The constant representation of the max certificate chain peer depth.
- BAD_FUNC_ARG returned if the CTX structure is NULL.

Example

```
WOLFSSL_METHOD method; // protocol method
WOLFSSL_CTX* ctx = WOLFSSL_CTX_new(method);
…
long ret = wolfSSL_CTX_get_verify_depth(ctx);

if(ret == EXPECTED){
    //  You have the expected value
} else {
    //  Handle an unexpected depth
}
```

C.52.2.39 function wolfSSL_use_certificate_file

```
int wolfSSL_use_certificate_file(
    WOLFSSL * ssl,
    const char * file,
    int format
)
```

This function loads a certificate file into the SSL session (WOLFSSL structure). The certificate file is provided by the file argument. The format argument specifies the format type of the file - either SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created with wolfSSL_new().
- **file** a pointer to the name of the file containing the certificate to be loaded into the wolfSSL SSL session, with format as specified by format.
- **format** the encoding type of the certificate specified by file. Possible values include SSL_FILETYPE_PEM and SSL_FILETYPE_ASN1.

See:

- wolfSSL_CTX_use_certificate_buffer
- wolfSSL_CTX_use_certificate_file
- wolfSSL_use_certificate_buffer

Return:

- SSL_SUCCESS upon success
- SSL_FAILURE If the function call fails, possible causes might include: The file is in the wrong format, or the wrong format has been given using the “format” argument, file doesn’t exist, can’t be read, or is corrupted, an out of memory condition occurs, Base16 decoding fails on the file

Example

```
int ret = 0;
WOLFSSL* ssl;
...
ret = wolfSSL_use_certificate_file(ssl, “./client-cert.pem”,
                                   SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
    // error loading cert file
}
...
```

C.52.2.40 function wolfSSL_use_PrivateKey_file

```
int wolfSSL_use_PrivateKey_file(
    WOLFSSL * ssl,
    const char * file,
    int format
)
```

This function loads a private key file into the SSL session (WOLFSSL structure). The key file is provided by the file argument. The format argument specifies the format type of the file - SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created with wolfSSL_new().
- **file** a pointer to the name of the file containing the key file to be loaded into the wolfSSL SSL session, with format as specified by format.
- **format** the encoding type of the key specified by file. Possible values include SSL_FILETYPE_PEM and SSL_FILETYPE_ASN1.

See:

- wolfSSL_CTX_use_PrivateKey_buffer
- wolfSSL_CTX_use_PrivateKey_file
- wolfSSL_use_PrivateKey_buffer
- wc_CryptoCb_RegisterDevice
- wolfSSL_SetDevId

Return:

- SSL_SUCCESS upon success.
- SSL_FAILURE If the function call fails, possible causes might include: The file is in the wrong format, or the wrong format has been given using the “format” argument, The file doesn’t exist, can’t be read, or is corrupted, An out of memory condition occurs, Base16 decoding fails on the file, The key file is encrypted but no password is provided

If using an external key store and do not have the private key you can instead provide the public key and register the crypro callback to handle the signing. For this you can build with either build with crypto callbacks or PK callbacks. To enable crypto callbacks use –enable-cryptocb or WOLF_CRYPTO_CB and register a crypto callback using wc_CryptoCb_RegisterDevice and set the associated devId using wolfSSL_SetDevId.

Example

```
int ret = 0;
WOLFSSL* ssl;
...
ret = wolfSSL_use_PrivateKey_file(ssl, “./server-key.pem”,
                                  SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
    // error loading key file
}
...
```

C.52.2.41 function wolfSSL_use_certificate_chain_file

```
int wolfSSL_use_certificate_chain_file(
    WOLFSSL * ssl,
    const char * file
)
```

This function loads a chain of certificates into the SSL session (WOLFSSL structure). The file containing the certificate chain is provided by the file argument, and must contain PEM-formatted certificates. This function will process up to MAX_CHAIN_DEPTH (default = 9, defined in internal.h) certificates, plus the subject certificate.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new()
- **file** a pointer to the name of the file containing the chain of certificates to be loaded into the wolfSSL SSL session. Certificates must be in PEM format.

See:

- wolfSSL_CTX_use_certificate_chain_file
- wolfSSL_CTX_use_certificate_chain_buffer
- wolfSSL_use_certificate_chain_buffer

Return:

- SSL_SUCCESS upon success.
- SSL_FAILURE If the function call fails, possible causes might include: The file is in the wrong format, or the wrong format has been given using the “format” argument, file doesn’t exist, can’t be read, or is corrupted, an out of memory condition occurs

Example

```
int ret = 0;
WOLFSSL* ctx;
...
ret = wolfSSL_use_certificate_chain_file(ssl, “./cert-chain.pem”);
if (ret != SSL_SUCCESS) {
    // error loading cert file
}
...
```

C.52.2.42 function wolfSSL_use_RSAPrivateKey_file

```
int wolfSSL_use_RSAPrivateKey_file(
    WOLFSSL * ssl,
    const char * file,
    int format
)
```

This function loads the private RSA key used in the SSL connection into the SSL session (WOLFSSL structure). This function is only available when wolfSSL has been compiled with the OpenSSL compatibility layer enabled (–enable-opensslExtra, #define OPENSSL_EXTRA), and is identical to the more-typically used wolfSSL_use_PrivateKey_file() function. The file argument contains a pointer to the RSA private key file, in the format specified by format.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new()
- **file** a pointer to the name of the file containing the RSA private key to be loaded into the wolfSSL SSL session, with format as specified by format.
- **format** the encoding type of the RSA private key specified by file. Possible values include SSL_FILETYPE_PEM and SSL_FILETYPE_ASN1.

See:

- wolfSSL_CTX_use_RSAPrivateKey_file
- wolfSSL_CTX_use_PrivateKey_buffer

- wolfSSL_CTX_use_PrivateKey_file
- wolfSSL_use_PrivateKey_buffer
- wolfSSL_use_PrivateKey_file

Return:

- SSL_SUCCESS upon success
- SSL_FAILURE If the function call fails, possible causes might include: The input key file is in the wrong format, or the wrong format has been given using the “format” argument, file doesn’t exist, can’t be read, or is corrupted, an out of memory condition occurs

Example

```
int ret = 0;
WOLFSSL* ssl;
...
ret = wolfSSL_use_RSAPrivateKey_file(ssl, “./server-key.pem”,
                                     SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
    // error loading private key file
}
...
```

C.52.2.43 function wolfSSL_CTX_der_load_verify_locations

```
int wolfSSL_CTX_der_load_verify_locations(
    WOLFSSL_CTX * ctx,
    const char * file,
    int format
)
```

This function is similar to wolfSSL_CTX_load_verify_locations, but allows the loading of DER-formatted CA files into the SSL context (WOLFSSL_CTX). It may still be used to load PEM-formatted CA files as well. These certificates will be treated as trusted root certificates and used to verify certs received from peers during the SSL handshake. The root certificate file, provided by the file argument, may be a single certificate or a file containing multiple certificates. If multiple CA certs are included in the same file, wolfSSL will load them in the same order they are presented in the file. The format argument specifies the format which the certificates are in either, SSL_FILETYPE_PEM or SSL_FILETYPE_ASN1 (DER). Unlike wolfSSL_CTX_load_verify_locations, this function does not allow the loading of CA certificates from a given directory path. Note that this function is only available when the wolfSSL library was compiled with WOLFSSL_DER_LOAD defined.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new()
- **file** a pointer to the name of the file containing the CA certificates to be loaded into the wolfSSL SSL context, with format as specified by format.
- **format** the encoding type of the certificates specified by file. Possible values include SSL_FILETYPE_PEM and SSL_FILETYPE_ASN1.

See:

- wolfSSL_CTX_load_verify_locations
- wolfSSL_CTX_load_verify_buffer

Return:

- SSL_SUCCESS upon success.
- SSL_FAILURE upon failure.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx;
...
ret = wolfSSL_CTX_der_load_verify_locations(ctx, "./ca-cert.der",
                                            SSL_FILETYPE_ASN1);
if (ret != SSL_SUCCESS) {
    // error loading CA certs
}
...
```

C.52.2.44 function wolfSSL_CTX_new

```
WOLFSSL_CTX * wolfSSL_CTX_new(
    WOLFSSL_METHOD *
)
```

This function creates a new SSL context, taking a desired SSL/TLS protocol method for input.

Parameters:

- **method** pointer to the desired WOLFSSL_METHOD to use for the SSL context. This is created using one of the wolfSSLvXX_XXXX_method() functions to specify SSL/TLS/DTLS protocol level.

See: wolfSSL_new

Return:

- pointer If successful the call will return a pointer to the newly-created WOLFSSL_CTX.
- NULL upon failure.

Example

```
WOLFSSL_CTX*    ctx    = 0;
WOLFSSL_METHOD* method = 0;

method = wolfSSLv3_client_method();
if (method == NULL) {
    // unable to get method
}

ctx = wolfSSL_CTX_new(method);
if (ctx == NULL) {
    // context creation failed
}
```

C.52.2.45 function wolfSSL_new

```
WOLFSSL * wolfSSL_new(
    WOLFSSL_CTX *
)
```

This function creates a new SSL session, taking an already created SSL context as input.

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().

See: wolfSSL_CTX_new

Return:

- – If successful the call will return a pointer to the newly-created wolfSSL structure.
- NULL Upon failure.

Example

```
#include <wolfssl/ssl.h>

WOLFSSL*     ssl = NULL;
WOLFSSL_CTX* ctx = 0;

ctx = wolfSSL_CTX_new(method);
if (ctx == NULL) {
    // context creation failed
}

ssl = wolfSSL_new(ctx);
if (ssl == NULL) {
    // SSL object creation failed
}
```

C.52.2.46 function wolfSSL_set_fd

```
int wolfSSL_set_fd(
    WOLFSSL * ssl,
    int fd
)
```

This function assigns a file descriptor (fd) as the input/output facility for the SSL connection. Typically this will be a socket file descriptor.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().
- **fd** file descriptor to use with SSL/TLS connection.

See:

- wolfSSL_CTX_SetIOSend
- wolfSSL_CTX_SetIORecv
- wolfSSL_SetIOReadCtx
- wolfSSL_SetIOWriteCtx

Return:

- SSL_SUCCESS upon success.
- BAD_FUNC_ARG upon failure.

Example

```
int sockfd;
WOLFSSL* ssl = 0;
...

ret = wolfSSL_set_fd(ssl, sockfd);
if (ret != SSL_SUCCESS) {
    // failed to set SSL file descriptor
}
```

C.52.2.47 function wolfSSL_set_dtls_fd_connected

```
int wolfSSL_set_dtls_fd_connected(
    WOLFSSL * ssl,
    int fd
)
```

This function assigns a file descriptor (fd) as the input/output facility for the SSL connection. Typically this will be a socket file descriptor. This is a DTLS specific API because it marks that the socket is connected. recvfrom and sendto calls on this fd will have the addr and addr_len parameters set to NULL.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().
- **fd** file descriptor to use with SSL/TLS connection.

See:

- wolfSSL_CTX_SetIOSend
- wolfSSL_CTX_SetIORecv
- wolfSSL_SetIOReadCtx
- wolfSSL_SetIOWriteCtx
- wolfDTLS_SetChGoodCb

Return:

- SSL_SUCCESS upon success.
- BAD_FUNC_ARG upon failure.

Example

```
int sockfd;
WOLFSSL* ssl = 0;
...
if (connect(sockfd, peer_addr, peer_addr_len) != 0) {
    // handle connect error
}
...
ret = wolfSSL_set_dtls_fd_connected(ssl, sockfd);
if (ret != SSL_SUCCESS) {
    // failed to set SSL file descriptor
}
```

C.52.2.48 function wolfDTLS_SetChGoodCb

```
int wolfDTLS_SetChGoodCb(
    WOLFSSL * ssl,
    ClientHelloGoodCb cb,
    void * user_ctx
)
```

Allows setting a callback for a correctly processed and verified DTLS client hello. When using a cookie exchange mechanism (either the HelloVerifyRequest in DTLS 1.2 or the HelloRetryRequest with a cookie extension in DTLS 1.3) this callback is called after the cookie exchange has succeeded. This is useful to use one WOLFSSL object as the listener for new connections and being able to isolate the WOLFSSL object once the ClientHello is verified (either through a cookie exchange or just checking if the ClientHello had the correct format). DTLS 1.2: https://datatracker.ietf.org/doc/html/rfc6347#section-4.2.1 DTLS 1.3: https://www.rfc-editor.org/rfc/rfc8446#section-4.2.2.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().
- **fd** file descriptor to use with SSL/TLS connection.

See: wolfSSL_set_dtls_fd_connected

Return:

- SSL_SUCCESS upon success.
- BAD_FUNC_ARG upon failure.

Example

```
// Called when we have verified a connection
static int chGoodCb(WOLFSSL* ssl, void* arg)
{
    // setup peer and file descriptors

}

if (wolfDTLS_SetChGoodCb(ssl, chGoodCb, NULL) != WOLFSSL_SUCCESS) {
     // error setting callback
}
```

C.52.2.49 function wolfSSL_get_cipher_list

```
char * wolfSSL_get_cipher_list(
    int priority
)
```

Get the name of cipher at priority level passed in.

Parameters:

- **priority** Integer representing the priority level of a cipher.

See:

- wolfSSL_CIPHER_get_name
- wolfSSL_get_current_cipher

Return:

- string Success
- 0 Priority is either out of bounds or not valid.

Example

```
printf("The cipher at 1 is %s", wolfSSL_get_cipher_list(1));
```

C.52.2.50 function wolfSSL_get_ciphers

```
int wolfSSL_get_ciphers(
    char * buf,
    int len
)
```

This function gets the ciphers enabled in wolfSSL.

Parameters:

- **buf** a char pointer representing the buffer.

- **len** the length of the buffer.

See:

- GetCipherNames
- wolfSSL_get_cipher_list
- ShowCiphers

Return:

- SSL_SUCCESS returned if the function executed without error.
- BAD_FUNC_ARG returned if the buf parameter was NULL or if the len argument was less than or equal to zero.
- BUFFER_E returned if the buffer is not large enough and will overflow.

Example

```
static void ShowCiphers(void){
char* ciphers;
int ret = wolfSSL_get_ciphers(ciphers, (int)sizeof(ciphers));

if(ret == SSL_SUCCES){
        printf("%s\n", ciphers);
    }
}
```

C.52.2.51 function wolfSSL_get_cipher_name

```
const char * wolfSSL_get_cipher_name(
    WOLFSSL * ssl
)
```

This function gets the cipher name in the format DHE-RSA by passing through argument to wolfSSL_get_cipher_name_internal.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_CIPHER_get_name
- wolfSSL_get_current_cipher
- wolfSSL_get_cipher_name_internal

Return:

- string This function returns the string representation of the cipher suite that was matched.
- NULL error or cipher not found.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
WOLFSSL* ssl = wolfSSL_new(ctx);
…
char* cipherS = wolfSSL_get_cipher_name(ssl);

if(cipher == NULL){
    // There was not a cipher suite matched
} else {
    // There was a cipher suite matched
```

```
    printf("%s\n", cipherS);
}
```

C.52.2.52 function wolfSSL_get_fd

```
int wolfSSL_get_fd(
    const WOLFSSL *
)
```

This function returns the read file descriptor (fd) used as the input facility for the SSL connection. Typically this will be a socket file descriptor.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().

See:

- wolfSSL_set_fd
- wolfSSL_set_read_fd
- wolfSSL_set_write_fd

Return: fd If successful the call will return the SSL session file descriptor.

Example

```
int sockfd;
WOLFSSL* ssl = 0;
...
sockfd = wolfSSL_get_fd(ssl);
...
```

C.52.2.53 function wolfSSL_get_wfd

```
int wolfSSL_get_wfd(
    const WOLFSSL *
)
```

This function returns the write file descriptor (fd) used as the output facility for the SSL connection. Typically this will be a socket file descriptor.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().

See:

- wolfSSL_set_fd
- wolfSSL_set_read_fd
- wolfSSL_set_write_fd

Return: fd If successful the call will return the SSL session file descriptor.

Example

```
int sockfd;
WOLFSSL* ssl = 0;
...
sockfd = wolfSSL_get_wfd(ssl);
...
```

C.52.2.54 function wolfSSL_set_using_nonblock

```
void wolfSSL_set_using_nonblock(
    WOLFSSL * ssl,
    int nonblock
)
```

This function informs the WOLFSSL object that the underlying I/O is non-blocking. After an application creates a WOLFSSL object, if it will be used with a non-blocking socket, call wolfSSL_set_using_nonblock() on it. This lets the WOLFSSL object know that receiving EWOULDBLOCK means that the recvfrom call would block rather than that it timed out.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().
- **nonblock** value used to set non-blocking flag on WOLFSSL object. Use 1 to specify non-blocking, otherwise 0.

See:

- wolfSSL_get_using_nonblock
- wolfSSL_dtls_got_timeout
- wolfSSL_dtls_get_current_timeout

Return: none No return.

Example

```
WOLFSSL* ssl = 0;
...
wolfSSL_set_using_nonblock(ssl, 1);
```

C.52.2.55 function wolfSSL_get_using_nonblock

```
int wolfSSL_get_using_nonblock(
    WOLFSSL *
)
```

This function allows the application to determine if wolfSSL is using non-blocking I/O. If wolfSSL is using non-blocking I/O, this function will return 1, otherwise 0. After an application creates a WOLFSSL object, if it will be used with a non-blocking socket, call wolfSSL_set_using_nonblock() on it. This lets the WOLFSSL object know that receiving EWOULDBLOCK means that the recvfrom call would block rather than that it timed out.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().

See: wolfSSL_set_session

Return:

- 0 underlying I/O is blocking.
- 1 underlying I/O is non-blocking.

Example

```
int ret = 0;
WOLFSSL* ssl = 0;
...
ret = wolfSSL_get_using_nonblock(ssl);
if (ret == 1) {
    // underlying I/O is non-blocking
```

```
}
...
```

C.52.2.56 function wolfSSL_write

```
int wolfSSL_write(
    WOLFSSL * ssl,
    const void * data,
    int sz
)
```

This function writes sz bytes from the buffer, data, to the SSL connection, ssl. If necessary, wolfSSL_write() will only return once the buffer data of size sz has been completely written or an error occurred.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().
- **data** data buffer which will be sent to peer.
- **sz** size, in bytes, of data to send to the peer (data).

See:

- wolfSSL_send
- wolfSSL_read
- wolfSSL_recv

Return:

- 0 the number of bytes written upon success.
- 0 will be returned upon failure. Call wolfSSL_get_error() for the specific error code.
- SSL_FATAL_ERROR will be returned upon failure when either an error occurred or, when using non-blocking sockets, the SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE error was received and and the application needs to call wolfSSL_write() to get a specific error code.

Example

```
WOLFSSL* ssl = 0;
char msg[64] = “hello wolfssl!”;
int msgSz = (int)strlen(msg);
int flags;
int ret;
...

ret = wolfSSL_write(ssl, msg, msgSz);
if (ret <= 0) {
    // wolfSSL_write() failed, call wolfSSL_get_error()
}
```

C.52.2.57 function wolfSSL_read

```
int wolfSSL_read(
    WOLFSSL * ssl,
    void * data,
    int sz
)
```

This function reads sz bytes from the SSL session (ssl) internal read buffer into the buffer data. The bytes read are removed from the internal receive buffer. If necessary wolfSSL_read() will trigger processing of the next record.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().
- **data** buffer where wolfSSL_read() will place data read.
- **sz** number of bytes to read into data.

See:

- wolfSSL_recv
- wolfSSL_write
- wolfSSL_peek
- wolfSSL_pending

Return:

- 0 the number of bytes read upon success.
- 0 will be returned upon failure. This may be caused by a either a clean (close notify alert) shutdown or just that the peer closed the connection. Call wolfSSL_get_error() for the specific error code.
- SSL_FATAL_ERROR will be returned upon failure when either an error occurred or, when using nonblocking sockets, the SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE error was received and and the application needs to call wolfSSL_read() to get a specific error code.

Example

```
WOLFSSL* ssl = 0;
char reply[1024];
...

input = wolfSSL_read(ssl, reply, sizeof(reply));
if (input > 0) {
    // “input” number of bytes returned into buffer “reply”
}

See wolfSSL examples (client, server, echoclient, echoserver) for more
complete examples of wolfSSL_read().
```

C.52.2.58 function wolfSSL_peek

```
int wolfSSL_peek(
    WOLFSSL * ssl,
    void * data,
    int sz
)
```

This function copies sz bytes from the SSL session (ssl) internal read buffer into the buffer data. This function is identical to wolfSSL_read() will trigger processing of the next record.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().
- **data** buffer where wolfSSL_peek() will place data read.
- **sz** number of bytes to read into data.

See: wolfSSL_read

Return:

- 0 the number of bytes read upon success.
- 0 will be returned upon failure. This may be caused by a either a clean (close notify alert) shutdown or just that the peer closed the connection. Call wolfSSL_get_error() for the specific error code.
- SSL_FATAL_ERROR will be returned upon failure when either an error occurred or, when using nonblocking sockets, the SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE error was received and and the application needs to call wolfSSL_peek() to get a specific error code.

Example

```
WOLFSSL* ssl = 0;
char reply[1024];
...

input = wolfSSL_peek(ssl, reply, sizeof(reply));
if (input > 0) {
    // "input" number of bytes returned into buffer "reply"
}
```

C.52.2.59 function wolfSSL_accept

```
int wolfSSL_accept(
    WOLFSSL *
)
```

This function is called on the server side and waits for an SSL client to initiate the SSL/TLS handshake. When this function is called, the underlying communication channel has already been set up. wolfSSL_accept() will only return once the handshake has been finished or an error occurred.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_get_error
- wolfSSL_connect

Return:

- SSL_SUCCESS upon success.
- SSL_FATAL_ERROR will be returned if an error occurred. To get a more detailed error code, call wolfSSL_get_error().

Example

```
int ret = 0;
int err = 0;
WOLFSSL* ssl;
char buffer[80];
...

ret = wolfSSL_accept(ssl);
if (ret != SSL_SUCCESS) {
    err = wolfSSL_get_error(ssl, ret);
```

```
    printf("error = %d, %s\n", err, wolfSSL_ERR_error_string(err, buffer));
}
```

C.52.2.60 function wolfDTLS_accept_stateless

```
int wolfDTLS_accept_stateless(
    WOLFSSL * ssl
)
```

This function is called on the server side and statelessly listens for an SSL client to initiate the DTLS handshake.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_accept
- wolfSSL_get_error
- wolfSSL_connect

Return:

- WOLFSSL_SUCCESS ClientHello containing a valid cookie was received. The connection can be continued with wolfSSL_accept().
- WOLFSSL_FAILURE The I/O layer returned WANT_READ. This is either because there is no data to read and we are using non-blocking sockets or we sent a cookie request and we are waiting for a reply. The user should call wolfDTLS_accept_stateless again after data becomes available in the I/O layer.
- WOLFSSL_FATAL_ERROR A fatal error occurred. The ssl object should be free'd and allocated again to continue.

Example

```
int ret = 0;
int err = 0;
WOLFSSL* ssl;
...
do {
    ret = wolfDTLS_accept_stateless(ssl);
    if (ret == WOLFSSL_FATAL_ERROR)
        // re-allocate the ssl object with wolfSSL_free() and wolfSSL_new()
} while (ret != WOLFSSL_SUCCESS);
ret = wolfSSL_accept(ssl);
if (ret != SSL_SUCCESS) {
    err = wolfSSL_get_error(ssl, ret);
    printf("error = %d, %s\n", err, wolfSSL_ERR_error_string(err, buffer));
}
```

C.52.2.61 function wolfSSL_CTX_free

```
void wolfSSL_CTX_free(
    WOLFSSL_CTX *
)
```

This function frees an allocated WOLFSSL_CTX object. This function decrements the CTX reference count and only frees the context when the reference count has reached 0.

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().

See:

- wolfSSL_CTX_new
- wolfSSL_new
- wolfSSL_free

Return: none No return.

Example

```
WOLFSSL_CTX* ctx = 0;
...
wolfSSL_CTX_free(ctx);
```

C.52.2.62 function wolfSSL_free

```
void wolfSSL_free(
    WOLFSSL *
)
```

This function frees an allocated wolfSSL object.

Parameters:

- **ssl** pointer to the SSL object, created with wolfSSL_new().

See:

- wolfSSL_CTX_new
- wolfSSL_new
- wolfSSL_CTX_free

Return: none No return.

Example

```
#include <wolfssl/ssl.h>

WOLFSSL* ssl = 0;
...
wolfSSL_free(ssl);
```

C.52.2.63 function wolfSSL_shutdown

```
int wolfSSL_shutdown(
    WOLFSSL *
)
```

This function shuts down an active SSL/TLS connection using the SSL session, ssl. This function will try to send a “close notify” alert to the peer. The calling application can choose to wait for the peer to send its “close notify” alert in response or just go ahead and shut down the underlying connection after directly calling wolfSSL_shutdown (to save resources). Either option is allowed by the TLS specification. If the underlying connection will be used again in the future, the complete two-directional shutdown procedure must be performed to keep synchronization intact between the peers. wolfSSL_shutdown() when the underlying I/O is ready.

Parameters:

- **ssl** pointer to the SSL session created with wolfSSL_new().

See:

- wolfSSL_free
- wolfSSL_CTX_free

Return:

- SSL_SUCCESS will be returned upon success.
- SSL_SHUTDOWN_NOT_DONE will be returned when shutdown has not finished, and the function should be called again.
- SSL_FATAL_ERROR will be returned upon failure. Call wolfSSL_get_error() for a more specific error code.

Example

```
#include <wolfssl/ssl.h>

int ret = 0;
WOLFSSL* ssl = 0;
...
ret = wolfSSL_shutdown(ssl);
if (ret != 0) {
    // failed to shut down SSL connection
}
```

C.52.2.64 function wolfSSL_send

```
int wolfSSL_send(
    WOLFSSL * ssl,
    const void * data,
    int sz,
    int flags
)
```

This function writes sz bytes from the buffer, data, to the SSL connection, ssl, using the specified flags for the underlying write operation. If necessary wolfSSL_send() will only return once the buffer data of size sz has been completely written or an error occurred.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().
- **data** data buffer to send to peer.
- **sz** size, in bytes, of data to be sent to peer.
- **flags** the send flags to use for the underlying send operation.

See:

- wolfSSL_write
- wolfSSL_read
- wolfSSL_recv

Return:

- 0 the number of bytes written upon success.
- 0 will be returned upon failure. Call wolfSSL_get_error() for the specific error code.
- SSL_FATAL_ERROR will be returned upon failure when either an error occurred or, when using non-blocking sockets, the SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE error was received and and the application needs to call wolfSSL_send() to get a specific error code.

Example

```
WOLFSSL* ssl = 0;
char msg[64] = “hello wolfssl!”;
int msgSz = (int)strlen(msg);
int flags = ... ;
...

input = wolfSSL_send(ssl, msg, msgSz, flags);
if (input != msgSz) {
    // wolfSSL_send() failed
}
```

C.52.2.65 function wolfSSL_recv

```
int wolfSSL_recv(
    WOLFSSL * ssl,
    void * data,
    int sz,
    int flags
)
```

This function reads sz bytes from the SSL session (ssl) internal read buffer into the buffer data using the specified flags for the underlying recv operation. The bytes read are removed from the internal receive buffer. This function is identical to wolfSSL_read() will trigger processing of the next record.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().
- **data** buffer where wolfSSL_recv() will place data read.
- **sz** number of bytes to read into data.
- **flags** the recv flags to use for the underlying recv operation.

See:

- wolfSSL_read
- wolfSSL_write
- wolfSSL_peek
- wolfSSL_pending

Return:

- 0 the number of bytes read upon success.
- 0 will be returned upon failure. This may be caused by a either a clean (close notify alert) shutdown or just that the peer closed the connection. Call wolfSSL_get_error() for the specific error code.
- SSL_FATAL_ERROR will be returned upon failure when either an error occurred or, when using nonblocking sockets, the SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE error was received and and the application needs to call wolfSSL_recv() to get a specific error code.

Example

```
WOLFSSL* ssl = 0;
char reply[1024];
int flags = ... ;
...

input = wolfSSL_recv(ssl, reply, sizeof(reply), flags);
if (input > 0) {
```

```
    // “input” number of bytes returned into buffer “reply”
}
```

C.52.2.66 function wolfSSL_get_error

```
int wolfSSL_get_error(
    WOLFSSL * ssl,
    int ret
)
```

This function returns a unique error code describing why the previous API function call (wolfSSL_connect, wolfSSL_accept, wolfSSL_read, wolfSSL_write, etc.) resulted in an error return code (SSL_FAILURE). The return value of the previous function is passed to wolfSSL_get_error through ret. After wolfSSL_get_error is called and returns the unique error code, wolfSSL_ERR_error_string() for more information.

Parameters:

- **ssl** pointer to the SSL object, created with wolfSSL_new().
- **ret** return value of the previous function that resulted in an error return code.

See:

- wolfSSL_ERR_error_string
- wolfSSL_ERR_error_string_n
- wolfSSL_ERR_print_errors_fp
- wolfSSL_load_error_strings

Return:

- On successful completion, this function will return the unique error code describing why the previous API function failed.
- SSL_ERROR_NONE will be returned if ret > 0. For ret <= 0, there are some cases when this value can also be returned when a previous API appeared to return an error code but no error actually occurred. An example is calling wolfSSL_read() is called afterwards, SSL_ERROR_NONE will be returned.

Example

```
int err = 0;
WOLFSSL* ssl;
char buffer[80];
...
err = wolfSSL_get_error(ssl, 0);
wolfSSL_ERR_error_string(err, buffer);
printf(“err = %d, %s\n”, err, buffer);
```

C.52.2.67 function wolfSSL_get_alert_history

```
int wolfSSL_get_alert_history(
    WOLFSSL * ssl,
    WOLFSSL_ALERT_HISTORY * h
)
```

This function gets the alert history.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

- **h** a pointer to a WOLFSSL_ALERT_HISTORY structure that will hold the WOLFSSL struct's alert_history member's value.

See: wolfSSL_get_error

Return: SSL_SUCCESS returned when the function completed successfully. Either there was alert history or there wasn't, either way, the return value is SSL_SUCCESS.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(protocol method);
WOLFSSL* ssl = wolfSSL_new(ctx);
WOLFSSL_ALERT_HISTORY* h;
...
wolfSSL_get_alert_history(ssl, h);
// h now has a copy of the ssl->alert_history  contents
```

C.52.2.68 function wolfSSL_set_session

```
int wolfSSL_set_session(
    WOLFSSL * ssl,
    WOLFSSL_SESSION * session
)
```

This function sets the session to be used when the SSL object, ssl, is used to establish a SSL/TLS connection. For session resumption, before calling wolfSSL_shutdown() needs to be freed after the application is done with it by calling wolfSSL_SESSION_free() on it.

Parameters:

- **ssl** pointer to the SSL object, created with wolfSSL_new().
- **session** pointer to the WOLFSSL_SESSION used to set the session for ssl.

See: wolfSSL_get1_session

Return:

- SSL_SUCCESS will be returned upon successfully setting the session.
- SSL_FAILURE will be returned on failure. This could be caused by the session cache being disabled, or if the session has timed out.
- When OPENSSL_EXTRA and WOLFSSL_ERROR_CODE_OPENSSL are defined, SSL_SUCCESS will be returned even if the session has timed out.

Example

```
int ret;
WOLFSSL* ssl;
WOLFSSL_SESSION* session;
...
session = wolfSSL_get1_session(ssl);
if (session == NULL) {
    // failed to get session object from ssl object
}
...
ret = wolfSSL_set_session(ssl, session);
if (ret != SSL_SUCCESS) {
    // failed to set the SSL session
}
wolfSSL_SESSION_free(session);
...
```

C.52.2.69 function wolfSSL_get_session

```
WOLFSSL_SESSION * wolfSSL_get_session(
    WOLFSSL * ssl
)
```

When NO_SESSION_CACHE_REF is defined this function returns a pointer to the current session (WOLFSSL_SESSION) used in ssl. This function returns a non-persistent pointer to the WOLFSSL_SESSION object. The pointer returned will be freed when wolfSSL_free is called. This call should only be used to inspect or modify the current session. For session resumption it is recommended to use wolfSSL_get1_session() for session resumption.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().

See:

- wolfSSL_get1_session
- wolfSSL_set_session

Return:

- pointer If successful the call will return a pointer to the the current SSL session object.
- NULL will be returned if ssl is NULL, the SSL session cache is disabled, wolfSSL doesn't have the Session ID available, or mutex functions fail.

Example

```
WOLFSSL* ssl;
WOLFSSL_SESSION* session;
...
session = wolfSSL_get_session(ssl);
if (session == NULL) {
    // failed to get session pointer
}
...
```

C.52.2.70 function wolfSSL_flush_sessions

```
void wolfSSL_flush_sessions(
    WOLFSSL_CTX * ctx,
    long tm
)
```

This function flushes session from the session cache which have expired. The time, tm, is used for the time comparison. Note that wolfSSL currently uses a static table for sessions, so no flushing is needed. As such, this function is currently just a stub. This function provides OpenSSL compatibility (SSL_flush_sessions) when wolfSSL is compiled with the OpenSSL compatibility layer.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().
- **tm** time used in session expiration comparison.

See:

- wolfSSL_get1_session
- wolfSSL_set_session

Return: none No returns.

Example

```
WOLFSSL_CTX* ssl;
...
wolfSSL_flush_sessions(ctx, time(0));
```

C.52.2.71 function wolfSSL_SetServerID

```
int wolfSSL_SetServerID(
    WOLFSSL * ssl,
    const unsigned char * id,
    int len,
    int newSession
)
```

This function associates the client session with the server id. If the newSession flag is on, an existing session won't be reused.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **id** a constant byte pointer that will be copied to the serverID member of the WOLFSSL_SESSION structure.
- **len** an int type representing the length of the session id parameter.
- **newSession** an int type representing the flag to denote whether to reuse a session or not.

See: wolfSSL_set_session

Return:

- SSL_SUCCESS returned if the function executed without error.
- BAD_FUNC_ARG returned if the WOLFSSL struct or id parameter is NULL or if len is not greater than zero.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol );
WOLFSSL* ssl = WOLFSSL_new(ctx);
const byte id[MAX_SIZE];  // or dynamically create space
int len = 0; // initialize length
int newSession = 0; // flag to allow
…
int ret = wolfSSL_SetServerID(ssl, id, len, newSession);

if (ret == WOLFSSL_SUCCESS) {
    // The Id was successfully set
}
```

C.52.2.72 function wolfSSL_GetSessionIndex

```
int wolfSSL_GetSessionIndex(
    WOLFSSL * ssl
)
```

This function gets the session index of the WOLFSSL structure.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See: wolfSSL_GetSessionAtIndex

Return: int The function returns an int type representing the sessionIndex within the WOLFSSL struct.

Example

```
WOLFSSL_CTX_new( protocol method );
WOLFSSL* ssl = WOLFSSL_new(ctx);
...
int sesIdx = wolfSSL_GetSessionIndex(ssl);

if(sesIdx < 0 || sesIdx > sizeof(ssl->sessionIndex)/sizeof(int)){
    // You have an out of bounds index number and something is not right.
}
```

C.52.2.73 function wolfSSL_GetSessionAtIndex

```
int wolfSSL_GetSessionAtIndex(
    int idx,
    WOLFSSL_SESSION * session
)
```

This function gets the session at specified index of the session cache and copies it into memory. The WOLFSSL_SESSION structure holds the session information.

Parameters:

- **idx** an int type representing the session index.
- **session** a pointer to the WOLFSSL_SESSION structure.

See:

- UnLockMutex
- LockMutex
- wolfSSL_GetSessionIndex

Return:

- SSL_SUCCESS returned if the function executed successfully and no errors were thrown.
- BAD_MUTEX_E returned if there was an unlock or lock mutex error.
- SSL_FAILURE returned if the function did not execute successfully.

Example

```
int idx; // The index to locate the session.
WOLFSSL_SESSION* session;  // Buffer to copy to.
...
if(wolfSSL_GetSessionAtIndex(idx, session) != SSL_SUCCESS){
    // Failure case.
}
```

C.52.2.74 function wolfSSL_SESSION_get_peer_chain

```
WOLFSSL_X509_CHAIN * wolfSSL_SESSION_get_peer_chain(
    WOLFSSL_SESSION * session
)
```

Returns the peer certificate chain from the WOLFSSL_SESSION struct.

Parameters:

- **session** a pointer to a WOLFSSL_SESSION structure.

See:

- wolfSSL_GetSessionAtIndex

- wolfSSL_GetSessionIndex
- AddSession

Return: pointer A pointer to a WOLFSSL_X509_CHAIN structure that contains the peer certification chain.

Example

```
WOLFSSL_SESSION* session;
WOLFSSL_X509_CHAIN* chain;
...
chain = wolfSSL_SESSION_get_peer_chain(session);
if(!chain){
    // There was no chain. Failure case.
}
```

C.52.2.75 function wolfSSL_CTX_set_verify

```
void wolfSSL_CTX_set_verify(
    WOLFSSL_CTX * ctx,
    int mode,
    VerifyCallback verify_callback
)
```

This function sets the verification method for remote peers and also allows a verify callback to be registered with the SSL context. The verify callback will be called only when a verification failure has occurred. If no verify callback is desired, the NULL pointer can be used for verify_callback. The verification mode of peer certificates is a logically OR'd list of flags. The possible flag values include: SSL_VERIFY_NONE Client mode: the client will not verify the certificate received from the server and the handshake will continue as normal. Server mode: the server will not send a certificate request to the client. As such, client verification will not be enabled. SSL_VERIFY_PEER Client mode: the client will verify the certificate received from the server during the handshake. This is turned on by default in wolfSSL, therefore, using this option has no effect. Server mode: the server will send a certificate request to the client and verify the client certificate received. SSL_VERIFY_FAIL_IF_NO_PEER_CERT Client mode: no effect when used on the client side. Server mode: the verification will fail on the server side if the client fails to send a certificate when requested to do so (when using SSL_VERIFY_PEER on the SSL server). SSL_VERIFY_FAIL_EXCEPT_PSK Client mode: no effect when used on the client side. Server mode: the verification is the same as SSL_VERIFY_FAIL_IF_NO_PEER_CERT except in the case of a PSK connection. If a PSK connection is being made then the connection will go through without a peer cert.

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().
- **mode** flags indicating verification mode for peer's cert.
- **verify_callback** callback to be called when verification fails. If no callback is desired, the NULL pointer can be used for verify_callback.

See: wolfSSL_set_verify

Return: none No return.

Example

```
WOLFSSL_CTX*    ctx    = 0;
...
wolfSSL_CTX_set_verify(ctx, (WOLFSSL_VERIFY_PEER |
                        WOLFSSL_VERIFY_FAIL_IF_NO_PEER_CERT), NULL);
```

C.52.2.76 function wolfSSL_set_verify

```
void wolfSSL_set_verify(
    WOLFSSL * ssl,
    int mode,
    VerifyCallback verify_callback
)
```

This function sets the verification method for remote peers and also allows a verify callback to be registered with the SSL session. The verify callback will be called only when a verification failure has occurred. If no verify callback is desired, the NULL pointer can be used for verify_callback. The verification mode of peer certificates is a logically OR'd list of flags. The possible flag values include: SSL_VERIFY_NONE Client mode: the client will not verify the certificate received from the server and the handshake will continue as normal. Server mode: the server will not send a certificate request to the client. As such, client verification will not be enabled. SSL_VERIFY_PEER Client mode: the client will verify the certificate received from the server during the handshake. This is turned on by default in wolfSSL, therefore, using this option has no effect. Server mode: the server will send a certificate request to the client and verify the client certificate received. SSL_VERIFY_FAIL_IF_NO_PEER_CERT Client mode: no effect when used on the client side. Server mode: the verification will fail on the server side if the client fails to send a certificate when requested to do so (when using SSL_VERIFY_PEER on the SSL server). SSL_VERIFY_FAIL_EXCEPT_PSK Client mode: no effect when used on the client side. Server mode: the verification is the same as SSL_VERIFY_FAIL_IF_NO_PEER_CERT except in the case of a PSK connection. If a PSK connection is being made then the connection will go through without a peer cert.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().
- **mode** flags indicating verification mode for peer's cert.
- **verify_callback** callback to be called when verification fails. If no callback is desired, the NULL pointer can be used for verify_callback.

See: wolfSSL_CTX_set_verify

Return: none No return.

Example

```
WOLFSSL* ssl = 0;
...
wolfSSL_set_verify(ssl, SSL_VERIFY_PEER | SSL_VERIFY_FAIL_IF_NO_PEER_CERT, 0);
```

C.52.2.77 function wolfSSL_SetCertCbCtx

```
void wolfSSL_SetCertCbCtx(
    WOLFSSL * ssl,
    void * ctx
)
```

This function stores user CTX object information for verify callback.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **ctx** a void pointer that is set to WOLFSSL structure's verifyCbCtx member's value.

See:

- wolfSSL_CTX_save_cert_cache
- wolfSSL_CTX_restore_cert_cache
- wolfSSL_CTX_set_verify

Return: none No return.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
(void*)ctx;
...
if(ssl != NULL){
wolfSSL_SetCertCbCtx(ssl, ctx);
} else {
    // Error case, the SSL is not initialized properly.
}
```

C.52.2.78 function wolfSSL_CTX_SetCertCbCtx

```
void wolfSSL_CTX_SetCertCbCtx(
    WOLFSSL_CTX * ctx,
    void * userCtx
)
```

This function stores user CTX object information for verify callback.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure.
- **userCtx** a void pointer that is used to set WOLFSSL_CTX structure's verifyCbCtx member's value.

See:

- wolfSSL_CTX_save_cert_cache
- wolfSSL_CTX_restore_cert_cache
- wolfSSL_CTX_set_verify

Return: none No return.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
void* userCtx = NULL; // Assign some user defined context
...
if(ctx != NULL){
    wolfSSL_SetCertCbCtx(ctx, userCtx);
} else {
    // Error case, the SSL is not initialized properly.
}
```

C.52.2.79 function wolfSSL_pending

```
int wolfSSL_pending(
    WOLFSSL *
)
```

This function returns the number of bytes which are buffered and available in the SSL object to be read by wolfSSL_read().

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().

See:

- wolfSSL_recv
- wolfSSL_read
- wolfSSL_peek

Return: int This function returns the number of bytes pending.

Example

```
int pending = 0;
WOLFSSL* ssl = 0;
...

pending = wolfSSL_pending(ssl);
printf("There are %d bytes buffered and available for reading", pending);
```

C.52.2.80 function wolfSSL_load_error_strings

```
void wolfSSL_load_error_strings(
    void
)
```

This function is for OpenSSL compatibility (SSL_load_error_string) only and takes no action.

Parameters:

- **none** No parameters.

See:

- wolfSSL_get_error
- wolfSSL_ERR_error_string
- wolfSSL_ERR_error_string_n
- wolfSSL_ERR_print_errors_fp
- wolfSSL_load_error_strings

Return: none No returns.

Example

```
wolfSSL_load_error_strings();
```

C.52.2.81 function wolfSSL_library_init

```
int wolfSSL_library_init(
    void
)
```

This function is called internally in wolfSSL_CTX_new() is the more typically-used wolfSSL initialization function.

Parameters:

- **none** No parameters.

See:

- wolfSSL_Init
- wolfSSL_Cleanup

Return:

- SSL_SUCCESS If successful the call will return.
- SSL_FATAL_ERROR is returned upon failure.

Example

```
int ret = 0;
ret = wolfSSL_library_init();
if (ret != SSL_SUCCESS) {
    failed to initialize wolfSSL
}
...
```

C.52.2.82 function wolfSSL_SetDevId

```
int wolfSSL_SetDevId(
    WOLFSSL * ssl,
    int devId
)
```

This function sets the Device Id at the WOLFSSL session level.

Parameters:

- **ssl** pointer to a SSL object, created with wolfSSL_new().
- **devId** ID to use with crypto callbacks or async hardware. Set to INVALID_DEVID (-2) if not used

See:

- wolfSSL_CTX_SetDevId
- wolfSSL_CTX_GetDevId

Return:

- WOLFSSL_SUCCESS upon success.
- BAD_FUNC_ARG if ssl is NULL.

Example

```
WOLFSSL* ssl;
int DevId = -2;

wolfSSL_SetDevId(ssl, devId);
```

C.52.2.83 function wolfSSL_CTX_SetDevId

```
int wolfSSL_CTX_SetDevId(
    WOLFSSL_CTX * ctx,
    int devId
)
```

This function sets the Device Id at the WOLFSSL_CTX context level.

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().
- **devId** ID to use with crypto callbacks or async hardware. Set to INVALID_DEVID (-2) if not used

See:

- wolfSSL_SetDevId
- wolfSSL_CTX_GetDevId

Return:

- WOLFSSL_SUCCESS upon success.
- BAD_FUNC_ARG if ssl is NULL.

Example

```
WOLFSSL_CTX* ctx;
int DevId = -2;

wolfSSL_CTX_SetDevId(ctx, devId);
```

C.52.2.84 function wolfSSL_CTX_GetDevId

```
int wolfSSL_CTX_GetDevId(
    WOLFSSL_CTX * ctx,
    WOLFSSL * ssl
)
```

This function retrieves the Device Id.

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().
- **ssl** pointer to a SSL object, created with wolfSSL_new().

See:

- wolfSSL_SetDevId
- wolfSSL_CTX_SetDevId

Return:

- devId upon success.
- INVALID_DEVID if both ssl and ctx are NULL.

Example

```
WOLFSSL_CTX* ctx;

wolfSSL_CTX_GetDevId(ctx, ssl);
```

C.52.2.85 function wolfSSL_CTX_set_session_cache_mode

```
long wolfSSL_CTX_set_session_cache_mode(
    WOLFSSL_CTX * ctx,
    long mode
)
```

This function enables or disables SSL session caching. Behavior depends on the value used for mode. The following values for mode are available: SSL_SESS_CACHE_OFF- disable session caching. Session caching is turned on by default. SSL_SESS_CACHE_NO_AUTO_CLEAR - Disable auto-flushing of the session cache. Auto-flushing is turned on by default.

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().
- **mode** modifier used to change behavior of the session cache.

See:

- wolfSSL_flush_sessions
- wolfSSL_get1_session
- wolfSSL_set_session
- wolfSSL_get_sessionID
- wolfSSL_CTX_set_timeout

Return: SSL_SUCCESS will be returned upon success.

Example

```
WOLFSSL_CTX* ctx = 0;
...
ret = wolfSSL_CTX_set_session_cache_mode(ctx, SSL_SESS_CACHE_OFF);
if (ret != SSL_SUCCESS) {
    // failed to turn SSL session caching off
}
```

C.52.2.86 function wolfSSL_set_session_secret_cb

```
int wolfSSL_set_session_secret_cb(
    WOLFSSL * ssl,
    SessionSecretCb cb,
    void * ctx
)
```

This function sets the session secret callback function. The SessionSecretCb type has the signature: int (*SessionSecretCb)(WOLFSSL* ssl, void* secret, int* secretSz, void* ctx). The sessionSecretCb member of the WOLFSSL struct is set to the parameter cb.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **cb** a SessionSecretCb type that is a function pointer with the above signature.
- **ctx** a pointer to the user context to be stored

See: SessionSecretCb

Return:

- SSL_SUCCESS returned if the execution of the function did not return an error.
- SSL_FATAL_ERROR returned if the WOLFSSL structure is NULL.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
// Signature of SessionSecretCb
int SessionSecretCB (WOLFSSL* ssl, void* secret, int* secretSz,
void* ctx) = SessionSecretCb;
…
int wolfSSL_set_session_secret_cb(ssl, SessionSecretCB, (void*)ssl->ctx){
    // Function body.
}
```

C.52.2.87 function wolfSSL_save_session_cache

```
int wolfSSL_save_session_cache(
    const char * fname
)
```

This function persists the session cache to file. It doesn't use memsave because of additional memory use.

Parameters:

- **fname** is a constant char pointer that points to a file for writing.

See:

- XFWRITE
- wolfSSL_restore_session_cache
- wolfSSL_memrestore_session_cache

Return:

- SSL_SUCCESS returned if the function executed without error. The session cache has been written to a file.
- SSL_BAD_FILE returned if fname cannot be opened or is otherwise corrupt.
- FWRITE_ERROR returned if XFWRITE failed to write to the file.
- BAD_MUTEX_E returned if there was a mutex lock failure.

Example

```
const char* fname;
...
if(wolfSSL_save_session_cache(fname) != SSL_SUCCESS){
    // Fail to write to file.
}
```

C.52.2.88 function wolfSSL_restore_session_cache

```
int wolfSSL_restore_session_cache(
    const char * fname
)
```

This function restores the persistent session cache from file. It does not use memstore because of additional memory use.

Parameters:

- **fname** a constant char pointer file input that will be read.

See:

- XFREAD
- XFOPEN

Return:

- SSL_SUCCESS returned if the function executed without error.
- SSL_BAD_FILE returned if the file passed into the function was corrupted and could not be opened by XFOPEN.
- FREAD_ERROR returned if the file had a read error from XFREAD.
- CACHE_MATCH_ERROR returned if the session cache header match failed.
- BAD_MUTEX_E returned if there was a mutex lock failure.

Example

```
const char *fname;
...
if(wolfSSL_restore_session_cache(fname) != SSL_SUCCESS){
    // Failure case. The function did not return SSL_SUCCESS.
}
```

C.52.2.89 function wolfSSL_memsave_session_cache

```
int wolfSSL_memsave_session_cache(
    void * mem,
```

```
    int sz
)
```

This function persists session cache to memory.

Parameters:

- **mem** a void pointer representing the destination for the memory copy, XMEMCPY().
- **sz** an int type representing the size of mem.

See:

- XMEMCPY
- wolfSSL_get_session_cache_memsize

Return:

- SSL_SUCCESS returned if the function executed without error. The session cache has been successfully persisted to memory.
- BAD_MUTEX_E returned if there was a mutex lock error.
- BUFFER_E returned if the buffer size was too small.

Example

```
void* mem;
int sz; // Max size of the memory buffer.
…
if(wolfSSL_memsave_session_cache(mem, sz) != SSL_SUCCESS){
    // Failure case, you did not persist the session cache to memory
}
```

C.52.2.90 function wolfSSL_memrestore_session_cache

```
int wolfSSL_memrestore_session_cache(
    const void * mem,
    int sz
)
```

This function restores the persistent session cache from memory.

Parameters:

- **mem** a constant void pointer containing the source of the restoration.
- **sz** an integer representing the size of the memory buffer.

See: wolfSSL_save_session_cache

Return:

- SSL_SUCCESS returned if the function executed without an error.
- BUFFER_E returned if the memory buffer is too small.
- BAD_MUTEX_E returned if the session cache mutex lock failed.
- CACHE_MATCH_ERROR returned if the session cache header match failed.

Example

```
const void* memoryFile;
int szMf;
...
if(wolfSSL_memrestore_session_cache(memoryFile, szMf) != SSL_SUCCESS){
    // Failure case. SSL_SUCCESS was not returned.
}
```

C.52.2.91 function wolfSSL_get_session_cache_memsize

```
int wolfSSL_get_session_cache_memsize(
    void
)
```

This function returns how large the session cache save buffer should be.

Parameters:

- **none** No parameters.

See: wolfSSL_memrestore_session_cache

Return: int This function returns an integer that represents the size of the session cache save buffer.

Example

```
int sz = // Minimum size for error checking;
...
if(sz < wolfSSL_get_session_cache_memsize()){
    // Memory buffer is too small
}
```

C.52.2.92 function wolfSSL_CTX_save_cert_cache

```
int wolfSSL_CTX_save_cert_cache(
    WOLFSSL_CTX * ctx,
    const char * fname
)
```

This function writes the cert cache from memory to file.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, holding the certificate information.
- **fname** a constant char pointer that points to a file for writing.

See:

- CM_SaveCertCache
- DoMemSaveCertCache

Return:

- SSL_SUCCESS if CM_SaveCertCache exits normally.
- BAD_FUNC_ARG is returned if either of the arguments are NULL.
- SSL_BAD_FILE if the cert cache save file could not be opened.
- BAD_MUTEX_E if the lock mutex failed.
- MEMORY_E the allocation of memory failed.
- FWRITE_ERROR Certificate cache file write failed.

Example

```
WOLFSSL_CTX* ctx = WOLFSSL_CTX_new( protocol def );
const char* fname;
...
if(wolfSSL_CTX_save_cert_cache(ctx, fname)){
    // file was written.
}
```

C.52.2.93 function wolfSSL_CTX_restore_cert_cache

```
int wolfSSL_CTX_restore_cert_cache(
    WOLFSSL_CTX * ctx,
    const char * fname
)
```

This function persistes certificate cache from a file.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, holding the certificate information.
- **fname** a constant char pointer that points to a file for reading.

See:

- CM_RestoreCertCache
- XFOPEN

Return:

- SSL_SUCCESS returned if the function, CM_RestoreCertCache, executes normally.
- SSL_BAD_FILE returned if XFOPEN returns XBADFILE. The file is corrupted.
- MEMORY_E returned if the allocated memory for the temp buffer fails.
- BAD_FUNC_ARG returned if fname or ctx have a NULL value.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
WOLFSSL* ssl = wolfSSL_new(ctx);
const char* fname = "path to file";
...
if(wolfSSL_CTX_restore_cert_cache(ctx, fname)){
    // check to see if the execution was successful
}
```

C.52.2.94 function wolfSSL_CTX_memsave_cert_cache

```
int wolfSSL_CTX_memsave_cert_cache(
    WOLFSSL_CTX * ctx,
    void * mem,
    int sz,
    int * used
)
```

This function persists the certificate cache to memory.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().
- **mem** a void pointer to the destination (output buffer).
- **sz** the size of the output buffer.
- **used** a pointer to size of the cert cache header.

See:

- DoMemSaveCertCache
- GetCertCacheMemSize
- CM_MemRestoreCertCache
- CM_GetCertCacheMemSize

Return:

- SSL_SUCCESS returned on successful execution of the function. No errors were thrown.
- BAD_MUTEX_E mutex error where the WOLFSSL_CERT_MANAGER member caLock was not 0 (zero).
- BAD_FUNC_ARG returned if ctx, mem, or used is NULL or if sz is less than or equal to 0 (zero).
- BUFFER_E output buffer mem was too small.

Example

```
WOLFSSL_CTX* ctx = WOLFSSL_CTX_new( protocol );
void* mem;
int sz;
int* used;
...
if(wolfSSL_CTX_memsave_cert_cache(ctx, mem, sz, used) != SSL_SUCCESS){
    // The function returned with an error
}
```

C.52.2.95 function wolfSSL_CTX_memrestore_cert_cache

```
int wolfSSL_CTX_memrestore_cert_cache(
    WOLFSSL_CTX * ctx,
    const void * mem,
    int sz
)
```

This function restores the certificate cache from memory.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().
- **mem** a void pointer with a value that will be restored to the certificate cache.
- **sz** an int type that represents the size of the mem parameter.

See: CM_MemRestoreCertCache

Return:

- SSL_SUCCESS returned if the function and subroutines executed without an error.
- BAD_FUNC_ARG returned if the ctx or mem parameters are NULL or if the sz parameter is less than or equal to zero.
- BUFFER_E returned if the cert cache memory buffer is too small.
- CACHE_MATCH_ERROR returned if there was a cert cache header mismatch.
- BAD_MUTEX_E returned if the lock mutex on failed.

Example

```
WOLFSSL_CTX* ctx = WOLFSSL_CTX_new( protocol method );
WOLFSSL* ssl = WOLFSSL_new(ctx);
void* mem;
int sz = (*int) sizeof(mem);
…
if(wolfSSL_CTX_memrestore_cert_cache(ssl->ctx, mem, sz)){
    // The success case
}
```

C.52.2.96 function wolfSSL_CTX_get_cert_cache_memsize

```
int wolfSSL_CTX_get_cert_cache_memsize(
    WOLFSSL_CTX * ctx
)
```

Returns the size the certificate cache save buffer needs to be.

Parameters:

- **ctx** a pointer to a wolfSSL_CTX structure, created using wolfSSL_CTX_new().

See: CM_GetCertCacheMemSize

Return:

- int integer value returned representing the memory size upon success.
- BAD_FUNC_ARG is returned if the WOLFSSL_CTX struct is NULL.
- BAD_MUTEX_E - returned if there was a mutex lock error.

Example

```
WOLFSSL_CTX* ctx = WOLFSSL_CTX_new(protocol);
...
int certCacheSize = wolfSSL_CTX_get_cert_cache_memsize(ctx);

if(certCacheSize != BAD_FUNC_ARG || certCacheSize != BAD_MUTEX_E){
// Successfully retrieved the memory size.
}
```

C.52.2.97 function wolfSSL_CTX_set_cipher_list

```
int wolfSSL_CTX_set_cipher_list(
    WOLFSSL_CTX * ctx,
    const char * list
)
```

This function sets cipher suite list for a given WOLFSSL_CTX. This cipher suite list becomes the default list for any new SSL sessions (WOLFSSL) created using this context. The ciphers in the list should be sorted in order of preference from highest to lowest. Each call to wolfSSL_CTX_set_cipher_list() resets the cipher suite list for the specific SSL context to the provided list each time the function is called. The cipher suite list, list, is a null_terminated text string, and a colon_delimited list. For example, one value for list may be "DHE_RSA_AES256_SHA256:DHE_RSA_AES128_SHA256:AES256_SHA256" Valid cipher values are the full name values from the cipher_names[] array in src/internal.c (for a definite list of valid cipher values check src/internal.c)

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().
- **list** null-terminated text string and a colon-delimited list of cipher suites to use with the specified SSL context.

See:

- wolfSSL_set_cipher_list
- wolfSSL_CTX_new

Return:

- SSL_SUCCESS will be returned upon successful function completion.
- SSL_FAILURE will be returned on failure.

Example

```
WOLFSSL_CTX* ctx = 0;
...
ret = wolfSSL_CTX_set_cipher_list(ctx,
“DHE-RSA-AES256-SHA256:DHE-RSA-AES128-SHA256:AES256-SHA256”);
if (ret != SSL_SUCCESS) {
    // failed to set cipher suite list
}
```

C.52.2.98 function wolfSSL_set_cipher_list

```
int wolfSSL_set_cipher_list(
    WOLFSSL * ssl,
    const char * list
)
```

This function sets cipher suite list for a given WOLFSSL object (SSL session). The ciphers in the list should be sorted in order of preference from highest to lowest. Each call to wolfSSL_set_cipher_list() resets the cipher suite list for the specific SSL session to the provided list each time the function is called. The cipher suite list, list, is a null_terminated text string, and a colon_delimited list. For example, one value for list may be “DHE_RSA_AES256_SHA256:DHE_RSA_AES128_SHA256:AES256_SHA256”. Valid cipher values are the full name values from the cipher_names[] array in src/internal.c (for a definite list of valid cipher values check src/internal.c)

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().
- **list** null-terminated text string and a colon-delimited list of cipher suites to use with the specified SSL session.

See:

- wolfSSL_CTX_set_cipher_list
- wolfSSL_new

Return:

- SSL_SUCCESS will be returned upon successful function completion.
- SSL_FAILURE will be returned on failure.

Example

```
int ret = 0;
WOLFSSL* ssl = 0;
...
ret = wolfSSL_set_cipher_list(ssl,
“DHE-RSA-AES256-SHA256:DHE-RSA-AES128-SHA256:AES256-SHA256”);
if (ret != SSL_SUCCESS) {
    // failed to set cipher suite list
}
```

C.52.2.99 function wolfSSL_dtls_set_using_nonblock

```
void wolfSSL_dtls_set_using_nonblock(
    WOLFSSL * ssl,
    int nonblock
)
```

This function informs the WOLFSSL DTLS object that the underlying UDP I/O is non-blocking. After an application creates a WOLFSSL object, if it will be used with a non-blocking UDP socket, call wolf-

SSL_dtls_set_using_nonblock() on it. This lets the WOLFSSL object know that receiving EWOULDBLOCK means that the recvfrom call would block rather than that it timed out.

Parameters:

- **ssl** pointer to the DTLS session, created with wolfSSL_new().
- **nonblock** value used to set non-blocking flag on WOLFSSL object. Use 1 to specify non-blocking, otherwise 0.

See:

- wolfSSL_dtls_get_using_nonblock
- wolfSSL_dtls_got_timeout
- wolfSSL_dtls_get_current_timeout

Return: none No return.

Example

```
WOLFSSL* ssl = 0;
...
wolfSSL_dtls_set_using_nonblock(ssl, 1);
```

C.52.2.100 function wolfSSL_dtls_get_using_nonblock

```
int wolfSSL_dtls_get_using_nonblock(
    WOLFSSL * ssl
)
```

This function allows the application to determine if wolfSSL is using non-blocking I/O with UDP. If wolfSSL is using non-blocking I/O, this function will return 1, otherwise 0. After an application creates a WOLFSSL object, if it will be used with a non-blocking UDP socket, call wolfSSL_dtls_set_using_nonblock() on it. This lets the WOLFSSL object know that receiving EWOULDBLOCK means that the recvfrom call would block rather than that it timed out. This function is only meaningful to DTLS sessions.

Parameters:

- **ssl** pointer to the DTLS session, created with wolfSSL_new().

See:

- wolfSSL_dtls_set_using_nonblock
- wolfSSL_dtls_got_timeout
- wolfSSL_dtls_set_using_nonblock

Return:

- 0 underlying I/O is blocking.
- 1 underlying I/O is non-blocking.

Example

```
int ret = 0;
WOLFSSL* ssl = 0;
...
ret = wolfSSL_dtls_get_using_nonblock(ssl);
if (ret == 1) {
    // underlying I/O is non-blocking
}
...
```

C.52.2.101 function wolfSSL_dtls_get_current_timeout

```
int wolfSSL_dtls_get_current_timeout(
    WOLFSSL * ssl
)
```

This function returns the current timeout value in seconds for the WOLFSSL object. When using non-blocking sockets, something in the user code needs to decide when to check for available recv data and how long it has been waiting. The value returned by this function indicates how long the application should wait.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_dtls
- wolfSSL_dtls_get_peer
- wolfSSL_dtls_got_timeout
- wolfSSL_dtls_set_peer

Return:

- seconds The current DTLS timeout value in seconds
- NOT_COMPILED_IN if wolfSSL was not built with DTLS support.

Example

```
int timeout = 0;
WOLFSSL* ssl;
...
timeout = wolfSSL_get_dtls_current_timeout(ssl);
printf("DTLS timeout (sec) = %d\n", timeout);
```

C.52.2.102 function wolfSSL_dtls13_use_quick_timeout

```
int wolfSSL_dtls13_use_quick_timeout(
    WOLFSSL * ssl
)
```

This function returns true if the application should setup a quicker timeout. When using non-blocking sockets, something in the user code needs to decide when to check for available data and how long it needs to wait. If this function returns true, it means that the library already detected some disruption in the communication, but it wants to wait for a little longer in case some messages from the other peers are still in flight. Is up to the application to fine tune the value of this timer, a good one may be dtls_get_current_timeout() / 4.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_dtls
- wolfSSL_dtls_get_peer
- wolfSSL_dtls_got_timeout
- wolfSSL_dtls_set_peer
- wolfSSL_dtls13_set_send_more_acks

Return: true if the application code should setup a quicker timeout

C.52.2.103 function wolfSSL_dtls13_set_send_more_acks

```
void wolfSSL_dtls13_set_send_more_acks(
    WOLFSSL * ssl,
    int value
)
```

This function sets whether the library should send ACKs to the other peer immediately when detecting disruption or not. Sending ACKs immediately assures minimum latency but it may consume more bandwidth than necessary. If the application manages the timer by itself and this option is set to 0 then application code can use wolfSSL_dtls13_use_quick_timeout() to determine if it should setup a quicker timeout to send those delayed ACKs.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **value** 1 to set the option, 0 to disable the option

See:

- wolfSSL_dtls
- wolfSSL_dtls_get_peer
- wolfSSL_dtls_got_timeout
- wolfSSL_dtls_set_peer
- wolfSSL_dtls13_use_quick_timeout

C.52.2.104 function wolfSSL_dtls_set_timeout_init

```
int wolfSSL_dtls_set_timeout_init(
    WOLFSSL * ssl,
    int
)
```

This function sets the dtls timeout.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **timeout** an int type that will be set to the dtls_timeout_init member of the WOLFSSL structure.

See:

- wolfSSL_dtls_set_timeout_max
- wolfSSL_dtls_got_timeout

Return:

- SSL_SUCCESS returned if the function executes without an error. The dtls_timeout_init and the dtls_timeout members of SSL have been set.
- BAD_FUNC_ARG returned if the WOLFSSL struct is NULL or if the timeout is not greater than 0. It will also return if the timeout argument exceeds the maximum value allowed.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
int timeout = TIMEOUT;
...
if(wolfSSL_dtls_set_timeout_init(ssl, timeout)){
    // the dtls timeout was set
} else {
```

```
    // Failed to set DTLS timeout.
}
```

C.52.2.105 function wolfSSL_dtls_set_timeout_max

```
int wolfSSL_dtls_set_timeout_max(
    WOLFSSL * ssl,
    int
)
```

This function sets the maximum dtls timeout.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **timeout** an int type representing the dtls maximum timeout.

See:

- wolfSSL_dtls_set_timeout_init
- wolfSSL_dtls_got_timeout

Return:

- SSL_SUCCESS returned if the function executed without an error.
- BAD_FUNC_ARG returned if the WOLFSSL struct is NULL or if the timeout argument is not greater than zero or is less than the dtls_timeout_init member of the WOLFSSL structure.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
int timeout = TIMEOUTVAL;
...
int ret = wolfSSL_dtls_set_timeout_max(ssl);
if(!ret){
    // Failed to set the max timeout
}
```

C.52.2.106 function wolfSSL_dtls_got_timeout

```
int wolfSSL_dtls_got_timeout(
    WOLFSSL * ssl
)
```

When using non-blocking sockets with DTLS, this function should be called on the WOLFSSL object when the controlling code thinks the transmission has timed out. It performs the actions needed to retry the last transmit, including adjusting the timeout value. If it has been too long, this will return a failure.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_dtls_get_current_timeout
- wolfSSL_dtls_get_peer
- wolfSSL_dtls_set_peer
- wolfSSL_dtls

Return:

- SSL_SUCCESS will be returned upon success
- SSL_FATAL_ERROR will be returned if there have been too many retransmissions/timeouts without getting a response from the peer.
- NOT_COMPILED_IN will be returned if wolfSSL was not compiled with DTLS support.

Example

```
See the following files for usage examples:
<wolfssl_root>/examples/client/client.c
<wolfssl_root>/examples/server/server.c
```

C.52.2.107 function wolfSSL_dtls_retransmit

```
int wolfSSL_dtls_retransmit(
    WOLFSSL * ssl
)
```

When using non-blocking sockets with DTLS, this function retransmits the last handshake flight ignoring the expected timeout value and retransmit count. It is useful for applications that are using DTLS and need to manage even the timeout and retry count.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_dtls_get_current_timeout
- wolfSSL_dtls_got_timeout
- wolfSSL_dtls

Return:

- SSL_SUCCESS will be returned upon success
- SSL_FATAL_ERROR will be returned if there have been too many retransmissions/timeouts without getting a response from the peer.

Example

```
int ret = 0;
WOLFSSL* ssl;
...
ret = wolfSSL_dtls_retransmit(ssl);
```

C.52.2.108 function wolfSSL_dtls

```
int wolfSSL_dtls(
    WOLFSSL * ssl
)
```

This function is used to determine if the SSL session has been configured to use DTLS.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_dtls_get_current_timeout
- wolfSSL_dtls_get_peer
- wolfSSL_dtls_got_timeout
- wolfSSL_dtls_set_peer

Return:

- 1 If the SSL session (ssl) has been configured to use DTLS, this function will return 1.
- 0 otherwise.

Example

```
int ret = 0;
WOLFSSL* ssl;
...
ret = wolfSSL_dtls(ssl);
if (ret) {
    // SSL session has been configured to use DTLS
}
```

C.52.2.109 function wolfSSL_dtls_set_peer

```
int wolfSSL_dtls_set_peer(
    WOLFSSL * ssl,
    void * peer,
    unsigned int peerSz
)
```

This function sets the DTLS peer, peer (sockaddr_in) with size of peerSz.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **peer** pointer to peer's sockaddr_in structure. If NULL then the peer information in ssl is cleared.
- **peerSz** size of the sockaddr_in structure pointed to by peer. If 0 then the peer information in ssl is cleared.

See:

- wolfSSL_dtls_get_current_timeout
- wolfSSL_dtls_set_pending_peer
- wolfSSL_dtls_get_peer
- wolfSSL_dtls_got_timeout
- wolfSSL_dtls

Return:

- SSL_SUCCESS will be returned upon success.
- SSL_FAILURE will be returned upon failure.
- SSL_NOT_IMPLEMENTED will be returned if wolfSSL was not compiled with DTLS support.

Example

```
int ret = 0;
WOLFSSL* ssl;
sockaddr_in addr;
...
ret = wolfSSL_dtls_set_peer(ssl, &addr, sizeof(addr));
if (ret != SSL_SUCCESS) {
    // failed to set DTLS peer
}
```

C.52.2.110 function wolfSSL_dtls_set_pending_peer

```
int wolfSSL_dtls_set_pending_peer(
    WOLFSSL * ssl,
    void * peer,
    unsigned int peerSz
)
```

This function sets the pending DTLS peer, peer (sockaddr_in) with size of peerSz. This sets the pending peer that will be upgraded to a regular peer when we successfully de-protect the next record. This is useful in scenarios where the peer's address can change to avoid off-path attackers from changing the peer address. This should be used with Connection ID's to allow seamless and safe transition to a new peer address.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **peer** pointer to peer's sockaddr_in structure. If NULL then the peer information in ssl is cleared.
- **peerSz** size of the sockaddr_in structure pointed to by peer. If 0 then the peer information in ssl is cleared.

See:

- wolfSSL_dtls_get_current_timeout
- wolfSSL_dtls_set_peer
- wolfSSL_dtls_get_peer
- wolfSSL_dtls_got_timeout
- wolfSSL_dtls

Return:

- SSL_SUCCESS will be returned upon success.
- SSL_FAILURE will be returned upon failure.
- SSL_NOT_IMPLEMENTED will be returned if wolfSSL was not compiled with DTLS support.

Example

```
int ret = 0;
WOLFSSL* ssl;
sockaddr_in addr;
...
ret = wolfSSL_dtls_set_pending_peer(ssl, &addr, sizeof(addr));
if (ret != SSL_SUCCESS) {
    // failed to set DTLS peer
}
```

C.52.2.111 function wolfSSL_dtls_get_peer

```
int wolfSSL_dtls_get_peer(
    WOLFSSL * ssl,
    void * peer,
    unsigned int * peerSz
)
```

This function gets the sockaddr_in (of size peerSz) of the current DTLS peer. The function will compare peerSz to the actual DTLS peer size stored in the SSL session. If the peer will fit into peer, the peer's sockaddr_in will be copied into peer, with peerSz set to the size of peer.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **peer** pointer to memory location to store peer's sockaddr_in structure.

- **peerSz** input/output size. As input, the size of the allocated memory pointed to by peer. As output, the size of the actual sockaddr_in structure pointed to by peer.

See:

- wolfSSL_dtls_get_current_timeout
- wolfSSL_dtls_got_timeout
- wolfSSL_dtls_set_peer
- wolfSSL_dtls

Return:

- SSL_SUCCESS will be returned upon success.
- SSL_FAILURE will be returned upon failure.
- SSL_NOT_IMPLEMENTED will be returned if wolfSSL was not compiled with DTLS support.

Example

```
int ret = 0;
WOLFSSL* ssl;
sockaddr_in addr;
...
ret = wolfSSL_dtls_get_peer(ssl, &addr, sizeof(addr));
if (ret != SSL_SUCCESS) {
    // failed to get DTLS peer
}
```

C.52.2.112 function wolfSSL_dtls_get0_peer

```
int wolfSSL_dtls_get0_peer(
    WOLFSSL * ssl,
    const void ** peer,
    unsigned int * peerSz
)
```

This function gets the sockaddr_in (of size peerSz) of the current DTLS peer. This is a zero-copy alternative to wolfSSL_dtls_get_peer().

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **peer** pointer to return the internal buffer holding the peer address
- **peerSz** output the size of the actual sockaddr_in structure pointed to by peer.

See:

- wolfSSL_dtls_get_current_timeout
- wolfSSL_dtls_got_timeout
- wolfSSL_dtls_set_peer
- wolfSSL_dtls

Return:

- SSL_SUCCESS will be returned upon success.
- SSL_FAILURE will be returned upon failure.
- SSL_NOT_IMPLEMENTED will be returned if wolfSSL was not compiled with DTLS support.

Example

```
int ret = 0;
WOLFSSL* ssl;
sockaddr_in* addr;
```

```
unsigned int addrSz;
...
ret = wolfSSL_dtls_get_peer(ssl, &addr, &addrSz);
if (ret != SSL_SUCCESS) {
    // failed to get DTLS peer
}
```

C.52.2.113 function wolfSSL_ERR_error_string

```
char * wolfSSL_ERR_error_string(
    unsigned long errNumber,
    char * data
)
```

This function converts an error code returned by wolfSSL_get_error() and data is the storage buffer which the error string will be placed in. The maximum length of data is 80 characters by default, as defined by MAX_ERROR_SZ is wolfssl/wolfcrypt/error.h.

Parameters:

- **errNumber** error code returned by wolfSSL_get_error().
- **data** output buffer containing human-readable error string matching errNumber.

See:

- wolfSSL_get_error
- wolfSSL_ERR_error_string_n
- wolfSSL_ERR_print_errors_fp
- wolfSSL_load_error_strings

Return:

- success On successful completion, this function returns the same string as is returned in data.
- failure Upon failure, this function returns a string with the appropriate failure reason, msg.

Example

```
int err = 0;
WOLFSSL* ssl;
char buffer[80];
...
err = wolfSSL_get_error(ssl, 0);
wolfSSL_ERR_error_string(err, buffer);
printf("err = %d, %s\n", err, buffer);
```

C.52.2.114 function wolfSSL_ERR_error_string_n

```
void wolfSSL_ERR_error_string_n(
    unsigned long e,
    char * buf,
    unsigned long sz
)
```

This function is a version of wolfSSL_ERR_error_string() into a more human-readable error string. The human-readable string is placed in buf.

Parameters:

- **e** error code returned by wolfSSL_get_error().
- **buff** output buffer containing human-readable error string matching e.

- **len** maximum length in characters which may be written to buf.

See:

- wolfSSL_get_error
- wolfSSL_ERR_error_string
- wolfSSL_ERR_print_errors_fp
- wolfSSL_load_error_strings

Return: none No returns.

Example

```
int err = 0;
WOLFSSL* ssl;
char buffer[80];
...
err = wolfSSL_get_error(ssl, 0);
wolfSSL_ERR_error_string_n(err, buffer, 80);
printf("err = %d, %s\n", err, buffer);
```

C.52.2.115 function wolfSSL_get_shutdown

```
int wolfSSL_get_shutdown(
    const WOLFSSL * ssl
)
```

This function checks the shutdown conditions in closeNotify or connReset or sentNotify members of the Options structure. The Options structure is within the WOLFSSL structure.

Parameters:

- **ssl** a constant pointer to a WOLFSSL structure, created using wolfSSL_new().

See: wolfSSL_SESSION_free

Return:

- 1 SSL_SENT_SHUTDOWN is returned.
- 2 SSL_RECEIVED_SHUTDOWN is returned.

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_CTX* ctx = WOLFSSL_CTX_new( protocol method );
WOLFSSL* ssl = WOLFSSL_new(ctx);
…
int ret;
ret = wolfSSL_get_shutdown(ssl);

if(ret == 1){
    SSL_SENT_SHUTDOWN
} else if(ret == 2){
    SSL_RECEIVED_SHUTDOWN
} else {
    Fatal error.
}
```

C.52.2.116 function wolfSSL_session_reused

```
int wolfSSL_session_reused(
    WOLFSSL * ssl
)
```

This function returns the resuming member of the options struct. The flag indicates whether or not to reuse a session. If not, a new session must be established.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_SESSION_free
- wolfSSL_GetSessionIndex
- wolfSSL_memsave_session_cache

Return: This function returns an int type held in the Options structure representing the flag for session reuse.

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
…
if(!wolfSSL_session_reused(sslResume)){
    // No session reuse allowed.
}
```

C.52.2.117 function wolfSSL_is_init_finished

```
int wolfSSL_is_init_finished(
    WOLFSSL * ssl
)
```

This function checks to see if the connection is established.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_set_accept_state
- wolfSSL_get_keys
- wolfSSL_set_shutdown

Return:

- 0 returned if the connection is not established, i.e. the WOLFSSL struct is NULL or the handshake is not done.
- 1 returned if the connection is established i.e. the WOLFSSL handshake is done.

EXAMPLE

```
#include <wolfssl/ssl.h>

WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
WOLFSSL* ssl = wolfSSL_new(ctx);
...
if(wolfSSL_is_init_finished(ssl)){
```

```
    Handshake is done and connection is established
}
```

C.52.2.118 function wolfSSL_get_version

```
const char * wolfSSL_get_version(
    WOLFSSL * ssl
)
```

Returns the SSL version being used as a string.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See: wolfSSL_lib_version

Return:

- "SSLv3" Using SSLv3
- "TLSv1" Using TLSv1
- "TLSv1.1" Using TLSv1.1
- "TLSv1.2" Using TLSv1.2
- "TLSv1.3" Using TLSv1.3
- "DTLS": Using DTLS
- "DTLSv1.2" Using DTLSv1.2
- "unknown" There was a problem determining which version of TLS being used.

Example

```
wolfSSL_Init();
WOLFSSL_CTX* ctx;
WOLFSSL* ssl;
WOLFSSL_METHOD method = // Some wolfSSL method
ctx = wolfSSL_CTX_new(method);
ssl = wolfSSL_new(ctx);
printf(wolfSSL_get_version("Using version: %s", ssl));
```

C.52.2.119 function wolfSSL_get_current_cipher_suite

```
int wolfSSL_get_current_cipher_suite(
    WOLFSSL * ssl
)
```

Returns the current cipher suit an ssl session is using.

Parameters:

- **ssl** The SSL session to check.

See:

- wolfSSL_CIPHER_get_name
- wolfSSL_get_current_cipher
- wolfSSL_get_cipher_list

Return:

- ssl->options.cipherSuite An integer representing the current cipher suite.
- 0 The ssl session provided is null.

Example

```
wolfSSL_Init();
WOLFSSL_CTX* ctx;
WOLFSSL* ssl;
WOLFSSL_METHOD method = // Some wolfSSL method
ctx = wolfSSL_CTX_new(method);
ssl = wolfSSL_new(ctx);

if(wolfSSL_get_current_cipher_suite(ssl) == 0)
{
    // Error getting cipher suite
}
```

C.52.2.120 function wolfSSL_get_current_cipher

```
WOLFSSL_CIPHER * wolfSSL_get_current_cipher(
    WOLFSSL * ssl
)
```

This function returns a pointer to the current cipher in the ssl session.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_get_cipher
- wolfSSL_get_cipher_name_internal
- wolfSSL_get_cipher_name

Return:

- The function returns the address of the cipher member of the WOLFSSL struct. This is a pointer to the WOLFSSL_CIPHER structure.
- NULL returned if the WOLFSSL structure is NULL.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
WOLFSSL* ssl = wolfSSL_new(ctx);
…
WOLFSSL_CIPHER* cipherCurr = wolfSSL_get_current_cipher;

if(!cipherCurr){
    // Failure case.
} else {
    // The cipher was returned to cipherCurr
}
```

C.52.2.121 function wolfSSL_CIPHER_get_name

```
const char * wolfSSL_CIPHER_get_name(
    const WOLFSSL_CIPHER * cipher
)
```

This function matches the cipher suite in the SSL object with the available suites and returns the string representation.

Parameters:

- **cipher** a constant pointer to a WOLFSSL_CIPHER structure.

See:

- wolfSSL_get_cipher
- wolfSSL_get_current_cipher
- wolfSSL_get_cipher_name_internal
- wolfSSL_get_cipher_name

Return:

- string This function returns the string representation of the matched cipher suite.
- none It will return “None” if there are no suites matched.

Example

```
// gets cipher name in the format DHE_RSA ...
const char* wolfSSL_get_cipher_name_internal(WOLFSSL* ssl){
WOLFSSL_CIPHER* cipher;
const char* fullName;
…
cipher = wolfSSL_get_curent_cipher(ssl);
fullName = wolfSSL_CIPHER_get_name(cipher);

if(fullName){
    // sanity check on returned cipher
}
```

C.52.2.122 function wolfSSL_get_cipher

```
const char * wolfSSL_get_cipher(
    WOLFSSL *
)
```

This function matches the cipher suite in the SSL object with the available suites.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_CIPHER_get_name
- wolfSSL_get_current_cipher

Return: This function returns the string value of the suite matched. It will return “None” if there are no suites matched.

Example

```
#ifdef WOLFSSL_DTLS
…
// make sure a valid suite is used
if(wolfSSL_get_cipher(ssl) == NULL){
    WOLFSSL_MSG(“Can not match cipher suite imported”);
    return MATCH_SUITE_ERROR;
}
…
#endif // WOLFSSL_DTLS
```

C.52.2.123 function wolfSSL_get1_session

```
WOLFSSL_SESSION * wolfSSL_get1_session(
    WOLFSSL * ssl
)
```

This function returns the WOLFSSL_SESSION from the WOLFSSL structure as a reference type. This requires calling wolfSSL_SESSION_free to release the session reference. The WOLFSSL_SESSION pointed to contains all the necessary information required to perform a session resumption and reestablish the connection without a new handshake. For session resumption, before calling wolfSSL_shutdown() needs to be freed after the application is done with it by calling wolfSSL_SESSION_free() on it.

Parameters:

- **ssl** WOLFSSL structure to get session from.

See:

- wolfSSL_new
- wolfSSL_free
- wolfSSL_SESSION_free

Return:

- WOLFSSL_SESSION On success return session pointer.
- NULL will be returned if ssl is NULL, the SSL session cache is disabled, wolfSSL doesn't have the Session ID available, or mutex functions fail.

Example

```
WOLFSSL* ssl;
WOLFSSL_SESSION* ses;
// attempt/complete handshake
wolfSSL_connect(ssl);
ses  = wolfSSL_get1_session(ssl);
// check ses information
// disconnect / setup new SSL instance
wolfSSL_set_session(ssl, ses);
// attempt/resume handshake
wolfSSL_SESSION_free(ses);
```

C.52.2.124 function wolfSSLv23_client_method

```
WOLFSSL_METHOD * wolfSSLv23_client_method(
    void
)
```

The wolfSSLv23_client_method() function will use the highest protocol version supported by the server and downgrade to SSLv3 if needed. In this case, the client will be able to connect to a server running SSLv3 - TLSv1.3.

Parameters:

- **none** No parameters

See:

- wolfSSLv3_client_method
- wolfTLSv1_client_method
- wolfTLSv1_1_client_method
- wolfTLSv1_2_client_method
- wolfTLSv1_3_client_method

- wolfDTLSv1_client_method
- wolfSSL_CTX_new

Return:

- pointer upon success a pointer to a WOLFSSL_METHOD.
- Failure If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Example

```
WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;
method = wolfSSLv23_client_method();
if (method == NULL) {
// unable to get method
}

ctx = wolfSSL_CTX_new(method);
...
```

C.52.2.125 function wolfSSL_BIO_get_mem_data

```
int wolfSSL_BIO_get_mem_data(
    WOLFSSL_BIO * bio,
    void * p
)
```

This is used to set a byte pointer to the start of the internal memory buffer.

Parameters:

- **bio** WOLFSSL_BIO structure to get memory buffer of.
- **p** byte pointer to set to memory buffer.

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_s_mem
- wolfSSL_BIO_set_fp
- wolfSSL_BIO_free

Return:

- size On success the size of the buffer is returned
- SSL_FATAL_ERROR If an error case was encountered.

Example

```
WOLFSSL_BIO* bio;
const byte* p;
int ret;
bio  = wolfSSL_BIO_new(wolfSSL_BIO_s_mem());
ret  = wolfSSL_BIO_get_mem_data(bio, &p);
// check ret value
```

C.52.2.126 function wolfSSL_BIO_set_fd

```
long wolfSSL_BIO_set_fd(
    WOLFSSL_BIO * b,
    int fd,
```

```
    int flag
)
```

Sets the file descriptor for bio to use.

Parameters:

- **bio** WOLFSSL_BIO structure to set fd.
- **fd** file descriptor to use.
- **closeF** flag for behavior when closing fd.

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_free

Return: SSL_SUCCESS(1) upon success.

Example

```
WOLFSSL_BIO* bio;
int fd;
// setup bio
wolfSSL_BIO_set_fd(bio, fd, BIO_NOCLOSE);
```

C.52.2.127 function wolfSSL_BIO_set_close

```
int wolfSSL_BIO_set_close(
    WOLFSSL_BIO * b,
    long flag
)
```

Sets the close flag, used to indicate that the i/o stream should be closed when the BIO is freed.

Parameters:

- **bio** WOLFSSL_BIO structure.
- **flag** flag for behavior when closing i/o stream.

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_free

Return: SSL_SUCCESS(1) upon success.

Example

```
WOLFSSL_BIO* bio;
// setup bio
wolfSSL_BIO_set_close(bio, BIO_NOCLOSE);
```

C.52.2.128 function wolfSSL_BIO_s_socket

```
WOLFSSL_BIO_METHOD * wolfSSL_BIO_s_socket(
    void
)
```

This is used to get a BIO_SOCKET type WOLFSSL_BIO_METHOD.

Parameters:

- **none** No parameters.

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_s_mem

Return: WOLFSSL_BIO_METHOD pointer to a WOLFSSL_BIO_METHOD structure that is a socket type

Example

```
WOLFSSL_BIO* bio;
bio = wolfSSL_BIO_new(wolfSSL_BIO_s_socket);
```

C.52.2.129 function wolfSSL_BIO_set_write_buf_size

```
int wolfSSL_BIO_set_write_buf_size(
    WOLFSSL_BIO * b,
    long size
)
```

This is used to set the size of write buffer for a WOLFSSL_BIO. If write buffer has been previously set this function will free it when resetting the size. It is similar to wolfSSL_BIO_reset in that it resets read and write indexes to 0.

Parameters:

- **bio** WOLFSSL_BIO structure to set fd.
- **size** size of buffer to allocate.

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_s_mem
- wolfSSL_BIO_free

Return:

- SSL_SUCCESS On successfully setting the write buffer.
- SSL_FAILURE If an error case was encountered.

Example

```
WOLFSSL_BIO* bio;
int ret;
bio = wolfSSL_BIO_new(wolfSSL_BIO_s_mem());
ret = wolfSSL_BIO_set_write_buf_size(bio, 15000);
// check return value
```

C.52.2.130 function wolfSSL_BIO_make_bio_pair

```
int wolfSSL_BIO_make_bio_pair(
    WOLFSSL_BIO * b1,
    WOLFSSL_BIO * b2
)
```

This is used to pair two bios together. A pair of bios acts similar to a two way pipe writing to one can be read by the other and vice versa. It is expected that both bios be in the same thread, this function is not thread safe. Freeing one of the two bios removes both from being paired. If a write buffer size was not previously set for either of the bios it is set to a default size of 17000 (WOLFSSL_BIO_SIZE) before being paired.

Parameters:

- **b1** WOLFSSL_BIO structure to set pair.
- **b2** second WOLFSSL_BIO structure to complete pair.

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_s_mem
- wolfSSL_BIO_free

Return:

- SSL_SUCCESS On successfully pairing the two bios.
- SSL_FAILURE If an error case was encountered.

Example

```
WOLFSSL_BIO* bio;
WOLFSSL_BIO* bio2;
int ret;
bio  = wolfSSL_BIO_new(wolfSSL_BIO_s_bio());
bio2 = wolfSSL_BIO_new(wolfSSL_BIO_s_bio());
ret = wolfSSL_BIO_make_bio_pair(bio, bio2);
// check ret value
```

C.52.2.131 function wolfSSL_BIO_ctrl_reset_read_request

```
int wolfSSL_BIO_ctrl_reset_read_request(
    WOLFSSL_BIO * bio
)
```

This is used to set the read request flag back to 0.

Parameters:

- **bio** WOLFSSL_BIO structure to set read request flag.

See:

- wolfSSL_BIO_new, wolfSSL_BIO_s_mem
- wolfSSL_BIO_new, wolfSSL_BIO_free

Return:

- SSL_SUCCESS On successfully setting value.
- SSL_FAILURE If an error case was encountered.

Example

```
WOLFSSL_BIO* bio;
int ret;
...
ret = wolfSSL_BIO_ctrl_reset_read_request(bio);
// check ret value
```

C.52.2.132 function wolfSSL_BIO_nread0

```
int wolfSSL_BIO_nread0(
    WOLFSSL_BIO * bio,
    char ** buf
)
```

This is used to get a buffer pointer for reading from. Unlike wolfSSL_BIO_nread the internal read index is not advanced by the number returned from the function call. Reading past the value returned can result in reading out of array bounds.

Parameters:

- **bio** WOLFSSL_BIO structure to read from.
- **buf** pointer to set at beginning of read array.

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_nwrite0

Return: >=0 on success return the number of bytes to read

Example

```
WOLFSSL_BIO* bio;
char* bufPt;
int ret;
// set up bio
ret = wolfSSL_BIO_nread0(bio, &bufPt); // read as many bytes as possible
// handle negative ret check
// read ret bytes from bufPt
```

C.52.2.133 function wolfSSL_BIO_nread

```
int wolfSSL_BIO_nread(
    WOLFSSL_BIO * bio,
    char ** buf,
    int num
)
```

This is used to get a buffer pointer for reading from. The internal read index is advanced by the number returned from the function call with buf being pointed to the beginning of the buffer to read from. In the case that less bytes are in the read buffer than the value requested with num the lesser value is returned. Reading past the value returned can result in reading out of array bounds.

Parameters:

- **bio** WOLFSSL_BIO structure to read from.
- **buf** pointer to set at beginning of read array.
- **num** number of bytes to try and read.

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_nwrite

Return:

- =0 on success return the number of bytes to read
- WOLFSSL_BIO_ERROR(-1) on error case with nothing to read return -1

Example

```
WOLFSSL_BIO* bio;
char* bufPt;
int ret;

// set up bio
```

```
ret = wolfSSL_BIO_nread(bio, &bufPt, 10); // try to read 10 bytes
// handle negative ret check
// read ret bytes from bufPt
```

C.52.2.134 function wolfSSL_BIO_nwrite

```
int wolfSSL_BIO_nwrite(
    WOLFSSL_BIO * bio,
    char ** buf,
    int num
)
```

Gets a pointer to the buffer for writing as many bytes as returned by the function. Writing more bytes to the pointer returned then the value returned can result in writing out of bounds.

Parameters:

- **bio** WOLFSSL_BIO structure to write to.
- **buf** pointer to buffer to write to.
- **num** number of bytes desired to be written.

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_free
- wolfSSL_BIO_nread

Return:

- int Returns the number of bytes that can be written to the buffer pointer returned.
- WOLFSSL_BIO_UNSET(-2) in the case that is not part of a bio pair
- WOLFSSL_BIO_ERROR(-1) in the case that there is no more room to write to

Example

```
WOLFSSL_BIO* bio;
char* bufPt;
int ret;
// set up bio
ret = wolfSSL_BIO_nwrite(bio, &bufPt, 10); // try to write 10 bytes
// handle negative ret check
// write ret bytes to bufPt
```

C.52.2.135 function wolfSSL_BIO_reset

```
int wolfSSL_BIO_reset(
    WOLFSSL_BIO * bio
)
```

Resets bio to an initial state. As an example for type BIO_BIO this resets the read and write index.

Parameters:

- **bio** WOLFSSL_BIO structure to reset.

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_free

Return:

- 0 On successfully resetting the bio.
- WOLFSSL_BIO_ERROR(-1) Returned on bad input or unsuccessful reset.

Example

```
WOLFSSL_BIO* bio;
// setup bio
wolfSSL_BIO_reset(bio);
//use pt
```

C.52.2.136 function wolfSSL_BIO_seek

```
int wolfSSL_BIO_seek(
    WOLFSSL_BIO * bio,
    int ofs
)
```

This function adjusts the file pointer to the offset given. This is the offset from the head of the file.

Parameters:

- **bio** WOLFSSL_BIO structure to set.
- **ofs** offset into file.

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_s_mem
- wolfSSL_BIO_set_fp
- wolfSSL_BIO_free

Return:

- 0 On successfully seeking.
- -1 If an error case was encountered.

Example

```
WOLFSSL_BIO* bio;
XFILE fp;
int ret;
bio  = wolfSSL_BIO_new(wolfSSL_BIO_s_file());
ret  = wolfSSL_BIO_set_fp(bio, &fp);
// check ret value
ret  = wolfSSL_BIO_seek(bio, 3);
// check ret value
```

C.52.2.137 function wolfSSL_BIO_write_filename

```
int wolfSSL_BIO_write_filename(
    WOLFSSL_BIO * bio,
    char * name
)
```

This is used to set and write to a file. WIll overwrite any data currently in the file and is set to close the file when the bio is freed.

Parameters:

- **bio** WOLFSSL_BIO structure to set file.
- **name** name of file to write to.

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_s_file
- wolfSSL_BIO_set_fp
- wolfSSL_BIO_free

Return:

- SSL_SUCCESS On successfully opening and setting file.
- SSL_FAILURE If an error case was encountered.

Example

```
WOLFSSL_BIO* bio;
int ret;
bio  = wolfSSL_BIO_new(wolfSSL_BIO_s_file());
ret  = wolfSSL_BIO_write_filename(bio, “test.txt”);
// check ret value
```

C.52.2.138 function wolfSSL_BIO_set_mem_eof_return

```
long wolfSSL_BIO_set_mem_eof_return(
    WOLFSSL_BIO * bio,
    int v
)
```

This is used to set the end of file value. Common value is -1 so as not to get confused with expected positive values.

Parameters:

- **bio** WOLFSSL_BIO structure to set end of file value.
- **v** value to set in bio.

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_s_mem
- wolfSSL_BIO_set_fp
- wolfSSL_BIO_free

Return: 0 returned on completion

Example

```
WOLFSSL_BIO* bio;
int ret;
bio  = wolfSSL_BIO_new(wolfSSL_BIO_s_mem());
ret  = wolfSSL_BIO_set_mem_eof_return(bio, -1);
// check ret value
```

C.52.2.139 function wolfSSL_BIO_get_mem_ptr

```
long wolfSSL_BIO_get_mem_ptr(
    WOLFSSL_BIO * bio,
    WOLFSSL_BUF_MEM ** m
)
```

This is a getter function for WOLFSSL_BIO memory pointer.

Parameters:

- **bio** pointer to the WOLFSSL_BIO structure for getting memory pointer.
- **ptr** structure that is currently a char*. Is set to point to bio's memory.

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_s_mem

Return:

- SSL_SUCCESS On successfully getting the pointer SSL_SUCCESS is returned (currently value of 1).
- SSL_FAILURE Returned if NULL arguments are passed in (currently value of 0).

Example

```
WOLFSSL_BIO* bio;
WOLFSSL_BUF_MEM* pt;
// setup bio
wolfSSL_BIO_get_mem_ptr(bio, &pt);
//use pt
```

C.52.2.140 function wolfSSL_X509_NAME_oneline

```
char * wolfSSL_X509_NAME_oneline(
    WOLFSSL_X509_NAME * name,
    char * in,
    int sz
)
```

This function copies the name of the x509 into a buffer.

Parameters:

- **name** a pointer to a WOLFSSL_X509 structure.
- **in** a buffer to hold the name copied from the WOLFSSL_X509_NAME structure.
- **sz** the maximum size of the buffer.

See:

- wolfSSL_X509_get_subject_name
- wolfSSL_X509_get_issuer_name
- wolfSSL_X509_get_isCA
- wolfSSL_get_peer_certificate
- wolfSSL_X509_version

Return: A char pointer to the buffer with the WOLFSSL_X509_NAME structures name member's data is returned if the function executed normally.

Example

```
WOLFSSL_X509 x509;
char* name;
...
name = wolfSSL_X509_NAME_oneline(wolfSSL_X509_get_issuer_name(x509), 0, 0);

if(name <= 0){
    // There's nothing in the buffer.
}
```

C.52.2.141 function wolfSSL_X509_get_issuer_name

```
WOLFSSL_X509_NAME * wolfSSL_X509_get_issuer_name(
    WOLFSSL_X509 * cert
)
```

This function returns the name of the certificate issuer.

Parameters:

- **cert** a pointer to a WOLFSSL_X509 structure.

See:

- wolfSSL_X509_get_subject_name
- wolfSSL_X509_get_isCA
- wolfSSL_get_peer_certificate
- wolfSSL_X509_NAME_oneline

Return:

- point a pointer to the WOLFSSL_X509 struct's issuer member is returned.
- NULL if the cert passed in is NULL.

Example

```
WOLFSSL_X509* x509;
WOLFSSL_X509_NAME issuer;
...
issuer = wolfSSL_X509_NAME_oneline(wolfSSL_X509_get_issuer_name(x509), 0, 0);

if(!issuer){
    // NULL was returned
} else {
    // issuer hods the name of the certificate issuer.
}
```

C.52.2.142 function wolfSSL_X509_get_subject_name

```
WOLFSSL_X509_NAME * wolfSSL_X509_get_subject_name(
    WOLFSSL_X509 * cert
)
```

This function returns the subject member of the WOLFSSL_X509 structure.

Parameters:

- **cert** a pointer to a WOLFSSL_X509 structure.

See:

- wolfSSL_X509_get_issuer_name
- wolfSSL_X509_get_isCA
- wolfSSL_get_peer_certificate

Return: pointer a pointer to the WOLFSSL_X509_NAME structure. The pointer may be NULL if the WOLFSSL_X509 struct is NULL or if the subject member of the structure is NULL.

Example

```
WOLFSSL_X509* cert;
WOLFSSL_X509_NAME name;
...
```

```
name = wolfSSL_X509_get_subject_name(cert);
if(name == NULL){
    // Deal with the NULL cacse
}
```

C.52.2.143 function wolfSSL_X509_get_isCA

```
int wolfSSL_X509_get_isCA(
    WOLFSSL_X509 * cert
)
```

Checks the isCa member of the WOLFSSL_X509 structure and returns the value.

Parameters:

- **cert** a pointer to a WOLFSSL_X509 structure.

See:

- wolfSSL_X509_get_issuer_name
- wolfSSL_X509_get_isCA

Return:

- isCA returns the value in the isCA member of the WOLFSSL_X509 structure is returned.
- 0 returned if there is not a valid x509 structure passed in.

Example

```
WOLFSSL* ssl;
...
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
...
if(wolfSSL_X509_get_isCA(ssl)){
    // This is the CA
}else {
    // Failure case
}
```

C.52.2.144 function wolfSSL_X509_NAME_get_text_by_NID

```
int wolfSSL_X509_NAME_get_text_by_NID(
    WOLFSSL_X509_NAME * name,
    int nid,
    char * buf,
    int len
)
```

This function gets the text related to the passed in NID value.

Parameters:

- **name** WOLFSSL_X509_NAME to search for text.
- **nid** NID to search for.
- **buf** buffer to hold text when found.
- **len** length of buffer.

See: none

Return: int returns the size of the text buffer.

Example

```
WOLFSSL_X509_NAME* name;
char buffer[100];
int bufferSz;
int ret;
// get WOLFSSL_X509_NAME
ret = wolfSSL_X509_NAME_get_text_by_NID(name, NID_commonName,
buffer, bufferSz);

//check ret value
```

C.52.2.145 function wolfSSL_X509_get_signature_type

```
int wolfSSL_X509_get_signature_type(
    WOLFSSL_X509 * cert
)
```

This function returns the value stored in the sigOID member of the WOLFSSL_X509 structure.

Parameters:

- **cert** a pointer to a WOLFSSL_X509 structure.

See:

- wolfSSL_X509_get_signature
- wolfSSL_X509_version
- wolfSSL_X509_get_der
- wolfSSL_X509_get_serial_number
- wolfSSL_X509_notBefore
- wolfSSL_X509_notAfter
- wolfSSL_X509_free

Return:

- 0 returned if the WOLFSSL_X509 structure is NULL.
- int an integer value is returned which was retrieved from the x509 object.

Example

```
WOLFSSL_X509 x509 = (WOLFSSL_X509*)XMALLOC(sizeof(WOLFSSL_X509), NULL,
                        DYNAMIC_TYPE_X509);
...
int x509SigType = wolfSSL_X509_get_signature_type(x509);

if(x509SigType != EXPECTED){
// Deal with an unexpected value
}
```

C.52.2.146 function wolfSSL_X509_free

```
void wolfSSL_X509_free(
    WOLFSSL_X509 * x509
)
```

This function frees a WOLFSSL_X509 structure.

Parameters:

- **x509** a pointer to the WOLFSSL_X509 struct.

See:

- wolfSSL_X509_get_signature
- wolfSSL_X509_version
- wolfSSL_X509_get_der
- wolfSSL_X509_get_serial_number
- wolfSSL_X509_notBefore
- wolfSSL_X509_notAfter

Example

```
WOLFSSL_X509* x509 = (WOLFSSL_X509*)XMALOC(sizeof(WOLFSSL_X509), NULL,
DYNAMIC_TYPE_X509) ;

wolfSSL_X509_free(x509);
```

C.52.2.147 function wolfSSL_X509_get_signature

```
int wolfSSL_X509_get_signature(
    WOLFSSL_X509 * x509,
    unsigned char * buf,
    int * bufSz
)
```

Gets the X509 signature and stores it in the buffer.

Parameters:

- **x509** pointer to a WOLFSSL_X509 structure.
- **buf** a char pointer to the buffer.
- **bufSz** an integer pointer to the size of the buffer.

See:

- wolfSSL_X509_get_serial_number
- wolfSSL_X509_get_signature_type
- wolfSSL_X509_get_device_type

Return:

- SSL_SUCCESS returned if the function successfully executes. The signature is loaded into the buffer.
- SSL_FATAL_ERRROR returns if the x509 struct or the bufSz member is NULL. There is also a check for the length member of the sig structure (sig is a member of x509).

Example

```
WOLFSSL_X509* x509 = (WOLFSSL_X509)XMALOC(sizeof(WOLFSSL_X509), NULL,
DYNAMIC_TYPE_X509);
unsigned char* buf; // Initialize
int* bufSz = sizeof(buf)/sizeof(unsigned char);
...
if(wolfSSL_X509_get_signature(x509, buf, bufSz) != SSL_SUCCESS){
    // The function did not execute successfully.
} else{
    // The buffer was written to correctly.
}
```

C.52.2.148 function wolfSSL_X509_STORE_add_cert

```
int wolfSSL_X509_STORE_add_cert(
    WOLFSSL_X509_STORE * store,
    WOLFSSL_X509 * x509
)
```

This function adds a certificate to the WOLFSSL_X509_STRE structure.

Parameters:

- **str** certificate store to add the certificate to.
- **x509** certificate to add.

See: wolfSSL_X509_free

Return:

- SSL_SUCCESS If certificate is added successfully.
- SSL_FATAL_ERROR: If certificate is not added successfully.

Example

```
WOLFSSL_X509_STORE* str;
WOLFSSL_X509* x509;
int ret;
ret = wolfSSL_X509_STORE_add_cert(str, x509);
//check ret value
```

C.52.2.149 function wolfSSL_X509_STORE_CTX_get_chain

```
WOLFSSL_STACK * wolfSSL_X509_STORE_CTX_get_chain(
    WOLFSSL_X509_STORE_CTX * ctx
)
```

This function is a getter function for chain variable in WOLFSSL_X509_STORE_CTX structure. Currently chain is not populated.

Parameters:

- **ctx** certificate store ctx to get parse chain from.

See: wolfSSL_sk_X509_free

Return:

- pointer if successful returns WOLFSSL_STACK (same as STACK_OF(WOLFSSL_X509)) pointer
- Null upon failure

Example

```
WOLFSSL_STACK* sk;
WOLFSSL_X509_STORE_CTX* ctx;
sk = wolfSSL_X509_STORE_CTX_get_chain(ctx);
//check sk for NULL and then use it. sk needs freed after done.
```

C.52.2.150 function wolfSSL_X509_STORE_set_flags

```
int wolfSSL_X509_STORE_set_flags(
    WOLFSSL_X509_STORE * store,
    unsigned long flag
)
```

This function takes in a flag to change the behavior of the WOLFSSL_X509_STORE structure passed in. An example of a flag used is WOLFSSL_CRL_CHECK.

Parameters:

- **str** certificate store to set flag in.
- **flag** flag for behavior.

See:

- wolfSSL_X509_STORE_new
- wolfSSL_X509_STORE_free

Return:

- SSL_SUCCESS If no errors were encountered when setting the flag.
- <0 a negative value will be returned upon failure.

Example

```
WOLFSSL_X509_STORE* str;
int ret;
// create and set up str
ret = wolfSSL_X509_STORE_set_flags(str, WOLFSSL_CRL_CHECKALL);
If (ret != SSL_SUCCESS) {
    //check ret value and handle error case
}
```

C.52.2.151 function wolfSSL_X509_notBefore

```
const byte * wolfSSL_X509_notBefore(
    WOLFSSL_X509 * x509
)
```

This function the certificate “not before” validity encoded as a byte array.

Parameters:

- **x509** pointer to a WOLFSSL_X509 structure.

See:

- wolfSSL_X509_get_signature
- wolfSSL_X509_version
- wolfSSL_X509_get_der
- wolfSSL_X509_get_serial_number
- wolfSSL_X509_notAfter
- wolfSSL_X509_free

Return:

- NULL returned if the WOLFSSL_X509 structure is NULL.
- byte is returned that contains the notBeforeData.

Example

```
WOLFSSL_X509* x509 = (WOLFSSL_X509*)XMALLOC(sizeof(WOLFSSL_X509), NULL,
                        DYNAMIC_TYPE_X509);
...
byte* notBeforeData = wolfSSL_X509_notBefore(x509);
```

C.52.2.152 function wolfSSL_X509_notAfter

```
const byte * wolfSSL_X509_notAfter(
    WOLFSSL_X509 * x509
)
```

This function the certificate “not after” validity encoded as a byte array.

Parameters:

- **x509** pointer to a WOLFSSL_X509 structure.

See:

- wolfSSL_X509_get_signature
- wolfSSL_X509_version
- wolfSSL_X509_get_der
- wolfSSL_X509_get_serial_number
- wolfSSL_X509_notBefore
- wolfSSL_X509_free

Return:

- NULL returned if the WOLFSSL_X509 structure is NULL.
- byte is returned that contains the notAfterData.

Example

```
WOLFSSL_X509* x509 = (WOLFSSL_X509*)XMALLOC(sizeof(WOLFSSL_X509), NULL,
                        DYNAMIC_TYPE_X509);
...
byte* notAfterData = wolfSSL_X509_notAfter(x509);
```

C.52.2.153 function wolfSSL_ASN1_INTEGER_to_BN

```
WOLFSSL_BIGNUM * wolfSSL_ASN1_INTEGER_to_BN(
    const WOLFSSL_ASN1_INTEGER * ai,
    WOLFSSL_BIGNUM * bn
)
```

This function is used to copy a WOLFSSL_ASN1_INTEGER value to a WOLFSSL_BIGNUM structure.

Parameters:

- **ai** WOLFSSL_ASN1_INTEGER structure to copy from.
- **bn** if wanting to copy into an already existing WOLFSSL_BIGNUM struct then pass in a pointer to it. Optionally this can be NULL and a new WOLFSSL_BIGNUM structure will be created.

See: none

Return:

- pointer On successfully copying the WOLFSSL_ASN1_INTEGER value a WOLFSSL_BIGNUM pointer is returned.
- Null upon failure.

Example

```
WOLFSSL_ASN1_INTEGER* ai;
WOLFSSL_BIGNUM* bn;
// create ai
bn = wolfSSL_ASN1_INTEGER_to_BN(ai, NULL);
```

```
// or if having already created bn and wanting to reuse structure
// wolfSSL_ASN1_INTEGER_to_BN(ai, bn);
// check bn is or return value is not NULL
```

C.52.2.154 function wolfSSL_CTX_add_extra_chain_cert

```
long wolfSSL_CTX_add_extra_chain_cert(
    WOLFSSL_CTX * ctx,
    WOLFSSL_X509 * x509
)
```

This function adds the certificate to the internal chain being built in the WOLFSSL_CTX structure.

Parameters:

- **ctx** WOLFSSL_CTX structure to add certificate to.
- **x509** certificate to add to the chain.

See:

- wolfSSL_CTX_new
- wolfSSL_CTX_free

Return:

- SSL_SUCCESS after successfully adding the certificate.
- SSL_FAILURE if failing to add the certificate to the chain.

Example

```
WOLFSSL_CTX* ctx;
WOLFSSL_X509* x509;
int ret;
// create ctx
ret = wolfSSL_CTX_add_extra_chain_cert(ctx, x509);
// check ret value
```

C.52.2.155 function wolfSSL_CTX_get_read_ahead

```
int wolfSSL_CTX_get_read_ahead(
    WOLFSSL_CTX * ctx
)
```

This function returns the get read ahead flag from a WOLFSSL_CTX structure.

Parameters:

- **ctx** WOLFSSL_CTX structure to get read ahead flag from.

See:

- wolfSSL_CTX_new
- wolfSSL_CTX_free
- wolfSSL_CTX_set_read_ahead

Return:

- flag On success returns the read ahead flag.
- SSL_FAILURE If ctx is NULL then SSL_FAILURE is returned.

Example

```
WOLFSSL_CTX* ctx;
int flag;
// setup ctx
flag = wolfSSL_CTX_get_read_ahead(ctx);
//check flag
```

C.52.2.156 function wolfSSL_CTX_set_read_ahead

```
int wolfSSL_CTX_set_read_ahead(
    WOLFSSL_CTX * ctx,
    int v
)
```

This function sets the read ahead flag in the WOLFSSL_CTX structure.

Parameters:

- **ctx** WOLFSSL_CTX structure to set read ahead flag.
- **v** read ahead flag

See:

- wolfSSL_CTX_new
- wolfSSL_CTX_free
- wolfSSL_CTX_get_read_ahead

Return:

- SSL_SUCCESS If ctx read ahead flag set.
- SSL_FAILURE If ctx is NULL then SSL_FAILURE is returned.

Example

```
WOLFSSL_CTX* ctx;
int flag;
int ret;
// setup ctx
ret = wolfSSL_CTX_set_read_ahead(ctx, flag);
// check return value
```

C.52.2.157 function wolfSSL_CTX_set_tlsext_status_arg

```
long wolfSSL_CTX_set_tlsext_status_arg(
    WOLFSSL_CTX * ctx,
    void * arg
)
```

This function sets the options argument to use with OCSP.

Parameters:

- **ctx** WOLFSSL_CTX structure to set user argument.
- **arg** user argument.

See:

- wolfSSL_CTX_new
- wolfSSL_CTX_free

Return:

- SSL_FAILURE If ctx or it's cert manager is NULL.

- SSL_SUCCESS If successfully set.

Example

```
WOLFSSL_CTX* ctx;
void* data;
int ret;
// setup ctx
ret = wolfSSL_CTX_set_tlsext_status_arg(ctx, data);

//check ret value
```

C.52.2.158 function wolfSSL_CTX_set_tlsext_opaque_prf_input_callback_arg

```
long wolfSSL_CTX_set_tlsext_opaque_prf_input_callback_arg(
    WOLFSSL_CTX * ctx,
    void * arg
)
```

This function sets the optional argument to be passed to the PRF callback.

Parameters:

- **ctx** WOLFSSL_CTX structure to set user argument.
- **arg** user argument.

See:

- wolfSSL_CTX_new
- wolfSSL_CTX_free

Return:

- SSL_FAILURE If ctx is NULL.
- SSL_SUCCESS If successfully set.

Example

```
WOLFSSL_CTX* ctx;
void* data;
int ret;
// setup ctx
ret = wolfSSL_CTX_set_tlsext_opaques_prf_input_callback_arg(ctx, data);
//check ret value
```

C.52.2.159 function wolfSSL_set_options

```
long wolfSSL_set_options(
    WOLFSSL * s,
    long op
)
```

This function sets the options mask in the ssl. Some valid options are, SSL_OP_ALL, SSL_OP_COOKIE_EXCHANGE, SSL_OP_NO_SSLv2, SSL_OP_NO_SSLv3, SSL_OP_NO_TLSv1, SSL_OP_NO_TLSv1_1, SSL_OP_NO_TLSv1_2, SSL_OP_NO_COMPRESSION.

Parameters:

- **s** WOLFSSL structure to set options mask.

- **op** This function sets the options mask in the ssl. Some valid options are: SSL_OP_ALL SSL_OP_COOKIE_EXCHANGE SSL_OP_NO_SSLv2 SSL_OP_NO_SSLv3 SSL_OP_NO_TLSv1 SSL_OP_NO_TLSv1_1 SSL_OP_NO_TLSv1_2 SSL_OP_NO_COMPRESSION

See:

- wolfSSL_new
- wolfSSL_free
- wolfSSL_get_options

Return: val Returns the updated options mask value stored in ssl.

Example

```
WOLFSSL* ssl;
unsigned long mask;
mask = SSL_OP_NO_TLSv1
mask  = wolfSSL_set_options(ssl, mask);
// check mask
```

C.52.2.160 function wolfSSL_get_options

```
long wolfSSL_get_options(
    const WOLFSSL * ssl
)
```

This function returns the current options mask.

Parameters:

- **ssl** WOLFSSL structure to get options mask from.

See:

- wolfSSL_new
- wolfSSL_free
- wolfSSL_set_options

Return: val Returns the mask value stored in ssl.

Example

```
WOLFSSL* ssl;
unsigned long mask;
mask  = wolfSSL_get_options(ssl);
// check mask
```

C.52.2.161 function wolfSSL_set_tlsext_debug_arg

```
long wolfSSL_set_tlsext_debug_arg(
    WOLFSSL * ssl,
    void * arg
)
```

This is used to set the debug argument passed around.

Parameters:

- **ssl** WOLFSSL structure to set argument in.
- **arg** argument to use.

See:

- wolfSSL_new
- wolfSSL_free

Return:

- SSL_SUCCESS On successful setting argument.
- SSL_FAILURE If an NULL ssl passed in.

Example

```
WOLFSSL* ssl;
void* args;
int ret;
// create ssl object
ret  = wolfSSL_set_tlsext_debug_arg(ssl, args);
// check ret value
```

C.52.2.162 function wolfSSL_set_tlsext_status_type

```
long wolfSSL_set_tlsext_status_type(
    WOLFSSL * s,
    int type
)
```

This function is called when the client application request that a server send back an OCSP status response (also known as OCSP stapling).Currently, the only supported type is TLSEXT_STATUSTYPE_ocsp.

Parameters:

- **s** pointer to WOLFSSL struct which is created by SSL_new() function
- **type** ssl extension type which TLSEXT_STATUSTYPE_ocsp is only supported.

See:

- wolfSSL_new
- wolfSSL_CTX_new
- wolfSSL_free
- wolfSSL_CTX_free

Return:

- 1 upon success.
- 0 upon error.

Example

```
WOLFSSL *ssl;
WOLFSSL_CTX *ctx;
int ret;
ctx = wolfSSL_CTX_new(wolfSSLv23_server_method());
ssl = wolfSSL_new(ctx);
ret = WolfSSL_set_tlsext_status_type(ssl,TLSEXT_STATUSTYPE_ocsp);
wolfSSL_free(ssl);
wolfSSL_CTX_free(ctx);
```

C.52.2.163 function wolfSSL_get_verify_result

```
long wolfSSL_get_verify_result(
    const WOLFSSL * ssl
)
```

This is used to get the results after trying to verify the peer's certificate.

Parameters:

- **ssl** WOLFSSL structure to get verification results from.

See:

- wolfSSL_new
- wolfSSL_free

Return:

- X509_V_OK On successful verification.
- SSL_FAILURE If an NULL ssl passed in.

Example

```
WOLFSSL* ssl;
long ret;
// attempt/complete handshake
ret  = wolfSSL_get_verify_result(ssl);
// check ret value
```

C.52.2.164 function wolfSSL_ERR_print_errors_fp

```
void wolfSSL_ERR_print_errors_fp(
    XFILE fp,
    int err
)
```

This function converts an error code returned by wolfSSL_get_error() and fp is the file which the error string will be placed in.

Parameters:

- **fp** output file for human-readable error string to be written to.
- **err** error code returned by wolfSSL_get_error().

See:

- wolfSSL_get_error
- wolfSSL_ERR_error_string
- wolfSSL_ERR_error_string_n
- wolfSSL_load_error_strings

Return: none No returns.

Example

```
int err = 0;
WOLFSSL* ssl;
FILE* fp = ...
...
err = wolfSSL_get_error(ssl, 0);
wolfSSL_ERR_print_errors_fp(fp, err);
```

C.52.2.165 function wolfSSL_ERR_print_errors_cb

```
void wolfSSL_ERR_print_errors_cb(
    int(*)(const char *str, size_t len, void *u) cb,
```

```
    void * u
)
```

This function uses the provided callback to handle error reporting. The callback function is executed for each error line. The string, length, and userdata are passed into the callback parameters.

Parameters:

- **cb** the callback function.
- **u** userdata to pass into the callback function.

See:

- wolfSSL_get_error
- wolfSSL_ERR_error_string
- wolfSSL_ERR_error_string_n
- wolfSSL_load_error_strings

Return: none No returns.

Example

```
int error_cb(const char *str, size_t len, void *u)
{ fprintf((FILE*)u, "%-*.*s\n", (int)len, (int)len, str); return 0; }
...
FILE* fp = ...
wolfSSL_ERR_print_errors_cb(error_cb, fp);
```

C.52.2.166 function wolfSSL_CTX_set_psk_client_callback

```
void wolfSSL_CTX_set_psk_client_callback(
    WOLFSSL_CTX * ctx,
    wc_psk_client_callback cb
)
```

The function sets the client_psk_cb member of the WOLFSSL_CTX structure.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().
- **cb** wc_psk_client_callback is a function pointer that will be stored in the WOLFSSL_CTX structure. Return value is the key length on success or zero on error. unsigned int (*wc_psk_client_callback) *PSK client callback parameters: WOLFSSL* ssl - Pointer to the wolfSSL structure const char* hint - A stored string that could be displayed to provide a hint to the user. char* identity - The ID will be stored here. unsigned int id_max_len - Size of the ID buffer. unsigned char* key - The key will be stored here. unsigned int key_max_len - The max size of the key.

See:

- wolfSSL_set_psk_client_callback
- wolfSSL_set_psk_server_callback
- wolfSSL_CTX_set_psk_server_callback
- wolfSSL_CTX_set_psk_client_callback

Return: none No returns.

Example

```
WOLFSSL_CTX* ctx = WOLFSSL_CTX_new( protocol def );
…
static WC_INLINE unsigned int my_psk_client_cb(WOLFSSL* ssl, const char* hint,
char* identity, unsigned int id_max_len, unsigned char* key,
```

```
Unsigned int key_max_len){
…
wolfSSL_CTX_set_psk_client_callback(ctx, my_psk_client_cb);
```

C.52.2.167 function wolfSSL_set_psk_client_callback

```
void wolfSSL_set_psk_client_callback(
    WOLFSSL * ssl,
    wc_psk_client_callback
)
```

Sets the PSK client side callback.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **cb** a function pointer to type wc_psk_client_callback. Return value is the key length on success or zero on error. unsigned int (*wc_psk_client_callback) PSK client callback parameters: WOLFSSL ssl - Pointer to the wolfSSL structure const char* hint - A stored string that could be displayed to provide a hint to the user. char* identity - The ID will be stored here. unsigned int id_max_len - Size of the ID buffer. unsigned char* key - The key will be stored here. unsigned int key_max_len - The max size of the key.

See:

- wolfSSL_CTX_set_psk_client_callback
- wolfSSL_CTX_set_psk_server_callback
- wolfSSL_set_psk_server_callback

Return: none No returns.

Example

```
WOLFSSL* ssl;
static WC_INLINE unsigned int my_psk_client_cb(WOLFSSL* ssl, const char* hint,
char* identity, unsigned int id_max_len, unsigned char* key,
Unsigned int key_max_len){
…
if(ssl){
wolfSSL_set_psk_client_callback(ssl, my_psk_client_cb);
} else {
    // could not set callback
}
```

C.52.2.168 function wolfSSL_get_psk_identity_hint

```
const char * wolfSSL_get_psk_identity_hint(
    const WOLFSSL *
)
```

This function returns the psk identity hint.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See: wolfSSL_get_psk_identity

Return:

- pointer a const char pointer to the value that was stored in the arrays member of the WOLFSSL structure is returned.
- NULL returned if the WOLFSSL or Arrays structures are NULL.

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
char* idHint;
...
idHint = wolfSSL_get_psk_identity_hint(ssl);
if(idHint){
    // The hint was retrieved
    return idHint;
} else {
    // Hint wasn't successfully retrieved
}
```

C.52.2.169 function wolfSSL_get_psk_identity

```
const char * wolfSSL_get_psk_identity(
    const WOLFSSL *
)
```

The function returns a constant pointer to the client_identity member of the Arrays structure.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_get_psk_identity_hint
- wolfSSL_use_psk_identity_hint

Return:

- string the string value of the client_identity member of the Arrays structure.
- NULL if the WOLFSSL structure is NULL or if the Arrays member of the WOLFSSL structure is NULL.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
const char* pskID;
...
pskID = wolfSSL_get_psk_identity(ssl);

if(pskID == NULL){
    // There is not a value in pskID
}
```

C.52.2.170 function wolfSSL_CTX_use_psk_identity_hint

```
int wolfSSL_CTX_use_psk_identity_hint(
    WOLFSSL_CTX * ctx,
    const char * hint
)
```

This function stores the hint argument in the server_hint member of the WOLFSSL_CTX structure.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().
- **hint** a constant char pointer that will be copied to the WOLFSSL_CTX structure.

See: wolfSSL_use_psk_identity_hint

Return: SSL_SUCCESS returned for successful execution of the function.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
const char* hint;
int ret;
…
ret = wolfSSL_CTX_use_psk_identity_hint(ctx, hint);
if(ret == SSL_SUCCESS){
    // Function was successful.
return ret;
} else {
    // Failure case.
}
```

C.52.2.171 function wolfSSL_use_psk_identity_hint

```
int wolfSSL_use_psk_identity_hint(
    WOLFSSL * ssl,
    const char * hint
)
```

This function stores the hint argument in the server_hint member of the Arrays structure within the WOLFSSL structure.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **hint** a constant character pointer that holds the hint to be saved in memory.

See: wolfSSL_CTX_use_psk_identity_hint

Return:

- SSL_SUCCESS returned if the hint was successfully stored in the WOLFSSL structure.
- SSL_FAILURE returned if the WOLFSSL or Arrays structures are NULL.

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
const char* hint;
...
if(wolfSSL_use_psk_identity_hint(ssl, hint) != SSL_SUCCESS){
    // Handle failure case.
}
```

C.52.2.172 function wolfSSL_CTX_set_psk_server_callback

```
void wolfSSL_CTX_set_psk_server_callback(
    WOLFSSL_CTX * ctx,
    wc_psk_server_callback cb
)
```

This function sets the psk callback for the server side in the WOLFSSL_CTX structure.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **cb** a function pointer for the callback and will be stored in the WOLFSSL_CTX structure. Return value is the key length on success or zero on error. unsigned int (*wc_psk_server_callback) PSK server callback parameters WOLFSSL* ssl - Pointer to the wolfSSL structure char* identity - The ID will be stored here. unsigned char* key - The key will be stored here. unsigned int key_max_len - The max size of the key.

See:

- wc_psk_server_callback
- wolfSSL_set_psk_client_callback
- wolfSSL_set_psk_server_callback
- wolfSSL_CTX_set_psk_client_callback

Return: none No returns.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
WOLFSSL* ssl = wolfSSL_new(ctx);
…
static unsigned int my_psk_server_cb(WOLFSSL* ssl, const char* identity,
                        unsigned char* key, unsigned int key_max_len)
{
    // Function body.
}
…
if(ctx != NULL){
    wolfSSL_CTX_set_psk_server_callback(ctx, my_psk_server_cb);
} else {
    // The CTX object was not properly initialized.
}
```

C.52.2.173 function wolfSSL_set_psk_server_callback

```
void wolfSSL_set_psk_server_callback(
    WOLFSSL * ssl,
    wc_psk_server_callback cb
)
```

Sets the psk callback for the server side by setting the WOLFSSL structure options members.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **cb** a function pointer for the callback and will be stored in the WOLFSSL structure. Return value is the key length on success or zero on error. unsigned int (*wc_psk_server_callback) PSK server callback parameters WOLFSSL* ssl - Pointer to the wolfSSL structure char* identity - The ID will be stored here. unsigned char* key - The key will be stored here. unsigned int key_max_len - The max size of the key.

See:

- wolfSSL_set_psk_client_callback
- wolfSSL_CTX_set_psk_server_callback
- wolfSSL_CTX_set_psk_client_callback
- wolfSSL_get_psk_identity_hint
- wc_psk_server_callback

- InitSuites

Return: none No returns.

Example

```
WOLFSSL_CTX* ctx;
WOLFSSL* ssl;
…
static unsigned int my_psk_server_cb(WOLFSSL* ssl, const char* identity,
                        unsigned char* key, unsigned int key_max_len)
{
    // Function body.
}
…
if(ssl != NULL && cb != NULL){
    wolfSSL_set_psk_server_callback(ssl, my_psk_server_cb);
}
```

C.52.2.174 function wolfSSL_set_psk_callback_ctx

```
int wolfSSL_set_psk_callback_ctx(
    WOLFSSL * ssl,
    void * psk_ctx
)
```

Sets a PSK user context in the WOLFSSL structure options member.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **psk_ctx** void pointer to user PSK context

See:

- wolfSSL_get_psk_callback_ctx
- wolfSSL_CTX_set_psk_callback_ctx
- wolfSSL_CTX_get_psk_callback_ctx

Return: WOLFSSL_SUCCESS or WOLFSSL_FAILURE

C.52.2.175 function wolfSSL_CTX_set_psk_callback_ctx

```
int wolfSSL_CTX_set_psk_callback_ctx(
    WOLFSSL_CTX * ctx,
    void * psk_ctx
)
```

Sets a PSK user context in the WOLFSSL_CTX structure.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().
- **psk_ctx** void pointer to user PSK context

See:

- wolfSSL_set_psk_callback_ctx
- wolfSSL_get_psk_callback_ctx
- wolfSSL_CTX_get_psk_callback_ctx

Return: WOLFSSL_SUCCESS or WOLFSSL_FAILURE

C.52.2.176 function wolfSSL_get_psk_callback_ctx

```
void * wolfSSL_get_psk_callback_ctx(
    WOLFSSL * ssl
)
```

Get a PSK user context in the WOLFSSL structure options member.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_set_psk_callback_ctx
- wolfSSL_CTX_set_psk_callback_ctx
- wolfSSL_CTX_get_psk_callback_ctx

Return: void pointer to user PSK context

C.52.2.177 function wolfSSL_CTX_get_psk_callback_ctx

```
void * wolfSSL_CTX_get_psk_callback_ctx(
    WOLFSSL_CTX * ctx
)
```

Get a PSK user context in the WOLFSSL_CTX structure.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().

See:

- wolfSSL_CTX_set_psk_callback_ctx
- wolfSSL_set_psk_callback_ctx
- wolfSSL_get_psk_callback_ctx

Return: void pointer to user PSK context

C.52.2.178 function wolfSSL_CTX_allow_anon_cipher

```
int wolfSSL_CTX_allow_anon_cipher(
    WOLFSSL_CTX *
)
```

This function enables the havAnon member of the CTX structure if HAVE_ANON is defined during compilation.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().

See: none

Return:

- SSL_SUCCESS returned if the function executed successfully and the haveAnnon member of the CTX is set to 1.
- SSL_FAILURE returned if the CTX structure was NULL.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
WOLFSSL* ssl = wolfSSL_new(ctx);
...
#ifdef HAVE_ANON
if(cipherList == NULL){
    wolfSSL_CTX_allow_anon_cipher(ctx);
    if(wolfSSL_CTX_set_cipher_list(ctx, “ADH_AES128_SHA”) != SSL_SUCCESS){
        // failure case
    }
}
#endif
```

C.52.2.179 function wolfSSLv23_server_method

```
WOLFSSL_METHOD * wolfSSLv23_server_method(
    void
)
```

The wolfSSLv23_server_method().

Parameters:

- **none** No parameters

See:

- wolfSSLv3_server_method
- wolfTLSv1_server_method
- wolfTLSv1_1_server_method
- wolfTLSv1_2_server_method
- wolfTLSv1_3_server_method
- wolfDTLSv1_server_method
- wolfSSL_CTX_new

Return:

- pointer If successful, the call will return a pointer to the newly created WOLFSSL_METHOD structure.
- Failure If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Example

```
WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfSSLv23_server_method();
if (method == NULL) {
    // unable to get method
}

ctx = wolfSSL_CTX_new(method);
...
```

C.52.2.180 function wolfSSL_state

```
int wolfSSL_state(
    WOLFSSL * ssl
)
```

This is used to get the internal error state of the WOLFSSL structure.

See:

- wolfSSL_new
- wolfSSL_free

Return:

- wolfssl_error returns ssl error state, usually a negative
- BAD_FUNC_ARG if ssl is NULL.
- ssl WOLFSSL structure to get state from.

Example

```
WOLFSSL* ssl;
int ret;
// create ssl object
ret  = wolfSSL_state(ssl);
// check ret value
```

C.52.2.181 function wolfSSL_get_peer_certificate

```
WOLFSSL_X509 * wolfSSL_get_peer_certificate(
    WOLFSSL * ssl
)
```

This function gets the peer's certificate.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_X509_get_issuer_name
- wolfSSL_X509_get_subject_name
- wolfSSL_X509_get_isCA

Return:

- pointer a pointer to the peerCert member of the WOLFSSL_X509 structure if it exists.
- 0 returned if the peer certificate issuer size is not defined.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
...
WOLFSSL_X509* peerCert = wolfSSL_get_peer_certificate(ssl);

if(peerCert){
    // You have a pointer peerCert to the peer certification
}
```

C.52.2.182 function wolfSSL_want_read

```
int wolfSSL_want_read(
    WOLFSSL *
)
```

This function is similar to calling wolfSSL_get_error() and getting SSL_ERROR_WANT_READ in return. If the underlying error state is SSL_ERROR_WANT_READ, this function will return 1, otherwise, 0.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().

See:

- wolfSSL_want_write
- wolfSSL_get_error

Return:

- 1 wolfSSL_get_error() would return SSL_ERROR_WANT_READ, the underlying I/O has data available for reading.
- 0 There is no SSL_ERROR_WANT_READ error state.

Example

```
int ret;
WOLFSSL* ssl = 0;
...

ret = wolfSSL_want_read(ssl);
if (ret == 1) {
    // underlying I/O has data available for reading (SSL_ERROR_WANT_READ)
}
```

C.52.2.183 function wolfSSL_want_write

```
int wolfSSL_want_write(
    WOLFSSL *
)
```

This function is similar to calling wolfSSL_get_error() and getting SSL_ERROR_WANT_WRITE in return. If the underlying error state is SSL_ERROR_WANT_WRITE, this function will return 1, otherwise, 0.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().

See:

- wolfSSL_want_read
- wolfSSL_get_error

Return:

- 1 wolfSSL_get_error() would return SSL_ERROR_WANT_WRITE, the underlying I/O needs data to be written in order for progress to be made in the underlying SSL connection.
- 0 There is no SSL_ERROR_WANT_WRITE error state.

Example

```
int ret;
WOLFSSL* ssl = 0;
...
ret = wolfSSL_want_write(ssl);
if (ret == 1) {
    // underlying I/O needs data to be written (SSL_ERROR_WANT_WRITE)
}
```

C.52.2.184 function wolfSSL_check_domain_name

```
int wolfSSL_check_domain_name(
    WOLFSSL * ssl,
    const char * dn
)
```

wolfSSL by default checks the peer certificate for a valid date range and a verified signature. Calling this function before wolfSSL_connect() will add a domain name check to the list of checks to perform. dn holds the domain name to check against the peer certificate when it's received.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **dn** domain name to check against the peer certificate when received.

See: none

Return:

- SSL_SUCCESS upon success.
- SSL_FAILURE will be returned if a memory error was encountered.

Example

```
int ret = 0;
WOLFSSL* ssl;
char* domain = (char*) "www.yassl.com";
...

ret = wolfSSL_check_domain_name(ssl, domain);
if (ret != SSL_SUCCESS) {
   // failed to enable domain name check
}
```

C.52.2.185 function wolfSSL_Init

```
int wolfSSL_Init(
    void
)
```

Initializes the wolfSSL library for use. Must be called once per application and before any other call to the library.

See: wolfSSL_Cleanup

Return:

- SSL_SUCCESS If successful the call will return.
- BAD_MUTEX_E is an error that may be returned.
- WC_INIT_E wolfCrypt initialization error returned.

Example

```
int ret = 0;
ret = wolfSSL_Init();
if (ret != SSL_SUCCESS) {
    failed to initialize wolfSSL library
}
```

C.52.2.186 function wolfSSL_Cleanup

```
int wolfSSL_Cleanup(
    void
)
```

Un-initializes the wolfSSL library from further use. Doesn't have to be called, though it will free any resources used by the library.

See: wolfSSL_Init

Return:

- SSL_SUCCESS return no errors.
- BAD_MUTEX_E a mutex error return.]

Example

```
wolfSSL_Cleanup();
```

C.52.2.187 function wolfSSL_lib_version

```
const char * wolfSSL_lib_version(
    void
)
```

This function returns the current library version.

Parameters:

- **none** No parameters.

See: word32_wolfSSL_lib_version_hex

Return: LIBWOLFSSL_VERSION_STRING a const char pointer defining the version.

Example

```
char version[MAXSIZE];
version = wolfSSL_KeepArrays();
…
if(version != ExpectedVersion){
    // Handle the mismatch case
}
```

C.52.2.188 function wolfSSL_lib_version_hex

```
word32 wolfSSL_lib_version_hex(
    void
)
```

This function returns the current library version in hexadecimal notation.

Parameters:

- **none** No parameters.

See: wolfSSL_lib_version

Return: LILBWOLFSSL_VERSION_HEX returns the hexadecimal version defined in wolfssl/version.h.

Example

```
word32 libV;
libV = wolfSSL_lib_version_hex();

if(libV != EXPECTED_HEX){
    // How to handle an unexpected value
} else {
    // The expected result for libV
}
```

C.52.2.189 function wolfSSL_negotiate

```
int wolfSSL_negotiate(
    WOLFSSL * ssl
)
```

Performs the actual connect or accept based on the side of the SSL method. If called from the client side then an wolfSSL_connect() is performed if called from the server side.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().

See:

- SSL_connect
- SSL_accept

Return:

- SSL_SUCCESS will be returned if successful. (Note, older versions will return 0.)
- SSL_FATAL_ERROR will be returned if the underlying call resulted in an error. Use wolfSSL_get_error() to get a specific error code.

Example

```
int ret = SSL_FATAL_ERROR;
WOLFSSL* ssl = 0;
...
ret = wolfSSL_negotiate(ssl);
if (ret == SSL_FATAL_ERROR) {
    // SSL establishment failed
int error_code = wolfSSL_get_error(ssl);
...
}
...
```

C.52.2.190 function wolfSSL_set_compression

```
int wolfSSL_set_compression(
    WOLFSSL * ssl
)
```

Turns on the ability to use compression for the SSL connection. Both sides must have compression turned on otherwise compression will not be used. The zlib library performs the actual data compression. To compile into the library use –with-libz for the configure system and define HAVE_LIBZ otherwise. Keep in mind that while compressing data before sending decreases the actual size of the messages being sent and received, the amount of data saved by compression usually takes longer in time to analyze than it does to send it raw on all but the slowest of networks.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().

See: none

Return:

- SSL_SUCCESS upon success.
- NOT_COMPILED_IN will be returned if compression support wasn't built into the library.

Example

```
int ret = 0;
WOLFSSL* ssl = 0;
...
ret = wolfSSL_set_compression(ssl);
if (ret == SSL_SUCCESS) {
    // successfully enabled compression for SSL session
}
```

C.52.2.191 function wolfSSL_set_timeout

```
int wolfSSL_set_timeout(
    WOLFSSL * ssl,
    unsigned int to
)
```

This function sets the SSL session timeout value in seconds.

Parameters:

- **ssl** pointer to the SSL object, created with wolfSSL_new().
- **to** value, in seconds, used to set the SSL session timeout.

See:

- wolfSSL_get1_session
- wolfSSL_set_session

Return:

- SSL_SUCCESS will be returned upon successfully setting the session.
- BAD_FUNC_ARG will be returned if ssl is NULL.

Example

```
int ret = 0;
WOLFSSL* ssl = 0;
...

ret = wolfSSL_set_timeout(ssl, 500);
if (ret != SSL_SUCCESS) {
    // failed to set session timeout value
}
...
```

C.52.2.192 function wolfSSL_CTX_set_timeout

```
int wolfSSL_CTX_set_timeout(
    WOLFSSL_CTX * ctx,
    unsigned int to
)
```

This function sets the timeout value for SSL sessions, in seconds, for the specified SSL context.

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().
- **to** session timeout value in seconds.

See:

- wolfSSL_flush_sessions
- wolfSSL_get1_session
- wolfSSL_set_session
- wolfSSL_get_sessionID
- wolfSSL_CTX_set_session_cache_mode

Return:

- the previous timeout value, if WOLFSSL_ERROR_CODE_OPENSSL is
- defined on success. If not defined, SSL_SUCCESS will be returned.
- BAD_FUNC_ARG will be returned when the input context (ctx) is null.

Example

```
WOLFSSL_CTX*    ctx    = 0;
...
ret = wolfSSL_CTX_set_timeout(ctx, 500);
if (ret != SSL_SUCCESS) {
    // failed to set session timeout value
}
```

C.52.2.193 function wolfSSL_get_peer_chain

```
WOLFSSL_X509_CHAIN * wolfSSL_get_peer_chain(
    WOLFSSL * ssl
)
```

Retrieves the peer's certificate chain.

Parameters:

- **ssl** pointer to a valid WOLFSSL structure.

See:

- wolfSSL_get_chain_count
- wolfSSL_get_chain_length
- wolfSSL_get_chain_cert
- wolfSSL_get_chain_cert_pem

Return:

- chain If successful the call will return the peer's certificate chain.
- 0 will be returned if an invalid WOLFSSL pointer is passed to the function.

Example

none

C.52.2.194 function wolfSSL_get_chain_count

```
int wolfSSL_get_chain_count(
    WOLFSSL_X509_CHAIN * chain
)
```

Retrieve's the peers certificate chain count.

Parameters:

- **chain** pointer to a valid WOLFSSL_X509_CHAIN structure.

See:

- wolfSSL_get_peer_chain
- wolfSSL_get_chain_length
- wolfSSL_get_chain_cert
- wolfSSL_get_chain_cert_pem

Return:

- Success If successful the call will return the peer's certificate chain count.
- 0 will be returned if an invalid chain pointer is passed to the function.

Example

none

C.52.2.195 function wolfSSL_get_chain_length

```
int wolfSSL_get_chain_length(
    WOLFSSL_X509_CHAIN * chain,
    int idx
)
```

Retrieves the peer's ASN1.DER certificate length in bytes at index (idx).

Parameters:

- **chain** pointer to a valid WOLFSSL_X509_CHAIN structure.
- **idx** index to start of chain.

See:

- wolfSSL_get_peer_chain
- wolfSSL_get_chain_count
- wolfSSL_get_chain_cert
- wolfSSL_get_chain_cert_pem

Return:

- Success If successful the call will return the peer's certificate length in bytes by index.
- 0 will be returned if an invalid chain pointer is passed to the function.

Example

none

C.52.2.196 function wolfSSL_get_chain_cert

```
unsigned char * wolfSSL_get_chain_cert(
    WOLFSSL_X509_CHAIN * chain,
    int idx
)
```

Retrieves the peer's ASN1.DER certificate at index (idx).

Parameters:

- **chain** pointer to a valid WOLFSSL_X509_CHAIN structure.

- **idx** index to start of chain.

See:

- wolfSSL_get_peer_chain
- wolfSSL_get_chain_count
- wolfSSL_get_chain_length
- wolfSSL_get_chain_cert_pem

Return:

- Success If successful the call will return the peer's certificate by index.
- 0 will be returned if an invalid chain pointer is passed to the function.

Example

none

C.52.2.197 function wolfSSL_get_chain_X509

```
WOLFSSL_X509 * wolfSSL_get_chain_X509(
    WOLFSSL_X509_CHAIN * chain,
    int idx
)
```

This function gets the peer's wolfSSL_X509_certificate at index (idx) from the chain of certificates.

Parameters:

- **chain** a pointer to the WOLFSSL_X509_CHAIN used for no dynamic memory SESSION_CACHE.
- **idx** the index of the WOLFSSL_X509 certificate.

See:

- InitDecodedCert
- ParseCertRelative
- CopyDecodedToX509

Return: pointer returns a pointer to a WOLFSSL_X509 structure.

Note that it is the user's responsibility to free the returned memory by calling wolfSSL_FreeX509().

Example

```
WOLFSSL_X509_CHAIN* chain = &session->chain;
int idx = 999; // set idx
...
WOLFSSL_X509_CHAIN ptr;
prt = wolfSSL_get_chain_X509(chain, idx);

if(ptr != NULL){
    // ptr contains the cert at the index specified
    wolfSSL_FreeX509(ptr);
} else {
    // ptr is NULL
}
```

C.52.2.198 function wolfSSL_get_chain_cert_pem

```
int wolfSSL_get_chain_cert_pem(
    WOLFSSL_X509_CHAIN * chain,
    int idx,
```

```
    unsigned char * buf,
    int inLen,
    int * outLen
)
```

Retrieves the peer's PEM certificate at index (idx).

Parameters:

- **chain** pointer to a valid WOLFSSL_X509_CHAIN structure.
- **idx** indexto start of chain.

See:

- wolfSSL_get_peer_chain
- wolfSSL_get_chain_count
- wolfSSL_get_chain_length
- wolfSSL_get_chain_cert

Return:

- Success If successful the call will return the peer's certificate by index.
- 0 will be returned if an invalid chain pointer is passed to the function.

Example

none

C.52.2.199 function wolfSSL_get_sessionID

```
const unsigned char * wolfSSL_get_sessionID(
    const WOLFSSL_SESSION * s
)
```

Retrieves the session's ID. The session ID is always 32 bytes long.

Parameters:

- **session** pointer to a valid wolfssl session.

See: SSL_get_session

Return: id The session ID.

Example

none

C.52.2.200 function wolfSSL_X509_get_serial_number

```
int wolfSSL_X509_get_serial_number(
    WOLFSSL_X509 * x509,
    unsigned char * in,
    int * inOutSz
)
```

Retrieves the peer's certificate serial number. The serial number buffer (in) should be at least 32 bytes long and be provided as the *inOutSz argument as input. After calling the function* inOutSz will hold the actual length in bytes written to the in buffer.

Parameters:

- **in** The serial number buffer and should be at least 32 bytes long
- **inOutSz** will hold the actual length in bytes written to the in buffer.

See: SSL_get_peer_certificate

Return:

- SSL_SUCCESS upon success.
- BAD_FUNC_ARG will be returned if a bad function argument was encountered.

Example

none

C.52.2.201 function wolfSSL_X509_get_subjectCN

```
char * wolfSSL_X509_get_subjectCN(
    WOLFSSL_X509 *
)
```

Returns the common name of the subject from the certificate.

Parameters:

- **x509** a pointer to a WOLFSSL_X509 structure containing certificate information.

See:

- wolfSSL_X509_Name_get_entry
- wolfSSL_X509_get_next_altname
- wolfSSL_X509_get_issuer_name
- wolfSSL_X509_get_subject_name

Return:

- NULL returned if the x509 structure is null
- string a string representation of the subject's common name is returned upon success

Example

```
WOLFSSL_X509 x509 = (WOLFSSL_X509*)XMALLOC(sizeof(WOLFSSL_X509), NULL,
                        DYNAMIC_TYPE_X509);
...
int x509Cn = wolfSSL_X509_get_subjectCN(x509);
if(x509Cn == NULL){
    // Deal with NULL case
} else {
    // x509Cn contains the common name
}
```

C.52.2.202 function wolfSSL_X509_get_der

```
const unsigned char * wolfSSL_X509_get_der(
    WOLFSSL_X509 * x509,
    int * outSz
)
```

This function gets the DER encoded certificate in the WOLFSSL_X509 struct.

Parameters:

- **x509** a pointer to a WOLFSSL_X509 structure containing certificate information.
- **outSz** length of the derBuffer member of the WOLFSSL_X509 struct.

See:

- wolfSSL_X509_version
- wolfSSL_X509_Name_get_entry
- wolfSSL_X509_get_next_altname
- wolfSSL_X509_get_issuer_name
- wolfSSL_X509_get_subject_name

Return:

- buffer This function returns the DerBuffer structure's buffer member, which is of type byte.
- NULL returned if the x509 or outSz parameter is NULL.

Example

```
WOLFSSL_X509 x509 = (WOLFSSL_X509*)XMALLOC(sizeof(WOLFSSL_X509), NULL,
                        DYNAMIC_TYPE_X509);
int* outSz; // initialize
...
byte* x509Der = wolfSSL_X509_get_der(x509, outSz);
if(x509Der == NULL){
    // Failure case one of the parameters was NULL
}
```

C.52.2.203 function wolfSSL_X509_get_notAfter

```
WOLFSSL_ASN1_TIME * wolfSSL_X509_get_notAfter(
    WOLFSSL_X509 *
)
```

This function checks to see if x509 is NULL and if it's not, it returns the notAfter member of the x509 struct.

Parameters:

- **x509** a pointer to the WOLFSSL_X509 struct.

See: wolfSSL_X509_get_notBefore

Return:

- pointer to struct with ASN1_TIME to the notAfter member of the x509 struct.
- NULL returned if the x509 object is NULL.

Example

```
WOLFSSL_X509* x509 = (WOLFSSL_X509)XMALOC(sizeof(WOLFSSL_X509), NULL,
DYNAMIC_TYPE_X509) ;
...
const WOLFSSL_ASN1_TIME* notAfter = wolfSSL_X509_get_notAfter(x509);
if(notAfter == NULL){
    // Failure case, the x509 object is null.
}
```

C.52.2.204 function wolfSSL_X509_version

```
int wolfSSL_X509_version(
    WOLFSSL_X509 *
)
```

This function retrieves the version of the X509 certificate.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_X509_get_subject_name
- wolfSSL_X509_get_issuer_name
- wolfSSL_X509_get_isCA
- wolfSSL_get_peer_certificate

Return:

- 0 returned if the x509 structure is NULL.
- version the version stored in the x509 structure will be returned.

Example

```
WOLFSSL_X509* x509;
int version;
...
version = wolfSSL_X509_version(x509);
if(!version){
    // The function returned 0, failure case.
}
```

C.52.2.205 function wolfSSL_X509_d2i_fp

```
WOLFSSL_X509 * wolfSSL_X509_d2i_fp(
    WOLFSSL_X509 ** x509,
    FILE * file
)
```

If NO_STDIO_FILESYSTEM is defined this function will allocate heap memory, initialize a WOLFSSL_X509 structure and return a pointer to it.

Parameters:

- **x509** a pointer to a WOLFSSL_X509 pointer.
- **file** a defined type that is a pointer to a FILE.

See:

- wolfSSL_X509_d2i
- XFTELL
- XREWIND
- XFSEEK

Return:

- *WOLFSSL_X509 WOLFSSL_X509 structure pointer is returned if the function executes successfully.
- NULL if the call to XFTELL macro returns a negative value.

Example

```
WOLFSSL_X509* x509a = (WOLFSSL_X509*)XMALLOC(sizeof(WOLFSSL_X509), NULL,
DYNAMIC_TYPE_X509);
WOLFSSL_X509** x509 = x509a;
XFILE file;  (mapped to struct fs_file*)
...
WOLFSSL_X509* newX509 = wolfSSL_X509_d2i_fp(x509, file);
if(newX509 == NULL){
```

```
    // The function returned NULL
}
```

C.52.2.206 function wolfSSL_X509_load_certificate_file

```
WOLFSSL_X509 * wolfSSL_X509_load_certificate_file(
    const char * fname,
    int format
)
```

The function loads the x509 certificate into memory.

Parameters:

- **fname** the certificate file to be loaded.
- **format** the format of the certificate.

See:

- InitDecodedCert
- PemToDer
- wolfSSL_get_certificate
- AssertNotNull

Return:

- pointer a successful execution returns pointer to a WOLFSSL_X509 structure.
- NULL returned if the certificate was not able to be written.

Example

```
#define cliCert    "certs/client-cert.pem"
…
X509* x509;
…
x509 = wolfSSL_X509_load_certificate_file(cliCert, SSL_FILETYPE_PEM);
AssertNotNull(x509);
```

C.52.2.207 function wolfSSL_X509_get_device_type

```
unsigned char * wolfSSL_X509_get_device_type(
    WOLFSSL_X509 * x509,
    unsigned char * in,
    int * inOutSz
)
```

This function copies the device type from the x509 structure to the buffer.

Parameters:

- **x509** pointer to a WOLFSSL_X509 structure, created with WOLFSSL_X509_new().
- **in** a pointer to a byte type that will hold the device type (the buffer).
- **inOutSz** the minimum of either the parameter inOutSz or the deviceTypeSz member of the x509 structure.

See:

- wolfSSL_X509_get_hw_type
- wolfSSL_X509_get_hw_serial_number
- wolfSSL_X509_d2i

Return:

- pointer returns a byte pointer holding the device type from the x509 structure.
- NULL returned if the buffer size is NULL.

Example

```
WOLFSSL_X509* x509 = (WOLFSSL_X509)XMALOC(sizeof(WOLFSSL_X509), NULL,
DYNAMIC_TYPE_X509);
byte* in;
int* inOutSz;
...
byte* deviceType = wolfSSL_X509_get_device_type(x509, in, inOutSz);

if(!deviceType){
    // Failure case, NULL was returned.
}
```

C.52.2.208 function wolfSSL_X509_get_hw_type

```
unsigned char * wolfSSL_X509_get_hw_type(
    WOLFSSL_X509 * x509,
    unsigned char * in,
    int * inOutSz
)
```

The function copies the hwType member of the WOLFSSL_X509 structure to the buffer.

Parameters:

- **x509** a pointer to a WOLFSSL_X509 structure containing certificate information.
- **in** pointer to type byte that represents the buffer.
- **inOutSz** pointer to type int that represents the size of the buffer.

See:

- wolfSSL_X509_get_hw_serial_number
- wolfSSL_X509_get_device_type

Return:

- byte The function returns a byte type of the data previously held in the hwType member of the WOLFSSL_X509 structure.
- NULL returned if inOutSz is NULL.

Example

```
WOLFSSL_X509* x509;  // X509 certificate
byte* in;  // initialize the buffer
int* inOutSz;  // holds the size of the buffer
...
byte* hwType = wolfSSL_X509_get_hw_type(x509, in, inOutSz);

if(hwType == NULL){
    // Failure case function returned NULL.
}
```

C.52.2.209 function wolfSSL_X509_get_hw_serial_number

```
unsigned char * wolfSSL_X509_get_hw_serial_number(
    WOLFSSL_X509 * x509,
    unsigned char * in,
```

```
    int * inOutSz
)
```

This function returns the hwSerialNum member of the x509 object.

Parameters:

- **x509** pointer to a WOLFSSL_X509 structure containing certificate information.
- **in** a pointer to the buffer that will be copied to.
- **inOutSz** a pointer to the size of the buffer.

See:

- wolfSSL_X509_get_subject_name
- wolfSSL_X509_get_issuer_name
- wolfSSL_X509_get_isCA
- wolfSSL_get_peer_certificate
- wolfSSL_X509_version

Return: pointer the function returns a byte pointer to the in buffer that will contain the serial number loaded from the x509 object.

Example

```
char* serial;
byte* in;
int* inOutSz;
WOLFSSL_X509 x509;
...
serial = wolfSSL_X509_get_hw_serial_number(x509, in, inOutSz);

if(serial == NULL || serial <= 0){
    // Failure case
}
```

C.52.2.210 function wolfSSL_connect_cert

```
int wolfSSL_connect_cert(
    WOLFSSL * ssl
)
```

This function is called on the client side and initiates an SSL/TLS handshake with a server only long enough to get the peer's certificate chain. When this function is called, the underlying communication channel has already been set up. wolfSSL_connect_cert() will only return once the peer's certificate chain has been received.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_get_error
- wolfSSL_connect
- wolfSSL_accept

Return:

- SSL_SUCCESS upon success.
- SSL_FAILURE will be returned if the SSL session parameter is NULL.
- SSL_FATAL_ERROR will be returned if an error occurred. To get a more detailed error code, call wolfSSL_get_error().

Example

```
int ret = 0;
int err = 0;
WOLFSSL* ssl;
char buffer[80];
...
ret = wolfSSL_connect_cert(ssl);
if (ret != SSL_SUCCESS) {
    err = wolfSSL_get_error(ssl, ret);
    printf("error = %d, %s\n", err, wolfSSL_ERR_error_string(err, buffer));
}
```

C.52.2.211 function wolfSSL_d2i_PKCS12_bio

```
WC_PKCS12 * wolfSSL_d2i_PKCS12_bio(
    WOLFSSL_BIO * bio,
    WC_PKCS12 ** pkcs12
)
```

wolfSSL_d2i_PKCS12_bio (d2i_PKCS12_bio) copies in the PKCS12 information from WOLFSSL_BIO to the structure WC_PKCS12. The information is divided up in the structure as a list of Content Infos along with a structure to hold optional MAC information. After the information has been divided into chunks (but not decrypted) in the structure WC_PKCS12, it can then be parsed and decrypted by calling.

Parameters:

- **bio** WOLFSSL_BIO structure to read PKCS12 buffer from.
- **pkcs12** WC_PKCS12 structure pointer for new PKCS12 structure created. Can be NULL

See:

- wolfSSL_PKCS12_parse
- wc_PKCS12_free

Return:

- WC_PKCS12 pointer to a WC_PKCS12 structure.
- Failure If function failed it will return NULL.

Example

```
WC_PKCS12* pkcs;
WOLFSSL_BIO* bio;
WOLFSSL_X509* cert;
WOLFSSL_EVP_PKEY* pkey;
STACK_OF(X509) certs;
//bio loads in PKCS12 file
wolfSSL_d2i_PKCS12_bio(bio, &pkcs);
wolfSSL_PKCS12_parse(pkcs, "a password", &pkey, &cert, &certs)
wc_PKCS12_free(pkcs)
//use cert, pkey, and optionally certs stack
```

C.52.2.212 function wolfSSL_i2d_PKCS12_bio

```
WC_PKCS12 * wolfSSL_i2d_PKCS12_bio(
    WOLFSSL_BIO * bio,
    WC_PKCS12 * pkcs12
)
```

wolfSSL_i2d_PKCS12_bio (i2d_PKCS12_bio) copies in the cert information from the structure WC_PKCS12 to WOLFSSL_BIO.

Parameters:

- **bio** WOLFSSL_BIO structure to write PKCS12 buffer to.
- **pkcs12** WC_PKCS12 structure for PKCS12 structure as input.

See:

- wolfSSL_PKCS12_parse
- wc_PKCS12_free

Return:

- 1 for success.
- Failure 0.

Example

```
WC_PKCS12 pkcs12;
FILE *f;
byte buffer[5300];
char file[] = "./test.p12";
int bytes;
WOLFSSL_BIO* bio;
pkcs12 = wc_PKCS12_new();
f = fopen(file, "rb");
bytes = (int)fread(buffer, 1, sizeof(buffer), f);
fclose(f);
//convert the DER file into an internal structure
wc_d2i_PKCS12(buffer, bytes, pkcs12);
bio = wolfSSL_BIO_new(wolfSSL_BIO_s_mem());
//convert PKCS12 structure into bio
wolfSSL_i2d_PKCS12_bio(bio, pkcs12);
wc_PKCS12_free(pkcs)
//use bio
```

C.52.2.213 function wolfSSL_PKCS12_parse

```
int wolfSSL_PKCS12_parse(
    WC_PKCS12 * pkcs12,
    const char * psw,
    WOLFSSL_EVP_PKEY ** pkey,
    WOLFSSL_X509 ** cert,
    WOLF_STACK_OF(WOLFSSL_X509) ** ca
)
```

PKCS12 can be enabled with adding –enable-opensslextra to the configure command. It can use triple DES and RC4 for decryption so would recommend also enabling these features when enabling opensslextra (–enable-des3 –enable-arc4). wolfSSL does not currently support RC2 so decryption with RC2 is currently not available. This may be noticeable with default encryption schemes used by OpenSSL command line to create .p12 files. wolfSSL_PKCS12_parse (PKCS12_parse). The first thing this function does is check the MAC is correct if present. If the MAC fails then the function returns and does not try to decrypt any of the stored Content Infos. This function then parses through each Content Info looking for a bag type, if the bag type is known it is decrypted as needed and either stored in the list of certificates being built or as a key found. After parsing through all bags the key found is then compared with the certificate list until a matching pair is found. This matching pair is then returned as the key and certificate, optionally the certificate list found is returned as a STACK_OF

certificates. At the moment a CRL, Secret or SafeContents bag will be skipped over and not parsed. It can be seen if these or other “Unknown” bags are skipped over by viewing the debug print out. Additional attributes such as friendly name are skipped over when parsing a PKCS12 file.

Parameters:

- **pkcs12** WC_PKCS12 structure to parse.
- **paswd** password for decrypting PKCS12.
- **pkey** structure to hold private key decoded from PKCS12.
- **cert** structure to hold certificate decoded from PKCS12.
- **stack** optional stack of extra certificates.

See:

- wolfSSL_d2i_PKCS12_bio
- wc_PKCS12_free

Return:

- SSL_SUCCESS On successfully parsing PKCS12.
- SSL_FAILURE If an error case was encountered.

Example

```
WC_PKCS12* pkcs;
WOLFSSL_BIO* bio;
WOLFSSL_X509* cert;
WOLFSSL_EVP_PKEY* pkey;
STACK_OF(X509) certs;
//bio loads in PKCS12 file
wolfSSL_d2i_PKCS12_bio(bio, &pkcs);
wolfSSL_PKCS12_parse(pkcs, “a password”, &pkey, &cert, &certs)
wc_PKCS12_free(pkcs)
//use cert, pkey, and optionally certs stack
```

C.52.2.214 function wolfSSL_SetTmpDH

```
int wolfSSL_SetTmpDH(
    WOLFSSL * ssl,
    const unsigned char * p,
    int pSz,
    const unsigned char * g,
    int gSz
)
```

Server Diffie-Hellman Ephemeral parameters setting. This function sets up the group parameters to be used if the server negotiates a cipher suite that uses DHE.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **p** Diffie-Hellman prime number parameter.
- **pSz** size of p.
- **g** Diffie-Hellman “generator” parameter.
- **gSz** size of g.

See: SSL_accept

Return:

- SSL_SUCCESS upon success.

- MEMORY_ERROR will be returned if a memory error was encountered.
- SIDE_ERROR will be returned if this function is called on an SSL client instead of an SSL server.

Example

```
WOLFSSL* ssl;
static unsigned char p[] = {...};
static unsigned char g[] = {...};
...
wolfSSL_SetTmpDH(ssl, p, sizeof(p), g, sizeof(g));
```

C.52.2.215 function wolfSSL_SetTmpDH_buffer

```
int wolfSSL_SetTmpDH_buffer(
    WOLFSSL * ssl,
    const unsigned char * b,
    long sz,
    int format
)
```

The function calls the wolfSSL_SetTMpDH_buffer_wrapper, which is a wrapper for Diffie-Hellman parameters.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **buf** allocated buffer passed in from wolfSSL_SetTMpDH_file_wrapper.
- **sz** a long int that holds the size of the file (fname within wolfSSL_SetTmpDH_file_wrapper).
- **format** an integer type passed through from wolfSSL_SetTmpDH_file_wrapper() that is a representation of the certificate format.

See:

- wolfSSL_SetTmpDH_buffer_wrapper
- wc_DhParamsLoad
- wolfSSL_SetTmpDH
- PemToDer
- wolfSSL_CTX_SetTmpDH
- wolfSSL_CTX_SetTmpDH_file

Return:

- SSL_SUCCESS on successful execution.
- SSL_BAD_FILETYPE if the file type is not PEM and is not ASN.1. It will also be returned if the wc_DhParamsLoad does not return normally.
- SSL_NO_PEM_HEADER returns from PemToDer if there is not a PEM header.
- SSL_BAD_FILE returned if there is a file error in PemToDer.
- SSL_FATAL_ERROR returned from PemToDer if there was a copy error.
- MEMORY_E - if there was a memory allocation error.
- BAD_FUNC_ARG returned if the WOLFSSL struct is NULL or if there was otherwise a NULL argument passed to a subroutine.
- DH_KEY_SIZE_E is returned if their is a key size error in wolfSSL_SetTmpDH().
- SIDE_ERROR returned if it is not the server side in wolfSSL_SetTmpDH.

Example

```
Static int wolfSSL_SetTmpDH_file_wrapper(WOLFSSL_CTX* ctx, WOLFSSL* ssl,
Const char* fname, int format);
long sz = 0;
byte* myBuffer = staticBuffer[FILE_BUFFER_SIZE];
```

```
…
if(ssl)
ret = wolfSSL_SetTmpDH_buffer(ssl, myBuffer, sz, format);
```

C.52.2.216 function wolfSSL_SetTmpDH_file

```
int wolfSSL_SetTmpDH_file(
    WOLFSSL * ssl,
    const char * f,
    int format
)
```

This function calls wolfSSL_SetTmpDH_file_wrapper to set server Diffie-Hellman parameters.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **fname** a constant char pointer holding the certificate.
- **format** an integer type that holds the format of the certification.

See:

- wolfSSL_CTX_SetTmpDH_file
- wolfSSL_SetTmpDH_file_wrapper
- wolfSSL_SetTmpDH_buffer
- wolfSSL_CTX_SetTmpDH_buffer
- wolfSSL_SetTmpDH_buffer_wrapper
- wolfSSL_SetTmpDH
- wolfSSL_CTX_SetTmpDH

Return:

- SSL_SUCCESS returned on successful completion of this function and its subroutines.
- MEMORY_E returned if a memory allocation failed in this function or a subroutine.
- SIDE_ERROR if the side member of the Options structure found in the WOLFSSL struct is not the server side.
- SSL_BAD_FILETYPE returns if the certificate fails a set of checks.
- DH_KEY_SIZE_E returned if the DH parameter's key size is less than the value of the minDhKeySz member in the WOLFSSL struct.
- DH_KEY_SIZE_E returned if the DH parameter's key size is greater than the value of the maxDhKeySz member in the WOLFSSL struct.
- BAD_FUNC_ARG returns if an argument value is NULL that is not permitted such as, the WOLFSSL structure.

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
const char* dhParam;
…
AssertIntNE(SSL_SUCCESS,
wolfSSL_SetTmpDH_file(ssl, dhParam, SSL_FILETYPE_PEM));
```

C.52.2.217 function wolfSSL_CTX_SetTmpDH

```
int wolfSSL_CTX_SetTmpDH(
    WOLFSSL_CTX * ctx,
    const unsigned char * p,
    int pSz,
    const unsigned char * g,
```

```
    int gSz
)
```

Sets the parameters for the server CTX Diffie-Hellman.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().
- **p** a constant unsigned char pointer loaded into the buffer member of the serverDH_P struct.
- **pSz** an int type representing the size of p, initialized to MAX_DH_SIZE.
- **g** a constant unsigned char pointer loaded into the buffer member of the serverDH_G struct.
- **gSz** an int type representing the size of g, initialized to MAX_DH_SIZE.

See:

- wolfSSL_SetTmpDH
- wc_DhParamsLoad

Return:

- SSL_SUCCESS returned if the function and all subroutines return without error.
- BAD_FUNC_ARG returned if the CTX, p or g parameters are NULL.
- DH_KEY_SIZE_E returned if the DH parameter's key size is less than the value of the minDhKeySz member of the WOLFSSL_CTX struct.
- DH_KEY_SIZE_E returned if the DH parameter's key size is greater than the value of the maxDhKeySz member of the WOLFSSL_CTX struct.
- MEMORY_E returned if the allocation of memory failed in this function or a subroutine.

Exmaple

```
WOLFSSL_CTX* ctx =  WOLFSSL_CTX_new( protocol );
byte* p;
byte* g;
word32 pSz = (word32)sizeof(p)/sizeof(byte);
word32 gSz = (word32)sizeof(g)/sizeof(byte);
…
int ret =  wolfSSL_CTX_SetTmpDH(ctx, p, pSz, g, gSz);

if(ret != SSL_SUCCESS){
    // Failure case
}
```

C.52.2.218 function wolfSSL_CTX_SetTmpDH_buffer

```
int wolfSSL_CTX_SetTmpDH_buffer(
    WOLFSSL_CTX * ctx,
    const unsigned char * b,
    long sz,
    int format
)
```

A wrapper function that calls wolfSSL_SetTmpDH_buffer_wrapper.

Parameters:

- **ctx** a pointer to a WOLFSSL structure, created using wolfSSL_CTX_new().
- **buf** a pointer to a constant unsigned char type that is allocated as the buffer and passed through to wolfSSL_SetTmpDH_buffer_wrapper.
- **sz** a long integer type that is derived from the fname parameter in wolfSSL_SetTmpDH_file_wrapper().
- **format** an integer type passed through from wolfSSL_SetTmpDH_file_wrapper().

See:

- wolfSSL_SetTmpDH_buffer_wrapper
- wolfSSL_SetTMpDH_buffer
- wolfSSL_SetTmpDH_file_wrapper
- wolfSSL_CTX_SetTmpDH_file

Return:

- 0 returned for a successful execution.
- BAD_FUNC_ARG returned if the ctx or buf parameters are NULL.
- MEMORY_E if there is a memory allocation error.
- SSL_BAD_FILETYPE returned if format is not correct.

Example

```
static int wolfSSL_SetTmpDH_file_wrapper(WOLFSSL_CTX* ctx, WOLFSSL* ssl,
Const char* fname, int format);
#ifdef WOLFSSL_SMALL_STACK
byte staticBuffer[1]; // force heap usage
#else
byte* staticBuffer;
long sz = 0;
…
if(ssl){
    ret = wolfSSL_SetTmpDH_buffer(ssl, myBuffer, sz, format);
} else {
ret = wolfSSL_CTX_SetTmpDH_buffer(ctx, myBuffer, sz, format);
}
```

C.52.2.219 function wolfSSL_CTX_SetTmpDH_file

```
int wolfSSL_CTX_SetTmpDH_file(
    WOLFSSL_CTX * ctx,
    const char * f,
    int format
)
```

The function calls wolfSSL_SetTmpDH_file_wrapper to set the server Diffie-Hellman parameters.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().
- **fname** a constant character pointer to a certificate file.
- **format** an integer type passed through from wolfSSL_SetTmpDH_file_wrapper() that is a representation of the certificate format.

See:

- wolfSSL_SetTmpDH_buffer_wrapper
- wolfSSL_SetTmpDH
- wolfSSL_CTX_SetTmpDH
- wolfSSL_SetTmpDH_buffer
- wolfSSL_CTX_SetTmpDH_buffer
- wolfSSL_SetTmpDH_file_wrapper
- AllocDer
- PemToDer

Return:

- SSL_SUCCESS returned if the wolfSSL_SetTmpDH_file_wrapper or any of its subroutines return successfully.
- MEMORY_E returned if an allocation of dynamic memory fails in a subroutine.
- BAD_FUNC_ARG returned if the ctx or fname parameters are NULL or if a subroutine is passed a NULL argument.
- SSL_BAD_FILE returned if the certificate file is unable to open or if the a set of checks on the file fail from wolfSSL_SetTmpDH_file_wrapper.
- SSL_BAD_FILETYPE returned if the format is not PEM or ASN.1 from wolfSSL_SetTmpDH_buffer_wrapper().
- DH_KEY_SIZE_E returned if the DH parameter's key size is less than the value of the minDhKeySz member of the WOLFSSL_CTX struct.
- DH_KEY_SIZE_E returned if the DH parameter's key size is greater than the value of the maxDhKeySz member of the WOLFSSL_CTX struct.
- SIDE_ERROR returned in wolfSSL_SetTmpDH() if the side is not the server end.
- SSL_NO_PEM_HEADER returned from PemToDer if there is no PEM header.
- SSL_FATAL_ERROR returned from PemToDer if there is a memory copy failure.

Example

```
#define dhParam     "certs/dh2048.pem"
#DEFINE aSSERTiNTne(x, y)     AssertInt(x, y, !=, ==)
WOLFSSL_CTX* ctx;
…
AssertNotNull(ctx = wolfSSL_CTX_new(wolfSSLv23_client_method()))
…
AssertIntNE(SSL_SUCCESS, wolfSSL_CTX_SetTmpDH_file(NULL, dhParam,
SSL_FILETYPE_PEM));
```

C.52.2.220 function wolfSSL_CTX_SetMinDhKey_Sz

```
int wolfSSL_CTX_SetMinDhKey_Sz(
    WOLFSSL_CTX * ctx,
    word16
)
```

This function sets the minimum size (in bits) of the Diffie Hellman key size by accessing the minDhKeySz member in the WOLFSSL_CTX structure.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **keySz_bits** a word16 type used to set the minimum DH key size in bits. The WOLFSSL_CTX struct holds this information in the minDhKeySz member.

See:

- wolfSSL_SetMinDhKey_Sz
- wolfSSL_CTX_SetMaxDhKey_Sz
- wolfSSL_SetMaxDhKey_Sz
- wolfSSL_GetDhKey_Sz
- wolfSSL_CTX_SetTMpDH_file

Return:

- SSL_SUCCESS returned if the function completes successfully.
- BAD_FUNC_ARG returned if the WOLFSSL_CTX struct is NULL or if the keySz_bits is greater than 16,000 or not divisible by 8.

Example

```
public static int CTX_SetMinDhKey_Sz(IntPtr ctx, short minDhKey){
…
return wolfSSL_CTX_SetMinDhKey_Sz(local_ctx, minDhKeyBits);
```

C.52.2.221 function wolfSSL_SetMinDhKey_Sz

```
int wolfSSL_SetMinDhKey_Sz(
    WOLFSSL * ssl,
    word16 keySz_bits
)
```

Sets the minimum size (in bits) for a Diffie-Hellman key in the WOLFSSL structure.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **keySz_bits** a word16 type used to set the minimum DH key size in bits. The WOLFSSL_CTX struct holds this information in the minDhKeySz member.

See:

- wolfSSL_CTX_SetMinDhKey_Sz
- wolfSSL_GetDhKey_Sz

Return:

- SSL_SUCCESS the minimum size was successfully set.
- BAD_FUNC_ARG the WOLFSSL structure was NULL or if the keySz_bits is greater than 16,000 or not divisible by 8.

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
word16 keySz_bits;
...
if(wolfSSL_SetMinDhKey_Sz(ssl, keySz_bits) != SSL_SUCCESS){
    // Failed to set.
}
```

C.52.2.222 function wolfSSL_CTX_SetMaxDhKey_Sz

```
int wolfSSL_CTX_SetMaxDhKey_Sz(
    WOLFSSL_CTX * ctx,
    word16 keySz_bits
)
```

This function sets the maximum size (in bits) of the Diffie Hellman key size by accessing the maxDhKeySz member in the WOLFSSL_CTX structure.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **keySz_bits** a word16 type used to set the maximum DH key size in bits. The WOLFSSL_CTX struct holds this information in the maxDhKeySz member.

See:

- wolfSSL_SetMinDhKey_Sz
- wolfSSL_CTX_SetMinDhKey_Sz
- wolfSSL_SetMaxDhKey_Sz
- wolfSSL_GetDhKey_Sz

- wolfSSL_CTX_SetTMpDH_file

Return:

- SSL_SUCCESS returned if the function completes successfully.
- BAD_FUNC_ARG returned if the WOLFSSL_CTX struct is NULL or if the keySz_bits is greater than 16,000 or not divisible by 8.

Example

```
public static int CTX_SetMaxDhKey_Sz(IntPtr ctx, short maxDhKey){
…
return wolfSSL_CTX_SetMaxDhKey_Sz(local_ctx, keySz_bits);
```

C.52.2.223 function wolfSSL_SetMaxDhKey_Sz

```
int wolfSSL_SetMaxDhKey_Sz(
    WOLFSSL * ssl,
    word16 keySz_bits
)
```

Sets the maximum size (in bits) for a Diffie-Hellman key in the WOLFSSL structure.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **keySz** a word16 type representing the bit size of the maximum DH key.

See:

- wolfSSL_CTX_SetMaxDhKey_Sz
- wolfSSL_GetDhKey_Sz

Return:

- SSL_SUCCESS the maximum size was successfully set.
- BAD_FUNC_ARG the WOLFSSL structure was NULL or the keySz parameter was greater than the allowable size or not divisible by 8.

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
word16 keySz;
...
if(wolfSSL_SetMaxDhKey(ssl, keySz) != SSL_SUCCESS){
    // Failed to set.
}
```

C.52.2.224 function wolfSSL_GetDhKey_Sz

```
int wolfSSL_GetDhKey_Sz(
    WOLFSSL *
)
```

Returns the value of dhKeySz (in bits) that is a member of the options structure. This value represents the Diffie-Hellman key size in bytes.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_SetMinDhKey_sz
- wolfSSL_CTX_SetMinDhKey_Sz
- wolfSSL_CTX_SetTmpDH
- wolfSSL_SetTmpDH
- wolfSSL_CTX_SetTmpDH_file

Return:

- dhKeySz returns the value held in ssl->options.dhKeySz which is an integer value representing a size in bits.
- BAD_FUNC_ARG returns if the WOLFSSL struct is NULL.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
WOLFSSL* ssl = wolfSSL_new(ctx);
int dhKeySz;
...
dhKeySz = wolfSSL_GetDhKey_Sz(ssl);

if(dhKeySz == BAD_FUNC_ARG || dhKeySz <= 0){
    // Failure case
} else {
    // dhKeySz holds the size of the key.
}
```

C.52.2.225 function wolfSSL_CTX_SetMinRsaKey_Sz

```
int wolfSSL_CTX_SetMinRsaKey_Sz(
    WOLFSSL_CTX * ctx,
    short keySz
)
```

Sets the minimum RSA key size in both the WOLFSSL_CTX structure and the WOLFSSL_CERT_MANAGER structure.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().
- **keySz** a short integer type stored in minRsaKeySz in the ctx structure and the cm structure converted to bytes.

See: wolfSSL_SetMinRsaKey_Sz

Return:

- SSL_SUCCESS returned on successful execution of the function.
- BAD_FUNC_ARG returned if the ctx structure is NULL or the keySz is less than zero or not divisible by 8.

Example

```
WOLFSSL_CTX* ctx = SSL_CTX_new(method);
(void)minDhKeyBits;
ourCert = myoptarg;
…
minDhKeyBits = atoi(myoptarg);
…
if(wolfSSL_CTX_SetMinRsaKey_Sz(ctx, minRsaKeyBits) != SSL_SUCCESS){
…
```

C.52.2.226 function wolfSSL_SetMinRsaKey_Sz

```
int wolfSSL_SetMinRsaKey_Sz(
    WOLFSSL * ssl,
    short keySz
)
```

Sets the minimum allowable key size in bits for RSA located in the WOLFSSL structure.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **keySz** a short integer value representing the the minimum key in bits.

See: wolfSSL_CTX_SetMinRsaKey_Sz

Return:

- SSL_SUCCESS the minimum was set successfully.
- BAD_FUNC_ARG returned if the ssl structure is NULL or if the ksySz is less than zero or not divisible by 8.

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
short keySz;
…

int isSet =  wolfSSL_SetMinRsaKey_Sz(ssl, keySz);
if(isSet != SSL_SUCCESS){
    Failed to set.
}
```

C.52.2.227 function wolfSSL_CTX_SetMinEccKey_Sz

```
int wolfSSL_CTX_SetMinEccKey_Sz(
    WOLFSSL_CTX * ssl,
    short keySz
)
```

Sets the minimum size in bits for the ECC key in the WOLF_CTX structure and the WOLFSSL_CERT_MANAGER structure.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().
- **keySz** a short integer type that represents the minimum ECC key size in bits.

See: wolfSSL_SetMinEccKey_Sz

Return:

- SSL_SUCCESS returned for a successful execution and the minEccKeySz member is set.
- BAD_FUNC_ARG returned if the WOLFSSL_CTX struct is NULL or if the keySz is negative or not divisible by 8.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
short keySz; // minimum key size
…
if(wolfSSL_CTX_SetMinEccKey(ctx, keySz) != SSL_SUCCESS){
```

```
    // Failed to set min key size
}
```

C.52.2.228 function wolfSSL_SetMinEccKey_Sz

```
int wolfSSL_SetMinEccKey_Sz(
    WOLFSSL * ssl,
    short keySz
)
```

Sets the value of the minEccKeySz member of the options structure. The options struct is a member of the WOLFSSL structure and is accessed through the ssl parameter.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **keySz** value used to set the minimum ECC key size. Sets value in the options structure.

See:

- wolfSSL_CTX_SetMinEccKey_Sz
- wolfSSL_CTX_SetMinRsaKey_Sz
- wolfSSL_SetMinRsaKey_Sz

Return:

- SSL_SUCCESS if the function successfully set the minEccKeySz member of the options structure.
- BAD_FUNC_ARG if the WOLFSSL_CTX structure is NULL or if the key size (keySz) is less than 0 (zero) or not divisible by 8.

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx); // New session
short keySz = 999; // should be set to min key size allowable
...
if(wolfSSL_SetMinEccKey_Sz(ssl, keySz) != SSL_SUCCESS){
    // Failure case.
}
```

C.52.2.229 function wolfSSL_make_eap_keys

```
int wolfSSL_make_eap_keys(
    WOLFSSL * ssl,
    void * key,
    unsigned int len,
    const char * label
)
```

This function is used by EAP_TLS and EAP-TTLS to derive keying material from the master secret.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **msk** a void pointer variable that will hold the result of the p_hash function.
- **len** an unsigned integer that represents the length of the msk variable.
- **label** a constant char pointer that is copied from in wc_PRF().

See:

- wc_PRF
- wc_HmacFinal

- wc_HmacUpdate

Return:

- BUFFER_E returned if the actual size of the buffer exceeds the maximum size allowable.
- MEMORY_E returned if there is an error with memory allocation.

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);;
void* msk;
unsigned int len;
const char* label;
…
return wolfSSL_make_eap_keys(ssl, msk, len, label);
```

C.52.2.230 function wolfSSL_writev

```
int wolfSSL_writev(
    WOLFSSL * ssl,
    const struct iovec * iov,
    int iovcnt
)
```

Simulates writev semantics but doesn't actually do block at a time because of SSL_write() behavior and because front adds may be small. Makes porting into software that uses writev easier.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().
- **iov** array of I/O vectors to write
- **iovcnt** number of vectors in iov array.

See: wolfSSL_write

Return:

- 0 the number of bytes written upon success.
- 0 will be returned upon failure. Call wolfSSL_get_error() for the specific error code.
- MEMORY_ERROR will be returned if a memory error was encountered.
- SSL_FATAL_ERROR will be returned upon failure when either an error occurred or, when using non-blocking sockets, the SSL_ERROR_WANT_READ or SSL_ERROR_WANT_WRITE error was received and and the application needs to call wolfSSL_write() to get a specific error code.

Example

```
WOLFSSL* ssl = 0;
char *bufA = “hello\n”;
char *bufB = “hello world\n”;
int iovcnt;
struct iovec iov[2];

iov[0].iov_base = buffA;
iov[0].iov_len = strlen(buffA);
iov[1].iov_base = buffB;
iov[1].iov_len = strlen(buffB);
iovcnt = 2;
...
```

```
ret = wolfSSL_writev(ssl, iov, iovcnt);
// wrote "ret" bytes, or error if <= 0.
```

C.52.2.231 function wolfSSL_CTX_UnloadCAs

```
int wolfSSL_CTX_UnloadCAs(
    WOLFSSL_CTX *
)
```

This function unloads the CA signer list and frees the whole signer table.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().

See:

- wolfSSL_CertManagerUnloadCAs
- LockMutex
- UnlockMutex

Return:

- SSL_SUCCESS returned on successful execution of the function.
- BAD_FUNC_ARG returned if the WOLFSSL_CTX struct is NULL or there are otherwise unpermitted argument values passed in a subroutine.
- BAD_MUTEX_E returned if there was a mutex error. The LockMutex() did not return 0.

Example

```
WOLFSSL_METHOD method = wolfTLSv1_2_client_method();
WOLFSSL_CTX* ctx = WOLFSSL_CTX_new(method);
…
if(wolfSSL_CTX_UnloadCAs(ctx) != SSL_SUCCESS){
    // The function did not unload CAs
}
```

C.52.2.232 function wolfSSL_CTX_UnloadIntermediateCerts

```
int wolfSSL_CTX_UnloadIntermediateCerts(
    WOLFSSL_CTX * ctx
)
```

This function unloads intermediate certificates added to the CA signer list and frees them.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().

See:

- wolfSSL_CTX_UnloadCAs
- wolfSSL_CertManagerUnloadIntermediateCerts

Return:

- SSL_SUCCESS returned on successful execution of the function.
- BAD_FUNC_ARG returned if the WOLFSSL_CTX struct is NULL or there are otherwise unpermitted argument values passed in a subroutine.
- BAD_STATE_E returned if the WOLFSSL_CTX has a reference count > 1.
- BAD_MUTEX_E returned if there was a mutex error. The LockMutex() did not return 0.

Example

```
WOLFSSL_METHOD method = wolfTLSv1_2_client_method();
WOLFSSL_CTX* ctx = WOLFSSL_CTX_new(method);
…
if(wolfSSL_CTX_UnloadIntermediateCerts(ctx) != NULL){
    // The function did not unload CAs
}
```

C.52.2.233 function wolfSSL_CTX_Unload_trust_peers

```
int wolfSSL_CTX_Unload_trust_peers(
    WOLFSSL_CTX *
)
```

This function is used to unload all previously loaded trusted peer certificates. Feature is enabled by defining the macro WOLFSSL_TRUST_PEER_CERT.

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().

See:

- wolfSSL_CTX_trust_peer_buffer
- wolfSSL_CTX_trust_peer_cert

Return:

- SSL_SUCCESS upon success.
- BAD_FUNC_ARG will be returned if ctx is NULL.
- SSL_BAD_FILE will be returned if the file doesn't exist, can't be read, or is corrupted.
- MEMORY_E will be returned if an out of memory condition occurs.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx;
...
ret = wolfSSL_CTX_Unload_trust_peers(ctx);
if (ret != SSL_SUCCESS) {
    // error unloading trusted peer certs
}
...
```

C.52.2.234 function wolfSSL_CTX_trust_peer_buffer

```
int wolfSSL_CTX_trust_peer_buffer(
    WOLFSSL_CTX * ctx,
    const unsigned char * in,
    long sz,
    int format
)
```

This function loads a certificate to use for verifying a peer when performing a TLS/SSL handshake. The peer certificate sent during the handshake is compared by using the SKID when available and the signature. If these two things do not match then any loaded CAs are used. Is the same functionality as wolfSSL_CTX_trust_peer_cert except is from a buffer instead of a file. Feature is enabled by defining the macro WOLFSSL_TRUST_PEER_CERT Please see the examples for proper usage.

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().
- **buffer** pointer to the buffer containing certificates.
- **sz** length of the buffer input.
- **type** type of certificate being loaded i.e. SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM.

See:

- wolfSSL_CTX_load_verify_buffer
- wolfSSL_CTX_use_certificate_file
- wolfSSL_CTX_use_PrivateKey_file
- wolfSSL_CTX_use_certificate_chain_file
- wolfSSL_CTX_trust_peer_cert
- wolfSSL_CTX_Unload_trust_peers
- wolfSSL_use_certificate_file
- wolfSSL_use_PrivateKey_file
- wolfSSL_use_certificate_chain_file

Return:

- SSL_SUCCESS upon success
- SSL_FAILURE will be returned if ctx is NULL, or if both file and type are invalid.
- SSL_BAD_FILETYPE will be returned if the file is the wrong format.
- SSL_BAD_FILE will be returned if the file doesn't exist, can't be read, or is corrupted.
- MEMORY_E will be returned if an out of memory condition occurs.
- ASN_INPUT_E will be returned if Base16 decoding fails on the file.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx;
...

ret = wolfSSL_CTX_trust_peer_buffer(ctx, bufferPtr, bufferSz,
SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
// error loading trusted peer cert
}
...
```

C.52.2.235 function wolfSSL_CTX_load_verify_buffer

```
int wolfSSL_CTX_load_verify_buffer(
    WOLFSSL_CTX * ctx,
    const unsigned char * in,
    long sz,
    int format
)
```

This function loads a CA certificate buffer into the WOLFSSL Context. It behaves like the non-buffered version, only differing in its ability to be called with a buffer as input instead of a file. The buffer is provided by the in argument of size sz. format specifies the format type of the buffer; SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM. More than one CA certificate may be loaded per buffer as long as the format is in PEM. Please see the examples for proper usage.

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().
- **in** pointer to the CA certificate buffer.
- **sz** size of the input CA certificate buffer, in.

- **format** format of the buffer certificate, either SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM.

See:

- wolfSSL_CTX_load_verify_locations
- wolfSSL_CTX_use_certificate_buffer
- wolfSSL_CTX_use_PrivateKey_buffer
- wolfSSL_CTX_use_certificate_chain_buffer
- wolfSSL_use_certificate_buffer
- wolfSSL_use_PrivateKey_buffer
- wolfSSL_use_certificate_chain_buffer

Return:

- SSL_SUCCESS upon success
- SSL_BAD_FILETYPE will be returned if the file is the wrong format.
- SSL_BAD_FILE will be returned if the file doesn't exist, can't be read, or is corrupted.
- MEMORY_E will be returned if an out of memory condition occurs.
- ASN_INPUT_E will be returned if Base16 decoding fails on the file.
- BUFFER_E will be returned if a chain buffer is bigger than the receiving buffer.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx;
byte certBuff[...];
long sz = sizeof(certBuff);
...

ret = wolfSSL_CTX_load_verify_buffer(ctx, certBuff, sz, SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
    // error loading CA certs from buffer
}
...
```

C.52.2.236 function wolfSSL_CTX_load_verify_buffer_ex

```
int wolfSSL_CTX_load_verify_buffer_ex(
    WOLFSSL_CTX * ctx,
    const unsigned char * in,
    long sz,
    int format,
    int userChain,
    word32 flags
)
```

This function loads a CA certificate buffer into the WOLFSSL Context. It behaves like the non-buffered version, only differing in its ability to be called with a buffer as input instead of a file. The buffer is provided by the in argument of size sz. format specifies the format type of the buffer; SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM. More than one CA certificate may be loaded per buffer as long as the format is in PEM. The _ex version was added in PR 2413 and supports additional arguments for userChain and flags.

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().
- **in** pointer to the CA certificate buffer.
- **sz** size of the input CA certificate buffer, in.
- **format** format of the buffer certificate, either SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM.

- **userChain** If using format WOLFSSL_FILETYPE_ASN1 this set to non-zero indicates a chain of DER's is being presented.
- **flags** See ssl.h around WOLFSSL_LOAD_VERIFY_DEFAULT_FLAGS.

See:

- wolfSSL_CTX_load_verify_buffer
- wolfSSL_CTX_load_verify_locations
- wolfSSL_CTX_use_certificate_buffer
- wolfSSL_CTX_use_PrivateKey_buffer
- wolfSSL_CTX_use_certificate_chain_buffer
- wolfSSL_use_certificate_buffer
- wolfSSL_use_PrivateKey_buffer
- wolfSSL_use_certificate_chain_buffer

Return:

- SSL_SUCCESS upon success
- SSL_BAD_FILETYPE will be returned if the file is the wrong format.
- SSL_BAD_FILE will be returned if the file doesn't exist, can't be read, or is corrupted.
- MEMORY_E will be returned if an out of memory condition occurs.
- ASN_INPUT_E will be returned if Base16 decoding fails on the file.
- BUFFER_E will be returned if a chain buffer is bigger than the receiving buffer.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx;
byte certBuff[...];
long sz = sizeof(certBuff);
...

// Example for force loading an expired certificate
ret = wolfSSL_CTX_load_verify_buffer_ex(ctx, certBuff, sz, SSL_FILETYPE_PEM,
    0, (WOLFSSL_LOAD_FLAG_DATE_ERR_OKAY));
if (ret != SSL_SUCCESS) {
    // error loading CA certs from buffer
}
...
```

C.52.2.237 function wolfSSL_CTX_load_verify_chain_buffer_format

```
int wolfSSL_CTX_load_verify_chain_buffer_format(
    WOLFSSL_CTX * ctx,
    const unsigned char * in,
    long sz,
    int format
)
```

This function loads a CA certificate chain buffer into the WOLFSSL Context. It behaves like the non-buffered version, only differing in its ability to be called with a buffer as input instead of a file. The buffer is provided by the in argument of size sz. format specifies the format type of the buffer; SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM. More than one CA certificate may be loaded per buffer as long as the format is in PEM. Please see the examples for proper usage.

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().
- **in** pointer to the CA certificate buffer.

- **sz** size of the input CA certificate buffer, in.
- **format** format of the buffer certificate, either SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM.

See:

- wolfSSL_CTX_load_verify_locations
- wolfSSL_CTX_use_certificate_buffer
- wolfSSL_CTX_use_PrivateKey_buffer
- wolfSSL_CTX_use_certificate_chain_buffer
- wolfSSL_use_certificate_buffer
- wolfSSL_use_PrivateKey_buffer
- wolfSSL_use_certificate_chain_buffer

Return:

- SSL_SUCCESS upon success
- SSL_BAD_FILETYPE will be returned if the file is the wrong format.
- SSL_BAD_FILE will be returned if the file doesn't exist, can't be read, or is corrupted.
- MEMORY_E will be returned if an out of memory condition occurs.
- ASN_INPUT_E will be returned if Base16 decoding fails on the file.
- BUFFER_E will be returned if a chain buffer is bigger than the receiving buffer.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx;
byte certBuff[...];
long sz = sizeof(certBuff);
...

ret = wolfSSL_CTX_load_verify_chain_buffer_format(ctx,
                     certBuff, sz, WOLFSSL_FILETYPE_ASN1);
if (ret != SSL_SUCCESS) {
    // error loading CA certs from buffer
}
...
```

C.52.2.238 function wolfSSL_CTX_use_certificate_buffer

```
int wolfSSL_CTX_use_certificate_buffer(
    WOLFSSL_CTX * ctx,
    const unsigned char * in,
    long sz,
    int format
)
```

This function loads a certificate buffer into the WOLFSSL Context. It behaves like the non-buffered version, only differing in its ability to be called with a buffer as input instead of a file. The buffer is provided by the in argument of size sz. format specifies the format type of the buffer; SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM. Please see the examples for proper usage.

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().
- **in** the input buffer containing the certificate to be loaded.
- **sz** the size of the input buffer.
- **format** the format of the certificate located in the input buffer (in). Possible values are SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM.

See:

- wolfSSL_CTX_load_verify_buffer
- wolfSSL_CTX_use_PrivateKey_buffer
- wolfSSL_CTX_use_certificate_chain_buffer
- wolfSSL_use_certificate_buffer
- wolfSSL_use_PrivateKey_buffer
- wolfSSL_use_certificate_chain_buffer

Return:

- SSL_SUCCESS upon success
- SSL_BAD_FILETYPE will be returned if the file is the wrong format.
- SSL_BAD_FILE will be returned if the file doesn't exist, can't be read, or is corrupted.
- MEMORY_E will be returned if an out of memory condition occurs.
- ASN_INPUT_E will be returned if Base16 decoding fails on the file.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx;
byte certBuff[...];
long sz = sizeof(certBuff);
...
ret = wolfSSL_CTX_use_certificate_buffer(ctx, certBuff, sz, SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
    // error loading certificate from buffer
}
...
```

C.52.2.239 function wolfSSL_CTX_use_PrivateKey_buffer

```
int wolfSSL_CTX_use_PrivateKey_buffer(
    WOLFSSL_CTX * ctx,
    const unsigned char * in,
    long sz,
    int format
)
```

This function loads a private key buffer into the SSL Context. It behaves like the non-buffered version, only differing in its ability to be called with a buffer as input instead of a file. The buffer is provided by the in argument of size sz. format specifies the format type of the buffer; SSL_FILETYPE_ASN1or SSL_FILETYPE_PEM. Please see the examples for proper usage.

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().
- **in** the input buffer containing the private key to be loaded.
- **sz** the size of the input buffer.
- **format** the format of the private key located in the input buffer (in). Possible values are SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM.

See:

- wolfSSL_CTX_load_verify_buffer
- wolfSSL_CTX_use_certificate_buffer
- wolfSSL_CTX_use_certificate_chain_buffer
- wolfSSL_use_certificate_buffer
- wolfSSL_use_PrivateKey_buffer

- wolfSSL_use_certificate_chain_buffer

Return:

- SSL_SUCCESS upon success
- SSL_BAD_FILETYPE will be returned if the file is the wrong format.
- SSL_BAD_FILE will be returned if the file doesn't exist, can't be read, or is corrupted.
- MEMORY_E will be returned if an out of memory condition occurs.
- ASN_INPUT_E will be returned if Base16 decoding fails on the file.
- NO_PASSWORD will be returned if the key file is encrypted but no password is provided.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx;
byte keyBuff[...];
long sz = sizeof(certBuff);
...
ret = wolfSSL_CTX_use_PrivateKey_buffer(ctx, keyBuff, sz, SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
    // error loading private key from buffer
}
...
```

C.52.2.240 function wolfSSL_CTX_use_certificate_chain_buffer

```
int wolfSSL_CTX_use_certificate_chain_buffer(
    WOLFSSL_CTX * ctx,
    const unsigned char * in,
    long sz
)
```

This function loads a certificate chain buffer into the WOLFSSL Context. It behaves like the non-buffered version, only differing in its ability to be called with a buffer as input instead of a file. The buffer is provided by the in argument of size sz. The buffer must be in PEM format and start with the subject's certificate, ending with the root certificate. Please see the examples for proper usage.

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().
- **in** the input buffer containing the PEM-formatted certificate chain to be loaded.
- **sz** the size of the input buffer.

See:

- wolfSSL_CTX_load_verify_buffer
- wolfSSL_CTX_use_certificate_buffer
- wolfSSL_CTX_use_PrivateKey_buffer
- wolfSSL_use_certificate_buffer
- wolfSSL_use_PrivateKey_buffer
- wolfSSL_use_certificate_chain_buffer

Return:

- SSL_SUCCESS upon success
- SSL_BAD_FILETYPE will be returned if the file is the wrong format.
- SSL_BAD_FILE will be returned if the file doesn't exist, can't be read, or is corrupted.
- MEMORY_E will be returned if an out of memory condition occurs.
- ASN_INPUT_E will be returned if Base16 decoding fails on the file.
- BUFFER_E will be returned if a chain buffer is bigger than the receiving buffer.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx;
byte certChainBuff[...];
long sz = sizeof(certBuff);
...
ret = wolfSSL_CTX_use_certificate_chain_buffer(ctx, certChainBuff, sz);
if (ret != SSL_SUCCESS) {
    // error loading certificate chain from buffer
}
...
```

C.52.2.241 function wolfSSL_use_certificate_buffer

```
int wolfSSL_use_certificate_buffer(
    WOLFSSL * ssl,
    const unsigned char * in,
    long sz,
    int format
)
```

This function loads a certificate buffer into the WOLFSSL object. It behaves like the non-buffered version, only differing in its ability to be called with a buffer as input instead of a file. The buffer is provided by the in argument of size sz. format specifies the format type of the buffer; SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM. Please see the examples for proper usage.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().
- **in** buffer containing certificate to load.
- **sz** size of the certificate located in buffer.
- **format** format of the certificate to be loaded. Possible values are SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM.

See:

- wolfSSL_CTX_load_verify_buffer
- wolfSSL_CTX_use_certificate_buffer
- wolfSSL_CTX_use_PrivateKey_buffer
- wolfSSL_CTX_use_certificate_chain_buffer
- wolfSSL_use_PrivateKey_buffer
- wolfSSL_use_certificate_chain_buffer

Return:

- SSL_SUCCESS upon success.
- SSL_BAD_FILETYPE will be returned if the file is the wrong format.
- SSL_BAD_FILE will be returned if the file doesn't exist, can't be read, or is corrupted.
- MEMORY_E will be returned if an out of memory condition occurs.
- ASN_INPUT_E will be returned if Base16 decoding fails on the file.

Example

```
int ret;
byte certBuff[...];
WOLFSSL* ssl = 0;
long buffSz = sizeof(certBuff);
...
```

```
ret = wolfSSL_use_certificate_buffer(ssl, certBuff, buffSz, SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
    // failed to load certificate from buffer
}
```

C.52.2.242 function wolfSSL_use_PrivateKey_buffer

```
int wolfSSL_use_PrivateKey_buffer(
    WOLFSSL * ssl,
    const unsigned char * in,
    long sz,
    int format
)
```

This function loads a private key buffer into the WOLFSSL object. It behaves like the non-buffered version, only differing in its ability to be called with a buffer as input instead of a file. The buffer is provided by the in argument of size sz. format specifies the format type of the buffer; SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM. Please see the examples for proper usage.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().
- **in** buffer containing private key to load.
- **sz** size of the private key located in buffer.
- **format** format of the private key to be loaded. Possible values are SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM.

See:

- wolfSSL_use_PrivateKey
- wolfSSL_CTX_load_verify_buffer
- wolfSSL_CTX_use_certificate_buffer
- wolfSSL_CTX_use_PrivateKey_buffer
- wolfSSL_CTX_use_certificate_chain_buffer
- wolfSSL_use_certificate_buffer
- wolfSSL_use_certificate_chain_buffer

Return:

- SSL_SUCCESS upon success.
- SSL_BAD_FILETYPE will be returned if the file is the wrong format.
- SSL_BAD_FILE will be returned if the file doesn't exist, can't be read, or is corrupted.
- MEMORY_E will be returned if an out of memory condition occurs.
- ASN_INPUT_E will be returned if Base16 decoding fails on the file.
- NO_PASSWORD will be returned if the key file is encrypted but no password is provided.

Example

```
int ret;
byte keyBuff[...];
WOLFSSL* ssl = 0;
long buffSz = sizeof(certBuff);
...
ret = wolfSSL_use_PrivateKey_buffer(ssl, keyBuff, buffSz, SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
    // failed to load private key from buffer
}
```

C.52.2.243 function wolfSSL_use_certificate_chain_buffer

```
int wolfSSL_use_certificate_chain_buffer(
    WOLFSSL * ssl,
    const unsigned char * in,
    long sz
)
```

This function loads a certificate chain buffer into the WOLFSSL object. It behaves like the non-buffered version, only differing in its ability to be called with a buffer as input instead of a file. The buffer is provided by the in argument of size sz. The buffer must be in PEM format and start with the subject's certificate, ending with the root certificate. Please see the examples for proper usage.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().
- **in** buffer containing certificate to load.
- **sz** size of the certificate located in buffer.

See:

- wolfSSL_CTX_load_verify_buffer
- wolfSSL_CTX_use_certificate_buffer
- wolfSSL_CTX_use_PrivateKey_buffer
- wolfSSL_CTX_use_certificate_chain_buffer
- wolfSSL_use_certificate_buffer
- wolfSSL_use_PrivateKey_buffer

Return:

- SSL_SUCCES upon success.
- SSL_BAD_FILETYPE will be returned if the file is the wrong format.
- SSL_BAD_FILE will be returned if the file doesn't exist, can't be read, or is corrupted.
- MEMORY_E will be returned if an out of memory condition occurs.
- ASN_INPUT_E will be returned if Base16 decoding fails on the file.
- BUFFER_E will be returned if a chain buffer is bigger than the receiving buffer.

Example

```
int ret;
byte certChainBuff[...];
WOLFSSL* ssl = 0;
long buffSz = sizeof(certBuff);
...
ret = wolfSSL_use_certificate_chain_buffer(ssl, certChainBuff, buffSz);
if (ret != SSL_SUCCESS) {
    // failed to load certificate chain from buffer
}
```

C.52.2.244 function wolfSSL_UnloadCertsKeys

```
int wolfSSL_UnloadCertsKeys(
    WOLFSSL *
)
```

This function unloads any certificates or keys that SSL owns.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See: wolfSSL_CTX_UnloadCAs

Return:

- SSL_SUCCESS - returned if the function executed successfully.
- BAD_FUNC_ARG - returned if the WOLFSSL object is NULL.

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
...
int unloadKeys = wolfSSL_UnloadCertsKeys(ssl);
if(unloadKeys != SSL_SUCCESS){
    // Failure case.
}
```

C.52.2.245 function wolfSSL_CTX_set_group_messages

```
int wolfSSL_CTX_set_group_messages(
    WOLFSSL_CTX *
)
```

This function turns on grouping of handshake messages where possible.

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().

See:

- wolfSSL_set_group_messages
- wolfSSL_CTX_new

Return:

- SSL_SUCCESS will be returned upon success.
- BAD_FUNC_ARG will be returned if the input context is null.

Example

```
WOLFSSL_CTX* ctx = 0;
...
ret = wolfSSL_CTX_set_group_messages(ctx);
if (ret != SSL_SUCCESS) {
    // failed to set handshake message grouping
}
```

C.52.2.246 function wolfSSL_set_group_messages

```
int wolfSSL_set_group_messages(
    WOLFSSL *
)
```

This function turns on grouping of handshake messages where possible.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().

See:

- wolfSSL_CTX_set_group_messages
- wolfSSL_new

Return:

- SSL_SUCCESS will be returned upon success.
- BAD_FUNC_ARG will be returned if the input context is null.

Example

```
WOLFSSL* ssl = 0;
...
ret = wolfSSL_set_group_messages(ssl);
if (ret != SSL_SUCCESS) {
// failed to set handshake message grouping
}
```

C.52.2.247 function wolfSSL_SetFuzzerCb

```
void wolfSSL_SetFuzzerCb(
    WOLFSSL * ssl,
    CallbackFuzzer cbf,
    void * fCtx
)
```

This function sets the fuzzer callback.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **cbf** a CallbackFuzzer type that is a function pointer of the form: int (*CallbackFuzzer)(WOLFSSL* ssl, const unsigned char* buf, int sz, int type, void* fuzzCtx);
- **fCtx** a void pointer type that will be set to the fuzzerCtx member of the WOLFSSL structure.

See: CallbackFuzzer

Return: none No returns.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
void* fCtx;

int callbackFuzzerCB(WOLFSSL* ssl, const unsigned char* buf, int sz,
            int type, void* fuzzCtx){
// function definition
}
…
wolfSSL_SetFuzzerCb(ssl, callbackFuzzerCB, fCtx);
```

C.52.2.248 function wolfSSL_DTLS_SetCookieSecret

```
int wolfSSL_DTLS_SetCookieSecret(
    WOLFSSL * ssl,
    const unsigned char * secret,
    unsigned int secretSz
)
```

This function sets a new dtls cookie secret.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

- **secret** a constant byte pointer representing the secret buffer.
- **secretSz** the size of the buffer.

See:

- ForceZero
- wc_RNG_GenerateBlock

Return:

- 0 returned if the function executed without an error.
- BAD_FUNC_ARG returned if there was an argument passed to the function with an unacceptable value.
- COOKIE_SECRET_SZ returned if the secret size is 0.
- MEMORY_ERROR returned if there was a problem allocating memory for a new cookie secret.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
const* byte secret;
word32 secretSz; // size of secret
…
if(!wolfSSL_DTLS_SetCookieSecret(ssl, secret, secretSz)){
    // Code block for failure to set DTLS cookie secret
} else {
    // Success! Cookie secret is set.
}
```

C.52.2.249 function wolfSSL_GetRNG

```
WC_RNG * wolfSSL_GetRNG(
    WOLFSSL * ssl
)
```

This function retrieves the random number.

Parameters:

- **ssl** pointer to a SSL object, created with wolfSSL_new().

See: wolfSSL_CTX_new_rng

Return:

- rng upon success.
- NULL if ssl is NULL.

Example

```
WOLFSSL* ssl;

wolfSSL_GetRNG(ssl);
```

C.52.2.250 function wolfSSL_CTX_SetMinVersion

```
int wolfSSL_CTX_SetMinVersion(
    WOLFSSL_CTX * ctx,
    int version
)
```

This function sets the minimum downgrade version allowed. Applicable only when the connection allows downgrade using (wolfSSLv23_client_method or wolfSSLv23_server_method).

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().
- **version** an integer representation of the version to be set as the minimum: WOLFSSL_SSLV3 = 0, WOLFSSL_TLSV1 = 1, WOLFSSL_TLSV1_1 = 2 or WOLFSSL_TLSV1_2 = 3.

See: SetMinVersionHelper

Return:

- SSL_SUCCESS returned if the function returned without error and the minimum version is set.
- BAD_FUNC_ARG returned if the WOLFSSL_CTX structure was NULL or if the minimum version is not supported.

Example

```
WOLFSSL_CTX* ctx = WOLFSSL_CTX_new( protocol method );
WOLFSSL* ssl = WOLFSSL_new(ctx);
int version; // macrop representation
…
if(wolfSSL_CTX_SetMinVersion(ssl->ctx, version) != SSL_SUCCESS){
    // Failed to set min version
}
```

C.52.2.251 function wolfSSL_SetMinVersion

```
int wolfSSL_SetMinVersion(
    WOLFSSL * ssl,
    int version
)
```

This function sets the minimum downgrade version allowed. Applicable only when the connection allows downgrade using (wolfSSLv23_client_method or wolfSSLv23_server_method).

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **version** an integer representation of the version to be set as the minimum: WOLFSSL_SSLV3 = 0, WOLFSSL_TLSV1 = 1, WOLFSSL_TLSV1_1 = 2 or WOLFSSL_TLSV1_2 = 3.

See: SetMinVersionHelper

Return:

- SSL_SUCCESS returned if this function and its subroutine executes without error.
- BAD_FUNC_ARG returned if the SSL object is NULL. In the subroutine this error is thrown if there is not a good version match.

Example

```
WOLFSSL_CTX* ctx = WOLFSSL_CTX_new(protocol method);
WOLFSSL* ssl = WOLFSSL_new(ctx);
int version;  macro representation
…
if(wolfSSL_CTX_SetMinVersion(ssl->ctx, version) != SSL_SUCCESS){
    Failed to set min version
}
```

C.52.2.252 function wolfSSL_GetObjectSize

```
int wolfSSL_GetObjectSize(
    void
)
```

This function returns the size of the WOLFSSL object and will be dependent on build options and settings. If SHOW_SIZES has been defined when building wolfSSL, this function will also print the sizes of individual objects within the WOLFSSL object (Suites, Ciphers, etc.) to stdout.

Parameters:

- **none** No parameters.

See: wolfSSL_new

Return: size This function returns the size of the WOLFSSL object.

Example

```
int size = 0;
size = wolfSSL_GetObjectSize();
printf("sizeof(WOLFSSL) = %d\n", size);
```

C.52.2.253 function wolfSSL_GetOutputSize

```
int wolfSSL_GetOutputSize(
    WOLFSSL * ssl,
    int inSz
)
```

Returns the record layer size of the plaintext input. This is helpful when an application wants to know how many bytes will be sent across the Transport layer, given a specified plaintext input size. This function must be called after the SSL/TLS handshake has been completed.

Parameters:

- **ssl** a pointer to a WOLFSSL object, created using wolfSSL_new().
- **inSz** size of plaintext data.

See: wolfSSL_GetMaxOutputSize

Return:

- size Upon success, the requested size will be returned
- INPUT_SIZE_E will be returned if the input size is greater than the maximum TLS fragment size (see wolfSSL_GetMaxOutputSize())
- BAD_FUNC_ARG will be returned upon invalid function argument, or if the SSL/TLS handshake has not been completed yet

Example

none

C.52.2.254 function wolfSSL_GetMaxOutputSize

```
int wolfSSL_GetMaxOutputSize(
    WOLFSSL *
)
```

Returns the maximum record layer size for plaintext data. This will correspond to either the maximum SSL/TLS record size as specified by the protocol standard, the maximum TLS fragment size as set by the TLS Max Fragment Length extension. This function is helpful when the application has called

wolfSSL_GetOutputSize() and received a INPUT_SIZE_E error. This function must be called after the SSL/TLS handshake has been completed.

Parameters:

- **ssl** a pointer to a WOLFSSL object, created using wolfSSL_new().

See: wolfSSL_GetOutputSize

Return:

- size Upon success, the maximum output size will be returned
- BAD_FUNC_ARG will be returned upon invalid function argument, or if the SSL/TLS handshake has not been completed yet.

Example

none

C.52.2.255 function wolfSSL_SetVersion

```
int wolfSSL_SetVersion(
    WOLFSSL * ssl,
    int version
)
```

This function sets the SSL/TLS protocol version for the specified SSL session (WOLFSSL object) using the version as specified by version. This will override the protocol setting for the SSL session (ssl) - originally defined and set by the SSL context (wolfSSL_CTX_new()) method type.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **version** SSL/TLS protocol version. Possible values include WOLFSSL_SSLV3, WOLFSSL_TLSV1, WOLFSSL_TLSV1_1, WOLFSSL_TLSV1_2.

See: wolfSSL_CTX_new

Return:

- SSL_SUCCESS upon success.
- BAD_FUNC_ARG will be returned if the input SSL object is NULL or an incorrect protocol version is given for version.

Example

```
int ret = 0;
WOLFSSL* ssl;
...

ret = wolfSSL_SetVersion(ssl, WOLFSSL_TLSV1);
if (ret != SSL_SUCCESS) {
    // failed to set SSL session protocol version
}
```

C.52.2.256 function wolfSSL_CTX_SetMacEncryptCb

```
void wolfSSL_CTX_SetMacEncryptCb(
    WOLFSSL_CTX * ctx,
    CallbackMacEncrypti cb
)
```

Allows caller to set the Atomic User Record Processing Mac/Encrypt Callback. The callback should return 0 for success or < 0 for an error. The ssl and ctx pointers are available for the user's convenience. macOut is the output buffer where the result of the mac should be stored. macIn is the mac input buffer and macInSz notes the size of the buffer. macContent and macVerify are needed for wolfSSL_SetTlsHmacInner() and be passed along as is. encOut is the output buffer where the result on the encryption should be stored. encIn is the input buffer to encrypt while encSz is the size of the input. An example callback can be found wolfssl/test.h myMacEncryptCb().

Parameters:

- **No** parameters.

See:

- wolfSSL_SetMacEncryptCtx
- wolfSSL_GetMacEncryptCtx

Return: none No return.

Example

none

C.52.2.257 function wolfSSL_SetMacEncryptCtx

```
void wolfSSL_SetMacEncryptCtx(
    WOLFSSL * ssl,
    void * ctx
)
```

Allows caller to set the Atomic User Record Processing Mac/Encrypt Callback Context to ctx.

Parameters:

- **none** No parameters.

See:

- wolfSSL_CTX_SetMacEncryptCb
- wolfSSL_GetMacEncryptCtx

Return: none No return.

Example

none

C.52.2.258 function wolfSSL_GetMacEncryptCtx

```
void * wolfSSL_GetMacEncryptCtx(
    WOLFSSL * ssl
)
```

Allows caller to retrieve the Atomic User Record Processing Mac/Encrypt Callback Context previously stored with wolfSSL_SetMacEncryptCtx().

Parameters:

- **none** No parameters.

See:

- wolfSSL_CTX_SetMacEncryptCb
- wolfSSL_SetMacEncryptCtx

Return:

- pointer If successful the call will return a valid pointer to the context.
- NULL will be returned for a blank context.

Example

none

C.52.2.259 function wolfSSL_CTX_SetDecryptVerifyCb

```
void wolfSSL_CTX_SetDecryptVerifyCb(
    WOLFSSL_CTX * ctx,
    CallbackDecryptVerify cb
)
```

Allows caller to set the Atomic User Record Processing Decrypt/Verify Callback. The callback should return 0 for success or < 0 for an error. The ssl and ctx pointers are available for the user's convenience. decOut is the output buffer where the result of the decryption should be stored. decIn is the encrypted input buffer and decInSz notes the size of the buffer. content and verify are needed for wolfSSL_SetTlsHmacInner() and be passed along as is. padSz is an output variable that should be set with the total value of the padding. That is, the mac size plus any padding and pad bytes. An example callback can be found wolfssl/test.h myDecryptVerifyCb().

Parameters:

- **none** No parameters.

See:

- wolfSSL_SetMacEncryptCtx
- wolfSSL_GetMacEncryptCtx

Return: none No returns.

Example

none

C.52.2.260 function wolfSSL_SetDecryptVerifyCtx

```
void wolfSSL_SetDecryptVerifyCtx(
    WOLFSSL * ssl,
    void * ctx
)
```

Allows caller to set the Atomic User Record Processing Decrypt/Verify Callback Context to ctx.

Parameters:

- **none** No parameters.

See:

- wolfSSL_CTX_SetDecryptVerifyCb
- wolfSSL_GetDecryptVerifyCtx

Return: none No returns.

Example

none

C.52.2.261 function wolfSSL_GetDecryptVerifyCtx

```
void * wolfSSL_GetDecryptVerifyCtx(
    WOLFSSL * ssl
)
```

Allows caller to retrieve the Atomic User Record Processing Decrypt/Verify Callback Context previously stored with wolfSSL_SetDecryptVerifyCtx().

Parameters:

- **none** No parameters.

See:

- wolfSSL_CTX_SetDecryptVerifyCb
- wolfSSL_SetDecryptVerifyCtx

Return:

- pointer If successful the call will return a valid pointer to the context.
- NULL will be returned for a blank context.

Example

none

C.52.2.262 function wolfSSL_GetMacSecret

```
const unsigned char * wolfSSL_GetMacSecret(
    WOLFSSL * ssl,
    int verify
)
```

Allows retrieval of the Hmac/Mac secret from the handshake process. The verify parameter specifies whether this is for verification of a peer message.

Parameters:

- **ssl** a pointer to a WOLFSSL object, created using wolfSSL_new().
- **verify** specifies whether this is for verification of a peer message.

See: wolfSSL_GetHmacSize

Return:

- pointer If successful the call will return a valid pointer to the secret. The size of the secret can be obtained from wolfSSL_GetHmacSize().
- NULL will be returned for an error state.

Example

none

C.52.2.263 function wolfSSL_GetClientWriteKey

```
const unsigned char * wolfSSL_GetClientWriteKey(
    WOLFSSL *
)
```

Allows retrieval of the client write key from the handshake process.

Parameters:

- **ssl** a pointer to a WOLFSSL object, created using wolfSSL_new().

See:

- wolfSSL_GetKeySize
- wolfSSL_GetClientWriteIV

Return:

- pointer If successful the call will return a valid pointer to the key. The size of the key can be obtained from wolfSSL_GetKeySize().
- NULL will be returned for an error state.

Example

none

C.52.2.264 function wolfSSL_GetClientWriteIV

```
const unsigned char * wolfSSL_GetClientWriteIV(
    WOLFSSL *
)
```

Allows retrieval of the client write IV (initialization vector) from the handshake process.

Parameters:

- **ssl** a pointer to a WOLFSSL object, created using wolfSSL_new().

See:

- wolfSSL_GetCipherBlockSize()
- wolfSSL_GetClientWriteKey()

Return:

- pointer If successful the call will return a valid pointer to the IV. The size of the IV can be obtained from wolfSSL_GetCipherBlockSize().
- NULL will be returned for an error state.

Example

none

C.52.2.265 function wolfSSL_GetServerWriteKey

```
const unsigned char * wolfSSL_GetServerWriteKey(
    WOLFSSL *
)
```

Allows retrieval of the server write key from the handshake process.

Parameters:

- **ssl** a pointer to a WOLFSSL object, created using wolfSSL_new().

See:

- wolfSSL_GetKeySize
- wolfSSL_GetServerWriteIV

Return:

- pointer If successful the call will return a valid pointer to the key. The size of the key can be obtained from wolfSSL_GetKeySize().
- NULL will be returned for an error state.

Example

none

C.52.2.266 function wolfSSL_GetServerWriteIV

```
const unsigned char * wolfSSL_GetServerWriteIV(
    WOLFSSL *
)
```

Allows retrieval of the server write IV (initialization vector) from the handshake process.

Parameters:

- **ssl** a pointer to a WOLFSSL object, created using wolfSSL_new().

See:

- wolfSSL_GetCipherBlockSize
- wolfSSL_GetClientWriteKey

Return:

- pointer If successful the call will return a valid pointer to the IV. The size of the IV can be obtained from wolfSSL_GetCipherBlockSize().
- NULL will be returned for an error state.

C.52.2.267 function wolfSSL_GetKeySize

```
int wolfSSL_GetKeySize(
    WOLFSSL *
)
```

Allows retrieval of the key size from the handshake process.

Parameters:

- **ssl** a pointer to a WOLFSSL object, created using wolfSSL_new().

See:

- wolfSSL_GetClientWriteKey
- wolfSSL_GetServerWriteKey

Return:

- size If successful the call will return the key size in bytes.
- BAD_FUNC_ARG will be returned for an error state.

Example

none

C.52.2.268 function wolfSSL_GetIVSize

```
int wolfSSL_GetIVSize(
    WOLFSSL *
)
```

Returns the iv_size member of the specs structure held in the WOLFSSL struct.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_GetKeySize
- wolfSSL_GetClientWriteIV
- wolfSSL_GetServerWriteIV

Return:

- iv_size returns the value held in ssl->specs.iv_size.
- BAD_FUNC_ARG returned if the WOLFSSL structure is NULL.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
int ivSize;
...
ivSize = wolfSSL_GetIVSize(ssl);

if(ivSize > 0){
    // ivSize holds the specs.iv_size value.
}
```

C.52.2.269 function wolfSSL_GetSide

```
int wolfSSL_GetSide(
    WOLFSSL *
)
```

Allows retrieval of the side of this WOLFSSL connection.

Parameters:

- **ssl** a pointer to a WOLFSSL object, created using wolfSSL_new().

See:

- wolfSSL_GetClientWriteKey
- wolfSSL_GetServerWriteKey

Return:

- success If successful the call will return either WOLFSSL_SERVER_END or WOLFSSL_CLIENT_END depending on the side of WOLFSSL object.
- BAD_FUNC_ARG will be returned for an error state.

Example

none

C.52.2.270 function wolfSSL_IsTLSv1_1

```
int wolfSSL_IsTLSv1_1(
    WOLFSSL *
)
```

Allows caller to determine if the negotiated protocol version is at least TLS version 1.1 or greater.

Parameters:

- **ssl** a pointer to a WOLFSSL object, created using wolfSSL_new().

See: wolfSSL_GetSide

Return:

- true/false If successful the call will return 1 for true or 0 for false.
- BAD_FUNC_ARG will be returned for an error state.

Example

none

C.52.2.271 function wolfSSL_GetBulkCipher

```
int wolfSSL_GetBulkCipher(
    WOLFSSL *
)
```

Allows caller to determine the negotiated bulk cipher algorithm from the handshake.

Parameters:

- **ssl** a pointer to a WOLFSSL object, created using wolfSSL_new().

See:

- wolfSSL_GetCipherBlockSize
- wolfSSL_GetKeySize

Return:

- If successful the call will return one of the following: wolfssl_cipher_null, wolfssl_des, wolfssl_triple_des, wolfssl_aes, wolfssl_aes_gcm, wolfssl_aes_ccm, wolfssl_camellia.
- BAD_FUNC_ARG will be returned for an error state.

Example

none

C.52.2.272 function wolfSSL_GetCipherBlockSize

```
int wolfSSL_GetCipherBlockSize(
    WOLFSSL *
)
```

Allows caller to determine the negotiated cipher block size from the handshake.

Parameters:

- **ssl** a pointer to a WOLFSSL object, created using wolfSSL_new().

See:

- wolfSSL_GetBulkCipher
- wolfSSL_GetKeySize

Return:

- size If successful the call will return the size in bytes of the cipher block size.
- BAD_FUNC_ARG will be returned for an error state.

Example

none

C.52.2.273 function wolfSSL_GetAeadMacSize

```
int wolfSSL_GetAeadMacSize(
    WOLFSSL *
)
```

Allows caller to determine the negotiated aead mac size from the handshake. For cipher type WOLFSSL_AEAD_TYPE.

Parameters:

- **ssl** a pointer to a WOLFSSL object, created using wolfSSL_new().

See:

- wolfSSL_GetBulkCipher
- wolfSSL_GetKeySize

Return:

- size If successful the call will return the size in bytes of the aead mac size.
- BAD_FUNC_ARG will be returned for an error state.

Example

none

C.52.2.274 function wolfSSL_GetHmacSize

```
int wolfSSL_GetHmacSize(
    WOLFSSL *
)
```

Allows caller to determine the negotiated (h)mac size from the handshake. For cipher types except WOLFSSL_AEAD_TYPE.

Parameters:

- **ssl** a pointer to a WOLFSSL object, created using wolfSSL_new().

See:

- wolfSSL_GetBulkCipher
- wolfSSL_GetHmacType

Return:

- size If successful the call will return the size in bytes of the (h)mac size.
- BAD_FUNC_ARG will be returned for an error state.

Example

none

C.52.2.275 function wolfSSL_GetHmacType

```
int wolfSSL_GetHmacType(
    WOLFSSL *
)
```

Allows caller to determine the negotiated (h)mac type from the handshake. For cipher types except WOLFSSL_AEAD_TYPE.

Parameters:

- **ssl** a pointer to a WOLFSSL object, created using wolfSSL_new().

See:

- wolfSSL_GetBulkCipher
- wolfSSL_GetHmacSize

Return:

- If successful the call will return one of the following: MD5, SHA, SHA256, SHA384.
- BAD_FUNC_ARG may be returned for an error state.
- SSL_FATAL_ERROR may also be returned for an error state.

Example

none

C.52.2.276 function wolfSSL_GetCipherType

```
int wolfSSL_GetCipherType(
    WOLFSSL *
)
```

Allows caller to determine the negotiated cipher type from the handshake.

Parameters:

- **ssl** a pointer to a WOLFSSL object, created using wolfSSL_new().

See:

- wolfSSL_GetBulkCipher
- wolfSSL_GetHmacType

Return:

- If successful the call will return one of the following: WOLFSSL_BLOCK_TYPE, WOLFSSL_STREAM_TYPE, WOLFSSL_AEAD_TYPE.
- BAD_FUNC_ARG will be returned for an error state.

Example

none

C.52.2.277 function wolfSSL_SetTlsHmacInner

```
int wolfSSL_SetTlsHmacInner(
    WOLFSSL * ssl,
    byte * inner,
    word32 sz,
    int content,
    int verify
)
```

Allows caller to set the Hmac Inner vector for message sending/receiving. The result is written to inner which should be at least wolfSSL_GetHmacSize() bytes. The size of the message is specified by sz, content is the type of message, and verify specifies whether this is a verification of a peer message. Valid for cipher types excluding WOLFSSL_AEAD_TYPE.

Parameters:

- **none** No parameters.

See:

- wolfSSL_GetBulkCipher
- wolfSSL_GetHmacType

Return:

- 1 upon success.
- BAD_FUNC_ARG will be returned for an error state.

Example

none

C.52.2.278 function wolfSSL_CTX_SetEccSignCb

```
void wolfSSL_CTX_SetEccSignCb(
    WOLFSSL_CTX * ctx,
    CallbackEccSign cb
)
```

Allows caller to set the Public Key Callback for ECC Signing. The callback should return 0 for success or < 0 for an error. The ssl and ctx pointers are available for the user's convenience. in is the input buffer to sign while inSz denotes the length of the input. out is the output buffer where the result of the signature should be stored. outSz is an input/output variable that specifies the size of the output buffer upon invocation and the actual size of the signature should be stored there before returning. keyDer is the ECC Private key in ASN1 format and keySz is the length of the key in bytes. An example callback can be found wolfssl/test.h myEccSign().

Parameters:

- **none** No parameters.

See:

- wolfSSL_SetEccSignCtx
- wolfSSL_GetEccSignCtx

Return: none No returns.

Example

none

C.52.2.279 function wolfSSL_SetEccSignCtx

```
void wolfSSL_SetEccSignCtx(
    WOLFSSL * ssl,
    void * ctx
)
```

Allows caller to set the Public Key Ecc Signing Callback Context to ctx.

Parameters:

- **ssl** a pointer to a WOLFSSL object, created using wolfSSL_new().
- **ctx** a pointer to the user context to be stored

See:

- wolfSSL_CTX_SetEccSignCb
- wolfSSL_GetEccSignCtx

Return: none No returns.

Example

none

C.52.2.280 function wolfSSL_GetEccSignCtx

```
void * wolfSSL_GetEccSignCtx(
    WOLFSSL * ssl
)
```

Allows caller to retrieve the Public Key Ecc Signing Callback Context previously stored with wolfSSL_SetEccSignCtx().

Parameters:

- **ssl** a pointer to a WOLFSSL object, created using wolfSSL_new().

See:

- wolfSSL_CTX_SetEccSignCb
- wolfSSL_SetEccSignCtx

Return:

- pointer If successful the call will return a valid pointer to the context.
- NULL will be returned for a blank context.

Example

none

C.52.2.281 function wolfSSL_CTX_SetEccSignCtx

```
void wolfSSL_CTX_SetEccSignCtx(
    WOLFSSL_CTX * ctx,
    void * userCtx
)
```

Allows caller to set the Public Key Ecc Signing Callback Context to ctx.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created with wolfSSL_CTX_new().
- **ctx** a pointer to the user context to be stored

See:

- wolfSSL_CTX_SetEccSignCb
- wolfSSL_CTX_GetEccSignCtx

Return: none No returns.

Example

none

C.52.2.282 function wolfSSL_CTX_GetEccSignCtx

```
void * wolfSSL_CTX_GetEccSignCtx(
    WOLFSSL_CTX * ctx
)
```

Allows caller to retrieve the Public Key Ecc Signing Callback Context previously stored with wolfSSL_SetEccSignCtx().

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created with wolfSSL_CTX_new().

See:

- wolfSSL_CTX_SetEccSignCb
- wolfSSL_CTX_SetEccSignCtx

Return:

- pointer If successful the call will return a valid pointer to the context.
- NULL will be returned for a blank context.

Example

none

C.52.2.283 function wolfSSL_CTX_SetEccVerifyCb

```
void wolfSSL_CTX_SetEccVerifyCb(
    WOLFSSL_CTX * ctx,
    CallbackEccVerify cb
)
```

Allows caller to set the Public Key Callback for ECC Verification. The callback should return 0 for success or < 0 for an error. The ssl and ctx pointers are available for the user's convenience. sig is the signature to verify and sigSz denotes the length of the signature. hash is an input buffer containing the digest of the message and hashSz denotes the length in bytes of the hash. result is an output variable where the result of the verification should be stored, 1 for success and 0 for failure. keyDer is the ECC Private key in ASN1 format and keySz is the length of the key in bytes. An example callback can be found wolfssl/test.h myEccVerify().

Parameters:

- **none** No parameters.

See:

- wolfSSL_SetEccVerifyCtx
- wolfSSL_GetEccVerifyCtx

Return: none No returns.

Example

none

C.52.2.284 function wolfSSL_SetEccVerifyCtx

```
void wolfSSL_SetEccVerifyCtx(
    WOLFSSL * ssl,
    void * ctx
)
```

Allows caller to set the Public Key Ecc Verification Callback Context to ctx.

Parameters:

- **none** No parameters.

See:

- wolfSSL_CTX_SetEccVerifyCb
- wolfSSL_GetEccVerifyCtx

Return: none No returns.

Example

none

C.52.2.285 function wolfSSL_GetEccVerifyCtx

```
void * wolfSSL_GetEccVerifyCtx(
    WOLFSSL * ssl
)
```

Allows caller to retrieve the Public Key Ecc Verification Callback Context previously stored with wolfSSL_SetEccVerifyCtx().

Parameters:

- **none** No parameters.

See:

- wolfSSL_CTX_SetEccVerifyCb
- wolfSSL_SetEccVerifyCtx

Return:

- pointer If successful the call will return a valid pointer to the context.
- NULL will be returned for a blank context.

Example

none

C.52.2.286 function wolfSSL_CTX_SetRsaSignCb

```
void wolfSSL_CTX_SetRsaSignCb(
    WOLFSSL_CTX * ctx,
    CallbackRsaSign cb
)
```

Allows caller to set the Public Key Callback for RSA Signing. The callback should return 0 for success or < 0 for an error. The ssl and ctx pointers are available for the user's convenience. in is the input buffer to sign while inSz denotes the length of the input. out is the output buffer where the result of the signature should be stored. outSz is an input/output variable that specifies the size of the output buffer upon invocation and the actual size of the signature should be stored there before returning. keyDer is the RSA Private key in ASN1 format and keySz is the length of the key in bytes. An example callback can be found wolfssl/test.h myRsaSign().

Parameters:

- **none** No parameters.

See:

- wolfSSL_SetRsaSignCtx
- wolfSSL_GetRsaSignCtx

Return: none No returns.

Example

none

C.52.2.287 function wolfSSL_SetRsaSignCtx

```
void wolfSSL_SetRsaSignCtx(
    WOLFSSL * ssl,
    void * ctx
)
```

Allows caller to set the Public Key RSA Signing Callback Context to ctx.

Parameters:

- **none** No parameters.

See:

- wolfSSL_CTX_SetRsaSignCb
- wolfSSL_GetRsaSignCtx

Return: none No Returns.

Example

none

C.52.2.288 function wolfSSL_GetRsaSignCtx

```
void * wolfSSL_GetRsaSignCtx(
    WOLFSSL * ssl
)
```

Allows caller to retrieve the Public Key RSA Signing Callback Context previously stored with wolfSSL_SetRsaSignCtx().

Parameters:

- **none** No parameters.
- **none** No parameters.

See:

- wolfSSL_CTX_SetRsaSignCb
- wolfSSL_SetRsaSignCtx

Return:

- pointer If successful the call will return a valid pointer to the context.
- NULL will be returned for a blank context.

Example

none

C.52.2.289 function wolfSSL_CTX_SetRsaVerifyCb

```
void wolfSSL_CTX_SetRsaVerifyCb(
    WOLFSSL_CTX * ctx,
    CallbackRsaVerify cb
)
```

Allows caller to set the Public Key Callback for RSA Verification. The callback should return the number of plaintext bytes for success or < 0 for an error. The ssl and ctx pointers are available for the user's convenience. sig is the signature to verify and sigSz denotes the length of the signature. out should be set to the beginning of the verification buffer after the decryption process and any padding. keyDer

is the RSA Public key in ASN1 format and keySz is the length of the key in bytes. An example callback can be found wolfssl/test.h myRsaVerify().

Parameters:

- **none** No parameters.

See:

- wolfSSL_SetRsaVerifyCtx
- wolfSSL_GetRsaVerifyCtx

Return: none No returns.

C.52.2.290 function wolfSSL_SetRsaVerifyCtx

```
void wolfSSL_SetRsaVerifyCtx(
    WOLFSSL * ssl,
    void * ctx
)
```

Allows caller to set the Public Key RSA Verification Callback Context to ctx.

Parameters:

- **none** No parameters.

See:

- wolfSSL_CTX_SetRsaVerifyCb
- wolfSSL_GetRsaVerifyCtx

Return: none No returns.

Example

none

C.52.2.291 function wolfSSL_GetRsaVerifyCtx

```
void * wolfSSL_GetRsaVerifyCtx(
    WOLFSSL * ssl
)
```

Allows caller to retrieve the Public Key RSA Verification Callback Context previously stored with wolfSSL_SetRsaVerifyCtx().

Parameters:

- **none** No parameters.

See:

- wolfSSL_CTX_SetRsaVerifyCb
- wolfSSL_SetRsaVerifyCtx

Return:

- pointer If successful the call will return a valid pointer to the context.
- NULL will be returned for a blank context.

Example

none

C.52.2.292 function wolfSSL_CTX_SetRsaEncCb

```
void wolfSSL_CTX_SetRsaEncCb(
    WOLFSSL_CTX * ctx,
    CallbackRsaEnc cb
)
```

Allows caller to set the Public Key Callback for RSA Public Encrypt. The callback should return 0 for success or < 0 for an error. The ssl and ctx pointers are available for the user's convenience. in is the input buffer to encrypt while inSz denotes the length of the input. out is the output buffer where the result of the encryption should be stored. outSz is an input/output variable that specifies the size of the output buffer upon invocation and the actual size of the encryption should be stored there before returning. keyDer is the RSA Public key in ASN1 format and keySz is the length of the key in bytes. An example callback can be found wolfssl/test.h myRsaEnc().

Parameters:

- **none** No parameters.

See:

- wolfSSL_SetRsaEncCtx
- wolfSSL_GetRsaEncCtx

Return: none No returns.

Examples

none

C.52.2.293 function wolfSSL_SetRsaEncCtx

```
void wolfSSL_SetRsaEncCtx(
    WOLFSSL * ssl,
    void * ctx
)
```

Allows caller to set the Public Key RSA Public Encrypt Callback Context to ctx.

Parameters:

- **none** No parameters.

See:

- wolfSSL_CTX_SetRsaEncCb
- wolfSSL_GetRsaEncCtx

Return: none No returns.

Example

none

C.52.2.294 function wolfSSL_GetRsaEncCtx

```
void * wolfSSL_GetRsaEncCtx(
    WOLFSSL * ssl
)
```

Allows caller to retrieve the Public Key RSA Public Encrypt Callback Context previously stored with wolfSSL_SetRsaEncCtx().

Parameters:

- **none** No parameters.

See:

- wolfSSL_CTX_SetRsaEncCb
- wolfSSL_SetRsaEncCtx

Return:

- pointer If successful the call will return a valid pointer to the context.
- NULL will be returned for a blank context.

Example

none

C.52.2.295 function wolfSSL_CTX_SetRsaDecCb

```
void wolfSSL_CTX_SetRsaDecCb(
    WOLFSSL_CTX * ctx,
    CallbackRsaDec cb
)
```

Allows caller to set the Public Key Callback for RSA Private Decrypt. The callback should return the number of plaintext bytes for success or < 0 for an error. The ssl and ctx pointers are available for the user's convenience. in is the input buffer to decrypt and inSz denotes the length of the input. out should be set to the beginning of the decryption buffer after the decryption process and any padding. keyDer is the RSA Private key in ASN1 format and keySz is the length of the key in bytes. An example callback can be found wolfssl/test.h myRsaDec().

Parameters:

- **none** No parameters.

See:

- wolfSSL_SetRsaDecCtx
- wolfSSL_GetRsaDecCtx

Return: none No returns.

Example

none

C.52.2.296 function wolfSSL_SetRsaDecCtx

```
void wolfSSL_SetRsaDecCtx(
    WOLFSSL * ssl,
    void * ctx
)
```

Allows caller to set the Public Key RSA Private Decrypt Callback Context to ctx.

Parameters:

- **none** No parameters.

See:

- wolfSSL_CTX_SetRsaDecCb
- wolfSSL_GetRsaDecCtx

Return: none No returns.

Example

none

C.52.2.297 function wolfSSL_GetRsaDecCtx

```
void * wolfSSL_GetRsaDecCtx(
    WOLFSSL * ssl
)
```

Allows caller to retrieve the Public Key RSA Private Decrypt Callback Context previously stored with wolfSSL_SetRsaDecCtx().

Parameters:

- **none** No parameters.

See:

- wolfSSL_CTX_SetRsaDecCb
- wolfSSL_SetRsaDecCtx

Return:

- pointer If successful the call will return a valid pointer to the context.
- NULL will be returned for a blank context.

Example

none

C.52.2.298 function wolfSSL_CTX_SetCACb

```
void wolfSSL_CTX_SetCACb(
    WOLFSSL_CTX * ctx,
    CallbackCACache cb
)
```

This function registers a callback with the SSL context (WOLFSSL_CTX) to be called when a new CA certificate is loaded into wolfSSL. The callback is given a buffer with the DER-encoded certificate.

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().
- **callback** function to be registered as the CA callback for the wolfSSL context, ctx. The signature of this function must follow that as shown above in the Synopsis section.

See: wolfSSL_CTX_load_verify_locations

Return: none No return.

Example

```
WOLFSSL_CTX* ctx = 0;

// CA callback prototype
int MyCACallback(unsigned char *der, int sz, int type);

// Register the custom CA callback with the SSL context
wolfSSL_CTX_SetCACb(ctx, MyCACallback);
```

```
int MyCACallback(unsigned char* der, int sz, int type)
{
    // custom CA callback function, DER-encoded cert
    // located in “der” of size “sz” with type “type”
}
```

C.52.2.299 function wolfSSL_CertManagerNew_ex

```
WOLFSSL_CERT_MANAGER * wolfSSL_CertManagerNew_ex(
    void * heap
)
```

Allocates and initializes a new Certificate Manager context. This context may be used independent of SSL needs. It may be used to load certificates, verify certificates, and check the revocation status.

Parameters:

- **none** No parameters.

See: wolfSSL_CertManagerFree

Return:

- WOLFSSL_CERT_MANAGER If successful the call will return a valid WOLFSSL_CERT_MANAGER pointer.
- NULL will be returned for an error state.

C.52.2.300 function wolfSSL_CertManagerNew

```
WOLFSSL_CERT_MANAGER * wolfSSL_CertManagerNew(
    void
)
```

Allocates and initializes a new Certificate Manager context. This context may be used independent of SSL needs. It may be used to load certificates, verify certificates, and check the revocation status.

Parameters:

- **none** No parameters.

See: wolfSSL_CertManagerFree

Return:

- WOLFSSL_CERT_MANAGER If successful the call will return a valid WOLFSSL_CERT_MANAGER pointer.
- NULL will be returned for an error state.

Example

```
#import <wolfssl/ssl.h>

WOLFSSL_CERT_MANAGER* cm;
cm = wolfSSL_CertManagerNew();
if (cm == NULL) {
// error creating new cert manager
}
```

C.52.2.301 function wolfSSL_CertManagerFree

```
void wolfSSL_CertManagerFree(
    WOLFSSL_CERT_MANAGER *
)
```

Frees all resources associated with the Certificate Manager context. Call this when you no longer need to use the Certificate Manager.

Parameters:

- **cm** a pointer to a WOLFSSL_CERT_MANAGER structure, created using wolfSSL_CertManagerNew().

See: wolfSSL_CertManagerNew

Return: none

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_CERT_MANAGER* cm;
...
wolfSSL_CertManagerFree(cm);
```

C.52.2.302 function wolfSSL_CertManagerLoadCA

```
int wolfSSL_CertManagerLoadCA(
    WOLFSSL_CERT_MANAGER * cm,
    const char * f,
    const char * d
)
```

Specifies the locations for CA certificate loading into the manager context. The PEM certificate CAfile may contain several trusted CA certificates. If CApath is not NULL it specifies a directory containing CA certificates in PEM format.

Parameters:

- **cm** a pointer to a WOLFSSL_CERT_MANAGER structure, created using wolfSSL_CertManagerNew().
- **file** pointer to the name of the file containing CA certificates to load.
- **path** pointer to the name of a directory path containing CA c ertificates to load. The NULL pointer may be used if no certificate directory is desired.

See: wolfSSL_CertManagerVerify

Return:

- SSL_SUCCESS If successful the call will return.
- SSL_BAD_FILETYPE will be returned if the file is the wrong format.
- SSL_BAD_FILE will be returned if the file doesn't exist, can't be read, or is corrupted.
- MEMORY_E will be returned if an out of memory condition occurs.
- ASN_INPUT_E will be returned if Base16 decoding fails on the file.
- BAD_FUNC_ARG is the error that will be returned if a pointer is not provided.
- SSL_FATAL_ERROR - will be returned upon failure.

Example

```
#include <wolfssl/ssl.h>

int ret = 0;
WOLFSSL_CERT_MANAGER* cm;
```

```
...
ret = wolfSSL_CertManagerLoadCA(cm, “path/to/cert-file.pem”, 0);
if (ret != SSL_SUCCESS) {
// error loading CA certs into cert manager
}
```

C.52.2.303 function wolfSSL_CertManagerLoadCABuffer

```
int wolfSSL_CertManagerLoadCABuffer(
    WOLFSSL_CERT_MANAGER * cm,
    const unsigned char * in,
    long sz,
    int format
)
```

Loads the CA Buffer by calling wolfSSL_CTX_load_verify_buffer and returning that result using a temporary cm so as not to lose the information in the cm passed into the function.

Parameters:

- **cm** a pointer to a WOLFSSL_CERT_MANAGER structure, created using wolfSSL_CertManagerNew().
- **in** buffer for cert information.
- **sz** length of the buffer.
- **format** certificate format, either PEM or DER.

See:

- wolfSSL_CTX_load_verify_buffer
- ProcessChainBuffer
- ProcessBuffer
- cm_pick_method

Return:

- SSL_FATAL_ERROR is returned if the WOLFSSL_CERT_MANAGER struct is NULL or if wolfSSL_CTX_new() returns NULL.
- SSL_SUCCESS is returned for a successful execution.

Example

```
WOLFSSL_CERT_MANAGER* cm = (WOLFSSL_CERT_MANAGER*)vp;
…
const unsigned char* in;
long sz;
int format;
…
if(wolfSSL_CertManagerLoadCABuffer(vp, sz, format) != SSL_SUCCESS){
    Error returned. Failure case code block.
}
```

C.52.2.304 function wolfSSL_CertManagerUnloadCAs

```
int wolfSSL_CertManagerUnloadCAs(
    WOLFSSL_CERT_MANAGER * cm
)
```

This function unloads the CA signer list.

Parameters:

- **cm** a pointer to a WOLFSSL_CERT_MANAGER structure, created using wolfSSL_CertManagerNew().

See: UnlockMutex

Return:

- SSL_SUCCESS returned on successful execution of the function.
- BAD_FUNC_ARG returned if the WOLFSSL_CERT_MANAGER is NULL.
- BAD_MUTEX_E returned if there was a mutex error.

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_CTX* ctx = wolfSSL_CTX_new(protocol method);
WOLFSSL_CERT_MANAGER* cm = wolfSSL_CTX_GetCertManager(ctx);
...
if(wolfSSL_CertManagerUnloadCAs(cm) != SSL_SUCCESS){
    Failure case.
}
```

C.52.2.305 function wolfSSL_CertManagerUnloadIntermediateCerts

```
int wolfSSL_CertManagerUnloadIntermediateCerts(
    WOLFSSL_CERT_MANAGER * cm
)
```

This function unloads intermediate certificates add to the CA signer list.

Parameters:

- **cm** a pointer to a WOLFSSL_CERT_MANAGER structure, created using wolfSSL_CertManagerNew().

See: UnlockMutex

Return:

- SSL_SUCCESS returned on successful execution of the function.
- BAD_FUNC_ARG returned if the WOLFSSL_CERT_MANAGER is NULL.
- BAD_MUTEX_E returned if there was a mutex error.

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_CTX* ctx = wolfSSL_CTX_new(protocol method);
WOLFSSL_CERT_MANAGER* cm = wolfSSL_CTX_GetCertManager(ctx);
...
if(wolfSSL_CertManagerUnloadIntermediateCerts(cm) != SSL_SUCCESS){
    Failure case.
}
```

C.52.2.306 function wolfSSL_CertManagerUnload_trust_peers

```
int wolfSSL_CertManagerUnload_trust_peers(
    WOLFSSL_CERT_MANAGER * cm
)
```

The function will free the Trusted Peer linked list and unlocks the trusted peer list.

Parameters:

- **cm** a pointer to a WOLFSSL_CERT_MANAGER structure, created using wolfSSL_CertManagerNew().

See: UnLockMutex

Return:

- SSL_SUCCESS if the function completed normally.
- BAD_FUNC_ARG if the WOLFSSL_CERT_MANAGER is NULL.
- BAD_MUTEX_E mutex error if tpLock, a member of the WOLFSSL_CERT_MANAGER struct, is 0 (nill).

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_CTX* ctx = WOLFSSL_CTX_new(Protocol define);
WOLFSSL_CERT_MANAGER* cm = wolfSSL_CertManagerNew();
...
if(wolfSSL_CertManagerUnload_trust_peers(cm) != SSL_SUCCESS){
    The function did not execute successfully.
}
```

C.52.2.307 function wolfSSL_CertManagerVerify

```
int wolfSSL_CertManagerVerify(
    WOLFSSL_CERT_MANAGER * cm,
    const char * f,
    int format
)
```

Specifies the certificate to verify with the Certificate Manager context. The format can be SSL_FILETYPE_PEM or SSL_FILETYPE_ASN1.

Parameters:

- **cm** a pointer to a WOLFSSL_CERT_MANAGER structure, created using wolfSSL_CertManagerNew().
- **fname** pointer to the name of the file containing the certificates to verify.
- **format** format of the certificate to verify - either SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM.

See:

- wolfSSL_CertManagerLoadCA
- wolfSSL_CertManagerVerifyBuffer

Return:

- SSL_SUCCESS If successful.
- ASN_SIG_CONFIRM_E will be returned if the signature could not be verified.
- ASN_SIG_OID_E will be returned if the signature type is not supported.
- CRL_CERT_REVOKED is an error that is returned if this certificate has been revoked.
- CRL_MISSING is an error that is returned if a current issuer CRL is not available.
- ASN_BEFORE_DATE_E will be returned if the current date is before the before date.
- ASN_AFTER_DATE_E will be returned if the current date is after the after date.
- SSL_BAD_FILETYPE will be returned if the file is the wrong format.
- SSL_BAD_FILE will be returned if the file doesn't exist, can't be read, or is corrupted.
- MEMORY_E will be returned if an out of memory condition occurs.
- ASN_INPUT_E will be returned if Base16 decoding fails on the file.
- BAD_FUNC_ARG is the error that will be returned if a pointer is not provided.

Example

```
int ret = 0;
WOLFSSL_CERT_MANAGER* cm;
...

ret = wolfSSL_CertManagerVerify(cm, “path/to/cert-file.pem”,
SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
    error verifying certificate
}
```

C.52.2.308 function wolfSSL_CertManagerVerifyBuffer

```
int wolfSSL_CertManagerVerifyBuffer(
    WOLFSSL_CERT_MANAGER * cm,
    const unsigned char * buff,
    long sz,
    int format
)
```

Specifies the certificate buffer to verify with the Certificate Manager context. The format can be SSL_FILETYPE_PEM or SSL_FILETYPE_ASN1.

Parameters:

- **cm** a pointer to a WOLFSSL_CERT_MANAGER structure, created using wolfSSL_CertManagerNew().
- **buff** buffer containing the certificates to verify.
- **sz** size of the buffer, buf.
- **format** format of the certificate to verify, located in buf - either SSL_FILETYPE_ASN1 or SSL_FILETYPE_PEM.

See:

- wolfSSL_CertManagerLoadCA
- wolfSSL_CertManagerVerify

Return:

- SSL_SUCCESS If successful.
- ASN_SIG_CONFIRM_E will be returned if the signature could not be verified.
- ASN_SIG_OID_E will be returned if the signature type is not supported.
- CRL_CERT_REVOKED is an error that is returned if this certificate has been revoked.
- CRL_MISSING is an error that is returned if a current issuer CRL is not available.
- ASN_BEFORE_DATE_E will be returned if the current date is before the before date.
- ASN_AFTER_DATE_E will be returned if the current date is after the after date.
- SSL_BAD_FILETYPE will be returned if the file is the wrong format.
- SSL_BAD_FILE will be returned if the file doesn’t exist, can’t be read, or is corrupted.
- MEMORY_E will be returned if an out of memory condition occurs.
- ASN_INPUT_E will be returned if Base16 decoding fails on the file.
- BAD_FUNC_ARG is the error that will be returned if a pointer is not provided.

Example

```
#include <wolfssl/ssl.h>

int ret = 0;
int sz = 0;
WOLFSSL_CERT_MANAGER* cm;
byte certBuff[...];
...
```

```
ret = wolfSSL_CertManagerVerifyBuffer(cm, certBuff, sz, SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
    error verifying certificate
}
```

C.52.2.309 function wolfSSL_CertManagerSetVerify

```
void wolfSSL_CertManagerSetVerify(
    WOLFSSL_CERT_MANAGER * cm,
    VerifyCallback vc
)
```

The function sets the verifyCallback function in the Certificate Manager. If present, it will be called for each cert loaded. If there is a verification error, the verify callback can be used to over-ride the error.

Parameters:

- **cm** a pointer to a WOLFSSL_CERT_MANAGER structure, created using wolfSSL_CertManagerNew().
- **vc** a VerifyCallback function pointer to the callback routine

See: wolfSSL_CertManagerVerify

Return: none No return.

Example

```
#include <wolfssl/ssl.h>

int myVerify(int preverify, WOLFSSL_X509_STORE_CTX* store)
{ // do custom verification of certificate }

WOLFSSL_CTX* ctx = WOLFSSL_CTX_new(Protocol define);
WOLFSSL_CERT_MANAGER* cm = wolfSSL_CertManagerNew();
...
wolfSSL_CertManagerSetVerify(cm, myVerify);
```

C.52.2.310 function wolfSSL_CertManagerCheckCRL

```
int wolfSSL_CertManagerCheckCRL(
    WOLFSSL_CERT_MANAGER * cm,
    unsigned char * der,
    int sz
)
```

Check CRL if the option is enabled and compares the cert to the CRL list.

Parameters:

- **cm** a pointer to a WOLFSSL_CERT_MANAGER struct.
- **der** pointer to a DER formatted certificate.
- **sz** size of the certificate.

See:

- CheckCertCRL
- ParseCertRelative
- wolfSSL_CertManagerSetCRL_CB
- InitDecodedCert

Return:

- SSL_SUCCESS returns if the function returned as expected. If the crlEnabled member of the WOLFSSL_CERT_MANAGER struct is turned on.
- MEMORY_E returns if the allocated memory failed.
- BAD_FUNC_ARG if the WOLFSSL_CERT_MANAGER is NULL.

Example

```
WOLFSSL_CERT_MANAGER* cm;
byte* der;
int sz; // size of der
...
if(wolfSSL_CertManagerCheckCRL(cm, der, sz) != SSL_SUCCESS){
    // Error returned. Deal with failure case.
}
```

C.52.2.311 function wolfSSL_CertManagerEnableCRL

```
int wolfSSL_CertManagerEnableCRL(
    WOLFSSL_CERT_MANAGER * cm,
    int options
)
```

Turns on Certificate Revocation List checking when verifying certificates with the Certificate Manager. By default, CRL checking is off. options include WOLFSSL_CRL_CHECKALL which performs CRL checking on each certificate in the chain versus the Leaf certificate only which is the default.

Parameters:

- **cm** a pointer to a WOLFSSL_CERT_MANAGER structure, created using wolfSSL_CertManagerNew().
- **options** options to use when enabling the Certification Manager, cm.

See: wolfSSL_CertManagerDisableCRL

Return:

- SSL_SUCCESS If successful the call will return.
- NOT_COMPILED_IN will be returned if wolfSSL was not built with CRL enabled.
- MEMORY_E will be returned if an out of memory condition occurs.
- BAD_FUNC_ARG is the error that will be returned if a pointer is not provided.
- SSL_FAILURE will be returned if the CRL context cannot be initialized properly.

Example

```
#include <wolfssl/ssl.h>

int ret = 0;
WOLFSSL_CERT_MANAGER* cm;
...

ret = wolfSSL_CertManagerEnableCRL(cm, 0);
if (ret != SSL_SUCCESS) {
    error enabling cert manager
}

...
```

C.52.2.312 function wolfSSL_CertManagerDisableCRL

```
int wolfSSL_CertManagerDisableCRL(
    WOLFSSL_CERT_MANAGER *
)
```

Turns off Certificate Revocation List checking when verifying certificates with the Certificate Manager. By default, CRL checking is off. You can use this function to temporarily or permanently disable CRL checking with this Certificate Manager context that previously had CRL checking enabled.

Parameters:

- **cm** a pointer to a WOLFSSL_CERT_MANAGER structure, created using wolfSSL_CertManagerNew().

See: wolfSSL_CertManagerEnableCRL

Return:

- SSL_SUCCESS If successful the call will return.
- BAD_FUNC_ARG is the error that will be returned if a function pointer is not provided.

Example

```
#include <wolfssl/ssl.h>

int ret = 0;
WOLFSSL_CERT_MANAGER* cm;
...
ret = wolfSSL_CertManagerDisableCRL(cm);
if (ret != SSL_SUCCESS) {
    error disabling cert manager
}
...
```

C.52.2.313 function wolfSSL_CertManagerLoadCRL

```
int wolfSSL_CertManagerLoadCRL(
    WOLFSSL_CERT_MANAGER * cm,
    const char * path,
    int type,
    int monitor
)
```

Error checks and passes through to LoadCRL() in order to load the cert into the CRL for revocation checking. An updated CRL can be loaded by first calling wolfSSL_CertManagerFreeCRL, then loading the new CRL.

Parameters:

- **cm** a pointer to a WOLFSSL_CERT_MANAGER structure, created using wolfSSL_CertManagerNew().
- **path** a constant char pointer holding the CRL path.
- **type** type of certificate to be loaded.
- **monitor** requests monitoring in LoadCRL().

See:

- wolfSSL_CertManagerEnableCRL
- wolfSSL_LoadCRL
- wolfSSL_CertManagerFreeCRL

Return:

- SSL_SUCCESS if there is no error in wolfSSL_CertManagerLoadCRL and if LoadCRL returns successfully.

- BAD_FUNC_ARG if the WOLFSSL_CERT_MANAGER struct is NULL.
- SSL_FATAL_ERROR if wolfSSL_CertManagerEnableCRL returns anything other than SSL_SUCCESS.
- BAD_PATH_ERROR if the path is NULL.
- MEMORY_E if LoadCRL fails to allocate heap memory.

Example

```
#include <wolfssl/ssl.h>

int wolfSSL_LoadCRL(WOLFSSL* ssl, const char* path, int type,
int monitor);
…
wolfSSL_CertManagerLoadCRL(SSL_CM(ssl), path, type, monitor);
```

C.52.2.314 function wolfSSL_CertManagerLoadCRLBuffer

```
int wolfSSL_CertManagerLoadCRLBuffer(
    WOLFSSL_CERT_MANAGER * cm,
    const unsigned char * buff,
    long sz,
    int type
)
```

The function loads the CRL file by calling BufferLoadCRL.

Parameters:

- **cm** a pointer to a WOLFSSL_CERT_MANAGER structure.
- **buff** a constant byte type and is the buffer.
- **sz** a long int representing the size of the buffer.
- **type** a long integer that holds the certificate type.

See:

- BufferLoadCRL
- wolfSSL_CertManagerEnableCRL

Return:

- SSL_SUCCESS returned if the function completed without errors.
- BAD_FUNC_ARG returned if the WOLFSSL_CERT_MANAGER is NULL.
- SSL_FATAL_ERROR returned if there is an error associated with the WOLFSSL_CERT_MANAGER.

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_CERT_MANAGER* cm;
const unsigned char* buff;
long sz; size of buffer
int type;  cert type
...
int ret = wolfSSL_CertManagerLoadCRLBuffer(cm, buff, sz, type);
if(ret == SSL_SUCCESS){
return ret;
} else {
    Failure case.
}
```

C.52.2.315 function wolfSSL_CertManagerSetCRL_Cb

```
int wolfSSL_CertManagerSetCRL_Cb(
    WOLFSSL_CERT_MANAGER * cm,
    CbMissingCRL cb
)
```

This function sets the CRL Certificate Manager callback. If HAVE_CRL is defined and a matching CRL record is not found then the cbMissingCRL is called (set via wolfSSL_CertManagerSetCRL_Cb). This allows you to externally retrieve the CRL and load it.

Parameters:

- **cm** the WOLFSSL_CERT_MANAGER structure holding the information for the certificate.
- **cb** a function pointer to (*CbMissingCRL) that is set to the cbMissingCRL member of the WOLFSSL_CERT_MANAGER.

See:

- CbMissingCRL
- wolfSSL_SetCRL_Cb

Return:

- SSL_SUCCESS returned upon successful execution of the function and subroutines.
- BAD_FUNC_ARG returned if the WOLFSSL_CERT_MANAGER structure is NULL.

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_CTX* ctx = wolfSSL_CTX_new(protocol method);
WOLFSSL* ssl = wolfSSL_new(ctx);
…
void cb(const char* url){
    Function body.
}
…
CbMissingCRL cb = CbMissingCRL;
…
if(ctx){
    return wolfSSL_CertManagerSetCRL_Cb(SSL_CM(ssl), cb);
}
```

C.52.2.316 function wolfSSL_CertManagerSetCRLUpdate_Cb

```
int wolfSSL_CertManagerSetCRLUpdate_Cb(
    WOLFSSL_CERT_MANAGER * cm,
    CbUpdateCRL cb
)
```

This function sets the CRL Update callback. If HAVE_CRL and HAVE_CRL_UPDATE_CB is defined , and an entry with the same issuer and a lower CRL number exists when a CRL is added, then the CbUpdateCRL is called with the details of the existing entry and the new one replacing it.

Parameters:

- **cm** the WOLFSSL_CERT_MANAGER structure holding the information for the certificate.
- **cb** a function pointer to (*CbUpdateCRL) that is set to the cbUpdateCRL member of the WOLFSSL_CERT_MANAGER. Signature requirement: void (*CbUpdateCRL)(CrlInfo *old, CrlInfo* new);

See: CbUpdateCRL

Return:

- SSL_SUCCESS returned upon successful execution of the function and subroutines.
- BAD_FUNC_ARG returned if the WOLFSSL_CERT_MANAGER structure is NULL.

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_CTX* ctx = wolfSSL_CTX_new(protocol method);
WOLFSSL* ssl = wolfSSL_new(ctx);
…
void cb(CrlInfo *old, CrlInfo *new){
    Function body.
}
…
CbUpdateCRL cb = CbUpdateCRL;
…
if(ctx){
    return wolfSSL_CertManagerSetCRLUpdate_Cb(SSL_CM(ssl), cb);
}
```

C.52.2.317 function wolfSSL_CertManagerGetCRLInfo

```
int wolfSSL_CertManagerGetCRLInfo(
    WOLFSSL_CERT_MANAGER * cm,
    CrlInfo * info,
    const byte * buff,
    long sz,
    int type
)
```

This function yields a structure with parsed CRL information from an encoded CRL buffer.

Parameters:

- **cm** the WOLFSSL_CERT_MANAGER structure..
- **info** pointer to caller managed CrlInfo structure that will receive the CRL information.
- **buff** input buffer containing encoded CRL.
- **sz** the length in bytes of the input CRL data in buff.
- **type** WOLFSSL_FILETYPE_PEM or WOLFSSL_FILETYPE_DER
- **cm** a pointer to a WOLFSSL_CERT_MANAGER structure, created using wolfSSL_CertManagerNew().

See:

- CbUpdateCRL
- wolfSSL_SetCRL_Cb
- wolfSSL_CertManagerLoadCRL

Return:

- SSL_SUCCESS returned upon successful execution of the function and subroutines.
- BAD_FUNC_ARG returned if the WOLFSSL_CERT_MANAGER structure is NULL.
- SSL_SUCCESS returned upon successful execution of the function and subroutines.
- BAD_FUNC_ARG returned if the WOLFSSL_CERT_MANAGER structure is NULL.

Example

```
#include <wolfssl/ssl.h>

CrlInfo info;
WOLFSSL_CERT_MANAGER* cm = NULL;

cm = wolfSSL_CertManagerNew();

// Read crl data from file into buffer

wolfSSL_CertManagerGetCRLInfo(cm, &info, crlData, crlDataLen,
                              WOLFSSL_FILETYPE_PEM);
```

This function frees the CRL stored in the Cert Manager. An application can update the CRL by calling wolfSSL_CertManagerFreeCRL and then loading the new CRL.

Example

```
#include <wolfssl/ssl.h>

const char* crl1     = "./certs/crl/crl.pem";
WOLFSSL_CERT_MANAGER* cm = NULL;

cm = wolfSSL_CertManagerNew();
wolfSSL_CertManagerLoadCRL(cm, crl1, WOLFSSL_FILETYPE_PEM, 0);
…
wolfSSL_CertManagerFreeCRL(cm);
```

C.52.2.318 function wolfSSL_CertManagerCheckOCSP

```
int wolfSSL_CertManagerCheckOCSP(
    WOLFSSL_CERT_MANAGER * cm,
    unsigned char * der,
    int sz
)
```

The function enables the WOLFSSL_CERT_MANAGER's member, ocspEnabled to signify that the OCSP check option is enabled.

Parameters:

- **cm** a pointer to a WOLFSSL_CERT_MANAGER structure, created using wolfSSL_CertManagerNew().
- **der** a byte pointer to the certificate.
- **sz** an int type representing the size of the DER cert.

See:

- ParseCertRelative
- CheckCertOCSP

Return:

- SSL_SUCCESS returned on successful execution of the function. The ocspEnabled member of the WOLFSSL_CERT_MANAGER is enabled.
- BAD_FUNC_ARG returned if the WOLFSSL_CERT_MANAGER structure is NULL or if an argument value that is not allowed is passed to a subroutine.
- MEMORY_E returned if there is an error allocating memory within this function or a subroutine.

Example

```
#import <wolfssl/ssl.h>

WOLFSSL* ssl = wolfSSL_new(ctx);
byte* der;
int sz; size of der
...
if(wolfSSL_CertManagerCheckOCSP(cm, der, sz) != SSL_SUCCESS){
 Failure case.
}
```

C.52.2.319 function wolfSSL_CertManagerEnableOCSP

```
int wolfSSL_CertManagerEnableOCSP(
    WOLFSSL_CERT_MANAGER * cm,
    int options
)
```

Turns on OCSP if it's turned off and if compiled with the set option available.

Parameters:

- **cm** a pointer to a WOLFSSL_CERT_MANAGER structure, created using wolfSSL_CertManagerNew().
- **options** used to set values in WOLFSSL_CERT_MANAGER struct.

See: wolfSSL_CertManagerNew

Return:

- SSL_SUCCESS returned if the function call is successful.
- BAD_FUNC_ARG if cm struct is NULL.
- MEMORY_E if WOLFSSL_OCSP struct value is NULL.
- SSL_FAILURE initialization of WOLFSSL_OCSP struct fails to initialize.
- NOT_COMPILED_IN build not compiled with correct feature enabled.

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_CTX* ctx = wolfSSL_CTX_new(protocol method);
WOLFSSL* ssl = wolfSSL_new(ctx);
WOLFSSL_CERT_MANAGER* cm = wolfSSL_CertManagerNew();
int options;
…
if(wolfSSL_CertManagerEnableOCSP(SSL_CM(ssl), options) != SSL_SUCCESS){
    Failure case.
}
```

C.52.2.320 function wolfSSL_CertManagerDisableOCSP

```
int wolfSSL_CertManagerDisableOCSP(
    WOLFSSL_CERT_MANAGER *
)
```

Disables OCSP certificate revocation.

Parameters:

- **ssl** - a pointer to a WOLFSSL structure, created using wolfSSL_new().

See: wolfSSL_DisableCRL

Return:

- SSL_SUCCESS wolfSSL_CertMangerDisableCRL successfully disabled the crlEnabled member of the WOLFSSL_CERT_MANAGER structure.
- BAD_FUNC_ARG the WOLFSSL structure was NULL.

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_CTX* ctx = wolfSSL_CTX_new(method);
WOLFSSL* ssl = wolfSSL_new(ctx);
...
if(wolfSSL_CertManagerDisableOCSP(ssl) != SSL_SUCCESS){
    Fail case.
}
```

C.52.2.321 function wolfSSL_CertManagerSetOCSPOverrideURL

```
int wolfSSL_CertManagerSetOCSPOverrideURL(
    WOLFSSL_CERT_MANAGER * cm,
    const char * url
)
```

The function copies the url to the ocspOverrideURL member of the WOLFSSL_CERT_MANAGER structure.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- ocspOverrideURL
- wolfSSL_SetOCSP_OverrideURL

Return:

- SSL_SUCCESS the function was able to execute as expected.
- BAD_FUNC_ARG the WOLFSSL_CERT_MANAGER struct is NULL.
- MEMEORY_E Memory was not able to be allocated for the ocspOverrideURL member of the certificate manager.

Example

```
#include <wolfssl/ssl.h>
WOLFSSL_CERT_MANAGER* cm = wolfSSL_CertManagerNew();
const char* url;
…
int wolfSSL_SetOCSP_OverrideURL(WOLFSSL* ssl, const char* url)
…
if(wolfSSL_CertManagerSetOCSPOverrideURL(SSL_CM(ssl), url) != SSL_SUCCESS){
    Failure case.
}
```

C.52.2.322 function wolfSSL_CertManagerSetOCSP_Cb

```
int wolfSSL_CertManagerSetOCSP_Cb(
    WOLFSSL_CERT_MANAGER * cm,
```

```
    CbOCSPIO ioCb,
    CbOCSPRespFree respFreeCb,
    void * ioCbCtx
)
```

The function sets the OCSP callback in the WOLFSSL_CERT_MANAGER.

Parameters:

- **cm** a pointer to a WOLFSSL_CERT_MANAGER structure.
- **ioCb** a function pointer of type CbOCSPIO.
- **respFreeCb** - a function pointer of type CbOCSPRespFree.
- **ioCbCtx** - a void pointer variable to the I/O callback user registered context.

See:

- wolfSSL_CertManagerSetOCSPOverrideURL
- wolfSSL_CertManagerCheckOCSP
- wolfSSL_CertManagerEnableOCSPStapling
- wolfSSL_EnableOCSP
- wolfSSL_DisableOCSP
- wolfSSL_SetOCSP_Cb

Return:

- SSL_SUCCESS returned on successful execution. The arguments are saved in the WOLFSSL_CERT_MANAGER structure.
- BAD_FUNC_ARG returned if the WOLFSSL_CERT_MANAGER is NULL.

Example

```
#include <wolfssl/ssl.h>

wolfSSL_SetOCSP_Cb(WOLFSSL* ssl, CbOCSPIO ioCb,
CbOCSPRespFree respFreeCb, void* ioCbCtx){
…
return wolfSSL_CertManagerSetOCSP_Cb(SSL_CM(ssl), ioCb, respFreeCb, ioCbCtx);
```

C.52.2.323 function wolfSSL_CertManagerEnableOCSPStapling

```
int wolfSSL_CertManagerEnableOCSPStapling(
    WOLFSSL_CERT_MANAGER * cm
)
```

This function turns on OCSP stapling if it is not turned on as well as set the options.

Parameters:

- **cm** a pointer to a WOLFSSL_CERT_MANAGER structure, a member of the WOLFSSL_CTX structure.

See: wolfSSL_CTX_EnableOCSPStapling

Return:

- SSL_SUCCESS returned if there were no errors and the function executed successfully.
- BAD_FUNC_ARG returned if the WOLFSSL_CERT_MANAGER structure is NULL or otherwise if there was a unpermitted argument value passed to a subroutine.
- MEMORY_E returned if there was an issue allocating memory.
- SSL_FAILURE returned if the initialization of the OCSP structure failed.
- NOT_COMPILED_IN returned if wolfSSL was not compiled with HAVE_CERTIFICATE_STATUS_REQUEST option.

Example

```
int wolfSSL_CTX_EnableOCSPStapling(WOLFSSL_CTX* ctx){
…
return wolfSSL_CertManagerEnableOCSPStapling(ctx->cm);
```

C.52.2.324 function wolfSSL_EnableCRL

```
int wolfSSL_EnableCRL(
    WOLFSSL * ssl,
    int options
)
```

Enables CRL certificate revocation.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **options** an integer that is used to determine the setting of crlCheckAll member of the WOLFSSL_CERT_MANAGER structure.

See:

- wolfSSL_CertManagerEnableCRL
- InitCRL

Return:

- SSL_SUCCESS the function and subroutines returned with no errors.
- BAD_FUNC_ARG returned if the WOLFSSL structure is NULL.
- MEMORY_E returned if the allocation of memory failed.
- SSL_FAILURE returned if the InitCRL function does not return successfully.
- NOT_COMPILED_IN HAVE_CRL was not enabled during the compiling.

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
…
if (wolfSSL_EnableCRL(ssl, WOLFSSL_CRL_CHECKALL) != SSL_SUCCESS){
    // Failure case. SSL_SUCCESS was not returned by this function or
a subroutine
}
```

C.52.2.325 function wolfSSL_DisableCRL

```
int wolfSSL_DisableCRL(
    WOLFSSL * ssl
)
```

Disables CRL certificate revocation.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_CertManagerDisableCRL
- wolfSSL_CertManagerDisableOCSP

Return:

- SSL_SUCCESS wolfSSL_CertMangerDisableCRL successfully disabled the crlEnabled member of the WOLFSSL_CERT_MANAGER structure.
- BAD_FUNC_ARG the WOLFSSL structure was NULL.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
...
if(wolfSSL_DisableCRL(ssl) != SSL_SUCCESS){
    // Failure case
}
```

C.52.2.326 function wolfSSL_LoadCRL

```
int wolfSSL_LoadCRL(
    WOLFSSL * ssl,
    const char * path,
    int type,
    int monitor
)
```

A wrapper function that ends up calling LoadCRL to load the certificate for revocation checking.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **path** a constant character pointer that holds the path to the crl file.
- **type** an integer representing the type of certificate.
- **monitor** an integer variable used to verify the monitor path if requested.

See:

- wolfSSL_CertManagerLoadCRL
- wolfSSL_CertManagerEnableCRL
- LoadCRL

Return:

- WOLFSSL_SUCCESS returned if the function and all of the subroutines executed without error.
- SSL_FATAL_ERROR returned if one of the subroutines does not return successfully.
- BAD_FUNC_ARG if the WOLFSSL_CERT_MANAGER or the WOLFSSL structure are NULL.

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
const char* crlPemDir;
…
if(wolfSSL_LoadCRL(ssl, crlPemDir, SSL_FILETYPE_PEM, 0) != SSL_SUCCESS){
    // Failure case. Did not return SSL_SUCCESS.
}
```

C.52.2.327 function wolfSSL_SetCRL_Cb

```
int wolfSSL_SetCRL_Cb(
    WOLFSSL * ssl,
    CbMissingCRL cb
)
```

Sets the CRL callback in the WOLFSSL_CERT_MANAGER structure.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **cb** a function pointer to CbMissingCRL.

See:

- CbMissingCRL
- wolfSSL_CertManagerSetCRL_Cb

Return:

- SSL_SUCCESS returned if the function or subroutine executes without error. The cbMissingCRL member of the WOLFSSL_CERT_MANAGER is set.
- BAD_FUNC_ARG returned if the WOLFSSL or WOLFSSL_CERT_MANAGER structure is NULL.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
WOLFSSL* ssl = wolfSSL_new(ctx);
…
void cb(const char* url) // required signature
{
    // Function body
}
…
int crlCb = wolfSSL_SetCRL_Cb(ssl, cb);
if(crlCb != SSL_SUCCESS){
    // The callback was not set properly
}
```

C.52.2.328 function wolfSSL_EnableOCSP

```
int wolfSSL_EnableOCSP(
    WOLFSSL * ssl,
    int options
)
```

This function enables OCSP certificate verification. The value of options if formed by or'ing one or more of the following options: WOLFSSL_OCSP_URL_OVERRIDE - use the override URL instead of the URL in certificates. The override URL is specified using the wolfSSL_CTX_SetOCSP_OverrideURL() function. WOLFSSL_OCSP_CHECKALL - Set all OCSP checks on WOLFSSL_OCSP_NO_NONCE - Set nonce option for creating OCSP requests.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **options** an integer type passed to wolfSSL_CertMangerENableOCSP() used for settings check.

See: wolfSSL_CertManagerEnableOCSP

Return:

- SSL_SUCCESS returned if the function and subroutines executes without errors.
- BAD_FUNC_ARG returned if an argument in this function or any subroutine receives an invalid argument value.
- MEMORY_E returned if there was an error allocating memory for a structure or other variable.
- NOT_COMPILED_IN returned if wolfSSL was not compiled with the HAVE_OCSP option.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
int options; // initialize to option constant
…
int ret = wolfSSL_EnableOCSP(ssl, options);
if(ret != SSL_SUCCESS){
    // OCSP is not enabled
}
```

C.52.2.329 function wolfSSL_DisableOCSP

```
int wolfSSL_DisableOCSP(
    WOLFSSL *
)
```

Disables the OCSP certificate revocation option.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See: wolfSSL_CertManagerDisableOCSP

Return:

- SSL_SUCCESS returned if the function and its subroutine return with no errors. The ocspEnabled member of the WOLFSSL_CERT_MANAGER structure was successfully set.
- BAD_FUNC_ARG returned if the WOLFSSL structure is NULL.

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
…
if(wolfSSL_DisableOCSP(ssl) != SSL_SUCCESS){
    // Returned with an error. Failure case in this block.
}
```

C.52.2.330 function wolfSSL_SetOCSP_OverrideURL

```
int wolfSSL_SetOCSP_OverrideURL(
    WOLFSSL * ssl,
    const char * url
)
```

This function sets the ocspOverrideURL member in the WOLFSSL_CERT_MANAGER structure.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **url** a constant char pointer to the url that will be stored in the ocspOverrideURL member of the WOLFSSL_CERT_MANAGER structure.

See: wolfSSL_CertManagerSetOCSPOverrideURL

Return:

- SSL_SUCCESS returned on successful execution of the function.
- BAD_FUNC_ARG returned if the WOLFSSL struct is NULL or if a unpermitted argument was passed to a subroutine.
- MEMORY_E returned if there was an error allocating memory in the subroutine.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
char url[URLSZ];
...
if(wolfSSL_SetOCSP_OverrideURL(ssl, url)){
    // The override url is set to the new value
}
```

C.52.2.331 function wolfSSL_SetOCSP_Cb

```
int wolfSSL_SetOCSP_Cb(
    WOLFSSL * ssl,
    CbOCSPIO ioCb,
    CbOCSPRespFree respFreeCb,
    void * ioCbCtx
)
```

This function sets the OCSP callback in the WOLFSSL_CERT_MANAGER structure.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **ioCb** a function pointer to type CbOCSPIO.
- **respFreeCb** a function pointer to type CbOCSPRespFree which is the call to free the response memory.
- **ioCbCtx** a void pointer that will be held in the ocspIOCtx member of the CM.

See:

- wolfSSL_CertManagerSetOCSP_Cb
- CbOCSPIO
- CbOCSPRespFree

Return:

- SSL_SUCCESS returned if the function executes without error. The ocspIOCb, ocspRespFreeCb, and ocspIOCtx members of the CM are set.
- BAD_FUNC_ARG returned if the WOLFSSL or WOLFSSL_CERT_MANAGER structures are NULL.

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
…
int OCSPIO_CB(void* , const char*, int , unsigned char* , int,
unsigned char**){  // must have this signature
// Function Body
}
…
void OCSPRespFree_CB(void* , unsigned char* ){ // must have this signature
    // function body
}
…
void* ioCbCtx;
CbOCSPRespFree CB_OCSPRespFree;

if(wolfSSL_SetOCSP_Cb(ssl, OCSPIO_CB( pass args ), CB_OCSPRespFree,
            ioCbCtx) != SSL_SUCCESS){
    // Callback not set
}
```

C.52.2.332 function wolfSSL_CTX_EnableCRL

```
int wolfSSL_CTX_EnableCRL(
    WOLFSSL_CTX * ctx,
    int options
)
```

Enables CRL certificate verification through the CTX.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_CertManagerEnableCRL
- InitCRL
- wolfSSL_CTX_DisableCRL

Return:

- SSL_SUCCESS returned if this function and it's subroutines execute without errors.
- BAD_FUNC_ARG returned if the CTX struct is NULL or there was otherwise an invalid argument passed in a subroutine.
- MEMORY_E returned if there was an error allocating memory during execution of the function.
- SSL_FAILURE returned if the crl member of the WOLFSSL_CERT_MANAGER fails to initialize correctly.
- NOT_COMPILED_IN wolfSSL was not compiled with the HAVE_CRL option.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
...
if(wolfSSL_CTX_EnableCRL(ssl->ctx, options) != SSL_SUCCESS){
    // The function failed
}
```

C.52.2.333 function wolfSSL_CTX_DisableCRL

```
int wolfSSL_CTX_DisableCRL(
    WOLFSSL_CTX * ctx
)
```

This function disables CRL verification in the CTX structure.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().

See: wolfSSL_CertManagerDisableCRL

Return:

- SSL_SUCCESS returned if the function executes without error. The crlEnabled member of the WOLFSSL_CERT_MANAGER struct is set to 0.
- BAD_FUNC_ARG returned if either the CTX struct or the CM struct has a NULL value.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
...
```

```
if(wolfSSL_CTX_DisableCRL(ssl->ctx) != SSL_SUCCESS){
    // Failure case.
}
```

C.52.2.334 function wolfSSL_CTX_LoadCRL

```
int wolfSSL_CTX_LoadCRL(
    WOLFSSL_CTX * ctx,
    const char * path,
    int type,
    int monitor
)
```

This function loads CRL into the WOLFSSL_CTX structure through wolfSSL_CertManagerLoadCRL().

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().
- **path** the path to the certificate.
- **type** an integer variable holding the type of certificate.
- **monitor** an integer variable used to determine if the monitor path is requested.

See:

- wolfSSL_CertManagerLoadCRL
- LoadCRL

Return:

- SSL_SUCCESS - returned if the function and its subroutines execute without error.
- BAD_FUNC_ARG - returned if this function or any subroutines are passed NULL structures.
- BAD_PATH_ERROR - returned if the path variable opens as NULL.
- MEMORY_E - returned if an allocation of memory failed.

Example

```
WOLFSSL_CTX* ctx;
const char* path;
…
return wolfSSL_CTX_LoadCRL(ctx, path, SSL_FILETYPE_PEM, 0);
```

C.52.2.335 function wolfSSL_CTX_SetCRL_Cb

```
int wolfSSL_CTX_SetCRL_Cb(
    WOLFSSL_CTX * ctx,
    CbMissingCRL cb
)
```

This function will set the callback argument to the cbMissingCRL member of the WOLFSSL_CERT_MANAGER structure by calling wolfSSL_CertManagerSetCRL_Cb.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created with wolfSSL_CTX_new().
- **cb** a pointer to a callback function of type CbMissingCRL. Signature requirement: void (*CbMissingCRL)(const char* url);

See:

- wolfSSL_CertManagerSetCRL_Cb
- CbMissingCRL

Return:

- SSL_SUCCESS returned for a successful execution. The WOLFSSL_CERT_MANAGER structure's member cbMssingCRL was successfully set to cb.
- BAD_FUNC_ARG returned if WOLFSSL_CTX or WOLFSSL_CERT_MANAGER are NULL.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
…
void cb(const char* url) // Required signature
{
    // Function body
}
…
if (wolfSSL_CTX_SetCRL_Cb(ctx, cb) != SSL_SUCCESS){
    // Failure case, cb was not set correctly.
}
```

C.52.2.336 function wolfSSL_CTX_EnableOCSP

```
int wolfSSL_CTX_EnableOCSP(
    WOLFSSL_CTX * ctx,
    int options
)
```

This function sets options to configure behavior of OCSP functionality in wolfSSL. The value of options if formed by or'ing one or more of the following options: WOLFSSL_OCSP_URL_OVERRIDE - use the override URL instead of the URL in certificates. The override URL is specified using the wolfSSL_CTX_SetOCSP_OverrideURL() function. WOLFSSL_OCSP_CHECKALL - Set all OCSP checks on WOLFSSL_OCSP_NO_NONCE - Set nonce option for creating OCSP requests.

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().
- **options** value used to set the OCSP options.

See:

- wolfSSL_CertManagerEnableOCSP
- wolfSSL_EnableOCSP

Return:

- SSL_SUCCESS is returned upon success.
- SSL_FAILURE is returned upon failure.
- NOT_COMPILED_IN is returned when this function has been called, but OCSP support was not enabled when wolfSSL was compiled.

This function only sets the OCSP options when wolfSSL has been compiled with OCSP support (–enable-ocsp, #define HAVE_OCSP).

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
int options; // initialize to option constant
…
int ret = wolfSSL_CTX_EnableOCSP(ctx, options);
if(ret != SSL_SUCCESS){
    // OCSP is not enabled
}
```

C.52.2.337 function wolfSSL_CTX_DisableOCSP

```
int wolfSSL_CTX_DisableOCSP(
    WOLFSSL_CTX *
)
```

This function disables OCSP certificate revocation checking by affecting the ocspEnabled member of the WOLFSSL_CERT_MANAGER structure.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().

See:

- wolfSSL_DisableOCSP
- wolfSSL_CertManagerDisableOCSP

Return:

- SSL_SUCCESS returned if the function executes without error. The ocspEnabled member of the CM has been disabled.
- BAD_FUNC_ARG returned if the WOLFSSL_CTX structure is NULL.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
...
if(!wolfSSL_CTX_DisableOCSP(ssl->ctx)){
    // OCSP is not disabled
}
```

C.52.2.338 function wolfSSL_CTX_SetOCSP_OverrideURL

```
int wolfSSL_CTX_SetOCSP_OverrideURL(
    WOLFSSL_CTX * ctx,
    const char * url
)
```

This function manually sets the URL for OCSP to use. By default, OCSP will use the URL found in the individual certificate unless the WOLFSSL_OCSP_URL_OVERRIDE option is set using the wolfSSL_CTX_EnableOCSP.

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().
- **url** pointer to the OCSP URL for wolfSSL to use.

See: wolfSSL_CTX_OCSP_set_options

Return:

- SSL_SUCCESS is returned upon success.
- SSL_FAILURE is returned upon failure.
- NOT_COMPILED_IN is returned when this function has been called, but OCSP support was not enabled when wolfSSL was compiled.

Example

```
WOLFSSL_CTX* ctx = 0;
...
wolfSSL_CTX_OCSP_set_override_url(ctx, “custom-url-here”);
```

C.52.2.339 function wolfSSL_CTX_SetOCSP_Cb

```
int wolfSSL_CTX_SetOCSP_Cb(
    WOLFSSL_CTX * ctx,
    CbOCSPIO ioCb,
    CbOCSPRespFree respFreeCb,
    void * ioCbCtx
)
```

Sets the callback for the OCSP in the WOLFSSL_CTX structure.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **ioCb** a CbOCSPIO type that is a function pointer.
- **respFreeCb** a CbOCSPRespFree type that is a function pointer.
- **ioCbCtx** a void pointer that will be held in the WOLFSSL_CERT_MANAGER.

See:

- wolfSSL_CertManagerSetOCSP_Cb
- CbOCSPIO
- CbOCSPRespFree

Return:

- SSL_SUCCESS returned if the function executed successfully. The ocspIOCb, ocspRespFreeCb, and ocspIOCtx members in the CM were successfully set.
- BAD_FUNC_ARG returned if the WOLFSSL_CTX or WOLFSSL_CERT_MANAGER structure is NULL.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
…
CbOCSPIO ocspIOCb;
CbOCSPRespFree ocspRespFreeCb;
…
void* ioCbCtx;

int isSetOCSP = wolfSSL_CTX_SetOCSP_Cb(ctx, ocspIOCb,
ocspRespFreeCb, ioCbCtx);

if(isSetOCSP != SSL_SUCCESS){
    // The function did not return successfully.
}
```

C.52.2.340 function wolfSSL_CTX_EnableOCSPStapling

```
int wolfSSL_CTX_EnableOCSPStapling(
    WOLFSSL_CTX *
)
```

This function enables OCSP stapling by calling wolfSSL_CertManagerEnableOCSPStapling().

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().

See:

- wolfSSL_CertManagerEnableOCSPStapling
- InitOCSP

Return:

- SSL_SUCCESS returned if there were no errors and the function executed successfully.
- BAD_FUNC_ARG returned if the WOLFSSL_CTX structure is NULL or otherwise if there was a unpermitted argument value passed to a subroutine.
- MEMORY_E returned if there was an issue allocating memory.
- SSL_FAILURE returned if the initialization of the OCSP structure failed.
- NOT_COMPILED_IN returned if wolfSSL was not compiled with HAVE_CERTIFICATE_STATUS_REQUEST option.

Example

```
WOLFSSL* ssl = WOLFSSL_new();
ssl->method.version; // set to desired protocol
...
if(!wolfSSL_CTX_EnableOCSPStapling(ssl->ctx)){
    // OCSP stapling is not enabled
}
```

C.52.2.341 function wolfSSL_KeepArrays

```
void wolfSSL_KeepArrays(
    WOLFSSL *
)
```

Normally, at the end of the SSL handshake, wolfSSL frees temporary arrays. Calling this function before the handshake begins will prevent wolfSSL from freeing temporary arrays. Temporary arrays may be needed for things such as wolfSSL_get_keys() or PSK hints. When the user is done with temporary arrays, either wolfSSL_FreeArrays() may be called to free the resources immediately, or alternatively the resources will be freed when the associated SSL object is freed.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See: wolfSSL_FreeArrays

Return: none No return.

Example

```
WOLFSSL* ssl;
...
wolfSSL_KeepArrays(ssl);
```

C.52.2.342 function wolfSSL_FreeArrays

```
void wolfSSL_FreeArrays(
    WOLFSSL *
)
```

Normally, at the end of the SSL handshake, wolfSSL frees temporary arrays. If wolfSSL_KeepArrays() has been called before the handshake, wolfSSL will not free temporary arrays. This function explicitly frees temporary arrays and should be called when the user is done with temporary arrays and does not want to wait for the SSL object to be freed to free these resources.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See: wolfSSL_KeepArrays

Return: none No return.

Example

```
WOLFSSL* ssl;
...
wolfSSL_FreeArrays(ssl);
```

C.52.2.343 function wolfSSL_UseSNI

```
int wolfSSL_UseSNI(
    WOLFSSL * ssl,
    unsigned char type,
    const void * data,
    unsigned short size
)
```

This function enables the use of Server Name Indication in the SSL object passed in the 'ssl' parameter. It means that the SNI extension will be sent on ClientHello by wolfSSL client and wolfSSL server will respond ClientHello + SNI with either ServerHello + blank SNI or alert fatal in case of SNI mismatch.

Parameters:

- **ssl** pointer to a SSL object, created with wolfSSL_new().
- **type** indicates which type of server name is been passed in data. The known types are: enum { WOLFSSL_SNI_HOST_NAME = 0 };
- **data** pointer to the server name data.
- **size** size of the server name data.

See:

- wolfSSL_new
- wolfSSL_CTX_UseSNI

Return:

- WOLFSSL_SUCCESS upon success.
- BAD_FUNC_ARG is the error that will be returned in one of these cases: ssl is NULL, data is NULL, type is a unknown value. (see below)
- MEMORY_E is the error returned when there is not enough memory.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx = 0;
WOLFSSL* ssl = 0;
ctx = wolfSSL_CTX_new(method);
if (ctx == NULL) {
    // context creation failed
}
ssl = wolfSSL_new(ctx);
if (ssl == NULL) {
    // ssl creation failed
}
ret = wolfSSL_UseSNI(ssl, WOLFSSL_SNI_HOST_NAME, "www.yassl.com",
    strlen("www.yassl.com"));
if (ret != WOLFSSL_SUCCESS) {
```

```
    // sni usage failed
}
```

C.52.2.344 function wolfSSL_CTX_UseSNI

```
int wolfSSL_CTX_UseSNI(
    WOLFSSL_CTX * ctx,
    unsigned char type,
    const void * data,
    unsigned short size
)
```

This function enables the use of Server Name Indication for SSL objects created from the SSL context passed in the 'ctx' parameter. It means that the SNI extension will be sent on ClientHello by wolfSSL clients and wolfSSL servers will respond ClientHello + SNI with either ServerHello + blank SNI or alert fatal in case of SNI mismatch.

Parameters:

- **ctx** pointer to a SSL context, created with wolfSSL_CTX_new().
- **type** indicates which type of server name is been passed in data. The known types are: enum { WOLFSSL_SNI_HOST_NAME = 0 };
- **data** pointer to the server name data.
- **size** size of the server name data.

See:

- wolfSSL_CTX_new
- wolfSSL_UseSNI

Return:

- WOLFSSL_SUCCESS upon success.
- BAD_FUNC_ARG is the error that will be returned in one of these cases: ctx is NULL, data is NULL, type is a unknown value. (see below)
- MEMORY_E is the error returned when there is not enough memory.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx = 0;
ctx = wolfSSL_CTX_new(method);
if (ctx == NULL) {
    // context creation failed
}
ret = wolfSSL_CTX_UseSNI(ctx, WOLFSSL_SNI_HOST_NAME, "www.yassl.com",
    strlen("www.yassl.com"));
if (ret != WOLFSSL_SUCCESS) {
    // sni usage failed
}
```

C.52.2.345 function wolfSSL_SNI_SetOptions

```
void wolfSSL_SNI_SetOptions(
    WOLFSSL * ssl,
    unsigned char type,
    unsigned char options
)
```

This function is called on the server side to configure the behavior of the SSL session using Server Name Indication in the SSL object passed in the 'ssl' parameter. The options are explained below.

Parameters:

- **ssl** pointer to a SSL object, created with wolfSSL_new().
- **type** indicates which type of server name is been passed in data. The known types are: enum { WOLFSSL_SNI_HOST_NAME = 0 };
- **options** a bitwise semaphore with the chosen options. The available options are: enum { WOLFSSL_SNI_CONTINUE_ON_MISMATCH = 0x01, WOLFSSL_SNI_ANSWER_ON_MISMATCH = 0x02 }; Normally the server will abort the handshake by sending a fatal-level unrecognized_name(112) alert if the hostname provided by the client mismatch with the servers.
- **WOLFSSL_SNI_CONTINUE_ON_MISMATCH** With this option set, the server will not send a SNI response instead of aborting the session.
- **WOLFSSL_SNI_ANSWER_ON_MISMATCH** - With this option set, the server will send a SNI response as if the host names match instead of aborting the session.

See:

- wolfSSL_new
- wolfSSL_UseSNI
- wolfSSL_CTX_SNI_SetOptions

Return: none No returns.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx = 0;
WOLFSSL* ssl = 0;
ctx = wolfSSL_CTX_new(method);
if (ctx == NULL) {
    // context creation failed
}
ssl = wolfSSL_new(ctx);
if (ssl == NULL) {
    // ssl creation failed
}
ret = wolfSSL_UseSNI(ssl, 0, "www.yassl.com", strlen("www.yassl.com"));
if (ret != WOLFSSL_SUCCESS) {
    // sni usage failed
}
wolfSSL_SNI_SetOptions(ssl, WOLFSSL_SNI_HOST_NAME,
    WOLFSSL_SNI_CONTINUE_ON_MISMATCH);
```

C.52.2.346 function wolfSSL_CTX_SNI_SetOptions

```
void wolfSSL_CTX_SNI_SetOptions(
    WOLFSSL_CTX * ctx,
    unsigned char type,
    unsigned char options
)
```

This function is called on the server side to configure the behavior of the SSL sessions using Server Name Indication for SSL objects created from the SSL context passed in the 'ctx' parameter. The options are explained below.

Parameters:

- **ctx** pointer to a SSL context, created with wolfSSL_CTX_new().
- **type** indicates which type of server name is been passed in data. The known types are: enum { WOLFSSL_SNI_HOST_NAME = 0 };
- **options** a bitwise semaphore with the chosen options. The available options are: enum { WOLFSSL_SNI_CONTINUE_ON_MISMATCH = 0x01, WOLFSSL_SNI_ANSWER_ON_MISMATCH = 0x02 }; Normally the server will abort the handshake by sending a fatal-level unrecognized_name(112) alert if the hostname provided by the client mismatch with the servers.
- **WOLFSSL_SNI_CONTINUE_ON_MISMATCH** With this option set, the server will not send a SNI response instead of aborting the session.
- **WOLFSSL_SNI_ANSWER_ON_MISMATCH** With this option set, the server will send a SNI response as if the host names match instead of aborting the session.

See:

- wolfSSL_CTX_new
- wolfSSL_CTX_UseSNI
- wolfSSL_SNI_SetOptions

Return: none No returns.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx = 0;
ctx = wolfSSL_CTX_new(method);
if (ctx == NULL) {
   // context creation failed
}
ret = wolfSSL_CTX_UseSNI(ctx, 0, "www.yassl.com", strlen("www.yassl.com"));
if (ret != WOLFSSL_SUCCESS) {
    // sni usage failed
}
wolfSSL_CTX_SNI_SetOptions(ctx, WOLFSSL_SNI_HOST_NAME,
WOLFSSL_SNI_CONTINUE_ON_MISMATCH);
```

C.52.2.347 function wolfSSL_SNI_GetFromBuffer

```
int wolfSSL_SNI_GetFromBuffer(
    const unsigned char * clientHello,
    unsigned int helloSz,
    unsigned char type,
    unsigned char * sni,
    unsigned int * inOutSz
)
```

This function is called on the server side to retrieve the Server Name Indication provided by the client from the Client Hello message sent by the client to start a session. It does not requires context or session setup to retrieve the SNI.

Parameters:

- **buffer** pointer to the data provided by the client (Client Hello).
- **bufferSz** size of the Client Hello message.
- **type** indicates which type of server name is been retrieved from the buffer. The known types are: enum { WOLFSSL_SNI_HOST_NAME = 0 };
- **sni** pointer to where the output is going to be stored.
- **inOutSz** pointer to the output size, this value will be updated to MIN("SNI's length", inOutSz).

See:

- wolfSSL_UseSNI
- wolfSSL_CTX_UseSNI
- wolfSSL_SNI_GetRequest

Return:

- WOLFSSL_SUCCESS upon success.
- BAD_FUNC_ARG is the error that will be returned in one of this cases: buffer is NULL, bufferSz <= 0, sni is NULL, inOutSz is NULL or <= 0
- BUFFER_ERROR is the error returned when there is a malformed Client Hello message.
- INCOMPLETE_DATA is the error returned when there is not enough data to complete the extraction.

Example

```
unsigned char buffer[1024] = {0};
unsigned char result[32]   = {0};
int           length       = 32;
// read Client Hello to buffer...
ret = wolfSSL_SNI_GetFromBuffer(buffer, sizeof(buffer), 0, result, &length));
if (ret != WOLFSSL_SUCCESS) {
    // sni retrieve failed
}
```

C.52.2.348 function wolfSSL_SNI_Status

```
unsigned char wolfSSL_SNI_Status(
    WOLFSSL * ssl,
    unsigned char type
)
```

This function gets the status of an SNI object.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **type** the SNI type.

See:

- TLSX_SNI_Status
- TLSX_SNI_find
- TLSX_Find

Return:

- value This function returns the byte value of the SNI struct's status member if the SNI is not NULL.
- 0 if the SNI object is NULL.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
WOLFSSL* ssl = wolfSSL_new(ctx);
…
#define AssertIntEQ(x, y) AssertInt(x, y, ==, !=)
…
Byte type = WOLFSSL_SNI_HOST_NAME;
char* request = (char*)&type;
AssertIntEQ(WOLFSSL_SNI_NO_MATCH, wolfSSL_SNI_Status(ssl, type));
…
```

C.52.2.349 function wolfSSL_SNI_GetRequest

```
unsigned short wolfSSL_SNI_GetRequest(
    WOLFSSL * ssl,
    unsigned char type,
    void ** data
)
```

This function is called on the server side to retrieve the Server Name Indication provided by the client in a SSL session.

Parameters:

- **ssl** pointer to a SSL object, created with wolfSSL_new().
- **type** indicates which type of server name is been retrieved in data. The known types are: enum { WOLFSSL_SNI_HOST_NAME = 0 };
- **data** pointer to the data provided by the client.

See:

- wolfSSL_UseSNI
- wolfSSL_CTX_UseSNI

Return: size the size of the provided SNI data.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx = 0;
WOLFSSL* ssl = 0;
ctx = wolfSSL_CTX_new(method);
if (ctx == NULL) {
    // context creation failed
}
ssl = wolfSSL_new(ctx);
if (ssl == NULL) {
    // ssl creation failed
}
ret = wolfSSL_UseSNI(ssl, 0, "www.yassl.com", strlen("www.yassl.com"));
if (ret != WOLFSSL_SUCCESS) {
    // sni usage failed
}
if (wolfSSL_accept(ssl) == SSL_SUCCESS) {
    void *data = NULL;
    unsigned short size = wolfSSL_SNI_GetRequest(ssl, 0, &data);
}
```

C.52.2.350 function wolfSSL_UseALPN

```
int wolfSSL_UseALPN(
    WOLFSSL * ssl,
    char * protocol_name_list,
    unsigned int protocol_name_listSz,
    unsigned char options
)
```

Setup ALPN use for a wolfSSL session.

Parameters:

- **ssl** The wolfSSL session to use.
- **protocol_name_list** List of protocol names to use. Comma delimited string is required.
- **protocol_name_listSz** Size of the list of protocol names.
- **options** WOLFSSL_ALPN_CONTINUE_ON_MISMATCH or WOLFSSL_ALPN_FAILED_ON_MISMATCH.

See: TLSX_UseALPN

Return:

- WOLFSSL_SUCCESS: upon success.
- BAD_FUNC_ARG Returned if ssl or protocol_name_list is null or protocol_name_listSz is too large or options contain something not supported.
- MEMORY_ERROR Error allocating memory for protocol list.
- SSL_FAILURE upon failure.

Example

```
wolfSSL_Init();
WOLFSSL_CTX* ctx;
WOLFSSL* ssl;
WOLFSSL_METHOD method = // Some wolfSSL method
ctx = wolfSSL_CTX_new(method);
ssl = wolfSSL_new(ctx);

char alpn_list[] = {};

if (wolfSSL_UseALPN(ssl, alpn_list, sizeof(alpn_list),
    WOLFSSL_APN_FAILED_ON_MISMATCH) != WOLFSSL_SUCCESS)
{
   // Error setting session ticket
}
```

C.52.2.351 function wolfSSL_ALPN_GetProtocol

```
int wolfSSL_ALPN_GetProtocol(
    WOLFSSL * ssl,
    char ** protocol_name,
    unsigned short * size
)
```

This function gets the protocol name set by the server.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **protocol_name** a pointer to a char that represents the protocol name and will be held in the ALPN structure.
- **size** a word16 type that represents the size of the protocol_name.

See:

- TLSX_ALPN_GetRequest
- TLSX_Find

Return:

- SSL_SUCCESS returned on successful execution where no errors were thrown.
- SSL_FATAL_ERROR returned if the extension was not found or if there was no protocol match with peer. There will also be an error thrown if there is more than one protocol name accepted.
- SSL_ALPN_NOT_FOUND returned signifying that no protocol match with peer was found.

- BAD_FUNC_ARG returned if there was a NULL argument passed into the function.

Example

```
WOLFSSL_CTX* ctx = WOLFSSL_CTX_new( protocol method );
WOLFSSL* ssl = WOLFSSL_new(ctx);
...
int err;
char* protocol_name = NULL;
Word16 protocol_nameSz = 0;
err = wolfSSL_ALPN_GetProtocol(ssl, &protocol_name, &protocol_nameSz);

if(err == SSL_SUCCESS){
    // Sent ALPN protocol
}
```

C.52.2.352 function wolfSSL_ALPN_GetPeerProtocol

```
int wolfSSL_ALPN_GetPeerProtocol(
    WOLFSSL * ssl,
    char ** list,
    unsigned short * listSz
)
```

This function copies the alpn_client_list data from the SSL object to the buffer.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **list** a pointer to the buffer. The data from the SSL object will be copied into it.
- **listSz** the buffer size.

See: wolfSSL_UseALPN

Return:

- SSL_SUCCESS returned if the function executed without error. The alpn_client_list member of the SSL object has been copied to the list parameter.
- BAD_FUNC_ARG returned if the list or listSz parameter is NULL.
- BUFFER_ERROR returned if there will be a problem with the list buffer (either it's NULL or the size is 0).
- MEMORY_ERROR returned if there was a problem dynamically allocating memory.

Example

```
#import <wolfssl/ssl.h>

WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method);
WOLFSSL* ssl = wolfSSL_new(ctx);
…
#ifdef HAVE_ALPN
char* list = NULL;
word16 listSz = 0;
…
err = wolfSSL_ALPN_GetPeerProtocol(ssl, &list, &listSz);

if(err == SSL_SUCCESS){
    List of protocols names sent by client
}
```

C.52.2.353 function wolfSSL_UseMaxFragment

```
int wolfSSL_UseMaxFragment(
    WOLFSSL * ssl,
    unsigned char mfl
)
```

This function is called on the client side to enable the use of Maximum Fragment Length in the SSL object passed in the 'ssl' parameter. It means that the Maximum Fragment Length extension will be sent on ClientHello by wolfSSL clients.

Parameters:

- **ssl** pointer to a SSL object, created with wolfSSL_new().
- **mfl** indicates which is the Maximum Fragment Length requested for the session. The available options are: enum { WOLFSSL_MFL_2_9 = 1, 512 bytes WOLFSSL_MFL_2_10 = 2, 1024 bytes WOLFSSL_MFL_2_11 = 3, 2048 bytes WOLFSSL_MFL_2_12 = 4, 4096 bytes WOLFSSL_MFL_2_13 = 5, 8192 bytes wolfSSL ONLY!!! };

See:

- wolfSSL_new
- wolfSSL_CTX_UseMaxFragment

Return:

- SSL_SUCCESS upon success.
- BAD_FUNC_ARG is the error that will be returned in one of these cases: ssl is NULL, mfl is out of range.
- MEMORY_E is the error returned when there is not enough memory.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx = 0;
WOLFSSL* ssl = 0;
ctx = wolfSSL_CTX_new(method);
if (ctx == NULL) {
    // context creation failed
}
ssl = wolfSSL_new(ctx);
if (ssl == NULL) {
    // ssl creation failed
}
ret = wolfSSL_UseMaxFragment(ssl, WOLFSSL_MFL_2_11);
if (ret != 0) {
    // max fragment usage failed
}
```

C.52.2.354 function wolfSSL_CTX_UseMaxFragment

```
int wolfSSL_CTX_UseMaxFragment(
    WOLFSSL_CTX * ctx,
    unsigned char mfl
)
```

This function is called on the client side to enable the use of Maximum Fragment Length for SSL objects created from the SSL context passed in the 'ctx' parameter. It means that the Maximum Fragment Length extension will be sent on ClientHello by wolfSSL clients.

Parameters:

- **ctx** pointer to a SSL context, created with wolfSSL_CTX_new().
- **mfl** indicates which is the Maximum Fragment Length requested for the session. The available options are: enum { WOLFSSL_MFL_2_9 = 1 512 bytes, WOLFSSL_MFL_2_10 = 2 1024 bytes, WOLFSSL_MFL_2_11 = 3 2048 bytes WOLFSSL_MFL_2_12 = 4 4096 bytes, WOLFSSL_MFL_2_13 = 5 8192 bytes wolfSSL ONLY!!!, WOLFSSL_MFL_2_13 = 6 256 bytes wolfSSL ONLY!!! };

See:

- wolfSSL_CTX_new
- wolfSSL_UseMaxFragment

Return:

- SSL_SUCCESS upon success.
- BAD_FUNC_ARG is the error that will be returned in one of these cases: ctx is NULL, mfl is out of range.
- MEMORY_E is the error returned when there is not enough memory.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx = 0;
ctx = wolfSSL_CTX_new(method);
if (ctx == NULL) {
    // context creation failed
}
ret = wolfSSL_CTX_UseMaxFragment(ctx, WOLFSSL_MFL_2_11);
if (ret != 0) {
    // max fragment usage failed
}
```

C.52.2.355 function wolfSSL_UseTruncatedHMAC

```
int wolfSSL_UseTruncatedHMAC(
    WOLFSSL * ssl
)
```

This function is called on the client side to enable the use of Truncated HMAC in the SSL object passed in the 'ssl' parameter. It means that the Truncated HMAC extension will be sent on ClientHello by wolfSSL clients.

Parameters:

- **ssl** pointer to a SSL object, created with wolfSSL_new()

See:

- wolfSSL_new
- wolfSSL_CTX_UseMaxFragment

Return:

- SSL_SUCCESS upon success.
- BAD_FUNC_ARG is the error that will be returned in one of these cases: ssl is NULL
- MEMORY_E is the error returned when there is not enough memory.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx = 0;
```

```
WOLFSSL* ssl = 0;
ctx = wolfSSL_CTX_new(method);
if (ctx == NULL) {
    // context creation failed
}
ssl = wolfSSL_new(ctx);
if (ssl == NULL) {
    // ssl creation failed
}
ret = wolfSSL_UseTruncatedHMAC(ssl);
if (ret != 0) {
    // truncated HMAC usage failed
}
```

C.52.2.356 function wolfSSL_CTX_UseTruncatedHMAC

```
int wolfSSL_CTX_UseTruncatedHMAC(
    WOLFSSL_CTX * ctx
)
```

This function is called on the client side to enable the use of Truncated HMAC for SSL objects created from the SSL context passed in the 'ctx' parameter. It means that the Truncated HMAC extension will be sent on ClientHello by wolfSSL clients.

Parameters:

- **ctx** pointer to a SSL context, created with wolfSSL_CTX_new().

See:

- wolfSSL_CTX_new
- wolfSSL_UseMaxFragment

Return:

- SSL_SUCCESS upon success.
- BAD_FUNC_ARG is the error that will be returned in one of these cases: ctx is NULL
- MEMORY_E is the error returned when there is not enough memory.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx = 0;
ctx = wolfSSL_CTX_new(method);
if (ctx == NULL) {
    // context creation failed
}
ret = wolfSSL_CTX_UseTruncatedHMAC(ctx);
if (ret != 0) {
    // truncated HMAC usage failed
}
```

C.52.2.357 function wolfSSL_UseOCSPStapling

```
int wolfSSL_UseOCSPStapling(
    WOLFSSL * ssl,
    unsigned char status_type,
    unsigned char options
)
```

Stapling eliminates the need to contact the CA. Stapling lowers the cost of certificate revocation check presented in OCSP.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **status_type** a byte type that is passed through to TLSX_UseCertificateStatusRequest() and stored in the CertificateStatusRequest structure.
- **options** a byte type that is passed through to TLSX_UseCertificateStatusRequest() and stored in the CertificateStatusRequest structure.

See:

- TLSX_UseCertificateStatusRequest
- wolfSSL_CTX_UseOCSPStapling

Return:

- SSL_SUCCESS returned if TLSX_UseCertificateStatusRequest executes without error.
- MEMORY_E returned if there is an error with the allocation of memory.
- BAD_FUNC_ARG returned if there is an argument that has a NULL or otherwise unacceptable value passed into the function.

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
…
if (wolfSSL_UseOCSPStapling(ssl, WOLFSSL_CSR2_OCSP,
WOLFSSL_CSR2_OCSP_USE_NONCE) != SSL_SUCCESS){
    // Failed case.
}
```

C.52.2.358 function wolfSSL_CTX_UseOCSPStapling

```
int wolfSSL_CTX_UseOCSPStapling(
    WOLFSSL_CTX * ctx,
    unsigned char status_type,
    unsigned char options
)
```

This function requests the certificate status during the handshake.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().
- **status_type** a byte type that is passed through to TLSX_UseCertificateStatusRequest() and stored in the CertificateStatusRequest structure.
- **options** a byte type that is passed through to TLSX_UseCertificateStatusRequest() and stored in the CertificateStatusRequest structure.

See:

- wolfSSL_UseOCSPStaplingV2
- wolfSSL_UseOCSPStapling
- TLSX_UseCertificateStatusRequest

Return:

- SSL_SUCCESS returned if the function and subroutines execute without error.
- BAD_FUNC_ARG returned if the WOLFSSL_CTX structure is NULL or otherwise if a unpermitted value is passed to a subroutine.
- MEMORY_E returned if the function or subroutine failed to properly allocate memory.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
byte statusRequest = 0; // Initialize status request
…
switch(statusRequest){
    case WOLFSSL_CSR_OCSP:
        if(wolfSSL_CTX_UseOCSPStapling(ssl->ctx, WOLFSSL_CSR_OCSP,
WOLF_CSR_OCSP_USE_NONCE) != SSL_SUCCESS){
// UseCertificateStatusRequest failed
}
// Continue switch cases
```

C.52.2.359 function wolfSSL_UseOCSPStaplingV2

```
int wolfSSL_UseOCSPStaplingV2(
    WOLFSSL * ssl,
    unsigned char status_type,
    unsigned char options
)
```

The function sets the status type and options for OCSP.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **status_type** a byte type that loads the OCSP status type.
- **options** a byte type that holds the OCSP options, set in wolfSSL_SNI_SetOptions() and wolfSSL_CTX_SNI_SetOptions().

See:

- TLSX_UseCertificatStatusRequestV2
- wolfSSL_SNI_SetOptions
- wolfSSL_CTX_SNI_SetOptions

Return:

- SSL_SUCCESS - returned if the function and subroutines executed without error.
- MEMORY_E - returned if there was an allocation of memory error.
- BAD_FUNC_ARG - returned if a NULL or otherwise unaccepted argument was passed to the function or a subroutine.

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
...
if (wolfSSL_UseOCSPStaplingV2(ssl, WOLFSSL_CSR2_OCSP_MULTI, 0) != SSL_SUCCESS){
    // Did not execute properly. Failure case code block.
}
```

C.52.2.360 function wolfSSL_CTX_UseOCSPStaplingV2

```
int wolfSSL_CTX_UseOCSPStaplingV2(
    WOLFSSL_CTX * ctx,
    unsigned char status_type,
    unsigned char options
)
```

Creates and initializes the certificate status request for OCSP Stapling.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().
- **status_type** a byte type that is located in the CertificatStatusRequest structure and must be either WOLFSSL_CSR2_OCSP or WOLFSSL_CSR2_OCSP_MULTI.
- **options** a byte type that will be held in CertificateStatusRequestItemV2 struct.

See:

- TLSX_UseCertificateStatusRequestV2
- wc_RNG_GenerateBlock
- TLSX_Push

Return:

- SSL_SUCCESS if the function and subroutines executed without error.
- BAD_FUNC_ARG returned if the WOLFSSL_CTX structure is NULL or if the side variable is not client side.
- MEMORY_E returned if the allocation of memory failed.

Example

```
WOLFSSL_CTX* ctx  = wolfSSL_CTX_new( protocol method );
byte status_type;
byte options;
...
if(wolfSSL_CTX_UseOCSPStaplingV2(ctx, status_type, options); != SSL_SUCCESS){
    // Failure case.
}
```

C.52.2.361 function wolfSSL_UseSupportedCurve

```
int wolfSSL_UseSupportedCurve(
    WOLFSSL * ssl,
    word16 name
)
```

This function is called on the client side to enable the use of Supported Elliptic Curves Extension in the SSL object passed in the 'ssl' parameter. It means that the supported curves enabled will be sent on ClientHello by wolfSSL clients. This function can be called more than one time to enable multiple curves.

Parameters:

- **ssl** pointer to a SSL object, created with wolfSSL_new().
- **name** indicates which curve will be supported for the session. The available options are: enum { WOLFSSL_ECC_SECP160R1 = 0x10, WOLFSSL_ECC_SECP192R1 = 0x13, WOLFSSL_ECC_SECP224R1 = 0x15, WOLFSSL_ECC_SECP256R1 = 0x17, WOLFSSL_ECC_SECP384R1 = 0x18, WOLFSSL_ECC_SECP521R1 = 0x19 };

See:

- wolfSSL_CTX_new
- wolfSSL_CTX_UseSupportedCurve

Return:

- SSL_SUCCESS upon success.
- BAD_FUNC_ARG is the error that will be returned in one of these cases: ssl is NULL, name is a unknown value. (see below)

- MEMORY_E is the error returned when there is not enough memory.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx = 0;
WOLFSSL* ssl = 0;
ctx = wolfSSL_CTX_new(method);
if (ctx == NULL) {
    // context creation failed
}
ssl = wolfSSL_new(ctx);
if (ssl == NULL) {
    // ssl creation failed
}
ret = wolfSSL_UseSupportedCurve(ssl, WOLFSSL_ECC_SECP256R1);
if (ret != 0) {
    // Elliptic Curve Extension usage failed
}
```

C.52.2.362 function wolfSSL_CTX_UseSupportedCurve

```
int wolfSSL_CTX_UseSupportedCurve(
    WOLFSSL_CTX * ctx,
    word16 name
)
```

This function is called on the client side to enable the use of Supported Elliptic Curves Extension for SSL objects created from the SSL context passed in the 'ctx' parameter. It means that the supported curves enabled will be sent on ClientHello by wolfSSL clients. This function can be called more than one time to enable multiple curves.

Parameters:

- **ctx** pointer to a SSL context, created with wolfSSL_CTX_new().
- **name** indicates which curve will be supported for the session. The available options are: enum { WOLFSSL_ECC_SECP160R1 = 0x10, WOLFSSL_ECC_SECP192R1 = 0x13, WOLFSSL_ECC_SECP224R1 = 0x15, WOLFSSL_ECC_SECP256R1 = 0x17, WOLFSSL_ECC_SECP384R1 = 0x18, WOLFSSL_ECC_SECP521R1 = 0x19 };

See:

- wolfSSL_CTX_new
- wolfSSL_UseSupportedCurve

Return:

- SSL_SUCCESS upon success.
- BAD_FUNC_ARG is the error that will be returned in one of these cases: ctx is NULL, name is a unknown value. (see below)
- MEMORY_E is the error returned when there is not enough memory.

Example

```
int ret = 0;
WOLFSSL_CTX* ctx = 0;
ctx = wolfSSL_CTX_new(method);
if (ctx == NULL) {
    // context creation failed
}
```

```
ret = wolfSSL_CTX_UseSupportedCurve(ctx, WOLFSSL_ECC_SECP256R1);
if (ret != 0) {
    // Elliptic Curve Extension usage failed
}
```

C.52.2.363 function wolfSSL_UseSecureRenegotiation

```
int wolfSSL_UseSecureRenegotiation(
    WOLFSSL * ssl
)
```

This function forces secure renegotiation for the supplied WOLFSSL structure. This is not recommended.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- TLSX_Find
- TLSX_UseSecureRenegotiation

Return:

- SSL_SUCCESS Successfully set secure renegotiation.
- BAD_FUNC_ARG Returns error if ssl is null.
- MEMORY_E Returns error if unable to allocate memory for secure renegotiation.

Example

```
wolfSSL_Init();
WOLFSSL_CTX* ctx;
WOLFSSL* ssl;
WOLFSSL_METHOD method = // Some wolfSSL method
ctx = wolfSSL_CTX_new(method);
ssl = wolfSSL_new(ctx);

if(wolfSSL_UseSecureRenegotiation(ssl) != SSL_SUCCESS)
{
    // Error setting secure renegotiation
}
```

C.52.2.364 function wolfSSL_Rehandshake

```
int wolfSSL_Rehandshake(
    WOLFSSL * ssl
)
```

This function executes a secure renegotiation handshake; this is user forced as wolfSSL discourages this functionality.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_negotiate
- wc_InitSha512
- wc_InitSha384

- wc_InitSha256
- wc_InitSha
- wc_InitMd5

Return:

- SSL_SUCCESS returned if the function executed without error.
- BAD_FUNC_ARG returned if the WOLFSSL structure was NULL or otherwise if an unacceptable argument was passed in a subroutine.
- SECURE_RENEGOTIATION_E returned if there was an error with renegotiating the handshake.
- SSL_FATAL_ERROR returned if there was an error with the server or client configuration and the renegotiation could not be completed. See wolfSSL_negotiate().

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
...
if(wolfSSL_Rehandshake(ssl) != SSL_SUCCESS){
    // There was an error and the rehandshake is not successful.
}
```

C.52.2.365 function wolfSSL_UseSessionTicket

```
int wolfSSL_UseSessionTicket(
    WOLFSSL * ssl
)
```

Force provided WOLFSSL structure to use session ticket. The constant HAVE_SESSION_TICKET should be defined and the constant NO_WOLFSSL_CLIENT should not be defined to use this function.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See: TLSX_UseSessionTicket

Return:

- SSL_SUCCESS Successfully set use session ticket.
- BAD_FUNC_ARG Returned if ssl is null.
- MEMORY_E Error allocating memory for setting session ticket.

Example

```
wolfSSL_Init();
WOLFSSL_CTX* ctx;
WOLFSSL* ssl;
WOLFSSL_METHOD method = // Some wolfSSL method
ctx = wolfSSL_CTX_new(method);
ssl = wolfSSL_new(ctx);

if(wolfSSL_UseSessionTicket(ssl) != SSL_SUCCESS)
{
    // Error setting session ticket
}
```

C.52.2.366 function wolfSSL_CTX_UseSessionTicket

```
int wolfSSL_CTX_UseSessionTicket(
    WOLFSSL_CTX * ctx
)
```

This function sets wolfSSL context to use a session ticket.

Parameters:

- **ctx** The WOLFSSL_CTX structure to use.

See: TLSX_UseSessionTicket

Return:

- SSL_SUCCESS Function executed successfully.
- BAD_FUNC_ARG Returned if ctx is null.
- MEMORY_E Error allocating memory in internal function.

Example

```
wolfSSL_Init();
WOLFSSL_CTX* ctx;
WOLFSSL_METHOD method = // Some wolfSSL method ;
ctx = wolfSSL_CTX_new(method);

if(wolfSSL_CTX_UseSessionTicket(ctx) != SSL_SUCCESS)
{
    // Error setting session ticket
}
```

C.52.2.367 function wolfSSL_get_SessionTicket

```
int wolfSSL_get_SessionTicket(
    WOLFSSL * ssl,
    unsigned char * buf,
    word32 * bufSz
)
```

This function copies the ticket member of the Session structure to the buffer.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **buf** a byte pointer representing the memory buffer.
- **bufSz** a word32 pointer representing the buffer size.

See:

- wolfSSL_UseSessionTicket
- wolfSSL_set_SessionTicket

Return:

- SSL_SUCCESS returned if the function executed without error.
- BAD_FUNC_ARG returned if one of the arguments was NULL or if the bufSz argument was 0.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
WOLFSSL* ssl = wolfSSL_new(ctx);
byte* buf;
word32 bufSz;  // Initialize with buf size
…
```

```
if(wolfSSL_get_SessionTicket(ssl, buf, bufSz) <= 0){
    // Nothing was written to the buffer
} else {
    // the buffer holds the content from ssl->session->ticket
}
```

C.52.2.368 function wolfSSL_set_SessionTicket

```
int wolfSSL_set_SessionTicket(
    WOLFSSL * ssl,
    const unsigned char * buf,
    word32 bufSz
)
```

This function sets the ticket member of the WOLFSSL_SESSION structure within the WOLFSSL struct. The buffer passed into the function is copied to memory.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **buf** a byte pointer that gets loaded into the ticket member of the session structure.
- **bufSz** a word32 type that represents the size of the buffer.

See: wolfSSL_set_SessionTicket_cb

Return:

- SSL_SUCCESS returned on successful execution of the function. The function returned without errors.
- BAD_FUNC_ARG returned if the WOLFSSL structure is NULL. This will also be thrown if the buf argument is NULL but the bufSz argument is not zero.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
WOLFSSL* ssl = wolfSSL_new(ctx);
byte* buffer; // File to load
word32 bufSz;
...
if(wolfSSL_KeepArrays(ssl, buffer, bufSz) != SSL_SUCCESS){
    // There was an error loading the buffer to memory.
}
```

C.52.2.369 function wolfSSL_set_SessionTicket_cb

```
int wolfSSL_set_SessionTicket_cb(
    WOLFSSL * ssl,
    CallbackSessionTicket cb,
    void * ctx
)
```

This function sets the session ticket callback. The type CallbackSessionTicket is a function pointer with the signature of: int (*CallbackSessionTicket)(WOLFSSL*, const unsigned char*, int, void*)

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **cb** a function pointer to the type CallbackSessionTicket.
- **ctx** a void pointer to the session_ticket_ctx member of the WOLFSSL structure.

See:

- wolfSSL_get_SessionTicket
- CallbackSessionTicket
- sessionTicketCB

Return:

- SSL_SUCCESS returned if the function executed without error.
- BAD_FUNC_ARG returned if the WOLFSSL structure is NULL.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
…
int sessionTicketCB(WOLFSSL* ssl, const unsigned char* ticket, int ticketSz,
            void* ctx){ … }
wolfSSL_set_SessionTicket_cb(ssl, sessionTicketCB, (void*)"initial session");
```

C.52.2.370 function wolfSSL_send_SessionTicket

```
int wolfSSL_send_SessionTicket(
    WOLFSSL * ssl
)
```

This function sends a session ticket to the client after a TLS v1.3 handhsake has been established.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_get_SessionTicket
- CallbackSessionTicket
- sessionTicketCB

Return:

- WOLFSSL_SUCCESS returned if a new session ticket was sent.
- BAD_FUNC_ARG returned if WOLFSSL structure is NULL, or not using TLS v1.3.
- SIDE_ERROR returned if not a server.
- NOT_READY_ERROR returned if the handshake has not completed.
- WOLFSSL_FATAL_ERROR returned if creating or sending message fails.

Example

```
int ret;
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
…
ret = wolfSSL_send_SessionTicket(ssl);
if (ret != WOLFSSL_SUCCESS) {
    // New session ticket not sent.
}
```

C.52.2.371 function wolfSSL_CTX_set_TicketEncCb

```
int wolfSSL_CTX_set_TicketEncCb(
    WOLFSSL_CTX * ctx,
```

```
    SessionTicketEncCb
)
```

This function sets the session ticket key encrypt callback function for a server to support session tickets as specified in RFC 5077.

Parameters:

- **ctx** pointer to the WOLFSSL_CTX object, created with wolfSSL_CTX_new().
- **cb** user callback function to encrypt/decrypt session tickets
- **ssl(Callback)** pointer to the WOLFSSL object, created with wolfSSL_new()
- **key_name(Callback)** unique key name for this ticket context, should be randomly generated
- **iv(Callback)** unique IV for this ticket, up to 128 bits, should be randomly generated
- **mac(Callback)** up to 256 bit mac for this ticket
- **enc(Callback)** if this encrypt parameter is true the user should fill in key_name, iv, mac, and encrypt the ticket in-place of length inLen and set the resulting output length in *outLen. Returning WOLFSSL_TICKET_RET_OK tells wolfSSL that the encryption was successful. If this encrypt parameter is false, the user should perform a decrypt of the ticket in-place of length inLen using key_name, iv, and mac. The resulting decrypt length should be set in* outLen. Returning WOLFSSL_TICKET_RET_OK tells wolfSSL to proceed using the decrypted ticket. Returning WOLFSSL_TICKET_RET_CREATE tells wolfSSL to use the decrypted ticket but also to generate a new one to send to the client, helpful if recently rolled keys and don't want to force a full handshake. Returning WOLFSSL_TICKET_RET_REJECT tells wolfSSL to reject this ticket, perform a full handshake, and create a new standard session ID for normal session resumption. Returning WOLFSSL_TICKET_RET_FATAL tells wolfSSL to end the connection attempt with a fatal error.
- **ticket(Callback)** the input/output buffer for the encrypted ticket. See the enc parameter
- **inLen(Callback)** the input length of the ticket parameter
- **outLen(Callback)** the resulting output length of the ticket parameter. When entering the callback outLen will indicate the maximum size available in the ticket buffer.
- **userCtx(Callback)** the user context set with wolfSSL_CTX_set_TicketEncCtx()

See:

- wolfSSL_CTX_set_TicketHint
- wolfSSL_CTX_set_TicketEncCtx

Return:

- SSL_SUCCESS will be returned upon successfully setting the session.
- BAD_FUNC_ARG will be returned on failure. This is caused by passing invalid arguments to the function.

Example

```
See wolfssl/test.h myTicketEncCb() used by the example
server and example echoserver.
```

C.52.2.372 function wolfSSL_CTX_set_TicketHint

```
int wolfSSL_CTX_set_TicketHint(
    WOLFSSL_CTX * ctx,
    int
)
```

This function sets the session ticket hint relayed to the client. For server side use.

Parameters:

- **ctx** pointer to the WOLFSSL_CTX object, created with wolfSSL_CTX_new().
- **hint** number of seconds the ticket might be valid for. Hint to client.

See: wolfSSL_CTX_set_TicketEncCb

Return:

- SSL_SUCCESS will be returned upon successfully setting the session.
- BAD_FUNC_ARG will be returned on failure. This is caused by passing invalid arguments to the function.

Example

none

C.52.2.373 function wolfSSL_CTX_set_TicketEncCtx

```
int wolfSSL_CTX_set_TicketEncCtx(
    WOLFSSL_CTX * ctx,
    void *
)
```

This function sets the session ticket encrypt user context for the callback. For server side use.

Parameters:

- **ctx** pointer to the WOLFSSL_CTX object, created with wolfSSL_CTX_new().
- **userCtx** the user context for the callback

See: wolfSSL_CTX_set_TicketEncCb

Return:

- SSL_SUCCESS will be returned upon successfully setting the session.
- BAD_FUNC_ARG will be returned on failure. This is caused by passing invalid arguments to the function.

Example

none

C.52.2.374 function wolfSSL_CTX_get_TicketEncCtx

```
void * wolfSSL_CTX_get_TicketEncCtx(
    WOLFSSL_CTX * ctx
)
```

This function gets the session ticket encrypt user context for the callback. For server side use.

Parameters:

- **ctx** pointer to the WOLFSSL_CTX object, created with wolfSSL_CTX_new().

See: wolfSSL_CTX_set_TicketEncCtx

Return:

- userCtx will be returned upon successfully getting the session.
- NULL will be returned on failure. This is caused by passing invalid arguments to the function, or when the user context has not been set.

Example

none

C.52.2.375 function wolfSSL_SetHsDoneCb

```
int wolfSSL_SetHsDoneCb(
    WOLFSSL * ssl,
    HandShakeDoneCb cb,
    void * user_ctx
)
```

This function sets the handshake done callback. The hsDoneCb and hsDoneCtx members of the WOLFSSL structure are set in this function.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **cb** a function pointer of type HandShakeDoneCb with the signature of the form: int (*HandShakeDoneCb)(WOLFSSL*, void*);
- **user_ctx** a void pointer to the user registered context.

See: HandShakeDoneCb

Return:

- SSL_SUCCESS returned if the function executed without an error. The hsDoneCb and hsDoneCtx members of the WOLFSSL struct are set.
- BAD_FUNC_ARG returned if the WOLFSSL struct is NULL.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
…
int myHsDoneCb(WOLFSSL* ssl, void* user_ctx){
    // callback function
}
…
wolfSSL_SetHsDoneCb(ssl, myHsDoneCb, NULL);
```

C.52.2.376 function wolfSSL_PrintSessionStats

```
int wolfSSL_PrintSessionStats(
    void
)
```

This function prints the statistics from the session.

Parameters:

- **none** No parameters.

See: wolfSSL_get_session_stats

Return:

- SSL_SUCCESS returned if the function and subroutines return without error. The session stats have been successfully retrieved and printed.
- BAD_FUNC_ARG returned if the subroutine wolfSSL_get_session_stats() was passed an unacceptable argument.
- BAD_MUTEX_E returned if there was a mutex error in the subroutine.

Example

```
// You will need to have a session object to retrieve stats from.
if(wolfSSL_PrintSessionStats(void) != SSL_SUCCESS   ){
    // Did not print session stats
}
```

C.52.2.377 function wolfSSL_get_session_stats

```
int wolfSSL_get_session_stats(
    unsigned int * active,
    unsigned int * total,
    unsigned int * peak,
    unsigned int * maxSessions
)
```

This function gets the statistics for the session.

Parameters:

- **active** a word32 pointer representing the total current sessions.
- **total** a word32 pointer representing the total sessions.
- **peak** a word32 pointer representing the peak sessions.
- **maxSessions** a word32 pointer representing the maximum sessions.

See: wolfSSL_PrintSessionStats

Return:

- SSL_SUCCESS returned if the function and subroutines return without error. The session stats have been successfully retrieved and printed.
- BAD_FUNC_ARG returned if the subroutine wolfSSL_get_session_stats() was passed an unacceptable argument.
- BAD_MUTEX_E returned if there was a mutex error in the subroutine.

Example

```
int wolfSSL_PrintSessionStats(void){
…
ret = wolfSSL_get_session_stats(&totalSessionsNow,
&totalSessionsSeen, &peak, &maxSessions);
…
return ret;
```

C.52.2.378 function wolfSSL_MakeTlsMasterSecret

```
int wolfSSL_MakeTlsMasterSecret(
    unsigned char * ms,
    word32 msLen,
    const unsigned char * pms,
    word32 pmsLen,
    const unsigned char * cr,
    const unsigned char * sr,
    int tls1_2,
    int hash_type
)
```

This function copies the values of cr and sr then passes through to wc_PRF (pseudo random function) and returns that value.

Parameters:

- **ms** the master secret held in the Arrays structure.
- **msLen** the length of the master secret.
- **pms** the pre-master secret held in the Arrays structure.
- **pmsLen** the length of the pre-master secret.
- **cr** the client random.
- **sr** the server random.
- **tls1_2** signifies that the version is at least tls version 1.2.
- **hash_type** signifies the hash type.

See:

- wc_PRF
- MakeTlsMasterSecret

Return:

- 0 on success
- BUFFER_E returned if there will be an error with the size of the buffer.
- MEMORY_E returned if a subroutine failed to allocate dynamic memory.

Example

```
WOLFSSL* ssl;

called in MakeTlsMasterSecret and retrieves the necessary
information as follows:

int MakeTlsMasterSecret(WOLFSSL* ssl){
int ret;
ret = wolfSSL_makeTlsMasterSecret(ssl->arrays->masterSecret, SECRET_LEN,
ssl->arrays->preMasterSecret, ssl->arrays->preMasterSz,
ssl->arrays->clientRandom, ssl->arrays->serverRandom,
IsAtLeastTLSv1_2(ssl), ssl->specs.mac_algorithm);
…
return ret;

}
```

C.52.2.379 function wolfSSL_DeriveTlsKeys

```
int wolfSSL_DeriveTlsKeys(
    unsigned char * key_data,
    word32 keyLen,
    const unsigned char * ms,
    word32 msLen,
    const unsigned char * sr,
    const unsigned char * cr,
    int tls1_2,
    int hash_type
)
```

An external facing wrapper to derive TLS Keys.

Parameters:

- **key_data** a byte pointer that is allocateded in DeriveTlsKeys and passed through to wc_PRF to hold the final hash.
- **keyLen** a word32 type that is derived in DeriveTlsKeys from the WOLFSSL structure's specs member.

- **ms** a constant pointer type holding the master secret held in the arrays structure within the WOLFSSL structure.
- **msLen** a word32 type that holds the length of the master secret in an enumerated define, SECRET_LEN.
- **sr** a constant byte pointer to the serverRandom member of the arrays structure within the WOLFSSL structure.
- **cr** a constant byte pointer to the clientRandom member of the arrays structure within the WOLFSSL structure.
- **tls1_2** an integer type returned from IsAtLeastTLSv1_2().
- **hash_type** an integer type held in the WOLFSSL structure.

See:

- wc_PRF
- DeriveTlsKeys
- IsAtLeastTLSv1_2

Return:

- 0 returned on success.
- BUFFER_E returned if the sum of labLen and seedLen (computes total size) exceeds the maximum size.
- MEMORY_E returned if the allocation of memory failed.

Example

```
int DeriveTlsKeys(WOLFSSL* ssl){
int ret;
…
ret = wolfSSL_DeriveTlsKeys(key_data, length, ssl->arrays->masterSecret,
SECRET_LEN, ssl->arrays->clientRandom,
IsAtLeastTLSv1_2(ssl), ssl->specs.mac_algorithm);
…
}
```

C.52.2.380 function wolfSSL_connect_ex

```
int wolfSSL_connect_ex(
    WOLFSSL * ssl,
    HandShakeCallBack hsCb,
    TimeoutCallBack toCb,
    WOLFSSL_TIMEVAL timeout
)
```

wolfSSL_connect_ex() is an extension that allows a HandShake Callback to be set. This can be useful in embedded systems for debugging support when a debugger isn't available and sniffing is impractical. The HandShake Callback will be called whether or not a handshake error occurred. No dynamic memory is used since the maximum number of SSL packets is known. Packet names can be accessed through packetNames[]. The connect extension also allows a Timeout Callback to be set along with a timeout value. This is useful if the user doesn't want to wait for the TCP stack to timeout. This extension can be called with either, both, or neither callbacks.

Parameters:

- **none** No parameters.

See: wolfSSL_accept_ex

Return:

- SSL_SUCCESS upon success.
- GETTIME_ERROR will be returned if gettimeofday() encountered an error.
- SETITIMER_ERROR will be returned if setitimer() encountered an error.
- SIGACT_ERROR will be returned if sigaction() encountered an error.
- SSL_FATAL_ERROR will be returned if the underlying SSL_connect() call encountered an error.

Example

none

C.52.2.381 function wolfSSL_accept_ex

```
int wolfSSL_accept_ex(
    WOLFSSL * ssl,
    HandShakeCallBacki hsCb,
    TimeoutCallBack toCb,
    WOLFSSL_TIMEVAL timeout
)
```

wolfSSL_accept_ex() is an extension that allows a HandShake Callback to be set. This can be useful in embedded systems for debugging support when a debugger isn't available and sniffing is impractical. The HandShake Callback will be called whether or not a handshake error occurred. No dynamic memory is used since the maximum number of SSL packets is known. Packet names can be accessed through packetNames[]. The connect extension also allows a Timeout Callback to be set along with a timeout value. This is useful if the user doesn't want to wait for the TCP stack to timeout. This extension can be called with either, both, or neither callbacks.

Parameters:

- **none** No parameters.

See: wolfSSL_connect_ex

Return:

- SSL_SUCCESS upon success.
- GETTIME_ERROR will be returned if gettimeofday() encountered an error.
- SETITIMER_ERROR will be returned if setitimer() encountered an error.
- SIGACT_ERROR will be returned if sigaction() encountered an error.
- SSL_FATAL_ERROR will be returned if the underlying SSL_accept() call encountered an error.

Example

none

C.52.2.382 function wolfSSL_BIO_set_fp

```
long wolfSSL_BIO_set_fp(
    WOLFSSL_BIO * bio,
    XFILE fp,
    int c
)
```

This is used to set the internal file pointer for a BIO.

Parameters:

- **bio** WOLFSSL_BIO structure to set pair.
- **fp** file pointer to set in bio.
- **c** close file behavior flag.

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_s_mem
- wolfSSL_BIO_get_fp
- wolfSSL_BIO_free

Return:

- SSL_SUCCESS On successfully setting file pointer.
- SSL_FAILURE If an error case was encountered.

Example

```
WOLFSSL_BIO* bio;
XFILE fp;
int ret;
bio  = wolfSSL_BIO_new(wolfSSL_BIO_s_file());
ret  = wolfSSL_BIO_set_fp(bio, fp, BIO_CLOSE);
// check ret value
```

C.52.2.383 function wolfSSL_BIO_get_fp

```
long wolfSSL_BIO_get_fp(
    WOLFSSL_BIO * bio,
    XFILE * fp
)
```

This is used to get the internal file pointer for a BIO.

Parameters:

- **bio** WOLFSSL_BIO structure to set pair.
- **fp** file pointer to set in bio.

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_s_mem
- wolfSSL_BIO_set_fp
- wolfSSL_BIO_free

Return:

- SSL_SUCCESS On successfully getting file pointer.
- SSL_FAILURE If an error case was encountered.

Example

```
WOLFSSL_BIO* bio;
XFILE fp;
int ret;
bio  = wolfSSL_BIO_new(wolfSSL_BIO_s_file());
ret  = wolfSSL_BIO_get_fp(bio, &fp);
// check ret value
```

C.52.2.384 function wolfSSL_check_private_key

```
int wolfSSL_check_private_key(
    const WOLFSSL * ssl
)
```

This function checks that the private key is a match with the certificate being used.

Parameters:

- **ssl** WOLFSSL structure to check.

See:

- wolfSSL_new
- wolfSSL_free

Return:

- SSL_SUCCESS On successfully match.
- SSL_FAILURE If an error case was encountered.
- <0 All error cases other than SSL_FAILURE are negative values.

Example

```
WOLFSSL* ssl;
int ret;
// create and set up ssl
ret  = wolfSSL_check_private_key(ssl);
// check ret value
```

C.52.2.385 function wolfSSL_X509_get_ext_by_NID

```
int wolfSSL_X509_get_ext_by_NID(
    const WOLFSSL_X509 * x509,
    int nid,
    int lastPos
)
```

This function looks for and returns the extension index matching the passed in NID value.

Parameters:

- **x509** certificate to get parse through for extension.
- **nid** extension OID to be found.
- **lastPos** start search from extension after lastPos. Set to -1 initially.

Return:

- = 0 If successful the extension index is returned.
- -1 If extension is not found or error is encountered.

Example

```
const WOLFSSL_X509* x509;
int lastPos = -1;
int idx;

idx = wolfSSL_X509_get_ext_by_NID(x509, NID_basic_constraints, lastPos);
```

C.52.2.386 function wolfSSL_X509_get_ext_d2i

```
void * wolfSSL_X509_get_ext_d2i(
    const WOLFSSL_X509 * x509,
    int nid,
    int * c,
```

```
    int * idx
)
```

This function looks for and returns the extension matching the passed in NID value.

Parameters:

- **x509** certificate to get parse through for extension.
- **nid** extension OID to be found.
- **c** if not NULL is set to -2 for multiple extensions found -1 if not found, 0 if found and not critical and 1 if found and critical.
- **idx** if NULL return first extension matched otherwise if not stored in x509 start at idx.

See: wolfSSL_sk_ASN1_OBJECT_free

Return:

- pointer If successful a STACK_OF(WOLFSSL_ASN1_OBJECT) pointer is returned.
- NULL If extension is not found or error is encountered.

Example

```
const WOLFSSL_X509* x509;
int c;
int idx = 0;
STACK_OF(WOLFSSL_ASN1_OBJECT)* sk;

sk = wolfSSL_X509_get_ext_d2i(x509, NID_basic_constraints, &c, &idx);
//check sk for NULL and then use it. sk needs freed after done.
```

C.52.2.387 function wolfSSL_X509_digest

```
int wolfSSL_X509_digest(
    const WOLFSSL_X509 * x509,
    const WOLFSSL_EVP_MD * digest,
    unsigned char * buf,
    unsigned int * len
)
```

This function returns the hash of the DER certificate.

Parameters:

- **x509** certificate to get the hash of.
- **digest** the hash algorithm to use.
- **buf** buffer to hold hash.
- **len** length of buffer.

See: none

Return:

- SSL_SUCCESS On successfully creating a hash.
- SSL_FAILURE Returned on bad input or unsuccessful hash.

Example

```
WOLFSSL_X509* x509;
unsigned char buffer[64];
unsigned int bufferSz;
int ret;
```

```
ret = wolfSSL_X509_digest(x509, wolfSSL_EVP_sha256(), buffer, &bufferSz);
//check ret value
```

C.52.2.388 function wolfSSL_use_certificate

```
int wolfSSL_use_certificate(
    WOLFSSL * ssl,
    WOLFSSL_X509 * x509
)
```

his is used to set the certificate for WOLFSSL structure to use during a handshake.

Parameters:

- **ssl** WOLFSSL structure to set certificate in.
- **x509** certificate to use.

See:

- wolfSSL_new
- wolfSSL_free

Return:

- SSL_SUCCESS On successful setting argument.
- SSL_FAILURE If a NULL argument passed in.

Example

```
 WOLFSSL* ssl;
WOLFSSL_X509* x509
int ret;
// create ssl object and x509
ret  = wolfSSL_use_certificate(ssl, x509);
// check ret value
```

C.52.2.389 function wolfSSL_use_certificate_ASN1

```
int wolfSSL_use_certificate_ASN1(
    WOLFSSL * ssl,
    unsigned char * der,
    int derSz
)
```

This is used to set the certificate for WOLFSSL structure to use during a handshake. A DER formatted buffer is expected.

Parameters:

- **ssl** WOLFSSL structure to set certificate in.
- **der** DER certificate to use.
- **derSz** size of the DER buffer passed in.

See:

- wolfSSL_new
- wolfSSL_free

Return:

- SSL_SUCCESS On successful setting argument.
- SSL_FAILURE If a NULL argument passed in.

Example

```
WOLFSSL* ssl;
unsigned char* der;
int derSz;
int ret;
// create ssl object and set DER variables
ret  = wolfSSL_use_certificate_ASN1(ssl, der, derSz);
// check ret value
```

C.52.2.390 function wolfSSL_use_PrivateKey

```
int wolfSSL_use_PrivateKey(
    WOLFSSL * ssl,
    WOLFSSL_EVP_PKEY * pkey
)
```

This is used to set the private key for the WOLFSSL structure.

Parameters:

- **ssl** WOLFSSL structure to set argument in.
- **pkey** private key to use.

See:

- wolfSSL_new
- wolfSSL_free

Return:

- SSL_SUCCESS On successful setting argument.
- SSL_FAILURE If a NULL ssl passed in. All error cases will be negative values.

Example

```
WOLFSSL* ssl;
WOLFSSL_EVP_PKEY* pkey;
int ret;
// create ssl object and set up private key
ret  = wolfSSL_use_PrivateKey(ssl, pkey);
// check ret value
```

C.52.2.391 function wolfSSL_use_PrivateKey_ASN1

```
int wolfSSL_use_PrivateKey_ASN1(
    int pri,
    WOLFSSL * ssl,
    unsigned char * der,
    long derSz
)
```

This is used to set the private key for the WOLFSSL structure. A DER formatted key buffer is expected.

Parameters:

- **pri** type of private key.
- **ssl** WOLFSSL structure to set argument in.
- **der** buffer holding DER key.
- **derSz** size of der buffer.

See:

- wolfSSL_new
- wolfSSL_free
- wolfSSL_use_PrivateKey

Return:

- SSL_SUCCESS On successful setting parsing and setting the private key.
- SSL_FAILURE If an NULL ssl passed in. All error cases will be negative values.

Example

```
WOLFSSL* ssl;
unsigned char* pkey;
long pkeySz;
int ret;
// create ssl object and set up private key
ret  = wolfSSL_use_PrivateKey_ASN1(1, ssl, pkey, pkeySz);
// check ret value
```

C.52.2.392 function wolfSSL_use_RSAPrivateKey_ASN1

```
int wolfSSL_use_RSAPrivateKey_ASN1(
    WOLFSSL * ssl,
    unsigned char * der,
    long derSz
)
```

This is used to set the private key for the WOLFSSL structure. A DER formatted RSA key buffer is expected.

Parameters:

- **ssl** WOLFSSL structure to set argument in.
- **der** buffer holding DER key.
- **derSz** size of der buffer.

See:

- wolfSSL_new
- wolfSSL_free
- wolfSSL_use_PrivateKey

Return:

- SSL_SUCCESS On successful setting parsing and setting the private key.
- SSL_FAILURE If an NULL ssl passed in. All error cases will be negative values.

Example

```
WOLFSSL* ssl;
unsigned char* pkey;
long pkeySz;
int ret;
// create ssl object and set up RSA private key
ret  = wolfSSL_use_RSAPrivateKey_ASN1(ssl, pkey, pkeySz);
// check ret value
```

C.52.2.393 function wolfSSL_DSA_dup_DH

```
WOLFSSL_DH * wolfSSL_DSA_dup_DH(
    const WOLFSSL_DSA * r
)
```

This function duplicates the parameters in dsa to a newly created WOLFSSL_DH structure.

Parameters:

- **dsa** WOLFSSL_DSA structure to duplicate.

See: none

Return:

- WOLFSSL_DH If duplicated returns WOLFSSL_DH structure
- NULL upon failure

Example

```
WOLFSSL_DH* dh;
WOLFSSL_DSA* dsa;
// set up dsa
dh = wolfSSL_DSA_dup_DH(dsa);

// check dh is not null
```

C.52.2.394 function wolfSSL_SESSION_get_master_key

```
int wolfSSL_SESSION_get_master_key(
    const WOLFSSL_SESSION * ses,
    unsigned char * out,
    int outSz
)
```

This is used to get the master key after completing a handshake.

Parameters:

- **ses** WOLFSSL_SESSION structure to get master secret buffer from.
- **out** buffer to hold data.
- **outSz** size of out buffer passed in. (if 0 function will return max buffer size needed)

See:

- wolfSSL_new
- wolfSSL_free

Return:

- 0 On successfully getting data returns a value greater than 0
- 0 If no random data buffer or an error state returns 0
- max If outSz passed in is 0 then the maximum buffer size needed is returned

Example

```
WOLFSSL_SESSION ssl;
unsigned char* buffer;
size_t bufferSz;
size_t ret;
// complete handshake and get session structure
```

```
bufferSz  = wolfSSL_SESSION_get_master_secret(ses, NULL, 0);
buffer = malloc(bufferSz);
ret  = wolfSSL_SESSION_get_master_secret(ses, buffer, bufferSz);
// check ret value
```

C.52.2.395 function wolfSSL_SESSION_get_master_key_length

```
int wolfSSL_SESSION_get_master_key_length(
    const WOLFSSL_SESSION * ses
)
```

This is used to get the master secret key length.

Parameters:

- **ses** WOLFSSL_SESSION structure to get master secret buffer from.

See:

- wolfSSL_new
- wolfSSL_free

Return: size Returns master secret key size.

Example

```
WOLFSSL_SESSION ssl;
unsigned char* buffer;
size_t bufferSz;
size_t ret;
// complete handshake and get session structure
bufferSz  = wolfSSL_SESSION_get_master_secret_length(ses);
buffer = malloc(bufferSz);
// check ret value
```

C.52.2.396 function wolfSSL_CTX_set_cert_store

```
void wolfSSL_CTX_set_cert_store(
    WOLFSSL_CTX * ctx,
    WOLFSSL_X509_STORE * str
)
```

This is a setter function for the WOLFSSL_X509_STORE structure in ctx.

Parameters:

- **ctx** pointer to the WOLFSSL_CTX structure for setting cert store pointer.
- **str** pointer to the WOLFSSL_X509_STORE to set in ctx.

See:

- wolfSSL_CTX_new
- wolfSSL_CTX_free

Return: none No return.

Example

```
WOLFSSL_CTX ctx;
WOLFSSL_X509_STORE* st;
// setup ctx and st
```

```
st = wolfSSL_CTX_set_cert_store(ctx, st);
//use st
```

C.52.2.397 function wolfSSL_d2i_X509_bio

```
WOLFSSL_X509 * wolfSSL_d2i_X509_bio(
    WOLFSSL_BIO * bio,
    WOLFSSL_X509 ** x509
)
```

This function get the DER buffer from bio and converts it to a WOLFSSL_X509 structure.

Parameters:

- **bio** pointer to the WOLFSSL_BIO structure that has the DER certificate buffer.
- **x509** pointer that get set to new WOLFSSL_X509 structure created.

See: none

Return:

- pointer returns a WOLFSSL_X509 structure pointer on success.
- Null returns NULL on failure

Example

```
WOLFSSL_BIO* bio;
WOLFSSL_X509* x509;
// load DER into bio
x509 = wolfSSL_d2i_X509_bio(bio, NULL);
Or
wolfSSL_d2i_X509_bio(bio, &x509);
// use x509 returned (check for NULL)
```

C.52.2.398 function wolfSSL_CTX_get_cert_store

```
WOLFSSL_X509_STORE * wolfSSL_CTX_get_cert_store(
    WOLFSSL_CTX * ctx
)
```

This is a getter function for the WOLFSSL_X509_STORE structure in ctx.

Parameters:

- **ctx** pointer to the WOLFSSL_CTX structure for getting cert store pointer.

See:

- wolfSSL_CTX_new
- wolfSSL_CTX_free
- wolfSSL_CTX_set_cert_store

Return:

- WOLFSSL_X509_STORE* On successfully getting the pointer.
- NULL Returned if NULL arguments are passed in.

Example

```
WOLFSSL_CTX ctx;
WOLFSSL_X509_STORE* st;
// setup ctx
```

```
st = wolfSSL_CTX_get_cert_store(ctx);
//use st
```

C.52.2.399 function wolfSSL_BIO_ctrl_pending

```
size_t wolfSSL_BIO_ctrl_pending(
    WOLFSSL_BIO * b
)
```

Gets the number of pending bytes to read. If BIO type is BIO_BIO then is the number to read from pair. If BIO contains an SSL object then is pending data from SSL object (wolfSSL_pending(ssl)). If is BIO_MEMORY type then returns the size of memory buffer.

Parameters:

- **bio** pointer to the WOLFSSL_BIO structure that has already been created.

See:

- wolfSSL_BIO_make_bio_pair
- wolfSSL_BIO_new

Return: >=0 number of pending bytes.

Example

```
WOLFSSL_BIO* bio;
int pending;
bio = wolfSSL_BIO_new();
…
pending = wolfSSL_BIO_ctrl_pending(bio);
```

C.52.2.400 function wolfSSL_get_server_random

```
size_t wolfSSL_get_server_random(
    const WOLFSSL * ssl,
    unsigned char * out,
    size_t outlen
)
```

This is used to get the random data sent by the server during the handshake.

Parameters:

- **ssl** WOLFSSL structure to get clients random data buffer from.
- **out** buffer to hold random data.
- **outSz** size of out buffer passed in. (if 0 function will return max buffer size needed)

See:

- wolfSSL_new
- wolfSSL_free

Return:

- 0 On successfully getting data returns a value greater than 0
- 0 If no random data buffer or an error state returns 0
- max If outSz passed in is 0 then the maximum buffer size needed is returned

Example

```
WOLFSSL ssl;
unsigned char* buffer;
size_t bufferSz;
size_t ret;
bufferSz  = wolfSSL_get_server_random(ssl, NULL, 0);
buffer = malloc(bufferSz);
ret  = wolfSSL_get_server_random(ssl, buffer, bufferSz);
// check ret value
```

C.52.2.401 function wolfSSL_get_client_random

```
size_t wolfSSL_get_client_random(
    const WOLFSSL * ssl,
    unsigned char * out,
    size_t outSz
)
```

This is used to get the random data sent by the client during the handshake.

Parameters:

- **ssl** WOLFSSL structure to get clients random data buffer from.
- **out** buffer to hold random data.
- **outSz** size of out buffer passed in. (if 0 function will return max buffer size needed)

See:

- wolfSSL_new
- wolfSSL_free

Return:

- 0 On successfully getting data returns a value greater than 0
- 0 If no random data buffer or an error state returns 0
- max If outSz passed in is 0 then the maximum buffer size needed is returned

Example

```
WOLFSSL ssl;
unsigned char* buffer;
size_t bufferSz;
size_t ret;
bufferSz  = wolfSSL_get_client_random(ssl, NULL, 0);
buffer = malloc(bufferSz);
ret  = wolfSSL_get_client_random(ssl, buffer, bufferSz);
// check ret value
```

C.52.2.402 function wolfSSL_CTX_get_default_passwd_cb

```
wc_pem_password_cb * wolfSSL_CTX_get_default_passwd_cb(
    WOLFSSL_CTX * ctx
)
```

This is a getter function for the password callback set in ctx.

Parameters:

- **ctx** WOLFSSL_CTX structure to get call back from.

See:

- wolfSSL_CTX_new
- wolfSSL_CTX_free

Return:

- func On success returns the callback function.
- NULL If ctx is NULL then NULL is returned.

Example

```
WOLFSSL_CTX* ctx;
wc_pem_password_cb cb;
// setup ctx
cb = wolfSSL_CTX_get_default_passwd_cb(ctx);
//use cb
```

C.52.2.403 function wolfSSL_CTX_get_default_passwd_cb_userdata

```
void * wolfSSL_CTX_get_default_passwd_cb_userdata(
    WOLFSSL_CTX * ctx
)
```

This is a getter function for the password callback user data set in ctx.

Parameters:

- **ctx** WOLFSSL_CTX structure to get user data from.

See:

- wolfSSL_CTX_new
- wolfSSL_CTX_free

Return:

- pointer On success returns the user data pointer.
- NULL If ctx is NULL then NULL is returned.

Example

```
WOLFSSL_CTX* ctx;
void* data;
// setup ctx
data = wolfSSL_CTX_get_default_passwd_cb(ctx);
//use data
```

C.52.2.404 function wolfSSL_PEM_read_bio_X509_AUX

```
WOLFSSL_X509 * wolfSSL_PEM_read_bio_X509_AUX(
    WOLFSSL_BIO * bp,
    WOLFSSL_X509 ** x,
    wc_pem_password_cb * cb,
    void * u
)
```

This function behaves the same as wolfSSL_PEM_read_bio_X509. AUX signifies containing extra information such as trusted/rejected use cases and friendly name for human readability.

Parameters:

- **bp** WOLFSSL_BIO structure to get PEM buffer from.
- **x** if setting WOLFSSL_X509 by function side effect.

- **cb** password callback.
- **u** NULL terminated user password.

See: wolfSSL_PEM_read_bio_X509

Return:

- WOLFSSL_X509 on successfully parsing the PEM buffer a WOLFSSL_X509 structure is returned.
- Null if failed to parse PEM buffer.

Example

```
WOLFSSL_BIO* bio;
WOLFSSL_X509* x509;
// setup bio
X509 = wolfSSL_PEM_read_bio_X509_AUX(bio, NULL, NULL, NULL);
//check x509 is not null and then use it
```

C.52.2.405 function wolfSSL_CTX_set_tmp_dh

```
long wolfSSL_CTX_set_tmp_dh(
    WOLFSSL_CTX * ctx,
    WOLFSSL_DH * dh
)
```

Initializes the WOLFSSL_CTX structure's dh member with the Diffie-Hellman parameters.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created using wolfSSL_CTX_new().
- **dh** a pointer to a WOLFSSL_DH structure.

See: wolfSSL_BN_bn2bin

Return:

- SSL_SUCCESS returned if the function executed successfully.
- BAD_FUNC_ARG returned if the ctx or dh structures are NULL.
- SSL_FATAL_ERROR returned if there was an error setting a structure value.
- MEMORY_E returned if their was a failure to allocate memory.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
WOLFSSL_DH* dh;
…
return wolfSSL_CTX_set_tmp_dh(ctx, dh);
```

C.52.2.406 function wolfSSL_PEM_read_bio_DSAparams

```
WOLFSSL_DSA * wolfSSL_PEM_read_bio_DSAparams(
    WOLFSSL_BIO * bp,
    WOLFSSL_DSA ** x,
    wc_pem_password_cb * cb,
    void * u
)
```

This function get the DSA parameters from a PEM buffer in bio.

Parameters:

- **bio** pointer to the WOLFSSL_BIO structure for getting PEM memory pointer.

- **x** pointer to be set to new WOLFSSL_DSA structure.
- **cb** password callback function.
- **u** null terminated password string.

See: none

Return:

- WOLFSSL_DSA on successfully parsing the PEM buffer a WOLFSSL_DSA structure is created and returned.
- Null if failed to parse PEM buffer.

Example

```
WOLFSSL_BIO* bio;
WOLFSSL_DSA* dsa;
// setup bio
dsa = wolfSSL_PEM_read_bio_DSAparams(bio, NULL, NULL, NULL);

// check dsa is not NULL and then use dsa
```

C.52.2.407 function wolfSSL_ERR_peek_last_error

```
unsigned long wolfSSL_ERR_peek_last_error(
    void
)
```

This function returns the absolute value of the last error from WOLFSSL_ERROR encountered.

Parameters:

- **none** No parameters.

See: wolfSSL_ERR_print_errors_fp

Return: error Returns absolute value of last error.

Example

```
unsigned long err;
...
err = wolfSSL_ERR_peek_last_error();
// inspect err value
```

C.52.2.408 function WOLF_STACK_OF

```
WOLF_STACK_OF(
    WOLFSSL_X509
) const
```

This function gets the peer's certificate chain.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_X509_get_issuer_name
- wolfSSL_X509_get_subject_name
- wolfSSL_X509_get_isCA

Return:

- pointer returns a pointer to the peer's Certificate stack.
- NULL returned if no peer certificate.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
...
wolfSSL_connect(ssl);
STACK_OF(WOLFSSL_X509)* chain = wolfSSL_get_peer_cert_chain(ssl);
ifchain){
    // You have a pointer to the peer certificate chain
}
```

C.52.2.409 function wolfSSL_CTX_clear_options

```
long wolfSSL_CTX_clear_options(
    WOLFSSL_CTX * ctx,
    long opt
)
```

This function resets option bits of WOLFSSL_CTX object.

Parameters:

- **ctx** pointer to the SSL context.

See:

- wolfSSL_CTX_new
- wolfSSL_new
- wolfSSL_free

Return: option new option bits

Example

```
WOLFSSL_CTX* ctx = 0;
...
wolfSSL_CTX_clear_options(ctx, SSL_OP_NO_TLSv1);
```

C.52.2.410 function wolfSSL_set_jobject

```
int wolfSSL_set_jobject(
    WOLFSSL * ssl,
    void * objPtr
)
```

This function sets the jObjectRef member of the WOLFSSL structure.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **objPtr** a void pointer that will be set to jObjectRef.

See: wolfSSL_get_jobject

Return:

- SSL_SUCCESS returned if jObjectRef is properly set to objPtr.
- SSL_FAILURE returned if the function did not properly execute and jObjectRef is not set.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
WOLFSSL* ssl = WOLFSSL_new();
void* objPtr = &obj;
...
if(wolfSSL_set_jobject(ssl, objPtr)){
    // The success case
}
```

C.52.2.411 function wolfSSL_get_jobject

```
void * wolfSSL_get_jobject(
    WOLFSSL * ssl
)
```

This function returns the jObjectRef member of the WOLFSSL structure.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See: wolfSSL_set_jobject

Return:

- value If the WOLFSSL struct is not NULL, the function returns the jObjectRef value.
- NULL returned if the WOLFSSL struct is NULL.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
WOLFSSL* ssl = wolfSSL(ctx);
...
void* jobject = wolfSSL_get_jobject(ssl);

if(jobject != NULL){
    // Success case
}
```

C.52.2.412 function wolfSSL_set_msg_callback

```
int wolfSSL_set_msg_callback(
    WOLFSSL * ssl,
    SSL_Msg_Cb cb
)
```

This function sets a callback in the ssl. The callback is to observe handshake messages. NULL value of cb resets the callback.

Parameters:

- **ssl** WOLFSSL structure to set callback argument.

See: wolfSSL_set_msg_callback_arg

Return:

- SSL_SUCCESS On success.
- SSL_FAILURE If an NULL ssl passed in.

Example

```
static cb(int write_p, int version, int content_type,
const void *buf, size_t len, WOLFSSL *ssl, void *arg)
…
WOLFSSL* ssl;
ret  = wolfSSL_set_msg_callback(ssl, cb);
// check ret
```

C.52.2.413 function wolfSSL_set_msg_callback_arg

```
int wolfSSL_set_msg_callback_arg(
    WOLFSSL * ssl,
    void * arg
)
```

This function sets associated callback context value in the ssl. The value is handed over to the callback argument.

Parameters:

- **ssl** WOLFSSL structure to set callback argument.

See: wolfSSL_set_msg_callback

Return: none No return.

Example

```
static cb(int write_p, int version, int content_type,
const void *buf, size_t len, WOLFSSL *ssl, void *arg)
…
WOLFSSL* ssl;
ret  = wolfSSL_set_msg_callback(ssl, cb);
// check ret
wolfSSL_set_msg_callback(ssl, arg);
```

C.52.2.414 function wolfSSL_X509_get_next_altname

```
char * wolfSSL_X509_get_next_altname(
    WOLFSSL_X509 *
)
```

This function returns the next, if any, altname from the peer certificate.

Parameters:

- **cert** a pointer to the wolfSSL_X509 structure.

See:

- wolfSSL_X509_get_issuer_name
- wolfSSL_X509_get_subject_name

Return:

- NULL if there is not a next altname.
- cert->altNamesNext->name from the WOLFSSL_X509 structure that is a string value from the altName list is returned if it exists.

Example

```
WOLFSSL_X509 x509 = (WOLFSSL_X509*)XMALLOC(sizeof(WOLFSSL_X509), NULL,
                                                    DYNAMIC_TYPE_X509);
```

```
…
int x509NextAltName = wolfSSL_X509_get_next_altname(x509);
if(x509NextAltName == NULL){
        //There isn't another alt name
}
```

C.52.2.415 function wolfSSL_X509_get_notBefore

```
WOLFSSL_ASN1_TIME * wolfSSL_X509_get_notBefore(
    WOLFSSL_X509 *
)
```

The function checks to see if x509 is NULL and if it's not, it returns the notBefore member of the x509 struct.

Parameters:

- **x509** a pointer to the WOLFSSL_X509 struct.

See: wolfSSL_X509_get_notAfter

Return:

- pointer to struct with ASN1_TIME to the notBefore member of the x509 struct.
- NULL the function returns NULL if the x509 structure is NULL.

Example

```
WOLFSSL_X509* x509 = (WOLFSSL_X509)XMALLOC(sizeof(WOLFSSL_X509), NULL,
DYNAMIC_TYPE_X509) ;
…
const WOLFSSL_ASN1_TIME* notAfter = wolfSSL_X509_get_notBefore(x509);
if(notAfter == NULL){
        //The x509 object was NULL
}
```

C.52.2.416 function wolfSSL_connect

```
int wolfSSL_connect(
    WOLFSSL * ssl
)
```

This function is called on the client side and initiates an SSL/TLS handshake with a server. When this function is called, the underlying communication channel has already been set up. wolfSSL_connect() will only return once the handshake has been finished or an error occurred. wolfSSL takes a different approach to certificate verification than OpenSSL does. The default policy for the client is to verify the server, this means that if you don't load CAs to verify the server you'll get a connect error, unable to verify (_155). It you want to mimic OpenSSL behavior of having SSL_connect succeed even if verifying the server fails and reducing security you can do this by calling: SSL_CTX_set_verify(ctx, SSL_VERIFY_NONE, 0); before calling SSL_new(); Though it's not recommended.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_get_error
- wolfSSL_accept

Return:

- SSL_SUCCESS If successful.
- SSL_FATAL_ERROR will be returned if an error occurred. To get a more detailed error code, call wolfSSL_get_error().

Example

```
int ret = 0;
int err = 0;
WOLFSSL* ssl;
char buffer[80];
...
ret = wolfSSL_connect(ssl);
if (ret != SSL_SUCCESS) {
err = wolfSSL_get_error(ssl, ret);
printf("error = %d, %s\n", err, wolfSSL_ERR_error_string(err, buffer));
}
```

C.52.2.417 function wolfSSL_send_hrr_cookie

```
int wolfSSL_send_hrr_cookie(
    WOLFSSL * ssl,
    const unsigned char * secret,
    unsigned int secretSz
)
```

This function is called on the server side to indicate that a HelloRetryRequest message must contain a Cookie and, in case of using protocol DTLS v1.3, that the handshake will always include a cookie exchange. Please note that when using protocol DTLS v1.3, the cookie exchange is enabled by default. The Cookie holds a hash of the current transcript so that another server process can handle the ClientHello in reply. The secret is used when generating the integrity check on the Cookie data.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **secret** a pointer to a buffer holding the secret. Passing NULL indicates to generate a new random secret.
- **secretSz** Size of the secret in bytes. Passing 0 indicates to use the default size: WC_SHA256_DIGEST_SIZE (or WC_SHA_DIGEST_SIZE when SHA-256 not available).

See:

- wolfSSL_new
- wolfSSL_disable_hrr_cookie

Return:

- BAD_FUNC_ARG if ssl is NULL or not using TLS v1.3.
- SIDE_ERROR if called with a client.
- WOLFSSL_SUCCESS if successful.
- MEMORY_ERROR if allocating dynamic memory for storing secret failed.
- Another -ve value on internal error.

Example

```
int ret;
WOLFSSL* ssl;
char secret[32];
...
ret = wolfSSL__send_hrr_cookie(ssl, secret, sizeof(secret));
if (ret != WOLFSSL_SUCCESS) {
```

```
    // failed to set use of Cookie and secret
}
```

C.52.2.418 function wolfSSL_disable_hrr_cookie

```
int wolfSSL_disable_hrr_cookie(
    WOLFSSL * ssl
)
```

This function is called on the server side to indicate that a HelloRetryRequest message must NOT contain a Cookie and that, if using protocol DTLS v1.3, a cookie exchange will not be included in the handshake. Please note that not doing a cookie exchange when using protocol DTLS v1.3 can make the server susceptible to DoS/Amplification attacks.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See: wolfSSL_send_hrr_cookie

Return:

- WOLFSSL_SUCCESS if successful
- BAD_FUNC_ARG if ssl is NULL or not using TLS v1.3
- SIDE_ERROR if invoked on client

C.52.2.419 function wolfSSL_CTX_no_ticket_TLSv13

```
int wolfSSL_CTX_no_ticket_TLSv13(
    WOLFSSL_CTX * ctx
)
```

This function is called on the server to stop it from sending a resumption session ticket once the handshake is complete.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created with wolfSSL_CTX_new().

See: wolfSSL_no_ticket_TLSv13

Return:

- BAD_FUNC_ARG if ctx is NULL or not using TLS v1.3.
- SIDE_ERROR if called with a client.
- 0 if successful.

Example

```
int ret;
WOLFSSL_CTX* ctx;
...
ret = wolfSSL_CTX_no_ticket_TLSv13(ctx);
if (ret != 0) {
    // failed to set no ticket
}
```

C.52.2.420 function wolfSSL_no_ticket_TLSv13

```
int wolfSSL_no_ticket_TLSv13(
    WOLFSSL * ssl
)
```

This function is called on the server to stop it from sending a resumption session ticket once the handshake is complete.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See: wolfSSL_CTX_no_ticket_TLSv13

Return:

- BAD_FUNC_ARG if ssl is NULL or not using TLS v1.3.
- SIDE_ERROR if called with a client.
- 0 if successful.

Example

```
int ret;
WOLFSSL* ssl;
...
ret = wolfSSL_no_ticket_TLSv13(ssl);
if (ret != 0) {
    // failed to set no ticket
}
```

C.52.2.421 function wolfSSL_CTX_no_dhe_psk

```
int wolfSSL_CTX_no_dhe_psk(
    WOLFSSL_CTX * ctx
)
```

This function is called on a TLS v1.3 wolfSSL context to disallow Diffie-Hellman (DH) style key exchanges when handshakes are using pre-shared keys for authentication.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created with wolfSSL_CTX_new().

See: wolfSSL_no_dhe_psk

Return:

- BAD_FUNC_ARG if ctx is NULL or not using TLS v1.3.
- 0 if successful.

Example

```
int ret;
WOLFSSL_CTX* ctx;
...
ret = wolfSSL_CTX_no_dhe_psk(ctx);
if (ret != 0) {
    // failed to set no DHE for PSK handshakes
}
```

C.52.2.422 function wolfSSL_no_dhe_psk

```
int wolfSSL_no_dhe_psk(
    WOLFSSL * ssl
)
```

This function is called on a TLS v1.3 client or server wolfSSL to disallow Diffie-Hellman (DH) style key exchanges when handshakes are using pre-shared keys for authentication.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See: wolfSSL_CTX_no_dhe_psk

Return:

- BAD_FUNC_ARG if ssl is NULL or not using TLS v1.3.
- 0 if successful.

Example

```
int ret;
WOLFSSL* ssl;
...
ret = wolfSSL_no_dhe_psk(ssl);
if (ret != 0) {
    // failed to set no DHE for PSK handshakes
}
```

C.52.2.423 function wolfSSL_update_keys

```
int wolfSSL_update_keys(
    WOLFSSL * ssl
)
```

This function is called on a TLS v1.3 client or server wolfSSL to force the rollover of keys. A KeyUpdate message is sent to the peer and new keys are calculated for encryption. The peer will send back a KeyUpdate message and the new decryption keys will then be calculated. This function can only be called after a handshake has been completed.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See: wolfSSL_write

Return:

- BAD_FUNC_ARG if ssl is NULL or not using TLS v1.3.
- WANT_WRITE if the writing is not ready.
- WOLFSSL_SUCCESS if successful.

Example

```
int ret;
WOLFSSL* ssl;
...
ret = wolfSSL_update_keys(ssl);
if (ret == WANT_WRITE) {
    // need to call again when I/O ready
}
else if (ret != WOLFSSL_SUCCESS) {
```

```
    // failed to send key update
}
```

C.52.2.424 function wolfSSL_key_update_response

```
int wolfSSL_key_update_response(
    WOLFSSL * ssl,
    int * required
)
```

This function is called on a TLS v1.3 client or server wolfSSL to determine whether a rollover of keys is in progress. When wolfSSL_update_keys() is called, a KeyUpdate message is sent and the encryption key is updated. The decryption key is updated when the response is received.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **required** 0 when no key update response required. 1 when no key update response required.

See: wolfSSL_update_keys

Return:

- 0 on successful.
- BAD_FUNC_ARG if ssl is NULL or not using TLS v1.3.

Example

```
int ret;
WOLFSSL* ssl;
int required;
...
ret = wolfSSL_key_update_response(ssl, &required);
if (ret != 0) {
    // bad parameters
}
if (required) {
    // encrypt Key updated, awaiting response to change decrypt key
}
```

C.52.2.425 function wolfSSL_CTX_allow_post_handshake_auth

```
int wolfSSL_CTX_allow_post_handshake_auth(
    WOLFSSL_CTX * ctx
)
```

This function is called on a TLS v1.3 client wolfSSL context to allow a client certificate to be sent post handshake upon request from server. This is useful when connecting to a web server that has some pages that require client authentication and others that don't.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created with wolfSSL_CTX_new().

See:

- wolfSSL_allow_post_handshake_auth
- wolfSSL_request_certificate

Return:

- BAD_FUNC_ARG if ctx is NULL or not using TLS v1.3.

- SIDE_ERROR if called with a server.
- 0 if successful.

Example

```
int ret;
WOLFSSL_CTX* ctx;
...
ret = wolfSSL_allow_post_handshake_auth(ctx);
if (ret != 0) {
    // failed to allow post handshake authentication
}
```

C.52.2.426 function wolfSSL_allow_post_handshake_auth

```
int wolfSSL_allow_post_handshake_auth(
    WOLFSSL * ssl
)
```

This function is called on a TLS v1.3 client wolfSSL to allow a client certificate to be sent post handshake upon request from server. A Post-Handshake Client Authentication extension is sent in the ClientHello. This is useful when connecting to a web server that has some pages that require client authentication and others that don't.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_CTX_allow_post_handshake_auth
- wolfSSL_request_certificate

Return:

- BAD_FUNC_ARG if ssl is NULL or not using TLS v1.3.
- SIDE_ERROR if called with a server.
- 0 if successful.

Example

```
int ret;
WOLFSSL* ssl;
...
ret = wolfSSL_allow_post_handshake_auth(ssl);
if (ret != 0) {
    // failed to allow post handshake authentication
}
```

C.52.2.427 function wolfSSL_request_certificate

```
int wolfSSL_request_certificate(
    WOLFSSL * ssl
)
```

This function requests a client certificate from the TLS v1.3 client. This is useful when a web server is serving some pages that require client authentication and others that don't. A maximum of 256 requests can be sent on a connection.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_allow_post_handshake_auth
- wolfSSL_write

Return:

- BAD_FUNC_ARG if ssl is NULL or not using TLS v1.3.
- WANT_WRITE if the writing is not ready.
- SIDE_ERROR if called with a client.
- NOT_READY_ERROR if called when the handshake is not finished.
- POST_HAND_AUTH_ERROR if posthandshake authentication is disallowed.
- MEMORY_E if dynamic memory allocation fails.
- WOLFSSL_SUCCESS if successful.

Example

```
int ret;
WOLFSSL* ssl;
...
ret = wolfSSL_request_certificate(ssl);
if (ret == WANT_WRITE) {
    // need to call again when I/O ready
}
else if (ret != WOLFSSL_SUCCESS) {
    // failed to request a client certificate
}
```

C.52.2.428 function wolfSSL_CTX_set1_groups_list

```
int wolfSSL_CTX_set1_groups_list(
    WOLFSSL_CTX * ctx,
    char * list
)
```

This function sets the list of elliptic curve groups to allow on a wolfSSL context in order of preference. The list is a null-terminated text string, and a colon-delimited list. Call this function to set the key exchange elliptic curve parameters to use with the TLS v1.3 connections.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created with wolfSSL_CTX_new().
- **list** a string that is a colon-delimited list of elliptic curve groups.

See:

- wolfSSL_set1_groups_list
- wolfSSL_CTX_set_groups
- wolfSSL_set_groups
- wolfSSL_UseKeyShare
- wolfSSL_preferred_group

Return:

- WOLFSSL_FAILURE if pointer parameters are NULL, there are more than WOLFSSL_MAX_GROUP_COUNT groups, a group name is not recognized or not using TLS v1.3.
- WOLFSSL_SUCCESS if successful.

Example

```
int ret;
WOLFSSL_CTX* ctx;
const char* list = "P-384:P-256";
...
ret = wolfSSL_CTX_set1_groups_list(ctx, list);
if (ret != WOLFSSL_SUCCESS) {
    // failed to set group list
}
```

C.52.2.429 function wolfSSL_set1_groups_list

```
int wolfSSL_set1_groups_list(
    WOLFSSL * ssl,
    char * list
)
```

This function sets the list of elliptic curve groups to allow on a wolfSSL in order of preference. The list is a null-terminated text string, and a colon-delimited list. Call this function to set the key exchange elliptic curve parameters to use with the TLS v1.3 connections.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **list** a string that is a colon separated list of key exchange groups.

See:

- wolfSSL_CTX_set1_groups_list
- wolfSSL_CTX_set_groups
- wolfSSL_set_groups
- wolfSSL_UseKeyShare
- wolfSSL_preferred_group

Return:

- WOLFSSL_FAILURE if pointer parameters are NULL, there are more than WOLFSSL_MAX_GROUP_COUNT groups, a group name is not recognized or not using TLS v1.3.
- WOLFSSL_SUCCESS if successful.

Example

```
int ret;
WOLFSSL* ssl;
const char* list = "P-384:P-256";
...
ret = wolfSSL_CTX_set1_groups_list(ssl, list);
if (ret != WOLFSSL_SUCCESS) {
    // failed to set group list
}
```

C.52.2.430 function wolfSSL_preferred_group

```
int wolfSSL_preferred_group(
    WOLFSSL * ssl
)
```

This function returns the key exchange group the client prefers to use in the TLS v1.3 handshake. Call this function to after a handshake is complete to determine which group the server prefers so that this information can be used in future connections to pre-generate a key pair for key exchange.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_UseKeyShare
- wolfSSL_CTX_set_groups
- wolfSSL_set_groups
- wolfSSL_CTX_set1_groups_list
- wolfSSL_set1_groups_list

Return:

- BAD_FUNC_ARG if ssl is NULL or not using TLS v1.3.
- SIDE_ERROR if called with a server.
- NOT_READY_ERROR if called before handshake is complete.
- Group identifier if successful.

Example

```
int ret;
int group;
WOLFSSL* ssl;
...
ret = wolfSSL_CTX_set1_groups_list(ssl)
if (ret < 0) {
    // failed to get group
}
group = ret;
```

C.52.2.431 function wolfSSL_CTX_set_groups

```
int wolfSSL_CTX_set_groups(
    WOLFSSL_CTX * ctx,
    int * groups,
    int count
)
```

This function sets the list of elliptic curve groups to allow on a wolfSSL context in order of preference. The list is an array of group identifiers with the number of identifiers specified in count. Call this function to set the key exchange elliptic curve parameters to use with the TLS v1.3 connections.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created with wolfSSL_CTX_new().
- **groups** a list of key exchange groups by identifier.
- **count** the number of key exchange groups in groups.

See:

- wolfSSL_set_groups
- wolfSSL_UseKeyShare
- wolfSSL_CTX_set_groups
- wolfSSL_set_groups
- wolfSSL_CTX_set1_groups_list
- wolfSSL_set1_groups_list
- wolfSSL_preferred_group

Return:

- BAD_FUNC_ARG if a pointer parameter is null, the number of groups exceeds WOLFSSL_MAX_GROUP_COUNT or not using TLS v1.3.
- WOLFSSL_SUCCESS if successful.

Example

```
int ret;
WOLFSSL_CTX* ctx;
int* groups = { WOLFSSL_ECC_X25519, WOLFSSL_ECC_SECP256R1 };
int count = 2;
...
ret = wolfSSL_CTX_set1_groups_list(ctx, groups, count);
if (ret != WOLFSSL_SUCCESS) {
    // failed to set group list
}
```

C.52.2.432 function wolfSSL_set_groups

```
int wolfSSL_set_groups(
    WOLFSSL * ssl,
    int * groups,
    int count
)
```

This function sets the list of elliptic curve groups to allow on a wolfSSL. The list is an array of group identifiers with the number of identifiers specified in count. Call this function to set the key exchange elliptic curve parameters to use with the TLS v1.3 connections.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **groups** a list of key exchange groups by identifier.
- **count** the number of key exchange groups in groups.

See:

- wolfSSL_CTX_set_groups
- wolfSSL_UseKeyShare
- wolfSSL_CTX_set_groups
- wolfSSL_set_groups
- wolfSSL_CTX_set1_groups_list
- wolfSSL_set1_groups_list
- wolfSSL_preferred_group

Return:

- BAD_FUNC_ARG if a pointer parameter is null, the number of groups exceeds WOLFSSL_MAX_GROUP_COUNT, any of the identifiers are unrecognized or not using TLS v1.3.
- WOLFSSL_SUCCESS if successful.

Example

```
int ret;
WOLFSSL* ssl;
int* groups = { WOLFSSL_ECC_X25519, WOLFSSL_ECC_SECP256R1 };
int count = 2;
...
ret = wolfSSL_set_groups(ssl, groups, count);
if (ret != WOLFSSL_SUCCESS) {
```

```
    // failed to set group list
}
```

C.52.2.433 function wolfSSL_connect_TLSv13

```
int wolfSSL_connect_TLSv13(
    WOLFSSL *
)
```

This function is called on the client side and initiates a TLS v1.3 handshake with a server. When this function is called, the underlying communication channel has already been set up. wolfSSL_connect() will only return once the handshake has been finished or an error occurred. wolfSSL takes a different approach to certificate verification than OpenSSL does. The default policy for the client is to verify the server, this means that if you don't load CAs to verify the server you'll get a connect error, unable to verify (_155). It you want to mimic OpenSSL behavior of having SSL_connect succeed even if verifying the server fails and reducing security you can do this by calling: SSL_CTX_set_verify(ctx, SSL_VERIFY_NONE, 0); before calling SSL_new(); Though it's not recommended.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_get_error
- wolfSSL_connect
- wolfSSL_accept_TLSv13
- wolfSSL_accept

Return:

- SSL_SUCCESS upon success.
- SSL_FATAL_ERROR will be returned if an error occurred. To get a more detailed error code, call wolfSSL_get_error().

Example

```
int ret = 0;
int err = 0;
WOLFSSL* ssl;
char buffer[80];
...

ret = wolfSSL_connect_TLSv13(ssl);
if (ret != SSL_SUCCESS) {
    err = wolfSSL_get_error(ssl, ret);
    printf("error = %d, %s\n", err, wolfSSL_ERR_error_string(err, buffer));
}
```

C.52.2.434 function wolfSSL_accept_TLSv13

```
wolfSSL_accept_TLSv13(
    WOLFSSL * ssl
)
```

This function is called on the server side and waits for a SSL/TLS client to initiate the SSL/TLS handshake. When this function is called, the underlying communication channel has already been set up. wolfSSL_accept() will only return once the handshake has been finished or an error occurred. Call

this function when expecting a TLS v1.3 connection though older version ClientHello messages are supported.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_get_error
- wolfSSL_connect_TLSv13
- wolfSSL_connect
- wolfSSL_accept_TLSv13
- wolfSSL_accept

Return:

- SSL_SUCCESS upon success.
- SSL_FATAL_ERROR will be returned if an error occurred. To get a more detailed error code, call wolfSSL_get_error().

Example

```
int ret = 0;
int err = 0;
WOLFSSL* ssl;
char buffer[80];
...

ret = wolfSSL_accept_TLSv13(ssl);
if (ret != SSL_SUCCESS) {
    err = wolfSSL_get_error(ssl, ret);
    printf("error = %d, %s\n", err, wolfSSL_ERR_error_string(err, buffer));
}
```

C.52.2.435 function wolfSSL_CTX_set_max_early_data

```
int wolfSSL_CTX_set_max_early_data(
    WOLFSSL_CTX * ctx,
    unsigned int sz
)
```

This function sets the maximum amount of early data that a TLS v1.3 client or server is willing to exchange using the wolfSSL context. Call this function to limit the amount of early data to process to mitigate replay attacks. Early data is protected by keys derived from those of the connection that the session ticket was sent and therefore will be the same every time a session ticket is used in resumption. The value is included in the session ticket for resumption. A server value of zero indicates no early data is to be sent by client using session tickets. A client value of zero indicates that the client will not send any early data. It is recommended that the number of early data bytes be kept as low as practically possible in the application.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created with wolfSSL_CTX_new().
- **sz** the amount of early data to accept in bytes.

See:

- wolfSSL_set_max_early_data
- wolfSSL_write_early_data
- wolfSSL_read_early_data

Return:

- BAD_FUNC_ARG if ctx is NULL or not using TLS v1.3.
- 0 if successful.

Example

```
int ret;
WOLFSSL_CTX* ctx;
...
ret = wolfSSL_CTX_set_max_early_data(ctx, 128);
if (ret != WOLFSSL_SUCCESS) {
    // failed to set group list
}
```

C.52.2.436 function wolfSSL_set_max_early_data

```
int wolfSSL_set_max_early_data(
    WOLFSSL * ssl,
    unsigned int sz
)
```

This function sets the maximum amount of early data that a TLS v1.3 client or server is willing to exchange. Call this function to limit the amount of early data to process to mitigate replay attacks. Early data is protected by keys derived from those of the connection that the session ticket was sent and therefore will be the same every time a session ticket is used in resumption. The value is included in the session ticket for resumption. A server value of zero indicates no early data is to be sent by client using session tickets. A client value of zero indicates that the client will not send any early data. It is recommended that the number of early data bytes be kept as low as practically possible in the application.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **sz** the amount of early data to accept from client in bytes.

See:

- wolfSSL_CTX_set_max_early_data
- wolfSSL_write_early_data
- wolfSSL_read_early_data

Return:

- BAD_FUNC_ARG if ssl is NULL or not using TLS v1.3.
- 0 if successful.

Example

```
int ret;
WOLFSSL* ssl;
...
ret = wolfSSL_set_max_early_data(ssl, 128);
if (ret != WOLFSSL_SUCCESS) {
    // failed to set group list
}
```

C.52.2.437 function wolfSSL_write_early_data

```
int wolfSSL_write_early_data(
    WOLFSSL * ssl,
    const void * data,
    int sz,
    int * outSz
)
```

This function writes early data to the server on resumption. Call this function instead of wolfSSL_connect() to connect to the server and send the data in the handshake. This function is only used with clients.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **data** the buffer holding the early data to write to server.
- **sz** the amount of early data to write in bytes.
- **outSz** the amount of early data written in bytes.

See:

- wolfSSL_read_early_data
- wolfSSL_connect
- wolfSSL_connect_TLSv13

Return:

- BAD_FUNC_ARG if a pointer parameter is NULL, sz is less than 0 or not using TLSv1.3.
- SIDE_ERROR if called with a server.
- WOLFSSL_FATAL_ERROR if the connection is not made.
- WOLFSSL_SUCCESS if successful.

Example

```
int ret = 0;
int err = 0;
WOLFSSL* ssl;
byte earlyData[] = { early data };
int outSz;
char buffer[80];
...

ret = wolfSSL_write_early_data(ssl, earlyData, sizeof(earlyData), &outSz);
if (ret != WOLFSSL_SUCCESS) {
    err = wolfSSL_get_error(ssl, ret);
    printf("error = %d, %s\n", err, wolfSSL_ERR_error_string(err, buffer));
    goto err_label;
}
if (outSz < sizeof(earlyData)) {
    // not all early data was sent
}
ret = wolfSSL_connect_TLSv13(ssl);
if (ret != SSL_SUCCESS) {
    err = wolfSSL_get_error(ssl, ret);
    printf("error = %d, %s\n", err, wolfSSL_ERR_error_string(err, buffer));
}
```

C.52.2.438 function wolfSSL_read_early_data

```
int wolfSSL_read_early_data(
    WOLFSSL * ssl,
    void * data,
    int sz,
    int * outSz
)
```

This function reads any early data from a client on resumption. Call this function instead of wolfSSL_accept() returns true. Early data may be sent by the client in multiple messages. If there is no early data then the handshake will be processed as normal. This function is only used with servers.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **data** a buffer to hold the early data read from client.
- **sz** size of the buffer in bytes.
- **outSz** number of bytes of early data read.

See:

- wolfSSL_write_early_data
- wolfSSL_accept
- wolfSSL_accept_TLSv13

Return:

- BAD_FUNC_ARG if a pointer parameter is NULL, sz is less than 0 or not using TLSv1.3.
- SIDE_ERROR if called with a client.
- WOLFSSL_FATAL_ERROR if accepting a connection fails.
- Number of early data bytes read (may be zero).

Example

```
int ret = 0;
int err = 0;
WOLFSSL* ssl;
byte earlyData[128];
int outSz;
char buffer[80];
...

do {
    ret = wolfSSL_read_early_data(ssl, earlyData, sizeof(earlyData), &outSz);
    if (ret < 0) {
        err = wolfSSL_get_error(ssl, ret);
        printf("error = %d, %s\n", err, wolfSSL_ERR_error_string(err, buffer));
    }
    if (outSz > 0) {
        // early data available
    }
} while (!wolfSSL_is_init_finished(ssl));
```

C.52.2.439 function wolfSSL_inject

```
int wolfSSL_inject(
    WOLFSSL * ssl,
    const void * data,
    int sz
)
```

This function is called to inject data into the WOLFSSL object. This is useful when data needs to be read from a single place and demultiplexed into multiple connections. The caller should then call wolfSSL_read() to extract the plaintext data from the WOLFSSL object.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **data** data to inject into the ssl object.
- **sz** number of bytes of data to inject.

See: wolfSSL_read

Return:

- BAD_FUNC_ARG if any pointer parameter is NULL or sz <= 0
- APP_DATA_READY if there is application data left to read
- MEMORY_E if allocation fails
- WOLFSSL_SUCCESS on success

Example

```
byte buf[2000]
sz = recv(fd, buf, sizeof(buf), 0);
if (sz <= 0)
    // error
if (wolfSSL_inject(ssl, buf, sz) != WOLFSSL_SUCCESS)
    // error
sz = wolfSSL_read(ssl, buf, sizeof(buf);
```

C.52.2.440 function wolfSSL_CTX_set_psk_client_tls13_callback

```
void wolfSSL_CTX_set_psk_client_tls13_callback(
    WOLFSSL_CTX * ctx,
    wc_psk_client_tls13_callback cb
)
```

This function sets the Pre-Shared Key (PSK) client side callback for TLS v1.3 connections. The callback is used to find a PSK identity and return its key and the name of the cipher to use for the handshake. The function sets the client_psk_tls13_cb member of the WOLFSSL_CTX structure.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created with wolfSSL_CTX_new().
- **cb** a Pre-Shared Key (PSK) callback for a TLS 1.3 client.

See:

- wolfSSL_set_psk_client_tls13_callback
- wolfSSL_CTX_set_psk_server_tls13_callback
- wolfSSL_set_psk_server_tls13_callback

Example

```
WOLFSSL_CTX* ctx;
...
wolfSSL_CTX_set_psk_client_tls13_callback(ctx, my_psk_client_tls13_cb);
```

C.52.2.441 function wolfSSL_set_psk_client_tls13_callback

```
void wolfSSL_set_psk_client_tls13_callback(
    WOLFSSL * ssl,
```

```
    wc_psk_client_tls13_callback cb
)
```

This function sets the Pre-Shared Key (PSK) client side callback for TLS v1.3 connections. The callback is used to find a PSK identity and return its key and the name of the cipher to use for the handshake. The function sets the client_psk_tls13_cb member of the options field in WOLFSSL structure.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **cb** a Pre-Shared Key (PSK) callback for a TLS 1.3 client.

See:

- wolfSSL_CTX_set_psk_client_tls13_callback
- wolfSSL_CTX_set_psk_server_tls13_callback
- wolfSSL_set_psk_server_tls13_callback

Example

```
WOLFSSL* ssl;
...
wolfSSL_set_psk_client_tls13_callback(ssl, my_psk_client_tls13_cb);
```

C.52.2.442 function wolfSSL_CTX_set_psk_server_tls13_callback

```
void wolfSSL_CTX_set_psk_server_tls13_callback(
    WOLFSSL_CTX * ctx,
    wc_psk_server_tls13_callback cb
)
```

This function sets the Pre-Shared Key (PSK) server side callback for TLS v1.3 connections. The callback is used to find a PSK identity and return its key and the name of the cipher to use for the handshake. The function sets the server_psk_tls13_cb member of the WOLFSSL_CTX structure.

Parameters:

- **ctx** a pointer to a WOLFSSL_CTX structure, created with wolfSSL_CTX_new().
- **cb** a Pre-Shared Key (PSK) callback for a TLS 1.3 server.

See:

- wolfSSL_CTX_set_psk_client_tls13_callback
- wolfSSL_set_psk_client_tls13_callback
- wolfSSL_set_psk_server_tls13_callback

Example

```
WOLFSSL_CTX* ctx;
...
wolfSSL_CTX_set_psk_server_tls13_callback(ctx, my_psk_client_tls13_cb);
```

C.52.2.443 function wolfSSL_set_psk_server_tls13_callback

```
void wolfSSL_set_psk_server_tls13_callback(
    WOLFSSL * ssl,
    wc_psk_server_tls13_callback cb
)
```

This function sets the Pre-Shared Key (PSK) server side callback for TLS v1.3 connections. The callback is used to find a PSK identity and return its key and the name of the cipher to use for the handshake. The function sets the server_psk_tls13_cb member of the options field in WOLFSSL structure.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **cb** a Pre-Shared Key (PSK) callback for a TLS 1.3 server.

See:

- wolfSSL_CTX_set_psk_client_tls13_callback
- wolfSSL_set_psk_client_tls13_callback
- wolfSSL_CTX_set_psk_server_tls13_callback

Example

```
WOLFSSL* ssl;
...
wolfSSL_set_psk_server_tls13_callback(ssl, my_psk_server_tls13_cb);
```

C.52.2.444 function wolfSSL_UseKeyShare

```
int wolfSSL_UseKeyShare(
    WOLFSSL * ssl,
    word16 group
)
```

This function creates a key share entry from the group including generating a key pair. The KeyShare extension contains all the generated public keys for key exchange. If this function is called, then only the groups specified will be included. Call this function when a preferred group has been previously established for the server.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **group** a key exchange group identifier.

See:

- wolfSSL_preferred_group
- wolfSSL_CTX_set1_groups_list
- wolfSSL_set1_groups_list
- wolfSSL_CTX_set_groups
- wolfSSL_set_groups
- wolfSSL_NoKeyShares

Return:

- BAD_FUNC_ARG if ssl is NULL.
- MEMORY_E when dynamic memory allocation fails.
- WOLFSSL_SUCCESS if successful.

Example

```
int ret;
WOLFSSL* ssl;
...
ret = wolfSSL_UseKeyShare(ssl, WOLFSSL_ECC_X25519);
if (ret != WOLFSSL_SUCCESS) {
    // failed to set key share
}
```

C.52.2.445 function wolfSSL_NoKeyShares

```
int wolfSSL_NoKeyShares(
    WOLFSSL * ssl
)
```

This function is called to ensure no key shares are sent in the ClientHello. This will force the server to respond with a HelloRetryRequest if a key exchange is required in the handshake. Call this function when the expected key exchange group is not known and to avoid the generation of keys unnecessarily. Note that an extra round-trip will be required to complete the handshake when a key exchange is required.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See: wolfSSL_UseKeyShare

Return:

- BAD_FUNC_ARG if ssl is NULL.
- SIDE_ERROR if called with a server.
- WOLFSSL_SUCCESS if successful.

Example

```
int ret;
WOLFSSL* ssl;
...
ret = wolfSSL_NoKeyShares(ssl);
if (ret != WOLFSSL_SUCCESS) {
    // failed to set no key shares
}
```

C.52.2.446 function wolfTLSv1_3_server_method_ex

```
WOLFSSL_METHOD * wolfTLSv1_3_server_method_ex(
    void * heap
)
```

This function is used to indicate that the application is a server and will only support the TLS 1.3 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().

Parameters:

- **heap** a pointer to a buffer that the static memory allocator will use during dynamic memory allocation.

See:

- wolfSSLv3_server_method
- wolfTLSv1_server_method
- wolfTLSv1_1_server_method
- wolfTLSv1_2_server_method
- wolfTLSv1_3_server_method
- wolfDTLSv1_server_method
- wolfSSLv23_server_method
- wolfSSL_CTX_new

Return:

- If successful, the call will return a pointer to the newly created WOLFSSL_METHOD structure.
- FAIL If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfTLSv1_3_server_method_ex(NULL);
if (method == NULL) {
    // unable to get method
}

ctx = wolfSSL_CTX_new(method);
...
```

C.52.2.447 function wolfTLSv1_3_client_method_ex

```
WOLFSSL_METHOD * wolfTLSv1_3_client_method_ex(
    void * heap
)
```

This function is used to indicate that the application is a client and will only support the TLS 1.3 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().

Parameters:

- **heap** a pointer to a buffer that the static memory allocator will use during dynamic memory allocation.

See:

- wolfSSLv3_client_method
- wolfTLSv1_client_method
- wolfTLSv1_1_client_method
- wolfTLSv1_2_client_method
- wolfTLSv1_3_client_method
- wolfDTLSv1_client_method
- wolfSSLv23_client_method
- wolfSSL_CTX_new

Return:

- If successful, the call will return a pointer to the newly created WOLFSSL_METHOD structure.
- FAIL If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfTLSv1_3_client_method_ex(NULL);
if (method == NULL) {
```

```
    // unable to get method
}

ctx = wolfSSL_CTX_new(method);
...
```

C.52.2.448 function wolfTLSv1_3_server_method

```
WOLFSSL_METHOD * wolfTLSv1_3_server_method(
    void
)
```

This function is used to indicate that the application is a server and will only support the TLS 1.3 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().

See:

- wolfSSLv3_server_method
- wolfTLSv1_server_method
- wolfTLSv1_1_server_method
- wolfTLSv1_2_server_method
- wolfTLSv1_3_server_method_ex
- wolfDTLSv1_server_method
- wolfSSLv23_server_method
- wolfSSL_CTX_new

Return:

- If successful, the call will return a pointer to the newly created WOLFSSL_METHOD structure.
- FAIL If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfTLSv1_3_server_method();
if (method == NULL) {
    // unable to get method
}

ctx = wolfSSL_CTX_new(method);
...
```

C.52.2.449 function wolfTLSv1_3_client_method

```
WOLFSSL_METHOD * wolfTLSv1_3_client_method(
    void
)
```

This function is used to indicate that the application is a client and will only support the TLS 1.3 protocol. This function allocates memory for and initializes a new wolfSSL_METHOD structure to be used when creating the SSL/TLS context with wolfSSL_CTX_new().

See:

- wolfSSLv3_client_method
- wolfTLSv1_client_method
- wolfTLSv1_1_client_method
- wolfTLSv1_2_client_method
- wolfTLSv1_3_client_method_ex
- wolfDTLSv1_client_method
- wolfSSLv23_client_method
- wolfSSL_CTX_new

Return:

- If successful, the call will return a pointer to the newly created WOLFSSL_METHOD structure.
- FAIL If memory allocation fails when calling XMALLOC, the failure value of the underlying malloc() implementation will be returned (typically NULL with errno will be set to ENOMEM).

Example

```
#include <wolfssl/ssl.h>

WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfTLSv1_3_client_method();
if (method == NULL) {
    // unable to get method
}

ctx = wolfSSL_CTX_new(method);
...
```

C.52.2.450 function wolfTLSv1_3_method_ex

```
WOLFSSL_METHOD * wolfTLSv1_3_method_ex(
    void * heap
)
```

This function returns a WOLFSSL_METHOD similar to wolfTLSv1_3_client_method except that it is not determined which side yet (server/client).

Parameters:

- **heap** a pointer to a buffer that the static memory allocator will use during dynamic memory allocation.

See:

- wolfSSL_new
- wolfSSL_free

Return:

- WOLFSSL_METHOD On successful creations returns a WOLFSSL_METHOD pointer
- NULL Null if memory allocation error or failure to create method

Example

```
WOLFSSL* ctx;
ctx  = wolfSSL_CTX_new(wolfTLSv1_3_method_ex(NULL));
// check ret value
```

C.52.2.451 function wolfTLSv1_3_method

```
WOLFSSL_METHOD * wolfTLSv1_3_method(
    void
)
```

This function returns a WOLFSSL_METHOD similar to wolfTLSv1_3_client_method except that it is not determined which side yet (server/client).

See:

- wolfSSL_new
- wolfSSL_free

Return:

- WOLFSSL_METHOD On successful creations returns a WOLFSSL_METHOD pointer
- NULL Null if memory allocation error or failure to create method

Example

```
WOLFSSL* ctx;
ctx  = wolfSSL_CTX_new(wolfTLSv1_3_method());
// check ret value
```

C.52.2.452 function wolfSSL_CTX_set_ephemeral_key

```
int wolfSSL_CTX_set_ephemeral_key(
    WOLFSSL_CTX * ctx,
    int keyAlgo,
    const char * key,
    unsigned int keySz,
    int format
)
```

This function sets a fixed / static ephemeral key for testing only.

Parameters:

- **ctx** A WOLFSSL_CTX context pointer
- **keyAlgo** enum wc_PkType like WC_PK_TYPE_DH and WC_PK_TYPE_ECDH
- **key** key file path (if keySz == 0) or actual key buffer (PEM or ASN.1)
- **keySz** key size (should be 0 for “key” arg is file path)
- **format** WOLFSSL_FILETYPE_ASN1 or WOLFSSL_FILETYPE_PEM

See: wolfSSL_CTX_get_ephemeral_key

Return: 0 Key loaded successfully

C.52.2.453 function wolfSSL_set_ephemeral_key

```
int wolfSSL_set_ephemeral_key(
    WOLFSSL * ssl,
    int keyAlgo,
    const char * key,
    unsigned int keySz,
    int format
)
```

This function sets a fixed / static ephemeral key for testing only.

Parameters:

- **ssl** A WOLFSSL object pointer
- **keyAlgo** enum wc_PkType like WC_PK_TYPE_DH and WC_PK_TYPE_ECDH
- **key** key file path (if keySz == 0) or actual key buffer (PEM or ASN.1)
- **keySz** key size (should be 0 for "key" arg is file path)
- **format** WOLFSSL_FILETYPE_ASN1 or WOLFSSL_FILETYPE_PEM

See: wolfSSL_get_ephemeral_key

Return: 0 Key loaded successfully

C.52.2.454 function wolfSSL_CTX_get_ephemeral_key

```
int wolfSSL_CTX_get_ephemeral_key(
    WOLFSSL_CTX * ctx,
    int keyAlgo,
    const unsigned char ** key,
    unsigned int * keySz
)
```

This function returns pointer to loaded key as ASN.1/DER.

Parameters:

- **ctx** A WOLFSSL_CTX context pointer
- **keyAlgo** enum wc_PkType like WC_PK_TYPE_DH and WC_PK_TYPE_ECDH
- **key** key buffer pointer
- **keySz** key size pointer

See: wolfSSL_CTX_set_ephemeral_key

Return: 0 Key returned successfully

C.52.2.455 function wolfSSL_get_ephemeral_key

```
int wolfSSL_get_ephemeral_key(
    WOLFSSL * ssl,
    int keyAlgo,
    const unsigned char ** key,
    unsigned int * keySz
)
```

This function returns pointer to loaded key as ASN.1/DER.

Parameters:

- **ssl** A WOLFSSL object pointer
- **keyAlgo** enum wc_PkType like WC_PK_TYPE_DH and WC_PK_TYPE_ECDH
- **key** key buffer pointer
- **keySz** key size pointer

See: wolfSSL_set_ephemeral_key

Return: 0 Key returned successfully

C.52.2.456 function wolfSSL_RSA_sign_generic_padding

```
int wolfSSL_RSA_sign_generic_padding(
    int type,
    const unsigned char * m,
    unsigned int mLen,
```

```
    unsigned char * sigRet,
    unsigned int * sigLen,
    WOLFSSL_RSA * rsa,
    int flag,
    int padding
)
```

Sign a message with the chosen message digest, padding, and RSA key.

Parameters:

- **type** Hash NID
- **m** Message to sign. Most likely this will be the digest of the message to sign
- **mLen** Length of message to sign
- **sigRet** Output buffer
- **sigLen** On Input: length of sigRet buffer On Output: length of data written to sigRet
- **rsa** RSA key used to sign the input
- **flag** 1: Output the signature 0: Output the value that the unpadded signature should be compared to. Note: for RSA_PKCS1_PSS_PADDING the wc_RsaPSS_CheckPadding_ex function should be used to check the output of a *Verify* function.
- **padding** Padding to use. Only RSA_PKCS1_PSS_PADDING and RSA_PKCS1_PADDING are currently supported for signing.

Return: WOLFSSL_SUCCESS on success and c on error

C.52.2.457 function wolfSSL_dtls13_has_pending_msg

```
int wolfSSL_dtls13_has_pending_msg(
    WOLFSSL * ssl
)
```

checks if DTLSv1.3 stack has some messages sent but not yet acknowledged by the other peer

Parameters:

- **ssl** A WOLFSSL object pointer

Return: 1 if there are pending messages, 0 otherwise

C.52.2.458 function wolfSSL_SESSION_get_max_early_data

```
unsigned int wolfSSL_SESSION_get_max_early_data(
    const WOLFSSL_SESSION * s
)
```

Get the maximum size of Early Data from a session.

Parameters:

- **s** the WOLFSSL_SESSION instance.

See:

- wolfSSL_set_max_early_data
- wolfSSL_write_early_data
- wolfSSL_read_early_data

Return: the value of max_early_data that was configured in the WOLFSSL* the session was derived from.

C.52.2.459 function wolfSSL_CRYPTO_get_ex_new_index

```
int wolfSSL_CRYPTO_get_ex_new_index(
    int ,
    void * ,
    void * ,
    void * ,
    void *
)
```

Get a new index for external data. This entry applies also for the following API:

Parameters:

- **All** input parameters are ignored. The callback functions are not supported with wolfSSL.

Return: The new index value to be used with the external data API for this object class.

- wolfSSL_CTX_get_ex_new_index
- wolfSSL_get_ex_new_index
- wolfSSL_SESSION_get_ex_new_index
- wolfSSL_X509_get_ex_new_index

C.52.2.460 function wolfSSL_CTX_set_client_cert_type

```
int wolfSSL_CTX_set_client_cert_type(
    WOLFSSL_CTX * ctx,
    const char * buf,
    int len
)
```

In case this function is called in a client side, set certificate types that can be sent to its peer. In case called in a server side, set certificate types that can be acceptable from its peer. Put cert types in the buffer with prioritised order. To reset the settings to default, pass NULL for the buffer or pass zero for len. By default, certificate type is only X509. In case both side intend to send or accept “Raw public key” cert, WOLFSSL_CERT_TYPE_RPK should be included in the buffer to set.

Parameters:

- **ctx** WOLFSSL_CTX object pointer
- **buf** A buffer where certificate types are stored
- **len** buf size in bytes (same as number of certificate types included) *Example*

```
int ret;
WOLFSSL_CTX* ctx;
char buf[] = {WOLFSSL_CERT_TYPE_RPK, WOLFSSL_CERT_TYPE_X509};
int len = sizeof(buf)/sizeof(char);
...

ret = wolfSSL_CTX_set_client_cert_type(ctx, buf, len);
```

See:

- wolfSSL_set_client_cert_type
- wolfSSL_CTX_set_server_cert_type
- wolfSSL_set_server_cert_type
- wolfSSL_get_negotiated_client_cert_type
- wolfSSL_get_negotiated_server_cert_type

Return:

- WOLFSSL_SUCCESS if cert types set successfully
- BAD_FUNC_ARG if NULL was passed for ctx, illegal value was specified as cert type, buf size exceed MAX_CLIENT_CERT_TYPE_CNT was specified or a duplicate value is found in buf.

C.52.2.461 function wolfSSL_CTX_set_server_cert_type

```
int wolfSSL_CTX_set_server_cert_type(
    WOLFSSL_CTX * ctx,
    const char * buf,
    int len
)
```

In case this function is called in a server side, set certificate types that can be sent to its peer. In case called in a client side, set certificate types that can be acceptable from its peer. Put cert types in the buffer with prioritised order. To reset the settings to default, pass NULL for the buffer or pass zero for len. By default, certificate type is only X509. In case both side intend to send or accept “Raw public key” cert, WOLFSSL_CERT_TYPE_RPK should be included in the buffer to set.

Parameters:

- **ctx** WOLFSSL_CTX object pointer
- **buf** A buffer where certificate types are stored
- **len** buf size in bytes (same as number of certificate types included) *Example*

```
int ret;
WOLFSSL_CTX* ctx;
char buf[] = {WOLFSSL_CERT_TYPE_RPK, WOLFSSL_CERT_TYPE_X509};
int len = sizeof(buf)/sizeof(char);
...

ret = wolfSSL_CTX_set_server_cert_type(ctx, buf, len);
```

See:

- wolfSSL_set_client_cert_type
- wolfSSL_CTX_set_client_cert_type
- wolfSSL_set_server_cert_type
- wolfSSL_get_negotiated_client_cert_type
- wolfSSL_get_negotiated_server_cert_type

Return:

- WOLFSSL_SUCCESS if cert types set successfully
- BAD_FUNC_ARG if NULL was passed for ctx, illegal value was specified as cert type, buf size exceed MAX_SERVER_CERT_TYPE_CNT was specified or a duplicate value is found in buf.

C.52.2.462 function wolfSSL_set_client_cert_type

```
int wolfSSL_set_client_cert_type(
    WOLFSSL * ssl,
    const char * buf,
    int len
)
```

In case this function is called in a client side, set certificate types that can be sent to its peer. In case called in a server side, set certificate types that can be acceptable from its peer. Put cert types in the buffer with prioritised order. To reset the settings to default, pass NULL for the buffer or pass zero for len. By default, certificate type is only X509. In case both side intend to send or accept “Raw public key” cert, WOLFSSL_CERT_TYPE_RPK should be included in the buffer to set.

Parameters:

- **ssl** WOLFSSL object pointer
- **buf** A buffer where certificate types are stored
- **len** buf size in bytes (same as number of certificate types included) *Example*

```
int ret;
WOLFSSL* ssl;
char buf[] = {WOLFSSL_CERT_TYPE_RPK, WOLFSSL_CERT_TYPE_X509};
int len = sizeof(buf)/sizeof(char);
...

ret = wolfSSL_set_client_cert_type(ssl, buf, len);
```

See:

- wolfSSL_CTX_set_client_cert_type
- wolfSSL_CTX_set_server_cert_type
- wolfSSL_set_server_cert_type
- wolfSSL_get_negotiated_client_cert_type
- wolfSSL_get_negotiated_server_cert_type

Return:

- WOLFSSL_SUCCESS if cert types set successfully
- BAD_FUNC_ARG if NULL was passed for ctx, illegal value was specified as cert type, buf size exceed MAX_CLIENT_CERT_TYPE_CNT was specified or a duplicate value is found in buf.

C.52.2.463 function wolfSSL_set_server_cert_type

```
int wolfSSL_set_server_cert_type(
    WOLFSSL * ssl,
    const char * buf,
    int len
)
```

In case this function is called in a server side, set certificate types that can be sent to its peer. In case called in a client side, set certificate types that can be acceptable from its peer. Put cert types in the buffer with prioritised order. To reset the settings to default, pass NULL for the buffer or pass zero for len. By default, certificate type is only X509. In case both side intend to send or accept “Raw public key” cert, WOLFSSL_CERT_TYPE_RPK should be included in the buffer to set.

Parameters:

- **ctx** WOLFSSL_CTX object pointer
- **buf** A buffer where certificate types are stored
- **len** buf size in bytes (same as number of certificate types included) *Example*

```
int ret;
WOLFSSL* ssl;
char buf[] = {WOLFSSL_CERT_TYPE_RPK, WOLFSSL_CERT_TYPE_X509};
int len = sizeof(buf)/sizeof(char);
...

ret = wolfSSL_set_server_cert_type(ssl, buf, len);
```

See:

- wolfSSL_set_client_cert_type
- wolfSSL_CTX_set_server_cert_type

- wolfSSL_set_server_cert_type
- wolfSSL_get_negotiated_client_cert_type
- wolfSSL_get_negotiated_server_cert_type

Return:

- WOLFSSL_SUCCESS if cert types set successfully
- BAD_FUNC_ARG if NULL was passed for ctx, illegal value was specified as cert type, buf size exceed MAX_SERVER_CERT_TYPE_CNT was specified or a duplicate value is found in buf.

C.52.2.464 function wolfSSL_get_negotiated_client_cert_type

```
int wolfSSL_get_negotiated_client_cert_type(
    WOLFSSL * ssl,
    int * tp
)
```

This function returns the result of the client certificate type negotiation done in ClientHello and ServerHello. WOLFSSL_SUCCESS is returned as a return value if no negotiation occurs and WOLFSSL_CERT_TYPE_UNKNOWN is returned as the certificate type.

Parameters:

- **ssl** WOLFSSL object pointer
- **tp** A buffer where a certificate type is to be returned. One of three certificate types will be returned: WOLFSSL_CERT_TYPE_RPK, WOLFSSL_CERT_TYPE_X509 or WOLFSSL_CERT_TYPE_UNKNOWN.

See:

- wolfSSL_set_client_cert_type
- wolfSSL_CTX_set_client_cert_type
- wolfSSL_set_server_cert_type
- wolfSSL_CTX_set_server_cert_type
- wolfSSL_get_negotiated_server_cert_type

Return:

- WOLFSSL_SUCCESS if a negotiated certificate type could be got
- BAD_FUNC_ARG if NULL was passed for ctx or tp

Example

```
int ret;
WOLFSSL* ssl;
int tp;
...

ret = wolfSSL_get_negotiated_client_cert_type(ssl, &tp);
```

C.52.2.465 function wolfSSL_get_negotiated_server_cert_type

```
int wolfSSL_get_negotiated_server_cert_type(
    WOLFSSL * ssl,
    int * tp
)
```

This function returns the result of the server certificate type negotiation done in ClientHello and ServerHello. WOLFSSL_SUCCESS is returned as a return value if no negotiation occurs and WOLFSSL_CERT_TYPE_UNKNOWN is returned as the certificate type.

Parameters:

- **ssl** WOLFSSL object pointer
- **tp** A buffer where a certificate type is to be returned. One of three certificate types will be returned: WOLFSSL_CERT_TYPE_RPK, WOLFSSL_CERT_TYPE_X509 or WOLFSSL_CERT_TYPE_UNKNOWN.
Example

```
int ret;
WOLFSSL* ssl;
int tp;
...

ret = wolfSSL_get_negotiated_server_cert_type(ssl, &tp);
```

See:

- wolfSSL_set_client_cert_type
- wolfSSL_CTX_set_client_cert_type
- wolfSSL_set_server_cert_type
- wolfSSL_CTX_set_server_cert_type
- wolfSSL_get_negotiated_client_cert_type

Return:

- WOLFSSL_SUCCESS if a negotiated certificate type could be got
- BAD_FUNC_ARG if NULL was passed for ctx or tp

C.52.2.466 function wolfSSL_dtls_cid_use

```
int wolfSSL_dtls_cid_use(
    WOLFSSL * ssl
)
```

Enable use of ConnectionID extensions for the SSL object. See RFC 9146 and RFC 9147.

Parameters:

- **ssl** A WOLFSSL object pointer

See:

- wolfSSL_dtls_cid_is_enabled
- wolfSSL_dtls_cid_set
- wolfSSL_dtls_cid_get_rx_size
- wolfSSL_dtls_cid_get_rx
- wolfSSL_dtls_cid_get_tx_size
- wolfSSL_dtls_cid_get_tx

Return: WOLFSSL_SUCCESS on success, error code otherwise

C.52.2.467 function wolfSSL_dtls_cid_is_enabled

```
int wolfSSL_dtls_cid_is_enabled(
    WOLFSSL * ssl
)
```

If invoked after the handshake is complete it checks if ConnectionID was successfully negotiated for the SSL object. See RFC 9146 and RFC 9147.

Parameters:

- **ssl** A WOLFSSL object pointer

See:

- wolfSSL_dtls_cid_use
- wolfSSL_dtls_cid_set
- wolfSSL_dtls_cid_get_rx_size
- wolfSSL_dtls_cid_get_rx
- wolfSSL_dtls_cid_get_tx_size
- wolfSSL_dtls_cid_get_tx

Return: 1 if ConnectionID was correctly negotiated, 0 otherwise

C.52.2.468 function wolfSSL_dtls_cid_set

```
int wolfSSL_dtls_cid_set(
    WOLFSSL * ssl,
    unsigned char * cid,
    unsigned int size
)
```

Set the ConnectionID used by the other peer to send records in this connection. See RFC 9146 and RFC 9147. The ConnectionID must be at maximum DTLS_CID_MAX_SIZE, that is an tunable compile time define, and it can't never be bigger than 255 bytes.

Parameters:

- **ssl** A WOLFSSL object pointern
- **cid** the ConnectionID to be used
- **size** of the ConnectionID provided

See:

- wolfSSL_dtls_cid_use
- wolfSSL_dtls_cid_is_enabled
- wolfSSL_dtls_cid_get_rx_size
- wolfSSL_dtls_cid_get_rx
- wolfSSL_dtls_cid_get_tx_size
- wolfSSL_dtls_cid_get_tx

Return: WOLFSSL_SUCCESS if ConnectionID was correctly set, error code otherwise

C.52.2.469 function wolfSSL_dtls_cid_get_rx_size

```
int wolfSSL_dtls_cid_get_rx_size(
    WOLFSSL * ssl,
    unsigned int * size
)
```

Get the size of the ConnectionID used by the other peer to send records in this connection. See RFC 9146 and RFC 9147. The size is stored in the parameter size.

Parameters:

- **ssl** A WOLFSSL object pointern
- **size** a pointer to an unsigned int where the size will be stored

See:

- wolfSSL_dtls_cid_use
- wolfSSL_dtls_cid_is_enabled
- wolfSSL_dtls_cid_set
- wolfSSL_dtls_cid_get_rx
- wolfSSL_dtls_cid_get_tx_size

- wolfSSL_dtls_cid_get_tx

Return: WOLFSSL_SUCCESS if ConnectionID was correctly negotiated, error code otherwise

C.52.2.470 function wolfSSL_dtls_cid_get_rx

```
int wolfSSL_dtls_cid_get_rx(
    WOLFSSL * ssl,
    unsigned char * buffer,
    unsigned int bufferSz
)
```

Copy the ConnectionID used by the other peer to send records in this connection into the buffer pointed by the parameter buffer. See RFC 9146 and RFC.

1. The available space in the buffer need to be provided in bufferSz. WOLFSSL_SUCCESS if ConnectionID was correctly copied, error code otherwise

sslA WOLFSSL object pointern

bufferA buffer where the ConnectionID will be copied

bufferSzavailable space in buffer

wolfSSL_dtls_cid_get0_rx

wolfSSL_dtls_cid_use

wolfSSL_dtls_cid_is_enabled

wolfSSL_dtls_cid_set

wolfSSL_dtls_cid_get_rx_size

wolfSSL_dtls_cid_get_tx_size

wolfSSL_dtls_cid_get_tx

C.52.2.471 function wolfSSL_dtls_cid_get0_rx

```
int wolfSSL_dtls_cid_get0_rx(
    WOLFSSL * ssl,
    unsigned char ** cid
)
```

Get the ConnectionID used by the other peer. See RFC 9146 and RFC 9147.

Parameters:

- **ssl** A WOLFSSL object pointern
- **cid** Pointer that will be set to the internal memory that holds the CID

See:

- wolfSSL_dtls_cid_get_rx
- wolfSSL_dtls_cid_use
- wolfSSL_dtls_cid_is_enabled
- wolfSSL_dtls_cid_set
- wolfSSL_dtls_cid_get_rx_size
- wolfSSL_dtls_cid_get_tx_size
- wolfSSL_dtls_cid_get_tx

Return: WOLFSSL_SUCCESS if ConnectionID was correctly set in cid.

C.52.2.472 function wolfSSL_dtls_cid_get_tx_size

```
int wolfSSL_dtls_cid_get_tx_size(
    WOLFSSL * ssl,
    unsigned int * size
)
```

Get the size of the ConnectionID used to send records in this connection. See RFC 9146 and RFC 9147. The size is stored in the parameter size.

Parameters:

- **ssl** A WOLFSSL object pointern
- **size** a pointer to an unsigned int where the size will be stored

See:

- wolfSSL_dtls_cid_use
- wolfSSL_dtls_cid_is_enabled
- wolfSSL_dtls_cid_set
- wolfSSL_dtls_cid_get_rx_size
- wolfSSL_dtls_cid_get_rx
- wolfSSL_dtls_cid_get_tx

Return: WOLFSSL_SUCCESS if ConnectionID size was correctly stored, error code otherwise

C.52.2.473 function wolfSSL_dtls_cid_get_tx

```
int wolfSSL_dtls_cid_get_tx(
    WOLFSSL * ssl,
    unsigned char * buffer,
    unsigned int bufferSz
)
```

Copy the ConnectionID used when sending records in this connection into the buffer pointer by the parameter buffer. See RFC 9146 and RFC 9147. The available size need to be provided in bufferSz.

Parameters:

- **ssl** A WOLFSSL object pointern
- **buffer** A buffer where the ConnectionID will be copied
- **bufferSz** available space in buffer

See:

- wolfSSL_dtls_cid_get0_tx
- wolfSSL_dtls_cid_use
- wolfSSL_dtls_cid_is_enabled
- wolfSSL_dtls_cid_set
- wolfSSL_dtls_cid_get_rx_size
- wolfSSL_dtls_cid_get_rx
- wolfSSL_dtls_cid_get_tx_size

Return: WOLFSSL_SUCCESS if ConnectionID was correctly copied, error code otherwise

C.52.2.474 function wolfSSL_dtls_cid_get0_tx

```
int wolfSSL_dtls_cid_get0_tx(
    WOLFSSL * ssl,
    unsigned char ** cid
)
```

Get the ConnectionID used when sending records in this connection. See RFC 9146 and RFC 9147.

Parameters:

- **ssl** A WOLFSSL object pointern
- **cid** Pointer that will be set to the internal memory that holds the CID

See:

- wolfSSL_dtls_cid_get_tx
- wolfSSL_dtls_cid_use
- wolfSSL_dtls_cid_is_enabled
- wolfSSL_dtls_cid_set
- wolfSSL_dtls_cid_get_rx_size
- wolfSSL_dtls_cid_get_rx
- wolfSSL_dtls_cid_get_tx_size

Return: WOLFSSL_SUCCESS if ConnectionID was correctly retrieved, error code otherwise

C.52.2.475 function wolfSSL_dtls_cid_parse

```
const unsigned char * wolfSSL_dtls_cid_parse(
    const unsigned char * msg,
    unsigned int msgSz,
    unsigned int cidSz
)
```

Extract the ConnectionID from a record datagram/message. See RFC 9146 and RFC 9147.

Parameters:

- **msg** buffer holding the datagram read from the network
- **msgSz** size of msg in bytes
- **cid** pointer to the start of the CID inside the msg buffer
- **cidSz** the expected size of the CID. The record layer does not have a CID size field so we have to know beforehand the size of the CID. It is recommended to use a constant CID for all connections.

See:

- wolfSSL_dtls_cid_get_tx
- wolfSSL_dtls_cid_use
- wolfSSL_dtls_cid_is_enabled
- wolfSSL_dtls_cid_set
- wolfSSL_dtls_cid_get_rx_size
- wolfSSL_dtls_cid_get_rx
- wolfSSL_dtls_cid_get_tx_size

C.52.2.476 function wolfSSL_get_client_suites_sigalgs

```
int wolfSSL_get_client_suites_sigalgs(
    const WOLFSSL * ssl,
    const byte ** suites,
    word16 * suiteSz,
    const byte ** hashSigAlgo,
    word16 * hashSigAlgoSz
)
```

This function returns the raw list of ciphersuites and signature algorithms offered by the client. The lists are only stored and returned inside a callback setup with wolfSSL_CTX_set_cert_cb(). This is useful

to be able to dynamically load certificates and keys based on the available ciphersuites and signature algorithms.

Parameters:

- **ssl** The WOLFSSL object to extract the lists from.
- **optional** suites Raw and unfiltered list of client ciphersuites
- **optional** suiteSz Size of suites in bytes
- **optional** hashSigAlgo Raw and unfiltered list of client signature algorithms
- **optional** hashSigAlgoSz Size of hashSigAlgo in bytes

See:

- wolfSSL_get_ciphersuite_info
- wolfSSL_get_sigalg_info

Return:

- WOLFSSL_SUCCESS when suites available
- WOLFSSL_FAILURE when suites not available

Example

```
int certCB(WOLFSSL* ssl, void* arg)
{
    const byte* suites = NULL;
    word16 suiteSz = 0;
    const byte* hashSigAlgo = NULL;
    word16 hashSigAlgoSz = 0;

    wolfSSL_get_client_suites_sigalgs(ssl, &suites, &suiteSz, &hashSigAlgo,
            &hashSigAlgoSz);

    // Choose certificate to load based on ciphersuites and sigalgs
}

WOLFSSL* ctx;
ctx  = wolfSSL_CTX_new(wolfTLSv1_3_method_ex(NULL));
wolfSSL_CTX_set_cert_cb(ctx, certCB, NULL);
```

C.52.2.477 function wolfSSL_get_ciphersuite_info

```
WOLFSSL_CIPHERSUITE_INFO wolfSSL_get_ciphersuite_info(
    byte first,
    byte second
)
```

This returns information about the ciphersuite directly from the raw ciphersuite bytes.

Parameters:

- **first** First byte of the ciphersuite
- **second** Second byte of the ciphersuite

See:

- wolfSSL_get_client_suites_sigalgs
- wolfSSL_get_sigalg_info

Return: WOLFSSL_CIPHERSUITE_INFO A struct containing information about the type of authentication used in the ciphersuite.

Example

```
WOLFSSL_CIPHERSUITE_INFO info =
        wolfSSL_get_ciphersuite_info(suites[0], suites[1]);
if (info.rsaAuth)
    haveRSA = 1;
else if (info.eccAuth)
    haveECC = 1;
```

C.52.2.478 function wolfSSL_get_sigalg_info

```
int wolfSSL_get_sigalg_info(
    byte first,
    byte second,
    int * hashAlgo,
    int * sigAlgo
)
```

This returns information about the hash and signature algorithm directly from the raw ciphersuite bytes.

Parameters:

- **first** First byte of the hash and signature algorithm
- **second** Second byte of the hash and signature algorithm
- **hashAlgo** The enum wc_HashType of the MAC algorithm
- **sigAlgo** The enum Key_Sum of the authentication algorithm

See:

- wolfSSL_get_client_suites_sigalgs
- wolfSSL_get_ciphersuite_info

Return:

- 0 when info was correctly set
- BAD_FUNC_ARG when either input parameters are NULL or the bytes are not a recognized sigalg suite

Example

```
enum wc_HashType hashAlgo;
enum Key_Sum sigAlgo;

wolfSSL_get_sigalg_info(hashSigAlgo[idx+0], hashSigAlgo[idx+1],
        &hashAlgo, &sigAlgo);

if (sigAlgo == RSAk || sigAlgo == RSAPSSk)
    haveRSA = 1;
else if (sigAlgo == ECDSAk)
    haveECC = 1;
```

C.52.3 Source code

```
WOLFSSL_METHOD *wolfDTLSv1_2_client_method_ex(void* heap);

WOLFSSL_METHOD *wolfSSLv23_method(void);
```

```
WOLFSSL_METHOD *wolfSSLv3_server_method(void);

WOLFSSL_METHOD *wolfSSLv3_client_method(void);

WOLFSSL_METHOD *wolfTLSv1_server_method(void);

WOLFSSL_METHOD *wolfTLSv1_client_method(void);

WOLFSSL_METHOD *wolfTLSv1_1_server_method(void);

WOLFSSL_METHOD *wolfTLSv1_1_client_method(void);

WOLFSSL_METHOD *wolfTLSv1_2_server_method(void);

WOLFSSL_METHOD *wolfTLSv1_2_client_method(void);

WOLFSSL_METHOD *wolfDTLSv1_client_method(void);

WOLFSSL_METHOD *wolfDTLSv1_server_method(void);
WOLFSSL_METHOD *wolfDTLSv1_3_server_method(void);
WOLFSSL_METHOD* wolfDTLSv1_3_client_method(void);
WOLFSSL_METHOD *wolfDTLS_server_method(void);
WOLFSSL_METHOD *wolfDTLS_client_method(void);
WOLFSSL_METHOD *wolfDTLSv1_2_server_method(void);

int wolfSSL_use_old_poly(WOLFSSL* ssl, int value);

int wolfSSL_dtls_import(WOLFSSL* ssl, unsigned char* buf,
                                                        unsigned int sz);


int wolfSSL_tls_import(WOLFSSL* ssl, const unsigned char* buf,
        unsigned int sz);

int wolfSSL_CTX_dtls_set_export(WOLFSSL_CTX* ctx,
                                                    wc_dtls_export func);

int wolfSSL_dtls_set_export(WOLFSSL* ssl, wc_dtls_export func);

int wolfSSL_dtls_export(WOLFSSL* ssl, unsigned char* buf,
                                                      unsigned int* sz);

int wolfSSL_tls_export(WOLFSSL* ssl, unsigned char* buf,
        unsigned int* sz);

int wolfSSL_CTX_load_static_memory(WOLFSSL_CTX** ctx,
                                          wolfSSL_method_func method,
                                         unsigned char* buf, unsigned int sz,
                                          int flag, int max);

int wolfSSL_CTX_is_static_memory(WOLFSSL_CTX* ctx,
                                                WOLFSSL_MEM_STATS* mem_stats);

int wolfSSL_is_static_memory(WOLFSSL* ssl,
```

```
                                                WOLFSSL_MEM_CONN_STATS* mem_stats);

int wolfSSL_CTX_use_certificate_file(WOLFSSL_CTX* ctx, const char* file,
                                     int format);

int wolfSSL_CTX_use_PrivateKey_file(WOLFSSL_CTX* ctx, const char* file, int
 ↪  format);

int wolfSSL_CTX_load_verify_locations(WOLFSSL_CTX* ctx, const char* file,
                                                  const char* path);

int wolfSSL_CTX_load_verify_locations_ex(WOLFSSL_CTX* ctx, const char* file,
                                         const char* path, unsigned int flags);

const char** wolfSSL_get_system_CA_dirs(word32* num);

int wolfSSL_CTX_load_system_CA_certs(WOLFSSL_CTX* ctx);

int wolfSSL_CTX_trust_peer_cert(WOLFSSL_CTX* ctx, const char* file, int type);

int wolfSSL_CTX_use_certificate_chain_file(WOLFSSL_CTX *ctx,
                                                       const char *file);

int wolfSSL_CTX_use_RSAPrivateKey_file(WOLFSSL_CTX* ctx, const char* file, int
 ↪  format);

long wolfSSL_get_verify_depth(WOLFSSL* ssl);

long wolfSSL_CTX_get_verify_depth(WOLFSSL_CTX* ctx);

int wolfSSL_use_certificate_file(WOLFSSL* ssl, const char* file, int format);

int wolfSSL_use_PrivateKey_file(WOLFSSL* ssl, const char* file, int format);

int wolfSSL_use_certificate_chain_file(WOLFSSL* ssl, const char *file);

int wolfSSL_use_RSAPrivateKey_file(WOLFSSL* ssl, const char* file, int format);

int wolfSSL_CTX_der_load_verify_locations(WOLFSSL_CTX* ctx,
                                          const char* file, int format);

WOLFSSL_CTX* wolfSSL_CTX_new(WOLFSSL_METHOD*);

WOLFSSL* wolfSSL_new(WOLFSSL_CTX*);

int  wolfSSL_set_fd(WOLFSSL* ssl, int fd);

int wolfSSL_set_dtls_fd_connected(WOLFSSL* ssl, int fd);

int wolfDTLS_SetChGoodCb(WOLFSSL* ssl, ClientHelloGoodCb cb, void* user_ctx);

char* wolfSSL_get_cipher_list(int priority);

int  wolfSSL_get_ciphers(char* buf, int len);
```

```
const char* wolfSSL_get_cipher_name(WOLFSSL* ssl);

int  wolfSSL_get_fd(const WOLFSSL*);

int  wolfSSL_get_wfd(const WOLFSSL*);

void wolfSSL_set_using_nonblock(WOLFSSL* ssl, int nonblock);

int  wolfSSL_get_using_nonblock(WOLFSSL*);

int  wolfSSL_write(WOLFSSL* ssl, const void* data, int sz);

int  wolfSSL_read(WOLFSSL* ssl, void* data, int sz);

int  wolfSSL_peek(WOLFSSL* ssl, void* data, int sz);

int  wolfSSL_accept(WOLFSSL*);

int  wolfDTLS_accept_stateless(WOLFSSL* ssl);

void wolfSSL_CTX_free(WOLFSSL_CTX*);

void wolfSSL_free(WOLFSSL*);

int  wolfSSL_shutdown(WOLFSSL*);

int  wolfSSL_send(WOLFSSL* ssl, const void* data, int sz, int flags);

int  wolfSSL_recv(WOLFSSL* ssl, void* data, int sz, int flags);

int  wolfSSL_get_error(WOLFSSL* ssl, int ret);

int  wolfSSL_get_alert_history(WOLFSSL* ssl, WOLFSSL_ALERT_HISTORY *h);

int        wolfSSL_set_session(WOLFSSL* ssl, WOLFSSL_SESSION* session);

WOLFSSL_SESSION* wolfSSL_get_session(WOLFSSL* ssl);

void       wolfSSL_flush_sessions(WOLFSSL_CTX* ctx, long tm);

int        wolfSSL_SetServerID(WOLFSSL* ssl, const unsigned char* id,
                                       int len, int newSession);

int wolfSSL_GetSessionIndex(WOLFSSL* ssl);

int wolfSSL_GetSessionAtIndex(int idx, WOLFSSL_SESSION* session);

    WOLFSSL_X509_CHAIN* wolfSSL_SESSION_get_peer_chain(WOLFSSL_SESSION*
 ↪  session);

void wolfSSL_CTX_set_verify(WOLFSSL_CTX* ctx, int mode,
                                      VerifyCallback verify_callback);
```

```
void wolfSSL_set_verify(WOLFSSL* ssl, int mode, VerifyCallback
 ↪  verify_callback);

void wolfSSL_SetCertCbCtx(WOLFSSL* ssl, void* ctx);

void wolfSSL_CTX_SetCertCbCtx(WOLFSSL_CTX* ctx, void* userCtx);

int  wolfSSL_pending(WOLFSSL*);

void wolfSSL_load_error_strings(void);

int  wolfSSL_library_init(void);

int wolfSSL_SetDevId(WOLFSSL* ssl, int devId);

int wolfSSL_CTX_SetDevId(WOLFSSL_CTX* ctx, int devId);

int wolfSSL_CTX_GetDevId(WOLFSSL_CTX* ctx, WOLFSSL* ssl);

long wolfSSL_CTX_set_session_cache_mode(WOLFSSL_CTX* ctx, long mode);

int  wolfSSL_set_session_secret_cb(WOLFSSL* ssl, SessionSecretCb cb, void*
 ↪  ctx);

int  wolfSSL_save_session_cache(const char* fname);

int  wolfSSL_restore_session_cache(const char* fname);

int  wolfSSL_memsave_session_cache(void* mem, int sz);

int  wolfSSL_memrestore_session_cache(const void* mem, int sz);

int  wolfSSL_get_session_cache_memsize(void);

int  wolfSSL_CTX_save_cert_cache(WOLFSSL_CTX* ctx, const char* fname);

int  wolfSSL_CTX_restore_cert_cache(WOLFSSL_CTX* ctx, const char* fname);

int  wolfSSL_CTX_memsave_cert_cache(WOLFSSL_CTX* ctx, void* mem, int sz, int*
 ↪  used);

int  wolfSSL_CTX_memrestore_cert_cache(WOLFSSL_CTX* ctx, const void* mem, int
 ↪  sz);

int  wolfSSL_CTX_get_cert_cache_memsize(WOLFSSL_CTX* ctx);

int  wolfSSL_CTX_set_cipher_list(WOLFSSL_CTX* ctx, const char* list);

int  wolfSSL_set_cipher_list(WOLFSSL* ssl, const char* list);

void wolfSSL_dtls_set_using_nonblock(WOLFSSL* ssl, int nonblock);
int  wolfSSL_dtls_get_using_nonblock(WOLFSSL* ssl);
int  wolfSSL_dtls_get_current_timeout(WOLFSSL* ssl);
int  wolfSSL_dtls13_use_quick_timeout(WOLFSSL *ssl);
```

```
void  wolfSSL_dtls13_set_send_more_acks(WOLFSSL *ssl, int value);

int  wolfSSL_dtls_set_timeout_init(WOLFSSL* ssl, int);

int  wolfSSL_dtls_set_timeout_max(WOLFSSL* ssl, int);

int  wolfSSL_dtls_got_timeout(WOLFSSL* ssl);

int wolfSSL_dtls_retransmit(WOLFSSL* ssl);

int  wolfSSL_dtls(WOLFSSL* ssl);

int  wolfSSL_dtls_set_peer(WOLFSSL* ssl, void* peer, unsigned int peerSz);

int  wolfSSL_dtls_set_pending_peer(WOLFSSL* ssl, void* peer,
                                   unsigned int peerSz);

int  wolfSSL_dtls_get_peer(WOLFSSL* ssl, void* peer, unsigned int* peerSz);

int  wolfSSL_dtls_get0_peer(WOLFSSL* ssl, const void** peer,
                            unsigned int* peerSz);

char* wolfSSL_ERR_error_string(unsigned long errNumber, char* data);

void  wolfSSL_ERR_error_string_n(unsigned long e, char* buf,
                                         unsigned long sz);

int  wolfSSL_get_shutdown(const WOLFSSL* ssl);

int  wolfSSL_session_reused(WOLFSSL* ssl);

int  wolfSSL_is_init_finished(WOLFSSL* ssl);

const char*  wolfSSL_get_version(WOLFSSL* ssl);

int  wolfSSL_get_current_cipher_suite(WOLFSSL* ssl);

WOLFSSL_CIPHER*  wolfSSL_get_current_cipher(WOLFSSL* ssl);

const char*  wolfSSL_CIPHER_get_name(const WOLFSSL_CIPHER* cipher);

const char*  wolfSSL_get_cipher(WOLFSSL*);

WOLFSSL_SESSION* wolfSSL_get1_session(WOLFSSL* ssl);

WOLFSSL_METHOD* wolfSSLv23_client_method(void);

int wolfSSL_BIO_get_mem_data(WOLFSSL_BIO* bio,void* p);

long wolfSSL_BIO_set_fd(WOLFSSL_BIO* b, int fd, int flag);

int wolfSSL_BIO_set_close(WOLFSSL_BIO *b, long flag);

WOLFSSL_BIO_METHOD *wolfSSL_BIO_s_socket(void);
```

```
int  wolfSSL_BIO_set_write_buf_size(WOLFSSL_BIO *b, long size);

int  wolfSSL_BIO_make_bio_pair(WOLFSSL_BIO *b1, WOLFSSL_BIO *b2);

int  wolfSSL_BIO_ctrl_reset_read_request(WOLFSSL_BIO *bio);

int  wolfSSL_BIO_nread0(WOLFSSL_BIO *bio, char **buf);

int  wolfSSL_BIO_nread(WOLFSSL_BIO *bio, char **buf, int num);

int  wolfSSL_BIO_nwrite(WOLFSSL_BIO *bio, char **buf, int num);

int  wolfSSL_BIO_reset(WOLFSSL_BIO *bio);

int  wolfSSL_BIO_seek(WOLFSSL_BIO *bio, int ofs);

int  wolfSSL_BIO_write_filename(WOLFSSL_BIO *bio, char *name);

long wolfSSL_BIO_set_mem_eof_return(WOLFSSL_BIO *bio, int v);

long wolfSSL_BIO_get_mem_ptr(WOLFSSL_BIO *bio, WOLFSSL_BUF_MEM **m);

char*       wolfSSL_X509_NAME_oneline(WOLFSSL_X509_NAME* name, char* in, int
 ↪  sz);

WOLFSSL_X509_NAME*  wolfSSL_X509_get_issuer_name(WOLFSSL_X509* cert);

WOLFSSL_X509_NAME*  wolfSSL_X509_get_subject_name(WOLFSSL_X509* cert);

int  wolfSSL_X509_get_isCA(WOLFSSL_X509* cert);

int wolfSSL_X509_NAME_get_text_by_NID(WOLFSSL_X509_NAME* name, int nid,
                                      char* buf, int len);

int wolfSSL_X509_get_signature_type(WOLFSSL_X509* cert);

void wolfSSL_X509_free(WOLFSSL_X509* x509);

int wolfSSL_X509_get_signature(WOLFSSL_X509* x509, unsigned char* buf, int*
 ↪  bufSz);

int wolfSSL_X509_STORE_add_cert(WOLFSSL_X509_STORE* store, WOLFSSL_X509* x509);

WOLFSSL_STACK* wolfSSL_X509_STORE_CTX_get_chain(
                                                  WOLFSSL_X509_STORE_CTX* ctx);

int wolfSSL_X509_STORE_set_flags(WOLFSSL_X509_STORE* store,
                                                        unsigned long flag);

const byte* wolfSSL_X509_notBefore(WOLFSSL_X509* x509);

const byte* wolfSSL_X509_notAfter(WOLFSSL_X509* x509);
```

```
WOLFSSL_BIGNUM *wolfSSL_ASN1_INTEGER_to_BN(const WOLFSSL_ASN1_INTEGER *ai,
                                           WOLFSSL_BIGNUM *bn);

long wolfSSL_CTX_add_extra_chain_cert(WOLFSSL_CTX* ctx, WOLFSSL_X509* x509);

int  wolfSSL_CTX_get_read_ahead(WOLFSSL_CTX* ctx);

int  wolfSSL_CTX_set_read_ahead(WOLFSSL_CTX* ctx, int v);

long wolfSSL_CTX_set_tlsext_status_arg(WOLFSSL_CTX* ctx, void* arg);

long wolfSSL_CTX_set_tlsext_opaque_prf_input_callback_arg(
       WOLFSSL_CTX* ctx, void* arg);

long wolfSSL_set_options(WOLFSSL *s, long op);

long wolfSSL_get_options(const WOLFSSL *ssl);

long wolfSSL_set_tlsext_debug_arg(WOLFSSL *ssl, void *arg);

long wolfSSL_set_tlsext_status_type(WOLFSSL *s, int type);

long wolfSSL_get_verify_result(const WOLFSSL *ssl);

void  wolfSSL_ERR_print_errors_fp(XFILE fp, int err);

void  wolfSSL_ERR_print_errors_cb (
       int (*cb)(const char *str, size_t len, void *u), void *u);

void wolfSSL_CTX_set_psk_client_callback(WOLFSSL_CTX* ctx,
                                         wc_psk_client_callback cb);

void wolfSSL_set_psk_client_callback(WOLFSSL* ssl,
                                         wc_psk_client_callback);

const char* wolfSSL_get_psk_identity_hint(const WOLFSSL*);

const char* wolfSSL_get_psk_identity(const WOLFSSL*);

int wolfSSL_CTX_use_psk_identity_hint(WOLFSSL_CTX* ctx, const char* hint);

int wolfSSL_use_psk_identity_hint(WOLFSSL* ssl, const char* hint);

void wolfSSL_CTX_set_psk_server_callback(WOLFSSL_CTX* ctx,
                                         wc_psk_server_callback cb);

void wolfSSL_set_psk_server_callback(WOLFSSL* ssl,
                                         wc_psk_server_callback cb);


int wolfSSL_set_psk_callback_ctx(WOLFSSL* ssl, void* psk_ctx);

int wolfSSL_CTX_set_psk_callback_ctx(WOLFSSL_CTX* ctx, void* psk_ctx);
```

```
void* wolfSSL_get_psk_callback_ctx(WOLFSSL* ssl);

void* wolfSSL_CTX_get_psk_callback_ctx(WOLFSSL_CTX* ctx);

int wolfSSL_CTX_allow_anon_cipher(WOLFSSL_CTX*);

WOLFSSL_METHOD *wolfSSLv23_server_method(void);

int  wolfSSL_state(WOLFSSL* ssl);

WOLFSSL_X509* wolfSSL_get_peer_certificate(WOLFSSL* ssl);

int wolfSSL_want_read(WOLFSSL*);

int wolfSSL_want_write(WOLFSSL*);

int wolfSSL_check_domain_name(WOLFSSL* ssl, const char* dn);

int wolfSSL_Init(void);

int wolfSSL_Cleanup(void);

const char* wolfSSL_lib_version(void);

word32 wolfSSL_lib_version_hex(void);

int wolfSSL_negotiate(WOLFSSL* ssl);

int wolfSSL_set_compression(WOLFSSL* ssl);

int wolfSSL_set_timeout(WOLFSSL* ssl, unsigned int to);

int wolfSSL_CTX_set_timeout(WOLFSSL_CTX* ctx, unsigned int to);

WOLFSSL_X509_CHAIN* wolfSSL_get_peer_chain(WOLFSSL* ssl);

int  wolfSSL_get_chain_count(WOLFSSL_X509_CHAIN* chain);

int  wolfSSL_get_chain_length(WOLFSSL_X509_CHAIN* chain, int idx);

unsigned char* wolfSSL_get_chain_cert(WOLFSSL_X509_CHAIN* chain, int idx);

WOLFSSL_X509* wolfSSL_get_chain_X509(WOLFSSL_X509_CHAIN* chain, int idx);

int  wolfSSL_get_chain_cert_pem(WOLFSSL_X509_CHAIN* chain, int idx,
                                unsigned char* buf, int inLen, int* outLen);

const unsigned char* wolfSSL_get_sessionID(const WOLFSSL_SESSION* s);

int  wolfSSL_X509_get_serial_number(WOLFSSL_X509* x509, unsigned char* in,
                                    int* inOutSz);

char*  wolfSSL_X509_get_subjectCN(WOLFSSL_X509*);
```

```
const unsigned char* wolfSSL_X509_get_der(WOLFSSL_X509* x509, int* outSz);

WOLFSSL_ASN1_TIME* wolfSSL_X509_get_notAfter(WOLFSSL_X509*);

int wolfSSL_X509_version(WOLFSSL_X509*);

WOLFSSL_X509*
        wolfSSL_X509_d2i_fp(WOLFSSL_X509** x509, FILE* file);

WOLFSSL_X509*
    wolfSSL_X509_load_certificate_file(const char* fname, int format);

unsigned char*
          wolfSSL_X509_get_device_type(WOLFSSL_X509* x509, unsigned char* in,
                                       int* inOutSz);

unsigned char*
          wolfSSL_X509_get_hw_type(WOLFSSL_X509* x509, unsigned char* in,
                                   int* inOutSz);

unsigned char*
          wolfSSL_X509_get_hw_serial_number(WOLFSSL_X509* x509,
                                            unsigned char* in, int* inOutSz);

int  wolfSSL_connect_cert(WOLFSSL* ssl);

WC_PKCS12* wolfSSL_d2i_PKCS12_bio(WOLFSSL_BIO* bio,
                                  WC_PKCS12** pkcs12);

WC_PKCS12* wolfSSL_i2d_PKCS12_bio(WOLFSSL_BIO* bio,
                                  WC_PKCS12* pkcs12);

int wolfSSL_PKCS12_parse(WC_PKCS12* pkcs12, const char* psw,
     WOLFSSL_EVP_PKEY** pkey, WOLFSSL_X509** cert,
 ↪  WOLF_STACK_OF(WOLFSSL_X509)** ca);

int  wolfSSL_SetTmpDH(WOLFSSL* ssl, const unsigned char* p, int pSz,
                      const unsigned char* g, int gSz);

int  wolfSSL_SetTmpDH_buffer(WOLFSSL* ssl, const unsigned char* b, long sz,
                             int format);

int  wolfSSL_SetTmpDH_file(WOLFSSL* ssl, const char* f, int format);

int  wolfSSL_CTX_SetTmpDH(WOLFSSL_CTX* ctx, const unsigned char* p,
                          int pSz, const unsigned char* g, int gSz);

int  wolfSSL_CTX_SetTmpDH_buffer(WOLFSSL_CTX* ctx, const unsigned char* b,
                                 long sz, int format);

int  wolfSSL_CTX_SetTmpDH_file(WOLFSSL_CTX* ctx, const char* f,
                               int format);

int wolfSSL_CTX_SetMinDhKey_Sz(WOLFSSL_CTX* ctx, word16);
```

```
int wolfSSL_SetMinDhKey_Sz(WOLFSSL* ssl, word16 keySz_bits);

int wolfSSL_CTX_SetMaxDhKey_Sz(WOLFSSL_CTX* ctx, word16 keySz_bits);

int wolfSSL_SetMaxDhKey_Sz(WOLFSSL* ssl, word16 keySz_bits);

int wolfSSL_GetDhKey_Sz(WOLFSSL*);

int wolfSSL_CTX_SetMinRsaKey_Sz(WOLFSSL_CTX* ctx, short keySz);

int wolfSSL_SetMinRsaKey_Sz(WOLFSSL* ssl, short keySz);

int wolfSSL_CTX_SetMinEccKey_Sz(WOLFSSL_CTX* ssl, short keySz);

int wolfSSL_SetMinEccKey_Sz(WOLFSSL* ssl, short keySz);

int wolfSSL_make_eap_keys(WOLFSSL* ssl, void* key, unsigned int len,
                                                        const char* label);

int wolfSSL_writev(WOLFSSL* ssl, const struct iovec* iov,
                                    int iovcnt);

int wolfSSL_CTX_UnloadCAs(WOLFSSL_CTX*);


int wolfSSL_CTX_UnloadIntermediateCerts(WOLFSSL_CTX* ctx);

int wolfSSL_CTX_Unload_trust_peers(WOLFSSL_CTX*);

int wolfSSL_CTX_trust_peer_buffer(WOLFSSL_CTX* ctx, const unsigned char* in,
                                  long sz, int format);

int wolfSSL_CTX_load_verify_buffer(WOLFSSL_CTX* ctx, const unsigned char* in,
                                   long sz, int format);


int wolfSSL_CTX_load_verify_buffer_ex(WOLFSSL_CTX* ctx,
                                      const unsigned char* in, long sz,
                                     int format, int userChain, word32 flags);

int wolfSSL_CTX_load_verify_chain_buffer_format(WOLFSSL_CTX* ctx,
                                                const unsigned char* in,
                                                long sz, int format);

int wolfSSL_CTX_use_certificate_buffer(WOLFSSL_CTX* ctx,
                                       const unsigned char* in, long sz,
                                       int format);

int wolfSSL_CTX_use_PrivateKey_buffer(WOLFSSL_CTX* ctx,
                                      const unsigned char* in, long sz,
                                      int format);

int wolfSSL_CTX_use_certificate_chain_buffer(WOLFSSL_CTX* ctx,
```

```
                                                const unsigned char* in, long sz);

int wolfSSL_use_certificate_buffer(WOLFSSL* ssl, const unsigned char* in,
                                                  long sz, int format);

int wolfSSL_use_PrivateKey_buffer(WOLFSSL* ssl, const unsigned char* in,
                                                  long sz, int format);

int wolfSSL_use_certificate_chain_buffer(WOLFSSL* ssl,
                                               const unsigned char* in, long sz);

int wolfSSL_UnloadCertsKeys(WOLFSSL*);

int wolfSSL_CTX_set_group_messages(WOLFSSL_CTX*);

int wolfSSL_set_group_messages(WOLFSSL*);

void wolfSSL_SetFuzzerCb(WOLFSSL* ssl, CallbackFuzzer cbf, void* fCtx);

int   wolfSSL_DTLS_SetCookieSecret(WOLFSSL* ssl,
                                                  const unsigned char* secret,
                                                  unsigned int secretSz);

WC_RNG* wolfSSL_GetRNG(WOLFSSL* ssl);

int wolfSSL_CTX_SetMinVersion(WOLFSSL_CTX* ctx, int version);

int wolfSSL_SetMinVersion(WOLFSSL* ssl, int version);

int wolfSSL_GetObjectSize(void);  /* object size based on build */
int wolfSSL_GetOutputSize(WOLFSSL* ssl, int inSz);

int wolfSSL_GetMaxOutputSize(WOLFSSL*);

int wolfSSL_SetVersion(WOLFSSL* ssl, int version);

void  wolfSSL_CTX_SetMacEncryptCb(WOLFSSL_CTX* ctx, CallbackMacEncrypti cb);

void  wolfSSL_SetMacEncryptCtx(WOLFSSL* ssl, void *ctx);

void* wolfSSL_GetMacEncryptCtx(WOLFSSL* ssl);

void  wolfSSL_CTX_SetDecryptVerifyCb(WOLFSSL_CTX* ctx,
                                                  CallbackDecryptVerify cb);

void  wolfSSL_SetDecryptVerifyCtx(WOLFSSL* ssl, void *ctx);

void* wolfSSL_GetDecryptVerifyCtx(WOLFSSL* ssl);

const unsigned char* wolfSSL_GetMacSecret(WOLFSSL* ssl, int verify);

const unsigned char* wolfSSL_GetClientWriteKey(WOLFSSL*);

const unsigned char* wolfSSL_GetClientWriteIV(WOLFSSL*);
```

```
const unsigned char* wolfSSL_GetServerWriteKey(WOLFSSL*);

const unsigned char* wolfSSL_GetServerWriteIV(WOLFSSL*);

int                  wolfSSL_GetKeySize(WOLFSSL*);

int                  wolfSSL_GetIVSize(WOLFSSL*);

int                  wolfSSL_GetSide(WOLFSSL*);

int                  wolfSSL_IsTLSv1_1(WOLFSSL*);

int                  wolfSSL_GetBulkCipher(WOLFSSL*);

int                  wolfSSL_GetCipherBlockSize(WOLFSSL*);

int                  wolfSSL_GetAeadMacSize(WOLFSSL*);

int                  wolfSSL_GetHmacSize(WOLFSSL*);

int                  wolfSSL_GetHmacType(WOLFSSL*);

int                  wolfSSL_GetCipherType(WOLFSSL*);

int wolfSSL_SetTlsHmacInner(WOLFSSL* ssl, byte* inner,
                            word32 sz, int content, int verify);

void  wolfSSL_CTX_SetEccSignCb(WOLFSSL_CTX* ctx, CallbackEccSign cb);

void  wolfSSL_SetEccSignCtx(WOLFSSL* ssl, void *ctx);

void* wolfSSL_GetEccSignCtx(WOLFSSL* ssl);

void  wolfSSL_CTX_SetEccSignCtx(WOLFSSL_CTX* ctx, void *userCtx);

void* wolfSSL_CTX_GetEccSignCtx(WOLFSSL_CTX* ctx);

void  wolfSSL_CTX_SetEccVerifyCb(WOLFSSL_CTX* ctx, CallbackEccVerify cb);

void  wolfSSL_SetEccVerifyCtx(WOLFSSL* ssl, void *ctx);

void* wolfSSL_GetEccVerifyCtx(WOLFSSL* ssl);

void  wolfSSL_CTX_SetRsaSignCb(WOLFSSL_CTX* ctx, CallbackRsaSign cb);

void  wolfSSL_SetRsaSignCtx(WOLFSSL* ssl, void *ctx);

void* wolfSSL_GetRsaSignCtx(WOLFSSL* ssl);

void  wolfSSL_CTX_SetRsaVerifyCb(WOLFSSL_CTX* ctx, CallbackRsaVerify cb);

void  wolfSSL_SetRsaVerifyCtx(WOLFSSL* ssl, void *ctx);
```

```
void* wolfSSL_GetRsaVerifyCtx(WOLFSSL* ssl);

void  wolfSSL_CTX_SetRsaEncCb(WOLFSSL_CTX* ctx, CallbackRsaEnc cb);

void  wolfSSL_SetRsaEncCtx(WOLFSSL* ssl, void *ctx);

void* wolfSSL_GetRsaEncCtx(WOLFSSL* ssl);

void  wolfSSL_CTX_SetRsaDecCb(WOLFSSL_CTX* ctx, CallbackRsaDec cb);

void  wolfSSL_SetRsaDecCtx(WOLFSSL* ssl, void *ctx);

void* wolfSSL_GetRsaDecCtx(WOLFSSL* ssl);

void wolfSSL_CTX_SetCACb(WOLFSSL_CTX* ctx, CallbackCACache cb);

WOLFSSL_CERT_MANAGER* wolfSSL_CertManagerNew_ex(void* heap);

WOLFSSL_CERT_MANAGER* wolfSSL_CertManagerNew(void);

void wolfSSL_CertManagerFree(WOLFSSL_CERT_MANAGER*);

int wolfSSL_CertManagerLoadCA(WOLFSSL_CERT_MANAGER* cm, const char* f,
                                                          const char* d);

int wolfSSL_CertManagerLoadCABuffer(WOLFSSL_CERT_MANAGER* cm,
                                const unsigned char* in, long sz, int format);

int wolfSSL_CertManagerUnloadCAs(WOLFSSL_CERT_MANAGER* cm);

int wolfSSL_CertManagerUnloadIntermediateCerts(WOLFSSL_CERT_MANAGER* cm);

int wolfSSL_CertManagerUnload_trust_peers(WOLFSSL_CERT_MANAGER* cm);

int wolfSSL_CertManagerVerify(WOLFSSL_CERT_MANAGER* cm, const char* f,
                                                            int format);

int wolfSSL_CertManagerVerifyBuffer(WOLFSSL_CERT_MANAGER* cm,
                              const unsigned char* buff, long sz, int format);

void wolfSSL_CertManagerSetVerify(WOLFSSL_CERT_MANAGER* cm,
                                                       VerifyCallback vc);

int wolfSSL_CertManagerCheckCRL(WOLFSSL_CERT_MANAGER* cm,
                                unsigned char* der, int sz);

int wolfSSL_CertManagerEnableCRL(WOLFSSL_CERT_MANAGER* cm,
                                                          int options);

int wolfSSL_CertManagerDisableCRL(WOLFSSL_CERT_MANAGER*);

int wolfSSL_CertManagerLoadCRL(WOLFSSL_CERT_MANAGER* cm,
                              const char* path, int type, int monitor);
```

```
int wolfSSL_CertManagerLoadCRLBuffer(WOLFSSL_CERT_MANAGER* cm,
                                     const unsigned char* buff, long sz,
                                     int type);

int wolfSSL_CertManagerSetCRL_Cb(WOLFSSL_CERT_MANAGER* cm,
                                 CbMissingCRL cb);

int wolfSSL_CertManagerSetCRLUpdate_Cb(WOLFSSL_CERT_MANAGER* cm,
                                       CbUpdateCRL cb);

int wolfSSL_CertManagerGetCRLInfo(WOLFSSL_CERT_MANAGER* cm, CrlInfo* info,
    const byte* buff, long sz, int type)


int wolfSSL_CertManagerFreeCRL(WOLFSSL_CERT_MANAGER* cm);

int wolfSSL_CertManagerCheckOCSP(WOLFSSL_CERT_MANAGER* cm,
                                 unsigned char* der, int sz);

int wolfSSL_CertManagerEnableOCSP(WOLFSSL_CERT_MANAGER* cm,
                                                          int options);

int wolfSSL_CertManagerDisableOCSP(WOLFSSL_CERT_MANAGER*);

int wolfSSL_CertManagerSetOCSPOverrideURL(WOLFSSL_CERT_MANAGER* cm,
                                          const char* url);

int wolfSSL_CertManagerSetOCSP_Cb(WOLFSSL_CERT_MANAGER* cm,
                                  CbOCSPIO ioCb, CbOCSPRespFree respFreeCb,
                                  void* ioCbCtx);

int wolfSSL_CertManagerEnableOCSPStapling(
                                                    WOLFSSL_CERT_MANAGER* cm);

int wolfSSL_EnableCRL(WOLFSSL* ssl, int options);

int wolfSSL_DisableCRL(WOLFSSL* ssl);

int wolfSSL_LoadCRL(WOLFSSL* ssl, const char* path, int type, int monitor);

int wolfSSL_SetCRL_Cb(WOLFSSL* ssl, CbMissingCRL cb);

int wolfSSL_EnableOCSP(WOLFSSL* ssl, int options);

int wolfSSL_DisableOCSP(WOLFSSL*);

int wolfSSL_SetOCSP_OverrideURL(WOLFSSL* ssl, const char* url);

int wolfSSL_SetOCSP_Cb(WOLFSSL* ssl, CbOCSPIO ioCb, CbOCSPRespFree respFreeCb,
                       void* ioCbCtx);

int wolfSSL_CTX_EnableCRL(WOLFSSL_CTX* ctx, int options);

int wolfSSL_CTX_DisableCRL(WOLFSSL_CTX* ctx);
```

```
int wolfSSL_CTX_LoadCRL(WOLFSSL_CTX* ctx, const char* path, int type, int
 ↪ monitor);

int wolfSSL_CTX_SetCRL_Cb(WOLFSSL_CTX* ctx, CbMissingCRL cb);

int wolfSSL_CTX_EnableOCSP(WOLFSSL_CTX* ctx, int options);

int wolfSSL_CTX_DisableOCSP(WOLFSSL_CTX*);

int wolfSSL_CTX_SetOCSP_OverrideURL(WOLFSSL_CTX* ctx, const char* url);

int wolfSSL_CTX_SetOCSP_Cb(WOLFSSL_CTX* ctx,
                           CbOCSPIO ioCb, CbOCSPRespFree respFreeCb,
                           void* ioCbCtx);

int wolfSSL_CTX_EnableOCSPStapling(WOLFSSL_CTX*);

void wolfSSL_KeepArrays(WOLFSSL*);

void wolfSSL_FreeArrays(WOLFSSL*);

int wolfSSL_UseSNI(WOLFSSL* ssl, unsigned char type,
                                        const void* data, unsigned short size);

int wolfSSL_CTX_UseSNI(WOLFSSL_CTX* ctx, unsigned char type,
                                        const void* data, unsigned short size);

void wolfSSL_SNI_SetOptions(WOLFSSL* ssl, unsigned char type,
                                                    unsigned char options);

void wolfSSL_CTX_SNI_SetOptions(WOLFSSL_CTX* ctx,
                                    unsigned char type, unsigned char options);

int wolfSSL_SNI_GetFromBuffer(
                 const unsigned char* clientHello, unsigned int helloSz,
                unsigned char type, unsigned char* sni, unsigned int* inOutSz);

unsigned char wolfSSL_SNI_Status(WOLFSSL* ssl, unsigned char type);

unsigned short wolfSSL_SNI_GetRequest(WOLFSSL *ssl,
                                         unsigned char type, void** data);

int wolfSSL_UseALPN(WOLFSSL* ssl, char *protocol_name_list,
                                unsigned int protocol_name_listSz,
                                unsigned char options);

int wolfSSL_ALPN_GetProtocol(WOLFSSL* ssl, char **protocol_name,
                                      unsigned short *size);

int wolfSSL_ALPN_GetPeerProtocol(WOLFSSL* ssl, char **list,
                                         unsigned short *listSz);

int wolfSSL_UseMaxFragment(WOLFSSL* ssl, unsigned char mfl);
```

```
int wolfSSL_CTX_UseMaxFragment(WOLFSSL_CTX* ctx, unsigned char mfl);

int wolfSSL_UseTruncatedHMAC(WOLFSSL* ssl);

int wolfSSL_CTX_UseTruncatedHMAC(WOLFSSL_CTX* ctx);

int wolfSSL_UseOCSPStapling(WOLFSSL* ssl,
                            unsigned char status_type, unsigned char options);

int wolfSSL_CTX_UseOCSPStapling(WOLFSSL_CTX* ctx,
                            unsigned char status_type, unsigned char options);

int wolfSSL_UseOCSPStaplingV2(WOLFSSL* ssl,
                            unsigned char status_type, unsigned char options);

int wolfSSL_CTX_UseOCSPStaplingV2(WOLFSSL_CTX* ctx,
                            unsigned char status_type, unsigned char options);

int wolfSSL_UseSupportedCurve(WOLFSSL* ssl, word16 name);

int wolfSSL_CTX_UseSupportedCurve(WOLFSSL_CTX* ctx,
                                                        word16 name);

int wolfSSL_UseSecureRenegotiation(WOLFSSL* ssl);

int wolfSSL_Rehandshake(WOLFSSL* ssl);

int wolfSSL_UseSessionTicket(WOLFSSL* ssl);

int wolfSSL_CTX_UseSessionTicket(WOLFSSL_CTX* ctx);

int wolfSSL_get_SessionTicket(WOLFSSL* ssl, unsigned char* buf, word32* bufSz);

int wolfSSL_set_SessionTicket(WOLFSSL* ssl, const unsigned char* buf,
                              word32 bufSz);

int wolfSSL_set_SessionTicket_cb(WOLFSSL* ssl,
                                 CallbackSessionTicket cb, void* ctx);

int wolfSSL_send_SessionTicket(WOLFSSL* ssl);

int wolfSSL_CTX_set_TicketEncCb(WOLFSSL_CTX* ctx,
                                            SessionTicketEncCb);

int wolfSSL_CTX_set_TicketHint(WOLFSSL_CTX* ctx, int);

int wolfSSL_CTX_set_TicketEncCtx(WOLFSSL_CTX* ctx, void*);

void* wolfSSL_CTX_get_TicketEncCtx(WOLFSSL_CTX* ctx);

int wolfSSL_SetHsDoneCb(WOLFSSL* ssl, HandShakeDoneCb cb, void* user_ctx);

int wolfSSL_PrintSessionStats(void);
```

```
int wolfSSL_get_session_stats(unsigned int* active,
                                        unsigned int* total,
                                        unsigned int* peak,
                                        unsigned int* maxSessions);

int wolfSSL_MakeTlsMasterSecret(unsigned char* ms, word32 msLen,
                                const unsigned char* pms, word32 pmsLen,
                               const unsigned char* cr, const unsigned char* sr,
                                int tls1_2, int hash_type);

int wolfSSL_DeriveTlsKeys(unsigned char* key_data, word32 keyLen,
                                const unsigned char* ms, word32 msLen,
                               const unsigned char* sr, const unsigned char* cr,
                                int tls1_2, int hash_type);

int wolfSSL_connect_ex(WOLFSSL* ssl, HandShakeCallBack hsCb,
                       TimeoutCallBack toCb, WOLFSSL_TIMEVAL timeout);

int wolfSSL_accept_ex(WOLFSSL* ssl, HandShakeCallBacki hsCb,
                      TimeoutCallBack toCb, WOLFSSL_TIMEVAL timeout);

long wolfSSL_BIO_set_fp(WOLFSSL_BIO *bio, XFILE fp, int c);

long wolfSSL_BIO_get_fp(WOLFSSL_BIO *bio, XFILE* fp);

int wolfSSL_check_private_key(const WOLFSSL* ssl);

int wolfSSL_X509_get_ext_by_NID(const WOLFSSL_X509* x509,
                                          int nid, int lastPos);

void* wolfSSL_X509_get_ext_d2i(const WOLFSSL_X509* x509,
                                             int nid, int* c, int* idx);

int wolfSSL_X509_digest(const WOLFSSL_X509* x509,
        const WOLFSSL_EVP_MD* digest, unsigned char* buf, unsigned int* len);

int wolfSSL_use_certificate(WOLFSSL* ssl, WOLFSSL_X509* x509);

int wolfSSL_use_certificate_ASN1(WOLFSSL* ssl, unsigned char* der,
                                                              int derSz);

int wolfSSL_use_PrivateKey(WOLFSSL* ssl, WOLFSSL_EVP_PKEY* pkey);

int wolfSSL_use_PrivateKey_ASN1(int pri, WOLFSSL* ssl,
                                         unsigned char* der, long derSz);

int wolfSSL_use_RSAPrivateKey_ASN1(WOLFSSL* ssl, unsigned char* der,
                                                        long derSz);

WOLFSSL_DH *wolfSSL_DSA_dup_DH(const WOLFSSL_DSA *r);

int wolfSSL_SESSION_get_master_key(const WOLFSSL_SESSION* ses,
        unsigned char* out, int outSz);
```

```
int wolfSSL_SESSION_get_master_key_length(const WOLFSSL_SESSION* ses);

void wolfSSL_CTX_set_cert_store(WOLFSSL_CTX* ctx,
                                                    WOLFSSL_X509_STORE* str);

WOLFSSL_X509* wolfSSL_d2i_X509_bio(WOLFSSL_BIO* bio, WOLFSSL_X509** x509);

WOLFSSL_X509_STORE* wolfSSL_CTX_get_cert_store(WOLFSSL_CTX* ctx);

size_t wolfSSL_BIO_ctrl_pending(WOLFSSL_BIO *b);

size_t wolfSSL_get_server_random(const WOLFSSL *ssl,
                                            unsigned char *out, size_t outlen);

size_t wolfSSL_get_client_random(const WOLFSSL* ssl,
                                             unsigned char* out, size_t outSz);

wc_pem_password_cb* wolfSSL_CTX_get_default_passwd_cb(WOLFSSL_CTX*
                                                                    ctx);

void *wolfSSL_CTX_get_default_passwd_cb_userdata(WOLFSSL_CTX *ctx);

WOLFSSL_X509 *wolfSSL_PEM_read_bio_X509_AUX
        (WOLFSSL_BIO *bp, WOLFSSL_X509 **x, wc_pem_password_cb *cb, void *u);

long wolfSSL_CTX_set_tmp_dh(WOLFSSL_CTX* ctx, WOLFSSL_DH* dh);

WOLFSSL_DSA *wolfSSL_PEM_read_bio_DSAparams(WOLFSSL_BIO *bp,
    WOLFSSL_DSA **x, wc_pem_password_cb *cb, void *u);

unsigned long wolfSSL_ERR_peek_last_error(void);

WOLF_STACK_OF(WOLFSSL_X509)* wolfSSL_get_peer_cert_chain(const WOLFSSL*);

long wolfSSL_CTX_clear_options(WOLFSSL_CTX* ctx, long opt);

int wolfSSL_set_jobject(WOLFSSL* ssl, void* objPtr);

void* wolfSSL_get_jobject(WOLFSSL* ssl);

int wolfSSL_set_msg_callback(WOLFSSL *ssl, SSL_Msg_Cb cb);

int wolfSSL_set_msg_callback_arg(WOLFSSL *ssl, void* arg);

char* wolfSSL_X509_get_next_altname(WOLFSSL_X509*);

WOLFSSL_ASN1_TIME* wolfSSL_X509_get_notBefore(WOLFSSL_X509*);

int  wolfSSL_connect(WOLFSSL* ssl);

int  wolfSSL_send_hrr_cookie(WOLFSSL* ssl,
    const unsigned char* secret, unsigned int secretSz);
```

```
int wolfSSL_disable_hrr_cookie(WOLFSSL* ssl);

int  wolfSSL_CTX_no_ticket_TLSv13(WOLFSSL_CTX* ctx);

int  wolfSSL_no_ticket_TLSv13(WOLFSSL* ssl);

int  wolfSSL_CTX_no_dhe_psk(WOLFSSL_CTX* ctx);

int  wolfSSL_no_dhe_psk(WOLFSSL* ssl);

int  wolfSSL_update_keys(WOLFSSL* ssl);

int  wolfSSL_key_update_response(WOLFSSL* ssl, int* required);

int  wolfSSL_CTX_allow_post_handshake_auth(WOLFSSL_CTX* ctx);

int  wolfSSL_allow_post_handshake_auth(WOLFSSL* ssl);

int  wolfSSL_request_certificate(WOLFSSL* ssl);

int  wolfSSL_CTX_set1_groups_list(WOLFSSL_CTX *ctx, char *list);

int  wolfSSL_set1_groups_list(WOLFSSL *ssl, char *list);

int  wolfSSL_preferred_group(WOLFSSL* ssl);

int  wolfSSL_CTX_set_groups(WOLFSSL_CTX* ctx, int* groups,
    int count);

int  wolfSSL_set_groups(WOLFSSL* ssl, int* groups, int count);

int  wolfSSL_connect_TLSv13(WOLFSSL*);

wolfSSL_accept_TLSv13(WOLFSSL* ssl);

int  wolfSSL_CTX_set_max_early_data(WOLFSSL_CTX* ctx,
    unsigned int sz);

int  wolfSSL_set_max_early_data(WOLFSSL* ssl, unsigned int sz);

int  wolfSSL_write_early_data(WOLFSSL* ssl, const void* data,
    int sz, int* outSz);

int  wolfSSL_read_early_data(WOLFSSL* ssl, void* data, int sz,
    int* outSz);

int wolfSSL_inject(WOLFSSL* ssl, const void* data, int sz);

void wolfSSL_CTX_set_psk_client_tls13_callback(WOLFSSL_CTX* ctx,
    wc_psk_client_tls13_callback cb);

void wolfSSL_set_psk_client_tls13_callback(WOLFSSL* ssl,
    wc_psk_client_tls13_callback cb);
```

```
void wolfSSL_CTX_set_psk_server_tls13_callback(WOLFSSL_CTX* ctx,
    wc_psk_server_tls13_callback cb);

void wolfSSL_set_psk_server_tls13_callback(WOLFSSL* ssl,
    wc_psk_server_tls13_callback cb);

int wolfSSL_UseKeyShare(WOLFSSL* ssl, word16 group);

int wolfSSL_NoKeyShares(WOLFSSL* ssl);

WOLFSSL_METHOD *wolfTLSv1_3_server_method_ex(void* heap);

WOLFSSL_METHOD *wolfTLSv1_3_client_method_ex(void* heap);

WOLFSSL_METHOD *wolfTLSv1_3_server_method(void);

WOLFSSL_METHOD *wolfTLSv1_3_client_method(void);

WOLFSSL_METHOD *wolfTLSv1_3_method_ex(void* heap);

WOLFSSL_METHOD *wolfTLSv1_3_method(void);

int wolfSSL_CTX_set_ephemeral_key(WOLFSSL_CTX* ctx, int keyAlgo, const char*
 ↪  key, unsigned int keySz, int format);

int wolfSSL_set_ephemeral_key(WOLFSSL* ssl, int keyAlgo, const char* key,
 ↪  unsigned int keySz, int format);

int wolfSSL_CTX_get_ephemeral_key(WOLFSSL_CTX* ctx, int keyAlgo,
    const unsigned char** key, unsigned int* keySz);

int wolfSSL_get_ephemeral_key(WOLFSSL* ssl, int keyAlgo,
    const unsigned char** key, unsigned int* keySz);

int wolfSSL_RSA_sign_generic_padding(int type, const unsigned char* m,
                                unsigned int mLen, unsigned char* sigRet,
                                unsigned int* sigLen, WOLFSSL_RSA* rsa,
                                int flag, int padding);
int wolfSSL_dtls13_has_pending_msg(WOLFSSL *ssl);

unsigned int wolfSSL_SESSION_get_max_early_data(const WOLFSSL_SESSION *s);

int wolfSSL_CRYPTO_get_ex_new_index(int, void*, void*, void*, void*);

int wolfSSL_CTX_set_client_cert_type(WOLFSSL_CTX* ctx, const char* buf, int
 ↪  len);

int wolfSSL_CTX_set_server_cert_type(WOLFSSL_CTX* ctx, const char* buf, int
 ↪  len);

int wolfSSL_set_client_cert_type(WOLFSSL* ssl, const char* buf, int len);

int wolfSSL_set_server_cert_type(WOLFSSL* ssl, const char* buf, int len);
```

```
int wolfSSL_get_negotiated_client_cert_type(WOLFSSL* ssl, int* tp);

int wolfSSL_get_negotiated_server_cert_type(WOLFSSL* ssl, int* tp);

int wolfSSL_dtls_cid_use(WOLFSSL* ssl);

int wolfSSL_dtls_cid_is_enabled(WOLFSSL* ssl);

int wolfSSL_dtls_cid_set(WOLFSSL* ssl, unsigned char* cid,
    unsigned int size);

int wolfSSL_dtls_cid_get_rx_size(WOLFSSL* ssl,
    unsigned int* size);

int wolfSSL_dtls_cid_get_rx(WOLFSSL* ssl, unsigned char* buffer,
    unsigned int bufferSz);

int wolfSSL_dtls_cid_get0_rx(WOLFSSL* ssl, unsigned char** cid);

int wolfSSL_dtls_cid_get_tx_size(WOLFSSL* ssl, unsigned int* size);

int wolfSSL_dtls_cid_get_tx(WOLFSSL* ssl, unsigned char* buffer,
    unsigned int bufferSz);

int wolfSSL_dtls_cid_get0_tx(WOLFSSL* ssl, unsigned char** cid);

const unsigned char* wolfSSL_dtls_cid_parse(const unsigned char* msg,
        unsigned int msgSz, unsigned int cidSz);

int wolfSSL_get_client_suites_sigalgs(const WOLFSSL* ssl,
        const byte** suites, word16* suiteSz,
        const byte** hashSigAlgo, word16* hashSigAlgoSz);

WOLFSSL_CIPHERSUITE_INFO wolfSSL_get_ciphersuite_info(byte first,
        byte second);

int wolfSSL_get_sigalg_info(byte first, byte second,
        int* hashAlgo, int* sigAlgo);
```

C.53 dox_comments/header_files/tfm.h

C.53.1 Functions

	Name
word32	**CheckRunTimeFastMath**(void)This function checks the runtime fastmath settings for the maximum size of an integer. It is important when a user is using a wolfCrypt library independently, as the FP_SIZE must match for each library in order for math to work correctly. This check is defined as CheckFastMathSettings(), which simply compares CheckRunTimeFastMath and FP_SIZE, returning 0 if there is a mismatch, or 1 if they match.

C.53.2 Functions Documentation

C.53.2.1 function CheckRunTimeFastMath

```
word32 CheckRunTimeFastMath(
    void
)
```

This function checks the runtime fastmath settings for the maximum size of an integer. It is important when a user is using a wolfCrypt library independently, as the FP_SIZE must match for each library in order for math to work correctly. This check is defined as CheckFastMathSettings(), which simply compares CheckRunTimeFastMath and FP_SIZE, returning 0 if there is a mismatch, or 1 if they match.

Parameters:

- **none** No parameters.

See: CheckRunTimeSettings

Return: FP_SIZE Returns FP_SIZE, corresponding to the max size available for the math library.

Example

```
if (CheckFastMathSettings() != 1) {
return err_sys("Build vs. runtime fastmath FP_MAX_BITS mismatch\n");
}
// This is converted by the preprocessor to:
// if ( (CheckRunTimeFastMath() == FP_SIZE) != 1) {
// and confirms that the fast math settings match
// the compile time settings
```

C.53.3 Source code

```
word32 CheckRunTimeFastMath(void);
```

C.54 dox_comments/header_files/types.h

C.54.1 Functions

	Name
void *	**XMALLOC**(size_t n, void * heap, int type)This is not actually a function, but rather a preprocessor macro, which allows the user to substitute in their own malloc, realloc, and free functions in place of the standard C memory functions. To use external memory functions, define XMALLOC_USER. This will cause the memory functions to be replaced by external functions of the form: extern void *XMALLOC(size_t n, void* heap, int type); extern void _XREALLOC(void *p, size_t n, void_ heap, int type); extern void XFREE(void* p, void *heap, int type); To use the basic C memory functions in place of wolfSSL_Malloc, wolfSSL_Realloc, wolfSSL_Free, define NO_WOLFSSL_MEMORY. This will replace the memory functions with: #define XMALLOC(s, h, t) ((void)h, (void)t, malloc((s))) #define XFREE(p, h, t) {void* xp = (p); if((xp)) free((xp));} #define XREALLOC(p, n, h, t) realloc((p), (n)) If none of these options are selected, the system will default to use the wolfSSL memory functions. A user can set custom memory functions through callback hooks, (see wolfSSL_Malloc, wolfSSL_Realloc, wolfSSL_Free). This option will replace the memory functions with: #define XMALLOC(s, h, t) ((void)h, (void)t, wolfSSL_Malloc((s))) #define XFREE(p, h, t) {void* xp = (p); if((xp)) wolfSSL_Free((xp));} #define XREALLOC(p, n, h, t) wolfSSL_Realloc((p), (n))

	Name
void *	**XREALLOC**(void * p, size_t n, void * heap, int type)This is not actually a function, but rather a preprocessor macro, which allows the user to substitute in their own malloc, realloc, and free functions in place of the standard C memory functions. To use external memory functions, define XMALLOC_USER. This will cause the memory functions to be replaced by external functions of the form: extern void *XMALLOC(size_t n, void* heap, int type); extern void _XREALLOC(void *p, size_t n, void_ heap, int type); extern void XFREE(void* p, void *heap, int type); To use the basic C memory functions in place of wolfSSL_Malloc, wolfSSL_Realloc, wolfSSL_Free, define NO_WOLFSSL_MEMORY. This will replace the memory functions with: #define XMALLOC(s, h, t) ((void)h, (void)t, malloc((s))) #define XFREE(p, h, t) {void* xp = (p); if((xp)) free((xp));} #define XREALLOC(p, n, h, t) realloc((p), (n)) If none of these options are selected, the system will default to use the wolfSSL memory functions. A user can set custom memory functions through callback hooks, (see wolfSSL_Malloc, wolfSSL_Realloc, wolfSSL_Free). This option will replace the memory functions with: #define XMALLOC(s, h, t) ((void)h, (void)t, wolfSSL_Malloc((s))) #define XFREE(p, h, t) {void* xp = (p); if((xp)) wolfSSL_Free((xp));} #define XREALLOC(p, n, h, t) wolfSSL_Realloc((p), (n))

COPYRIGHT ©2024 wolfSSL Inc.

	Name
void	**XFREE**(void * p, void * heap, int type)This is not actually a function, but rather a preprocessor macro, which allows the user to substitute in their own malloc, realloc, and free functions in place of the standard C memory functions. To use external memory functions, define XMALLOC_USER. This will cause the memory functions to be replaced by external functions of the form: extern void *XMALLOC(size_t n, void* heap, int type); extern void _XREALLOC(void *p, size_t n, void_ heap, int type); extern void XFREE(void* p, void *heap, int type); To use the basic C memory functions in place of wolfSSL_Malloc, wolfSSL_Realloc, wolfSSL_Free, define NO_WOLFSSL_MEMORY. This will replace the memory functions with: #define XMALLOC(s, h, t) ((void)h, (void)t, malloc((s))) #define XFREE(p, h, t) {void* xp = (p); if((xp)) free((xp));} #define XREALLOC(p, n, h, t) realloc((p), (n)) If none of these options are selected, the system will default to use the wolfSSL memory functions. A user can set custom memory functions through callback hooks, (see wolfSSL_Malloc, wolfSSL_Realloc, wolfSSL_Free). This option will replace the memory functions with: #define XMALLOC(s, h, t) ((void)h, (void)t, wolfSSL_Malloc((s))) #define XFREE(p, h, t) {void* xp = (p); if((xp)) wolfSSL_Free((xp));} #define XREALLOC(p, n, h, t) wolfSSL_Realloc((p), (n))
word32	**CheckRunTimeSettings**(void)This function checks the compile time class settings. It is important when a user is using a wolfCrypt library independently, as the settings must match between libraries for math to work correctly. This check is defined as CheckCtcSettings(), which simply compares CheckRunTimeSettings and CTC_SETTINGS, returning 0 if there is a mismatch, or 1 if they match.

C.54.2 Functions Documentation

C.54.2.1 function XMALLOC

```
void * XMALLOC(
    size_t n,
    void * heap,
    int type
)
```

This is not actually a function, but rather a preprocessor macro, which allows the user to substitute in their own malloc, realloc, and free functions in place of the standard C memory functions. To use ex-

ternal memory functions, define XMALLOC_USER. This will cause the memory functions to be replaced by external functions of the form: extern void *XMALLOC(size_t n, void* heap, int type); extern void _XREALLOC(void *p, size_t n, void_ heap, int type); extern void XFREE(void* p, void *heap, int type); To use the basic C memory functions in place of wolfSSL_Malloc, wolfSSL_Realloc, wolfSSL_Free, define NO_WOLFSSL_MEMORY. This will replace the memory functions with: #define XMALLOC(s, h, t) ((void)h, (void)t, malloc((s))) #define XFREE(p, h, t) {void* xp = (p); if((xp)) free((xp));} #define XREALLOC(p, n, h, t) realloc((p), (n)) If none of these options are selected, the system will default to use the wolfSSL memory functions. A user can set custom memory functions through callback hooks, (see wolfSSL_Malloc, wolfSSL_Realloc, wolfSSL_Free). This option will replace the memory functions with: #define XMALLOC(s, h, t) ((void)h, (void)t, wolfSSL_Malloc((s))) #define XFREE(p, h, t) {void* xp = (p); if((xp)) wolfSSL_Free((xp));} #define XREALLOC(p, n, h, t) wolfSSL_Realloc((p), (n))

Parameters:

- **s** size of memory to allocate
- **h** (used by custom XMALLOC function) pointer to the heap to use
- **t** memory allocation types for user hints. See enum in types.h

See:

- wolfSSL_Malloc
- wolfSSL_Realloc
- wolfSSL_Free
- wolfSSL_SetAllocators

Return:

- pointer Return a pointer to allocated memory on success
- NULL on failure

Example

```
int* tenInts = XMALLOC(sizeof(int)*10, NULL, DYNAMIC_TYPE_TMP_BUFFER);
if (tenInts == NULL) {
    // error allocating space
    return MEMORY_E;
}
```

C.54.2.2 function XREALLOC

```
void * XREALLOC(
    void * p,
    size_t n,
    void * heap,
    int type
)
```

This is not actually a function, but rather a preprocessor macro, which allows the user to substitute in their own malloc, realloc, and free functions in place of the standard C memory functions. To use external memory functions, define XMALLOC_USER. This will cause the memory functions to be replaced by external functions of the form: extern void *XMALLOC(size_t n, void* heap, int type); extern void _XREALLOC(void *p, size_t n, void_ heap, int type); extern void XFREE(void* p, void *heap, int type); To use the basic C memory functions in place of wolfSSL_Malloc, wolfSSL_Realloc, wolfSSL_Free, define NO_WOLFSSL_MEMORY. This will replace the memory functions with: #define XMALLOC(s, h, t) ((void)h, (void)t, malloc((s))) #define XFREE(p, h, t) {void* xp = (p); if((xp)) free((xp));} #define XREALLOC(p, n, h, t) realloc((p), (n)) If none of these options are selected, the system will default to use the wolfSSL memory functions. A user can set custom memory functions through callback hooks, (see wolfSSL_Malloc, wolfSSL_Realloc, wolfSSL_Free).

This option will replace the memory functions with: #define XMALLOC(s, h, t) ((void)h, (void)t, wolfSSL_Malloc((s))) #define XFREE(p, h, t) {void* xp = (p); if((xp)) wolfSSL_Free((xp));} #define XREALLOC(p, n, h, t) wolfSSL_Realloc((p), (n))

Parameters:

- **p** pointer to the address to reallocate
- **n** size of memory to allocate
- **h** (used by custom XREALLOC function) pointer to the heap to use
- **t** memory allocation types for user hints. See enum in types.h

See:

- wolfSSL_Malloc
- wolfSSL_Realloc
- wolfSSL_Free
- wolfSSL_SetAllocators

Return:

- Return a pointer to allocated memory on success
- NULL on failure

Example

```
int* tenInts = (int*)XMALLOC(sizeof(int)*10, NULL, DYNAMIC_TYPE_TMP_BUFFER);
int* twentyInts = (int*)XREALLOC(tenInts, sizeof(int)*20, NULL,
    DYNAMIC_TYPE_TMP_BUFFER);
```

C.54.2.3 function XFREE

```
void XFREE(
    void * p,
    void * heap,
    int type
)
```

This is not actually a function, but rather a preprocessor macro, which allows the user to substitute in their own malloc, realloc, and free functions in place of the standard C memory functions. To use external memory functions, define XMALLOC_USER. This will cause the memory functions to be replaced by external functions of the form: extern void *XMALLOC(size_t n, void* heap, int type); extern void _XREALLOC(void *p, size_t n, void_ heap, int type); extern void XFREE(void* p, void *heap, int type); To use the basic C memory functions in place of wolfSSL_Malloc, wolfSSL_Realloc, wolfSSL_Free, define NO_WOLFSSL_MEMORY. This will replace the memory functions with: #define XMALLOC(s, h, t) ((void)h, (void)t, malloc((s))) #define XFREE(p, h, t) {void* xp = (p); if((xp)) free((xp));} #define XREALLOC(p, n, h, t) realloc((p), (n)) If none of these options are selected, the system will default to use the wolfSSL memory functions. A user can set custom memory functions through callback hooks, (see wolfSSL_Malloc, wolfSSL_Realloc, wolfSSL_Free). This option will replace the memory functions with: #define XMALLOC(s, h, t) ((void)h, (void)t, wolfSSL_Malloc((s))) #define XFREE(p, h, t) {void* xp = (p); if((xp)) wolfSSL_Free((xp));} #define XREALLOC(p, n, h, t) wolfSSL_Realloc((p), (n))

Parameters:

- **p** pointer to the address to free
- **h** (used by custom XFREE function) pointer to the heap to use
- **t** memory allocation types for user hints. See enum in types.h

See:

- wolfSSL_Malloc
- wolfSSL_Realloc

- wolfSSL_Free
- wolfSSL_SetAllocators

Return: none No returns.

Example

```
int* tenInts = XMALLOC(sizeof(int) * 10, NULL, DYNAMIC_TYPE_TMP_BUFFER);
if (tenInts == NULL) {
    // error allocating space
    return MEMORY_E;
}
```

C.54.2.4 function CheckRunTimeSettings

```
word32 CheckRunTimeSettings(
    void
)
```

This function checks the compile time class settings. It is important when a user is using a wolfCrypt library independently, as the settings must match between libraries for math to work correctly. This check is defined as CheckCtcSettings(), which simply compares CheckRunTimeSettings and CTC_SETTINGS, returning 0 if there is a mismatch, or 1 if they match.

Parameters:

- **none** No Parameters.

See: CheckRunTimeFastMath

Return: settings Returns the runtime CTC_SETTINGS (Compile Time Settings)

Example

```
if (CheckCtcSettings() != 1) {
    return err_sys("Build vs. runtime math mismatch\n");
}
// This is converted by the preprocessor to:
// if ( (CheckCtcSettings() == CTC_SETTINGS) != 1) {
// and will compare whether the compile time class settings
// match the current settings
```

C.54.3 Source code

```
void* XMALLOC(size_t n, void* heap, int type);

void* XREALLOC(void *p, size_t n, void* heap, int type);

void XFREE(void *p, void* heap, int type);

word32 CheckRunTimeSettings(void);
```

C.55 dox_comments/header_files/wc_encrypt.h

C.55.1 Functions

	Name
int	**wc_AesCbcDecryptWithKey**(byte * out, const byte * in, word32 inSz, const byte * key, word32 keySz, const byte * iv)Decrypts a cipher from the input buffer in, and places the resulting plain text in the output buffer out using cipher block chaining with AES. This function does not require an AES structure to be initialized. Instead, it takes in a key and an iv (initialization vector) and uses these to initialize an AES object and then decrypt the cipher text.
int	**wc_Des_CbcDecryptWithKey**(byte * out, const byte * in, word32 sz, const byte * key, const byte * iv)This function decrypts the input ciphertext, in, and stores the resulting plaintext in the output buffer, out. It uses DES encryption with cipher block chaining (CBC) mode. This function is a substitute for wc_Des_CbcDecrypt, allowing the user to decrypt a message without directly instantiating a Des structure.
int	**wc_Des_CbcEncryptWithKey**(byte * out, const byte * in, word32 sz, const byte * key, const byte * iv)This function encrypts the input plaintext, in, and stores the resulting ciphertext in the output buffer, out. It uses DES encryption with cipher block chaining (CBC) mode. This function is a substitute for wc_Des_CbcEncrypt, allowing the user to encrypt a message without directly instantiating a Des structure.
int	**wc_Des3_CbcEncryptWithKey**(byte * out, const byte * in, word32 sz, const byte * key, const byte * iv)This function encrypts the input plaintext, in, and stores the resulting ciphertext in the output buffer, out. It uses Triple DES (3DES) encryption with cipher block chaining (CBC) mode. This function is a substitute for wc_Des3_CbcEncrypt, allowing the user to encrypt a message without directly instantiating a Des3 structure.
int	**wc_Des3_CbcDecryptWithKey**(byte * out, const byte * in, word32 sz, const byte * key, const byte * iv)This function decrypts the input ciphertext, in, and stores the resulting plaintext in the output buffer, out. It uses Triple Des (3DES) encryption with cipher block chaining (CBC) mode. This function is a substitute for wc_Des3_CbcDecrypt, allowing the user to decrypt a message without directly instantiating a Des3 structure.

C.55.2 Functions Documentation

C.55.2.1 function wc_AesCbcDecryptWithKey

```
int wc_AesCbcDecryptWithKey(
    byte * out,
    const byte * in,
    word32 inSz,
    const byte * key,
    word32 keySz,
    const byte * iv
)
```

Decrypts a cipher from the input buffer in, and places the resulting plain text in the output buffer out using cipher block chaining with AES. This function does not require an AES structure to be initialized. Instead, it takes in a key and an iv (initialization vector) and uses these to initialize an AES object and then decrypt the cipher text.

Parameters:

- **out** pointer to the output buffer in which to store the plain text of the decrypted message
- **in** pointer to the input buffer containing cipher text to be decrypted
- **inSz** size of input message
- **key** 16, 24, or 32 byte secret key for decryption
- **keySz** size of key used for decryption

See:

- wc_AesSetKey
- wc_AesSetIV
- wc_AesCbcEncrypt
- wc_AesCbcDecrypt

Return:

- 0 On successfully decrypting message
- BAD_ALIGN_E Returned on block align error
- BAD_FUNC_ARG Returned if key length is invalid or AES object is null during AesSetIV
- MEMORY_E Returned if WOLFSSL_SMALL_STACK is enabled and XMALLOC fails to instantiate an AES object.

Example

```
int ret = 0;
byte key[] = { some 16, 24, or 32 byte key };
byte iv[]  = { some 16 byte iv };
byte cipher[AES_BLOCK_SIZE * n]; //n being a positive integer making
cipher some multiple of 16 bytes
// fill cipher with cipher text
byte plain [AES_BLOCK_SIZE * n];
if ((ret = wc_AesCbcDecryptWithKey(plain, cipher, AES_BLOCK_SIZE, key,
AES_BLOCK_SIZE, iv)) != 0 ) {
// Decrypt Error
}
```

C.55.2.2 function wc_Des_CbcDecryptWithKey

```
int wc_Des_CbcDecryptWithKey(
    byte * out,
    const byte * in,
    word32 sz,
    const byte * key,
```

```
    const byte * iv
)
```

This function decrypts the input ciphertext, in, and stores the resulting plaintext in the output buffer, out. It uses DES encryption with cipher block chaining (CBC) mode. This function is a substitute for wc_Des_CbcDecrypt, allowing the user to decrypt a message without directly instantiating a Des structure.

Parameters:

- **out** pointer to the buffer in which to store the decrypted plaintext
- **in** pointer to the input buffer containing the encrypted ciphertext
- **sz** length of the ciphertext to decrypt
- **key** pointer to the buffer containing the 8 byte key to use for decryption
- **iv** pointer to the buffer containing the 8 byte iv to use for decryption. If no iv is provided, the iv defaults to 0

See: wc_Des_CbcDecrypt

Return:

- 0 Returned upon successfully decrypting the given ciphertext
- MEMORY_E Returned if there is an error allocating space for a Des structure

3

Example

```
int ret;
byte key[] = { // initialize with 8 byte key };
byte iv[]  = { // initialize with 8 byte iv };

byte cipher[]  = { // initialize with ciphertext };
byte decoded[sizeof(cipher)];

if ( wc_Des_CbcDecryptWithKey(decoded, cipher, sizeof(cipher), key,
iv) != 0) {
    // error decrypting message
}
```

C.55.2.3 function wc_Des_CbcEncryptWithKey

```
int wc_Des_CbcEncryptWithKey(
    byte * out,
    const byte * in,
    word32 sz,
    const byte * key,
    const byte * iv
)
```

This function encrypts the input plaintext, in, and stores the resulting ciphertext in the output buffer, out. It uses DES encryption with cipher block chaining (CBC) mode. This function is a substitute for wc_Des_CbcEncrypt, allowing the user to encrypt a message without directly instantiating a Des structure.

Parameters:

- **out** Final encrypted data
- **in** Data to be encrypted, must be padded to Des block size.
- **sz** Size of input buffer.

- **key** Pointer to the key to use for encryption.
- **iv** Initialization vector

See:

- wc_Des_CbcDecryptWithKey
- wc_Des_CbcEncrypt

Return:

- 0 Returned after successfully encrypting data.
- MEMORY_E Returned if there's an error allocating memory for a Des structure.
- <0 Returned on any error during encryption.

3

Example

```
byte key[] = { // initialize with 8 byte key };
byte iv[]  = { // initialize with 8 byte iv };
byte in[] = { // Initialize with plaintext };
byte out[sizeof(in)];
if ( wc_Des_CbcEncryptWithKey(&out, in, sizeof(in), key, iv) != 0)
{
    // error encrypting message
}
```

C.55.2.4 function wc_Des3_CbcEncryptWithKey

```
int wc_Des3_CbcEncryptWithKey(
    byte * out,
    const byte * in,
    word32 sz,
    const byte * key,
    const byte * iv
)
```

This function encrypts the input plaintext, in, and stores the resulting ciphertext in the output buffer, out. It uses Triple DES (3DES) encryption with cipher block chaining (CBC) mode. This function is a substitute for wc_Des3_CbcEncrypt, allowing the user to encrypt a message without directly instantiating a Des3 structure.

Parameters:

- **out** Final encrypted data
- **in** Data to be encrypted, must be padded to Des block size.
- **sz** Size of input buffer.
- **key** Pointer to the key to use for encryption.
- **iv** Initialization vector

See:

- wc_Des3_CbcDecryptWithKey
- wc_Des_CbcEncryptWithKey
- wc_Des_CbcDecryptWithKey

Return:

- 0 Returned after successfully encrypting data.
- MEMORY_E Returned if there's an error allocating memory for a Des structure.
- <0 Returned on any error during encryption.

3

Example

```
byte key[] = { // initialize with 8 byte key };
byte iv[]  = { // initialize with 8 byte iv };

byte in[] = { // Initialize with plaintext };
byte out[sizeof(in)];

if ( wc_Des3_CbcEncryptWithKey(&out, in, sizeof(in), key, iv) != 0)
{
    // error encrypting message
}
```

C.55.2.5 function wc_Des3_CbcDecryptWithKey

```
int wc_Des3_CbcDecryptWithKey(
    byte * out,
    const byte * in,
    word32 sz,
    const byte * key,
    const byte * iv
)
```

This function decrypts the input ciphertext, in, and stores the resulting plaintext in the output buffer, out. It uses Triple Des (3DES) encryption with cipher block chaining (CBC) mode. This function is a substitute for wc_Des3_CbcDecrypt, allowing the user to decrypt a message without directly instantiating a Des3 structure.

Parameters:

- **out** pointer to the buffer in which to store the decrypted plaintext
- **in** pointer to the input buffer containing the encrypted ciphertext
- **sz** length of the ciphertext to decrypt
- **key** pointer to the buffer containing the 24 byte key to use for decryption
- **iv** pointer to the buffer containing the 8 byte iv to use for decryption. If no iv is provided, the iv defaults to 0

See: wc_Des3_CbcDecrypt

Return:

- 0 Returned upon successfully decrypting the given ciphertext
- MEMORY_E Returned if there is an error allocating space for a Des structure

3

Example

```
int ret;
byte key[] = { // initialize with 24 byte key };
byte iv[]  = { // initialize with 8 byte iv };

byte cipher[]  = { // initialize with ciphertext };
byte decoded[sizeof(cipher)];

if ( wc_Des3_CbcDecryptWithKey(decoded, cipher, sizeof(cipher),
key, iv) != 0) {
```

```
    // error decrypting message
}
```

C.55.3 Source code

```
int  wc_AesCbcDecryptWithKey(byte* out, const byte* in, word32 inSz,
                                        const byte* key, word32 keySz,
                                        const byte* iv);

int  wc_Des_CbcDecryptWithKey(byte* out,
                                         const byte* in, word32 sz,
                                         const byte* key, const byte* iv);

int  wc_Des_CbcEncryptWithKey(byte* out,
                                         const byte* in, word32 sz,
                                         const byte* key, const byte* iv);

int  wc_Des3_CbcEncryptWithKey(byte* out,
                                          const byte* in, word32 sz,
                                          const byte* key, const byte* iv);

int  wc_Des3_CbcDecryptWithKey(byte* out,
                                         const byte* in, word32 sz,
                                         const byte* key, const byte* iv);
```

C.56 dox_comments/header_files/wc_port.h

C.56.1 Functions

	Name
int	**wolfCrypt_Init**(void)Used to initialize resources used by wolfCrypt.
int	**wolfCrypt_Cleanup**(void)Used to clean up resources used by wolfCrypt.

C.56.2 Functions Documentation

C.56.2.1 function wolfCrypt_Init

```
int wolfCrypt_Init(
    void
)
```

Used to initialize resources used by wolfCrypt.

Parameters:

- **none** No parameters.

See: wolfCrypt_Cleanup

Return:

- 0 upon success.

- <0 upon failure of init resources.

Example

```
...
if (wolfCrypt_Init() != 0) {
    WOLFSSL_MSG("Error with wolfCrypt_Init call");
}
```

C.56.2.2 function wolfCrypt_Cleanup

```
int wolfCrypt_Cleanup(
    void
)
```

Used to clean up resources used by wolfCrypt.

Parameters:

- **none** No parameters.

See: wolfCrypt_Init

Return:

- 0 upon success.
- <0 upon failure of cleaning up resources.

Example

```
...
if (wolfCrypt_Cleanup() != 0) {
    WOLFSSL_MSG("Error with wolfCrypt_Cleanup call");
}
```

C.56.3 Source code

```
int wolfCrypt_Init(void);

int wolfCrypt_Cleanup(void);
```

C.57 dox_comments/header_files/wolfio.h

C.57.1 Functions

	Name
int	**EmbedReceive**(WOLFSSL * ssl, char * buf, int sz, void * ctx)This function is the receive embedded callback.
int	**EmbedSend**(WOLFSSL * ssl, char * buf, int sz, void * ctx)This function is the send embedded callback.
int	**EmbedReceiveFrom**(WOLFSSL * ssl, char * buf, int sz, void *)This function is the receive embedded callback.

	Name
int	**EmbedSendTo**(WOLFSSL * ssl, char * buf, int sz, void * ctx)This function is the send embedded callback.
int	**EmbedGenerateCookie**(WOLFSSL * ssl, unsigned char * buf, int sz, void *)This function is the DTLS Generate Cookie callback.
void	**EmbedOcspRespFree**(void * ctx, byte * resp)This function frees the response buffer.
void	**wolfSSL_CTX_SetIORecv**(WOLFSSL_CTX * ctx, CallbackIORecv CBIORecv)This function registers a receive callback for wolfSSL to get input data. By default, wolfSSL uses EmbedReceive() as the callback which uses the system's TCP recv() function. The user can register a function to get input from memory, some other network module, or from anywhere. Please see the EmbedReceive() function in src/io.c as a guide for how the function should work and for error codes. In particular, IO_ERR_WANT_READ should be returned for non blocking receive when no data is ready.
void	**wolfSSL_SetIOReadCtx**(WOLFSSL * ssl, void * ctx)This function registers a context for the SSL session's receive callback function. By default, wolfSSL sets the file descriptor passed to wolfSSL_set_fd() as the context when wolfSSL is using the system's TCP library. If you've registered your own receive callback you may want to set a specific context for the session. For example, if you're using memory buffers the context may be a pointer to a structure describing where and how to access the memory buffers.
void	**wolfSSL_SetIOWriteCtx**(WOLFSSL * ssl, void * ctx)This function registers a context for the SSL session's send callback function. By default, wolfSSL sets the file descriptor passed to wolfSSL_set_fd() as the context when wolfSSL is using the system's TCP library. If you've registered your own send callback you may want to set a specific context for the session. For example, if you're using memory buffers the context may be a pointer to a structure describing where and how to access the memory buffers.
void *	**wolfSSL_GetIOReadCtx**(WOLFSSL * ssl)This function returns the IOCB_ReadCtx member of the WOLFSSL struct.
void *	**wolfSSL_GetIOWriteCtx**(WOLFSSL * ssl)This function returns the IOCB_WriteCtx member of the WOLFSSL structure.

	Name
void	**wolfSSL_SetIOReadFlags**(WOLFSSL * ssl, int flags)This function sets the flags for the receive callback to use for the given SSL session. The receive callback could be either the default wolfSSL EmbedReceive callback, or a custom callback specified by the user (see wolfSSL_CTX_SetIORecv). The default flag value is set internally by wolfSSL to the value of 0. The default wolfSSL receive callback uses the recv() function to receive data from the socket. From the recv() man page: “The flags argument to a recv() function is formed by or’ing one or more of the values: MSG_OOB process out-of-band data, MSG_PEEK peek at incoming message, MSG_WAITALL wait for full request or error. The MSG_OOB flag requests receipt of out-of-band data that would not be received in the normal data stream. Some protocols place expedited data at the head of the normal data queue, and thus this flag cannot be used with such protocols. The MSG_PEEK flag causes the receive operation to return data from the beginning of the receive queue without removing that data from the queue. Thus, a subsequent receive call will return the same data. The MSG_WAITALL flag requests that the operation block until the full request is satisfied. However, the call may still return less data than requested if a signal is caught, an error or disconnect occurs, or the next data to be received is of a different type than that returned.”

	Name
void	**wolfSSL_SetIOWriteFlags**(WOLFSSL * ssl, int flags)This function sets the flags for the send callback to use for the given SSL session. The send callback could be either the default wolfSSL EmbedSend callback, or a custom callback specified by the user (see wolfSSL_CTX_SetIOSend). The default flag value is set internally by wolfSSL to the value of 0. The default wolfSSL send callback uses the send() function to send data from the socket. From the send() man page: “The flags parameter may include one or more of the following: #define MSG_OOB 0x1 // process out_of_band data, #define MSG_DONTROUTE 0x4 // bypass routing, use direct interface. The flag MSG_OOB is used to send ‘out_of_band’ data on sockets that support this notion (e.g. SOCK_STREAM); the underlying protocol must also support ‘out-of-band’ data. MSG_DONTROUTE is usually used only by diagnostic or routing programs.”
void	**wolfSSL_SetIO_NetX**(WOLFSSL * ssl, NX_TCP_SOCKET * nxsocket, ULONG waitoption)This function sets the nxSocket and nxWait members of the nxCtx struct within the WOLFSSL structure.
void	**wolfSSL_CTX_SetGenCookie**(WOLFSSL_CTX * ctx, CallbackGenCookie cb)This function sets the callback for the CBIOCookie member of the WOLFSSL_CTX structure. The CallbackGenCookie type is a function pointer and has the signature: int (*CallbackGenCookie)(WOLFSSL* ssl, unsigned char* buf, int sz, void* ctx);.
void *	**wolfSSL_GetCookieCtx**(WOLFSSL * ssl)This function returns the IOCB_CookieCtx member of the WOLFSSL structure.
int	**wolfSSL_SetIO_ISOTP**(WOLFSSL * ssl, isotp_wolfssl_ctx * ctx, can_recv_fn recv_fn, can_send_fn send_fn, can_delay_fn delay_fn, word32 receive_delay, char * receive_buffer, int receive_buffer_size, void * arg)This function sets up the ISO-TP context if wolfSSL, for use when wolfSSL is compiled with WOLFSSL_ISOTP.
void	**wolfSSL_SSLDisableRead**(WOLFSSL * ssl)This function disables reading from the IO layer.
void	**wolfSSL_SSLEnableRead;.

C.57.2 Functions Documentation

C.57.2.1 function EmbedReceive

```
int EmbedReceive(
    WOLFSSL * ssl,
    char * buf,
    int sz,
    void * ctx
)
```

This function is the receive embedded callback.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **buf** a char pointer representation of the buffer.
- **sz** the size of the buffer.
- **ctx** a void pointer to user registered context. In the default case the ctx is a socket descriptor pointer.

See:

- EmbedSend
- wolfSSL_CTX_SetIORecv
- wolfSSL_SSLSetIORecv

Return:

- Success This function returns the number of bytes read.
- WOLFSSL_CBIO_ERR_WANT_READ returned with a “Would block” message if the last error was SOCKET_EWOULDBLCOK or SOCKET_EAGAIN.
- WOLFSSL_CBIO_ERR_TIMEOUT returned with a “Socket timeout” message.
- WOLFSSL_CBIO_ERR_CONN_RST returned with a “Connection reset” message if the last error was SOCKET_ECONNRESET.
- WOLFSSL_CBIO_ERR_ISR returned with a “Socket interrupted” message if the last error was SOCKET_EINTR.
- WOLFSSL_CBIO_ERR_WANT_READ returned with a “Connection refused” message if the last error was SOCKET_ECONNREFUSED.
- WOLFSSL_CBIO_ERR_CONN_CLOSE returned with a “Connection aborted” message if the last error was SOCKET_ECONNABORTED.
- WOLFSSL_CBIO_ERR_GENERAL returned with a “General error” message if the last error was not specified.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
WOLFSSL* ssl = wolfSSL_new(ctx);
char* buf;
int sz;
void* ctx;
int bytesRead = EmbedReceive(ssl, buf, sz, ctx);
if(bytesRead <= 0){
    // There were no bytes read. Failure case.
}
```

C.57.2.2 function EmbedSend

```
int EmbedSend(
    WOLFSSL * ssl,
    char * buf,
    int sz,
```

```
    void * ctx
)
```

This function is the send embedded callback.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **buf** a char pointer representing the buffer.
- **sz** the size of the buffer.
- **ctx** a void pointer to user registered context.

See:

- EmbedReceive
- wolfSSL_CTX_SetIOSend
- wolfSSL_SSLSetIOSend

Return:

- Success This function returns the number of bytes sent.
- WOLFSSL_CBIO_ERR_WANT_WRITE returned with a “Would block” message if the last error was SOCKET_EWOULDBLOCK or SOCKET_EAGAIN.
- WOLFSSL_CBIO_ERR_CONN_RST returned with a “Connection reset” message if the last error was SOCKET_ECONNRESET.
- WOLFSSL_CBIO_ERR_ISR returned with a “Socket interrupted” message if the last error was SOCKET_EINTR.
- WOLFSSL_CBIO_ERR_CONN_CLOSE returned with a “Socket EPIPE” message if the last error was SOCKET_EPIPE.
- WOLFSSL_CBIO_ERR_GENERAL returned with a “General error” message if the last error was not specified.

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
char* buf;
int sz;
void* ctx;
int dSent = EmbedSend(ssl, buf, sz, ctx);
if(dSent <= 0){
    // No byes sent. Failure case.
}
```

C.57.2.3 function EmbedReceiveFrom

```
int EmbedReceiveFrom(
    WOLFSSL * ssl,
    char * buf,
    int sz,
    void *
)
```

This function is the receive embedded callback.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **buf** a constant char pointer to the buffer.
- **sz** an int type representing the size of the buffer.
- **ctx** a void pointer to the WOLFSSL_CTX context.

See:

- EmbedSendTo
- wolfSSL_CTX_SetIORecv
- wolfSSL_SSLSetIORecv
- wolfSSL_dtls_get_current_timeout

Return:

- Success This function returns the nb bytes read if the execution was successful.
- WOLFSSL_CBIO_ERR_WANT_READ if the connection refused or if a 'would block' error was thrown in the function.
- WOLFSSL_CBIO_ERR_TIMEOUT returned if the socket timed out.
- WOLFSSL_CBIO_ERR_CONN_RST returned if the connection reset.
- WOLFSSL_CBIO_ERR_ISR returned if the socket was interrupted.
- WOLFSSL_CBIO_ERR_GENERAL returned if there was a general error.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
WOLFSSL* ssl = WOLFSSL_new(ctx);
char* buf;
int sz = sizeof(buf)/sizeof(char);
(void*)ctx;
…
int nb = EmbedReceiveFrom(ssl, buf, sz, ctx);
if(nb > 0){
    // nb is the number of bytes written and is positive
}
```

C.57.2.4 function EmbedSendTo

```
int EmbedSendTo(
    WOLFSSL * ssl,
    char * buf,
    int sz,
    void * ctx
)
```

This function is the send embedded callback.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **buf** a char pointer representing the buffer.
- **sz** the size of the buffer.
- **ctx** a void pointer to the user registered context. The default case is a WOLFSSL_DTLS_CTX structure.

See:

- EmbedReceiveFrom
- wolfSSL_CTX_SetIOSend
- wolfSSL_SSLSetIOSend

Return:

- Success This function returns the number of bytes sent.
- WOLFSSL_CBIO_ERR_WANT_WRITE returned with a "Would Block" message if the last error was either SOCKET_EWOULDBLOCK or SOCKET_EAGAIN error.

- WOLFSSL_CBIO_ERR_CONN_RST returned with a “Connection reset” message if the last error was SOCKET_ECONNRESET.
- WOLFSSL_CBIO_ERR_ISR returned with a “Socket interrupted” message if the last error was SOCKET_EINTR.
- WOLFSSL_CBIO_ERR_CONN_CLOSE returned with a “Socket EPIPE” message if the last error was WOLFSSL_CBIO_ERR_CONN_CLOSE.
- WOLFSSL_CBIO_ERR_GENERAL returned with a “General error” message if the last error was not specified.

Example

```
WOLFSSL* ssl;
…
char* buf;
int sz;
void* ctx;

int sEmbed = EmbedSendto(ssl, buf, sz, ctx);
if(sEmbed <= 0){
    // No bytes sent. Failure case.
}
```

C.57.2.5 function EmbedGenerateCookie

```
int EmbedGenerateCookie(
    WOLFSSL * ssl,
    unsigned char * buf,
    int sz,
    void *
)
```

This function is the DTLS Generate Cookie callback.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **buf** byte pointer representing the buffer. It is the destination from XMEMCPY().
- **sz** the size of the buffer.
- **ctx** a void pointer to user registered context.

See: wolfSSL_CTX_SetGenCookie

Return:

- Success This function returns the number of bytes copied into the buffer.
- GEN_COOKIE_E returned if the getpeername failed in EmbedGenerateCookie.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
byte buffer[BUFFER_SIZE];
int sz = sizeof(buffer)/sizeof(byte);
void* ctx;
…
int ret = EmbedGenerateCookie(ssl, buffer, sz, ctx);

if(ret > 0){
```

```
    // EmbedGenerateCookie code block for success
}
```

C.57.2.6 function EmbedOcspRespFree

```
void EmbedOcspRespFree(
    void * ctx,
    byte * resp
)
```

This function frees the response buffer.

Parameters:

- **ctx** a void pointer to heap hint.
- **resp** a byte pointer representing the response.

See:

- wolfSSL_CertManagerSetOCSP_Cb
- wolfSSL_CertManagerEnableOCSPStapling
- wolfSSL_CertManagerEnableOCSP

Return: none No returns.

Example

```
void* ctx;
byte* resp; // Response buffer.
…
EmbedOcspRespFree(ctx, resp);
```

C.57.2.7 function wolfSSL_CTX_SetIORecv

```
void wolfSSL_CTX_SetIORecv(
    WOLFSSL_CTX * ctx,
    CallbackIORecv CBIORecv
)
```

This function registers a receive callback for wolfSSL to get input data. By default, wolfSSL uses EmbedReceive() as the callback which uses the system's TCP recv() function. The user can register a function to get input from memory, some other network module, or from anywhere. Please see the EmbedReceive() function in src/io.c as a guide for how the function should work and for error codes. In particular, IO_ERR_WANT_READ should be returned for non blocking receive when no data is ready.

Parameters:

- **ctx** pointer to the SSL context, created with wolfSSL_CTX_new().
- **callback** function to be registered as the receive callback for the wolfSSL context, ctx. The signature of this function must follow that as shown above in the Synopsis section.

See:

- wolfSSL_CTX_SetIOSend
- wolfSSL_SetIOReadCtx
- wolfSSL_SetIOWriteCtx

Return: none no Returns.

Example

```
WOLFSSL_CTX* ctx = 0;
// Receive callback prototype
int MyEmbedReceive(WOLFSSL* ssl, char* buf, int sz, void* ctx);
// Register the custom receive callback with wolfSSL
wolfSSL_CTX_SetIORecv(ctx, MyEmbedReceive);
int MyEmbedReceive(WOLFSSL* ssl, char* buf, int sz, void* ctx)
{
    // custom EmbedReceive function
}
```

C.57.2.8 function wolfSSL_SetIOReadCtx

```
void wolfSSL_SetIOReadCtx(
    WOLFSSL * ssl,
    void * ctx
)
```

This function registers a context for the SSL session's receive callback function. By default, wolfSSL sets the file descriptor passed to wolfSSL_set_fd() as the context when wolfSSL is using the system's TCP library. If you've registered your own receive callback you may want to set a specific context for the session. For example, if you're using memory buffers the context may be a pointer to a structure describing where and how to access the memory buffers.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().
- **rctx** pointer to the context to be registered with the SSL session's (ssl) receive callback function.

See:

- wolfSSL_CTX_SetIORecv
- wolfSSL_CTX_SetIOSend
- wolfSSL_SetIOWriteCtx

Return: none No returns.

Example

```
int sockfd;
WOLFSSL* ssl = 0;
...
// Manually setting the socket fd as the receive CTX, for example
wolfSSL_SetIOReadCtx(ssl, &sockfd);
...
```

C.57.2.9 function wolfSSL_SetIOWriteCtx

```
void wolfSSL_SetIOWriteCtx(
    WOLFSSL * ssl,
    void * ctx
)
```

This function registers a context for the SSL session's send callback function. By default, wolfSSL sets the file descriptor passed to wolfSSL_set_fd() as the context when wolfSSL is using the system's TCP library. If you've registered your own send callback you may want to set a specific context for the session. For example, if you're using memory buffers the context may be a pointer to a structure describing where and how to access the memory buffers.

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().
- **wctx** pointer to the context to be registered with the SSL session's (ssl) send callback function.

See:

- wolfSSL_CTX_SetIORecv
- wolfSSL_CTX_SetIOSend
- wolfSSL_SetIOReadCtx

Return: none No returns.

Example

```
int sockfd;
WOLFSSL* ssl = 0;
...
// Manually setting the socket fd as the send CTX, for example
wolfSSL_SetIOWriteCtx(ssl, &sockfd);
...
```

C.57.2.10 function wolfSSL_GetIOReadCtx

```
void * wolfSSL_GetIOReadCtx(
    WOLFSSL * ssl
)
```

This function returns the IOCB_ReadCtx member of the WOLFSSL struct.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_GetIOWriteCtx
- wolfSSL_SetIOReadFlags
- wolfSSL_SetIOWriteCtx
- wolfSSL_SetIOReadCtx
- wolfSSL_CTX_SetIOSend

Return:

- pointer This function returns a void pointer to the IOCB_ReadCtx member of the WOLFSSL structure.
- NULL returned if the WOLFSSL struct is NULL.

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
void* ioRead;
...
ioRead = wolfSSL_GetIOReadCtx(ssl);
if(ioRead == NULL){
    // Failure case. The ssl object was NULL.
}
```

C.57.2.11 function wolfSSL_GetIOWriteCtx

```
void * wolfSSL_GetIOWriteCtx(
    WOLFSSL * ssl
)
```

This function returns the IOCB_WriteCtx member of the WOLFSSL structure.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_GetIOReadCtx
- wolfSSL_SetIOWriteCtx
- wolfSSL_SetIOReadCtx
- wolfSSL_CTX_SetIOSend

Return:

- pointer This function returns a void pointer to the IOCB_WriteCtx member of the WOLFSSL structure.
- NULL returned if the WOLFSSL struct is NULL.

Example

```
WOLFSSL* ssl;
void* ioWrite;
...
ioWrite = wolfSSL_GetIOWriteCtx(ssl);
if(ioWrite == NULL){
    // The function returned NULL.
}
```

C.57.2.12 function wolfSSL_SetIOReadFlags

```
void wolfSSL_SetIOReadFlags(
    WOLFSSL * ssl,
    int flags
)
```

This function sets the flags for the receive callback to use for the given SSL session. The receive callback could be either the default wolfSSL EmbedReceive callback, or a custom callback specified by the user (see wolfSSL_CTX_SetIORecv). The default flag value is set internally by wolfSSL to the value of 0. The default wolfSSL receive callback uses the recv() function to receive data from the socket. From the recv() man page: "The flags argument to a recv() function is formed by or'ing one or more of the values: MSG_OOB process out-of-band data, MSG_PEEK peek at incoming message, MSG_WAITALL wait for full request or error. The MSG_OOB flag requests receipt of out-of-band data that would not be received in the normal data stream. Some protocols place expedited data at the head of the normal data queue, and thus this flag cannot be used with such protocols. The MSG_PEEK flag causes the receive operation to return data from the beginning of the receive queue without removing that data from the queue. Thus, a subsequent receive call will return the same data. The MSG_WAITALL flag requests that the operation block until the full request is satisfied. However, the call may still return less data than requested if a signal is caught, an error or disconnect occurs, or the next data to be received is of a different type than that returned."

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().
- **flags** value of the I/O read flags for the specified SSL session (ssl).

See:

- wolfSSL_CTX_SetIORecv
- wolfSSL_CTX_SetIOSend
- wolfSSL_SetIOReadCtx

Return: none No returns.

Example

```
WOLFSSL* ssl = 0;
...
// Manually setting recv flags to 0
wolfSSL_SetIOReadFlags(ssl, 0);
...
```

C.57.2.13 function wolfSSL_SetIOWriteFlags

```
void wolfSSL_SetIOWriteFlags(
    WOLFSSL * ssl,
    int flags
)
```

This function sets the flags for the send callback to use for the given SSL session. The send callback could be either the default wolfSSL EmbedSend callback, or a custom callback specified by the user (see wolfSSL_CTX_SetIOSend). The default flag value is set internally by wolfSSL to the value of 0. The default wolfSSL send callback uses the send() function to send data from the socket. From the send() man page: "The flags parameter may include one or more of the following: #define MSG_OOB 0x1 // process out-of-band data, #define MSG_DONTROUTE 0x4 // bypass routing, use direct interface. The flag MSG_OOB is used to send 'out-of-band' data on sockets that support this notion (e.g. SOCK_STREAM); the underlying protocol must also support 'out-of-band' data. MSG_DONTROUTE is usually used only by diagnostic or routing programs."

Parameters:

- **ssl** pointer to the SSL session, created with wolfSSL_new().
- **flags** value of the I/O send flags for the specified SSL session (ssl).

See:

- wolfSSL_CTX_SetIORecv
- wolfSSL_CTX_SetIOSend
- wolfSSL_SetIOReadCtx

Return: none No returns.

Example

```
WOLFSSL* ssl = 0;
...
// Manually setting send flags to 0
wolfSSL_SetIOWriteFlags(ssl, 0);
...
```

C.57.2.14 function wolfSSL_SetIO_NetX

```
void wolfSSL_SetIO_NetX(
    WOLFSSL * ssl,
    NX_TCP_SOCKET * nxsocket,
    ULONG waitoption
)
```

This function sets the nxSocket and nxWait members of the nxCtx struct within the WOLFSSL structure.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

- **nxSocket** a pointer to type NX_TCP_SOCKET that is set to the nxSocket member of the nxCTX structure.
- **waitOption** a ULONG type that is set to the nxWait member of the nxCtx structure.

See:

- set_fd
- NetX_Send
- NetX_Receive

Return: none No returns.

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
NX_TCP_SOCKET* nxSocket;
ULONG waitOption;
…
if(ssl != NULL || nxSocket != NULL || waitOption <= 0){
wolfSSL_SetIO_NetX(ssl, nxSocket, waitOption);
} else {
    // You need to pass in good parameters.
}
```

C.57.2.15 function wolfSSL_CTX_SetGenCookie

```
void wolfSSL_CTX_SetGenCookie(
    WOLFSSL_CTX * ctx,
    CallbackGenCookie cb
)
```

This function sets the callback for the CBIOCookie member of the WOLFSSL_CTX structure. The CallbackGenCookie type is a function pointer and has the signature: int (*CallbackGenCookie)(WOLFSSL* ssl, unsigned char* buf, int sz, void* ctx);.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().
- **cb** a CallbackGenCookie type function pointer with the signature of CallbackGenCookie.

See: CallbackGenCookie

Return: none No returns.

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
…
int SetGenCookieCB(WOLFSSL* ssl, unsigned char* buf, int sz, void* ctx){
// Callback function body.
}
…
wolfSSL_CTX_SetGenCookie(ssl->ctx, SetGenCookieCB);
```

C.57.2.16 function wolfSSL_GetCookieCtx

```
void * wolfSSL_GetCookieCtx(
    WOLFSSL * ssl
)
```

This function returns the IOCB_CookieCtx member of the WOLFSSL structure.

Parameters:

- **ssl** a pointer to a WOLFSSL structure, created using wolfSSL_new().

See:

- wolfSSL_SetCookieCtx
- wolfSSL_CTX_SetGenCookie

Return:

- pointer The function returns a void pointer value stored in the IOCB_CookieCtx.
- NULL if the WOLFSSL struct is NULL

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
void* cookie;
...
cookie = wolfSSL_GetCookieCtx(ssl);
if(cookie != NULL){
// You have the cookie
}
```

C.57.2.17 function wolfSSL_SetIO_ISOTP

```
int wolfSSL_SetIO_ISOTP(
    WOLFSSL * ssl,
    isotp_wolfssl_ctx * ctx,
    can_recv_fn recv_fn,
    can_send_fn send_fn,
    can_delay_fn delay_fn,
    word32 receive_delay,
    char * receive_buffer,
    int receive_buffer_size,
    void * arg
)
```

This function sets up the ISO-TP context if wolfSSL, for use when wolfSSL is compiled with WOLFSSL_ISOTP.

Parameters:

- **ssl** the wolfSSL context
- **ctx** a user created ISOTP context which this function initializes
- **recv_fn** a user CAN bus receive callback
- **send_fn** a user CAN bus send callback
- **delay_fn** a user microsecond granularity delay function
- **receive_delay** a set amount of microseconds to delay each CAN bus packet
- **receive_buffer** a user supplied buffer to receive data, recommended that is allocated to ISOTP_DEFAULT_BUFFER_SIZE bytes
- **receive_buffer_size** - The size of receive_buffer
- **arg** an arbitrary pointer sent to recv_fn and send_fn

Return: 0 on success, WOLFSSL_CBIO_ERR_GENERAL on failure

Example

```
struct can_info can_con_info;
isotp_wolfssl_ctx isotp_ctx;
char *receive_buffer = malloc(ISOTP_DEFAULT_BUFFER_SIZE);
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(method);
WOLFSSL* ssl = wolfSSL_new(ctx);
...
wolfSSL_SetIO_ISOTP(ssl, &isotp_ctx, can_receive, can_send, can_delay, 0,
        receive_buffer, ISOTP_DEFAULT_BUFFER_SIZE, &can_con_info);
```

C.57.2.18 function wolfSSL_SSLDisableRead

```
void wolfSSL_SSLDisableRead(
    WOLFSSL * ssl
)
```

This function disables reading from the IO layer.

Parameters:

- **ssl** the wolfSSL context

See:

- wolfSSL_CTX_SetIORecv
- wolfSSL_SSLSetIORecv
- wolfSSL_SSLEnableRead

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(method);
WOLFSSL* ssl = wolfSSL_new(ctx);
wolfSSL_SSLDisableRead(ssl);
```

C.57.2.19 function wolfSSL_SSLEnableRead

```
void wolfSSL_SSLEnableRead(
    WOLFSSL * ssl
)
```

This function enables reading from the IO layer. Reading is enabled by default and should be used to undo wolfSSL_SSLDisableRead();.

Parameters:

- **ssl** the wolfSSL context

See:

- wolfSSL_CTX_SetIORecv
- wolfSSL_SSLSetIORecv
- wolfSSL_SSLEnableRead

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(method);
WOLFSSL* ssl = wolfSSL_new(ctx);
wolfSSL_SSLDisableRead(ssl);
...
wolfSSL_SSLEnableRead(ssl);
```

C.57.3 Source code

```
int EmbedReceive(WOLFSSL* ssl, char* buf, int sz, void* ctx);

int EmbedSend(WOLFSSL* ssl, char* buf, int sz, void* ctx);

int EmbedReceiveFrom(WOLFSSL* ssl, char* buf, int sz, void*);

int EmbedSendTo(WOLFSSL* ssl, char* buf, int sz, void* ctx);

int EmbedGenerateCookie(WOLFSSL* ssl, unsigned char* buf,
                                          int sz, void*);

void EmbedOcspRespFree(void* ctx, byte* resp);

void wolfSSL_CTX_SetIORecv(WOLFSSL_CTX* ctx, CallbackIORecv CBIORecv);

void wolfSSL_SetIOReadCtx( WOLFSSL* ssl, void *ctx);

void wolfSSL_SetIOWriteCtx(WOLFSSL* ssl, void *ctx);

void* wolfSSL_GetIOReadCtx( WOLFSSL* ssl);

void* wolfSSL_GetIOWriteCtx(WOLFSSL* ssl);

void wolfSSL_SetIOReadFlags( WOLFSSL* ssl, int flags);

void wolfSSL_SetIOWriteFlags(WOLFSSL* ssl, int flags);

void wolfSSL_SetIO_NetX(WOLFSSL* ssl, NX_TCP_SOCKET* nxsocket,
                                      ULONG waitoption);

void  wolfSSL_CTX_SetGenCookie(WOLFSSL_CTX* ctx, CallbackGenCookie cb);

void* wolfSSL_GetCookieCtx(WOLFSSL* ssl);


int wolfSSL_SetIO_ISOTP(WOLFSSL *ssl, isotp_wolfssl_ctx *ctx,
        can_recv_fn recv_fn, can_send_fn send_fn, can_delay_fn delay_fn,
        word32 receive_delay, char *receive_buffer, int receive_buffer_size,
        void *arg);

void wolfSSL_SSLDisableRead(WOLFSSL *ssl);

void wolfSSL_SSLEnableRead(WOLFSSL *ssl);
```

D SSL/TLS Overview

D.1 General Architecture

The wolfSSL (formerly CyaSSL) embedded SSL library implements SSL 3.0, TLS 1.0, TLS 1.1, TLS 1.2, and TLS 1.3 protocols. TLS 1.3 is currently the most secure and up to date version of the standard. wolfSSL does not support SSL 2.0 due to the fact that it has been insecure for several years.

The TLS protocol in wolfSSL is implemented as defined in RFC 5246 (https://tools.ietf.org/html/rfc5246). Two record layer protocols exist within SSL - the message layer and the handshake layer. Handshake messages are used to negotiate a common cipher suite, create secrets, and enable a secure connection. The message layer encapsulates the handshake layer while also supporting alert processing and application data transfer.

A general diagram of how the SSL protocol fits into existing protocols can be seen in **Figure 1**. SSL sits in between the Transport and Application layers of the OSI model, where any number of protocols (including TCP/IP, Bluetooth, etc.) may act as the transport medium. Application protocols are layered on top of SSL (such as HTTP, FTP, and SMTP).

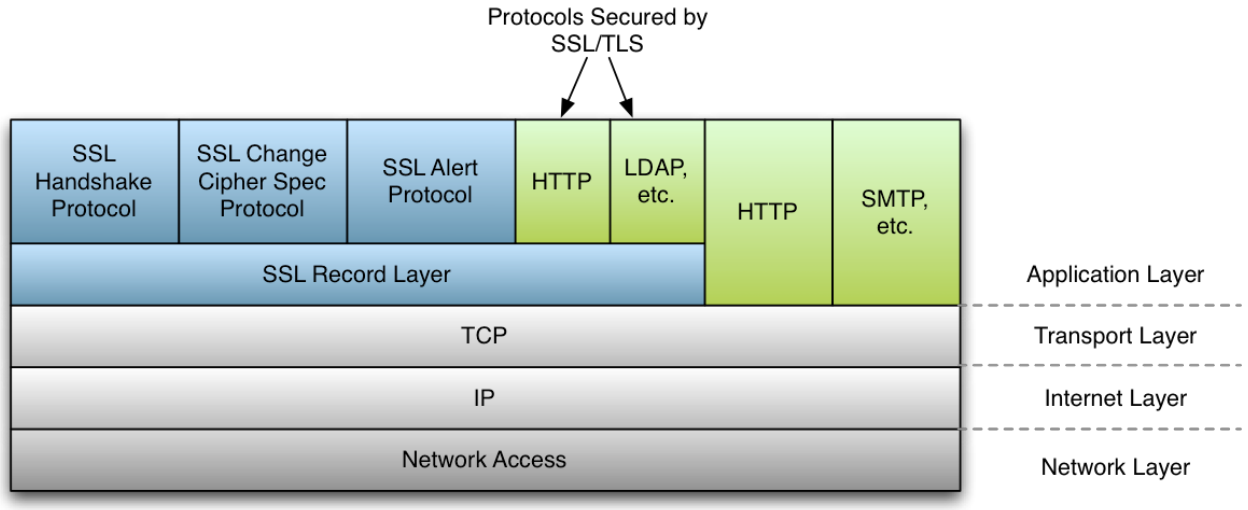

Figure 5: SSL Protocol Diagram

D.2 SSL Handshake

The SSL handshake involves several steps, some of which are optional depending on what options the SSL client and server have been configured with. Below, in **Figure 2**, you will find a simplified diagram of the SSL handshake process.

D.3 Differences between SSL and TLS Protocol Versions

SSL (Secure Sockets Layer) and TLS (Transport Security Layer) are both cryptographic protocols which provide secure communication over networks. These two protocols (and the several versions of each) are in widespread use today in applications ranging from web browsing to e-mail to instant messaging and VoIP. Each protocol, and the underlying versions of each, are slightly different from the other.

Below you will find both an explanation of, and the major differences between the different SSL and TLS protocol versions. For specific details about each protocol, please reference the RFC specification mentioned.

COPYRIGHT ©2024 wolfSSL Inc.

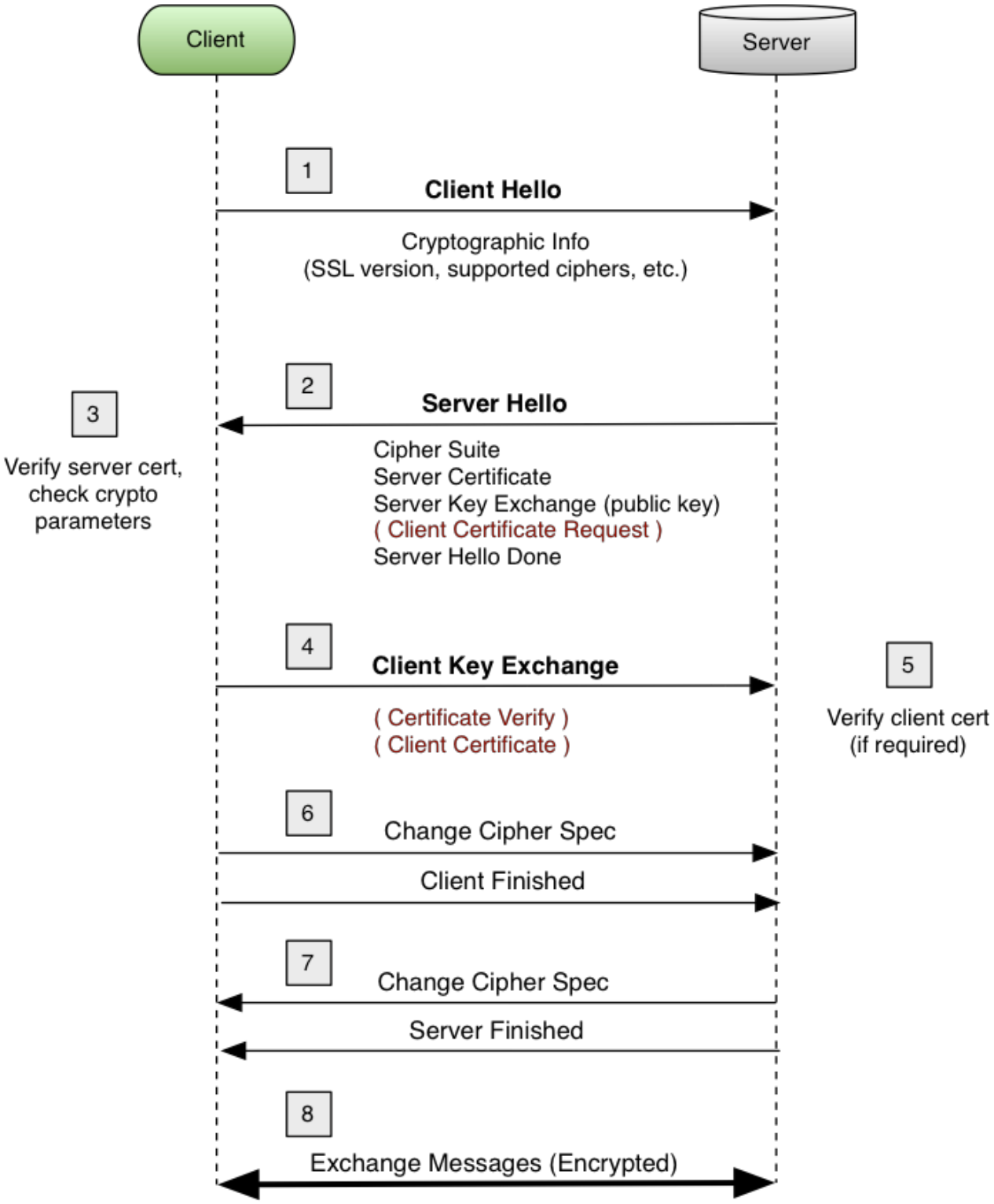

Figure 6: SSL Handshake Diagram

COPYRIGHT ©2024 wolfSSL Inc.

D.3.1 SSL 3.0

This protocol was released in 1996 but began with the creation of SSL 1.0 developed by Netscape. Version 1.0 wasn't released, and version 2.0 had a number of security flaws, thus leading to the release of SSL 3.0. Some major improvements of SSL 3.0 over SSL 2.0 are:

- Separation of the transport of data from the message layer
- Use of a full 128 bits of keying material even when using the Export cipher
- Ability of the client and server to send chains of certificates, thus allowing organizations to use certificate hierarchy which is more than two certificates deep.
- Implementing a generalized key exchange protocol, allowing Diffie-Hellman and Fortezza key exchanges as well as non-RSA certificates.
- Allowing for record compression and decompression
- Ability to fall back to SSL 2.0 when a 2.0 client is encountered

D.3.2 TLS 1.0

This protocol was first defined in RFC 2246 in January of 1999. This was an upgrade from SSL 3.0 and the differences were not dramatic, but they are significant enough that SSL 3.0 and TLS 1.0 don't interoperate. Some of the major differences between SSL 3.0 and TLS 1.0 are:

- Key derivation functions are different
- MACs are different - SSL 3.0 uses a modification of an early HMAC while TLS 1.0 uses HMAC.
- The Finished messages are different
- TLS has more alerts
- TLS requires DSS/DH support

D.3.3 TLS 1.1

This protocol was defined in RFC 4346 in April of 2006, and is an update to TLS 1.0. The major changes are:

- The Implicit Initialization Vector (IV) is replaced with an explicit IV to protect against Cipher block chaining (CBC) attacks.
- Handling of padded errors is changed to use the `bad_record_mac` alert rather than the `decryption_failed` alert to protect against CBC attacks.
- IANA registries are defined for protocol parameters
- Premature closes no longer cause a session to be non-resumable.

D.3.4 TLS 1.2

This protocol was defined in RFC 5246 in August of 2008. Based on TLS 1.1, TLS 1.2 contains improved flexibility. The major differences include:

- The MD5/SHA-1 combination in the pseudorandom function (PRF) was replaced with cipher-suite-specified PRFs.
- The MD5/SHA-1 combination in the digitally-signed element was replaced with a single hash. Signed elements include a field explicitly specifying the hash algorithm used.
- There was substantial cleanup to the client's and server's ability to specify which hash and signature algorithms they will accept.
- Addition of support for authenticated encryption with additional data modes.
- TLS Extensions definition and AES Cipher Suites were merged in.
- Tighter checking of EncryptedPreMasterSecret version numbers.
- Many of the requirements were tightened
- `Verify_data` length depends on the cipher suite
- Description of Bleichenbacher/Dlima attack defenses cleaned up.

COPYRIGHT ©2024 wolfSSL Inc.

D.3.5 TLS 1.3

This protocol was defined in RFC 8446 in August of 2018. TLS 1.3 contains improved security and speed. The major differences include:

- The list of supported symmetric algorithms has been pruned of all legacy algorithms. The remaining algorithms all use Authenticated Encryption with Associated Data (AEAD) algorithms.
- A zero-RTT (0-RTT) mode was added, saving a round-trip at connection setup for some application data at the cost of certain security properties.
- All handshake messages after the ServerHello are now encrypted.
- Key derivation functions have been re-designed, with the HMAC-based Extract-and-Expand Key Derivation Function (HKDF) being used as a primitive.
- The handshake state machine has been restructured to be more consistent and remove superfluous messages.
- ECC is now in the base spec and includes new signature algorithms. Point format negotiation has been removed in favor of single point format for each curve.
- Compression, custom DHE groups, and DSA have been removed, RSA padding now uses PSS.
- TLS 1.2 version negotiation verification mechanism was deprecated in favor of a version list in an extension.
- Session resumption with and without server-side state and the PSK-based ciphersuites of earlier versions of TLS have been replaced by a single new PSK exchange.

COPYRIGHT ©2024 wolfSSL Inc.

E RFCs, Specifications, and Reference

E.1 Protocols

- SSL v3.0 - IETF Draft
- TLS v1.0 - RFC2246
- TLS v1.1 - RFC4346
- TLS v1.2 - RFC5246
- TLS v1.3 - RFC8446
- DTLS - RFC4347 Specification document
- IPv4 - Wikipedia
- IPv6 - Wikipedia

E.2 Stream Ciphers

- Stream Cipher Information - Wikipedia
- RC4 / ARC4 - IETF Draft Wikipedia

E.3 Block Ciphers

- Block Cipher Information - Wikipedia
- AES - NIST Publication Wikipedia
- AES-GCM - NIST Specification
- AES-NI - Intel Sofware Network
- DES/3DES - NIST Publication Wikipedia

E.4 Hashing Functions

- SHA - NIST FIPS180-1 Publication NIST FIPS180-2 Publication Wikipedia
- MD4 - RFC1320
- MD5 - RFC1321
- RIPEMD-160 - Specification document

E.5 Public Key Cryptography

- Diffie-Hellman - Wikipedia
- RSA - MIT Paper Wikipedia
- DSA/DSS - NIST FIPS186-3
- ECDSA - Specification Document
- X.509 - RFC3279
- ASN.1 - Specification Document Wikipedia
- PSK - RFC4279

E.6 Other

- PKCS#5, PBKDF1, PBKDF2 - RFC2898
- PKCS#8 - RFC5208
- PKCS#12 - Wikipedia

COPYRIGHT ©2024 wolfSSL Inc.

F Error Codes

F.1 wolfSSL Error Codes

wolfSSL (formerly CyaSSL) error codes can be found in `wolfssl/ssl.h`. For detailed descriptions of the following errors, see the OpenSSL man page for `SSL_get_error` (`man SSL_get_error`).

Error Code Enum	Error Code	Error Description
SSL_ERROR_WANT_READ	2	
SSL_ERROR_WANT_WRITE	3	
SSL_ERROR_WANT_CONNECT	7	
SSL_ERROR_WANT_ACCEPT	8	
SSL_ERROR_SYSCALL	5	
SSL_ERROR_WANT_X509_LOOKUP	83	
SSL_ERROR_ZERO_RETURN	6	
SSL_ERROR_SSL	85	

Additional wolfSSL error codes can be found in `wolfssl/error-ssl.h`

Error Code Enum	Error Code	Error Description
INPUT_CASE_ERROR	-301	process input state error
PREFIX_ERROR	-302	bad index to key rounds
MEMORY_ERROR	-303	out of memory
VERIFY_FINISHED_ERROR	-304	verify problem on finished
VERIFY_MAC_ERROR	-305	verify mac problem
PARSE_ERROR	-306	parse error on header
UNKNOWN_HANDSHAKE_TYPE	-307	weird handshake type
SOCKET_ERROR_E	-308	error state on socket
SOCKET_NODATA	-309	expected data, not there
INCOMPLETE_DATA	-310	don't have enough data to complete task
UNKNOWN_RECORD_TYPE	-311	unknown type in record hdr
DECRYPT_ERROR	-312	error during decryption
FATAL_ERROR	-313	revcd alert fatal error
ENCRYPT_ERROR	-314	error during encryption
FREAD_ERROR	-315	fread problem
NO_PEER_KEY	-316	need peer's key
NO_PRIVATE_KEY	-317	need the private key
RSA_PRIVATE_ERROR	-318	error during rsa priv op
NO_DH_PARAMS	-319	server missing DH params
BUILD_MSG_ERROR	-320	build message failure
BAD_HELLO	-321	client hello malformed
DOMAIN_NAME_MISMATCH	-322	peer subject name mismatch
WANT_READ	-323	want read, call again
NOT_READY_ERROR	-324	handshake layer not ready
VERSION_ERROR	-326	record layer version error
WANT_WRITE	-327	want write, call again
BUFFER_ERROR	-328	malformed buffer input
VERIFY_CERT_ERROR	-329	verify cert error
VERIFY_SIGN_ERROR	-330	verify sign error
CLIENT_ID_ERROR	-331	psk client identity error
SERVER_HINT_ERROR	-332	psk server hint error

Error Code Enum	Error Code	Error Description
PSK_KEY_ERROR	-333	psk key error
GETTIME_ERROR	-337	gettimeofday failed ???
GETITIMER_ERROR	-338	getitimer failed ???
SIGACT_ERROR	-339	sigaction failed ???
SETITIMER_ERROR	-340	setitimer failed ???
LENGTH_ERROR	-341	record layer length error
PEER_KEY_ERROR	-342	cant decode peer key
ZERO_RETURN	-343	peer sent close notify
SIDE_ERROR	-344	wrong client/server type
NO_PEER_CERT	-345	peer didn't send key
ECC_CURVETYPE_ERROR	-350	Bad ECC Curve Type
ECC_CURVE_ERROR	-351	Bad ECC Curve
ECC_PEERKEY_ERROR	-352	Bad Peer ECC Key
ECC_MAKEKEY_ERROR	-353	Bad Make ECC Key
ECC_EXPORT_ERROR	-354	Bad ECC Export Key
ECC_SHARED_ERROR	-355	Bad ECC Shared Secret
NOT_CA_ERROR	-357	Not CA cert error
BAD_CERT_MANAGER_ERROR	-359	Bad Cert Manager
OCSP_CERT_REVOKED	-360	OCSP Certificate revoked
CRL_CERT_REVOKED	-361	CRL Certificate revoked
CRL_MISSING	-362	CRL Not loaded
MONITOR_SETUP_E	-363	CRL Monitor setup error
THREAD_CREATE_E	-364	Thread Create Error
OCSP_NEED_URL	-365	OCSP need an URL for lookup
OCSP_CERT_UNKNOWN	-366	OCSP responder doesn't know
OCSP_LOOKUP_FAIL	-367	OCSP lookup not successful
MAX_CHAIN_ERROR	-368	max chain depth exceeded
COOKIE_ERROR	-369	dtls cookie error
SEQUENCE_ERROR	-370	dtls sequence error
SUITES_ERROR	-371	suites pointer error
OUT_OF_ORDER_E	-373	out of order message
BAD_KEA_TYPE_E	-374	bad KEA type found
SANITY_CIPHER_E	-375	sanity check on cipher error
RECV_OVERFLOW_E	-376	RXCB returned more than rqed
GEN_COOKIE_E	-377	Generate Cookie Error
NO_PEER_VERIFY	-378	Need peer cert verify Error
FWRITE_ERROR	-379	fwrite problem
CACHE_MATCH_ERROR	-380	cache hrd match error
UNKNOWN_SNI_HOST_NAME_E	-381	Unrecognized host name Error
UNKNOWN_MAX_FRAG_LEN_E	-382	Unrecognized max frag len Error
KEYUSE_SIGNATURE_E	-383	KeyUse digSignature error
KEYUSE_ENCIPHER_E	-385	KeyUse KeyEncipher error
EXTKEYUSE_AUTH_E	-386	ExtKeyUse server
SEND_OOB_READ_E	-387	Send Cb out of bounds read
SECURE_RENEGOTIATION_E	-388	Invalid renegotiation info
SESSION_TICKET_LEN_E	-389	Session Ticket too large
SESSION_TICKET_EXPECT_E	-390	Session Ticket missing
SCR_DIFFERENT_CERT_E	-391	SCR Different cert error
SESSION_SECRET_CB_E	-392	Session secret CB fcn failure
NO_CHANGE_CIPHER_E	-393	Finished before change cipher
SANITY_MSG_E	-394	Sanity check on msg order error
DUPLICATE_MST_E	-395	Duplicate message error

Error Code Enum	Error Code	Error Description
SNI_UNSUPPORTED	-396	SSL 3.0 does not support SNI
SOCKET_PEER_CLOSED_E	-397	Underlying transport closed
BAD_TICKET_KEY_CB_SZ	-398	Bad session ticket key cb size
BAD_TICKET_MSG_SZ	-399	Bad session ticket msg size
BAD_TICKET_ENCRYPT	-400	Bad user ticket encrypt
DH_KEY_SIZE_E	-401	DH key too small
SNI_ABSENT_ERROR	-402	No SNI request
RSA_SIGN_FAULT	-403	RSA sign fault
HANDSHAKE_SIZE_ERROR	-404	Handshake message too large
UNKNOWN_ALPN_PROTOCOL_NAME_E	-405	Unrecognized protocol name error
BAD_CERTIFICATE_STATUS_ERROR	-406	Bad certificate status message
OCSP_INVALID_STATUS	-407	Invalid OCSP status
OCSP_WANT_READ	-408	OCSP callback response
RSA_KEY_SIZE_E	-409	RSA key too small
ECC_KEY_SIZE_E	-410	ECC key too small
DTLS_EXPORT_VER_E	-411	Export version error
INPUT_SIZE_E	-412	Input size too big error
CTX_INIT_MUTEX_E	-413	Initialize ctx mutex error
EXT_MASTER_SECRET_NEEDED_E	-414	Need EMS enabled to resume
DTLS_POOL_SZ_E	-415	Exceeded DTLS pool size
DECODE_E	-416	Decode handshake message error
HTTP_TIMEOUT	-417	HTTP timeout for OCSP or CRL req
WRITE_DUP_READ_E	-418	Write dup write side can't read
WRITE_DUP_WRITE_E	-419	Write dup read side can't write
INVALID_CERT_CTX_E	-420	TLS cert ctx not matching
BAD_KEY_SHARE_DATA	-421	Key share data invalid
MISSING_HANDSHAKE_DATA	-422	Handshake message missing data
BAD_BINDER	-423	Binder does not match
EXT_NOT_ALLOWED	-424	Extension not allowed in msg
INVALID_PARAMETER	-425	Security parameter invalid
MCAST_HIGHWATER_CB_E	-426	Multicast highwater cb err
ALERT_COUNT_E	-427	Alert count exceeded err
EXT_MISSING	-428	Required extension not found
UNSUPPORTED_EXTENSION	-429	TLSX not requested by client
PRF_MISSING	-430	PRF not compiled in
DTLS_RETX_OVER_TX	-431	Retransmit DTLS flight over
DH_PARAMS_NOT_FFDHE_E	-432	DH params from server not FFDHE
TCA_INVALID_ID_TYPE	-433	TLSX TCA ID type invalid
TCA_ABSENT_ERROR	-434	TLSX TCA ID no response

Negotiation Parameter Errors

Error Code Enum	Error Code	Error Description
UNSUPPORTED_SUITE	-500	Unsupported cipher suite
MATCH_SUITE_ERROR	-501	Can't match cipher suite
COMPRESSION_ERROR	-502	Compression mismatch
KEY_SHARE_ERROR	-503	Key share mismatch
POST_HAND_AUTH_ERROR	-504	Client won't do post-hand auth
HRR_COOKIE_ERROR	-505	HRR msg cookie mismatch

F.2 wolfCrypt Error Codes

wolfCrypt error codes can be found in `wolfssl/wolfcrypt/error.h`.

Error Code Enum	Error Code	Error Description
OPEN_RAN_E	-101	opening random device error
READ_RAN_E	-102	reading random device error
WINCRYPT_E	-103	windows crypt init error
CRYPTGEN_E	-104	windows crypt generation error
RAN_BLOCK_E	-105	reading random device would block
BAD_MUTEX_E	-106	Bad mutex operation
MP_INIT_E	-110	mp_init error state
MP_READ_E	-111	mp_read error state
MP_EXPTMOD_E	-112	mp_exptmod error state
MP_TO_E	-113	mp_to_xxx error state, can't convert
MP_SUB_E	-114	mp_sub error state, can't subtract
MP_ADD_E	-115	mp_add error state, can't add
MP_MUL_E	-116	mp_mul error state, can't multiply
MP_MULMOD_E	-117	mp_mulmod error state, can't multiply mod
MP_MOD_E	-118	mp_mod error state, can't mod
MP_INVMOD_E	-119	mp_invmod error state, can't inv mod
MP_CMP_E	-120	mp_cmp error state
MP_ZERO_E	-121	got a mp zero result, not expected
MEMORY_E	-125	out of memory error
RSA_WRONG_TYPE_E	-130	RSA wrong block type for RSA function
RSA_BUFFER_E	-131	RSA buffer error, output too small or input too large
BUFFER_E	-132	output buffer too small or input too large
ALGO_ID_E	-133	setting algo id error
PUBLIC_KEY_E	-134	setting public key error
DATE_E	-135	setting date validity error
SUBJECT_E	-136	setting subject name error
ISSUER_E	-137	setting issuer name error
CA_TRUE_E	-138	setting CA basic constraint true error
EXTENSIONS_E	-139	setting extensions error
ASN_PARSE_E	-140	ASN parsing error, invalid input
ASN_VERSION_E	-141	ASN version error, invalid number
ASN_GETINT_E	-142	ASN get big int error, invalid data
ASN_RSA_KEY_E	-143	ASN key init error, invalid input
ASN_OBJECT_ID_E	-144	ASN object id error, invalid id
ASN_TAG_NULL_E	-145	ASN tag error, not null
ASN_EXPECT_0_E	-146	ASN expect error, not zero
ASN_BITSTR_E	-147	ASN bit string error, wrong id
ASN_UNKNOWN_OID_E	-148	ASN oid error, unknown sum id
ASN_DATE_SZ_E	-149	ASN date error, bad size
ASN_BEFORE_DATE_E	-150	ASN date error, current date before
ASN_AFTER_DATE_E	-151	ASN date error, current date after
ASN_SIG_OID_E	-152	ASN signature error, mismatched oid
ASN_TIME_E	-153	ASN time error, unknown time type
ASN_INPUT_E	-154	ASN input error, not enough data
ASN_SIG_CONFIRM_E	-155	ASN sig error, confirm failure
ASN_SIG_HASH_E	-156	ASN sig error, unsupported hash type
ASN_SIG_KEY_E	-157	ASN sig error, unsupported key type
ASN_DH_KEY_E	-158	ASN key init error, invalid input
ASN_CRIT_EXT_E	-160	ASN unsupported critical extension

Error Code Enum	Error Code	Error Description
ECC_BAD_ARG_E	-170	ECC input argument of wrong type
ASN_ECC_KEY_E	-171	ASN ECC bad input
ECC_CURVE_OID_E	-172	Unsupported ECC OID curve type
BAD_FUNC_ARG	-173	Bad function argument provided
NOT_COMPILED_IN	-174	Feature not compiled in
UNICODE_SIZE_E	-175	Unicode password too big
NO_PASSWORD	-176	no password provided by user
ALT_NAME_E	-177	alt name size problem, too big
AES_GCM_AUTH_E	-180	AES-GCM Authentication check failure
AES_CCM_AUTH_E	-181	AES-CCM Authentication check failure
CAVIUM_INIT_E	-182	Cavium Init type error
COMPRESS_INIT_E	-183	Compress init error
COMPRESS_E	-184	Compress error
DECOMPRESS_INIT_E	-185	DeCompress init error
DECOMPRESS_E	-186	DeCompress error
BAD_ALIGN_E	-187	Bad alignment for operation, no alloc
ASN_NO_SIGNER_E	-188	ASN sig error, no CA signer to verify certificate
ASN_CRL_CONFIRM_E	-189	ASN CRL no signer to confirm failure
ASN_CRL_NO_SIGNER_E	-190	ASN CRL no signer to confirm failure
ASN_OCSP_CONFIRM_E	-191	ASN OCSP signature confirm failure
BAD_ENC_STATE_E	-192	Bad ecc enc state operation
BAD_PADDING_E	-193	Bad padding, msg not correct length
REQ_ATTRIBUTE_E	-194	Setting cert request attributes error
PKCS7_OID_E	-195	PKCS#7, mismatched OID error
PKCS7_RECIP_E	-196	PKCS#7, recipient error
FIPS_NOT_ALLOWED_E	-197	FIPS not allowed error
ASN_NAME_INVALID_E	-198	ASN name constraint error
RNG_FAILURE_E	-199	RNG Failed, Reinitialize
HMAC_MIN_KEYLEN_E	-200	FIPS Mode HMAC Minimum Key Length error
RSA_PAD_E	-201	RSA Padding Error
LENGTH_ONLY_E	-202	Returning output length only
IN_CORE_FIPS_E	-203	In Core Integrity check failure
AES_KAT_FIPS_E	-204	AES KAT failure
DES3_KAT_FIPS_E	-205	DES3 KAT failure
HMAC_KAT_FIPS_E	-206	HMAC KAT failure
RSA_KAT_FIPS_E	-207	RSA KAT failure
DRBG_KAT_FIPS_E	-208	HASH DRBG KAT failure
DRBG_CONT_FIPS_E	-209	HASH DRBG Continuous test failure
AESGCM_KAT_FIPS_E	-210	AESGCM KAT failure
THREAD_STORE_KEY_E	-211	Thread local storage key create failure
THREAD_STORE_SET_E	-212	Thread local storage key set failure
MAC_CMP_FAILED_E	-213	MAC comparison failed
IS_POINT_E	-214	ECC is point on curve failed
ECC_INF_E	-215	ECC point infinity error
ECC_PRIV_KEY_E	-216	ECC private key not valid error
SRP_CALL_ORDER_E	-217	SRP function called in the wrong order
SRP_VERIFY_E	-218	SRP proof verification failed
SRP_BAD_KEY_E	-219	SRP bad ephemeral values
ASN_NO_SKID	-220	ASN no Subject Key Identifier found
ASN_NO_AKID	-221	ASN no Authority Key Identifier found
ASN_NO_KEYUSAGE	-223	ASN no Key Usage found
SKID_E	-224	Setting Subject Key Identifier error

Error Code Enum	Error Code	Error Description
AKID_E	-225	Setting Authority Key Identifier error
KEYUSAGE_E	-226	Bad Key Usage value
CERTPOLICIES_E	-227	Setting Certificate Policies error
WC_INIT_E	-228	wolfCrypt failed to initialize
SIG_VERIFY_E	-229	wolfCrypt signature verify error
BAD_PKCS7_SIGNEEDS_CHECKCOND_E	-230	Bad condition variable operation
SIG_TYPE_E	-231	Signature Type not enabled/available
HASH_TYPE_E	-232	Hash Type not enabled/available
WC_KEY_SIZE_E	-234	Key size error, either too small or large
ASN_COUNTRY_SIZE_E	-235	ASN Cert Gen, invalid country code size
MISSING_RNG_E	-236	RNG required but not provided
ASN_PATHLEN_SIZE_E	-237	ASN CA path length too large error
ASN_PATHLEN_INV_E	-238	ASN CA path length inversion error
BAD_KEYWRAP_ALG_E	-239	Algorithm error with keywrap
BAD_KEYWRAP_IV_E	-240	Decrypted AES key wrap IV incorrect
WC_CLEANUP_E	-241	wolfCrypt cleanup failed
ECC_CDH_KAT_FIPS_E	-242	ECC CDH known answer test failure
DH_CHECK_PUB_E	-243	DH check public key error
BAD_PATH_ERROR	-244	Bad path for opendir
ASYNC_OP_E	-245	Async operation error
ECC_PRIVATEONLY_E	-246	Invalid use of private only ECC key
EXTKEYUSAGE_E	-247	Bad extended key usage value
WC_HW_E	-248	Error with hardware crypto use
WC_HW_WAIT_E	-249	Hardware waiting on resource
PSS_SALTLEN_E	-250	PSS length of salt is too long for hash
PRIME_GEN_E	-251	Failure finding a prime
BER_INDEF_E	-252	Cannot decode indefinite length BER
RSA_OUT_OF_RANGE_E	-253	Ciphertext to decrypt out of range
RSAPSS_PAT_FIPS_E	-254	RSA-PSS PAT failure
ECDSA_PAT_FIPS_E	-255	ECDSA PAT failure
DH_KAT_FIPS_E	-256	DH KAT failure
AESCCM_KAT_FIPS_E	-257	AESCCM KAT failure
SHA3_KAT_FIPS_E	-258	SHA-3 KAT failure
ECDHE_KAT_FIPS_E	-259	ECDHE KAT failure
AES_GCM_OVERFLOW_E	-260	AES-GCM invocation counter overflow
AES_CCM_OVERFLOW_E	-261	AES-CCM invocation counter overflow
RSA_KEY_PAIR_E	-262	RSA Key Pair-Wise consistency check fail
DH_CHECK_PRIVE_E	-263	DH check private key error
WC_AFALG_SOCK_E	-264	AF_ALG socket error
WC_DEVCRYPTO_E	-265	/dev/crypto error
ZLIB_INIT_ERROR	-266	Zlib init error
ZLIB_COMPRESS_ERROR	-267	Zlib compression error
ZLIB_DECOMPRESS_ERROR	-268	Zlib decompression error
PKCS7_NO_SIGNER_E	-269	No signer in PKCS7 signed data msg
WC_PKCS7_WANT_READ_E	-270	PKCS7 stream operation wants more input
CRYPTOCB_UNAVAILABLE	-271	Crypto callback unavailable
PKCS7_SIGNEEDS_CHECK	-272	Signature needs verified by caller
ASN_SELF_SIGNED_E	-275	ASN self-signed certificate error
MIN_CODE_E	-300	errors -101 - -299

F.3 Common Error Codes and their Solution

There are several error codes that commonly happen when getting an application up and running with wolfSSL.

F.3.1 ASN_NO_SIGNER_E (-188)

This error occurs when using a certificate and the signing CA certificate was not loaded. This can be seen using the wolfSSL example server or client against another client or server, for example connecting to Google using the wolfSSL example client:

```
./examples/client/client -g -h www.google.com -p 443
```

This fails with error -188 because Google's CA certificate wasn't loaded with the "-A" command line option.

F.3.2 WANT_READ (-323)

The WANT_READ error happens often when using non-blocking sockets, and isn't actually an error when using non-blocking sockets, but it is passed up to the caller as an error. When a call to receive data from the I/O callback would block as there isn't data currently available to receive, the I/O callback returns WANT_READ. The caller should wait and try receiving again later. This is usually seen from calls to wolfSSL_read(), wolfSSL_negotiate(), wolfSSL_accept(), and wolfSSL_connect(). The example client and server will indicate the WANT_READ incidents when debugging is enabled.

G Experimenting with Post-Quantum Cryptography

A while back, the wolfSSL team integrated experimental post-quantum cryptographic algorithms into the wolfSSL library. This was done by integrating with the Open Quantum Safe team's liboqs. Currently, wolfCrypt implements LMS, XMSS, ML-DSA and ML-KEM. So, for the purpose of code size reduction and ease of maintence, the wolfSSL team removed the integration with liboqs.

This appendix is intended for anyone that wants to start learning about post-quantum cryptography in the context of (D)TLS 1.3. It explains why post-quantum algorithms are important, what we have done in response to the quantum threat and how you can start experimenting with these new algorithms.

Note: Some of the post-quantum algorithms are not fully standardized yet. Some OIDs and codepoints are temporary and expected to change in the future. You should have no expectation of backwards compatibility until they are fully standardized.

G.1 A Gentle Introduction to Post-Quantum Cryptography

G.1.1 Why Post-Quantum Cryptography?

For some time now, many resources have been devoted to the development of quantum computers. So much so that commercialization of cloud quantum computing resources has already begun. While the current state of the art is still not in the realm of being cryptographically relevant, some threat models such as "harvest now, decrypt later" mean that preparations need to happen sooner than the appearance of cryptographically relevant quantum computers.

NIST is leading the way for standardization of a new class of algorithms designed to replace the public key cryptography algorithms that will become vulnerable to quantum computers. At the time of the writing of this passage, NIST has already standardized ML-DSA, ML-KEM, and SLH-DSA. Currently, standards organizations have various draft documents describing OIDs and codepoints. NIST is working on bringing these algorithms under the ubmbrella of the CMVP regulatory framework allowing for FIPS-140-3 validations of implementations of these algorithms.

G.1.2 How do we Protect Ourselves?

From a high level perspective, for every TLS 1.3 connection, authentication, integrity and confidentiality are the main security goals that protect each connection. Authentication is maintained via signature schemes such as ECDSA. Confidentiality and integrity are maintained by key establishment algorithms such as ECDHE and then using the established key with symmetric encryption algorithms such as AES to encrypt a communication stream. We can thus decompose the security of the TLS 1.3 protocol into 3 types of cryptographic algorithms:

- authentication algorithms
- key establishment algorithms
- symmetric cipher algorithms

The threat of quantum computers to conventional cryptography takes two forms. Grover's algorithm reduces the security of modern symmetric cipher algorithms by approximately half while Shor's algorithm completely breaks the security of modern authentication and key establishment algorithms. As a result, we can continue to protect our communications using the AES-256 symmetric cipher which is considered sufficiently secure even in the presence of a cryptographically relevant quantum computer. We can then replace our conventional authentication and key establishment algorithms with post-quantum algorithms. Note that during TLS 1.3 handshakes, the ciphersuite specifies the symmetric cipher to be used for the duration of the connection. Both CNSA (Commercial National Security Algorithm Suite) 1.0 and 2.0 prescribe using the AES_256_GCM_SHA384 ciphersuite. For key establishment and authentication, there are post-quantum KEMs (Key Encapsulation Mechanisms) and signature schemes.

COPYRIGHT ©2024 wolfSSL Inc.

These use different kinds of math from the conventional algorithms. They are designed specifically for resistance to quantum-computers. The authentication algorithm and KEM that NIST has standardized for use with network protocols are lattice-based algorithms.

- ML-DSA (Dilithium) Signature Scheme
- ML-KEM (KYBER) KEM

Note: SABER KEM and NTRU KEM were deprecated and removed as they did not move on to standardization.

Note: KYBER KEM 90s variants were deprecated and removed as NIST is not considering them for standardization.

Note: Dilithium Signature Scheme's AES variants were deprecated and removed as NIST is not considering them for standardization.

Note: When the liboqs integration was removed, we also removed the FALCON and SPHINCS+ signature schemes. We will have our own implmentations in the future.

An explanation of lattice-based cryptography would fall outside the scope of this document but more information about these algorithms can be found in their NIST submissions at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions.

Unfortunately, it might come as a shock, but we do not actually know that these algorithms will resist attacks from quantum computers. In fact, we do not even know that these algorithms are safe against a conventional computer. Its getting less and less likely, but someone could break lattice-based cryptography. However, as security experts will tell you, this is how cryptography has always worked. Algorithms are good when we start using them, but weaknesses and vulnerabilities are discovered and technology gets better. The post-quantum algorithms are somewhat problematic in that they are relatively new and could use a bit more attention from the community.

One solution is to not put our full faith into these new algorithms. For now, we can hedge our bets by hybridizing post-quantum algorithms with the conventional algorithms that we actually trust. ECC with NIST standardized curves seem like good candidates as we have to keep using them since FIPS 140-3 compliance is a priority. For this reason, we have not only implemented post-quantum KEMs but also hybridized them with ECDSA over NIST approved curves. Please see our list of hybrid groups below.

G.2 Getting Started with Post-Quantum algorithms in wolfSSL

The following instructions will get you started from a clean Linux development environment and lead you step by step to performing a quantum-safe TLS 1.3 connection.

G.2.1 Build Instructions

Please see the wolfSSL repo's INSTALL file (https://github.com/wolfSSL/wolfssl/blob/master/INSTALL). Item 15 has instructions on how to configure and build wolfSSL with ML-KEM and ML-DSA enabled.

You will need the patched OQS OpenSSL Provider fork in order to generate X.509 certificates with post-quantum cryptographic keys and signatures. Instructions can be found at https://github.com/wolfSSL/osp/tree/maste
For your convenience, pre-generated certificates can be found there as well.

G.2.2 Making a Quantum Safe TLS Connection

You can run the server and client like this in separate terminals:

```
$ examples/server/server -v 4 -l TLS_AES_256_GCM_SHA384 \
  -A certs/mldsa87_root_cert.pem \
  -c certs/mldsa44_entity_cert.pem \
```

```
    -k certs/mldsa44_entity_key.pem \
    --pqc P521_ML_KEM_1024

$ examples/client/client -v 4 -l TLS_AES_256_GCM_SHA384 \
    -A certs/mldsa44_root_cert.pem \
    -c certs/mldsa87_entity_cert.pem \
    -k certs/mldsa87_entity_key.pem \
    --pqc P521_ML_KEM_1024
```

You have just achieved a fully quantum-safe TLS 1.3 connection using AES-256 for symmetric encryption, the ML-DSA signature scheme for authentication and ECDHE hybridized with ML-KEM for key establishment.

Further information about other post-quantum examples can be found at https://github.com/wolfSSL/wolfssl-examples/blob/master/pq/README.md.

G.3 Post Quantum Algorithm Variant Names

Post-Quantum algorithm variant names:

NIST Security Level	PQC Variant Name
2	ML_DSA_44
3	ML_DSA_65
5	ML_DSA_87
1	ML_KEM_512
3	ML_KEM_768
5	ML_KEM_1024

Post-Quantum hybrid KEM names:

wolfSSL Variant Name	NIST ECC Curve and PQC Submission Variant Name
P256_ML_KEM_512	ECDSA P-256 and KYBER512
P384_ML_KEM_768	ECDSA P-384 and KYBER768
P521_ML_KEM_1024	ECDSA P-521 and KYBER1024

G.4 Cryptographic Artifact Sizes

All sizes are in bytes.

Post-Quantum Signature Scheme Artifact Sizes:

PQC Variant Name	Public Key Size	Private Key Size	Maximum Signature Size
ML_DSA_44	1312	2560	2420
ML_DSA_65	1952	4032	3309
ML_DSA_87	2592	4896	4627

Post-Quantum KEM Artifact Sizes:

PQC Variant Name	Public Key Size	Private Key Size	Ciphertext Size	Shared Secret Size
ML_KEM_512	800	1632	768	32
ML_KEM_768	1184	2400	1088	32
ML_KEM_1024	1568	3168	1568	32

G.5 Statistics

The following statistics and benchmarks were taken on an 11th GenIntel Core i7-1185G7@3-GHz with 8 cores running Ubuntu 22.04.5 LTS

wolfSSL:

```
./configure --enable-kyber \
            --enable-dilithium \
            --disable-psk \
            --disable-shared \
            --enable-intelasm \
            --enable-aesni \
            --enable-sp-math-all \
            --enable-sp-asm \
            CFLAGS="-Os"
```

Note: We are primarily benchmarking the post-quantum algorithms, but leave some conventional algorithms for comparison purposes.

G.5.1 Runtime Binary Sizes

The `tls_bench` example application binary file is 2498432 bytes after being built then stripped (Approximately 2.4M). Without `--enable-kyber --enable-dilithium` it is 2290912 bytes after being built then stripped (Approximately 2.2M). This is a difference of 207520 bytes (Approximately 200K).

G.5.2 TLS 1.3 Data Transmission Sizes

The following results were taken by running the example server and client and recording all information being transmitted via wireshark. This includes the TLS 1.3 handshake with mutual authentication, “hello wolfssl!” and “I hear you fa shizzle!” messages. The `tcp.len` of all packets were summed:

Ciphersuite	Authentication	Key Establishment	Total Bytes
TLS_AES_256_GCM_SHA384	RSA 2048 bit	ECC SECP256R1	5455
TLS_AES_256_GCM_SHA384	RSA 2048 bit	ML_KEM_512	6633
TLS_AES_256_GCM_SHA384	RSA 2048 bit	ML_KEM_768	7337
TLS_AES_256_GCM_SHA384	RSA 2048 bit	ML_KEM_1024	8201
TLS_AES_256_GCM_SHA384	RSA 2048 bit	P256_ML_KEM_512	6763
TLS_AES_256_GCM_SHA384	RSA 2048 bit	P384_ML_KEM_768	7531
TLS_AES_256_GCM_SHA384	RSA 2048 bit	P521_ML_KEM_1024	8467
TLS_AES_256_GCM_SHA384	ML_DSA_44	ECC SECP256R1	7918
TLS_AES_256_GCM_SHA384	ML_DSA_65	ECC SECP256R1	10233
TLS_AES_256_GCM_SHA384	ML_DSA_87	ECC SECP256R1	13477

COPYRIGHT ©2024 wolfSSL Inc.

G.5.3 Heap and Stack Usage

NOTE: This is out of date. These were obtained when wolfSSL was using the liboqs implementations of these algorithms. This is left here for historical purposes.

These statistics were obtained by adding the following configuration flags: `--enable-trackmemory --enable-stacksize`.

Memory use for server sign and client verify without server authentication of the client, TLS13-AES256-GCM-SHA384 ciphersuite and ECC SECP256R1 for key exchange.

```
Server FALCON_LEVEL1

stack used     =  48960
total Allocs   =    250
heap total     = 113548
heap peak      =  40990

Client FALCON_LEVEL1

stack used     =  29935
total Allocs   =    768
heap total     = 179427
heap peak      =  41765

Server FALCON_LEVEL5

stack used     =  89088
total Allocs   =    250
heap total     = 125232
heap peak      =  45630

Client FALCON_LEVEL5

stack used     =  29935
total Allocs   =    768
heap total     = 191365
heap peak      =  47469

Server DILITHIUM_LEVEL2

stack used = 56328
total   Allocs   =       243
total   Deallocs =       243
total   Bytes    =    128153
peak    Bytes    =     50250

Client DILITHIUM_LEVEL2

stack used = 30856
total   Allocs   =       805
total   Deallocs =       805
total   Bytes    =    206412
peak    Bytes    =     56299

Server DILITHIUM_LEVEL3
```

```
stack used = 86216
total   Allocs   =       243
total   Deallocs =       243
total   Bytes    =    140128
peak    Bytes    =     55161

Client DILITHIUM_LEVEL3

stack used = 33928
total   Allocs   =       805
total   Deallocs =       805
total   Bytes    =    220633
peak    Bytes    =     61245

Server DILITHIUM_LEVEL5

stack used = 119944
total   Allocs   =       243
total   Deallocs =       243
total   Bytes    =    152046
peak    Bytes    =     59829

Client DILITHIUM_LEVEL5

stack used = 40328
total   Allocs   =       805
total   Deallocs =       805
total   Bytes    =    238167
peak    Bytes    =     67049

Server RSA 2048

stack used     =  52896
total Allocs   =    253
heap total     = 121784
heap peak      =  39573

Client RSA 2048

stack used     =  54640
total Allocs   =    897
heap total     = 202472
heap peak      =  41760
```

Memory use for KEM groups. TLS13-AES256-GCM-SHA384 ciphersuite and RSA-2048 for client authentication of the server and without server authentication of the client.

```
Server KYBER_LEVEL1

stack used     = 52896
total Allocs   =   206
heap total     = 66864
heap peak      = 28474
```

```
Client KYBER_LEVEL1

stack used     =  54640
total Allocs   =    879
heap total     = 147235
heap peak      =  44538

Server KYBER_LEVEL3

stack used     =  52896
total Allocs   =    206
heap total     =  67888
heap peak      =  28794

Client KYBER_LEVEL3

stack used     =  54640
total Allocs   =    879
heap total     = 149411
heap peak      =  46010

Server KYBER_LEVEL5

stack used     =  52896
total Allocs   =    206
heap total     =  69232
heap peak      =  29274

Client KYBER_LEVEL5

stack used     =  54640
total Allocs   =    879
heap total     = 151907
heap peak      =  47642

Server KYBER_90S_LEVEL1

stack used     =  52896
total Allocs   =    206
heap total     =  66864
heap peak      =  28474

Client KYBER_90S_LEVEL1

stack used     =  54640
total Allocs   =    879
heap total     = 147235
heap peak      =  44538

Server KYBER_90S_LEVEL3

stack used     =  52896
total Allocs   =    206
heap total     =  67888
```

```
heap peak      =  28794

Client KYBER_90S_LEVEL3

stack used     =  54640
total Allocs   =    879
heap total     = 149411
heap peak      =  46010

Server KYBER_90S_LEVEL5

stack used     =  52896
total Allocs   =    206
heap total     =  69232
heap peak      =  29274

Client KYBER_90S_LEVEL5

stack used     =  54640
total Allocs   =    879
heap total     = 151907
heap peak      =  47642

Server P256_KYBER_LEVEL1

stack used     =  52896
total Allocs   =    223
heap total     = 118940
heap peak      =  37652

Client P256_KYBER_LEVEL1

stack used     =  54640
total Allocs   =    896
heap total     = 199376
heap peak      =  48932

Server P384_KYBER_LEVEL3

stack used     =  52896
total Allocs   =    223
heap total     = 120108
heap peak      =  38468

Client P384_KYBER_LEVEL3

stack used     =  54640
total Allocs   =    896
heap total     = 201728
heap peak      =  50468

Client Server P521_KYBER_LEVEL5

stack used     =  52896
```

```
total Allocs   =    223
heap total     = 121614
heap peak      =  39458

Client P521_KYBER_LEVEL5

stack used     =  54640
total Allocs   =    896
heap total     = 204422
heap peak      =  52172

Client Server P256_KYBER_90S_LEVEL1

stack used     =  52896
total Allocs   =    223
heap total     = 118940
heap peak      =  37652

Client P256_KYBER_90S_LEVEL1

stack used     =  54640
total Allocs   =    896
heap total     = 199376
heap peak      =  48932

Server P384_KYBER_90S_LEVEL3

stack used     =  52896
total Allocs   =    223
heap total     = 120108
heap peak      =  38468

Client P384_KYBER_90S_LEVEL3

stack used     =  54640
total Allocs   =    896
heap total     = 201728
heap peak      =  50468

Server P521_KYBER_90S_LEVEL5

stack used     =  52896
total Allocs   =    223
heap total     = 121614
heap peak      =  39458

Client P521_KYBER_90S_LEVEL5

stack used     =  54640
total Allocs   =    896
heap total     = 204422
heap peak      =  52172

Server ECDSA SECP256R1
```

```
stack used     =  52896
total Allocs   =    253
heap total     = 121784
heap peak      =  39573

Client ECDSA SECP256R1

stack used     =  54640
total Allocs   =    897
heap total     = 202472
heap peak      =  41760
```

G.5.4 Benchmarks

The following benchmarks were obtained with the following configuration flags:

```
./configure --enable-kyber \
            --enable-dilithium \
            --disable-shared \
            --enable-intelasm \
            --enable-aesni \
            --enable-sp \
            --enable-sp-math \
            --enable-sp-asm \
            CFLAGS="-Os -DECC_USER_CURVES -DHAVE_ECC256 -DHAVE_ECC384"
```

G.5.4.1 Benchmarks from wolfCrypt **Note**: Only a single core is used.

```
CPU: Intel x86_64 - avx1 avx2 rdrand rdseed bmi2 aesni adx movbe bmi1 sha
Math:   Multi-Precision: Disabled
    Single Precision: ecc 256 384 521 rsa/dh 2048 3072 4096 asm sp_x86_64.c

ECC    SECP256R1 key gen 95600 ops took 1.000 sec, avg 0.010 ms, 95587.830 ops
   /sec
ECDHE  SECP256R1 agree   24800 ops took 1.003 sec, avg 0.040 ms, 24737.512 ops
   /sec
ECDSA  SECP256R1 sign    61400 ops took 1.001 sec, avg 0.016 ms, 61337.775 ops
   /sec
ECDSA  SECP256R1 verify  23000 ops took 1.001 sec, avg 0.044 ms, 22976.012 ops
   /sec
ML-KEM 512       key gen 284600 ops took 1.000 sec, avg 0.004 ms, 284565.467
   ops/sec
ML-KEM 512       encap   270800 ops took 1.000 sec, avg 0.004 ms, 270749.585
   ops/sec
ML-KEM 512       decap   172900 ops took 1.000 sec, avg 0.006 ms, 172896.249
   ops/sec
ML-KEM 768       key gen 159800 ops took 1.000 sec, avg 0.006 ms, 159776.306
   ops/sec
ML-KEM 768       encap   152800 ops took 1.000 sec, avg 0.007 ms, 152765.071
   ops/sec
ML-KEM 768       decap   100100 ops took 1.000 sec, avg 0.010 ms, 100091.147
   ops/sec
```

```
ML-KEM 1024      key gen 108300 ops took 1.000 sec, avg 0.009 ms, 108277.024
   ops/sec
ML-KEM 1024      encap   104400 ops took 1.000 sec, avg 0.010 ms, 104388.900
   ops/sec
ML-KEM 1024      decap   74100 ops took 1.001 sec, avg 0.014 ms, 74057.147 ops
   /sec
ML-DSA 44        key gen 20700 ops took 1.004 sec, avg 0.049 ms, 20617.041 ops
   /sec
ML-DSA 44        sign    5100 ops took 1.019 sec, avg 0.200 ms, 5003.233 ops/
   sec
ML-DSA 44        verify  18500 ops took 1.005 sec, avg 0.054 ms, 18403.134 ops
   /sec
ML-DSA 65        key gen 10200 ops took 1.007 sec, avg 0.099 ms, 10133.468 ops
   /sec
ML-DSA 65        sign    2900 ops took 1.004 sec, avg 0.346 ms, 2887.112 ops/
   sec
ML-DSA 65        verify  11600 ops took 1.005 sec, avg 0.087 ms, 11544.122 ops
   /sec
ML-DSA 87        key gen 7700 ops took 1.013 sec, avg 0.132 ms, 7598.278 ops/
   sec
ML-DSA 87        sign    2600 ops took 1.000 sec, avg 0.385 ms, 2599.634 ops/
   sec
ML-DSA 87        verify  7200 ops took 1.007 sec, avg 0.140 ms, 7152.274 ops/
   sec
```

G.5.4.2 Benchmarks from wolfSSL The following benchmarks were obtained with the following configuration flags:

```
./configure --enable-kyber \
            --enable-dilithium \
            --disable-shared \
            --enable-intelasm \
            --enable-aesni \
            --enable-sp \
            --enable-sp-math \
            --enable-sp-asm \
            CFLAGS="-Os -DECC_USER_CURVES -DHAVE_ECC256"
```

Note: Only two cores are used for these benchmarks.

```
wolfSSL Server Benchmark on TLS13-AES128-GCM-SHA256 with group ECC_SECP256R1:
        Total       :   6029312 bytes
        Num Conns   :        24
        Rx Total    :   965.511 ms
        Tx Total    :     7.469 ms
        Rx          :     2.978 MB/s
        Tx          :   384.903 MB/s
        Connect     :    48.343 ms
        Connect Avg :     2.014 ms
wolfSSL Client Benchmark on TLS13-AES128-GCM-SHA256 with group ECC_SECP256R1:
        Total       :   6029312 bytes
        Num Conns   :        24
        Rx Total    :   967.748 ms
        Tx Total    :     6.789 ms
```

```
        Rx          :     2.971 MB/s
        Tx          :   423.496 MB/s
        Connect     :    48.574 ms
        Connect Avg :     2.024 ms

wolfSSL Server Benchmark on TLS13-AES128-GCM-SHA256 with group ECC_SECP384R1:
        Total       :   6029312 bytes
        Num Conns   :        24
        Rx Total    :   960.296 ms
        Tx Total    :     7.494 ms
        Rx          :     2.994 MB/s
        Tx          :   383.617 MB/s
        Connect     :    56.255 ms
        Connect Avg :     2.344 ms
wolfSSL Client Benchmark on TLS13-AES128-GCM-SHA256 with group ECC_SECP384R1:
        Total       :   6029312 bytes
        Num Conns   :        24
        Rx Total    :   962.002 ms
        Tx Total    :     7.367 ms
        Rx          :     2.989 MB/s
        Tx          :   390.259 MB/s
        Connect     :    56.220 ms
        Connect Avg :     2.343 ms

wolfSSL Server Benchmark on TLS13-AES128-GCM-SHA256 with group ECC_SECP521R1:
        Total       :   5767168 bytes
        Num Conns   :        23
        Rx Total    :   938.745 ms
        Tx Total    :     7.889 ms
        Rx          :     2.929 MB/s
        Tx          :   348.596 MB/s
        Connect     :    61.261 ms
        Connect Avg :     2.664 ms
wolfSSL Client Benchmark on TLS13-AES128-GCM-SHA256 with group ECC_SECP521R1:
        Total       :   5767168 bytes
        Num Conns   :        23
        Rx Total    :   940.382 ms
        Tx Total    :     7.540 ms
        Rx          :     2.924 MB/s
        Tx          :   364.711 MB/s
        Connect     :    61.433 ms
        Connect Avg :     2.671 ms

wolfSSL Server Benchmark on TLS13-AES128-GCM-SHA256 with group ML_KEM_512:
        Total       :   6029312 bytes
        Num Conns   :        24
        Rx Total    :   952.389 ms
        Tx Total    :     5.561 ms
        Rx          :     3.019 MB/s
        Tx          :   517.005 MB/s
        Connect     :    50.177 ms
        Connect Avg :     2.091 ms
wolfSSL Client Benchmark on TLS13-AES128-GCM-SHA256 with group ML_KEM_512:
        Total       :   6029312 bytes
```

```
        Num Conns   :        24
        Rx Total    :   954.202 ms
        Tx Total    :     4.751 ms
        Rx          :     3.013 MB/s
        Tx          :   605.110 MB/s
        Connect     :    48.602 ms
        Connect Avg :     2.025 ms

wolfSSL Server Benchmark on TLS13-AES128-GCM-SHA256 with group ML_KEM_768:
        Total       :   6029312 bytes
        Num Conns   :        24
        Rx Total    :   955.030 ms
        Tx Total    :     5.882 ms
        Rx          :     3.010 MB/s
        Tx          :   488.757 MB/s
        Connect     :    51.283 ms
        Connect Avg :     2.137 ms
wolfSSL Client Benchmark on TLS13-AES128-GCM-SHA256 with group ML_KEM_768:
        Total       :   6029312 bytes
        Num Conns   :        24
        Rx Total    :   955.658 ms
        Tx Total    :     6.200 ms
        Rx          :     3.008 MB/s
        Tx          :   463.686 MB/s
        Connect     :    49.717 ms
        Connect Avg :     2.072 ms

wolfSSL Server Benchmark on TLS13-AES128-GCM-SHA256 with group ML_KEM_1024:
        Total       :   6029312 bytes
        Num Conns   :        24
        Rx Total    :   973.042 ms
        Tx Total    :     7.294 ms
        Rx          :     2.955 MB/s
        Tx          :   394.150 MB/s
        Connect     :    51.750 ms
        Connect Avg :     2.156 ms
wolfSSL Client Benchmark on TLS13-AES128-GCM-SHA256 with group ML_KEM_1024:
        Total       :   6029312 bytes
        Num Conns   :        24
        Rx Total    :   973.655 ms
        Tx Total    :     7.996 ms
        Rx          :     2.953 MB/s
        Tx          :   359.573 MB/s
        Connect     :    50.328 ms
        Connect Avg :     2.097 ms

wolfSSL Server Benchmark on TLS13-AES128-GCM-SHA256 with group P256_ML_KEM_512
    :
        Total       :   6029312 bytes
        Num Conns   :        24
        Rx Total    :   961.483 ms
        Tx Total    :     7.430 ms
        Rx          :     2.990 MB/s
        Tx          :   386.966 MB/s
```

```
        Connect     :    55.885 ms
        Connect Avg :     2.329 ms
wolfSSL Client Benchmark on TLS13-AES128-GCM-SHA256 with group P256_ML_KEM_512
    :
        Total       :   6029312 bytes
        Num Conns   :        24
        Rx Total    :   963.042 ms
        Tx Total    :     7.088 ms
        Rx          :     2.985 MB/s
        Tx          :   405.605 MB/s
        Connect     :    53.236 ms
        Connect Avg :     2.218 ms

wolfSSL Server Benchmark on TLS13-AES128-GCM-SHA256 with group P384_ML_KEM_768
    :
        Total       :   5767168 bytes
        Num Conns   :        23
        Rx Total    :   927.519 ms
        Tx Total    :     7.338 ms
        Rx          :     2.965 MB/s
        Tx          :   374.747 MB/s
        Connect     :    64.464 ms
        Connect Avg :     2.803 ms
wolfSSL Client Benchmark on TLS13-AES128-GCM-SHA256 with group P384_ML_KEM_768
    :
        Total       :   5767168 bytes
        Num Conns   :        23
        Rx Total    :   929.281 ms
        Tx Total    :     6.923 ms
        Rx          :     2.959 MB/s
        Tx          :   397.229 MB/s
        Connect     :    60.200 ms
        Connect Avg :     2.617 ms

wolfSSL Server Benchmark on TLS13-AES128-GCM-SHA256 with group
   P521_ML_KEM_1024:
        Total       :   5767168 bytes
        Num Conns   :        23
        Rx Total    :   918.122 ms
        Tx Total    :     7.598 ms
        Rx          :     2.995 MB/s
        Tx          :   361.941 MB/s
        Connect     :    79.426 ms
        Connect Avg :     3.453 ms
wolfSSL Client Benchmark on TLS13-AES128-GCM-SHA256 with group
   P521_ML_KEM_1024:
        Total       :   5767168 bytes
        Num Conns   :        23
        Rx Total    :   919.900 ms
        Tx Total    :     7.563 ms
        Rx          :     2.989 MB/s
        Tx          :   363.618 MB/s
        Connect     :    71.686 ms
        Connect Avg :     3.117 ms
```

```
wolfSSL Server Benchmark on TLS13-AES256-GCM-SHA384 with group ECC_SECP256R1:
        Total       :   6029312 bytes
        Num Conns   :        24
        Rx Total    :   962.723 ms
        Tx Total    :     6.394 ms
        Rx          :     2.986 MB/s
        Tx          :   449.663 MB/s
        Connect     :    52.042 ms
        Connect Avg :     2.168 ms
wolfSSL Client Benchmark on TLS13-AES256-GCM-SHA384 with group ECC_SECP256R1:
        Total       :   6029312 bytes
        Num Conns   :        24
        Rx Total    :   963.166 ms
        Tx Total    :     7.537 ms
        Rx          :     2.985 MB/s
        Tx          :   381.433 MB/s
        Connect     :    52.348 ms
        Connect Avg :     2.181 ms



wolfSSL Server Benchmark on TLS13-AES256-GCM-SHA384 with group ECC_SECP384R1:
        Total       :   6029312 bytes
        Num Conns   :        24
        Rx Total    :   966.071 ms
        Tx Total    :     8.458 ms
        Rx          :     2.976 MB/s
        Tx          :   339.929 MB/s
        Connect     :    56.135 ms
        Connect Avg :     2.339 ms
wolfSSL Client Benchmark on TLS13-AES256-GCM-SHA384 with group ECC_SECP384R1:
        Total       :   6029312 bytes
        Num Conns   :        24
        Rx Total    :   968.053 ms
        Tx Total    :     7.895 ms
        Rx          :     2.970 MB/s
        Tx          :   364.155 MB/s
        Connect     :    56.188 ms
        Connect Avg :     2.341 ms

wolfSSL Server Benchmark on TLS13-AES256-GCM-SHA384 with group ECC_SECP521R1:
        Total       :   5767168 bytes
        Num Conns   :        23
        Rx Total    :   930.195 ms
        Tx Total    :     7.849 ms
        Rx          :     2.956 MB/s
        Tx          :   350.364 MB/s
        Connect     :    62.644 ms
        Connect Avg :     2.724 ms
wolfSSL Client Benchmark on TLS13-AES256-GCM-SHA384 with group ECC_SECP521R1:
        Total       :   5767168 bytes
        Num Conns   :        23
        Rx Total    :   932.128 ms
```

```
        Tx Total    :     7.440 ms
        Rx          :     2.950 MB/s
        Tx          :   369.619 MB/s
        Connect     :    62.538 ms
        Connect Avg :     2.719 ms

wolfSSL Server Benchmark on TLS13-AES256-GCM-SHA384 with group ML_KEM_512:
        Total       :   6029312 bytes
        Num Conns   :        24
        Rx Total    :   973.208 ms
        Tx Total    :     8.190 ms
        Rx          :     2.954 MB/s
        Tx          :   351.021 MB/s
        Connect     :    49.608 ms
        Connect Avg :     2.067 ms
wolfSSL Client Benchmark on TLS13-AES256-GCM-SHA384 with group ML_KEM_512:
        Total       :   6029312 bytes
        Num Conns   :        24
        Rx Total    :   975.874 ms
        Tx Total    :     7.051 ms
        Rx          :     2.946 MB/s
        Tx          :   407.772 MB/s
        Connect     :    48.708 ms
        Connect Avg :     2.030 ms

wolfSSL Server Benchmark on TLS13-AES256-GCM-SHA384 with group ML_KEM_768:
        Total       :   6029312 bytes
        Num Conns   :        24
        Rx Total    :   965.259 ms
        Tx Total    :     8.098 ms
        Rx          :     2.978 MB/s
        Tx          :   355.041 MB/s
        Connect     :    51.284 ms
        Connect Avg :     2.137 ms
wolfSSL Client Benchmark on TLS13-AES256-GCM-SHA384 with group ML_KEM_768:
        Total       :   6029312 bytes
        Num Conns   :        24
        Rx Total    :   967.507 ms
        Tx Total    :     7.774 ms
        Rx          :     2.972 MB/s
        Tx          :   369.828 MB/s
        Connect     :    49.899 ms
        Connect Avg :     2.079 ms

wolfSSL Server Benchmark on TLS13-AES256-GCM-SHA384 with group ML_KEM_1024:
        Total       :   6029312 bytes
        Num Conns   :        24
        Rx Total    :   972.588 ms
        Tx Total    :     7.835 ms
        Rx          :     2.956 MB/s
        Tx          :   366.959 MB/s
        Connect     :    52.259 ms
        Connect Avg :     2.177 ms
wolfSSL Client Benchmark on TLS13-AES256-GCM-SHA384 with group ML_KEM_1024:
```

```
        Total       :   6029312 bytes
        Num Conns   :        24
        Rx Total    :   974.238 ms
        Tx Total    :     7.838 ms
        Rx          :     2.951 MB/s
        Tx          :   366.813 MB/s
        Connect     :    50.758 ms
        Connect Avg :     2.115 ms

wolfSSL Server Benchmark on TLS13-AES256-GCM-SHA384 with group P256_ML_KEM_512
   :
        Total       :   6029312 bytes
        Num Conns   :        24
        Rx Total    :   971.832 ms
        Tx Total    :     7.544 ms
        Rx          :     2.958 MB/s
        Tx          :   381.096 MB/s
        Connect     :    54.727 ms
        Connect Avg :     2.280 ms
wolfSSL Client Benchmark on TLS13-AES256-GCM-SHA384 with group P256_ML_KEM_512
   :
        Total       :   6029312 bytes
        Num Conns   :        24
        Rx Total    :   972.623 ms
        Tx Total    :     8.807 ms
        Rx          :     2.956 MB/s
        Tx          :   326.456 MB/s
        Connect     :    52.613 ms
        Connect Avg :     2.192 ms

wolfSSL Server Benchmark on TLS13-AES256-GCM-SHA384 with group P384_ML_KEM_768
   :
        Total       :   5767168 bytes
        Num Conns   :        23
        Rx Total    :   921.217 ms
        Tx Total    :     7.740 ms
        Rx          :     2.985 MB/s
        Tx          :   355.285 MB/s
        Connect     :    69.367 ms
        Connect Avg :     3.016 ms
wolfSSL Client Benchmark on TLS13-AES256-GCM-SHA384 with group P384_ML_KEM_768
   :
        Total       :   5767168 bytes
        Num Conns   :        23
        Rx Total    :   923.622 ms
        Tx Total    :     6.928 ms
        Rx          :     2.977 MB/s
        Tx          :   396.956 MB/s
        Connect     :    63.739 ms
        Connect Avg :     2.771 ms

wolfSSL Server Benchmark on TLS13-AES256-GCM-SHA384 with group
   P521_ML_KEM_1024:
        Total       :   5767168 bytes
```

```
        Num Conns   :        23
        Rx Total    :   920.447 ms
        Tx Total    :     7.735 ms
        Rx          :     2.988 MB/s
        Tx          :   355.548 MB/s
        Connect     :    78.446 ms
        Connect Avg :     3.411 ms
wolfSSL Client Benchmark on TLS13-AES256-GCM-SHA384 with group
   P521_ML_KEM_1024:
        Total       :   5767168 bytes
        Num Conns   :        23
        Rx Total    :   921.889 ms
        Tx Total    :     7.585 ms
        Rx          :     2.983 MB/s
        Tx          :   362.578 MB/s
        Connect     :    71.310 ms
        Connect Avg :     3.100 ms
```

G.6 Documentation

Technical documentation and other resources such as known answer tests can be found at the NIST PQC website:

https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions.

For more algorithm-specific benchmarking information, the OQS Project has benchmarking information at their website:

https://openquantumsafe.org/benchmarking/

G.7 Post-Quantum Stateful Hash-Based Signatures

This section covers post-quantum stateful hash-based signature (HBS) schemes such as LMS/HSS, and XMSS/XMSS^MT, for which wolfSSL has recently added support.

G.7.1 Motivation

Stateful HBS schemes are of growing interest for a number of reasons. The primary motivation for stateful HBS schemes is post-quantum security. As discussed previously in this appendix, Shor's algorithm would allow a quantum computer to efficiently factorize large integers and compute discrete logarithms, thus completely breaking public-key cryptography schemes such as RSA and ECC.

In contrast, stateful HBS schemes are founded on the security of their underlying hash functions and Merkle trees (typically implemented with SHA256), which are not expected to be broken by the advent of cryptographically relevant quantum computers. For these reasons they have been recommended by NIST SP 800-208 and the NSA's CNSA 2.0 suite. See these two links for more info:

- https://csrc.nist.gov/publications/detail/sp/800-208/final
- https://media.defense.gov/2022/Sep/07/2003071834/-1/-1/0/CSA_CNSA_2.0_ALGORITHMS_.PDF

Furthermore, the CNSA 2.0 timeline has specified that post-quantum stateful HBS schemes should be used exclusively by 2030, and adoption should begin *immediately*. In fact, adoption of LMS is the earliest requirement in the CNSA 2.0 suite timeline.

However, the nature of stateful HBS schemes requires that significant care is given to their use and tracking their state. In a stateful HBS system, the private key is actually a finite set of one-time signature (OTS) keys, which may never be reused. If the same OTS key were used to sign two different messages, it would be possible for an attacker to fabricate signatures, and the security of the entire scheme would unravel. Therefore, stateful HBS schemes are not suitable for general use such as the public internet.

Instead, because of these unique strengths and characteristics, and NIST and NSA backing, stateful HBS schemes such as LMS/HSS are of particular interest for offline firmware authentication and signature verification, especially on embedded or constrained systems that are expected to have a long operational lifetime and thus need to be resilient against a cryptographically relevant quantum computer.

G.7.2 LMS/HSS signatures

wolfSSL is adding support for the LMS/HSS hash-based signature scheme to our wolfCrypt embedded crypto engine. This will be achieved by experimental integration with the hash-sigs LMS/HSS library (https://github.com/cisco/hash-sigs), similar to our previous libOQS integration.

Leighton-Micali Signatures (LMS), and its multi-tree variant, the Hierarchical Signature System (HSS), is a post-quantum, stateful hash-based signature scheme. It is noted for having small public and private keys, and fast signing and verifying. Its signature sizes are larger, but are tunable via its Winternitz parameter. See these two links from RFC8554 for more details:

- LMS: https://datatracker.ietf.org/doc/html/rfc8554
- HSS: https://datatracker.ietf.org/doc/html/rfc8554#section-6

As previously discussed, the LMS/HSS signature system consists of a finite number of one-time signature (OTS) keys, and thus may only safely generate a finite number of signatures. However the number of signatures, and the signature size, are tunable via a set of defined parameters, which will be discussed next.

G.7.2.1 Supported Parameters LMS/HSS signatures are defined by 3 parameters: - levels: number of levels of Merkle trees. - height: height of an individual Merkle tree. - Winternitz: number of bits from hash used in a Winternitz chain. Used as a space-time tradeoff for the signature size.

wolfSSL supports all LMS/HSS parameters defined in RFC8554:

- levels = {1..8}
- height = {5, 10, 15, 20, 25}
- Winternitz = {1, 2, 4, 8}

The number of available signatures is: - N = 2 ** (levels * height)

For convenience some parameter sets have been predefined in the enum `wc_LmsParm`. Its values are shown in the table below:

parameter set	description
WC_LMS_PARM_NONE	Not set, use default (WC_LMS_PARM_L1_H15_W2)
WC_LMS_PARM_L1_H15_W2	1 level Merkle tree of 15 height, Winternitz 2
WC_LMS_PARM_L1_H15_W4	same as above, Winternitz 4
WC_LMS_PARM_L2_H10_W2	2 level Merkle tree of 10 height, Winternitz 4
WC_LMS_PARM_L2_H10_W4	same as above, Winternitz 4
WC_LMS_PARM_L2_H10_W8	same as above, Winternitz 8
WC_LMS_PARM_L3_H5_W2	3 level Merkle tree of 5 height, Winternitz 2
WC_LMS_PARM_L3_H5_W4	same as above, Winternitz 4
WC_LMS_PARM_L3_H5_W8	same as above, Winternitz 8

COPYRIGHT ©2024 wolfSSL Inc.

parameter set	description
WC_LMS_PARM_L3_H10_W4	3 level Merkle tree of 10 height, Winternitz 4
WC_LMS_PARM_L4_H5_W8	4 level Merkle tree of 5 height, Winternitz 8

The signature size and number of signatures is shown with respect to the parameter set here:

parameter set	signature size	number of signatures
WC_LMS_PARM_L1_H15_W2	4784	32768
WC_LMS_PARM_L1_H15_W4	2672	32768
WC_LMS_PARM_L2_H10_W2	9300	1048576
WC_LMS_PARM_L2_H10_W4	5076	1048576
WC_LMS_PARM_L2_H10_W8	2964	1048576
WC_LMS_PARM_L3_H5_W2	13496	32768
WC_LMS_PARM_L3_H5_W4	7160	32768
WC_LMS_PARM_L3_H5_W8	3992	32768
WC_LMS_PARM_L3_H10_W4	7640	1073741824
WC_LMS_PARM_L4_H5_W8	5340	1048576

As can be seen from the tables, signature sizes are primarily determined by the levels and Winternitz parameters, and height to a lesser extent: - Larger levels values increase signature size significantly. - Larger height values increase signature size modestly. - Larger winternitz values will reduce the signature size, at the expense of longer key generation and sign/verify times.

Key generation time is strongly determined by the height of the first level tree. A 3 level, 5 height tree is much faster than 1 level, 15 height at initial key gen, even if the number of available signatures is the same.

G.7.2.2 LMS/HSS Build Instructions Please see the wolfSSL repo's INSTALL file (https://github.com/wolfSSL/wolfssl Item 17 (Building with hash-sigs lib for LMS/HSS support [EXPERIMENTAL]) has instructions on how to configure and build wolfSSL and the hash-sigs LMS/HSS library.

G.7.2.3 Benchmark Data The following benchmark data was taken on an 8-core Intel i7-8700 CPU @ 3.20GHz, on Fedora 38 (`6.2.9-300.fc38.x86_64`). The multi-threaded example used 4 worker threads and 4 cores, while the single-threaded example used only a single core.

As discussed in item 17 of the INSTALL file, the hash-sigs lib offers two static libraries: - `hss_lib.a`: a single-threaded version. - `hss_lib_thread.a`: a multi-threaded version.

The multi-threaded version will spawn worker threads to accelerate cpu intensive tasks, such as key generation. This will mainly speedup key generation and signing for all parameter values, and to a lesser extent will speedup verifying for larger levels values.

For reference, wolfSSL was built with the following to obtain both benchmarks:

```
./configure \
  --enable-static \
  --disable-shared \
  --enable-lms=yes \
  --with-liblms=<path to hash sigs install>
```

multi-threaded benchmark

The following is benchmark data obtained when built against the multi-threaded `hss_lib_thread.a`, which used 4 worker threads to parallelize intensive tasks, and used 4 cores.

```
./wolfcrypt/benchmark/benchmark -lms_hss
------------------------------------------------------------------------------
 wolfSSL version 5.6.3
------------------------------------------------------------------------------
Math:   Multi-Precision: Wolf(SP) word-size=64 bits=4096 sp_int.c
wolfCrypt Benchmark (block bytes 1048576, min 1.0 sec each)
LMS/HSS L2_H10_W2  9300     sign      1500 ops took 1.075 sec, avg 0.717 ms,
   1394.969 ops/sec
LMS/HSS L2_H10_W2  9300   verify      5200 ops took 1.002 sec, avg 0.193 ms,
   5189.238 ops/sec
LMS/HSS L2_H10_W4  5076     sign       800 ops took 1.012 sec, avg 1.265 ms,
   790.776 ops/sec
LMS/HSS L2_H10_W4  5076   verify      2500 ops took 1.003 sec, avg 0.401 ms,
   2493.584 ops/sec
LMS/HSS L3_H5_W4  7160     sign      1500 ops took 1.051 sec, avg 0.701 ms,
   1427.485 ops/sec
LMS/HSS L3_H5_W4  7160   verify      2700 ops took 1.024 sec, avg 0.379 ms,
   2636.899 ops/sec
LMS/HSS L3_H5_W8  3992     sign       300 ops took 1.363 sec, avg 4.545 ms,
   220.030 ops/sec
LMS/HSS L3_H5_W8  3992   verify       400 ops took 1.066 sec, avg 2.664 ms,
   375.335 ops/sec
LMS/HSS L3_H10_W4  7640     sign       900 ops took 1.090 sec, avg 1.211 ms,
   825.985 ops/sec
LMS/HSS L3_H10_W4  7640   verify      2400 ops took 1.037 sec, avg 0.432 ms,
   2314.464 ops/sec
LMS/HSS L4_H5_W8  5340     sign       300 ops took 1.310 sec, avg 4.367 ms,
   228.965 ops/sec
LMS/HSS L4_H5_W8  5340   verify       400 ops took 1.221 sec, avg 3.053 ms,
   327.599 ops/sec
Benchmark complete
```

single-threaded benchmark

The following is benchmark data obtained when built against the single-threaded `hss_lib.a`, which will use only a single core.

```
$ ./wolfcrypt/benchmark/benchmark -lms_hss
------------------------------------------------------------------------------
 wolfSSL version 5.6.3
------------------------------------------------------------------------------
Math:   Multi-Precision: Wolf(SP) word-size=64 bits=4096 sp_int.c
wolfCrypt Benchmark (block bytes 1048576, min 1.0 sec each)
LMS/HSS L2_H10_W2  9300     sign       800 ops took 1.115 sec, avg 1.394 ms,
   717.589 ops/sec
LMS/HSS L2_H10_W2  9300   verify      4500 ops took 1.001 sec, avg 0.223 ms,
   4493.623 ops/sec
LMS/HSS L2_H10_W4  5076     sign       500 ops took 1.239 sec, avg 2.478 ms,
   403.519 ops/sec
LMS/HSS L2_H10_W4  5076   verify      2100 ops took 1.006 sec, avg 0.479 ms,
   2087.944 ops/sec
LMS/HSS L3_H5_W4  7160     sign       800 ops took 1.079 sec, avg 1.349 ms,
   741.523 ops/sec
LMS/HSS L3_H5_W4  7160   verify      1600 ops took 1.012 sec, avg 0.632 ms,
   1581.686 ops/sec
```

```
LMS/HSS L3_H5_W8  3992     sign       100 ops took 1.042 sec, avg 10.420 ms,
   95.971 ops/sec
LMS/HSS L3_H5_W8  3992   verify       200 ops took 1.220 sec, avg 6.102 ms,
   163.894 ops/sec
LMS/HSS L3_H10_W4  7640     sign       400 ops took 1.010 sec, avg 2.526 ms,
   395.864 ops/sec
LMS/HSS L3_H10_W4  7640   verify      1500 ops took 1.052 sec, avg 0.701 ms,
   1426.284 ops/sec
LMS/HSS L4_H5_W8  5340     sign       100 ops took 1.066 sec, avg 10.665 ms,
   93.768 ops/sec
LMS/HSS L4_H5_W8  5340   verify       200 ops took 1.478 sec, avg 7.388 ms,
   135.358 ops/sec
Benchmark complete
```

G.7.3 XMSS/XMSS^MT signatures

wolfSSL is adding support for XMSS/XMSS^MT stateful hash-based signatures. Similar to LMS, this will be done by experimental integration with the xmss-reference repository (https://github.com/XMSS/xmss-reference.git), from RFC 8391 (https://www.rfc-editor.org/rfc/rfc8391.html).

The xmss-reference supports `xmss_core_fast`, and `xmss_core` implementations. The `xmss_core_fast` implementation was designed to prioritize performance with larger private key sizes as a tradeoff. Our integration uses `xmss_core_fast`, with a patch applied so that the wolfCrypt SHA256 implementation may be used instead.

The patch may be found here

```
in the wolfssl-examples repository: https://github.com/wolfSSL/wolfssl-
   examples.

Overall, XMSS/XMSS^MT is similar to LMS/HSS. For a more detailed comparison
   see
"LMS vs XMSS: Comparison of two Hash-Based Signature Standards" (https://
   eprint.iacr.org/2017/349.pdf).


XMSS^MT is the Multi-Tree generalization of XMSS,
analogous to HSS with LMS, with the distinction that the Winternitz value is
   fixed to `w=16` in
XMSS/XMSS^MT. The public key is slightly larger in XMSS/XMSS^MT (at 68 bytes
   in XMSS/XMSS^MT, vs 60 bytes in LMS/HSS), while signatures are slightly
   smaller.

#### Supported Parameters

wolfSSL supports the SHA256 XMSS/XMSS^MT parameter sets from Tables 10 and 11
   from NIST SP 800-208 (https://csrc.nist.gov/pubs/sp/800/208/final).

parameter set name      | Oid         | n   | w   | h   | d   | h/d | Sig len
----------------------- | ----------- | --- | --- | --- | --- | --- | --
XMSS                    |             |     |     |     |     |     |
"XMSS-SHA2_10_256"      |  0x00000001 |  32 |  16 |  10 |  1  |  10 | 2500
"XMSS-SHA2_16_256"      |  0x00000002 |  32 |  16 |  16 |  1  |  16 | 2692
"XMSS-SHA2_20_256"      |  0x00000003 |  32 |  16 |  20 |  1  |  20 | 2820
XMSS^MT                 |             |     |     |     |     |     |
```

```
"XMSSMT-SHA2_20/2_256"  |  0x00000001 |  32 |  16 |  20 |  2  |  10 | 4963
"XMSSMT-SHA2_20/4_256"  |  0x00000002 |  32 |  16 |  20 |  4  |   5 | 9251
"XMSSMT-SHA2_40/2_256"  |  0x00000003 |  32 |  16 |  40 |  2  |  20 | 5605
"XMSSMT-SHA2_40/4_256"  |  0x00000004 |  32 |  16 |  40 |  4  |  10 | 9893
"XMSSMT-SHA2_40/8_256"  |  0x00000005 |  32 |  16 |  40 |  8  |   5 | 18469
"XMSSMT-SHA2_60/3_256"  |  0x00000006 |  32 |  16 |  60 |  3  |  20 | 8392
"XMSSMT-SHA2_60/6_256"  |  0x00000007 |  32 |  16 |  60 |  6  |  10 | 14824
"XMSSMT-SHA2_60/12_256" |  0x00000008 |  32 |  16 |  60 |  12 |   5 | 27688

In the table above, `n` is the number of bytes in the HASH function, `w` the
   Winternitz value, `h` the total height of the tree system, and `d` the
   number of levels of trees.

Key generation time is strongly determined by the height of the
first level tree (or `h/d`), while signature length grows primarily with `d` (
   the number of hyper
tree levels).

Similar to LMS/HSS, the number of available signatures grows as `2**h`, where
   h is the
total height of the tree system.

#### Benchmark Data

In the following, benchmark data is shown for several XMSS/XMSS^MT parameter
   sets,
for intel x86_64 and aarch64.  The SHA256 performance on these systems is also
    listed for reference,
as computing the large number of required hash chains will constitute the bulk
    of the CPU
work for XMSS/XMSS^MT. Additionally, our patch to xmss-reference substitutes
   wolfCrypt's SHA256 implementation, and therefore benefits from the same ASM
    speedups.

As previously mentioned, our xmss integration is using the `xmss_core_fast`
   implementation from xmss-reference, which has faster performance at the
   tradeoff of larger private key sizes.

**x86_64**

The following x86_64 benchmark data were taken on an 8-core Intel i7-8700 CPU
   @ 3.20GHz, on Fedora 38 (`6.2.9-300.fc38.x86_64`). This CPU has `avx avx2`
   flags, which can accelerate hash operations and be utilized with `--enable-
   intelasm`.

With `--enable-intelasm`:

```text
$./wolfcrypt/benchmark/benchmark -xmss_xmssmt -sha256
------------------------------------------------------------------------------
 wolfSSL version 5.6.3
------------------------------------------------------------------------------
Math:   Multi-Precision: Wolf(SP) word-size=64 bits=4096 sp_int.c
wolfCrypt Benchmark (block bytes 1048576, min 1.0 sec each)
```
```

```
SHA-256                    500 MiB took 1.009 seconds,  495.569 MiB/s Cycles
   per byte =   6.14
XMSS-SHA2_10_256  2500     sign       200 ops took 1.010 sec, avg 5.052 ms,
   197.925 ops/sec
XMSS-SHA2_10_256  2500   verify      1600 ops took 1.011 sec, avg 0.632 ms,
   1582.844 ops/sec
XMSSMT-SHA2_20/2_256  4963     sign       200 ops took 1.286 sec, avg 6.431 ms
   , 155.504 ops/sec
XMSSMT-SHA2_20/2_256  4963   verify       700 ops took 1.009 sec, avg 1.441 ms
   , 693.905 ops/sec
XMSSMT-SHA2_20/4_256  9251     sign       300 ops took 1.223 sec, avg 4.076 ms
   , 245.335 ops/sec
XMSSMT-SHA2_20/4_256  9251   verify       400 ops took 1.027 sec, avg 2.569 ms
   , 389.329 ops/sec
XMSSMT-SHA2_40/4_256  9893     sign       200 ops took 1.466 sec, avg 7.332 ms
   , 136.394 ops/sec
XMSSMT-SHA2_40/4_256  9893   verify       400 ops took 1.024 sec, avg 2.560 ms
   , 390.627 ops/sec
XMSSMT-SHA2_40/8_256 18469     sign       300 ops took 1.202 sec, avg 4.006 ms
   , 249.637 ops/sec
XMSSMT-SHA2_40/8_256 18469   verify       200 ops took 1.089 sec, avg 5.446 ms
   , 183.635 ops/sec
XMSSMT-SHA2_60/6_256 14824     sign       200 ops took 1.724 sec, avg 8.618 ms
   , 116.033 ops/sec
XMSSMT-SHA2_60/6_256 14824   verify       300 ops took 1.136 sec, avg 3.788 ms
   , 263.995 ops/sec
XMSSMT-SHA2_60/12_256 27688     sign       300 ops took 1.210 sec, avg 4.034
   ms, 247.889 ops/sec
XMSSMT-SHA2_60/12_256 27688   verify       200 ops took 1.575 sec, avg 7.877
   ms, 126.946 ops/sec
Benchmark complete
```

Without `--enable-intelasm`:

```
$./wolfcrypt/benchmark/benchmark -xmss_xmssmt -sha256
------------------------------------------------------------------------------
 wolfSSL version 5.6.3
------------------------------------------------------------------------------
Math:   Multi-Precision: Wolf(SP) word-size=64 bits=4096 sp_int.c
wolfCrypt Benchmark (block bytes 1048576, min 1.0 sec each)
SHA-256                    275 MiB took 1.005 seconds,  273.549 MiB/s Cycles
   per byte =  11.13
XMSS-SHA2_10_256  2500     sign       200 ops took 1.356 sec, avg 6.781 ms,
   147.480 ops/sec
XMSS-SHA2_10_256  2500   verify      1200 ops took 1.025 sec, avg 0.854 ms,
   1170.547 ops/sec
XMSSMT-SHA2_20/2_256  4963     sign       200 ops took 1.687 sec, avg 8.436 ms
   , 118.546 ops/sec
XMSSMT-SHA2_20/2_256  4963   verify       600 ops took 1.187 sec, avg 1.978 ms
   , 505.663 ops/sec
XMSSMT-SHA2_20/4_256  9251     sign       200 ops took 1.119 sec, avg 5.593 ms
   , 178.785 ops/sec
XMSSMT-SHA2_20/4_256  9251   verify       300 ops took 1.086 sec, avg 3.622 ms
   , 276.122 ops/sec
XMSSMT-SHA2_40/4_256  9893     sign       200 ops took 1.991 sec, avg 9.954 ms
```

```
    , 100.460 ops/sec
XMSSMT-SHA2_40/4_256  9893   verify       300 ops took 1.043 sec, avg 3.478 ms
    , 287.545 ops/sec
XMSSMT-SHA2_40/8_256 18469     sign       200 ops took 1.114 sec, avg 5.572 ms
    , 179.454 ops/sec
XMSSMT-SHA2_40/8_256 18469   verify       200 ops took 1.495 sec, avg 7.476 ms
    , 133.770 ops/sec
XMSSMT-SHA2_60/6_256 14824     sign       100 ops took 1.111 sec, avg 11.114
   ms, 89.975 ops/sec
XMSSMT-SHA2_60/6_256 14824   verify       200 ops took 1.070 sec, avg 5.349 ms
    , 186.963 ops/sec
XMSSMT-SHA2_60/12_256 27688     sign       200 ops took 1.148 sec, avg 5.739
   ms, 174.247 ops/sec
XMSSMT-SHA2_60/12_256 27688   verify       100 ops took 1.080 sec, avg 10.797
   ms, 92.618 ops/sec
Benchmark complete
```

aarch64

The following aarch64 data were taken on Ubuntu linux (`5.15.0-71-generic`) running on an Apple M1, with cpu flags `sha1 sha2 sha3 sha512`, which will specifically significantly accelerate SHA hash operations when built with `--enable-armasm`.

With `--enable-armasm`:

```
$ ./wolfcrypt/benchmark/benchmark -xmss_xmssmt -sha256
------------------------------------------------------------------------------
 wolfSSL version 5.6.3
------------------------------------------------------------------------------
Math: Multi-Precision: Wolf(SP) word-size=64 bits=4096 sp_int.c
wolfCrypt Benchmark (block bytes 1048576, min 1.0 sec each)
SHA-256                    2305 MiB took 1.001 seconds, 2303.346 MiB/s
XMSS-SHA2_10_256  2500     sign       800 ops took 1.079 sec, avg 1.349 ms,
   741.447 ops/sec
XMSS-SHA2_10_256  2500   verify      6500 ops took 1.007 sec, avg 0.155 ms,
   6455.445 ops/sec
XMSSMT-SHA2_20/2_256  4963     sign       700 ops took 1.155 sec, avg 1.650 ms
    , 606.154 ops/sec
XMSSMT-SHA2_20/2_256  4963   verify      3100 ops took 1.021 sec, avg 0.329 ms
    , 3037.051 ops/sec
XMSSMT-SHA2_20/4_256  9251     sign      1100 ops took 1.006 sec, avg 0.915 ms
    , 1093.191 ops/sec
XMSSMT-SHA2_20/4_256  9251   verify      1700 ops took 1.013 sec, avg 0.596 ms
    , 1677.399 ops/sec
XMSSMT-SHA2_40/4_256  9893     sign       600 ops took 1.096 sec, avg 1.827 ms
    , 547.226 ops/sec
XMSSMT-SHA2_40/4_256  9893   verify      1600 ops took 1.062 sec, avg 0.664 ms
    , 1506.946 ops/sec
XMSSMT-SHA2_40/8_256 18469     sign      1100 ops took 1.007 sec, avg 0.916 ms
    , 1092.214 ops/sec
XMSSMT-SHA2_40/8_256 18469   verify       900 ops took 1.088 sec, avg 1.209 ms
    , 827.090 ops/sec
XMSSMT-SHA2_60/6_256 14824     sign       600 ops took 1.179 sec, avg 1.966 ms
    , 508.728 ops/sec
XMSSMT-SHA2_60/6_256 14824   verify      1100 ops took 1.038 sec, avg 0.944 ms
    , 1059.590 ops/sec
```

```
XMSSMT-SHA2_60/12_256 27688     sign      1100 ops took 1.015 sec, avg 0.923
   ms, 1083.767 ops/sec
XMSSMT-SHA2_60/12_256 27688   verify       600 ops took 1.149 sec, avg 1.914
   ms, 522.367 ops/sec
Benchmark complete
```

Without `--enable-armasm`:

```
$ ./wolfcrypt/benchmark/benchmark -xmss_xmssmt -sha256
------------------------------------------------------------------------------
 wolfSSL version 5.6.3
------------------------------------------------------------------------------
Math: Multi-Precision: Wolf(SP) word-size=64 bits=4096 sp_int.c
wolfCrypt Benchmark (block bytes 1048576, min 1.0 sec each)
SHA-256                     190 MiB took 1.020 seconds,  186.277 MiB/s
XMSS-SHA2_10_256  2500      sign       200 ops took 1.908 sec, avg 9.538 ms,
   104.845 ops/sec
XMSS-SHA2_10_256  2500    verify       800 ops took 1.002 sec, avg 1.253 ms,
   798.338 ops/sec
XMSSMT-SHA2_20/2_256  4963     sign       100 ops took 1.084 sec, avg 10.843
   ms, 92.222 ops/sec
XMSSMT-SHA2_20/2_256  4963   verify       500 ops took 1.240 sec, avg 2.479 ms
   , 403.334 ops/sec
XMSSMT-SHA2_20/4_256  9251     sign       200 ops took 1.615 sec, avg 8.074 ms
   , 123.855 ops/sec
XMSSMT-SHA2_20/4_256  9251   verify       200 ops took 1.071 sec, avg 5.355 ms
   , 186.726 ops/sec
XMSSMT-SHA2_40/4_256  9893     sign       100 ops took 1.354 sec, avg 13.543
   ms, 73.840 ops/sec
XMSSMT-SHA2_40/4_256  9893   verify       300 ops took 1.483 sec, avg 4.945 ms
   , 202.237 ops/sec
XMSSMT-SHA2_40/8_256 18469     sign       200 ops took 1.588 sec, avg 7.941 ms
   , 125.922 ops/sec
XMSSMT-SHA2_40/8_256 18469   verify       100 ops took 1.042 sec, avg 10.415
   ms, 96.014 ops/sec
XMSSMT-SHA2_60/6_256 14824     sign       100 ops took 1.571 sec, avg 15.710
   ms, 63.654 ops/sec
XMSSMT-SHA2_60/6_256 14824   verify       200 ops took 1.526 sec, avg 7.632 ms
   , 131.033 ops/sec
XMSSMT-SHA2_60/12_256 27688     sign       200 ops took 1.607 sec, avg 8.036
   ms, 124.434 ops/sec
XMSSMT-SHA2_60/12_256 27688   verify       100 ops took 1.501 sec, avg 15.011
   ms, 66.616 ops/sec
Benchmark complete
```

G.7.4 Developer Notes

- If you are trying to thwart the “harvest now, decrypt later” threat model and you are willing to sacrifice some interoperability, then you do not want to advertise support for conventional algorithms in the Supported Groups extension. Be sure to call `wolfSSL_UseKeyShare()` and `wolfSSL_set_groups()` with your chosen algorithms. Only calling `wolfSSL_UseKeyShare()` is insufficient as that will advertise your support for quantum-vulnerable algorithms. If your peer does not support post-quantum algorithms, they will then send a HelloRetryRequest which will then result in a connection with a conventional algorithm.

H wolfSSL Porting Guide

H.1 Purpose

This guide provides a reference for developers and engineers porting the wolfSSL lightweight SSL/TLS library to new embedded platforms, operating systems, or transport mediums (TCP/IP, bluetooth, etc.). It calls out areas in the wolfSSL codebase which typically require modification when porting wolfSSL. It should be considered a "guide" and as such, it is an evolving work. If there is something you find missing, please let us know and we'll be happy to add instructions or clarification to the document.

H.2 Audience

This guide caters to developers or engineers porting the wolfSSL and wolfCrypt to new platforms or environments that are not supported by default.

H.3 Introduction

Several steps need to be iterated through when getting wolfSSL to run on an embedded platform. Some of these steps are outlined in Section 2.4.

Apart from steps in Chapter 2 of the wolfSSL Manual, there are areas in the code which may need porting or modifications in order to accommodate a specific platform. wolfSSL abstracts many of these areas - attempting to make it as easy as possible to port wolfSSL to a new platform.

In the `./wolfssl/wolfcrypt/settings.h` file, there are several defines specific to different operating systems, TCP/IP stacks, and chipsets (ex: MBED, FREESCALE_MQX, MICROCHIP_PIC32, MICRIUM, EBSNET, etc.). There are two main locations to put `#defines` when compiling and porting wolfSSL to a new platform:

1. New defines for a Operating System or TCP/IP stack port are typically added to the `settings.h` file when a new port of wolfSSL is completed. This provides an easy way to turn on/off features as well as customize build settings that should be "default" for that build. New custom defines can be added in this file when doing a port of wolfSSL to a new platform. We encourage users to contribute ports of wolfSSL back to the master open source code branch on GitHub. This helps keep wolfSSL up to date and allows different ports to remain updated as the wolfSSL project improves and moves forward.
2. For users not wanting to contribute back their changes to wolfSSL proper, or for users who want to customize the wolfSSL build with additional preprocessor defines, wolfSSL recommends the use of a custom `user_settings.h` header file. If `WOLFSSL_USER_SETTINGS` is defined when compiling the wolfSSL source files, wolfSSL will automatically include a custom header file called `user_settings.h`. This header should be created by the user and placed on the include path. This allows users to maintain one single file for their wolfSSL build, and makes it much easier to update to newer versions of wolfSSL.

wolfSSL encourages the submission of patches and code changes through either direct email (facts@wolfssl.com), or through GitHub pull request.

H.4 Porting wolfSSL

H.4.1 Data Types

Q: When do I need to read this section?

A: Setting the correct data type size for your platform is always important.

wolfSSL benefits speed-wise from having a 64-bit type available. Define SIZEOF_LONG and SIZEOF_LONG_LONG to match the result of sizeof(long) and sizeof(long long) on your platform. This can be added to a custom define in the settings.h file or to user_settings.h. For example, in settings.h under a sample define of MY_NEW_PLATFORM:

```
#ifdef MY_NEW_PLATFORM
    #define SIZEOF_LONG 4
    #define SIZEOF_LONG_LONG 8
    ...
#endif
```

There are two additional data types used by wolfSSL and wolfCrypt, called word32 and word16. The default type mappings for these are:

```
#ifndef WOLFSSL_TYPES
#ifndef byte
    typedef unsigned char  byte;
#endif
    typedef unsigned short word16;
    typedef unsigned int   word32;
    typedef byte           word24[3];
#endif
```

word32 should be mapped to the compiler's 32-bit type, and word16 to the compiler's 16-bit type. If these default mappings are incorrect for your platform, you should define WOLFSSL_TYPES in settings.h or user_settings.h and assign your own custom typedefs for word32 and word16.

The fastmath library in wolfSSL uses the fp_digit and fp_word types. By default these are mapped in <wolfssl/wolfcrypt/tfm.h> depending on build configuration.

fp_word should be twice the size of fp_digit. If the default cases do not hold true for your platform, you should define WOLFSSL_BIGINT_TYPES in settings.h or user_settings.h and assign your own custom typedefs for fp_word and fp_digit.

wolfSSL does use a 64-bit type when available for some operations. The wolfSSL build tries to detect and set up the correct underlying data type for word64 based on what SIZEOF_LONG and SIZEOF_LONG_LONG have been set to. On some platforms that don't have a true 64-bit type, where two 32-bit types are used in conjunction, performance can be slow. To compile out the use of 64-bit types, define NO_64BIT.

H.4.2 Endianness

Q: When do I need to read this section?

A: Your platform is a big endian system.

Is your platform big endian or little endian? wolfSSL defaults to a little endian system. If your system is big endian, define BIG_ENDIAN_ORDER when building wolfSSL. Example of setting this in settings.h:

```
#ifdef MY_NEW_PLATFORM
    ...
    #define BIG_ENDIAN_ORDER
    ...
#endif
```

H.4.3 writev

Q: When do I need to read this section?

A: <sys/uio.h> is not available.

By default, the wolfSSL API makes available wolfSSL_writev() to applications, which simulates writev() semantics. On systems that don't have the <sys/uio.h> header available, define NO_WRITEV to exclude this feature.

H.4.4 Input / Output

Q: When do I need to read this section?

A: A BSD-style socket API is not available, you are using a custom transport layer or TCP/IP stack, or only want to use static buffers.

wolfSSL defaults to using a BSD-style socket interface. If your transport layer provides a BSD socket interface, wolfSSL should integrate into it as-is, unless custom headers are needed.

wolfSSL provides a custom I/O abstraction layer which allows users to tailor wolfSSL's I/O functionality to their system. Full details can be found in Section 5.1.2.

Simply put, you can define WOLFSSL_USER_IO, then write your own I/O callback functions using wolfSSL's default EmbedSend() and EmbedReceive() as templates. These two functions are located in ./src/io.c.

wolfSSL uses dynamic buffers for input and output, which default to 0 bytes. If an input record is received that is greater in size than the buffer, then a dynamic buffer is temporarily used to handle the request and then freed.

If you prefer using large, 16kB static buffers which will never need dynamic memory, you can enable this option by defining LARGE_STATIC_BUFFERS.

If dynamic buffers are used and the user requests an wolfSSL_write() that is bigger than the buffer size, then a dynamic block up to MAX_RECORD_SIZE is used to send the data. Users wishing to only send the data in chunks of the current buffer size at maximum, as defined by RECORD_SIZE, can do this by defining STATIC_CHUNKS_ONLY. When using this define, RECORD_SIZE defaults to 128 bytes.

H.4.5 Filesystem

Q: When do I need to read this section?

A: No file system is available, standard file system functions are not available, or you have a custom file system.

wolfSSL uses the filesystem for loading keys and certificates into the SSL session or context. wolfSSL also allows loading these from memory buffers. If strictly using memory buffers, a filesystem is not needed.

You can disable wolfSSL's usage of the filesystem by defining NO_FILESYSTEM when building the library. This means that certificates and keys will need to be loaded from memory buffers instead of files. An example of setting this in settings.h:

```
#ifdef MY_NEW_PLATFORM
    ...
    #define NO_FILESYSTEM
    ...
#endif
```

Test key and certificate buffers can be found in the ./wolfssl/certs_test.h header file. These will match up to corresponding certificates and keys found in the ./certs directory.

The certs_test.h header file can be updated using the ./gencertbuf.pl script if needed. Inside gencertbuf.pl, there are two arrays: fileList_1024 and fileList_2048. Additional certificates

or keys may be added to the respective array, depending on key size, and must be in DER format. The above mentioned arrays map a certificate/key file location with the desired buffer name. After modifying gencertbuf.pl, running it from the wolfSSL root directory will update the certificate and key buffers in ./wolfssl/certs_test.h:

```
./gencertbuf.pl
```

If you would like to use a filesystem other than the default, the filesystem abstraction layer is located in ./wolfssl/wolfcrypt/wc_port.h. Here you will see filesystem ports for various platforms including EBSNET, FREESCALE_MQX, and MICRIUM. You can add a custom define for your platform if needed - allowing you to define file system functions with XFILE, XFOPEN, XFSEEK, etc. For example, the filesystem layer in wc_port.h for Micrium's µC/OS (MICRIUM) is as follows:

```
#elif defined(MICRIUM)
#include <fs.h>
#define XFILE      FS_FILE*
#define XFOPEN     fs_fopen
#define XFSEEK     fs_fseek
#define XFTELL     fs_ftell
#define XREWIND    fs_rewind
#define XFREAD     fs_fread
#define XFCLOSE    fs_fclose
#define XSEEK_END  FS_SEEK_END
#define XBADFILE   NULL
```

H.4.6 Threading

Q: When do I need to read this section?

A: You want to use wolfSSL in a multithreaded environment, or want to just compile it in single threaded mode.

If wolfSSL will only be used in a single threaded environment, the wolfSSL mutex layer can be disabled when compiling wolfSSL by defining SINGLE_THREADED. This will negate the need to port the wolfSSL mutex layer.

If wolfSSL needs to be used in a multithreaded environment, the wolfSSL mutex layer will need to be ported to the new environment. The mutex layer can be found in ./wolfssl/wolfcrypt/wc_port.h and ./wolfcrypt/src/wc_port.c. wolfSSL_Mutex will need to be defined for the new system in wc_port.h and the mutex functions (wc_InitMutex, wc_FreeMutex, wc_LockMutex and wc_UnLockMutex) in wc_port.c. You can search in wc_port.h and wc_port.c to see an example for some existing platform port layers (EBSNET, FREESCALE_MQX, etc.).

H.4.7 Random Seed

Q: When do I need to read this section?

A: Either /dev/random or /dev/urandom is not available or you want to integrate into a hardware RNG.

By default, wolfSSL uses /dev/urandom or /dev/random to generate a RNG seed. The NO_DEV_RANDOM define can be used when building wolfSSL to disable the default GenerateSeed() function. If this is defined, you need to write a custom GenerateSeed() function in ./wolfcrypt/src/random.c, specific to your target platform. This allows you to seed wolfSSL's PRNG with a hardware-based random entropy source if available.

For examples of how GenerateSeed() needs to be written, reference wolfSSL's existing GenerateSeed() implementations in ./wolfcrypt/src/random.c.

H.4.8 Memory

Q: When do I need to read this section?

A: When you don't have standard memory functions available or are interested in memory usage differences between optional math libraries.

wolfSSL proper uses both `malloc()` and `free()` by default. When using the normal big integer math library, wolfCrypt will also use `realloc()`.

By default wolfSSL/wolfCrypt use the normal big integer math library, which uses quite a bit of dynamic memory. When building wolfSSL, the fastmath library can be enabled, which is both faster and uses no dynamic memory for crypto operations (all on the stack). By using fastmath, wolfSSL won't need a `realloc()` implementation at all. As the SSL layer of wolfSSL still uses some dynamic memory, `malloc()` and `free()` are still required.

For a comparison of resource usage (stack/heap) between the big integer math library and fastmath library, ask us to see our Resource Use document.

To enable fastmath, define `USE_FAST_MATH` and build in `./wolfcrypt/src/tfm.c` instead of `./wolfcrypt/src/integer.c`. Since the stack memory can be large when using fastmath, we recommend defining `TFM_TIMING_RESISTANT` as well.

If the normal `malloc()`, `free()`, and possibly `realloc()` functions are not available, define `XMALLOC_USER`, then provide custom memory function hooks in `./wolfssl/wolfcrypt/types.h` specific to the target environment.

Please read section 5.1.1.1 for details about using `XMALLOC_USER`.

H.4.9 Time

Q: When do I need to read this section?

A: When standard time functions (`time()`, `gmtime()`) are not available, or you need to specify a custom clock tick function.

By default, wolfSSL uses `time()`, `gmtime()`, and `ValidateDate()`, as specified in `./wolfcrypt/src/asn.c`. These are abstracted to `XTIME`, `XGMTIME`, and `XVALIDATE_DATE`. If the standard time functions, and `time.h`, are not available, the user can define `USER_TIME`. After defining `USER_TIME`, the user can define their own `XTIME`, `XGMTIME`, and `XVALIDATE_DATE` functions.

wolfSSL uses `time(0)` by default for the clock tick function. This is located in `./src/internal.c` inside of the `LowResTimer()` function.

Defining `USER_TICKS` allows the user to define their own clock tick function if `time(0)` is not wanted. The custom function needs second accuracy, but doesn't have to be correlated to EPOCH. See `LowResTimer()` function in `./src/internal.c` for reference.

H.4.10 C Standard Library

Q: When do I need to read this section?

A: When you don't have a C standard library available, or have a custom one.

wolfSSL can be built without the C standard library to provide a higher level of portability and flexibility to developers. When doing so, the user needs to map functions they wish to use instead of the C standard ones.

Section 7, above, covered memory functions. In addition to memory function abstraction, wolfSSL also abstracts string function and math functions, where the specific functions are typically abstracted to a define in the form of X<FUNC>, where <FUNC> is the name of the function being abstracted.

Please read Section 5.1 for details.

H.4.11 Logging

Q: When do I need to read this section?

A: You want to enable debug messages but don't have stderr available.

By default, wolfSSL provides debug output through stderr. In order for debug messages to be enabled, wolfSSL must be compiled with `DEBUG_WOLFSSL` defined, and `wolfSSL_Debugging_ON()` must be called from the application code. `wolfSSL_Debugging_OFF()` may be used by the application layer to turn off wolfSSL debug messages.

For environments which do not have stderr available, or wish to output debug messages over a different output stream or in a different format, wolfSSL allows applications to register a logging callback.

Please read Section 8.1 for details.

H.4.12 Public Key Operations

Q: When do I need to read this section?

A: You want to use your own public key implementation with wolfSSL.

wolfSSL allows users to write their own public key callbacks which will be called when the SSL/TLS layer needs to do public key operations. The user can optionally define 6 functions:

1. ECC sign callback
2. ECC verify callback
3. RSA sign callback
4. RSA verify callback
5. RSA encrypt callback
6. RSA decrypt callback

For full details, please read Section 6.4.

H.4.13 Atomic Record Layer Processing

Q: When do I need to read this section?

A: You want to do your own processing of record layers, specifically MAC/encrypt and decrypt/verify operations.

By default, wolfSSL handles record layer processing for the user using its cryptography library, wolfCrypt. wolfSSL provides Atomic Record Processing callbacks for users who wish to have more control over MAC/encrypt and decrypt/verify functionality during the SSL/TLS connection.

The user will need to define 2 functions:

1. MAC/encrypt callback function
2. Decrypt/verify callback function

For full details, please read Section 6.3.

H.4.14 Features

Q: When do I need to read this section?

A: When you want to disable features.

COPYRIGHT ©2024 wolfSSL Inc.

Features can be disabled when building wolfSSL by using the appropriate defines. For a list of defines available, please refer to Chapter 2.

H.5 Next Steps

H.5.1 wolfCrypt Test Application

After getting wolfSSL proper to build on the target platform, a good next step is to port the wolfCrypt test application. Running this application on the target system will verify that all the crypto algorithms are working correctly, using NIST test vectors.

If this step is skipped, and you instead proceed directly to establishing an SSL connection, it can be more difficult to debug problems caused by underlying crypto operations failing.

The wolfCrypt test application is located in `./wolfcrypt/test/test.c`. If an embedded application has its own `main()` function, then `NO_MAIN_DRIVER` must be defined when compiling `./wolfcrypt/test/test.c`. This will allow the application's `main()` to call each cipher/algorithm test individually on its own.

If an embedded device does not have enough resources to run the entire wolfCrypt test application, individual tests can be broken out of `test.c` and compiled individually. Please ensure that correct header files needed for the specific test case are included in the build when extracting isolated crypto tests from `test.c`.

H.6 Support

General support questions may be sent directly to wolfSSL either through email, support forums, or wolfSSL's Zendesk ticket tracking system.

Website: https://www.wolfssl.com

Support Email: support@wolfssl.com

Zendesk: https://wolfssl.zendesk.com

Forums: https://www.wolfssl.com/forums

wolfSSL offers several support packages as well as consulting services to help users and customers port wolfSSL to new environments.

Support Packages: https://www.wolfssl.com/wolfSSL/Support/support_tiers.php

Consulting Services: https://www.wolfssl.com/wolfSSL/wolfssl-consulting.html

General Inquiries: facts@wolfssl.com

I wolfSM (ShangMi)

This appendix provides information about the Chinese National Standard's cryptographic algorithms known as ShangMi (SM) in wolfSSL.

wolfSM support includes: * SM3 - Hash Function * SM4 - Cipher * SM2 - ECDH key agreement and a signature scheme using the specified 256-bit elliptic curve.

The code must be installed into wolfSSL in order to be used.

Note that the test and build configuration code is already in wolfSSL.

I.1 Getting and Installing wolfSM

I.1.1 Get wolfSM from GitHub

Clone the wolfSM repository from GitHub:

```
git clone https://github.com/wolfssl/wolfsm.git
```

I.1.2 Get wolfSSL from GitHub

wolfSSL is needed to build and test the SM algorithm implementations. Checkout the wolfSSL repository from GitHub beside wolfSM:

```
# Directory structure should be:
# <install-dir>
# ├── wolfsm
# └── wolfssl

cd .. # To directory containing wolfsm
git clone https://github.com/wolfssl/wolfssl.git
```

I.1.3 Install SM code into wolfSSL

To install the SM code into wolfSSL, use the install script:

```
cd wolfsm
./install.sh
```

I.2 Building wolfSM

Once the wolfSM files have been installed into wolfSSL, you can build SM algorithms into wolfSSL.

Choose which algorithms you require on the configure line: `* --enable-sm3 * --enable-sm4-ecb * --enable-sm4-cbc * --enable-sm4-ctr * --enable-sm4-gcm * --enable-sm4-ccm * --enable-sm2`

For example, to include SM3, SM4-GCM and SM2:

```
./autogen.sh
./configure --enable-sm3 --enable-sm4-gcm --enable-sm2
make
sudo make install
```

I.2.1 Optimized SM2

To use optimized implementations of SM2 you can either use C only code or C code with the faster assembly code.

For C code only: `--enable-sp` For C and assembly code: `--enable-sp --enable-sp-asm`

Optimized C code is available for 32 and 64 bit CPUs.

Assembly code is available for the following platforms: * Intel x64 * Aarch64 * ARM 32-bit * ARM Thumb2 * ARM Thumb

I.3 Testing wolfSM

To test that the SM ciphers are working use the following command:

```
make test
```

To benchmark the algorithms enabled:

```
./wolfcrypt/benchmark/benchmark
```

To benchmark specific algorithms, add to the command line the option/s matching the algorithm/s: * SM2: `-sm2` * SM3: `-sm3` * SM4: `-sm4` or * SM4-CBC: `-sm4-cbc` * SM4-GCM: `-sm4-gcm` * SM4-CCM: `-sm4-ccm`

I.3.1 Testing TLS

SM ciphers are able to be used with TLSv1.2 and TLSv1.3.

Note: SM2, SM3 and at least one SM4 cipher must be built in order for SM ciphers suite to work. All algorithms must be SM.

The cipher suites added are: - ECDHE-ECDSA-SM4-CBC-SM3 (TLSv1.2, `--enable-sm2 --enable-sm3 --enable-sm4-cbc`) - ECDHE-ECDSA-SM4-GCM-SM3 (TLSv1.2, `--enable-sm2 --enable-sm3 --enable-sm4-gcm`) - ECDHE-ECDSA-SM4-CCM-SM3 (TLSv1.2, `--enable-sm2 --enable-sm3 --enable-sm4-ccm`) - TLS13-SM4-GCM-SM3 (TLSv1.3, `--enable-sm2 --enable-sm3 --enable-sm4-gcm`) - TLS13-SM4-CCM-SM3 (TLSv1.3, `--enable-sm2 --enable-sm3 --enable-sm4-ccm`)

I.3.1.1 Example of using SM cipher suites with TLSv1.2 An example of testing TLSv1.2 with "ECDHE-ECDSA-SM4-CBC-SM3" cipher suite:

```
./examples/server/server -v 3 -l ECDHE-ECDSA-SM4-CBC-SM3 \
    -c ./certs/sm2/server-sm2.pem -k ./certs/sm2/server-sm2-priv.pem \
    -A ./certs/sm2/client-sm2.pem -V &
./examples/client/client -v 3 -l ECDHE-ECDSA-SM4-CBC-SM3 \
    -c ./certs/sm2/client-sm2.pem -k ./certs/sm2/client-sm2-priv.pem \
    -A ./certs/sm2/root-sm2.pem -C
```

The output using the commands above will be:

```
SSL version is TLSv1.2
SSL cipher suite is TLS_ECDHE_ECDSA_WITH_SM4_CBC_SM3
SSL curve name is SM2P256V1
SSL version is TLSv1.2
SSL cipher suite is TLS_ECDHE_ECDSA_WITH_SM4_CBC_SM3
SSL curve name is SM2P256V1
Client message: hello wolfssl!
I hear you fa shizzle!
```

I.3.1.2 Example of using SM cipher suites with TLSv1.3 An example of testing TLSv1.3 with "TLS13-SM4-GCM-SM3" cipher suite:

```
./examples/server/server -v 4 -l TLS13-SM4-GCM-SM3 \
    -c ./certs/sm2/server-sm2.pem -k ./certs/sm2/server-sm2-priv.pem \
    -A ./certs/sm2/client-sm2.pem -V &
./examples/client/client -v 4 -l TLS13-SM4-GCM-SM3 \
    -c ./certs/sm2/client-sm2.pem -k ./certs/sm2/client-sm2-priv.pem \
    -A ./certs/sm2/root-sm2.pem -C
```

The output using the commands above will be:

```
SSL version is TLSv1.3
SSL cipher suite is TLS_SM4_GCM_SM3
SSL curve name is SM2P256V1
SSL version is TLSv1.3
SSL cipher suite is TLS_SM4_GCM_SM3
SSL curve name is SM2P256V1
Client message: hello wolfssl!
I hear you fa shizzle!
```