wolfSSL Documentation

2025-04-22

Contents

1 序章

このマニュアルは、wolfSSL 組み込み SSL/TLS ライブラリの技術ガイドとして書かれており、wolfssl のビルド、実行方法、ビルドオプションの概要、提供機能、移植性の向上の方法、サポートなどに関する情報を提供します。

このドキュメントの PDF バージョンはこちらを参照下さい。

1.1 なぜ wolfSSL を選ぶのですか？

組み込み機器向けの SSL ソリューションとして wolfSSL を選択する理由はたくさんあります。最大の理由としてはそのサイズ (典型的なフットプリントサイズが 20~100 KB の範囲) と、サポートであるといえるでしょう。加えて、最新の標準 (SSL 3.0、TLS 1.0、TLS 1.1、TLS 1.2、TLS 1.3、DTLS 1.0、DTLS 1.2 および DTLS1.3) に準拠している点、先進的な暗号アルゴリズムのサポート (ストリーム暗号を含む)、マルチプラットフォーム、ロイヤリティフリー、および OpenSSL パッケージを使用した既存のアプリケーションの移植を容易にするための OpenSSL 互換性 API の提供などが理由として挙げられます。完全な機能リストは、機能の概要を参照下さい。

2 wolfSSL のビルド

wolfSSL はポータビリティを念頭に置いて書かれており、一般的にほとんどのシステムで容易にビルドできます。wolfSSL をビルドするのが難しい場合は、サポートフォーラム (https://www.wolfssl.com/forums) を通してサポートを求めるか、support@wolfssl.comに遠慮なく直接お問い合わせください。

この章では、Unix と Windows で wolfSSL をビルドする方法を説明し、非標準環境で wolfSSL をビルドする場合のガイダンスも提供します。第 3 章に入門ガイド、第 11 章に SSL チュートリアルをご用意しています。

autoconf/automake システムを使用して wolfSSL をビルドする場合、単一の MakeFile を使用して wolfSSL ライブラリの全てのパーツとサンプルプログラムを一度にビルドします。

2.1 wolfSSL ソースコードの取得

最新バージョンの wolfSSL は、wolfSSL の Web サイトから ZIP ファイルとしてダウンロードできます。

https://wolfssl.jp/download/

zip ファイルをダウンロードしたら、unzip コマンドを使用してファイルを解凍します。ネイティブの行末文字を使用するには、unzip を使用する際に-a オプションを有効にします。[unzip man page] で、-a オプションが説明されています。

> [...] -a オプションでは、zip によってテキストファイル（ZipInfo リストで「b」ではなく「t」ラベルがついているファイル）と識別されると行末文字、ファイル終端文字を変換します。必要に応じて指定してください。[...]

注：wolfSSL 2.0.0RC3 のリリースから、wolfSSL のディレクトリ構造と標準のインストールフォルダーが変更されました。これらの変更は、オープンソースプロジェクトが wolfSSL を統合しやすくするために行われました。ヘッダーと構造の変更の詳細については、第 9 章の「ライブラリヘッダー」および「構造の使用」節を参照してください。

2.2 *nix 上でビルド

Linux、*BSD、OS X、Solaris、またはその他の *nix のようなシステムに wolfSSL をビルドする場合、autoconf システムを使用します。wolfSSL をビルドするには、wolfSSL ルートディレクトリから 2 つのコマンド ./configure および make を実行するだけです。

./configure スクリプトはビルド環境をセットアップし、任意の数のビルドオプションを ./configure に追加します。利用可能なビルドオプションのリストについては、この章のビルドオプション節を参照するか、次のコマンドラインを実行して、./configure に渡すことができるオプションのリストを表示させてください。

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

./configure が正常に実行されたら、wolfSSL をビルドするために make を実行してください。

```
make
```

wolfSSL をインストールするには以下を実行します。

```
make install
```

インストールするには特権を必要とする場合もあるかもしれません。その場合は次のように sudo をコマンドの前に加えてください。

```
sudo make install
```

ビルドをテストするには、wolfSSL のルートディレクトリから testsuite プログラムを実行します。

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


あるいは、autoconf を使用して testsuite を実行し、標準の wolfSSL API および暗号テストを実行します。

```
make test
```

testsuite プログラムの予想される出力に関する詳細は、第 3 章のテストスイート節にあります。

もし、wolfSSL ライブラリのみのビルドが必要で、そのほかのコンポーネント（サンプルプログラム、テストスイート、ベンチマークアプリケーション等）はビルドしたくない場合には、wolfSSL のルートディレクトリで下記コマンドを実行してください、

```
make src/libwolfSSL.la
```

2.3 Windows の上でのビルド

以下の説明に加えて、Visual Studio で wolfSSL をビルドするための手順とヒントはここでも提供しています。

2.3.1 Visual Studio 2008

インストールのルートディレクトリに Visual Studio 2008 用のソリューションが含まれています。Visual Studio 2010 以降で使用する場合、既存のプロジェクトファイルはインポートプロセス中に変換できるはずです。

注意: 新しいバージョンの Visual Studio にインポートする場合、「プロジェクトとインポートされたプロパティシートを上書きしますか？」と尋ねられます。「いいえ」を選択すると、以下の問題を回避できます。「はい」を選択すると、SAFESEH 指定のために EDITANDCONTINUE が無視されるという警告が表示されます。testsuite、sslSniffer、server、echoserver、echoclient、client をそれぞれ右クリックして、プロパティ-> 構成プロパティ-> リンカー->Advanced（Advanced ウィンドウの一番下までスクロール）を変更する必要があります。「安全な例外ハンドラーを含むイメージ」を見つけ、右端の下矢印をクリックします。これを前述の各プロジェクトに対して「いいえ」（/SAFESEH:NO）に変更します。もう一つの選択肢は EDITANDCONTINUE を無効にすることですが、デバッグ目的では有用であるため推奨されません。

2.3.2 Visual Studio 2010

wolfSSL ソリューションをビルドするには、Service Pack 1 をダウンロードする必要があります。Visual Studio がリンカエラーを報告する場合は、プロジェクトをクリーンしてリビルドすると解消するはずです。

2.3.3 Visual Studio 2013 (64 ビットソリューション)

wolfSSL ソリューションをビルドするには、Service Pack 4 をダウンロードする必要があります。Visual Studio がリンカエラーを報告する場合は、プロジェクトをクリーンしてリビルドすると解消するはずです。

各ビルドをテストするには、Visual Studio メニューから「Build All」を選択し、testsuite プログラムを実行します。Visual Studio プロジェクトのビルドオプションを編集するには、目的のプロジェクト（wolfSSL、echoclient、echoserver など）を選択し、「プロパティ」パネルを参照してください。

注意: wolfSSL v3.8.0 リリース以降、ビルドプリプロセッサマクロは IDE/WIN/user_settings.h にある中央ファイルに移動しました。このファイルはプロジェクト内でも見つけることができます。ECC や ChaCha20/Poly1305 などの機能を追加するには、HAVE_ECC や HAVE_CHACHA / HAVE_POLY1305 などの #defines をここに追加します。

2.3.4 Cygwin

Windows 開発マシン上で Windows 用の wolfSSL をビルドする場合は、リポジトリに含まれる Visual Studio プロジェクトファイルを使用してビルドすることをお勧めします。しかし Cygwin が必要な場合は、当社チームがビルドに成功した際のガイダンスがありますから以下にご紹介します。

1. https://www.cygwin.com/install.html にアクセスして setup-x86_64.exe をダウンロードします、
2. setup-x86_64.exe を実行し、希望する方法でインストールします。「Select Packages」段階に到達するまでインストールメニューをクリックします。
3.「+」アイコンをクリックして「All」を展開します、
4.「Archive」セクションに移動し、「unzip」ドロップダウンを選択して、「Skip」を 6.0-15（または他のバージョン）に変更します。
5.「Devel」の下で「autoconf」ドロップダウンをクリックし、「Skip」を「10-1」（または他のバージョン）に変更します、
6.「Devel」の下で「automake」ドロップダウンをクリックし、「Skip」を「10-1」（または他のバージョン）に変更します。
7.「Devel」の下で「gcc-core」ドロップダウンをクリックし、「Skip」を 7.4.0-1 に変更します。（注：wolfSSL は GCC 9 または 10 をテストしておらず、比較的新しいものであるため、開発用に微調整する時間が多少経つまで使用することをお勧めしません）
8.「Devel」の下で「git」ドロップダウンをクリックし、「Skip」を 2.29.0-1（または他のバージョン）に変更します。
9.「Devel」の下で「libtool」ドロップダウンをクリックし、「Skip」を「2.4.6-5」（または他のバージョン）に変更します。
10.「Devel」の下で「make」ドロップダウンをクリックし、「Skip」を 4.2.1-1（または他のバージョン）に変更します。
11.「Next」をクリックして、インストールの残りの部分を進めます。

追加のパッケージとして、以下が必要です。

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

2.3.4.1 インストール後の作業

Cygwin ターミナルを開き、wolfSSL をクローンします。

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

2.4 非標準環境でのビルド

公式にはサポートしていませんが、特に組み込み向けややクロスコンパイルシステムを使用して、非標準環境で wolfSSL をビルドしたいユーザーを支援するよう努めています。以下にいくつかの注意点を示します。

1. ソースファイルとヘッダーファイルは、wolfSSL ダウンロードパッケージ内にあるのと同じディレクトリ構造を維持する必要があります。
2. 一部のビルドシステムでは、wolfSSL ヘッダーファイルの場所を明示的に知りたい場合があるため、それを指定する必要があるかもしれません。それらは <wolfSSL_root>/wolfSSL ディレクトリにあります。通常は、インクルードパスに <wolfSSL_root> ディレクトリを追加してください。
3. wolfSSL は、configure プロセスがビッグエンディアンを検出しない限り、デフォルトでリトルエンディアンシステムとなります。非標準環境でビルドするユーザーは configure プロセスを使用していないため、ビッグエンディアンシステムを使用している場合は BIG_ENDIAN_ORDER を定義する必要があります。

4. wolfSSL は 64 ビット型を利用できると速度が向上します。configure プロセスは long または long long が 64 ビットかどうかを判断し、define を設定します。したがって、システム上で `sizeof(long)` が 8 バイトの場合は、`SIZEOF_LONG 8` を定義してください。`sizeof(long long)` が 8 バイトの場合は、`SIZEOF_LONG_LONG 8` を定義します。
5. ライブラリのビルドで問題が発生した場合は、お知らせください。サポートが必要な場合は、support@wolfssl.com までご連絡ください。
6. ビルドを変更できる定義は、本章の後の節に示しています。多くのオプションの詳細な説明については、ビルドオプション節を参照してください。

2.4.1 Yocto Linux へのビルド

wolfSSL は Yocto Linux と OpenEmbedded で wolfSSL をビルドするためのレシピも含んでいます。これらのレシピは meta-wolfSSL レイヤーとして GitHub リポジトリで管理しています。

https://github.com/wolfSSL/meta-wolfSSL

Yocto Linux で wolfSSL をビルドするには Git と bitbake が必要です。以下に、Yocto Linux 上で wolfSSL 製品（レシピが存在するもの）をビルドする方法を説明します。

1. **wolfSSL meta をクローン**

   ```
   git clone https://github.com/wolfSSL/meta-wolfSSL
   ```

2. **「meta-wolfSSL」レイヤーをビルドの bblayers.conf に挿入**

   BBLAYERS セクション内で、meta-wolfSSL がクローン作成された場所へのパスを追加します。

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

3. **wolfSSL 製品レシピをビルド**

   bitbake を使用して、*wolfSSL*、*wolfssh*、*wolfmqtt* をビルドします。これらのレシピのいずれかを bitbake コマンドに渡すだけです。（例：`bitbake wolfSSL`）これにより、ユーザーはコンパイルが問題なく成功することを確認できます。

4. **local.conf を編集**

   最後のステップは、ビルドの `local.conf` ファイルを編集して、ビルド中のイメージに必要なライブラリを含めることです。希望するレシピの名前を含むように `IMAGE_INSTALL_append` 行を編集します。以下にその例を示します。

   ```
   IMAGE_INSTALL_append="wolfSSL wolfssh wolfmqtt"
   ```

イメージがビルドされると、wolfSSL のデフォルトの場所（またはレシピからの関連製品）は`/usr/lib/`ディレクトリになります。

さらに、wolfSSL は後のビルドオプション節にリストされている有効化および無効化オプションを使用して、Yocto にビルドする際にカスタマイズできます。これには、.bbappend ファイルを作成し、wolfSSL アプリケーション/レシピレイヤー内に配置する必要があります。このファイルの内容には、`EXTRA_OECONF` 変数に連結する内容を指定する行を含める必要があります。以下に、TLS 1.3 有効化オプションを通じて TLS 1.3 サポートを有効にする例を示します。

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

Yocto へのビルドに関する詳細なドキュメントは、meta-wolfSSL README に掲載しています。

https://github.com/wolfSSL/meta-wolfSSL/blob/master/readme.md

2.4.2 Atollic TrueSTUDIO でのビルド

3.15.5 以降の wolfSSL バージョンには、ARM M4-Cortex デバイス上で wolfSSL をビルドするために使用される TrueSTUDIO プロジェクトファイルが含まれています。TrueSTUDIO プロジェクトファイルは、ST Microelectronics の一部である Atollic によって作成されたもので、無料でダウンロードできます。これを使用することで、STM32 デバイス上でのビルドプロセスを簡素化できます。TrueSTUDIO で wolfSSL 静的ライブラリプロジェクトファイルをビルドするには、TrueSTUDIO を開いた後、ユーザーが以下の手順を実行する必要があります。

1. プロジェクトをワークスペースにインポートする（ファイル > インポート）
2. プロジェクトをビルドする（プロジェクト > プロジェクトのビルド）

ビルド時に `user_settings.h` 内にある設定がインクルードされます。`user_settings.h` ファイルのデフォルトの内容は最小限であり、多くの機能は含まれていません。ユーザーはこのファイルを変更し、後の章に示すオプションで機能を追加または削除できます。

2.4.3 IAR でのビルド

`<wolfSSL_root>/IDE/IAR-EWARM` ディレクトリには、以下のファイルが含まれています。

1. ワークスペース：`wolfSSL.eww` ワークスペースには、wolfSSL-Lib ライブラリと wolfCrypt-test、wolfCrypt-benchmark 実行可能プロジェクトが含まれています。
2. wolfSSL-Lib プロジェクト：`lib/wolfSSL-lib.ewp` wolfCrypt と wolfSSL 関数の完全なセットライブラリを生成します。
3. テストスイートプロジェクト：`test/wolfCrypt-test.ewp` test.out テストスイート実行可能ファイルを生成します。
4. ベンチマークプロジェクト：`benchmark/wolfCrypt-benchmark.ewp` benchmark.out ベンチマーク実行可能ファイルを生成します。

これらのプロジェクトは一般的な ARM Cortex-M MPU 向けに設定されています。特定のターゲット MPU 用のプロジェクトを生成するには、以下の手順を実行してください。

1. デフォルト設定：プロジェクトのデフォルトターゲットは Cortex-M3 シミュレータに設定されています。user_settings.h にはプロジェクトのデフォルトオプションが含まれています。シミュレータに対してビルドしてダウンロードできます。「view」->「Terminal I/O」でターミナル I/O ウィンドウを開き、実行を開始します。
2. プロジェクトオプション設定：各プロジェクトに適切な「ターゲット」オプションを選択します。
3. 実行可能プロジェクトの場合：MPU 用の「SystemInit」と「startup」を追加し、デバッグ用の「ドライバ」を選択します。
4. ベンチマークプロジェクトの場合：current_time 関数のオプションを選択するか、`WOLFSSL_USER_CURRTIME` オプションで独自の「current_time」ベンチマークタイマーを作成します。
5. ビルドとダウンロード：EWARM ビルドとダウンロードのために、メニューバーの「Project->Make」と「Download and Debug」に進みます。

2.4.4 macOS や iOS でのビルド

2.4.4.1 Xcode

`<wolfSSL_root>/IDE/XCODE` ディレクトリには、以下のファイルが含まれています：

1. `wolfSSL.xcworkspace` – ライブラリとテストスイートクライアントを含むワークスペース
2. `wolfSSL_testsuite.xcodeproj` – テストスイートを実行するプロジェクト
3. `wolfSSL.xcodeproj` – wolfSSL および/または wolfCrypt 用の OS/x と iOS ライブラリをビルドするプロジェクト
4. `wolfSSL-FIPS.xcodeproj` – 利用可能な場合、wolfSSL と wolfCrypt-FIPS をビルドするプロジェクト

5. user_settings.h – プロジェクト間で共有されるカスタムライブラリ設定

ライブラリは、ターゲットに応じて libwolfSSL_osx.a または libwolfSSL_ios.a として出力されます。また、wolfSSL/wolfCrypt（および CyaSSL/CtaoCrypt 互換性）ヘッダーを Build/Products/Debug または Build/Products/Release にある include ディレクトリにコピーします。

ライブラリとテストスイートを適切にリンクするには、ビルドの場所をワークスペースに対して相対的に設定する必要があります。

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

これらの Xcode プロジェクトは、複数のプロジェクトでマクロを設定するための user_settings.h ファイルを有効にするために、WOLFSSL_USER_SETTINGS プリプロセッサを定義しています。

必要に応じて、Xcode プリプロセッサは以下の手順で変更できます。

1.「Project Navigator」でプロジェクトをクリックします。
2.「Build Settings」タブをクリックします。
3.「Apple LLVM 6.0 - Preprocessing」セクションまでスクロールします。
4.「Preprocessor Macros」の開示を開き、「+」と「-」ボタンを使用して変更します。これを Debug と Release の両方に対して行ってください。

このプロジェクトは、デフォルト設定を使用して wolfSSL と wolfCrypt をビルドする必要があります。

2.4.5 GCC ARM でのビルド

<wolfSSL_root>/IDE/GCC-ARM ディレクトリには、Cortex M シリーズ用の wolfSSL プロジェクト例がありますが、他のアーキテクチャに適用することもできます。

1. gcc-arm-none-eabi がインストールされていることを確認してください。
2. Makefile.common を変更します：

- 正しいツールチェーンパス TOOLCHAIN を使用します。
- 正しいアーキテクチャ「ARCHFLAGS」を使用します。GCC ARM Options -mcpu=name を参照してください。
- linker.ld のメモリマップがフラッシュ/RAM と一致することを確認するか、Makefile.common の SRC_LD = -T./linker.ld をコメントアウトします。

3. make を使用して、静的ライブラリ（libwolfSSL.a）、wolfCrypt テスト/ベンチマーク、および wolfSSL TLS クライアントターゲットを/Build 内の.elf と.hex としてビルドします。

2.4.5.1 一般的な Makefile を使ったクロスコンパイルによるビルド Cortex-A53 を搭載した Raspberry Pi 用の Makefile.common 変更例を示します。

1. Makefile.common 内の ARCHFLAGS を-mcpu=cortex-a53 -mthumb に変更します。
2. カスタムメモリマップが適用されないため、SRC_LD をコメントアウトします。
3. TOOLCHAIN をクリアしてデフォルト gcc を使用します。“TOOLCHAIN =” とします。
4. LDFLAGS += --specs=nano.specs と LDFLAGS += --specs=nosys.specs をコメントアウトしてください。

2.4.5.2 configure でのクロスコンパイルによるビルド メインプロジェクトディレクトリの構成スクリプトは、gcc-arm-none-eabi ツールでビルドするようにクロスコンパイルを実行できます。以下の例ではツールへのパスが適切に設定されていると仮定します。

```
./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
```

32 ビットアーキテクチャをビルドしている場合は、-DTIME_T_NOT_64BIT を CFLAGS に追加してください。

2.4.6 Keil MDK-ARM でのビルド

Keil MDK-ARM で wolfSSL をビルドするための詳細な手順とヒントはこちらで見つけることができます。

注意：MDK-ARM がデフォルトではない場所にインストールされている場合、プロジェクトファイルの参照パス定義をすべて変更する必要があります。

2.5 C プリプロセッサマクロとして定義される機能

2.5.1 機能の削除

以下の定義は、wolfSSL から機能を削除するために使用できます。これは、ライブラリ全体のフットプリントサイズを削減しようとしている場合に役立ちます。NO_< 機能名 > 定義を定義することに加えて、ビルドから対応するソースファイルも削除できます。ヘッダーファイルは対象外です。

2.5.1.1 NO_WOLFSSL_CLIENT サーバーのみのビルド用です。クライアントに固有の呼び出しを削除します。サイズのためにいくつかの呼び出しを削除する場合にのみ、これを使用します。

2.5.1.2 NO_WOLFSSL_SERVER クライアントのみのビルド用です。サーバーに固有の呼び出しを削除します。サイズのためにいくつかの呼び出しを削除する場合にのみ、これを使用します。

2.5.1.3 NO_DES3 DES3 暗号化の使用を削除します。一部の古いサーバーがそれを使用しているため、DES3 はデフォルトで組み込まれており、SSL 3.0 では必要です。NO_DH および NO_AES は上記 2 つと同じで、広く使用されています。

2.5.1.4 NO_DSA DSA の使用を削除します。DSA は、一般的な使用が段階的に廃止されています。

2.5.1.5 NO_ERROR_STRINGS エラー文字列を無効にします。エラー文字列は、wolfSSL の場合は src/internal.c または wolfCrypt の wolfcrypt/src/asn.c にあります。

2.5.1.6 NO_HMAC ビルドから HMAC を削除します。

注意：SSL/TLS は HMAC に依存します。wolfCrypt のみを使用している場合、（ビルドオプション WOLFCRYPT_ONLY）HMAC を無効にできます。

2.5.1.7 NO_MD4 ビルドから MD4 を削除します。MD4 は解読されているため、使用してはなりません。

2.5.1.8 NO_MD5 ビルドから MD5 を削除します。

2.5.1.9 NO_SHA ビルドから SHA-1 を削除します。

2.5.1.10 NO_SHA256 ビルドから SHA-256 を削除します。

2.5.1.11 NO_PSK 事前共有キー拡張機能の使用をオフにします。デフォルトでは組み込まれています。

2.5.1.12 NO_PWDBASED PKCS # 12 から PBKDF1、PBKDF2、PBKDF などのパスワードベースの鍵導出関数を無効にします。

2.5.1.13 NO_RC4 ビルドから ARC4 ストリーム暗号の使用を削除します。ARC4 はまだ人気があり広く使用されているため、デフォルトで組み込まれています。

2.5.1.14 NO_SESSION_CACHE セッションキャッシュが不要なときに定義できます。これにより、メモリ使用量を約 3KB 減らすことができます。

2.5.1.15 NO_TLS TLS をオフにします。TLS をオフに設定することはお勧めしません。

2.5.1.16 SMALL_SESSION_CACHE wolfSSL が使用する SSL セッションキャッシュのサイズを制限するように定義できます。これにより、デフォルトのセッションキャッシュが 33 セッションから 6 セッションに短縮され、約 2.5KB 削減できます。

2.5.1.17 NO_RSA RSA アルゴリズムのサポートを削除します。

2.5.1.18 WC_NO_RSA_OAEP OAEP パディングのコードを削除します。

2.5.1.19 NO_AES_CBC AES-CBC アルゴリズムサポートをオフにします。

2.5.1.20 NO_AES_DECRYPT コードサイズを削減するために設定できます。復号を無効にし、暗号化のみをサポートするように設定します。

2.5.1.21 WOLFCRYPT_ONLY TLS を無効にして wolfCrypt のみを有効にします。

2.5.1.22 NO_CAMELLIA_CBC Camellia CBC サポートを無効にします。TLS 暗号スイートにのみ適用されます。

2.5.1.23 NO_AES AES アルゴリズムサポートを無効にします。

2.5.1.24 NO_AES_128 コンパイル時の AES キーサイズ選択に使用されます。

2.5.1.25 NO_AES_192 コンパイル時の AES キーサイズ選択に使用されます。

2.5.1.26 NO_AES_256 コンパイル時の AES キーサイズ選択に使用されます。

2.5.1.27 NO_AESGCM_AEAD AES GCM を使用する TLS 暗号スイートを無効にします。AES GCM 暗号スイートが有効になっていない場合に内部的に使用するものですが、暗号スイートを制限するためにも使用できます。

2.5.1.28 NO_ASN_TIME ASN の時間チェックを無効にします。

注意：すべての証明書の開始/終了日チェックがスキップされるため、これは注意して使用する必要があります。

2.5.1.29 NO_CHECK_PRIVATE_KEY このマクロは、デフォルトでオンになっている追加の秘密鍵チェックを無効にします。これにより、秘密鍵が公開鍵のペアであることを検証するチェックが有効になります。RSA、ECDSA、ED25519、ED448、Falcon、Dilithium および Sphincs でサポートされています。

2.5.1.30 NO_DH Diffie-Hellman（DH）サポートを無効にします。

2.5.1.31 NO_ED25519_CLIENT_AUTH ED25519 の TLS クライアント認証サポートを無効にします。メッセージのキャッシュが必要なため、ED25519 が使用されていない場合、TLS 中のメモリ使用量を削減するために使用されます。

2.5.1.32 NO_ED448_CLIENT_AUTH ED448 のクライアント認証を無効にします。

2.5.1.33 NO_FORCE_SCR_SAME_SUITE デフォルトでは、セキュアな再ネゴシエーションでは同じ暗号スイートを使用する必要があります。これはその要件を無効にします。

2.5.1.34 NO_MULTIBYTE_PRINT 組み込みデバイスが問題を抱える可能性のある、マルチバイト文字をコンパイルアウトするために使用されます。

2.5.1.35 NO_OLD_SSL_NAMES wolfSSL と OpenSSL を一緒に使用するための、古い OpenSSL 互換性マクロの一部を無効にします。

2.5.1.36 NO_OLD_WC_NAMES 不要な名前空間を削除します。

2.5.1.37 NO_OLD_POLY1305 相互運用性のために使用される、古い ChaCha20/Poly1305 TLS 1.2 暗号スイートのサポートを無効にします。

2.5.1.38 NO_HANDSHAKE_DONE_CB `wolfSSL_SetHsDoneCb` で設定されるハンドシェイクコールバックのサポートを無効にします。このオプションはコードサイズを削減するのに役立ちます。

2.5.1.39 NO_STDIO_FILESYSTEM これは stdio.h のインクルードを無効にします。移植性を保つために使用されます。

2.5.1.40 NO_TLS_DH TLS DH を除外します。一時的な有限体 Diffie-Hellman 鍵合意に基づく暗号スイートをネゴシエートすべきではありません。

2.5.1.41 NO_WOLFSSL_CM_VERIFY 証明書マネージャの検証コールバックを無効にします。検証コールバックはエラーをインターセプトして上書きすることを可能にします。このオプションはコードサイズを削減するのに役立ちます。

2.5.1.42 NO_WOLFSSL_DIR ディレクトリサポートを無効にします。

2.5.1.43 NO_WOLFSSL_RENESAS_TSIP_TLS_SESSION TSIP TLS リンク共通鍵暗号化方式のみを無効します。

注意：これはルネサス RX TSIP 固有の定義です。

2.5.1.44 NO_WOLFSSL_SHA256 これは TLS 1.3 にのみ適用されます。SHA2-256 を wolfCrypt から有効にして使用できるようにしますが、TLS 1.3 からは除外します。

2.5.1.45 WOLFSSL_BLIND_PRIVATE_KEY 秘密鍵をブラインドするためのマスクとして使用されます。ブラインド処理はロウハンマー攻撃から保護するために使用されます。

2.5.1.46 WOLFSSL_DTLS13_NO_HRR_ON_RESUME これが定義されている場合、DTLS サーバーはクライアントの再開が成功した場合にクッキー交換を行いません。

再開はより速く（1 RTT 少なく）なり、帯域幅の消費が少なくなります。（1 つの ClientHello と、1 つの HelloVerifyRequest/HelloRetryRequest が少なくなります）一方、有効な SessionID/チケット/psk が収集された場合、偽造された clientHello メッセージはサーバー上のリソースを消費します。

DTLS 1.3 の場合、このオプションを使用すると、サーバーが Early Data/0-RTT Data を処理することもできます。これがなければ、サーバーはクッキーを持つ検証済み ClientHello を受信するまでステートフル処理に入らないため、Early Data はドロップされます。

クッキー交換なしで DTLS 1.3 再開を許可するには、次のようにします。

1. `WOLFSSL_DTLS13_NO_HRR_ON_RESUME` を定義して wolfSSL をコンパイルする
2. `wolfSSL_dtls13_no_hrr_on_resume(ssl, 1)` を WOLFSSL オブジェクトに呼び出して、再開時のクッキー交換を無効にする
3. 通常の接続と同様に続行する

2.5.1.47 WOLFSSL_NO_CLIENT_AUTH Ed25519 と Ed448 を使用するために必要な、キャッシングコードを無効にします。

2.5.1.48 WOLFSSL_NO_CURRDIR `wolfSSL/test.h` のテストパスに./をサポートしないプラットフォーム用の移植性マクロです。テストツールにのみ適用されます。

2.5.1.49 WOLFSSL_NO_DEF_TICKET_ENC_CB デフォルトのチケット暗号化コールバックを定義しません。サーバーのみに適用されます。アプリケーションはセッションチケットを使用するために独自のコールバックを設定する必要があります。

2.5.1.50 WOLFSSL_NO_SOCK 移植のため、組み込みソケットサポートを無効にするためのマクロです。ソケットなしで TLS を使用する場合、通常は `WOLFSSL_USER_IO` を定義し、送信/受信にコールバックを使用します。

2.5.1.51 WOLFSSL_NO_TLS12 TLS 1.2 を除外するために定義します。

2.5.1.52 WOLFSSL_PEM_TO_DER PEM から DER への変換を無効にします。

2.5.1.53 NO_DEV_URANDOM `/dev/urandom` の使用を無効にします。

2.5.1.54 WOLFSSL_NO_SIGALG 署名アルゴリズムの拡張子を無効にします。

2.5.1.55 NO_RESUME_SUITE_CHECK TLS 接続を再開するときに暗号スイートのチェックを無効にします。

2.5.1.56 NO_ASN ASN フォーマットの証明書処理のサポートをオフにします。

2.5.1.57 NO_OLD_TLS SSLV3、TLSV1.0、TLSV1.1 のサポートを削除します。

2.5.1.58 WOLFSSL_AEAD_ONLY 非 AEAD アルゴリズムのサポートを削除します。AEAD は、「認証された暗号化」の略であり、これらのアルゴリズム (AES-GCM など) がデータを暗号化および復号化するだけでなく、そのデータの機密性と信頼性を保証するアルゴリズムです。

2.5.1.59 WOLFSSL_SP_NO_2048 RSA/DH 2048 ビット単精度 (SP) 最適化を削除します。

2.5.1.60 WOLFSSL_SP_NO_3072 RSA/DH 3072 ビット単精度 (SP) 最適化を削除します。

2.5.1.61 WOLFSSL_SP_NO_256 SECP256R1 の ECC 単精度 (SP) 最適化を削除します。WOLFSSL_SP_MATH にのみ適用されます。

2.5.2 機能の有効化 (デフォルトで有効)

2.5.2.1 HAVE_TLS_EXTENSIONS ほとんどの TLS ビルドに必要な TLS 拡張機能のサポートを有効にします。

./configure を使用する場合これはデフォルトでオンになっていますが、WOLFSSL_USER_SETTINGS でビルドする場合は手動で定義する必要があります。

2.5.2.2 HAVE_SUPPORTED_CURVES TLS でサポートされている曲線と、TLS で使用されるキー共有拡張機能を有効にします。ECC、Curve25519、および Curve448 で必要です。

./configure を使用する場合これはデフォルトでオンになっていますが、WOLFSSL_USER_SETTINGS でビルドする場合は手動で定義する必要があります。

2.5.2.3 HAVE_EXTENDED_MASTER TLS v1.2 以前で使用されるセッションキーの計算用に拡張マスターシークレット PRF を有効にします。PRF 方式はデフォルトでオンになっており、より安全であると考えられています。

./configure を使用する場合これはデフォルトでオンになっていますが、WOLFSSL_USER_SETTINGS でビルドする場合は手動で定義する必要があります。

2.5.2.4 HAVE_ENCRYPT_THEN_MAC ブロック暗号による暗号化後に MAC を実行するための encrypt-then-mac サポートを有効にします。これがデフォルトで、セキュリティが向上します。

./configure を使用する場合これはデフォルトでオンになっていますが、WOLFSSL_USER_SETTINGS でビルドする場合は手動で定義する必要があります。

2.5.2.5 HAVE_ONE_TIME_AUTH Poly 認証を設定するため、TLS v1.2 で Chacha20/Poly1305 を使用する場合に必要です。

./configure を使用する場合、これは ChaCha20/Poly1305 でデフォルトで有効になりますが、WOLFSSL_USER_SETTINGS でビルドする場合は手動で定義する必要があります。

2.5.3 機能の有効化 (デフォルトで無効)

2.5.3.1 WOLFSSL_CERT_GEN wolfSSL の証明書生成機能をオンにします。詳細については、第 7 章 キーと証明書を参照してください。

2.5.3.2 WOLFSSL_DER_LOAD `wolfSSL_CTX_der_load_verify_locations()` 関数を使用して、wolfSSL コンテキスト (`WOLFSSL_CTX`) への DER フォーマットされた CA 証明書をロードできます。

2.5.3.3 WOLFSSL_DTLS DTLS(Datagram TLS) の使用をオンにします。これは広くサポートされていないか、使用されていません。

2.5.3.4 WOLFSSL_KEY_GEN wolfSSL の RSA 鍵生成機能をオンにします。詳細については、第 7 章 キーと証明書を参照してください。

2.5.3.5 WOLF_PRIVATE_KEY_ID これは PKCS11 で使用され、キー ID とラベル API のサポートを有効にします。FIPS v5 以前は暗号コールバックで `WOLF_PRIVATE_KEY_ID` をサポートしていません。

2.5.3.6 WOLFSSL_WOLFSENTRY_HOOKS `wolfSSL_CTX_set_AcceptFilter()` と `wolfSSL_CTX_set_ConnectFilter()` を使用した一般的なネットワーク `accept` および `connect` フィルターフックのサポートを TLS レイヤーに追加します。また、サンプルクライアントおよびサーバーアプリケーションでの wolfSentry 統合も有効化します。

2.5.3.7 WOLFSSL_CERT_EXT 証明書拡張、鍵および証明書生成機能です。

2.5.3.8 WOLFSSL_CERT_REQ 証明書要求、鍵および証明書生成機能です。

2.5.3.9 WOLFSSL_SSLKEYLOGFILE これは Wireshark で使用されるキーロギングを有効にします。マスターシークレットとクライアントランダムがファイルに書き込まれるため、コンパイラが警告を発するようになります。これはテストに役立ちますが、本番環境ではお勧めしません。

2.5.3.10 WOLFSSL_SSLKEYLOGFILE_OUTPUT このマクロはキーロギングのファイル名を定義します。`WOLFSSL_SSLKEYLOGFILE` と共に使用してください。

2.5.3.11 WOLFSSL_HAVE_WOLFSCEP wolfSCEP が利用可能かどうかを確認するために autoconf で使用される機能を有効にします。

2.5.3.12 WOLFSSL_HAVE_MIN このマクロはライブラリの移植性を高めるためのもので、MIN/MAX がすでにプラットフォームによって定義されているかどうかを示します。重複定義を防ぎます。

2.5.3.13 WOLFSSL_HAVE_TLS_UNIQUE 「tls-unique」チャネルバインディングとして使用するために、TLS ハンドシェイク後に「Finished」メッセージを保持します。

libest port で追加：アプリケーションが「tls-unique」チャネルバインディングタイプを取得できるようにします。

- RFC 5929 Section 3

これは EST プロトコルで使用され、「所有証明」を通じて TLS セッションに登録をバインドします。

- RFC 7030 Section 3.4
- RFC 7030 Section 3.5

2.5.3.14 WOLFSSL_ENCRYPTED_KEYS 暗号化キー PKCS8 サポートを有効にします。このマクロは PKCS8 パスワードベースのキー暗号化を有効にします。

- RFC 5208

2.5.3.15 WOLFSSL_CUSTOM_OID サブジェクトおよび要求拡張用のカスタム OID サポートを有効にする証明書機能です。これはカスタム OID を持つ証明書の解析にも適用されます。

2.5.3.16 WOLFSSL_RIPEMD RIPEMD-160 サポートを有効にします。

2.5.3.17 WOLFSSL_SHA384 SHA-384 サポートを有効にします。

2.5.3.18 WOLFSSL_SHA512 SHA-512 サポートを有効にします。

2.5.3.19 WOLFSSL_AES_DIRECT AES ECB モードのダイレクトサポートを有効にします。それ自体では ECB モードは安全とは見なされません。この機能は PKCS7 に必要です。

警告：ほぼすべてのユースケースで、ECB モードは安全性が低いと考えられています。可能な限り ECB API を直接使用することは避けてください。

2.5.3.20 DEBUG_WOLFSSL デバッグ機能を含めてビルドします。詳細は第 8 章 デバッグを参照してください。

2.5.3.21 HAVE_AESCCM AES-CCM サポートを有効にします。

2.5.3.22 HAVE_AESGCM AES-GCM サポートを有効にします。

2.5.3.23 WOLFSSL_AES_XTS AES-XTS サポートを有効にします。

2.5.3.24 HAVE_CAMELLIA Camellia サポートを有効にします。

2.5.3.25 HAVE_CHACHA Chacha20 サポートを有効にします。

2.5.3.26 HAVE_POLY1305 Poly1305 サポートを有効にします。

2.5.3.27 HAVE_CRL 証明書失効リスト (CRL) サポートを有効にします。

2.5.3.28 HAVE_CRL_IO CRL URL でインライン HTTP リクエストをブロックできるようにします。CRL を `WOLFSSL_CTX` にロードし、作成したすべての wolfSSL オブジェクトに適用します。

2.5.3.29 HAVE_ECC 楕円曲線暗号化 (ECC) サポートを有効にします。

2.5.3.30 HAVE_LIBZ 接続上のデータの圧縮を可能にする拡張機能です。通常、使用すべきではありません。「ビルドオプション」節にある`--with-libz=PATH` の下のメモを参照してください。

2.5.3.31 OPENSSL_EXTRA ライブラリにさらに多くの OpenSSL 互換性を組み込み、wolfSSL OpenSSL 互換性レイヤーを有効にします。これにより、OpenSSL で動作するように設計された既存のアプリケーションへの移植を容易にします。デフォルトでオフになっています。

2.5.3.32 HAVE_EXT_CACHE 内部キャッシュの代わりに外部セッションキャッシュを使用するためのサポートを有効にします。

2.5.3.33 WOLFSSL_WPAS_SMALL WPA サプリカントサポート用の互換性レイヤーの小さなサブセットを有効にします。

2.5.3.34 OPENSSL_ALL 統合テスト用のすべての互換性機能のサポートを有効にします。

2.5.3.35 OPENSSL_COEXIST OpenSSL 互換レイヤーです。古い名前を無効にする必要があります。wolfSSL と OpenSSL が共存できるようにするモードです。

2.5.3.36 OPENSSL_VERSION_NUMBER OpenSSL 互換性を実装するバージョン番号を指定します。

2.5.3.37 WOLFSSL_NGINX OpenSSL 互換性をもつアプリケーション固有のマクロです。(`--enable-nginx`) `WOLFSSL_NGINX` を使用します。

2.5.3.38 WOLFSSL_ERROR_CODE_OPENSSL OpenSSL 互換性 API `wolfSSL_EVP_PKEY_cmp` は成功時に 0、失敗時に-1 を返します。この動作は OpenSSL とは異なります。`EVP_PKEY_cmp` は以下を返します。

- 1: 2 つのキーが一致する
- 0: 一致しない
- -1: キータイプが異なる
- -2: 操作がサポートされていない

この関数が OpenSSL と同じ動作をするようにしたい場合は、`WS_RETURN_CODE` が OpenSSL と同等の動作に戻り値を変換するように、`WOLFSSL_ERROR_CODE_OPENSSL` を定義してください。

2.5.3.39 WOLFSSL_HARDEN_TLS RFC 9325で指定された推奨事項を実装します。このマクロは、望ましいセキュリティビット数に定義する必要があります。現在実装されている値は 112 ビットと 128 ビットです。以下のマクロは特定のチェックを無効にします。

- `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 互換性をもつアプリケーション固有のマクロです。

2.5.3.41 WOLFSSL_QT OpenSSL 互換性をもつアプリケーション固有のマクロです。QT 用の DH Extra、OpenSSL all、OpenSSH、および static ephemeral を有効にします。

2.5.3.42 WOLFSSL_HAPROXY OpenSSL 互換性をもつアプリケーション固有のマクロです。

2.5.3.43 WOLFSSL_ASN_TEMPLATE デュアルアルゴリズム証明書機能です。デフォルトで新しい ASN テンプレート `asn.c` コードを使用するため、ASN.1 テンプレート機能が必要です。

2.5.3.44 WOLFSSL_ASYNC_IO 非同期クリーンアップで使用されます。

2.5.3.45 WOLFSSL_ATMEL atmel_get_random_number 関数を使用してランダムデータをシードする ASF フックを有効にします。

2.5.3.46 WOLFSSL_CMAC 追加の CMAC アルゴリズムを有効にします。

注意：WOLFSSL_AES_DIRECT が必要です。

2.5.3.47 WOLFSSL_ESPIDF_ERROR_PAUSE test.c でのみ使用され、テストエラー時にデバッグ目的で遅延を追加します。

2.5.3.48 TEST_IPV6 テストアプリケーションでの IPv6 のテストをオンにします。wolfSSL は IP ニュートラルですが、テストアプリケーションはデフォルトで IPv4 を使用しています。

2.5.3.49 TEST_NONBLOCK_CERTS 非ブロッキング OCSP レスポンスのテストにのみ使用されます。WOLFSSL_NONBLOCK_OCSP と OCSP_WANT_READ で有効になります。

2.5.3.50 TEST_OPENSSL_COEXIST 以下のビルドオプションを有効にする場合に使用します。

```
./configure --enable-opensslcoexist
```

2.5.3.51 TEST_PK_PRIVKEY PK コールバックのテストにのみ使用されます。wolfSSL/test.h では PK コールバックでロード・使用される実際の秘密鍵を渡すために、コンテキストを使用します。

2.5.3.52 TEST_BUFFER_SIZE サンプルのクライアント/サーバー-B オプションで使用される、TLS ベンチマークテストバッファサイズのオーバーライドを可能にします。

2.5.3.53 FORCE_BUFFER_TEST ファイルシステムを使用する代わりに、test_certs.h バッファの使用を強制します。wolfSSL/test.h の内部テストにのみ使用されます。

2.5.3.54 WOLFSSL_FORCE_MALLOC_FAIL_TEST ランダムな malloc の失敗を誘発するための内部テスト用に定義します。

2.5.3.55 WOLFSSL_POST_HANDSHAKE_AUTH TLS 拡張機能、ポストハンドシェイク認証に使用されます。

2.5.3.56 WOLFSSL_PSK_MULTI_ID_PER_CS TLS 1.3 PSK において、暗号スイートごとに複数の ID を処理します。

2.5.3.57 WOLFSSL_PUBLIC_ASN 内部で使用される ASN.1 API を公開します。これは、内部の asn.h API を使用して解析したいお客様に役立ちます。

2.5.3.58 WOLFSSL_QUIC QUIC プロトコルのサポートを有効にします。詳細についてはこちらを参照してください。

2.5.3.59 WOLFSSL_QUIC_MAX_RECORD_CAPACITY 最大 quic キャパシティを 1024*1024 – 1 MB として定義します。

2.5.3.60 WOLFSSL_RENESAS_FSPSM_TLS まだサポートしていない、TLS 関連機能です。

2.5.3.61 WOLFSSL_RENESAS_TSIP_TLS TSIP TLS リンク共通鍵暗号化方式のみを無効にするためのものです。

2.5.3.62 WOLFSSL_SM2 SM 暗号を使用するために定義します。

2.5.3.63 WOLFSSL_SM3 SM 暗号を使用するために定義します。

2.5.3.64 WOLFSSL_SM4 SM 暗号を使用するために定義します。

2.5.3.65 WOLFSSL_SM4_CBC SM4 CBC の SM 設定です。

2.5.3.66 WOLFSSL_SM4_CCM SM4 CCM の SM 設定です。

2.5.3.67 WOLFSSL_SM4_GCM SM4 GCM の SM 設定です。

2.5.3.68 WOLFSSL_SNIFFER_CHAIN_INPUT Chain Input オプションでは、スニファーが入力を生のパケットへのポインタではなく、`struct iovec` リストとして受け取ることができます。

2.5.3.69 XSLEEP_MS テストのみに使用されます。カスタム遅延を定義できます。

2.5.3.70 XSNPRINTF snprintf 関数をオーバーライドできます。

2.5.3.71 DEFAULT_TIMEOUT_SEC `HAVE_IO_TIMEOUT` と一緒に使用して、`wolfio.c` ソケットタイムアウトを秒単位で指定します。これは OCSP および CRL HTTP の内部ソケットコードで使用されます。

2.5.3.72 HAVE_IO_TIMEOUT 証明書の失効に関するものです。IO オプションは接続タイムアウトのサポートを有効にしますが、デフォルトではオフです。

2.5.3.73 HAVE_OCSP オンライン証明書ステータスプロトコル（OCSP）サポートを有効にします。

2.5.3.74 HAVE_CSHARP C # ラッパーに必要な構成オプションをオンにします。

2.5.3.75 HAVE_CURVE25519 Curve25519 アルゴリズムの使用をオンにします。

2.5.3.76 HAVE_ED25519 ED25519 アルゴリズムの使用をオンにします。

2.5.3.77 WOLFSSL_DH_CONST Diffie Hellman Operations を実行するときにフローティングポイント値の使用をオフにし、`XPOW()` および `XLOG()` のテーブルを使用します。外部数学ライブラリへの依存関係を削除します。

2.5.3.78 WOLFSSL_TRUST_PEER_CERT 信頼できるピア証明書の使用をオンにします。これにより、CA 証明書を使用するのではなく、ピア証明書に接続することができます。信頼できるピア証明書が Peer Cert チェーンよりも一致している場合にオンになっていると、ピアが検証されたと見なされます。CA 証明書を使用することが望ましいです。

2.5.3.79 WOLFSSL_STATIC_MEMORY 静的メモリバッファと機能の使用をオンにします。これにより、動的ではなく静的メモリを使用できます。

2.5.3.80 WOLFSSL_STATIC_MEMORY_LEAN 合わせて、WOLFSSL_STATIC_MEMORY の定義が必要です。65k 未満のメモリプールサイズを必要とする構造体に対してより小さい型サイズを使用し、IO バッファなどの使用可能な機能を制限してフットプリントサイズを削減します。

2.5.3.81 WOLFSSL_SESSION_EXPORT DTLS セッションのエクスポートとインポートの使用をオンにします。これにより、DTLS セッションの現在の状態をシリアル化および送受信することができます。

2.5.3.82 WOLFSSL_ARMASM ARMv8 ハードウェアアクセラレーションの使用をオンにします。

2.5.3.83 WC_RSA_NONBLOCK Fast Math RSA ノンブロッキングサポートをオンにして、RSA 操作をより小さな仕事の塊に分割して処理します。この機能は、wc_RsaSetNonBlock() を呼び出し、FP_WOULDBLOCK 戻りコードをチェックすることにより有効になります。

2.5.3.84 WC_RSA_BLINDING タイミング耐性を有効にするために使用されます。

2.5.3.85 WC_RSA_PSS RSA PSS パディングを有効にします。TLS 1.3 でサポートされる唯一の RSA パディングスキームは PSS です（仕様によるものです）。PSS パディングはランダムパディングを使用します。

2.5.3.86 WOLFSSL_RSA_VERIFY_ONLY RSA 用の小さなビルドをオンにします。WOLFSSL_RSA_PUBLIC_ONLY、WOLFSSL_RSA_VERIFY_INLINE、NO_SIG_WRAPPER、WOLFCRYPT_ONLYを定義してください。

2.5.3.87 WOLFSSL_RSA_PUBLIC_ONLY RSA の公開キーのみの小さなビルドをオンにします。WOLFCRYPT_ONLYを定義してください。

2.5.3.88 WOLFSSL_SHA3 SHA3 使用のビルドをオンにします。これは、SHA3-224、SHA3-256、SHA3-384、SHA3-512 の SHA3 Keccak のサポートです。さらに、WOLFSSL_SHA3_SMALL を使用してパフォーマンスとリソース使用のトレードオフを行うことができます。

2.5.3.89 USE_ECDSA_KEYSZ_HASH_ALGO エフェメラル ECDHE 鍵サイズまたは次に使用可能な次の最高値と一致するハッシュアルゴリズムを選択します。この回避策は、SHA512 でハッシュされた P-256 鍵などのシナリオを正しくサポートしていないいくつかのピアに関する問題を解決します。

2.5.3.90 WOLFSSL_ALT_CERT_CHAINS この定義によって CA 証明書が通信相手から提示されることを許可しますが、有効なチェーンの一部としては使用しません。デフォルトの wolfSSL の動作は、提示されたすべてのピア証明書の検証を要求することです。これにより、中間 CA 証明書を信頼できるものとして扱い、Root CA 証明書まで至る CA の署名エラーは無視されません。代替の証明書チェーンモードでは、ピア証明書が信頼できる CA に検証する必要があります。

2.5.3.91 WOLFSSL_SYS_CA_CERTS wolfSSL_CTX_load_system_CA_certs() が呼び出されたとき、wolfSSL 証明書マネージャーにそれらをロードするか、システム認証 API を呼び出すことで、wolfSSL が検証のためにシステムの CA 証明書を使用できるようにします。詳細は wolfSSL_CTX_load_system_CA_certs() を参照してください。このプリプロセッサマクロは、--enable-sys-ca-certs 構成オプションによって自動的に設定されます。

2.5.3.92 WOLFSSL_APPLE_NATIVE_CERT_VERIFICATION TLS ピア証明書を認証する際に Apple のネイティブトラスト API の使用を有効にします。WOLFSSL_SYS_CA_CERTSが定義されている必要があります。iOS や他の Apple デバイスで `configure` または `CMake` でビルドする場合、このマクロをユーザーが設定する必要はありませんが、macOS でネイティブ検証方法を使用したい場合は明示的に設定する必要があります。

2.5.3.93 WOLFSSL_CUSTOM_CURVES 標準以外の曲線を許可します。計算では、曲線 "a" 変数が含まれています。HAVE_ECC_SECPR2、HAVE_ECC_SECPR3、HAVE_ECC_BRAINPOOL および HAVE_ECC_KOBLITZ を使用して、追加の曲線タイプを有効にできます。

2.5.3.94 HAVE_COMP_KEY ECC 圧縮鍵サポートを有効にします。

2.5.3.95 WOLFSSL_EXTRA_ALERTS TLS 接続中に追加のアラートを送信できるようにします。この機能は、`--enable-opensslextra`を使用すると自動的に有効になります。

2.5.3.96 WOLFSSL_DEBUG_TLS TLS 接続中に追加のデバッグプリントアウトを有効にします。

2.5.3.97 HAVE_BLAKE2 Blake2S アルゴリズムのサポートを有効にします。

2.5.3.98 HAVE_FALLBACK_SCSV サーバー側での Signaling Cipher Suite Value(SCSV) サポートを有効にします。これはクライアントから送信される暗号スイート `0x56 0x00` を処理し、TLS バージョンのダウングレードを許可しないことを示します。

2.5.3.99 HAVE_AEAD TLS 1.3 に必要な AEAD を有効にします。

2.5.3.100 HAVE_AES_CBC AES CBC のオプションを有効にします。

2.5.3.101 HAVE_ALPN ALPN を有効にします。

2.5.3.102 HAVE_CAVIUM_OCTEON_SYNC Marvell Cavium/Octeon ハードウェアのブロッキング（同期）バージョンを有効にします。

2.5.3.103 HAVE_CERTIFICATE_STATUS_REQUEST 証明書ステータス要求機能としての証明書失効に使用されます。

2.5.3.104 HAVE_CERTIFICATE_STATUS_REQUEST_V2 証明書ステータス要求機能としての証明書失効に使用されます。

2.5.3.105 HAVE_IO_TIMEOUT 証明書失効に関するものです。IO オプションは接続タイムアウトのサポートを有効にしますが、デフォルトではオフです。

2.5.3.106 HAVE_CURL cURL とリンクする際、wolfSSL ライブラリのサブセットをビルドするために使用されます。

2.5.3.107 HAVE_CURVE448 Curve448 サポート用に定義します。追加のマクロ設定を変更できます。デフォルトでは共有シークレット、キーのエクスポート、およびインポートが有効になっています。

2.5.3.108 HAVE_DANE このオプションは HAVE_RPK（生の公開鍵）でのみサポートされ、将来追加される可能性がある場合のプレースホルダーです。

2.5.3.109 HAVE_DILITHIUM DILITHIUM 量子暗号化/署名アルゴリズムを含めるために有効にします。

2.5.3.110 HAVE_ED25519_KEY_IMPORT ED25519 の設定です。署名、検証、共有シークレット、インポート、およびエクスポートの詳細な制御のために Ed255519 および Curve25519 オプションを有効にします。

2.5.3.111 HAVE_EX_DATA CTX/WOLFSSL のユーザー情報用の「追加」EX データ API を有効にします。

2.5.3.112 HAVE_EX_DATA_CLEANUP_HOOKS インデックスで RSA キーに対して追加データとクリーンアップコールバックを設定します。

2.5.3.113 HAVE_FALCON OpenQuantumSafe からの量子後暗号 FALCON を有効にします。

2.5.3.114 HAVE_FIPS さまざまな FIPS バージョンを実装する際に使用されます。

2.5.3.115 HAVE_KEYING_MATERIAL RFC 8446 Section 7.5に基づいてキーイング材料のエクスポートを有効にします。

2.5.3.116 HAVE_OID_DECODING ASN テンプレートコードに含まれています。一部のケースでデコードに使用されます。

2.5.3.117 HAVE_MAX_FRAGMENT 最大フラグメントサイズを設定します。TLS 拡張機能です。

2.5.3.118 WOLFSSL_PSK_ONE_ID TLS 1.3 を持つ PSK ID を 1 つだけサポートできます。

2.5.3.119 SHA256_MANY_REGISTERS すべてのデータをレジスタに保持し、部分的にループを展開する SHA256 の処理を指定します。

2.5.3.120 WOLFCRYPT_HAVE_SRP wolfCrypt セキュアリモートパスワードサポートを有効にします。

2.5.3.121 WOLFSSL_MAX_STRENGTH 最強のセキュリティ機能のみを有効にし、弱いまたは廃止予定の機能を無効にします。タイミングベースのサイドチャネル攻撃から保護するためのコンスタント実行により性能が劣化します。

2.5.3.122 MAX_RECORD_SIZE 最大レコードサイズを決定します。標準では 2^14 が最大サイズです。

2.5.3.123 MAX_CERTIFICATE_SZ 証明書メッセージペイロードの最大サイズを定義します。証明書あたり 2KB、MAX_CHAIN_DEPTH 個の証明書を想定しています。

2.5.3.124 MAX_CHAIN_DEPTH 最大チェーン深度を定義します。

2.5.3.125 MAX_CIPHER_NAME 最大暗号名を定義します。

2.5.3.126 MAX_DATE_SIZE バイト lastdate またはバイト nextdate として使用される日付の最大サイズを定義します。

2.5.3.127 MAX_EARLY_DATA_SZ 最大早期データサイズを定義するために使用されます。

2.5.3.128 WOLFSSL_MAX_SEND_SZ 最大送信サイズを指定するために定義します。

2.5.3.129 WOLFSSL_MAX_SUITE_SZ 最大スイートサイズを指定するために定義します。小さすぎるとエラーが発生します。

2.5.3.130 MAX_WOLFSSL_FILE_SIZE 4 MB の割り当てサイズ制限があります。

2.5.3.131 WOLFSSL_MAXQ10XX_TLS maxq10xx に使用している TLS バージョンを知らせます。

2.5.3.132 WOLFSSL_MAX_SIGALGO 最大署名アルゴリズムをオーバーライドする機能を有効にします。

2.5.3.133 WOLFSSL_MEM_GUARD 指定されたメモリガードを割り当てることができます。

2.5.3.134 WOLFSSL_STATIC_EPHEMERAL TLS スニファーサポートです

2.5.3.135 SSL_SNIFFER_EXPORTS WIN32 スニファーエクスポートです。

2.5.3.136 WOLFSSL_SNIFFER_KEYLOGFILE SSL キーログファイルオプションは、スニファーがNSS キーログファイルから取得したマスターシークレットを使用して TLS トラフィックを復号化できるようにします。これにより、スニファーは一時的な暗号スイートを使用する TLS 接続でも、すべての TLS トラフィックを復号化できます。キーログファイルスニッフィングは TLS バージョン 1.2 および 1.3 でサポートされています。

wolfSSL は、スニファー機能とは別に、`--enable-keylog-export` 構成オプションを使用してキーログファイルをエクスポートするように構成できます。（注意：これは本質的に安全ではないため、本番環境では絶対に行わないでください）

キーログファイルのスニファーサポートを有効にするには、以下の構成コマンドラインを使用してビルドします。

```
./configure --enable-sniffer CPPFLAGS=-DWOLFSSL_SNIFFER_KEYLOGFILE
```

2.5.3.137 WOLFSSL_SNIFFER_STORE_DATA_CB Store Data Callback オプションを使用すると、スニファーは再割り当てされたデータポインタではなくカスタムバッファにアプリケーションデータを保存する際に呼び出されるコールバックを取ることができます。コールバックはすべてのデータが消費されるまでループで呼び出されます。このオプションを有効にするには、以下の構成コマンドラインを使用してビルドします。

```
./configure --enable-sniffer CPPFLAGS=-DWOLFSSL_SNIFFER_STORE_DATA_CB
```

2.5.3.138 WOLFSSL_SNIFFER_WATCH Session Watching オプションを使用すると、スニファーは初期設定なしで提供された任意のパケットを監視できます。すべての TLS セッションのデコードを開始し、サーバーの証明書が検出されると、その証明書はユーザーが提供したコールバック関数に渡され、適切な秘密鍵を提供する必要があります。このオプションを有効にするには、以下の構成コマンドラインを使用してビルドします。

```
./configure --enable-sniffer CPPFLAGS=-DWOLFSSL_SNIFFER_WATCH
```

2.5.3.139 STATIC_BUFFER_LEN レコードヘッダーからメモリをフラグメント化しないでください。RECORD_HEADER_SZ に展開します。

2.5.3.140 STATIC_CHUNKS_ONLY ユーザーは、小さな静的バッファを使用している場合（デフォルト）に 16K 出力オプションをオフにするオプションがあり、SSL_write が持っているレコードよりも大きなデータを書き込もうとすると、ユーザーが静的バッファチャンクでのみ書き込むように指示しない限り、動的に取得します。

2.5.3.141 WOLFSSL_DEF_PSK_CIPHER ユーザー定義の PSK 暗号を有効にします。

2.5.3.142 WOLFSSL_OLD_PRIME_CHECK より高速な DH と RSA の素数チェックを使用する機能を有効にします。

2.5.3.143 WOLFSSL_STATIC_RSA 静的暗号スイートのみをサポートするレガシーシステム向けに残されています。静的な鍵を使用する暗号は強く非推奨とされており、避けられない場合以外は決して使用しないでください。WOLFSSL_STATIC_PSKおよびWOLFSSL_STATIC_DHも参照してください。

2.5.3.144 WOLFSSL_STATIC_PSK 静的な鍵を使用する暗号は非推奨とされています。WOLFSSL_STATIC_RSAを参照してください。

2.5.3.145 WOLFSSL_STATIC_DH 静的な鍵を使用する暗号は非推奨とされています。WOLFSSL_STATIC_RSAを参照してください。

2.5.3.146 HAVE_NULL_CIPHER NULL 暗号のサポートをオンにします。このオプションはセキュリティの観点から強く非推奨ですが、稀に暗号化/復号化操作を実行するには小さすぎるシステムがあります。そのようなシステムでも、このマクロを有効化して少なくともメッセージやピアを認証し、メッセージの改ざんを防ぐことだけができます。

2.5.3.147 HAVE_ANON 匿名の暗号スイートのサポートをオンにします。推奨されませんが、インターネットから切り離された、またはプライベートネットワークに関連するいくつかの有効なユースケースがあります。

2.5.3.148 HAVE_LIBOQS OpenQuantumSafe チームの LiboQS 統合のサポートをオンにします。詳細は付録 G ポスト量子暗号の実験を参照してください。

2.5.3.149 WOLFSSL_SP_4096 RSA/DH 4096 ビット単精度 (SP) サポートを有効にします。

2.5.3.150 WOLFSSL_SP_384 ECC SECP384R1 単精度 (SP) サポートを有効にします。WOLFSSL_SP_MATH にのみ適用されます。

2.5.3.151 WOLFSSL_SP_1024 Sakke ペアリングベースの単精度 (SP) サポートを有効にします。

2.5.3.152 ATOMIC_USER アトミックレコードレイヤーコールバックを有効にします。

2.5.3.153 BIG_ENDIAN_ORDER デフォルトはリトルエンディアンです。このマクロを定義することで、ビッグエンディアン環境で動作できます。

2.5.3.154 WOLFSSL_32BIT_MILLI_TIME 関数 `TimeNowInMilliseconds()` は符号なし 32 ビット値を返します。デフォルトの動作は符号付き 64 ビット値を返すことです。

2.5.3.155 WOLFSSL_MAX_DHKEY_BITS DH 最大ビットサイズは 8 の倍数でなければなりません。DH 最大ビットサイズは 16384 を超えたり、`WOLFSSL_MIN_DHKEY_BITS` より大きくすることはできません。

2.5.3.156 WOLFSSL_MIN_DHKEY_BITS DH 最小ビットサイズは 8 の倍数でなければなりません。112 ビットのセキュリティでは、DH は少なくとも 2048 ビットのキーが必要であり、最小ビットサイズは 16000 を超えてはなりません。

2.5.3.157 WOLFSSL_MAX_MTU 最大予想 MTU です。1500 - 100 バイトで、UDP と IP ヘッダーを考慮しています。

2.5.3.158 IGNORE_NETSCAPE_CERT_TYPE ネットスケープ証明書タイプを入力する場所を確保するために定義します。

2.5.3.159 SESSION_CERTS 証明書用の TLS セッションキャッシュです。

2.5.3.160 WOLFSSL_DUAL_ALG_CERTS デュアルアルゴリズム証明書の必須機能です。

2.5.3.161 CRL_MAX_REVOKED_CERTS RevokedCerts を保持するバッファの数を指定します。デフォルト値は 4 に設定されています。

2.5.3.162 CRL_STATIC_REVOKED_LIST バイナリ検索を可能にする RevokedCerts の固定静的リストを有効にします。

2.5.3.163 SESSION_INDEX キャッシュ内のセッションの場所を識別します。インデックスセッション/行シフトを指定します。

2.5.3.164 SESSION_TICKET_HINT_DEFAULT チケットヒントのデフォルトは、デフォルトのヒント値を設定するために使用されます。チケットキーの寿命はチケットのライフヒントよりも長くなければなりません。

2.5.3.165 WOLFSSL_DTLS13 wolfSSL DTLS 1.3 を有効にします。

2.5.3.166 WOLFSSL_TLS13 TLS 1.3 プロトコル実装を有効にします。

2.5.3.167 WOLFSSL_TLS13_IGNORE_AEAD_LIMITS RFC 9147 Section 4.5.3 に示された制限です。鍵更新が必要な制限を、ハードな復号化失敗制限の中間地点として指定しています。

2.5.3.168 WOLFSSL_TLS13_MIDDLEBOX_COMPAT TLS 1.3 ハンドシェイクでミドルボックス互換性を有効にします。これには、暗号化されたメッセージの前に ChangeCipherSpec を送信することとセッション ID を含めることが含まれます。

2.5.3.169 WOLFSSL_TLS13_SHA512 ハンドシェイクでの SHA-512 ダイジェストの生成を許可します。ただし、現時点ではどの暗号スイートも SHA-512 を必要としません。これにより、TLS v1.3 ではまだ使用されていませんが、ハンドシェイクメッセージの SHA2-512 ハッシュの計算が可能になります。

2.5.3.170 WOLFSSL_UIP CONTIKI が定義されている場合、これは UIP の実装です。

2.5.3.171 TLS13_MAX_TICKET_AGE 最大チケット期間を指定します。TLS 1.3 の場合、これは 7 日間です。

2.5.3.172 TLS13_TICKET_NONCE_STATIC_SZ TLS13_TICKET_NONCE_STATIC_SZ はこの FIPS_VERSION_GE ではサポートされていません。

2.5.3.173 TLS13_TICKET_NONCE_MAX_SZ チケットナンス用のバージョン最大サイズを定義します。最大サイズは 255 バイトとして定義されています。

2.5.3.174 WOLFSSL_TICKET_ENC_AES128_GCM デフォルトコールバックでセッションチケットの暗号化/復号化に AES128-GCM を使用します。これはサーバーのみで適用されます。ChaCha20/Poly1305 がコンパイルされていない場合、これがデフォルトのアルゴリズムです。

2.5.3.175 WOLFSSL_TICKET_ENC_AES256_GCM デフォルトコールバックでセッションチケットの暗号化/復号化に AES256-GCM を使用します。これはサーバーのみで適用されます。

2.5.3.176 WOLFSSL_TICKET_ENC_CHACHA20_POLY1305 デフォルトコールバックでセッションチケットの暗号化/復号化に ChaCha20-Poly1305 を使用します。何も定義されていない場合、デフォルトのアルゴリズムが使用され、アルゴリズムがコンパイルされます。これはサーバーのみで適用されます。

2.5.3.177 WOLFSSL_TICKET_EXTRA_PADDING_SZ チケットの追加パディングサイズを 32 として定義します。

2.5.3.178 WOLFSSL_TICKET_HAVE_ID チケットに ID があることを確認するために使用します。サポートが組み込まれていて、チケットに ID が含まれている場合にのみキャッシュに追加します。そうでなければ、キャッシュからチケットを取得する方法がありません。

2.5.3.179 WOLFSSL_TICKET_KEY_LIFETIME デフォルトの寿命は、キーを使用した最初のチケットの発行から 1 時間です。これはヒントよりも長くなければなりません。

2.5.3.180 WOLFSSL_TICKET_NONCE_MALLOC チケットナンスの動的割り当てを有効にします。HKDF 展開コールバックを無効にする必要があります。

2.5.3.181 SHOW_CERTS 定義されている場合、証明書を表示します。組み込みデバッグに使用します。

2.5.3.182 SHOW_SECRETS デバッグに使用されます。適用可能なシークレットを表示します。

2.5.3.183 DEBUG_UNIT_TEST_CERTS 名前制約テストをデバッグする際に使用されます。複雑な定義ガードを持つ複数の場所で使用できるように静的ではありません。

2.5.3.184 DEBUG_WOLFSSL_VERBOSE OPENSSL_EXTRA または DEBUG_WOLFSSL_VERBOSE マクロを使用する場合、WOLFSSL_ERROR は新しい関数 WOLFSSL_ERROR_LINE にマップされ、WOLFSSL_ERROR が呼び出される行番号と関数名を取得します。

2.5.3.185 SOCKET_INVALID 無効なソケットを定義するために使用され、-1 として定義されています。

2.5.3.186 WOLFSSL_SOCKET_INVALID テストに使用され、無効なソケットを示すために使用される値のオーバーライドのみを許可します。通常は-1 です。

2.5.3.187 WOLFSSL_SOCKET_IS_INVALID ソケット処理で使用されます。

2.5.3.188 WOLFSSL_SRTP SRTP をアクティブにするために使用されます。

2.5.3.189 WOLFSSL_CIPHER_CHECK_SZ 暗号化操作が機能したことを確認するために 64 ビットが必要な暗号チェックサイズとして定義されています。

2.5.3.190 DTLS_CID_MAX_SIZE DTLS 1.3 パーシングコードは、レコードを復号化するために静的バッファにレコードヘッダーをコピーします。CID 最大サイズを増やすと、このバッファも増加し、セッションごとの実行時メモリフットプリントに影響します。DTLS CID の最大サイズは 255 バイトです。

2.5.3.191 DTLS13_EPOCH_SIZE DTLS 1.3 エポックを使用した、移植性を高めるためのマクロです。DTLS エポックにバインドされた鍵を保存し、必要に応じて適切な鍵/エポックを設定する方法を実装します。

2.5.3.192 DTLS13_RETRANS_RN_SIZE DTLS 1.3 における移植性を高めるためのマクロです。DTLS 1.3 で再送信前のサイズを識別するために使用されます。

2.5.3.193 WOLFSSL_DTLS_FRAG_POOL_SZ 指定された時間あたりに許可されるフラグメントの数を定義します。

2.5.3.194 WOLFSSL_CLIENT_SESSION_DEFINED API が使用する不透明な構造体を宣言します。

2.5.3.195 WOLFSSL_COND このシステムが COND_TYPE シグナリングをサポートしている場合に定義されます。シグナリング API に渡されるべきタイプです。

2.5.3.196 WOLFSSL_DTLS_CH_FRAG サーバーがフラグメント化された 2 番目/検証済み（有効なクッキー応答を含む）ClientHello メッセージを処理できるようにします。1 番目/未検証（クッキー拡張を含まない）ClientHello はフラグメント化されていてはならず、DTLS サーバーがステートレスに処理できるようにする必要があります。これは DTLS 1.3 でのみ実装されています。ユーザーは、実行時にこれを明示的に有効にするために、サーバーで `wolfSSL_dtls13_allow_ch_frag()` を呼び出す必要があります。

注意：DTLS 1.3 + pqc を WOLFSSL_DTLS_CH_FRAG なしで使用すると、おそらく失敗します。この場合、`--enable-dtls-frag-ch` を使用して有効にします。

2.5.3.197 WOLFSSL_DTLS_MTU_ADDITIONAL_READ_BUFFER 私たちとは少し異なる MTU を持つピアと連携できるように、追加のバイトを読み取る必要がある場合に使用します。

2.5.3.198 WOLFSSL_DTLS_WINDOW_WORDS ウィンドウのストレージサイズをチェックするか、インデックスがウィンドウに対して有効かどうかを確認するために使用されます。

2.5.3.199 WOLFSSL_EXPORT_SPC_SZ CipherSpecs から使用されるバイト数を指定するために定義します。

2.5.3.200 WOLFSSL_MIN_DOWNGRADE 最小ダウングレードバージョンを指定します。

2.5.3.201 WOLFSSL_MIN_DTLS_DOWNGRADE 最小 DTLS ダウングレードバージョンを指定します。

2.5.3.202 WOLFSSL_MIN_ECC_BITS 許可される ECC キーサイズの最小値を設定できます。

2.5.3.203 WOLFSSL_MIN_RSA_BITS デフォルトでは、wolfSSL は RSA キーサイズを最小 1024 ビットに制限しています。512 ビットキーなどの小さく安全性の低い RSA キーのデコードを許可するには、コンパイラフラグ `-DWOLFSSL_MIN_RSA_BITS=512` を CFLAGS または CPPFLAGS に追加するか、ユーザー設定ヘッダーで定義する必要があります。

2.5.3.204 WOLFSSL_MODE_AUTO_RETRY_ATTEMPTS 無限リトライループの可能性を制限するために使用されます。

2.5.3.205 WOLFSSL_MULTICAST DTLS マルチキャスト機能です。

2.5.3.206 WOLFSSL_MULTICAST_PEERS 最大許容 100 ピアとして定義されたマルチキャスト機能です。

2.5.3.207 WOLFSSL_NAMES_STATIC Position Independent Code（PIC）のための静的 ECC 構造体を使用します。

2.5.3.208 WOLFSSL_SEND_HRR_COOKIE DTLS 1.3 で使用される TLS 拡張機能です。

2.5.3.209 WOLFSSL_SEP 機能証明書ポリシーセット拡張です。

2.5.3.210 WOLFSSL_SESSION_ID_CTX アプリケーションセッションコンテキスト ID をコピーするために使用されます。

2.5.3.211 WOLFSSL_SESSION_TIMEOUT 秒単位のデフォルトセッション再開キャッシュタイムアウトは、タイムアウトを手動で定義するために使用されます。

2.5.3.212 KEEP_OUR_CERT SSL 証明書を返す能力を確保するために使用されます。

2.5.3.213 KEEP_PEER_CERT ピア証明書を保持します。OpenSSL 互換性レイヤーの一部はピア証明書を必要とします。

2.5.3.214 WOLFSSL_SIGNER_DER_CERT これは署名に使用される DER/ASN.1 の保持を可能にします。これは `wolfSSL_X509_STORE_get1_certs` など、互換性レイヤーで使用されます。

2.5.3.215 CA_TABLE_SIZE wolfSSL 証明書マネージャー署名者テーブルで使用されます。デフォルトの CA_TABLE_SIZE は 11 ですが、実際のニーズに基づいて調整できます。各 WOLFSSL_CTX には独自の証明書マネージャー（CM）があります。

2.5.3.216 ECDHE_SIZE コンパイル時にこれをオーバーライドできるようにするために定義します。ECDHE サーバーサイズはデフォルトで 256 ビットであり、これは事前に決められた ECDHE 曲線サイズを設定できます。デフォルトは 32 バイトです。

2.5.3.217 CIPHER_NONCE 認証に実装される暗号化番号として使用されます。これは擬似乱数であり、整合性のみの暗号スイートです。

2.5.3.218 WOLFSSL_USE_POPEN_HOST wolfio.c ソケットオープンコードでホストとポストでソケットを作成するために popen を使用します。CRL と OCSP で使用されます。

2.5.3.219 CloseSocket ソケットを閉じるために使用される関数をオーバーライドする方法です。CRL、OCSP、BIO で使用されます。

2.5.3.220 CONFIG_POSIX_API ネットワーキングシステムコール用の POSIX 名を有効にします。

2.5.3.221 WOLFSSL_USER_CURRTIME マクロ WOLFSSL_USER_CURRTIME を使用して gettimeday なしで test.h で使用するオプションを追加します。

2.5.3.222 WOLFSSL_USER_MUTEX ユーザー定義ミューテックスのオプションです。

2.5.3.223 DEFAULT_MIN_ECCKEY_BITS ECCkey の最小ビット数を識別します。

2.5.3.224 DEFAULT_MIN_RSAKEY_BITS RSA キーの最小ビット数を識別します。

2.5.3.225 EXTERNAL_SERIAL_SIZE バッファに符号なしバイナリで X509 シリアル番号を書き込む生のシリアル番号バイトです。すべての場合において、バッファは少なくとも EXTERNAL_SERIAL_SIZE（32）である必要があります。成功した場合、WOLFSSL_SUCCESS を返します。

注意：これはユーザーが定義できない内部マクロです。

2.5.3.226 LARGE_STATIC_BUFFERS 組み込みコールバックには大きな静的バッファが必要です。16K までの大きなバッファを有効にするオプションを提供してください。

2.5.3.227 LIBWOLFSSL_VERSION_STRING これは wolfSSL バージョン文字列であり、リリースバンドルまたは./configure が実行されたときに入力されます。また、LIBWOLFSSL_VERSION_HEX にはこの 32 ビット HEX バージョンもあります。これらは wolfSSL/version.h から来ています。

2.5.4 wolfSSL のカスタマイズ及び移植

2.5.4.1 WOLFSSL_USER_SETTINGS 定義されている場合、ユーザー固有の設定ファイルを使用できます。ファイルには user_settings.h と名前が付けられ、インクルートパスに存在する必要があります。これは、標準の settings.h ファイルの前に含まれるため、デフォルト設定をオーバーライドできます。

2.5.4.2 WOLFSSL_CALLBACKS デバッガがない環境でシグナルを使用するデバック用のコールバックの利用を可能にする拡張機能です。デフォルトではオフです。ブロッキングソケットを使用してタイマーを設定するためにも使用できます。詳しくは第 6 章 コールバックをご覧ください。

2.5.4.3 WOLF_CRYPTO_CB 暗号コールバックサポートを有効にします。この機能は、`--enable-cryptocb`を使用すると自動的に有効になります。

2.5.4.4 WOLFSSL_DYN_CERT WOLFSSL_NO_MALLOC が設定されていても、証明書を解析するときに subjectCN および publicKey フィールドの割り当てを許可します。RSA 証明書で WOLFSSL_NO_MALLOC オプションを使用する場合、ピアの証明書で証明書を検証するために、CA の公開鍵を保持する必要があります。ca->publicKey が NULL であるため、これは ConfirmSignature エラー -173 BAD_FUNC_ARG として表示されます。

2.5.4.5 WOLFSSL_USER_IO ユーザーがデフォルトの I/O 関数`EmbedSend()`および`EmbedReceive()`の自動設定を削除できます。カスタム I/O 抽象化レイヤに使用されます (詳細については第 5 章「抽象化レイヤー」節を参照)。

2.5.4.6 NO_FILESYSTEM stdio(標準入出力関数) が使用できないために、証明書とキーファイルをロードできない場合に使用されます。これにより、ファイルの代わりにバッファを使用できます。

2.5.4.7 NO_INLINE 頻繁に使用されるライン数の少ない関数の自動インライン化を無効にします。この定義により、wolfSSL が遅くなり、実際にはこれらが小さな関数であるため、通常は関数呼び出し/ Return よりもはるかに小さくなります。autoconf を使用していない場合は、コンパイル済みファイルのリストに `wolfcrypt/src/misc.c` を追加する必要があります。

2.5.4.8 NO_DEV_RANDOM デフォルトの/dev/random 乱数ジェネレーターの使用を無効にします。定義されている場合、ユーザーは OS 固有の `GenerateSeed()` 関数 (`wolfcrypt/src/random.c` で見つかった) を記述する必要があります。

2.5.4.9 NO_MAIN_DRIVER 通常のビルド環境で使用されて、テストアプリケーションが独自に呼び出されるか、テストスイートドライバーアプリケーションを介して呼び出されます。`test.c`、`client.c`、`server.c`、`echoclient.c`、`echoserver.c`、および `testsuite.c` でテストファイルで使用する必要があります。

2.5.4.10 NO_WRITEV `writev()` セマンティクスのシミュレーションを無効にします。

2.5.4.11 SINGLE_THREADED ミューテックスの使用をオフにします。wolfSSL は現在、セッションキャッシュの保護にのみ使用しています。wolfSSL の使用が常に単一スレッドからに限定されている場合は、この機能をオンにすることができます。

2.5.4.12 USER_TICKS time(0) の使用が可能でない場合、ユーザーは自分のクロックチェック関数を定義できます。カスタム機能には秒単位の正確さが必要ですが、エポックと相関がある必要はありません。`wolfSSL_int.c` の機能を参照してください。

2.5.4.13 USER_TIME ユーザーが自身で定義した構造体を使用する (または必要とする) 場合の time.h 構造体の使用を無効にします。実装の詳細については `wolfcrypt/src/asn.c` を参照してください。ユーザーは `XTIME(),XGMTIME()`、および `XVALIDATE_DATE()` を定義および/または実装する必要があります。

2.5.4.14 USE_CERT_BUFFERS_256 <wolfSSL_root>/wolfSSL/certs_test.h にある 256 ビットのテスト用証明書と鍵バッファを有効にします。ファイルシステムのないエンベデッドシステムに移植するときあるいはテストに役立ちます。

2.5.4.15 USE_CERT_BUFFERS_1024 <wolfSSL_root>/wolfSSL/certs_test.h にある 1024 ビットのテスト用証明書と鍵バッファを有効にします。ファイルシステムのないエンベデッドシステムに移植するときあるいはテストに役立ちます。

2.5.4.16 USE_CERT_BUFFERS_2048 <wolfSSL_root>/wolfSSL/certs_test.h にある 2048 ビットテスト証明書と鍵バッファを有効にします。ファイルシステムのないエンベデッドシステムに移植するときあるいはテストに役立ちます。

2.5.4.17 USE_CERT_BUFFERS_3072 <wolfSSL_root>/wolfSSL/certs_test.h にある 3072 ビットのテスト用証明書と鍵バッファを有効にします。ファイルシステムのないエンベデッドシステムに移植するときあるいはテストに役立ちます。

2.5.4.18 USE_CERT_BUFFERS_4096 <wolfSSL_root>/wolfSSL/certs_test.h にある 4096 ビットのテスト用証明書と鍵バッファを有効にします。ファイルシステムのないエンベデッドシステムに移植するときあるいはテストに役立ちます。

2.5.4.19 USE_CERT_BUFFERS_25519 <wolfSSL_root>/wolfSSL/certs_test.h にある Ed25519 のテスト用証明書と鍵バッファを有効にします。ファイルシステムのないエンベデッドシステムに移植するときあるいはテストに役立ちます。

2.5.4.20 USE_WOLFSSL_IO このマクロは send/recv コールバックを有効にします。使用例はこちらで見ることができます。

2.5.4.21 CUSTOM_RAND_GENERATE_SEED ユーザーが wc_GenerateSeed(byte* output, word32 sz) に相当するカスタム機能を定義できるようにします。

2.5.4.22 CUSTOM_RAND_GENERATE_BLOCK ユーザーがカスタム乱数生成機能を定義できるようにします。使用例は以下の通りです。

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

```
/* 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 独自のカスタムハードウェアポートを作成していて、wc_AesGcmSetIV()の公開実装が提供されていない場合は、これを使用してください。

2.5.4.24 NO_PUBLIC_CCM_SET_NONCE 独自のカスタムハードウェアポートを作成していて、wc_AesGcmSetNonce() の公開実装が提供されていない場合は、これを使用してください。

2.5.4.25 NO_GCM_ENCRYPT_EXTRA 独自のカスタムハードウェアポートを行っていて、wc_AesGcmEncrypt_ex() の実装が提供されていない場合は、これを使用してください。

2.5.4.26 WOLFSSL_STM32[F1 | F2 | F4 | F7 | L4] 適切な STM32 デバイス用にビルドするときにこれらの定義のいずれかを使用します。必要に応じて、wolfSSL ポーティングガイドを確認し wolfSSL-root/wolfSSL/wolfcrypt/settings.h を更新します。

2.5.4.27 WOLFSSL_STM32_CUBEMX Cubemx ツールを使用してハードウェア抽象化レイヤー(HAL)API を生成する場合、この設定を使用して wolfSSL に適切なサポートを追加します。

2.5.4.28 WOLFSSL_CUBEMX_USE_LL Cubemx ツールを使用して API を生成する場合、HAL(ハードウェア抽象化層) または低層 (LL) の 2 つのオプションがあります。この定義を使用して、WOLFSSL_STM32[F1/F2/F4/F7/L4] の wolfSSL-root/wolfSSL/wolfcrypt/settings.h に含まれるヘッダーを制御します。

2.5.4.29 NO_STM32_CRYPTO STM32 のハードウェア暗号サポートを提供しない場合に定義します。

2.5.4.30 NO_STM32_HASH STM32 のハードウェアハッシュサポートを提供しない場合に定義します。

2.5.4.31 NO_STM32_RNG STM32 のハードウェア RNG サポートを提供しない場合のためにに定義します。

2.5.4.32 XTIME_MS TLS 1.3 を使用するときに使用するための関数をミリ秒単位でマッピングするためのマクロです。

使用例

```
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 TLS 接続中の AES 暗号化操作に対する可能なグリッチ攻撃をチェックするためにこれを定義します。

2.5.4.34 RTTHREAD RT-THREAD マクロは、rtthread IoT を wolfSSL にポーティングする際に使用されます。

2.5.4.35 SO_REUSEPORT ローカルアドレスとポートの再利用を許可します。

2.5.4.36 INTIME_RTOS INtime RTOS 用のポート設定です。

2.5.4.37 WOLFSSL_SGX SGX へのポーティング時に使用します。

2.5.5 メモリまたはコードの使用量の削減

2.5.5.1 TFM_TIMING_RESISTANT スタックサイズが小さいシステムで Fast Math(USE_FAST_MATH) を使用するときに定義できます。これにより大きな静的配列が削除されます。

2.5.5.2 ECC_TIMING_RESISTANT サイドチャネルと差分電力分析（DPA）攻撃を防ぐために ecc.c でコードを有効にするタイミング耐性機能として使用されます。

2.5.5.3 FUSION_RTOS Fusion RTOS 実装は、チケットが最初に見られた時と送信された時の違いを表すために使用されます。32 ビット値としてミリ秒単位の時間を返します。

2.5.5.4 WOLFSSL_SMALL_STACK スタックサイズが小さいデバイスに使用できます。これにより、wolfcrypt/src/integer.c の動的メモリの使用が増加しますが、パフォーマンスが遅くなる可能性があります。

2.5.5.5 ALT_ECC_SIZE Fast Math と RSA/DH を使用する場合は、ECC メモリ消費量を削減するためにこれを定義できます。ECC ポイントにスタックを使用する代わりに、ヒープから割り当てます。

2.5.5.6 ECC_SHAMIR ECC Math のバリエーションは、わずかに高速ですが、ヒープの使用量を 2 倍にします。

2.5.5.7 RSA_LOW_MEM 定義された場合、CRT は使用されていないため、一部のメモリを保存しますが、RSA 操作を遅くします。デフォルトではオフになっています。

2.5.5.8 WOLFSSL_SHA3_SMALL SHA3 を有効の場合、このマクロはビルドサイズを縮小します。

2.5.5.9 WOLFSSL_SMALL_CERT_VERIFY DecodedCert を使用せずに証明書署名を確認します。一部のコードでは 2 倍になりますが、ピークヒープメモリの使用が小さくなります。WOLFSSL_NONBLOCK_OCSPでは使用できません。

2.5.5.10 GCM_SMALL テーブルを使用する代わりに実行時に計算することで、AES GCM コードサイズを減らすオプションです。可能なオプションは、GCM_SMALL,GCM_WORD32,GCM_TABLE です。

2.5.5.11 CURVED25519_SMALL CURVE25519_SMALLおよびED25519_SMALLを定義します。

2.5.5.12 CURVE25519_SMALL CURVE25519 のスモールメモリオプションです。これはより少ないメモリを使用しますが、遅いです。

2.5.5.13 ED25519_SMALL ED25519 のスモールメモリオプションです これはより少ないメモリを使用しますが、遅いです。

2.5.5.14 USE_SLOW_SHA ローリングループを使用しないことでコードサイズを縮小します。これにより、SHA のパフォーマンスが低下します。

2.5.5.15 USE_SLOW_SHA256 ローリングループを使用しないことでコードサイズを縮小します。これにより、SHA のパフォーマンスが低下します。約 2k 小さくできますが、約 25%遅くなります。

2.5.5.16 USE_SLOW_SHA512 ローリングループを使用しないことでコードサイズを縮小します。これにより、SHA のパフォーマンスが低下します。2 倍以上小さくできますが、50%が遅くなります。

2.5.5.17 ECC_USER_CURVES ユーザーが有効になっている ECC カーブサイズを選択できるようにします。デフォルトでは 256 ビットカーブのみが有効になっています。他の曲線を使用できるようにするには、HAVE_ECC192,HAVE_ECC224 などを使用します。

2.5.5.18 WOLFSSL_SP_NO_MALLOC SP コードでは常に Stack を使用して、Heap xmalloc()/xrealloc()/xfree() 呼び出しは行われません。

2.5.5.19 WOLFSSL_SP_NO_DYN_STACK 動的スタックアイテムの使用を無効にします。コードサイズが小さく、小さなスタックではなく使用されます。

2.5.5.20 WOLFSSL_SP_FAST_MODEXP コードサイズを犠牲にして、より高速な mod_exp 実装でコンパイルします。

2.5.5.21 WC_DISABLE_RADIX_ZERO_PAD 16 進文字列出力で先頭ゼロの印刷を無効にします。

例えば、値 8 は通常「0x08」と表示されますが、このマクロが定義されている場合、「0x8」と表示されます。このマクロを定義すると、コードサイズを削減できます。

2.5.5.22 WC_ASN_NAME_MAX X.509 証明書フィールドの最大名前サポートのオーバーライドを許可します。

2.5.5.23 OPENSSL_EXTRA_X509_SMALL 証明書用の特別な小さな OpenSSL 互換レイヤーです。

2.5.6 パフォーマンスを向上させる

2.5.6.1 USE_INTEL_SPEEDUP AES、Chacha20、Poly1305、SHA256、SHA512、Ed25519 および Curve25519 の加速に Intel の AVX/AVX2 命令を使用できます。

2.5.6.2 WOLFSSL_AESNI Intel と AMD チップセットに組み込まれている AES アクセラレーション操作を使用できます。この定義を使用する場合、aes_asm.asm(AT & T 構文を備えた Windows 用) または aes_asm.S ファイルは、Intel AES の新しい命令セット (AESNI) を介して最適化するために使用されます。

2.5.6.3 HAVE_INTEL_RDSEED DRBG シードソース用の Intel の RDSEED を有効にします。

2.5.6.4 HAVE_INTEL_RDRAND wolfsSL のランダムソースの Intel の RDRAND 命令を有効にします。

2.5.6.5 FP_ECC ECC 固定点キャッシュを有効にします。これにより、同じ秘密鍵に対する繰り返し操作が高速化されます。FP_ENTRIES および FP_LUT を使用してエントリと LUT ビットの数を定義して、デフォルトの静的メモリ使用量を削減することもできます。

2.5.6.6 WOLFSSL_ASYNC_CRYPT これにより、Intel QuickAssist や Marvell(Cavium)Nitrox V などのハードウェアベースのアダプターを使用した非同期暗号化のサポートが可能になります。非同期コードは公開コードに含まれていません。評価のために必要でしたら、info@wolfssl.jp までお問い合わせください。

2.5.6.7 WOLFSSL_NO_ASYNC_IO これは非同期 I/O ネットワーキングを無効にします。非同期 I/O はデフォルトでオンになっており、ハンドシェイキングプロセス中に最大約 140 バイトを占めることがあります。ネットワークインターフェースが書き込み時に SOCKET_EWOULDBLOCK または SOCKET_EAGAIN（またはカスタム I/O コールバックの場合は WOLFSSL_CBIO_ERR_WANT_WRITE）を返さない場合、WOLFSSL_NO_ASYNC_IO を定義して、wolfSSL がハンドシェイクメッセージを構築している間に状態を保存しないようにできます。

2.5.7 GCM パフォーマンスチューニング

GCM パフォーマンスには 4 つのバリエーションがあります。

- GCM_SMALL - 最小のフットプリント、最も遅い (FIPS 検証済み)
- GCM_WORD32 - 中程度 (FIPS 検証済み)
- GCM_TABLE - 高速 (FIPS 検証済み)
- GCM_TABLE_4BIT - 最速 (FIPS 検証済み)

2.5.8 wolfSSL の数学ライブラリオプション

wolfSSL には 3 つの数学ライブラリがあります。

- Big Integer
- Fast Math
- Single Precision Math

wolfSSL をビルドするときは、これらの 1 つだけを使用する必要があります。

Big Integer Library は最も移植性の高いオプションで、アセンブリなしで C 言語で書かれています。そのため、特定のアーキテクチャに最適化されていません。すべての数学変数はヒープ上でインスタンス化されます。スタック使用量は最小限です。残念ながら、Big Integer Library はタイミング耐性がありません。

Fast Math Library は良いオプションです。C とアセンブリの両方を使用して実装されています。そのため、特定のアーキテクチャに最適化されています。すべての数学変数はスタック上でインスタンス化されます。ヒープ使用量は最小限です。TFM_TIMING_RESISTANT マクロが定義されている場合、タイミング耐性を持たせることができます。このライブラリは FIPS 140-2 および 140-3 認証を取得しています。

Single Precision (SP) Math Library は推奨ライブラリです。C とアセンブリの両方を使用して実装されています。そのため、特定のアーキテクチャに最適化されています。すべての数学変数はスタック上でインスタンス化されます。ヒープ使用量は最小限です。常にタイミング耐性があります。一般的にはコードサイズを犠牲にして速度が最適化されていますが、不要なコードをコンパイルしないように高度に構成可能です。このライブラリは DO-178C 認証を取得しています。

2.5.8.1 Big Integer ライブラリ (廃止予定)

このライブラリは 2023 年末までに wolfSSL/wolfCrypt ライブラリから廃止および削除される予定です。必要に応じて、`--enable-heapmath` または `CFLAGS=-DUSE_INTEGER_HEAP_MATH` で有効にすることができます。

パブリックドメインの LibTomMath ライブラリからフォークされました。LibTomMath の詳細については、https://www.libtom.net/LibTomMath/を参照してください。私たちのフォークは元のパブリックドメインコードよりもかなり活発でセキュアであることに注意してください。

これは一般的に最も移植性が高く、使い始めるのが最も簡単です。通常の big integer ライブラリのマイナス点は、より遅く、すべてのメモリがヒープから割り当てられるため多くのヒープメモリを使用し、`XREALLOC()` 実装を必要とし、タイミング耐性がないことです。実装は `integer.c` にあります。

2.5.8.2 Fast Math

2.5.8.2.1 USE_FAST_MATH

パブリックドメインの LibTomFastMath ライブラリからフォークされました。LibTomFastMath の詳細については、https://www.libtom.net/TomsFastMathを参照してください。私たちのフォークは元の LibTomFastMath のパブリックドメインコードよりもかなり活発でセキュアであることに注意してください。私たちはパフォーマンス、セキュリティ、コード品質を向上させました。また、FastMath コードは FIPS 140-2 および 140-3 認証を取得しています。

FastMath ライブラリは可能であればアセンブリを使用し、RSA、DH、DSA などの非対称秘密/公開鍵操作を高速化します。アセンブリの組み込みはコンパイラとプロセッサの組み合わせに依存します。一部の組み合わせでは追加の構成フラグが必要になり、一部は不可能な場合もあります。新しいアセンブリルーチンで FastMath を最適化するための支援はコンサルティングベースで利用可能です。アーキテクチャ固有の最適化を参照してください。

FastMath では、すべてのメモリはスタック上に割り当てられます。FastMath を使用する場合、スタックメモリの使用量が大きくなる可能性があるため、このオプションを使用する場合はTFM_TIMING_RESISTANTも定義することをお勧めします。TFM_TIMING_RESISTANT が定義されている場合、FastMath コードはタイミング耐性があります。これにより、一定時間のための大きな数学ウィンドウが減少し、メモリ使用量が少なくなります。秘密鍵操作中にショートカットが少なく、したがって分岐が少ないため、スタックの使用量

が少なくなります。これにより、タイミング攻撃が実際の脅威であり、悪意のある第三者に秘密鍵を複製するのに十分な情報を与える可能性があるため、実装がより安全になります。

例えば、ia32 では、すべてのレジスタが利用可能である必要があるため、高い最適化とフレームポインタの省略に注意する必要があります。wolfSSL はデバッグビルド以外では GCC に -O3 -fomit-frame-pointer を追加します。異なるコンパイラを使用している場合は、これらを構成中に手動で CFLAGS に追加する必要があるかもしれません。

macOS では、CFLAGS に -mdynamic-no-pic も追加する必要があります。さらに、OS X で ia32 の共有モードでビルドしている場合は、LDFLAGS にもオプションを渡す必要があります。

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

これにより、一部のシンボルに対してエラーではなく警告が表示されます。

FastMath は動的およびスタックメモリの使用方法も変更します。通常の数学ライブラリは大きな整数に動的メモリを使用します。FastMath はデフォルトで 4096 ビットの整数を保持する固定サイズのバッファを使用し、2048 ビットで 2048 ビットの乗算を可能にします。4096 ビットで 4096 ビットの乗算が必要な場合は、wolfSSL/wolfcrypt/tfm.h の FP_MAX_BITS を変更してください。FP_MAX_BITS が増加すると、公開鍵操作で使用されるバッファがより大きくなるため、実行時のスタック使用量も増加します。FP_MAX_BITS は最大キーサイズの 2 倍である必要があります。例えば、最大のキーが 2048 ビットの場合、FP_MAX_BITS は 4096 であるべきで、4096 ビットの場合は FP_MAX_BITS は 8192 であるべきです。ECC のみを使用する場合、これは最大 ECC キーサイズの 2 倍に減らすことができます。ライブラリのいくつかの関数は複数の一時的な大きな整数を使用するため、スタックは比較的大きくなる可能性があります。これは、スタックサイズが低い値に設定されている組み込みシステムやスレッド環境でのみ問題になるはずです。FastMath を使用している環境で公開鍵操作中にスタック破壊が発生した場合は、スタック使用量に対応するためにスタックサイズを増やしてください。

autoconf システムを使用せずに FastMath を有効にする場合は、USE_FAST_MATH を定義し、tfm.c を wolfSSL ビルドに追加し、integer.c を削除する必要があります。ALT_ECC_SIZE を定義すると、ECC ポイントはスタックではなくヒープからのみ割り当てられます。

2.5.8.2.2 アーキテクチャ固有の最適化 USE_FAST_MATH でアセンブリの最適化のために以下のマクロを定義できます。

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

これらのいずれも定義されていないか、TFM_NO_ASM が定義されている場合、TFM_ISO が定義され、ISO C ポータブルコードが使用されます。

2.5.8.2.3 アルゴリズム固有の最適化 有効にすると、それぞれの ECC 曲線に対して乗算と二乗計算の最適化された実装が使用されます。

- TFM_ECC192
- TFM_ECC224
- TFM_ECC256
- TFM_ECC384
- TFM_ECC521

2.5.8.2.4 TFM_SMALL_SET 小さい数の乗算のためのスピード最適化を行います。1-16 ワードのコンバ (Comba) 乗算と二乗の実装を含みます。ECC 操作の性能を向上させるのに役立ちます。

2.5.8.2.5 TFM_HUGE_SET より大きな数の乗算のための速度最適化を行います。20、24、28、32、48、64 ワードコンバ (Comba) 乗算と、ビットサイズが許す場所での二乗の実装が含まれています。RSA/DH/DSA 操作のパフォーマンスの向上に役立ちます。

2.5.8.2.6 TFM_SMALL_MONT_SET Intel アーキテクチャ上の小さな数値のモンゴメリーリダクションのための速度最適化を行います。1〜16 ワードモンゴメリーリダクションの実装が含まれています。ECC 操作の性能を向上させるのに役立ちます。

2.5.8.3 独自の単精度 (SP) 数学ライブラリのサポート SP 数学ライブラリは推奨されるデフォルトのオプションであり、DO-178C 認定を受けています。このライブラリを使用すると、特定の鍵サイズと一般的な曲線に対する公開鍵の操作が高速化されます。次のような正しいコードファイルが含まれていることをご確認ください。

- `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 単精度演算ライブラリのサポートを有効にします。

2.5.8.4 WOLFSSL_SP_MATH SP 数学とアルゴリズムのみを有効にします。通常 (`integer.c`) または FAST(`tfm.c`) などの大きな整数演算コードを排除します。鍵サイズと曲線を SP でサポートされているものだけに制限します。

2.5.8.5 WOLFSSL_SP_MATH_ALL SP 数学とアルゴリズムを有効にします。SP ではサポートされていない鍵サイズと曲線のために、通常 (`integer.c`) または FAST(`tfm.c`) などの大きな整数数学コードを実装しています。

2.5.8.6 WOLFSSL_SP_SMALL SP Math を使用する場合、これはコードの小さなバージョンを使用し、大きなスタック変数を回避します。

2.5.8.6.1 SP_WORD_SIZE 1 ワードを 1 ワードに保存するための 32 ビットまたは 64 ビットのデータ型。

2.5.8.6.2 WOLFSSL_SP_NONBLOCK 単一の精度数学の「非ブロッキング」モードを有効にします。これにより、長い操作のために FP_WouldBlock を返し、機能が完了するまで再度呼び出す必要があります。現在、これは ECC でのみサポートされており、`WC_ECC_NONBLOCK` と組み合わせて使用されています。

2.5.8.6.3 WOLFSSL_SP_FAST_NCT_EXPTMOD より速い非一定の時間モジュール式指数の実装を可能にします。公開鍵操作にのみ使用されます。秘密鍵操作ではありません。

2.5.8.6.4 WOLFSSL_SP_INT_NEGATIVE マルチプレシジョン数値を負にすることができます。(暗号化操作には必要ありません。)

2.5.8.6.5 WOLFSSL_SP_INT_DIGIT_ALIGN `sp_int_digit` ポインタの非整列アクセスが許可されていない場合に有効にします。

2.5.8.6.6 WOLFSSL_HAVE_SP_RSA 2048、3072、4096 ビットのための単精度 RSA。

2.5.8.6.7 WOLFSSL_HAVE_SP_DH 2048、3072、4096 ビットの単精度 DH。

2.5.8.6.8 WOLFSSL_HAVE_SP_ECC SECP256R1 および SECP384R1 用の単精度 ECC。

2.5.8.6.9 WOLFSSL_SP_LARGE_CODE 大きなバイナリーサイズとなりますが、単精度 (SP) スピードアップを許可します。一部の組み込みプラットフォームには適していない可能性があります。

2.5.8.6.10 WOLFSSL_SP_DIV_WORD_HALF 2倍の長さのワードを使用して除算が利用できないことを示します。たとえば、32 ビット CPU で、ライブラリから 64 ビット除算でコンパイルしたくない場合は、このマクロを定義すると、半分のワードサイズを使用して除算が行われる実装を有効化します。

2.5.8.6.11 WOLFSSL_SP_DIV_32 32 ビット除算が利用できず、wolfSSL が独自の単精度 (SP) 実装を使用する必要があることを示します。

2.5.8.6.12 WOLFSSL_SP_DIV_64 64 ビット除算が利用できず、WOLFSSL は独自の単精度 (SP) 実装を使用する必要があることを示します。

2.5.8.6.13 WOLFSSL_SP_ASM より高速な単一精度 (SP) プラットフォーム固有のアセンブリコードの実装を有効にします。プラットフォームが検出されます。

2.5.8.6.14 WOLFSSL_SP_X86_64_ASM 単精度 (SP) Intel x64 アセンブリの実装を有効にします。

2.5.8.6.15 WOLFSSL_SP_ARM32_ASM 単精度 (SP) Aarch32 アセンブリの実装を有効にします。

2.5.8.6.16 WOLFSSL_SP_ARM64_ASM 単精度 (SP) Aarch64 アセンブリの実装を有効にします。

2.5.8.6.17 WOLFSSL_SP_ARM_CORTEX_M_ASM 単精度 (SP) Cortex-M ファミリー (Cortex-M4 を含む) アセンブリの実装を有効にします。

2.5.8.6.18 WOLFSSL_SP_ARM_THUMB_ASM 単精度 (SP) ARM Thumb アセンブリの実装を有効にします (-mthumb と一緒に使用)。

2.5.8.6.19 WOLFSSL_SP_X86_64 単精度 (SP)I ntel X86 64 ビットアセンブリスピードアップマクロを有効にします。`WOLFSSL_SP_MATH_ALL` が定義されている場合にのみ適用されます。`sp_int.c` を参照してください。

2.5.8.6.20 WOLFSSL_SP_X86 単精度 (SP) Intel X86 アセンブリスピードアップマクロを有効にします。`WOLFSSL_SP_MATH_ALL` が定義されている場合にのみ適用されます。`sp_int.c` を参照してください。

2.5.8.6.21 WOLFSSL_SP_PPC64 単精度 (SP) PPC64 アセンブリスピードアップマクロを有効にします。WOLFSSL_SP_MATH_ALL が定義されている場合にのみ適用されます。sp_int.c を参照してください。

2.5.8.6.22 WOLFSSL_SP_PPC 単精度 (SP) PPC アセンブリスピードアップマクロを有効にします。WOLFSSL_SP_MATH_ALL が定義されている場合にのみ適用されます。sp_int.c を参照してください。

2.5.8.6.23 WOLFSSL_SP_MIPS64 単精度 (SP) MIPS64 アセンブリスピードアップマクロを有効にします。WOLFSSL_SP_MATH_ALL が定義されている場合にのみ適用されます。sp_int.c を参照してください。

2.5.8.6.24 WOLFSSL_SP_MIPS 単精度 (SP) MIPS アセンブリスピードアップマクロを有効にします。WOLFSSL_SP_MATH_ALL が定義されている場合にのみ適用されます。sp_int.c を参照してください。

2.5.8.6.25 WOLFSSL_SP_RISCV64 単精度 (SP) RISCV64 アセンブリスピードアップマクロを有効にします。WOLFSSL_SP_MATH_ALL が定義されている場合にのみ適用されます。sp_int.c を参照してください。

2.5.8.6.26 WOLFSSL_SP_RISCV32 単精度 (SP) RISCV32 アセンブリスピードアップマクロを有効にします。WOLFSSL_SP_MATH_ALL が定義されている場合にのみ適用されます。sp_int.c を参照してください。

2.5.8.6.27 WOLFSSL_SP_S390X 単精度 (SP) S390X アセンブリスピードアップマクロを有効にします。WOLFSSL_SP_MATH_ALL が定義されている場合にのみ適用されます。sp_int.c を参照してください。

2.5.9 スタックまたはチップ固有の定義

wolfSSL は、さまざまなプラットフォームと TCP/IP スタック用にビルドできます。次の定義のほとんどは wolfSSL/wolfcrypt/settings.h にあり、デフォルトでコメントアウトされています。以下を参照する特定のチップまたはスタックのサポートを有効にするために、それぞれのコメントを解除できます。

2.5.9.1 IPHONE iOS で使用するためにビルドする場合に定義します。

2.5.9.2 THREADX ThreadX RTOS(https://www.rtos.com) で使用するためにビルドするときに定義します。

2.5.9.3 MICRIUM Micrium の µC/OS-III RTOS(https://www.micrium.com) で使用するためにビルドするときに定義します。

2.5.9.4 MBED mbed プロトタイピングプラットフォーム (https://www.mbed.org) で使用するためにビルドするときに定義します。

2.5.9.5 MICROCHIP_PIC32 マイクロチップの PIC32 プラットフォーム (https://www.microchip.com) で使用するためにビルドするときに定義します。

2.5.9.6 MICROCHIP_TCPIP_V5 マイクロチップ TCP/IP スタックのバージョン 5 を定義できます。

2.5.9.7 MICROCHIP_TCPIP マイクロチップ TCP/IP スタックバージョン 6 以降に定義できます。

2.5.9.8 WOLFSSL_MICROCHIP_PIC32MZ PIC32MZ ハードウェア暗号化エンジン用に定義できます。

2.5.9.9 FREERTOS Freertos(https://www.freertos.org) で使用するためにビルドするときに定義できます。LwIP を使用している場合は、WOLFSSL_LWIPも定義します。

2.5.9.10 FREERTOS_WINSIM Freertos Windows Simulator(https://www.freertos.org) で使用するためにビルドするときに定義できます。

2.5.9.11 WOLFSSL_CHIBIOS ChibiOS RTOS で使用するためにビルドする場合に定義します。

2.5.9.12 WOLFSSL_CMSIS_RTOS Mbed CMIS-RTOS で使用するためにビルドする場合に定義します。

2.5.9.13 WOLFSSL_CMSIS_RTOSv2 Mbed CMIS-RTOSv2 で使用するためにビルドする場合に定義します。

2.5.9.14 WOLFSSL_LWIP_NATIVE LWIP ネイティブのプラットフォームで使用します。

2.5.9.15 WOLFSSL_DEOS この定義を使用して、ここで利用可能な Deos RTOS の wolfSSL サポートを有効にできます。

2.5.9.16 WOLFSSL_ESPIDF ESP-IDF でビルドする場合に使用します。

2.5.9.17 WOLFSSL_LINUXKM Linux カーネルモジュール用にビルドする場合に使用します。

2.5.9.18 WORD64_AVAILABLE 64 ビット型がサポートされていることを示す移植用マクロです。通常は SIZEOF_LONG_LONG 8 を使用する方が良いです。

2.5.9.19 WOLFSSL_NUCLEUS_1_2 Nucleus 1.2 でビルドする場合に使用します。

2.5.9.20 WOLFSSL_PICOTCP PicoTCP でビルドする場合に使用します。

2.5.9.21 WOLFSSL_RENESAS_RA6M3G RENESAS RA6M3G でビルドする場合に使用します。

2.5.9.22 WOLFSSL_RENESAS_RA6M4 RENESAS RA6M4 でビルドする場合に使用します。

2.5.9.23 WOLFSSL_RIOT_OS RIOT-OS でビルドする場合に使用します。

2.5.9.24 WOLFSSL_uITRON4 uITRON4 用にビルドする場合に使用します。

2.5.9.25 WOLFSSL_uTKERNEL2 uT-Kernel でビルドする場合に使用します。

2.5.9.26 WOLFSSL_VXWORKS VxWorks でビルドする場合に使用します。

2.5.9.27 DEVKITPRO devkitPro 用にビルドする場合に使用します。

2.5.9.28 WOLFSSL_VXWORKS_6_x VxWorks 6.x 専用の実装でのみ使用されます。

2.5.9.29 WOLFSSL_WICED WICED Studio 用にビルドする場合に使用されます。

2.5.9.30 FREESCALE_KSDK_FREERTOS このマクロの古い名前は FREESCALE_FREE_RTOS です。Freescale KSDK FreeRTOS 用にビルドする場合に使用されます。

2.5.9.31 FREESCALE_KSDK_MQX Freescale KSDK MQX/RTCS/MFS 用にビルドする場合に使用されます。

2.5.9.32 FREESCALE_MQX_5_0 Freescale Classic MQX バージョン 5.0 用にビルドする場合に使用されます。

2.5.9.33 WOLFSSL_KEIL_TCP_NET TCP スタック（MDK_CONF_NETWORK）を構成します。デフォルトでは Keil TCP WOLFSSL_KEIL_TCP_NET を使用します。なしの場合は 0、ユーザー IO コールバックの場合は 2 を使用します。

2.5.9.34 INTEL_GALILEO Arduino と wolfSSL を構成する際に使用されます。Intel Galileo プラットフォーム用にビルドする場合は #define INTEL_GALILEO を追加します。

2.5.9.35 HAVE_KEIL_RTX MDK-RTX-TCP-FS 構成用の wolfSSL。

2.5.9.36 EBSNET EBSNET 製品と RTIP を使用するときに定義できます。

2.5.9.37 WOLFSSL_EMBOS SEGGER embOS (https://www.segger.com/products/rtos/embos/) のビルド時に定義できます。emNET を使用する場合は、WOLFSSL_EMNET も定義します。

2.5.9.38 WOLFSSL_EMNET SEGGER emNET TCP/IP スタック (https://www.segger.com/products/connectivity/emnet/) のビルド時に定義できます。

2.5.9.39 WOLFSSL_LWIP LWIP TCP/IP スタック (https://savannah.nongnu.org/projects/lwip/) で wolfSSL を使用するときに定義できます。

2.5.9.40 WOLFSSL_ISOTP 通常、CAN バスに使用される ISO-TP トランスポートプロトコルで wolfSSL を使用する場合、定義できます。使用例はwolfssl-examples リポジトリにあります。

2.5.9.41 WOLFSSL_GAME_BUILD ゲームコンソールのために wolfSSL をビルドするときに定義できます。

2.5.9.42 WOLFSSL_LSR LSR 用にビルドする場合は定義できます。

2.5.9.43 FREESCALE_MQX Freescale MQX/RTCS/MFS(https://www.freescale.com) 用にビルドするときに定義できます。これにより、FREESCALE_K70_RNGA が定義され、Kinetis H/W 乱数ジェネレーターアクセラレータのサポートが可能になります。

2.5.9.44 WOLFSSL_STM32F2 STM32F2(https://www.st.com/internet/mcu/subclass/1520.jsp) 用のビルド時に定義できます。これは、wolfSSL で STM32F2 ハードウェア暗号化およびハードウェア RNG サポートを可能にします。

2.5.9.45 COMVERGE Comverge 設定を使用する場合は定義できます。

2.5.9.46 WOLFSSL_QL QL SEP 設定を使用している場合は定義できます。

2.5.9.47 WOLFSSL_EROAD eroad のためにビルドを定義することができます。

2.5.9.48 WOLFSSL_IAR_ARM IAR Ewarm 用にビルドする場合は定義できます。

2.5.9.49 WOLFSSL_TIRTOS TI-RTOS 用のビルド時に定義できます。

2.5.9.50 WOLFSSL_ROWLEY_ARM Rowley CrossWorks でビルドするときに定義できます。

2.5.9.51 WOLFSSL_NRF51 Nordic NRF51 に移植するときに定義できます。

2.5.9.52 WOLFSSL_NRF51_AES NORDIC NRF51 に移植するときに ENCRYPT を AES 128 ECB Encrypt に内蔵 AES ハードウェアを使用するように定義できます。

2.5.9.53 WOLFSSL_CONTIKI Contiki オペレーティングシステムのサポートを有効にするために定義できます。

2.5.9.54 WOLFSSL_APACHE_MYNEWT Apache MyNewt ポート層を有効にするように定義できます。

2.5.9.55 WOLFSSL_APACHE_HTTPD Apache HTTPD Web サーバーのサポートを有効にするために定義できます。

2.5.9.56 ASIO_USE_WOLFSSL wolfSSL を ASIO 互換バージョンとしてビルドするように定義できます。その後、ASIO は `BOOST_ASIO_USE_WOLFSSL` プリプロセッサ定義に依存します。

2.5.9.57 WOLFSSL_CRYPTOCELL ARM Cryptocell を使用できるように定義できます。

2.5.9.58 WOLFSSL_SIFIVE_RISC_V RISC-V SiFive/HiFive ポートの使用を有効にするために定義できます。

2.5.9.59 WOLFSSL_MDK_ARM MDK ARM のサポートを追加します。

2.5.9.60 WOLFSSL_MDK5 MDK5 ARM のサポートを追加します。

2.5.10 OS 特有の定義

2.5.10.1 USE_WINDOWS_API UNIX/Linux API に対して、Windows ライブラリ API の使用を指定します。

2.5.10.2 WIN32_LEAN_AND_MEAN Microsoft Win32 Lean と Mean Build のサポートを追加します。

2.5.10.3 FREERTOS_TCP FreeRTOS TCP スタックのサポートを追加します。

2.5.10.4 WOLFSSL_SAFERTOS SafeRTOS のサポートを追加します。

2.6 ビルドオプション

以下は、wolfSSL ライブラリのビルド方法をカスタマイズするために./configure スクリプトに追加される可能性のあるオプションです。デフォルトでは共有ライブラリとしてのみビルドされ、スタティックライブラリとしてのビルドが無効になっています。これによりビルド時間が半分に短縮されます。必要に応じて、どちらのモードも明示的に無効化または有効化できます。

2.6.1 --enable-debug

wolfSSL デバッグサポートを有効にし、デバッグ情報を含めてコンパイルします。デバッグログを出力するには、このビルドオプションを加えた上で メッセージを stderr に出力するマクロDEBUG_WOLFSSLを定義します。その後wolfSSL_Debugging_ON()を実行するとデバッグログ出力が有効化されます。同様にwolfSSL_Debugging_OFF()を実行することで、デバッグログ出力を停止できます。詳細は第 8 章 デバッグを参照してください。

2.6.2 --enable-distro

wolfSSL Distro Build を有効にします。

2.6.3 --enable-singlethread

シングルスレッドモードを有効にし、マルチスレッド保護を無効にします。

シングルスレッドモードを有効にすると、セッションキャッシュのマルチスレッド保護がオフになります。アプリケーションがシングルスレッドであること、またはアプリケーションがマルチスレッドであっても一度に 1 つのスレッドだけがライブラリにアクセスする場合にのみ、シングルスレッドモードを有効にしてください。

2.6.4 --enable-dtls

wolfSSL DTLS のサポートを有効にします。

DTLS サポートを有効にすると、ライブラリのユーザーは TLS および SSL に加えて DTLS プロトコルも使用できるようになります。詳細については、第 4 章「DTLS」節を参照してください。

2.6.5 --disable-rng

RNG のコンパイルと使用を無効にします。

2.6.6 --enable-sctp

wolfSSL DTLS-SCTP サポートを有効にします。

2.6.7 --enable-openssh

OpenSSH 互換ビルドを有効にします。

2.6.8 --enable-apachehttpd

Apache httpd 互換ビルドを有効にします。

2.6.9 --enable-openvpn

OpenVPN 互換ビルドを有効にします。

2.6.10 --enable-opensslextra

追加の OpenSSL API 互換性を有効にし、サイズを増加させます。

OpenSSL Extra を有効にすると、より多くの OpenSSL 互換関数が含まれます。これを指定しないベーシックビルドで、ほとんどの TLS/SSL ニーズ対応できる関数を使用できます。しかし 10〜100 の OpenSSL 関数を使用するアプリケーションを移植する場合には、これを有効にすることでよりスムーズに移植できるようになります。wolfSSL OpenSSL 互換レイヤーは現在も継続して開発しています。必要な関数が欠落している場合はお問い合わせください。OpenSSL 互換レイヤーの詳細については、第 13 章 OpenSSL 互換性を参照してください。

2.6.11 --enable-opensslall

wolfSSL がサポートする openSSL 互換性レイヤーの OpenSSL API をすべて有効にします。

2.6.12 --enable-maxstrength

最大強度ビルドを有効にし、TSLv1.2-AEAD-PFS 暗号のみを許可します。また、グリッチ検出も有効になります。相互運用性の問題が発生する可能性があるため、これはデフォルトで無効になっています。

2.6.13 --disable-harden

ハードニング、タイミング耐性と RSA ブラインドを無効にします。この機能を無効にするとパフォーマンスが向上する可能性があります。

注意 ハードニングはサイドチャネル攻撃に対する緩和策を提供します。慎重に 検討した後にのみこの機能を無効にしてください。

user_settings.h で無効にするには、以下のようにします。

- `#define WC_NO_CACHE_RESISTANT`
- `#define WC_NO_HARDEN`
- `WC_RSA_BLINDING` が定義されている場合、削除する
- `ECC_TIMING_RESISTANT` が定義されている場合、削除する
- `TFM_TIMING_RESISTANT` が定義されている場合、削除する

2.6.14 --enable-ipv6

IPv6 のテストを有効にします。wolfSSL 自体は IP 中立です

IPv6 を有効にすると、テストアプリケーションが IPv4 の代わりに IPv6 を使用するように変更されます。wolfSSL 自体は IP 中立で、どちらのバージョンも使用できますが、現在のテストアプリケーションは IP 依存です。

2.6.15 --enable-bump

SSL バンプビルドを有効にします。

2.6.16 --enable-leanpsk

LEAN PSK ビルドを有効にします。

PSK を使用し、ライブラリから多くの機能を削除して非常に小さなビルドを構成します。これを有効にすると、ビルドサイズは約 21kB になります。

2.6.17 --enable-leantls

LEAN TLS 1.2 クライアントのみ（クライアント認証なし）、ECC256、AES128 および SHA256（Shamir なし）をサポートするよう構成します。現時点では、他のビルドオプションと組み合わせて使用することを想定していません。

2.6.18 --enable-bigcache

ビッグセッションキャッシュを有効にします。

ビッグセッションキャッシュを有効にすると、セッションキャッシュが 33 セッションから 20,027 セッションまで増加します。デフォルトのセッションキャッシュサイズ 33 は、一般的な TLS クライアントと組み込みサーバーに適しています。ビッグセッションキャッシュは、1 分あたり約 200 の新しいセッションを処理するような、中程度の負荷がかかるサーバーに適しています。

2.6.19 --enable-hugecache

巨大なセッションキャッシュを有効にします。

巨大なセッションキャッシュを有効にすると、セッションキャッシュサイズが 65,791 セッションに増加します。このオプションは、1 分あたり 13,000(1 秒あたり 200) を越える新規セッションを処理するような、重い負荷のかかるサーバー向けです。

2.6.20 --enable-smallcache

小さなセッションキャッシュを有効にします。

小さなセッションキャッシュを有効にすると、wolfSSL は 6 セッションのみを保存し、RAM 使用量を 500 バイト未満に収めます。これは、デフォルトの RAM 使用量 3kB が大きすぎると感じる埋め込みデバイスに適しています。

2.6.21 --enable-savesession

永続的なセッションキャッシュを有効にします。

このオプションを有効にすることで、アプリケーションが wolfSSL セッションキャッシュをメモリバッファに保持 (保存) して復元できるようになります。

2.6.22 --enable-savecert

永続的な証明書キャッシュを有効にします。

このオプションを有効にすることで、アプリケーションが wolfSSL 証明書キャッシュをメモリバッファに保持 (保存) して復元できるようになります。

2.6.23 --enable-atomicuser

アトミックユーザーレコードレイヤを有効にします。

このオプションを有効にすると、ユーザーアトミックレコードレイヤの処理コールバックがオンになります。これにより、アプリケーションは独自の MAC /暗号化および復号化/検証コールバックを登録することができます。

2.6.24 --enable-pkcallbacks

公開鍵コールバックを有効にします。

2.6.25 --enable-sniffer

wolfSSL Sniffer のサポートを有効にします。

スニファー（SSL 検査）サポートを有効にすると、SSL トラフィックパケットの収集と、正しいキーファイルを使用したそれらのパケットの復号が可能になります。

現在、スニファーは以下の RSA 暗号をサポートしています。

CBC 暗号：

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

ストリーム暗号：

- RC4

2.6.26 --enable-aesgcm

AES-GCM サポートを有効にします。

このオプションを有効にすると公開鍵コールバックが有効になり、アプリケーションは独自の ECC 署名/検証と RSA 署名/検証を許可し、コールバックを識別して暗号化/復号できます。

2.6.27 --enable-aesccm

AES-CCM サポートを有効にします。

AES-CCM を有効にすると、以下の暗号スイートが wolfSSL に追加されます。wolfSSL は、速度とメモリ消費のバランスを取った 4 つの異なる AES-GCM 実装を提供しています。利用可能な場合、wolfSSL は 64 ビットまたは 32 ビットの計算を使用します。組み込みアプリケーションの場合、RAM ベースのルックアップテーブル（セッションあたり 8KB）を使用する高速な 8 ビットバージョン（64 ビットバージョンと同程度の速度）と、追加の RAM を消費しないより遅い 8 ビットバージョンがあります。–enable-aesgcm 設定オプションは、`=word32`、`=table`、または `=small` などのオプションで変更できます（例：`--enable-aesgcm=table`）。

2.6.28 --disable-aescbc

AES-CBC をコンパイルアウトするために`--disable-aescbc`で使用されていました。

AES-GCM は、AES の 8 バイト認証 (CCM-8) を備えた CBC-MAC モードでカウンターを有効にします。

2.6.29 --enable-aescfb

AES-CFB モードサポートをオンにします。

2.6.30 --enable-aesctr

wolfSSL AES-CTR サポートを有効にします。

AES-CTR を有効にすると、カウンタモードが有効になります。

2.6.31 --enable-aesni

wolfSSL Intel AES-NI サポートを有効にします。

AES-NI サポートを有効にすると、AES-NI をサポートするチップを使用する際にチップから直接 AES 命令を呼び出せるようになります。これにより、AES 関数の速度が向上します。AES-NI に関する詳細については、第 4 章 機能を参照してください。

2.6.32 --enable-intelasm

Intel および AMD プロセッサ向けの ASM 高速化を有効にします。

wolfSSL 用に intelasm オプションを有効にすると、プロセッサの拡張機能を活用して AES のパフォーマンスを劇的に向上させます。この設定オプションが有効な場合に活用される命令セットには、AVX1、AVX2、BMI2、RDRAND、RDSEED、AESNI、および ADX が含まれます。これらは Intel プロセッサに最初に導入され、AMD プロセッサも近年採用し始めています。有効にすると、wolfSSL はプロセッサをチェックし、プロセッサがサポートする命令セットを活用します。

2.6.33 --enable-camellia

Camellia サポートを有効にします。

2.6.34 --enable-md2

MD2 サポートを有効にします。

2.6.35 --enable-nullcipher

wolfSSL NULL 暗号サポート（暗号化なし）を有効にします。

2.6.36 --enable-ripemd

wolfSSL ripemd-160 サポートを有効にします。

2.6.37 --enable-blake2

wolfSSL Blake2 サポートを有効にします。

2.6.38 --enable-blake2s

wolfSSL Blake2S サポートを有効にします。

2.6.39 --enable-sha3

x86_64 および Aarch64 では、デフォルトで有効になっています。wolfSSL SHA3 のサポートを有効にします。小型ビルド用に、=small を設定できます。

2.6.40 --enable-sha512

x86_64 ではデフォルトで有効になっています。wolfSSL SHA-512 サポートを有効にします。

2.6.41 --enable-sessioncerts

セッション証明書保存を有効にします。

2.6.42 --enable-keygen

鍵生成機能を有効にします。RSA 鍵生成にのみ適用されます。

2.6.43 --enable-certgen

証明書生成機能を有効にします。

2.6.44 --enable-cert

証明書の拡張機能を有効にします。サポートしている拡張機能については、第 7 章を参照してください。

2.6.45 --enable-certreq

証明書リクエスト生成を有効にします。

2.6.46 --enable-sep

SEP 拡張機能を有効にします。

2.6.47 --enable-hkdf

HKDF（HMAC-KDF）を有効にします。

2.6.48 --enable-x963kdf

X9.63 KDF サポートを有効にします。

2.6.49 --enable-dsa

デジタル署名アルゴリズム (DSA) を有効にします。FIPS 186-4 で定義されている NIST 承認のデジタル署名アルゴリズムと RSA および ECDSA は、Secure Hash Standard (FIPS 180-4) で定義されている承認されたハッシュ関数と組み合わせて使用する場合、デジタル署名の生成と検証に使用されます。

2.6.50 --enable-eccshamir

x86_64 ではデフォルトで有効になっています。ECC Shamir を有効にします。

2.6.51 --enable-ecc

x86_64 ではデフォルトで有効になっています。ECC を有効にします。このオプションを有効にすると、ECC サポートと暗号スイートが wolfSSL に組み込まれます。

2.6.52 --enable-ecccustcurves

ECC カスタムカーブを有効にします。すべてのカーブタイプを有効にするには、=all を指定します。

2.6.53 --enable-compkey

圧縮鍵のサポートを有効にします。

2.6.54 --enable-curve25519

Curve25519 を有効にします。Curve25519_SMALL の場合は--enable-curve25519=small と指定します。

楕円曲線は 128 ビットのセキュリティを提供し、ECDH 鍵合意と共に使用されます (後の「クロスコンパイル」節を参照)。Curve25519 を有効にすると、Curve25519 アルゴリズムを使用できます。

デフォルトの Curve25519 は、より多くのメモリを使用し実行時間が速くなるように設計されています。オプション--enable-curve25519=small を使用することで、速度は低下しますがメモリの使用量を抑えることができます。

2.6.55 --enable-ed25519

Ed25519 を有効にします。Ed25519_SMALL の場合は--enable-ed25519=small と指定します。

Ed25519 オプションを有効にすると、Ed25519 アルゴリズムを使用できます。

デフォルトの Ed25519 は、より多くのメモリを使用し実行時間が速くなるように設計されています。オプション--enable-ed25519=small を使用することで、速度は低下しますがメモリの使用量を抑えることができます。

2.6.56 --enable-fpecc

固定小数点キャッシュ ECC を有効にします。

2.6.57 --enable-eccencrypt

ECC 暗号化を有効にします。

2.6.58 --enable-psk

PSK（事前共有鍵）を有効にします。

2.6.59 --disable-errorstrings

エラー文字列テーブルを無効にします。

2.6.60 --disable-oldtls

古い TLS バージョン（1.2 未満）を無効にします。

2.6.61 --enable-sslv3

SSL バージョン 3.0 を有効にします。

2.6.62 --enable-stacksize

サンプルプログラムでのスタックサイズ情報を有効にします。

2.6.63 --disable-memory

メモリコールバックを無効にします。

2.6.64 --disable-rsa

RSA を無効にします。

2.6.65 --enable-rsapss

RSA-PSS を有効にします。

2.6.66 --disable-dh

DH を無効にします。

2.6.67 --enable-anon

匿名認証を有効にします。

2.6.68 --disable-asn

ASN を無効にします。

2.6.69 --disable-aes

AES を無効にします。

2.6.70 --disable-coding

Base16/64 コーディングを無効にします。

2.6.71 --enable-base64encode

x86_64 ではデフォルトで有効になっています。Base64 エンコーディングを有効にします。

2.6.72 --disable-des3

DES3 を無効にします。

2.6.73 --enable-arc4

ARC4 を有効にします。

2.6.74 --disable-md5

MD5 を無効にします。

2.6.75 --disable-sha

Sha を無効にします。

2.6.76 --enable-webserver

Web サーバーを有効にします。

これにより、wolfSSL 組み込み Web サーバーで構築する際に完全な機能を提供するための、標準ビルドに必要な機能がオンになります。

2.6.77 --enable-fips

FIPS 140-2 を有効にします。

注意: 別途、ライセンス契約が必要です。

2.6.78 --enable-sha224

x86_64 ではデフォルトで有効になっています。wolfSSL SHA-224 サポートを有効にします。

2.6.79 --disable-poly1305

wolfSSL Poly1305 サポートを無効にします。

2.6.80 --disable-chacha

chacha を無効にします。

2.6.81 --disable-hashdrbg

ハッシュ DRBG サポートを無効にします。

2.6.82 --disable-filesystem

ファイルシステムのサポートを無効にします。

これにより、ファイルシステムの使用を無効にできます。このオプションはNO_FILESYSTEMを定義します。

2.6.83 --disable-inline

インライン関数を無効にします。

このオプションを無効にすると、wolfSSL の関数インライン化が無効になります。関数インライン化が有効な場合、関数プレースホルダはリンクされず、代わりに関数呼び出し時にコードブロックが挿入されます。

2.6.84 --enable-ocsp

オンライン証明書ステータスプロトコル (OCSP) を有効にします。

このオプションを有効にすると、OCSP(Online Certificate Status Protocol) サポートが wolfSSL に追加されます。RFC 6960に記載されているように、X.509 証明書の失効状態を取得するために使用されます。

2.6.85 --enable-ocspstapling

OCSP ステーリングを有効にします。TLS1.3 使用時、Certificate メッセージへの複数 OCSP Stapling を無効にするには =no-multi を指定します。

2.6.86 --enable-ocspstapling2

OCSP ステープリングバージョン 2 を有効にします。

2.6.87 --enable-crl

CRL(証明書失効リスト) を有効にします。

2.6.88 --enable-crl-monitor

CRL モニターを有効にします。

このオプションを有効にすると、特定の CRL(証明書の失効リスト) ディレクトリを積極的に監視する機能が追加されます。

2.6.89 --enable-sni

サーバー名表示を有効にします (SNI)。

このオプションを有効にすると、TLS サーバー名表示 (SNI) 拡張機能がオンになります。

2.6.90 --enable-maxfragment

最大フラグメント長を有効にします。

このオプションを有効にすると、TLS の最大フラグメント長拡張機能がオンになります。

2.6.91 --enable-alpn

アプリケーションレイヤープロトコルネゴシエーション (ALPN) を有効にします。

2.6.92 --enable-truncatedhmac

切り詰められたキー付きハッシュ MAC（HMAC）を有効にします。

このオプションを有効にすると、TLS 切り詰め HMAC 拡張機能がオンになります。

2.6.93 --enable-renegotiation-indication

再ネゴシエーションの表示を有効にします。

RFC 5746で説明されているように、この仕様は、再ネゴシエーションを実行する TLS 接続に結び付けることにより、再ネゴシエーションスプライシングを含む SSL/TLS 攻撃を防ぎます。

2.6.94 --enable-secure-renegotiation

安全な再ネゴシエーションを有効にします。

2.6.95 --enable-supportedcurves

サポートされている楕円曲線を有効にします。

このオプションを有効にすると、TLS サポート ECC 曲線拡張機能がオンになります。

2.6.96 --enable-session-ticket

セッションチケットを有効にします。

2.6.97 --enable-extended-master

拡張マスターシークレットを有効にします。

2.6.98 --enable-tlsx

すべての TLS 拡張機能を有効にします。

このオプションを有効にすると、現在 wolfSSL でサポートしているすべての TLS 拡張機能がオンになります。

2.6.99 --enable-pkcs7

PKCS # 7 サポートを有効にします。

2.6.100 --enable-pkcs11

PKCS # 11 アクセスを有効にします。

2.6.101 --enable-ssh

wolfSSH オプションを有効にします。

2.6.102 --enable-scep

wolfSCEP（Simple Certificate Enrollment Protocol）を有効にします。

IETF によって定義されているように、SCEP は HTTP 上で PKCS#7 と PKCS#10 を活用する PKI です。CERT は SCEP が証明書要求を強力に認証しないことに注意しています。

2.6.103 --enable-srp

安全なリモートパスワードを有効にします。

2.6.104 --enable-smallstack

スタックの使用量を抑えます。

2.6.105 --enable-valgrind

ユニットテストで Valgrind を有効にします。

このオプションを有効にすると、wolfSSL ユニットテストを実行するときに Valgrind がオンになります。これは、開発サイクルの早い段階で問題を発見するのに役立ちます。

2.6.106 --enable-testcert

テスト証明書を有効にします。

このオプションが有効になると、通常は公開されない ASN 証明書 API の一部が公開されます。これは wolfCrypt テストアプリケーション（wolfcrypt/test/test.c）で使用されているように、テストに役立ちます。

2.6.107 --enable-iopool

I/O プールの例を有効にします。

2.6.108 --enable-certservice

証明書サービスを有効にします。(Windows Server)

2.6.109 --enable-jni

wolfSSL JNI を有効にします。

2.6.110 --enable-lighty

Lighttpd/Lighty を有効にします。

2.6.111 --enable-stunnel

stunnel を有効にします。

2.6.112 --enable-md4

MD4 を有効にします。

2.6.113 --enable-pwdbased

pwdbased を有効にします。

2.6.114 --enable-scrypt

scrypt を有効にします。

2.6.115 --enable-cryptonly

wolfCrypt のみを有効にします。

2.6.116 --disable-examples

サンプルプログラムのビルドを無効にします。

サンプルプログラムのビルドを有効にすると、wolfSSL のサンプルアプリケーションがビルドされます。詳細は、第 3 章 入門をご参照ください。

2.6.117 --disable-crypttests

暗号ベンチマーク/テストを無効にします。

2.6.118 --enable-fast-rsa

Intel IPP を使用した RSA を有効にします。

fast-rsa を有効にすると、IPP ライブラリを使用して RSA 操作が高速化されます。通常よりも多くのメモリを使用するようになります。IPP ライブラリが見つからない場合、設定中にエラーメッセージが表示されます。autoconf ははじめに wolfssl_root/IPP ディレクトリを探索し、次は Linux システムの/usr/lib/などのマシン上のライブラリの標準的な場所を確認します。

RSA 操作に使用されるライブラリは、wolfssl-X.X.X/IPP/ディレクトリにあります。（X.X.X は現在の wolfSSL バージョン番号）バンドルされたライブラリからのビルドはディレクトリの場所と IPP の名前に依存しているため、サブディレクトリ IPP のファイル構造は変更しないでください。

メモリを割り当てる際、fast-rsa 操作のメモリタグは DYNAMIC_TYPE_USER_CRYPTO です。これにより、fast-rsa オプションを有効化した場合の、RSA 操作によるメモリ使用状況を確認できます。

2.6.119 --enable-staticmemory

静的メモリ使用を有効にします。

2.6.120 --enable-mcapi

マイクロチップ API を有効にします。

2.6.121 --enable-asynccrypt

非同期暗号を有効にします。

2.6.122 --enable-sessionexport

セッションのエクスポートとインポートを有効にします。

2.6.123 --enable-aeskeywrap

AES キーラップサポートを有効にします。

2.6.124 --enable-jobserver

値：yes(デフォルト)/ no/#

make を使用する場合、マルチスレッドビルドを使用して wolfSSL をビルドします。yes（デフォルト）は CPU コアの数を検出し、そのカウントに対して推奨されるジョブ数を使用してビルドします。これは make -j オプションと同様に機能します。手動で固定値を指定したい場合には、その値を使用します。

2.6.125 --enable-shared[=PKGS]

wolfSSL ライブラリを共有ライブラリとしてビルドします。デフォルトで有効です。

共有ライブラリビルドを無効にすると、wolfSSL 共有ライブラリがビルドされるのを除外します。デフォルトでは、時間とスペースを節約するために共有ライブラリのみがビルドされています。

2.6.126 --enable-static[=PKGS]

wolfSSL ライブラリをスタティックライブラリとしてビルドします。デフォルトでは無効です。使用する場合はこのオプションを指定してください。

2.6.127 --with-liboqs=PATH

OpenQuantumsafe インストールへのパスです。デフォルトでは/usr/local を参照します。

これにより、wolfSSL が実験的な TLS 1.3 Quantum-Safe KEM グループ、ハイブリッド量子セーフ KEM グループ、Liboqs との wolfSSL 統合を介して Falcon Signature Scheme を使用する機能がオンになります。詳細については、このドキュメントの「Quantum-Safe Cryptography の実験」を参照してください。

2.6.128 --with-libz=PATH

オプションで圧縮用に libz を含めます。

libz を有効にすると、libz ライブラリからの圧縮サポートが wolfSSL で可能になります。このオプションを含めて`wolfSSL_set_compression()`を呼び出すことについては、よくご検討ください。送信前にデータを圧縮すると、送受信されるメッセージの実際のサイズが減少します。しかし、圧縮されたデータはほとんどのネットワークにおいて、そのまま送信するよりも分析に時間がかかります。

2.6.129 --with-cavium

cavium/software ディレクトリへのパス。

2.6.130 --with-user-crypto

USER_CRYPTO インストールへのパス。デフォルトでは/usr/local を参照します。

2.6.131 --enable-rsavfy

RSA 検証のみのサポートを有効にします。

注意: `--enable-cryptonly`が必要です。

2.6.132 --enable-rsapub

デフォルト値：RSA 公開鍵のみのサポートを有効にします。

注意: `--enable-cryptonly`が必要です。

2.6.133 --enable-armasm

ARMV8 ASM サポートを有効にします。

デフォルトの構成は、64 か 32 ビットシステムに基づいて MCPU または MFPU を設定します。CPPFLAGS を使用して渡された MCPU または MFPU 設定を上書きしません。一部のコンパイラでは、constraiants のために-mstrict-align が必要になる場合があり、CPPFLAGS でユーザーが MCPU/MFPU フラグを渡さない限り、-mstrict-align もデフォルトで設定されます。

2.6.134 --disable-tlsv12

TLS 1.2 のサポートを無効にします。

2.6.135 --enable-tls13

TLS 1.3 サポートを有効にします。

このビルドオプションを`--disable-tlsv12`および`--disable-oldtls`と組み合わせて、TLS 1.3 のみの wolfSSL ビルドを生成できます。

2.6.136 --enable-all

SSL v3 を除くすべての wolfSSL 機能を有効にします。

2.6.137 --enable-xts

AES-XTS モードを有効にします。

2.6.138 --enable-asio

ASIO を有効にします。

一緒に--enable-opensslextraと--enable-openssl allを有効にする必要があります。これら 2 つのオプションが有効になっていない場合、autoconf ツールは自動的にこれらのオプションを有効にします。

2.6.139 --enable-qt

Qt 5.12 以降のサポートを有効にします。

wolfSSL QT ポートと互換性のある wolfSSL ビルド設定を有効にします。QT ソースファイルにパッチを当てるには、wolfSSL が提供するパッチファイルが必要です。

2.6.140 --enable-qt-test

Qt テスト互換ビルドを有効にします。

組み込みの QT テストの実行との互換性のために、wolfSSL をビルドするためのサポートを有効にします。

2.6.141 --enable-apache-httpd

Apache httpd サポートを有効にします。

2.6.142 --enable-afalg

ハードウェアアクセラレーション用の Linux モジュール AF_ALG の使用を有効にします。Xilinx での追加使用は =xilinx、=xilinx-rsa、=xilinx-aes、=xilinx-sha3 で可能です。

--enable-devcryptoと同様に、暗号操作のオフロードのために Linux カーネルモジュール（AF_ALG）を活用します。一部のハードウェアでは、Linux の暗号ドライバを通じてパフォーマンス加速が可能です。Petalinux と Xilinx の場合、フラグ--enable-afalg=xilinx を使用して、AF_ALG の Xilinx インターフェースを使用するように wolfSSL に指示できます。

2.6.143 --enable-devcrypto

ハードウェアアクセラレーション用の Linux /dev/crypto の使用を有効にします。

aes（すべての aes サポート）、hash（すべてのハッシュアルゴリズム）、および cbc（aes-cbc のみ）の任意の組み合わせを引数で指定できます。オプションが指定されていない場合、デフォルトで all を使用します。

2.6.144 --enable-mcast

wolfSSL DTLS マルチキャストサポートを有効にします。

2.6.145 --disable-pkcs12

PKCS12 コードを無効にします。

2.6.146 --enable-fallback-scsv

シグナリング暗号スイート値 (SCSV) を有効にします。

2.6.147 --enable-psk-one-id

TLS 1.3 を使用した単一の PSK ID のサポートを有効にします。

2.6.148 --enable-cryptocb

暗号コールバックを有効にします。wc_CryptoCb_RegisterDevice を使用して暗号コールバックを登録し、wolfSSL_CTX_SetDevid を使用して関連付けられている devid を設定します。`--enable-cryptocb` で次の 2 つの定義を使用して、RSA または ECC ソフトウェアフォールバックをコンパイルして、ソフトウェア RSA/ECC が不要な場合のフットプリント削減のために最適化できます。

- WOLF_CRYPTO_CB_ONLY_RSA - RSA ソフトウェア暗号フォールバックをコンパイルアウトします。
- WOLF_CRYPTO_CB_ONLY_ECC - ECC ソフトウェア暗号フォールバックをコンパイルアウトします。

WOLF_CRYPTO_CB_ONLY_* オプションを使用するには、サンプルプログラムのビルドを無効にする必要があります。`--disable-examples`を参照してください

2.6.149 --enable-reproducible-build

バイナリジッター (タイムスタンプやその他の非機能メタデータ) を抑制し、同一のハッシュを持つビット単位で同一のバイナリパッケージを生成できるようにします。

2.6.150 --enable-sys-ca-certs

wolfSSL が `wolfSSL_CTX_load_system_CA_certs()` が呼び出されたときに、検証のために信頼されたシステム CA 証明書を使用できるようにします。これは、wolfSSL 証明書マネージャーにそれらをロードするか、システム認証 API を呼び出すことによって行われます。詳細については、`wolfSSL_CTX_load_system_CA_certs()` を参照してください。

2.7 特別な数学最適化フラグ

2.7.1 --enable-fastmath

FastMath の実装を有効にします。単精度 (SP) 演算が有効な場合、FastMath と Big Integer ライブラリの両方が無効になります。

USE_FAST_MATH および Big Integer Math Library 節を参照してください。

2.7.2 --enable-fasthugemath

Fast Math + 巨大なコードを有効にします。

FastHugeMath の有効化には FastMath ライブラリのサポートが含まれます。公開鍵に関する実装に一般的な鍵サイズのループを展開することで、コードサイズが大幅に増加します。fasthugemath を使用する前後でベンチマークユーティリティを実行して、わずかな速度向上がコードサイズの増加に見合うかどうかをご確認ください。

2.7.3 --enable-sp-math

制限されたアルゴリズムスイートを使用した単精度 (SP) 数学実装を有効にします。サポートされていないアルゴリズムは無効になっています。--enable-sp、--enable-sp-math-all、--enable-fastmath、--enable-fasthugemath よりも優先されます。

- 整数ライブラリーの実装を sp_int.c の実装に置き換えます。
- 最小限の実装で、sp_int.c の一部を有効にしますが、すべてではありません。
- RSA/ECC/DH 操作を実行できるようにするには、--enable-sp と組み合わせて、sp_x86_64.c や sp_arm.c などのソリューション（ターゲットシステムに応じた以下のファイルリスト）をオンにする必要があります。
- --enable-sp-math-all (下記) と組み合わせてはいけません

ファイルリスト (プラットフォームに依存します。システムの仕様に基づいて構成によって選択されるか、Makefile/IDE ソリューションを使用する場合は手動で制御できます):

- 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

デフォルトで有効になっています。完全なアルゴリズムスイートを持つ単精度（SP）数学実装を有効にします。サポートされていないアルゴリズムが有効になっていますが、最適化されていません。--enable-sp、--enable-fastmath、--enable-fasthugemath よりも優先されます。

- 数学の実装を sp_int.c の実装に置き換えます。
- 完全な実装であり、--enable-sp の動作に依存しません
- --enable-sp と組み合わせて、可能であれば、32 ビットの sp_c32.c または 64 ビットの sp_c64.c で移植可能な C アセンブリ (ハードウェア固有でないアセンブリ) で記述された実装を使用できるようにすることができます。それ以外 (不可能な場合) には、sp_int.c の実装が使用されます。移植可能な C アセンブリは、ハードウェアの最適化が利用できないターゲットでパフォーマンスを大幅に向上させます。
- --enable-sp-math (上記) と組み合わせてはいけません

注: 鍵の長さがビット [256, 384, 521, 1024, 2048, 3072, 4096] の非対称暗号を使用している場合は、最大のパフォーマンスを得るために --enable-sp-math オプションの使用をご検討ください。ただしフットプリントサイズは大きくなります。

2.7.5 --enable-sp-asm

単精度 (SP) アセンブリの実装を有効にします。

Intel x86_64 および ARM アーキテクチャを使用したアセンブリを通して、単精度性能の向上を可能にするために使用できます。

2.7.6 --enable-sp=OPT

RSA、DH、および ECC の単一精度 (SP) 数学を有効にして、パフォーマンスを改善します。

+OPT には多くの値を設定できます。以下は、enable-sp を呼び出す方法と、結果として定義される結果のマクロのリストです。これらはすべて、カンマ区切りのリストで組み合わせることができます。例えば、

--enable-sp=ec256,ec384 のようにします。定義されるマクロの意味は、上記の「wolfSSL 独自の単精度 (SP) 数学サポート」節に記載しています。

注: 1) --enable-sp=small --enable-sp-math を使用することで、より小さくできる可能性があります。2) --enable-sp-math-all=small...

(1) は特定のキーサイズのみ実装しているのに対し、(2) はすべてのキーサイズをサポートする実装を持っているためです。

注：これは x86_64 用で、他の構成フラグはありません。結果は、指定するアーキテクチャやその他の構成フラグによって異なる場合があります。たとえば、WOLFSSL_SP_384 と WOLFSSL_SP_4096 は、Intel X86_64 に対してのみ有効になります。

2.7.6.1 --enable-sp=no または--disable-sp

新しいマクロは定義されません。--enable-sp を使用しないのと同等です。

2.7.6.2 --enable-sp または--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 または--enable-sp=p256 または--enable-sp=p256

- WOLFSSL_HAVE_SP_ECC

2.7.6.6 --enable-sp=smallec256 または--enable-sp=smallp256 または--enable-sp=small256

- WOLFSSL_HAVE_SP_ECC
- WOLFSSL_SP_SMALL

2.7.6.7 --enable-sp=ec384 または--enable-sp=p384 または--enable-sp=384

- WOLFSSL_HAVE_SP_ECC
- WOLFSSL_SP_384
- WOLFSSL_SP_NO_256

2.7.6.8 --enable-sp=smallec384 または--enable-sp=smallp384 または--enable-sp=small384

- WOLFSSL_HAVE_SP_ECC
- WOLFSSL_SP_384
- WOLFSSL_SP_NO_256
- WOLFSSL_SP_SMALL

2.7.6.9 --enable-sp=ec1024 または--enable-sp=p1024 または--enable-sp=1024

- WOLFSSL_HAVE_SP_ECC
- WOLFSSL_SP_1024
- WOLFSSL_SP_NO_256

2.7.6.10 --enable-sp=smallec1024 または--enable-sp=smallp1024 または--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 または --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 他のアルゴリズムオプションと組み合わせて、それらのオプションに対してアセンブリコードがオンになっていることを示します。例えば、`--enable-sp=rsa2048,asm` のように設定します。

2.8 クロスコンパイル

組み込み機器プラットフォーム上の多くのユーザーは、wolfSSL をクロスコンパイルしています。ライブラリをクロスコンパイルさせる最も簡単な方法は、`./configure` システムを使用することです。MakeFile を生成し、wolfSSL をビルドするために使用できます。

クロスコンパイルを行う場合、次のように`./configure` にホストを指定する必要があります。

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

また、使用するコンパイラ、リンカーなどを指定する必要がある場合があります。

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

malloc をオーバーライドしていることを検出し、`rpl_malloc` および/または `rpl_realloc` について configure システムがエラーを報告することがあります。

未定義参照が発生する場合は、`./configure` に以下を追加してください。

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

クロスコンパイルのために wolfSSL を正しく設定した後は、ライブラリのビルドとインストールのための標準的な autoconf の手順に従うことができるはずです。

```
make
sudo make install
```

wolfSSL のクロスコンパイルに関する追加のヒントやフィードバックがある場合は、info@wolfssl.jpまでお知らせください。

2.8.1 サンプルツールチェーンを使ったクロスコンパイルのコンフィグオプション例

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 移植向けビルド

wolfSSL は多くの環境やデバイスに移植（ポート）されています。これらのポートの一部とそれらのマニュアルはディレクトリ `wolfSSL-X.X.X/IDE` にあります。ここで、x.x.x は現在の wolfSSL バージョン番号です。このディレクトリには、環境用の wolfSSL をビルドするために使用されている IDE の有用な情報とコードも含まれています。

ポートリスト：

- Arduino

- LPCXPRESSO
- Wiced Studio
- CSBench
- SGX Windows と Linux
  - これらのディレクトリ (wolfSSL/IDE/WIN-SGX および wolfSSL/IDE/LINUX-SGX) には、Intel SGX プロジェクトで使用されるライブラリとして wolfSSL をビルドするための Makefiles および Visual Studio ソリューションが含まれています。
- Hexagon
  - このディレクトリ (wolfSSL/IDE/HEXAGON) には、Hexagon ツールチェーンを備えたビルド用の MakeFile が含まれています。ECC 検証操作を DSP プロセッサにオフロードするための wolfSSL をビルドするために使用できます。ディレクトリには、ビルドに必要な手順を支援する ReadMe ファイルが含まれています。
- Hexiwear
- NetBurner M68K
  - ディレクトリ (wolfSSL/IDE/M68K) には、NetBurner RTO を使用して MCF5441X デバイス用の wolfSSL をビルドするための MakeFile があります。
- Renesas
  - このディレクトリ (wolfSSL/IDE/Renesas) は、ルネサスデバイスの異なる複数のビルドを含みます。ハードウェアアクセラレーションを使用して実証するビルド例もあります。
- XCode
- Eclipse
- Espressif
- IAR-EWARM
- Kinetis Design Studio(KDS)
- Rowley Crossworks ARM
- OpenSTM32
- RISCV
- Zephyr
- mynewt
- INTIME-RTOS

2.10 NXP CAAM 向けのビルド

2.10.1 i.MX8(Linux)

2.10.1.1 既知の問題

- HSM セッション (wc_SECO_CloseHSM および wolfSSL_CLeanup または wolfCrypt_Cleanup) を閉じる前に、開いている HSM キーストアセッションを終了すると、次回 NVM を起動したときにセグメンテーションフォールトが発生します。これを回避するには、電源を入れ直す必要があります。

2.10.1.2 制限事項

- 多量の入力（1 MByte）を扱う SECO での AES 操作は” 共有メモリに十分なスペースがありません” とエラーになります。
- 2 つの鍵ストアを生成した後、3 つ目を生成しようとすると失敗します。鍵ストアをリセットするために `rm -rf /etc/seco_hsm` を実行し電源を再投入してください。

2.10.1.3 イントロ i.MX8 デバイスには、高度なセキュリティのための SECO ハードウェアモジュールが用意されています。このモジュールは AES 操作とキーストレージ、限定的な ECC 操作とキーストレージを処理し、RNG を提供します。wolfSSL は可能な限り SECO を利用するように拡張されています。

一部のアルゴリズムは i.MX8 の CAAM でサポートされていますが、SECO モジュールではまだサポートされていません。これらのケースでは、wolfSSL は/dev/crypto を通じて Linux CAAM ドライバーを呼び出し、CAAM に直接ジョブを作成します。

NXP Linux CAAM ドライバーではデフォルトでサポートされているアルゴリズムもありますが、CAAM がサポートするすべてのアルゴリズムがサポートされているわけではありません。wolfSSL は Linux CAAM ドライバーを拡張して、追加のアルゴリズムのサポートを追加しています。

同じアプリケーションから両方の経路で CAAM にアクセスするには、異なるコードパスに関連付けられた devId を WOLFSSL_CAAM_DEVID または WOLFSSL_SECO_DEVID のいずれかに設定できます。これらの ID は、構造体を最初に初期化する際に使用され、構造体の存続期間中に使用されるコードパスを設定します。ソフトウェアのみを使用する場合は、デフォルトの INVALID_DEVID を設定する必要があります。この例外は、キーストアを使用しない SECO 項目です：TRNG およびハッシュ。

使用されたソフトウェアのバージョンは以下のとおりです。

- imx-seco-libs ブランチ imx_5.4.24_2.1.0
- NXP「repo」ツールと Yocto ビルド
- wolfSSL 5.2.0 以降（5.2.0 リリース後に開発したためです）

2.10.1.4 サポートされるアルゴリズム サポートされるアルゴリズム、モード、オペレーションとして、以下を含みます。

- AES (ECB, CBC, CCM, GCM)
- AES CMAC
- SHA256, SHA384
- ECC 256/384（鍵生成、署名、検証、ecdh）
- RSA 3072（鍵生成、署名、検証）
- HMAC
- Curve25519
- TRNG

2.10.1.5 イメージのビルド

2.10.1.5.1 “repo” セットアップ 以下に示す NXP「repo」コマンドツールのセットアップは、Ubuntu 18.04 LTS で実施しました。

```
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"
```

2022 年 1 月 11 日現在、GitHub は認証されていない Git 接続を許可しなくなりました。このドキュメントが作成された時点 (2022 年 3 月) では、NXP のリポジトリツールはこれをまだ考慮していません。この問題を回避するには、次のコマンドを使用して git://github.com/ を https://github.com/ にリダイレクトしてください。

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

ビルド用のディレクトリを作成します。

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

NXP の「repo」コマンドツールを設定した後、目的の Linux バージョンでディレクトリを初期化して同期します。この場合は 5.4.24_2.1.0 です。

```
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 追加の Yocto CAAM レイヤー 次に、cryptodev-module、cryptodev-linux、および linux-imx/drivers/crypto/caam/* のファイルにパッチを適用する CAAM ドライバー拡張レイヤーをダウンロードします。ベース blob と ECDSA（署名/検証/鍵生成）は [こちら]https://source.codeaurora.org/external/imxsupport/imx_sec_apps/）にあります。RSA ブラックキー、ECDH、および Curve25519 サポートを持つ拡張レイヤーは meta-imx-expand-caam です。これらのディレクトリを他の既存の meta-* ディレクトリの横にある sources ディレクトリに配置します。

```
# 直前のコマンドの build-xwayland ディレクトリに居る想定

# クローンするかあるいは zip ファイルを解凍
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/

# あるいは

git clone https://source.codeaurora.org/external/imxsupport/imx_sec_apps.git
# "meta-imx-expand-caam" は wolfSSL が提供します
unzip meta-imx-expand-caam.zip

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

これらのレイヤーをビルドに追加します。最初に ecdsa を、次に CAAM 拡張を追加します。

```
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 "
```

このビルドではデバッグツールと SSH サーバーを追加しましたが、必ずしも必要ではありません。追加する重要な項目は cryptodev-module と cryptodev-linux です。eckey は、BLOB のカプセル化とカプセル化解除を行うための NXP のデモツールです。

オプション: cryptodev モジュールの自動ロードを追加するには、次の行を conf/local.conf に追加します。

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

それ以外の場合は、電源を入れ直すたびに modprobe cryptodev を使用してモジュールをロードする必要があります。

2.10.1.5.3 ビルドとデプロイ イメージのビルドの開始には次のコマンドを使います。そのあとで、もし SD カードを使っているならばカードに書き込みます。

```
bitbake core-image-base

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

注: 5m は macOS の場合に使用し、Linux の場合は 5M を使用します。diskX は SD カードの場所、つまり Mac であれば disc2、Linux であれば sdbX に置き換える必要があります。実行する前に、SD カードのディスク番号を確認してください。

後ほど wolfSSL/examples のビルドに使用するために、次のコマンドを使用してインストールディレクトリをエクスポートします。

```
export CRYPTODEV_DIR=`pwd`/tmp/sysroots-components/aarch64/cryptodev-
↪  linux/usr/include/
```

クロスコンパイルのためにツールチェーンをインストールするに、次の Yocto コマンドを使用します。

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

2.10.1.6 NXP HSM のビルド

2.10.1.6.1 zlib のビルド これを行うには複数の方法があり、その一つは Yocto ビルドに追加する方法です。bitbake を使用してビルドする方法は次のとおりです。

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

このコマンドは実行結果を tmp/sysroots-components/aarch64/zlib/usr/ディレクトリに配置します。

後ほど wolfSSL/examples のビルドに使用するために、次のコマンドを使用してインストールディレクトリをエクスポートします。

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

2.10.1.6.2 NXP HSM ライブラリのビルド NXP HSM ライブラリをダウンロードし、必要な zlib が見つかるように Makefile や環境変数を調整します。

```
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
```

さらに

```
make
make install
```

後ほど wolfSSL/examples のビルドに使用するために、次のコマンドを使用してインストールディレクトリをエクスポートします。

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

make install を行うとデフォルトで結果を export サブディレクトリに配置します。

2.10.1.7 wolfSSL のビルド

2.10.1.7.1 Autoconf を使用したビルド 開発ツールを使用して Yocto イメージをセットアップすると、wolfSSL をシステム上に直接構築できます。より最小限のアプローチとして、クロスコンパイルを使用できます。

デバッグメッセージは --enable-debug で有効にできます。SECO 作業に固有の追加のデバッグメッセージは、マクロ DEBUG_SECO を定義し、/dev/crypto 呼び出しの DEBUG_DEVCRYPTO を定義することで有効にできます。どちらの追加のデバッグメッセージも printf を使用し、stdout パイプに出力します。

SECO で使用するための重要な有効化オプションがいくつかあります。--enable-caam=seco、--enable-devcrypto=seco、--with-seco=/hsm-lib/export などです。

HSM SECO のみを使用したビルド例 (追加アルゴリズムの devcrypto サポートなし)

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

# wolfSSL のビルドで依存しているコンポーネントをインストール
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
```

HSM SECO と追加の devcrypto サポートを使ってのビルド例。インクルードパスとして crypto/cryptodev.h を指定する必要があります。

```
./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
```

wolfCrypt_Init / wolfSSL_Init 関数呼び出しで早期にエラーを出すフェイルセーフがあります。一つのケースは、cryptodev モジュールがロードされていないか、目的の操作に利用可能なサポートがない場合です。初期化操作が失敗する可能性がある別のケースは、NXP HSM をセットアップできなかった場合です。

アプリケーションが初期化に失敗している場合、wolfSSL ビルドに--enable-debug を追加し、wolfSSL の初期化前に wolfSSL_Debugging_ON() 関数呼び出しを行うと、失敗の理由に関する有用なデバッグメッセージが表示されます。

デバッグオプションを有効にしてビルドする例

```
./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 user_settings.h を使用したビルド autotools なしでビルドするために有効にできるマクロは次のとおりです。

CAAM

- WOLFSSL_CAAM - CAAM サポートを有効にするメインマクロスイッチ。
- WOLF_CRYPTO_CB - CAAM サポートは暗号コールバックを利用します。
- WOLFSSL_SECO_CAAM - CAAM での SECO HSM 使用を有効にします（AES-GCM が必要で、平文鍵を HSM にインポートするためのアルゴリズムとして使用されます）。
- WOLFSSL_HASH_KEEP - SHA256 などのアルゴリズムでハッシュする場合、メッセージを構築し、Final 呼び出し時にのみハッシュするために送信します。
- WOLFSSL_CAAM_ECC - CAAM ECC サポートを有効にします。
- WOLFSSL_CAAM_CMAC - CAAM CMAC サポートを有効にします。
- WOLFSSL_CAAM_CIPHER - CAAM AES サポートを有効にします。
- WOLFSSL_CAAM_HMAC - CAAM HMAC サポートを有効にします。
- WOLFSSL_CAAM_HASH - SHA256 などの CAAM ハッシングサポートを有効にします。
- WOLFSSL_CAAM_CURVE25519 - CAAM Curve25519 サポートを有効にします。

cryptodev-linux

- WOLFSSL_DEVCRYPTO - cryptodev-linux 使用を有効にするメインマクロスイッチ。
- WOLFSSL_DEVCRYPTO_HMAC - cryptodev-linux での HMAC サポートを有効にします。
- WOLFSSL_DEVCRYPTO_RSA - cryptodev-linux での RSA サポートを有効にします。
- WOLFSSL_DEVCRYPTO_CURVE25519 - cryptodev-linux での Curve25519 サポートを有効にします。
- WOLFSSL_DEVCRYPTO_ECDSA - cryptodev-linux での ECDSA サポートを有効にします。
- WOLFSSL_DEVCRYPTO_HASH_KEEP - cryptodev-linux でのハッシュ蓄積サポートを有効にします。

***CAAM* サポートのためにコンパイルする必要がある追加ファイル**

- 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 サンプルプログラム

2.10.1.8.1 Testwolfcrypt の実行 wolfSSL にバンドルされている単体テストは、wolfcrypt/test/test.c にあります。デバイス上でテストを構築して実行する例は次のとおりです。これは WOLFSSL_CAAM_DEVID を使用するため、NXP HSM ライブラリではなく cryptodev モジュールを使用していることにご注意ください。

```
./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:~#
```

そのほかのサンプルプログラムを、wolfSSL-examples リポジトリの caam/seco ディレクトリ配下に用意しています。

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

2.10.1.8.2 ソースコードをコンパイル [user_settings.h を使用] wolfSSL にリンクする単一のソースファイルを構築するには、次のコマンドを使用します。前のステップの環境変数がまだ設定されていると仮定します。

```
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 追加された API

- void wc_SECO_AesSetKeyID(Aes* aes, int keyId);
  - この関数は SECO キー ID を Aes 構造体にセットする際に使用します。
  - Aes 構造体が初期化された後で、暗号化/復号操作に使用される前に呼び出される必要があります。
- int wc_SECO_AesGetKeyID(Aes* aes);
  - Aes 構造体にセットされている SECO キー ID のゲッター関数です。
- void wc_SECO_CMACSetKeyID(Cmac* cmac, int keyId);
  - wc_SECO_AesSetKeyID 関数と似ていますが、対象が Cmac 構造体になっています。
- int wc_SECO_CMACGetKeyID(Cmac* cmac);
  - Cmac 構造体にセットされている SECO キー ID のゲッター関数です。
- int wc_SECO_OpenHSM(word32 keyStoreId, word32 nonce, word16 maxUpdates, byte flag);
  - この関数はキーストアが必要となるどの操作（ECC や AES）の前に呼び出される必要があります。
  - 最初の引数はキーストア ID、nonce は特定の 32 ビットの連番で、キーストアを新規に作成する場合や、既存のキーストアをアンロックするのに使用されます。
  - maxUpdates はキーストアが更新できる最大回数を指定します。
  - flag はオプショナルフラグで、キーストアが生成されているのかを示すために使用されます。
  - HSM キーストアを生成するためには flag は CAAM_KEYSTORE_CREATE でなければなりません。
  - CAAM_KEYSTORE_UPDATE は既存のキーストアをオープンし更新します。
- int wc_SECO_CloseHSM(void);
  - この関数は、キーストアの使用が完了したとき、wolfCrypt_Cleanup/wolfSSL_Cleanup を呼び出す前に呼び出す必要があります。
  - 現在オープンしているキーストアを閉じます。
- int wc_SECO_GenerateKey(int flags, int group, byte* out, int outSz, int keyType, int keyInfo, unsigned int* keyIdOut);

- この関数は、SECO で新しいキーを生成するために使用できます。
- キー生成の場合、flags は CAAM_GENERATE_KEY である必要があります。
- キーを更新する場合、flags は CAAM_UPDATE_KEY である必要があります。
- keyIdOut 引数は、キーの作成時に設定される入出力引数であり、キーの更新時に入力として設定する必要があります。
- キーを更新する場合は、一時的なタイプである必要があり、更新のために group を 0 に設定する必要があります。
- 一時的なタイプは keyInfo 引数として設定されます。
- keyInfo と keyType の可能なオプションは以下のとおりです。
  * 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);
  - キーストアから鍵を削除するのに使用されます。

2.10.1.9.2 CAAM をサポートするネイティブ wolfSSL API これは、このドキュメントで概説されている SECO ビルドで CAAM サポートを備えたネイティブ wolfSSL API のリストです。

AES 暗号化と復号化操作の生成には、次のプロセスで鍵を生成できます。
wc_SECO_GenerateKey(CAAM_GENERATE_KEY, groupID, pubOut, 0, CAAM_KEYTPE_AES128, CAAM_KEY_PERSISTENT, &keyIdOut) を使用します。
ここで groupID は指定されたグループ番号、pubOut は 32 バイトのバッファ、変数 keyIdOut は生成された新しい鍵 ID に設定されます。
この生成された新しい鍵 ID は、wc_SECO_AesSetKeyID(Aes, keyIdOut) を使用して Aes 構造体に設定できます。
鍵 ID が構造体に設定され、Aes 構造体が WOLFSSL_SECO_DEVID タイプとして初期化されると、すべての暗号化および復号化操作にその鍵 ID が使用されます。

AES (ECB/CBC)

AES ECB/CBC 鍵を生成する代わりに、Aes 構造体が WOLFSSL_SECO_DEVID で初期化されている場合は、プレーンテキストキーを渡して関数 wc_AesSetKey を呼び出すことができます。
API wc_AesSetKey は、一意の KEK を使用して鍵を暗号化し、SECO HSM にインポートしようとします。インポートが成功すると、値 0 が返され、鍵 ID が Aes 構造体に設定されます。

- CBC 暗号化は wc_AesCbcEncrypt で行われ、復号は wc_AesCbcDecrypt で行われます。
- ECB 暗号化は wc_AesEcbEncrypt で行われ、復号は wc_AesEcbDecrypt で行われます。

Aes 構造体の使用が完了したら、wc_AesFree(Aes) を使用して解放する必要があります。

AES-GCM

- GCM 暗号化は wc_AesGcmEncrypt で行われ、復号は wc_AesGcmDecrypt で行われます。

AES-GCM 暗号化関数は、Aes 構造体、出力バッファ、入力バッファ、入力バッファサイズ、ノンス、ノンスサイズ (12 バイト必要)、MAC またはタグとして知られる、タグサイズ (16 バイトが必要)、追加データ、

追加データサイズ（4 バイト）を引数として取ります。
暗号化の場合、入力バッファは暗号化され、タグバッファは作成された MAC で埋められます。

AES-GCM 復号の場合、関数は Aes 構造体、プレーンテキスト出力バッファ、暗号テキスト入力バッファ、入力バッファサイズ、ノンス、ノンスサイズ (12 バイト)、暗号化呼び出しで以前に作成されたタグ、タグバッファサイズ、追加データ、追加のデータサイズを引数として取ります。
復号化時にタグバッファがチェックされ、メッセージの整合性が検証されます。

Aes 構造体の使用が完了したら、wc_AesFree(Aes) を使用して解放する必要があります。

AES-CCM

- CCM 暗号化は wc_AesCcmEncrypt、復号は wc_AesCcmDecrypt で行われます。

AES-CCM 暗号化関数は、Aes 構造体、出力バッファ、入力バッファ、入力バッファサイズ、ノンス、ノンスサイズ (12 バイトが必要)、MAC またはタグとして知られる、タグサイズ (16 バイト必要)、追加データ、追加データサイズ（0 バイト）を引数として取ります。追加のデータバッファは NULL にする必要があり、NXP HSM ライブラリではサイズ 0 が必要です。暗号化の場合、入力バッファは暗号化され、タグバッファは作成された MAC で埋められます。

AES-CCM 復号の場合、関数は Aes 構造体、プレーンテキスト出力バッファ、暗号テキスト入力バッファ、入力バッファサイズ、ノンス、ノンスサイズ (12 バイト)、暗号化呼び出しで以前に作成されたタグ、タグバッファサイズ、追加データ、追加のデータサイズを引数に取ります。暗号化関数と同様に、追加のデータバッファーは NULL である必要があります。復号時にタグバッファがチェックされ、メッセージの整合性が検証されます。

Aes 構造体の使用が完了したら、wc_AesFree(Aes) を使用して解放する必要があります。

AES CMAC

AES CMAC 操作の場合、wc_SECO_GenerateKey(CAAM_GENERATE_KEY, groupID, pubOut, 0, CAAM_KEYTPE_AES128, CAAM_KEY_PERSISTENT, &keyIdOut); を使用して AES 鍵を生成できます。ここで、groupID は指定されたグループ番号、pubOut は 32 バイトのバッファーであり、変数 keyIdOut は生成された新しいキー ID に設定されます。生成されたこの新しいキー ID は、wc_SECO_CMACSetKeyID(Cmac, keyIdOut) を使用して Aes 構造体に設定できます。キー ID が構造内に設定され、Aes 構造体が WOLFSSL_SECO_DEVID タイプとして初期化されると、すべての暗号化および復号化操作にそのキー ID が使用されます。

SM ライブラリはシングル ショットタイプであるため、wc_CmacUpdate を呼び出すたびに入力が内部バッファに保存されます。その後、wc_CmacFinal が呼び出されると、MAC を作成するためにバッファ全体がハードウェアに渡されます。

RSA

RSA 操作では、cryptodev-linux モジュールが使用されます。これには、WOLFSSL_CAAM_DEVID で初期化されたときのデフォルトである AES-ECB 暗号化されたブラックプライベートキーのサポートが含まれます。

cryptodev-linux モジュールで使用されるネイティブ wolfSSL API の例は次のとおりです。

```
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 署名および検証操作では、cryptodev-linux モジュールまたは NXP HSM ライブラリのいずれかを使用できます。共有シークレットを作成するための ECDH 操作は、cryptodev-linux モジュールを使用してのみ実行できます。

SECO で使用する場合 (NXP HSM ライブラリを使用)、ecc_key 構造体を初期化するときに WOLFSSL_SECO_DEVID の dev ID フラグを使用する必要があります。cryptodev-linux モジュールで使用するには、dev ID フラグ WOLFSSL_CAAM_DEVID を使用する必要があります。関数 wc_ecc_init_ec(key, heap-hint (NULL 可), dev ID) による初期化後、どちらのユースケースも、署名と検証を行うためのネイティブの wolfSSL 関数呼び出しの同じ関数に従います。

ecc_key 構造体の初期化後の関数呼び出しの例は次のようになります。

```
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);
```

また、cryptodev-linux モジュール (WOLFSSL_CAAM_DEVID) を使用すると、ECDH 関数を使用できます。

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

ハッシュ (Sha256, Sha384, HMAC)

SHA256 および SHA384 の操作では、NXP HSM ライブラリを使用します。HMAC 操作では、cryptodev-linux モジュールが使用されます。

デフォルトでは、SHA 操作は NXP HSM ライブラリを利用しようとしますが、明示的に dev ID に設定すると、WOLFSSL_SECO_DEVID を使用できます。

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

NXP HSM ライブラリはハッシュのシングルショット操作をサポートしているため、“update” を呼び出すたびに “final” 関数が呼び出されるまでバッファが保存され、その後ハッシュダイジェストを作成するためにバッファ全体がハードウェアに渡されます。HMAC が cryptodev-linux を使用する場合、Hmac 構造体は dev ID WOLFSSL_CAAM_DEVID を使用して初期化する必要があります。

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

その後、通常のネイティブ wolfSSL API と同じように使用できます。

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

Curve25519

Curve25519 ポイント乗算は cryptodev-linux モジュールを使用して行われ、ハードウェアで使用するために dev ID WOLFSSL_CAAM_DEVID で初期化する必要があります。

API 呼び出しの例は次のようになります。

```
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

wolfSSL HASH-DRBG に乱数シードを与えるために使われる TRNG では、NXP HSM ライブラリを利用します。これは、wolfSSL が--enable-caam=seco でビルドされるときに、wolfcrypt/src/random.c ファイルにコンパイルされます。wolfSSL のすべての RNG 初期化では、シードに TRNG が使用されます。標準の RNG API 呼び出しは次のようになります。

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

2.10.2 i.MX8 (QNX)

(ドキュメントは現在準備中です。ご入用でしたら、info@wolfssl.jp までお問い合わせください。)

2.10.3 i.MX6 (QNX)

(ドキュメントは現在準備中です。ご入用でしたら、info@wolfssl.jp までお問い合わせください。)

2.10.4 IMXRT1170 (FreeRTOS)

IMXRT1170 で使用する IDE セットアップの例は、IDE/MCUEXPRESSO/RT1170 ディレクトリにあります。

2.10.4.1 ビルド手順

1. MCUEXPRESSO を開き、ワークスペースを wolfSSL/IDE/MCUEXPRESSO/RT1170 に設定します。
2. ファイル -> ファイル システムからプロジェクトを開く... -> ディレクトリ: 参照先を wolfSSL/IDE/MCUEXPRESSO/RT1170 ディレクトリに設定し、「ディレクトリの選択」をクリックします。
3. wolfSSL_cm7、wolfcrypt_test_cm7、CSR_example、PKCS7_example を選択します。
4. プロジェクトを右クリック -> SDK 管理 -> SDK コンポーネントを更新し、「はい」をクリックします。
5. FreeRTOSConfig.h の configTOTAL_HEAP_SIZE のサイズを、CSR および PKCS7 の例では 60240、wolfcrypt_test_cm7 では約 100000 に増やします。
6. (ノートボードファイルを再作成する必要があります。これは、同じ設定を持つ新しいプロジェクトを作成し、生成された board/* ファイルをコピーすることで実行できます)
7. プロジェクトをビルドします。

2.10.4.2 RT1170 CAAM ドライバーの拡張 ファイル RT1170/fsl_caam_h.patch および RT1170/fsl_caam_c.patch には、Blob の作成/オープン、および ECC ブラックキーの生成と使用のための既存の NXP CAAM ドライバーへの変更が含まれています。

パッチを適用するには、まず caam ドライバーを含むプロジェクトを作成します。これにより、ドライバーディレクトリにベースとなる fsl_caam.c と fsl_caam.h が生成されます (つまり、PKCS7_example_cm7/drivers/fls_caam.{c,h})。基本ファイルが生成されたら、ドライバー ディレクトリに “cd” してパッチを適用します。

```
cd PKCS7_example_cm7/drivers/
patch -p1 < ../../fsl_caam_c.patch
patch -p1 < ../../fsl_caam_h.patch
```

fsl_caam.h のパッチには、ECC と Blob の拡張(CAAM_ECC_EXPANSION および CAAM_BLOB_EXPANSION)の両方に対するマクロが定義されています。wolfSSL コードはこれらのマクロが定義されている（パッチが適用されている）ことを検出すると、拡張ドライバーの使用をコンパイルしようとします。

3 入門

3.1 概要

以前の CyaSSL であった wolfSSL は、第 2 章で説明されているコンパイルオプションを使用する場合、yaSSL の約 10 倍小さく、OpenSSL の 20 倍小さくなります。ユーザーのベンチマークとフィードバックからも、大部分の標準 SSL 操作で wolfSSL と OpenSSL のパフォーマンスが大幅に向上したことが報告されています。

ビルドプロセスの指示については、第 2 章を参照してください。

3.2 TestSuite

TestSuite プログラムは、wolfSSL とその暗号化ライブラリである wolfCrypt がシステムで実行する能力をテストするように設計されています。

wolfSSL のすべてのサンプルプログラムとテストを実行する際には、wolfSSL ホームディレクトリから実行することが必要です。これは、./certs から証明書と鍵を見つけるために必要です。TestSuite を実行するには次のように実行してください:

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

または autoconf を使用する場合：

```
make test
```

*nix または Windows では、サンプルプログラムと TestSuite が現在のディレクトリがソースディレクトリであるかどうかを確認し、もしそうなら、wolfSSL Home Directory に変更しようとします。これは、ほとんどのセットアップケースで動作するはずです。そうでない場合は、上記の最初の方法を使用して、フルパスを指定するだけです。

テストに成功した場合、このような出力が表示されるはずです。ユニットテストと暗号スイートテストのための追加の出力があります。

```
------------------------------------------------------------------------------
 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!
```

これは、すべてが設定され正しくビルドされていることを示しています。いずれかのテストが失敗した場合は、ビルドシステムが正しく設定されていることを確認してください。おそらく、誤ったエンディアンを持つこと、または 64 ビットタイプを正しく設定しないことが含まれます。デフォルト以外の設定に設定した場合は、それらを削除して、wolfSSL を再ビルドしてから再テストしてください。

3.3 Client サンプルプログラム

examples/client フォルダにあるクライアントのサンプルプログラムを使用して、任意の SSL サーバーに対して wolfSSL をテストできます。利用可能なコマンドランタイムオプションのリストを表示するには、--help 引数でクライアントを実行します：

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

出力表示：

```
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)
```

example.com:443 に対してテストするには、次のことを試してください。これは、--enable-opensslextra および--enable-supportedcurves ビルドオプションでコンパイルされた wolfSSL を使用しています。

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

出力表示：

```
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:20640B8131D8E542646D395B362354F9308057B1624C2442C0B5FCDD064BFE29
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
```

これにより、クライアントは 443 の HTTPS ポート (-p) の (-h)Example.com に接続し、Generic(-g)GET リクエストを送信するように指示します。(-d) オプションは、クライアントにサーバー証明書の認証をしないように指示します。残りは、読み取りバッファーに収まるサーバーからの初期出力です。

コマンドライン引数が指定されていない場合、クライアントは 11111 の wolfSSL デフォルトポートの LOCALHOST への接続を試みます。サーバーがクライアント認証を実行したい場合には、クライアント証明書もロードします。

クライアントは、-b <num> 引数を使用するときに接続をベンチマークすることができます。使用されると、クライアントは引数回数を指定したサーバー/ポートへの接続を試み、その平均時間をミリ秒単位で SSL_connect() で実行します。例：

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

出力表示：

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

LocalHost からデフォルトのホスト、または 11111 からデフォルトポートを変更する場合は、これらの設定を/wolfssl/test.h で変更できます。これらの設定を変更する際のテストスイートを含むすべての例を再ビルドします。そうしないと、テストプログラムが互いに接続できません。

デフォルトでは、wolfSSL の例クライアントは、TLS 1.2 を使用して指定されたサーバーに接続しようとします。ユーザーは、-v コマンドラインオプションを使用してクライアントが使用する SSL/TLS バージョンを変更できます。このオプションでは、次の値が利用できます。

- -v 0 - SSL 3.0(デフォルトでは無効)
- -v 1 - TLS 1.0
- -v 2 -TLS 1.1
- -v 3 -TLS 1.2(デフォルトで選択)
- -v 4 -TLS 1.3

一般的に見るエラーはクライアントのサンプルを使用する場合、-188 です。

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

これは通常、wolfSSL クライアントが接続しているサーバーの証明書を確認できないことによって引き起こされます。デフォルトでは、wolfSSL クライアントは、yaSSL Test CA 証明書を信頼できるルート証明書としてロードします。このテスト CA 証明書は、別の CA によって署名された外部サーバー証明書を確認することができません。そのため、この問題を解決するには、ユーザーは-d オプションを使用して、ピア (サーバー) の検証をオフにする必要があります。

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

または、-A コマンドラインオプションを使用して、正しい CA 証明書を wolfSSL クライアントにロードします。

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

3.4 Server サンプルプログラム

サーバーのサンプルプログラムは、クライアント認証をオプションで実行する単純な SSL サーバーを示しています。1 つのクライアント接続のみが受け入れられ、サーバーが終了します。通常モードのクライアントのサンプルプログラム (コマンドライン引数なし) は、サンプルサーバーに対して正常に動作しますが、クライアントのサンプルプログラムにコマンドライン引数を指定すると、クライアント証明書が読み込まれず、wolfSSL_connect()が失敗します (クライアント証明書がない限り-d オプションを使用してチェックが無効になります)。サーバーはエラー「-245、Peer は CERT を送信しませんでした」を報告します。サ

ンプルプログラムのクライアントと同様に、サーバーはいくつかで使用できます コマンドラインの引数も同様です。

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

出力表示：

```
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 エコーサーバーのサンプルプログラム

エコーサーバのサンプルプログラムは、無制限の数のクライアント接続を待っている無限ループに収まります。クライアントが EchoServer Echoes を返送します。クライアント認証は実行されませんので、サンプルクライアントをすべてのモードでエコーサーバーに対して使用できます。4 つの特別なコマンドは返送しないで、エコーサーバーに別の行動をとるように指示します。

1. quit - エコーサーバーが文字列 “Quit” を受信した場合は、シャットダウンします。
2. break エコーサーバーが文字列 “break” を受信した場合、現在のセッションは停止しますが、要求を処理し続けます。これは DTLS テストに特に便利です。
3. • エコーサーバーが文字列 “PrintStats” を受信した場合は、セッションキャッシュの統計を印刷します。
4. GET -ECHOSERVER が文字列を「取得」すると、HTTP を取得して処理し、「wolfSSL からのグリーティング」というメッセージで簡単なページを送り返します。これにより、Safari、IE、Firefox、Gnutls などのさまざまな TLS/SSL クライアントのテストが可能になります。

NO_MAIN_DRIVER が定義されていない限り、エコーサーバーの出力は STDOUT にエコーされます。シェルを介して、または最初のコマンドライン引数を介して出力をリダイレクトできます。EchoServer Run からの出力を指定して output.txt という名前のファイルを作成するには：

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

3.6 エコークライアントのサンプルプログラム

EchoClient の例は、Interactive Mode または Batch Mode で実行できます。インタラクティブモードで実行して 3 つの文字列 “HELLO”、“WOLFSSL”、および “QUIT” の結果を書き込んでください。

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

入力ファイルを使用するには、最初の引数としてコマンドラインのファイル名を指定します。ファイルの内容をエコーするには input.txt 問題：

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

結果をファイルに書き出す場合は、追加のコマンドライン引数として出力ファイル名を指定できます。次のコマンドはファイル input.txt の内容をエコーし、その結果をサーバーから output.txt に書きます。

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

TestSuite プログラムはそれを行いますが、クライアントとサーバーが正しい結果と期待された結果を取得/送信していることを確認するために、入力ファイルと出力ファイルをハッシュします。

3.7 Benchmark

多くのユーザーは、組み込み機器向け wolfSSLL ライブラリが特定のハードウェアデバイスまたは特定の環境でどのように機能するかに興味があります。現在、組み込み、企業、クラウドベースの環境で使用されているさまざまなプラットフォームとコンパイラがあるため、全面的に一般的なパフォーマンス計算を提供することは困難です。

wolfSSL/WolfCrypt の SSL パフォーマンスを決定する際の wolfSSL ユーザーと顧客を支援するために、wolfSSL にバンドルされているベンチマークアプリケーションが提供されています。wolfSSL は、デフォルトですべての暗号操作に対して wolfCrypt 暗号化ライブラリを使用します。基礎となる暗号は SSL/TLS の非常にパフォーマンスが重要な側面であるため、ベンチマークアプリケーションは wolfCrypt のアルゴリズムでパフォーマンステストを実行します。

wolfcrypt/benchmark にあるベンチマーク ユーティリティ (./wolfcrypt/benchmark/benchmark) を使用して、wolfCrypt の暗号機能のベンチマークを行うことができます。典型的な出力は次のようになりま

す (この出力では、ECC、SHA-256、SHA-512、AES-GCM、AES-CCM、Camellia など、いくつかのオプションのアルゴリズム/暗号が有効になっています)。

```
./wolfcrypt/benchmark/benchmark
```

出力結果:

```
------------------------------------------------------------------------------
 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
```

これは、数学ライブラリを変更する前後に公開キーの速度を比較するのに特に役立ちます。通常の数学ライブラリ (`./configure`)、Fastmath Library(`./configure --enable-fastmath`)、および Fasthugemath Library(`./configure --enable-fasthugemath`) を使用して結果をテストできます。

詳細やベンチマークの結果については、wolfSSL ベンチマークページを参照してください。<https://www.wolfssl.com/docs/benchmarks>

3.7.1 相対パフォーマンス

個々の暗号とアルゴリズムの性能はホストプラットフォームによって異なりますが、次のグラフは wolfCrypt の暗号の間の相対性能を示しています。これらのテストは、2.2 GHz Intel Core I7 を実行している MacBook Pro(OS X 10.6.8) で行われました。

暗号のサブセットのみを使用する場合は、SSL/TLS 接続を作成するときに wolfSSL が使用する特定の暗号

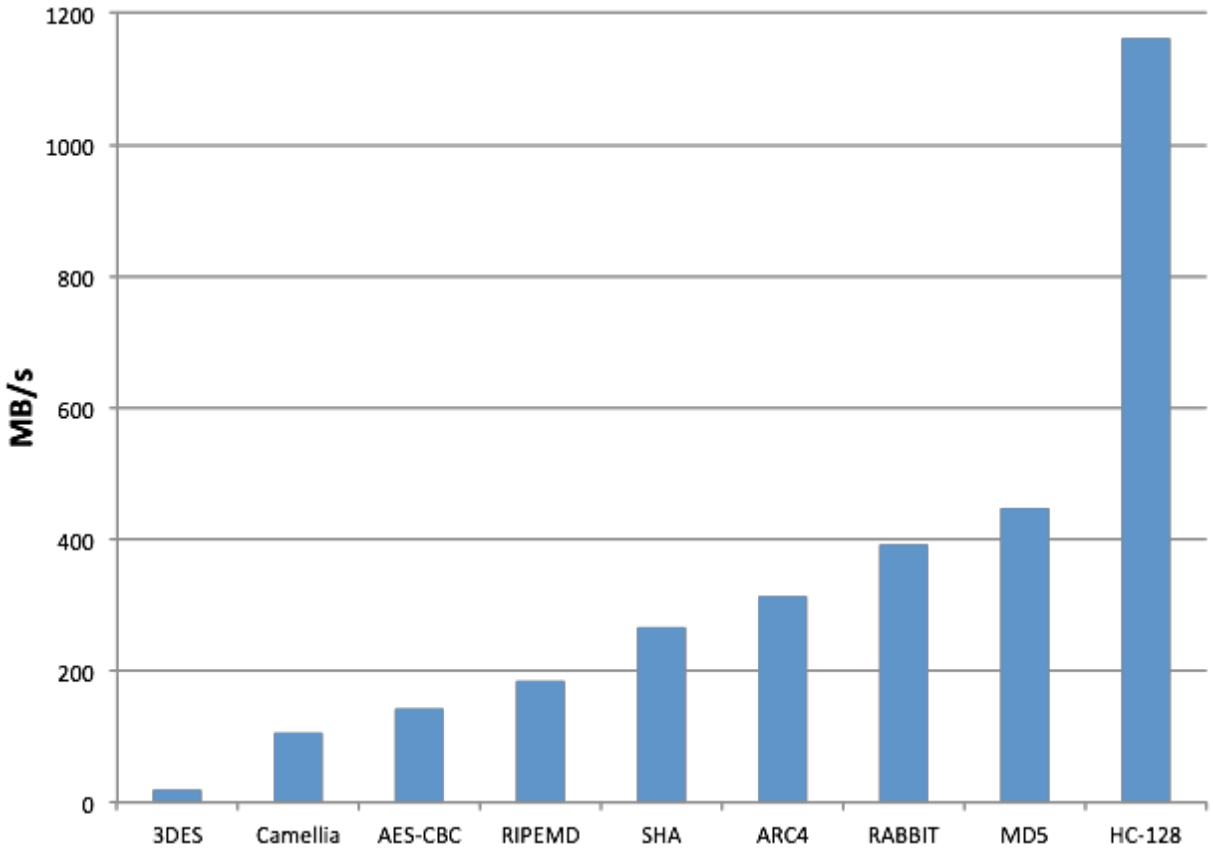

Figure 1: Benchmark

スイートおよび/または Ciphers をカスタマイズできます。たとえば、128 ビット AES を強制するには、コールの後にwolfSSL_CTX_new(SSL_CTX_new)に次の行を追加します。

```
wolfSSL_CTX_set_cipher_list(ctx, “AES128-SHA”);
```

3.7.2 Benchmark の補足

1. プロセッサネイティブレジスタサイズ (32 対 64 ビット) は、1000 ビット以上の公開キー操作を実行する場合、大きな違いを生じる可能性があります。
2. ** keygen **(--enable-keygen) では、ベンチマークユーティリティを実行するときに鍵生成速度をベンチマークすることもできます。
3. ** FastMath **(--enable-fastmath) は動的メモリ使用量を減らし、公開鍵操作を高速化します。FastMath を使用して 32 ビットプラットフォームでビルドするのに問題がある場合は、PIC がレジスタを占有しないように共有ライブラリビルドを無効にしてみてください。(ReadMe のノートも参照)。

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

注：wolfSSL で設定オプションを切り替える際に、make clean をすることは良い習慣です。

4. デフォルトでは、FastMath は可能であればアセンブリ最適化を使用しようとします。アセンブリの最適化が機能しない場合は、wolfSSL をビルドするときに TFM_NO_ASM から CFLAGS を追加することで、FastMath を使用することができます。

   ```
   ./configure --enable-fastmath C_EXTRA_FLAGS=”-DTFM_NO_ASM”
   ```

5. fasthugemath を使用すると、組み込みプラットフォームで実行されていないユーザーのために、Fastmath をさらにプッシュしようとすることができます。

   ```
   ./configure --enable-fasthugemath
   ```

6. デフォルトの wolfSSL ビルドを使用すると、メモリ使用量とパフォーマンスのバランスをうまくとろうとしています。メモリ使用量とパフォーマンスのいずれかに関心がある場合は、追加の wolfSSL 設定オプションについてビルドオプションを参照してください。
7. **バルク転送**：wolfSSL はデフォルトで 128 バイト I/O バッファーを使用します。SSL トラフィックの約 80%はこのサイズ内で、動的メモリの使用を制限します。バルク転送が必要な場合は、16K バッファー (最大 SSL サイズ) を使用するように構成できます。

3.7.3 組み込みシステムのベンチマーク

組み込みシステムにベンチマークアプリケーションをビルドするために、いくつかのビルドオプションがあります。これらには以下が含まれます：

3.7.3.1 BENCH_EMBEDDED この定義を有効にすると、ベンチマークアプリケーションがメモリ使用量をメガバイトの使用からキロバイトの使用へ切り替えるため使用量が削減されます。デフォルトでは、この定義を使用する場合、暗号とアルゴリズムは 25KB でベンチマークを実行します。公開鍵アルゴリズムは、1 回以上の反復を超えてのみベンチマークされます (一部の組み込みプロセッサの公開鍵操作がかなり遅くなる可能性がある)。numBlocks と times を変更することで、numBlocks と times を変更することで benchmark.c で調整できます。

3.7.3.2 USE_CERT_BUFFERS_1024 この定義を有効にすると、ベンチマークアプリケーションがファイルシステムからのテスト用鍵と証明書のロードから切り替えられ、代わりに <wolfssl_root>/wolfssl/certs_test.h にある 1024 ビット鍵と証明書バッファを使用しま

す。組み込みプラットフォームにファイルシステムがない場合 (NO_FILESYSTEM) で 2048 ビットの公開鍵操作が合理的でない程遅いプロセッサにこの定義を使用するのは便利です。

3.7.3.3 USE_CERT_BUFFERS_2048 この定義を有効にすることは、USE_CERT_BUFFERS_1024と同様に 2048 ビット公開鍵と証明書を受け入れます。この定義は、プロセッサが十分な速さでできる場合に役立ちます 2048 ビット公開キー操作ですが、ファイルからキーと証明書をロードできるファイルシステムがない場合。

3.8 クライアントアプリケーションを変更して wolfSSL を使用します

このセクションでは、wolfSSL ネイティブ API を使用して、クライアントアプリケーションに wolfSSL を追加するために必要な基本的な手順について説明します。サーバーの説明については、wolfSSL を使用するためにサーバーアプリケーションを変更しますを参照してください。11 章の SSL チュートリアルに例を挙げたより完全なウォークスルーがあります。OpenSSL 互換性レイヤーの詳細については、OpenSSL 互換性を参照してください。

1. wolfSSL ヘッダーを含める：

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

2. wolfSSL と WOLFSSL_CTX を初期化します。最終的に作成する WOLFSSL オブジェクトの数に関係なく、1 つの WOLFSSL_CTX を使用できます。基本的に、接続しているサーバーを確認するために CA 証明書を読み込む必要があります。基本的な初期化は次のようになります：

   ```
   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. 各 TCP 接続後に wolfSSL オブジェクトを作成し、ファイル記述子をセッションに関連付けます。

   ```
   /*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. すべての呼び出しを read()(または recv()) からwolfSSL_read()に変更します。

   ```
   result=read(fd, buffer, bytes);
   ```

   次のようになります

   ```
   result=wolfSSL_read(ssl, buffer, bytes);
   ```

5. すべての呼び出しを write()(または send()) からwolfSSL_write()に変更します。

   ```
       result=write(fd, buffer, bytes);
   ```

次のようになります

```
result=wolfSSL_write(ssl, buffer, bytes);
```

6. wolfSSL_connect()を開始します。
7. エラーチェック。各wolfSSL_read()呼び出しは、read() と write() と同じように、成功時に 0、接続クロージャーの場合は 0、およびエラーの間のバイト数を返します。エラーが発生した場合は、2 つの関数を使用してエラー情報を入手できます：

   ```
   char errorString[80];
   int err=wolfSSL_get_error(ssl, 0);
   wolfSSL_ERR_error_string(err, errorString);
   ```

   ノンブロッキング ソケットを使用している場合は、errno EAGAIN/EWOULDBLOCK をテストできます。より正確には、SSL_ERROR_WANT_READ または SSL_ERROR_WANT_WRITE に対してwolfSSL_get_error() によって返される特定のエラー コードをテストできます。
8. 後処理。各 wolfSSL オブジェクトが使用された後、それらを解放します。

   呼び出し：

   ```
   wolfSSL_free(ssl);
   ```

   SSL/TLS の使用が完全に終了したら、WOLFSSL_CTX オブジェクトを呼び出して解放できます。

   ```
   wolfSSL_CTX_free(ctx);
   wolfSSL_Cleanup();
   ```

wolfSSL を使用したクライアントアプリケーションの例については、<wolfssl_root>/examples/client.c ファイルにあるクライアントの例を参照してください。

3.9 wolfSSL を使用するためにサーバーアプリケーションを変更します

このセクションでは、wolfSSL ネイティブ API を使用して wolfSSL をサーバーアプリケーションに追加するために必要な基本的な手順について説明します。クライアントの説明については、クライアントアプリケーションを変更して wolfSSL を使用しますをご覧ください。

1. ステップ 5 でクライアントメソッド呼び出しをサーバーに変更する以外のクライアントの手順に従ってください。

   ```
   wolfSSL_CTX_new(wolfTLSv1_client_method());
   ```

   次のようになります

   ```
   wolfSSL_CTX_new(wolfTLSv1_server_method());
   ```

   あるいは：

   ```
   wolfSSL_CTX_new(wolfSSLv23_server_method());
   ```

   SSLV3 および TLSV1+ クライアントがサーバーに接続できるようにします。
2. 上記のステップ 5 の初期化にサーバーの証明書とキーファイルを追加します。

   ```
   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);
    }
```

使用可能なファイルシステムがない場合は、バッファーから証明書とキーをロードすることもできます。この場合、詳細については、ここにリンクされているwolfSSL_CTX_use_certificate_buffer() API ドキュメントを参照してください。

wolfSSL を使用したサーバーアプリケーションの例については、<wolfssl_root>/examples/server.c ファイルにあるサーバーの例を参照してください。

4 機能

wolfSSL (以前の CyaSSL) は、主要なインターフェースとして C プログラミング言語をサポートしていますが、Java、PHP、Perl、Python など、他のいくつかのホスト言語もサポートしています (SWIG インターフェースを介して)。現在サポートされていない別のプログラミング言語で wolfSSL をホストすることに関心がある場合は、お問い合わせください。

この章では、ストリーム暗号、AES-NI、IPv6 サポート、SSL 検査 (SNIFFER) サポートなど、wolfSSL のいくつかの機能について、より詳しく説明しています。

4.1 機能の概要

wolfSSL 機能の概要については、wolfSSL 製品 Web ページを参照してください：https://www.wolfssl.com/products/wolfssl

4.2 プロトコルサポート

wolfssl は ** SSL 3.0 **、** tls **(1.0 **、** 1.1 **、** 1.2、1.3 **)、および** dtls **(1.0 **および** 1.2 **)。以下の機能のいずれかを使用して、使用するプロトコルを簡単に選択できます (クライアントまたはサーバーのいずれかに示すように)。wolfSSL は、SSL 2.0 をサポートしていません。OpenSSL 互換性レイヤーを使用すると場合、クライアントとサーバーの機能はわずかに異なります。OpenSSL 互換機能については、OpenSSL 互換性を参照してください。

4.2.1 サーバー機能

- `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()` - SSLv3 - TLS 1.2 から最高のバージョンを使用する

wolfSSL は、`wolfSSLv23_server_method()`関数で堅牢なサーバーダウングレードをサポートしています。詳細については、堅牢なクライアントとサーバーのダウングレードを参照してください。

4.2.2 クライアント機能

- `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()` - SSLv3 - TLS 1.2 から最高のバージョンを使用する

wolfSSL は、wolfSSLv23_client_method()関数でロバストクライアントのダウングレードをサポートしています。詳細については、堅牢なクライアントとサーバーのダウングレードを参照してください。

これらの機能の使用方法の詳細については、入門の章を参照してください。SSL 3.0、TLS 1.0、1.1、1.2、および DTLS の比較については、付録 A を参照してください。

4.2.3 堅牢なクライアントとサーバーのダウングレード

wolfSSL クライアントとサーバーの両方に、堅牢なバージョンのダウングレード機能があります。どちらの側で特定のプロトコルバージョンメソッドが使用されている場合、そのバージョンのみがネゴシエートされるか、エラーが返されます。たとえば、TLS 1.0 を使用して SSL 3.0 のみのサーバーに接続しようとするクライアントは、接続が失敗し、同様に TLS 1.1 に接続すると同様に失敗します。

この問題を解決するために、wolfSSLv23_client_method()関数を使用するクライアントは、必要に応じてダウングレードすることによりサーバーがサポートする最高のプロトコルバージョンをサポートします。この場合、クライアントは TLS 1.0 -TLS 1.3 を実行しているサーバーに接続できます (または、wolfSSL で構成されているプロトコルバージョンに応じて SSL 3.0 を含むサブセットまたはスーパーセット)。接続できない唯一のバージョンは、長年にわたって不安定である SSL 2.0 と、デフォルトで無効になっている SSL 3.0 です。

同様に、wolfSSLv23_server_method()関数を使用するサーバーは、TLS 1.0 -TLS 1.2 のプロトコルバージョンをサポートするクライアントを処理できます。wolfSSL サーバーは、セキュリティが提供されていないため、SSLV2 からの接続を受け入れることができません。

4.2.4 IPv6 サポート

IPv6 を採用していて、組み込み SSL 実装を使用したい場合、wolfSSL が IPv6 をサポートしているかどうか疑問に思っているかもしれません。答えはイエスです。IPv6 の上で実行されている wolfSSL をサポートしています。

wolfSSL は IP ニュートラルとして設計されており、IPv4 と IPv6 の両方で動作しますが、現在のテストアプリケーションは IPv4 にデフォルトになります (より広い範囲のシステムに適用されます)。テストアプリケーションを IPv6 に変更するには、wolfSSL のコンフィグれションに ** –enable-IPv6 ** オプションを使用します。

IPv6 に関する詳細情報はここにあります。

https://en.wikipedia.org/wiki/IPv6。

4.2.5 DTLS

wolfSSL は、クライアントとサーバーの両方の DTLS(“データグラム” TLS) をサポートしています。現在のサポートされているバージョンは DTLS 1.0 です。

TLS プロトコルは、**信頼性の高い**媒体 (TCP など) に安全なトランスポートチャネルを提供するように設計されています。アプリケーション層プロトコルが UDP トランスポート (SIP やさまざまな電子ゲーム プロトコルなど) を使用して開発され始めたため、遅延に敏感なアプリケーションに通信セキュリティを提供する方法が必要になりました。この必要性は、DTLS プロトコルの作成につながります。

多くの人々が TLS と DTLS の違いが TCP と UDP と同じであると考えています。これは正しくありません。UDP には、(TCP と比較して) 何かが失われた場合に、ハンドシェイク、ティアダウン、および途中での遅延がないという利点があります。一方、DTLS は、拡張 SSL ハンドシェイクと引き裂きを持ち、ハンドシェイクの TCP のような動作を実装する必要があります。本質的に、DTLS は安全な接続と引き換えに UDP によって提供される利点を逆にします。

DTL は、--enable-dtlsビルドオプションを使用して wolfSSL をビルドするときに有効にできます。

4.2.6 LWIP(軽量インターネットプロトコル)

wolfSSL は、軽量のインターネットプロトコルの実装をすぐに使えるようにサポートしています。このプロトコルを使用するには、DEFINE WOLFSSL_LWIP、または settings.h ファイルに移動して以下の行のコメントアウトを外してください。

```
/*#define WOLFSSL_LWIP*/
```

LWIP の焦点は、完全な TCP スタックを提供しながら、RAM の使用量を減らすことです。その焦点は、wolfSSL が SSL/TLS ニーズに理想的なマッチであるエリアである Embedded Systems での使用に最適です。

4.2.7 TLS エクステンション

wolfSSL によってサポートされている TLS 拡張機能のリストと、指定された拡張子に対して RFC を参照することができます。

RFC	拡張	wolfssl タイプ
6066	サーバー名表示	TLSX_SERVER_NAME
6066	最大フラグメント長ネゴシエーション	TLSX_MAX_FRAGMENT_LENGTH
6066	切り捨てられた hmac	TLSX_TRUNCATED_HMAC
6066	ステータスリクエスト	TLSX_STATUS_REQUEST
7919	サポートされているグループ	TLSX_SUPPORTED_GROUPS
5246	署名アルゴリズム	TLSX_SIGNATURE_ALGORITHMS
7301	アプリケーション層プロトコルネゴシエーション	TLSX_APPLICATION_LAYER_PROTOCOL
6961	証明書ステータスリクエスト	TLSX_STATUS_REQUEST_V2
5077	セッションチケット	TLSX_SESSION_TICKET
5746	再ネゴシエーションの提示	TLSX_RENEGOTIATION_INFO
8446	鍵共有	TLSX_KEY_SHARE
8446	事前共有鍵	TLSX_PRE_SHARED_KEY
8446	PSK 交換モード	TLSX_PSK_KEY_EXCHANGE_MODES
8446	アーリーデータ	TLSX_EARLY_DATA
8446	クッキー	TLSX_COOKIE
8446	サポートバージョン	TLSX_SUPPORTED_VERSIONS
8446	ハンドシェーク承認	TLSX_POST_HANDSHAKE_AUTH

4.3 暗号サポート

4.3.1 暗号スイート強度と適切な鍵サイズの選択

どの暗号が現在使用されているかを確認するには､メソッドを呼び出すことができます.wolfSSL_get_ciphers()。

この関数は、現在有効な暗号スイートを返します。

暗号スイートにはさまざまな強みがあります。それらはいくつかの異なるタイプのアルゴリズム (認証、暗号化、およびメッセージ認証コード (MAC)) で構成されているため、それぞれの強度は選択された鍵サイズによって異なります。

暗号スイートの強度を評価する多くの方法があります。使用される特定の方法は、プロジェクトや企業によって異なると思われ、対称および公開鍵アルゴリズムの鍵サイズ、アルゴリズムの種類、パフォーマンス、既知の弱点などを含めることができます。

** nist **(米国立標準技術研究所) は、それぞれのさまざまな鍵サイズに同等のアルゴリズム強度を提供することにより、許容可能な暗号スイートを選択することを推奨します。暗号化アルゴリズムの強度は、アルゴリズムと使用される鍵サイズに依存します。NIST Special Publication、SP800-57は、2 つのアルゴリズムが次のように同等の強度であると見なされると述べています。

2 つのアルゴリズムは、「アルゴリズムを壊す」または鍵を決定するか (与えられた鍵サイズで) 鍵を使用してほぼ同じである場合、2 つのアルゴリズムが与えられた鍵サイズ (x と y) に対して匹敵する強さと考えられています。資源。特定の鍵サイズのアルゴリズムのセキュリティ強度は、ショートカット攻撃を有していない「X」の鍵サイズを持つすべての鍵を試してみるのにかかる作業量に関して説明しています (すなわち、最も効率的なもの)。攻撃はすべての可能な鍵を試すことです)。

次の 2 つの表は、NIST SP 800-57の表 2(pg。56) と表 4(pg。59) の両方に基づいており、アルゴリズム間の同等のセキュリティ強度と強度測定 (NIST が推奨するアルゴリズムのセキュリティ ライフタイムに基づいており、セキュリティの一部を使用しています)。

注：次の表「L」は有限フィールド暗号化 (FFC) の公開鍵のサイズであり、「n」は FFC の秘密鍵のサイズで、「K」は重要なサイズと見なされます。整数因数分解暗号化 (IFC)、および「F」は、楕円曲線暗号の重要なサイズと見なされます。

セキュリティのビット	対称鍵アルゴリズム	**FFC** 鍵サイズ (DSA、DH など)	**ifc** 鍵サイズ (RSA など)	**ECC** 鍵サイズ (E
80	2tdea など	L=1024、n=160	K=1024	F=160-223
128	AES-128 など	L=3072、n=256	K=3072	F=256-383
192	AES-192 など	L=7680、n=384	K=7680	f=384-511
256	AES-256 など	L=15360、n=512	K=15360	f=512+

このテーブルをガイドとして使用して、暗号スイートを分類し始めるために、対称暗号化アルゴリズムの強度に基づいて分類します。これを行う際には、セキュリティのビットに基づいて各暗号スイートを分類することを考案することができます (対称鍵サイズを考慮してください)。

- **低** - 128 ビット未満のセキュリティのビット
- **中** - セキュリティのビット 128 ビット
- **高** - 128 ビットより大きいセキュリティのビット

対称暗号化アルゴリズムの強度以外では、暗号スイートの強度は、鍵交換および認証アルゴリズム鍵の鍵サイズに大きく依存します。強度は、暗号スイートの最も弱いリンクと同程度です。

上記の等級付け方法に従って (対称暗号化アルゴリズムの強度のみに基づいて)、wolfSSL 2.0.0 は現在、以下に示すように、低強度の暗号スイート 0 個、中強度の暗号スイート 12 個、高強度の暗号スイート 8 個をサポートしています。以下の強度の分類は、関与する他のアルゴリズムの選択された鍵サイズによって変わる可能性があります。ハッシュ関数セキュリティ強度については、NIST SP 800-57の表 3(56) を参照のこと。

場合によっては、“**EXPORT**” の暗号として参照されている暗号が表示されます。これらの暗号は、米国から強い暗号のソフトウェアを輸出することが違法であったときの米国履歴の期間 (1992 年に遅く) に由来しました。強い暗号は、米国政府によって (核兵器、戦車、弾道ミサイルなど) によって「弾薬」として分類されました。この制限のため、エクスポートされているソフトウェアは「弱められた」暗号 (主により小さな鍵サイズ) を含んでいます。現在、この制限は解除されているため、EXPORT 暗号は必須ではなくなりました。

4.3.2 サポートされている暗号スイート

次の暗号スイートは、wolfSSL によってサポートされています。暗号スイートは、TLS または SSL ハンドシェイク中に使用される認証、暗号化、およびメッセージ認証コード (MAC) アルゴリズムの組み合わせで、接続のセキュリティ設定をネゴシエートします。

各暗号スイートは、キー交換アルゴリズム、一括暗号化アルゴリズム、およびメッセージ認証コード アルゴリズム (MAC) を定義します。**キー交換アルゴリズム** (RSA、DSS、DH、EDH) は、ハンドシェイク プロセス中にクライアントとサーバーが認証する方法を決定します。メッセージ ストリームの暗号化には、ブロック暗号とストリーム暗号を含む **一括暗号化アルゴリズム** (DES、3DES、AES、ARC4) が使用されます。**メッセージ認証コード (MAC) アルゴリズム** (MD2、MD5、SHA-1、SHA-256、SHA-512、RIPEMD) は、メッセージ ダイジェストの作成に使用されるハッシュ関数です。

以下の表は、<wolfssl_root>/wolfssl/internal.h(約 706 行から始まる) にある暗号スイート (およびカテゴリ) と一致しています。次のリストにない暗号スイートをお探しの場合は、wolfSSL に追加することについてお問い合わせください。

ECC 暗号スイート：

- 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

静的 ECC 暗号スイート：

- 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 暗号スイート：

- TLS_RSA_WITH_AES_128_CBC_B2B256
- TLS_RSA_WITH_AES_256_CBC_B2B256

SHA-256 暗号スイート：

- 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 暗号スイート：

- TLS_DHE_PSK_WITH_AES_256_CBC_SHA384
- TLS_DHE_PSK_WITH_NULL_SHA384

AES-GCM 暗号スイート：

- 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 暗号スイート：

- 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 暗号スイート：

- 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 暗号スイート：

- 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

再ネゴシエーション提示拡張特別スイート：

- TLS_EMPTY_RENEGOTIATION_INFO_SCSV

4.3.3 AEAD スイート

wolfSSL は、AES-GCM、AES-CCM、Chacha-Poly1305 などの AEAD スイーツをサポートしています。これらの AEAD スイートと他のスイートの大きな違いは、追加のクリアテキストデータを使用して暗号化されたデータを認証することです。これは、データを改ざんすることになる中間者攻撃を緩和するのに役立ちます。AEAD スイートは、キー付きハッシュアルゴリズムによって生成されたタグと組み合わせたブロック暗号 (または最近でもストリーム暗号) アルゴリズムの組み合わせを使用します。これらの 2 つのアルゴリズムを組み合わせることで、これら 2 つのアルゴリズムの組み合わせは、ユーザーにとって簡単な wolfSSL 暗号化および復号化プロセスによって処理されます。特定の AEAD スイートを使用するために必要なのは、サポートされているスイートで使用されるアルゴリズムを単純に有効にすることです。

4.3.4 ブロックとストリーム暗号

wolfSSL は **AES**、**DES**、**3DES**、**Camellia** ブロック暗号と **RC4**、および **Chacha20** ストリーム暗号をサポートしています。AES、DES、3DES、RC4 はデフォルトで有効になっています。Camellia、および Chacha20 は、(`--enable-camellia`、および`--disable-chacha`ビルドオプションで)wolfSSL をビルドするときに有効にすることができます。AES のデフォルトモードは CBC モードです。AES を使用して GCM または CCM モードを有効にするには、`--enable-aesgcm`と`--enable-aesccm`ビルドオプションを使用します。特定の使用状況については、使用例とwolfCrypt の使用法の例をご覧ください。

SSL は RC4 をデフォルトのストリーム暗号として使用しますが、脆弱性のために廃止されています。最近、wolfSSL は Chacha20 を追加しました。RC4 は Chacha よりも約 11%パフォーマンスが高いですが、RC4 は一般に Chacha よりも安全性が低いと考えられています。Chacha は、トレードオフとしてセキュリティを追加することで一時代を作り出すと考えられます。

暗号性能の比較を見るには、ここにある wolfSSL ベンチマーク Web ページを参照してください。<https：//www.wolfssl.com/docs/benchmarks>。

4.3.4.1 違いは何ですか？ ブロック暗号は、暗号のブロックサイズであるチャンク単位で暗号化する必要があります。たとえば、AES には 16 バイトのブロックサイズがあります。そのため、2 バイトまたは 3 バイトの小さなチャンクを何度も暗号化している場合、データの 80% 以上が無駄なパディングであり、暗号化/復号化プロセスの速度が低下し、起動するためにネットワーク帯域幅が不必要に浪費されます。基本的にブロック暗号は、大きなデータのチャンク用に設計されており、パディングを必要とするブロックサイズを有し、固定されていない変換を使用します。

ストリーム暗号は、大量または小さなデータに適しています。ブロックサイズが不要なため、データサイズが小さい場合に適しています。速度が懸念される場合、ストリーム暗号はあなたの答えです。なぜなら、それらは通常、Xor のキーストリームを含むよりシンプルな変換を使用するからです。したがって、メディアをストリーミングする必要がある場合、小さなデータを含むさまざまなデータサイズを暗号化する、または高速暗号が必要な場合は、暗号をストリーミングすることが最善の策です。

4.3.5 ハッシュ機能

wolfSSL は **MD2,MD4,MD5,SHA-1,SHA2**(SHA-224,SHA-256,SHA-384,SHA-512), **SHA-3**(BLAKE2)、および **RIPEMD-160** などのハッシュ機能をサポートしています。これらの機能の詳細な使用法は、wolfCrypt の使用法、ハッシュ関数にあります。

4.3.6 公開鍵オプション

wolfSSL は **RSA**、**ECC**、**DSA/DSS** と **DH** 公開鍵オプションをサポートしています。これらの機能の詳細な使用法は、wolfCrypt の使用法、公開鍵暗号にあります。

4.3.7 ECC サポート

wolfSSL は、ECDH-ECDSA、ECDHE-ECDSA、ECDH-RSA、ECDHE-PSK、ECDHE-RSA を含むがこれらに限定されない楕円曲線暗号化 (ECC) をサポートしています。

wolfSSL の ECC 実装は、<wolfssl_root>/wolfssl/wolfcrypt/ecc.h ヘッダーファイルと <wolfssl_root>/wolfcrypt/src/ecc.c ソースファイルにあります。

サポートされている暗号スイートは、上の表に示されています。ECC は、x86_64 以外のビルドでデフォルトで無効になりますが、HAVE_ECC を使用して wolfSSL をビルドするときまたは autoconf システムを使用してオンにすることができます。

```
./configure --enable-ecc
make
make check
```

make check が実行される場合は、wolfSSL がチェックする多数の暗号スイートに注意してください (チェックが Cipher Suites のリストが作成されていない場合、それ自体で ./testsuite/testsuite.test を実行します)。これらの暗号スイートのいずれかを個別にテストすることができます。たとえば、AES256-SHA で ECDH-ECDSA を試すために、wolfSSL サーバーの例は次のように開始できます。

```
./examples/server/server -d -l ECDHE-ECDSA-AES256-SHA -c
↪  ./certs/server-ecc.pem -k ./certs/ecc-key.pem
```

(-d) クライアントの証明書チェックを無効にし、(-l) は暗号スイートリストを指定します。(-c) は使用する証明書であり、(-k) は、使用する対応する秘密鍵です。クライアントを接続するには、次のことを試してください。

```
./examples/client/client -A ./certs/server-ecc.pem
```

ここで、(-A) は、サーバーの検証に使用する CA 証明書です。

4.3.8 PKCS サポート

PKCS(公開鍵暗号化スタンダード) は、RSA Security、Inc. によって作成および公開された標準のグループを指します。wolfSSL は **PKCS # 1、PKCS # 3、PKCS # 5、PKCS # 7、PKCS # 8、PKCS # 9、PKCS # 10、PKCS # 11** および **PKCS # 12** をサポートしています。

さらに、wolfSSL は、PKCS # 1 の一部として標準化されている RSA-Probabilistic Signature Scheme(PSS) のサポートも提供しています。

4.3.8.1 PKCS #5, PBKDF1, PBKDF2, PKCS #12 PKCS # 5 は、パスワード、ソルト、および繰り返し回数を合成してパスワードベースのキーを生成するパスワードベースの鍵導出方法です。wolfSSL は PBKDF1 と PBKDF2 鍵導出機能の両方をサポートしています。鍵導出関数は、基本鍵および他のパラメータ (上述のようにソルトおよび反復回数など) から派生鍵を生成する。PBKDF1 は、派生鍵長がハッシュ関数出力の長さによって囲まれている鍵を導出するためにハッシュ関数 (MD5、SHA1 など) を適用します。PBKDF2 では、鍵を導出するために疑似ランダム関数 (HMAC-SHA-1 など) が適用されます。PBKDF2 の場合、派生鍵の長さは無制限です。

wolfSSL は、PBKDF1 および PBKDF2 に加えて、PKCS # 12 の PBKDF 関数もサポートしています。関数プロトタイプは次のようになります：

```
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 には派生鍵が含まれ、passwd は長さ pLen のユーザーパスワードを保持します pLen、salt は長さ sLen、iterations のソルト入力を保持します。md5、sha1、または sha2 にすることができます。

./configure を使用して wolfSSL をビルドしている場合、この機能を有効にする方法は、オプション--enable-pwdbasedを使用することです

完全な例は <wolfSSL Root>/wolfcrypt/test.c にあります。詳細については、次の仕様から PKCS # 5、PBKDF1、および PBKDF2 にあります。

PKCS # 5、PBKDF1、PBKDF2：https://tools.ietf.org/html/rfc2898

4.3.8.2 PKCS #8 PKCS # 8 は、秘密鍵情報を格納するために使用されている秘密鍵情報構文標準として設計されています。

PKCS #8 標準には、暗号化された秘密鍵と暗号化されていない鍵の両方を格納するための構文を説明する 2 つのバージョンがあります。wolfSSL は、暗号化されていない PKCS # 8 の両方をサポートしています。サポートされている形式には、PKCS # 5 バージョン 1-バージョン 2、および PKCS # 12 が含まれます。利用可能な暗号化の種類には、DES、3DES、RC4、および AES が含まれます。

PKCS # 8：https://tools.ietf.org/html/rfc5208

4.3.8.3 PKCS #7 PKCS #7 は、エンベロープ証明書であれ、暗号化されていないが署名されたデータ文字列であれ、データのバンドルを転送するように設計されています。この機能は、有効化オプション (--enable-pkcs7) を使用するか、マクロ HAVE_PKCS7 を使用してオンにします。署名者の空のセットを持つ RFC に従って、デフォルトで縮退ケースが許可されることに注意してください。関数 wc_PKCS7_AllowDegenerate() を呼び出して、縮退ケースのオンとオフを切り替えることができます。

サポートされている機能は次のとおりです。

- 縮退した束
- Kari、Kekri、Pwri、Ori、Ktri バンドル
- 切り離された署名
- 圧縮およびファームウェアパッケージバンドル
- カスタムコールバックサポート
- 限られたストリーミング機能

4.3.8.3.1 PKCS # 7 コールバック ユーザーが PKCS7 バンドルが解析された後にユーザーが自分の鍵を選択できるように追加のコールバックおよびサポート機能が追加されました。CEK をアンラップするには、関数 wc_PKCS7_SetWrapCEKCb() を呼び出すことができます。この関数によって設定されたコールバックは、KARI と KEKRI バンドルの場合に呼び出されます。キー ID または SKID は、KARI の場合はオリジネータ キーとともに wolfSSL からユーザーに渡されます。ユーザーが自分の KEK で CEK をアンラップした後、使用する復号化されたキーを wolfSSL に戻す必要があります。この例は、ファイル signedData-EncryptionFirmwareCB.c の wolfSSL-examples リポジトリにあります。

PKCS7 バンドルの復号のために追加のコールバックが追加されました。復号コールバック関数を設定するには、API wc_PKCS7_SetDecodeEncryptedCb() を使用できます。ユーザー定義のコンテキストを設定するには、API wc_PKCS7_SetDecodeEncryptedCtx() を使用する必要があります。このコールバックは、wc_PKCS7_DecodeEncryptedData() の呼び出しで実行されます。

4.3.8.3.2 PKCS # 7 ストリーミング PKCS7 デコード用のストリーム指向 API により、入力を一度に渡すのではなく、小さなチャンクで渡すオプションが提供されます。デフォルトでは、PKCS7 によるストリーミング機能はオンになっています。ストリーミング PKCS7 API のサポートをオフにするには、マクロ NO_PKCS7_STREAM を定義できます。Autotools でこれを行う例は./configure --enable-pkcs7 CFLAGS=-DNO_PKCS7_STREAM です。

バンドルのデコード/検証時のストリーミングでは、次の関数がサポートされています。

1. wc_PKCS7_DecodeEncryptedData()
2. wc_PKCS7_VerifySignedData()
3. wc_PKCS7_VerifySignedData_ex()
4. wc_PKCS7_DecodeEnvelopedData()
5. wc_PKCS7_DecodeAuthEnvelopedData()

** NOTE **：wc_PKCS7_VerifySignedData_exを呼び出したとき、引数 pkimsgfoot がフルバッファであることが予想されます。内部構造は、この場合に pkiMsgHead になる 1 つのバッファのストリーミングのみをサポートします。

4.3.9 特定の暗号スイート強制使用

デフォルトでは、wolfSSL は、接続の両側がサポートできる「最高の」(最高のセキュリティ) 暗号スイートを選択します。128 ビット AES などの特定の暗号を強制するには、次のようなものを追加します。

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

wolfSSL_CTX_new()への呼び出しの後に：

```
ctx=wolfSSL_CTX_new(method);
wolfSSL_CTX_set_cipher_list(ctx, "AES128-SHA");
```

4.3.10 OpenQuantumsafe の liboqs 統合

詳細については、このドキュメントの「Quantum-Safe Cryptography の実験」を参照してください。

4.4 ハードウェアを使っての暗号の高速化

wolfSSL は、さまざまなプロセッサやチップの中のいくつかのハードウェアが加速された (または「支援」) 暗号機能を利用することができます。次のセクションでは、wolfSSL がどのテクノロジをサポートしているかについて説明します。

4.4.1 AES-NI

AES は、wolfSSL が常にサポートしてきた世界中の政府が使用する重要な暗号化基準です。Intel は、AES をより高速に実装する新しい命令セットをリリースしました。wolfSSL は、生産環境向けの新しい命令セットを完全にサポートする最初の SSL ライブラリです。

基本的に、Intel と AMD は、AES アルゴリズムの計算集約型部分を実行する CHIP レベルで AES 命令を追加し、パフォーマンスを向上させました。現在 AES-NI をサポートしている Intel のチップのリストについては、こちらをご覧ください。

https://ark.intel.com/search/advanced/?s=t&AESTech=true

ソフトウェアでアルゴリズムを実行する代わりに、wolfSSL に機能を追加して、チップから命令を直接呼び出すことができます。これは、AES-NI をサポートするチップセットで wolfSSL を実行している場合、AES Crypto を 5〜10 倍速く実行できることを意味します。

AES-NI サポートされたチップセットで実行されている場合は、--enable-aesni build optionで AES-NI を有効にします。AES-NI で wolfSSL をビルドするには、GCC 4.4.3 以降はアセンブリコードを使用する必要があります。wolfSSL は、同じビルドオプションを使用して AMD プロセッサの ASM 命令をサポートしています。

AES-NI に関する参考文献と参考資料を、一般的なものから特定のものまで、以下に示します。AES-NI によるパフォーマンス向上の詳細については、Intel Software Network ページへの 3 番目のリンクを参照してください。

- AES(Wikipedia)
- AES-NI(Wikipedia)
- AES-NI(Intel Software Network page)

AES-NI は、次の AES 暗号モードを加速します：AES-CBC、AES-GCM、AES-CCM-8、AES-CCM、および AES-CTR。AES-GCM は、Ghash 認証のために Intel チップに追加された 128 ビットの多数関数を使用することにより、さらに加速されます。

4.4.2 STM32F2

wolfSSL は、STM32F2 標準周辺ライブラリを介して STM32F2 ハードウェアベースの暗号化と乱数ジェネレーターを使用できます。

必要な定義については、WOLFSSL_STM32F2 を settings.h に定義してください。WOLFSSL_STM32F2 定義は、デフォルトで STM32F2 ハードウェアクリプトと RNG サポートを有効にします。これらを個別に有効にするための定義は、STM32F2_CRYPTO(ハードウェア暗号サポート用) および STM32F2_RNG(ハードウェア RNG サポート用) です。

STM32F2 標準周辺ライブラリのドキュメントは、次の文書にあります。https://www.stech.com/internet/com/technical_resources/technical_literature/user_manual/dm00023896.pdf

4.4.3 Cavium Nitrox

wolfSSL は Marvell (以前の Cavium) NITROX (https://www.marvell.com/products/security-solutions.html) をサポートしています。wolfSSL のビルド時に Marvell NITROX サポートを有効にするには、次の構成オプションを使用します。

```
./configure --with-cavium=/home/user/cavium/software
```

--with-cavium=**オプションは、ライセンスされた Cavium/Software ディレクトリを指しています。Cavium がライブラリをビルドしないため、wolfSSL は cavium_common.o ファイルを引っ張ります。また、github ソースツリーを使用している場合は、キャビウムヘッダーがこの警告に準拠していないため、生成されたメイクファイルから-Wredundant-decls 警告を削除する必要があります。

現在、wolfSSL は Cavium RNG、AES、3DES、RC4、HMAC、および RSA を暗号層で直接サポートしています。SSL レベルでのサポートは部分的であり、現在、AES、3DES、および RC4 を実行しています。RSA と HMAC は、非ブロッキングモードでキャビウムの呼び出しが利用できるまで遅くなります。クライアントのサンプルプログラムは、暗号テストとベンチマークと同様に、Cavium サポートをオンにします。HAVE_CAVIUM 定義を参照してください。

4.4.4 ESP32-WROOM-32

wolfSSL は、ESP32-WROOM-32 ハードウェアベースの暗号化を使用できます。

必要な定義については、WOLFSSL_ESPWROOM32 を settings.h に定義してください。WOLFSSL_ESPWROOM32 の定義は、デフォルトで ESP32-WROOM-32 ハードウェアクリプトと RNG サポートを有効にします。現在、wolfSSL は、Crypt 層で RNG、AES、SHA、RSA プリミティブをサポートしています。

TLS サーバー/クライアント、WolfCrypt テスト、ベンチマークを含むプロジェクトの例は、ファイルを展開した後、ESP-IDF の/Examples/Protocols Directory で見つけることができます。

4.4.5 ESP8266

ESP32 とは異なり、ESP8266 で使用できるハードウェアベースの暗号化はありません。「user_settings.h」の「WOLFSSL_ESP8266」定義を参照してください。または ./configure CFLAGS="-DWOLFSSL_ESP8266" を使用して、組み込み ESP8266 ターゲット用にコンパイルします。

4.4.6 ERF32

wolfSSL は ERF32 ファミリーのハードウェアベースの暗号化を使用できます。user_settings.h に WOLFSSL_SILABS_SE_ACCEL を定義してください。現在 wolfSSL は、RNG、AES-CBC、AES-GCM、AES-CCM、SHA-1、SHA-2、ECDHE と ECDSA のハードウェアアクセラレーションをサポートしています。さらに詳細な情報とベンチマーク結果は wolfSSL レポジトリツリーの wolfcrypt/src/port/silabs の README.md を参照してください。

4.5 SSL 検査 (Sniffer)

wolfSSL 1.5.0 のリリースから始めて、wolfSSL には、SSL Sniffer(SSL 検査) 機能を持つようにビルドできるビルドオプションが含まれています。これは、SSL トラフィックパケットを収集し、正しいキーファイルを使用して、それらを復号化できることを意味します。SSL トラフィックを「検査」する機能は、いくつかの理由で役立ちます。その一部には以下が含まれます。

- ネットワークの問題の分析
- 内部および外部ユーザーによるネットワークの誤用を検出します
- 動きのネットワークの使用とデータの監視
- クライアント/サーバー通信のデバッグ

Sniffer サポートを有効にするには、*nix で--enable-snifferオプションを使用して wolfSSL をビルドするか、Windows で **vcproj** ファイルを使用します。*nix または **winpcap** に **pcap** を Windows にインストールする必要があります。sniffer.h で見つけることができるメインスニファー機能は、それぞれの簡単な説明とともに以下にリストされています。

- ssl_SetPrivateKey - 特定のサーバーとポートの秘密鍵を設定します。
- ssl_SetNamedPrivateKey - 特定のサーバー、ポート、ドメイン名の秘密鍵を設定します。
- ssl_DecodePacket-デコードのために TCP/IP パケットで通過します。
- ssl_Trace - デバッグトレースを TraceFile に有効/無効にします。
- ssl_InitSniffer - 全体的なスニファーを初期化します。
- ssl_FreeSniffer - 全体的なスニファーを解放します。
- ssl_EnableRecovery-失われたパケットの場合、SSL トラフィックのデコードを取得しようとするオプションを有効にします。
- ssl_GetSessionStats - スニファセッションのメモリ使用量を取得します。

wolfSSL の Sniffer のサポートを確認し、完全な例を参照するには、wolfSSL ダウンロードの sslSniffer/sslSnifferTest フォルダーの snifftest アプリを参照してください。

暗号化キーは SSL ハンドシェイクにセットアップされているため、その後のアプリケーションデータをデコードするためには、スニファーによってハンドシェークをデコードする必要があることに注意してください。たとえば、wolfSSL の echoserver と echoclient で「snifftest」を使用している場合、サーバーとクライアントの間でハンドシェークが始まる前に Snifftest アプリケーションを開始する必要があります。

スニファは、AES-CBC、DES3-CBC、ARC4、および Camellia-CBC で暗号化されたストリームのみをデコードできます。ECDHE または DHE キー合意が使用されている場合、ストリームは監視できません。RSA または ECDH 鍵交換のみがサポートされています。

wolfSSL Sniffer を使用した Callbacks を WOLFSSL_SNIFFER_WATCH でオンにすることができます。SnifferWatch 機能をコンパイルした状態で、関数 ssl_SetWatchKeyCallback() を使用してカスタムコールバックを設定できます。コールバックは、ピアから送信された証明書チェーン、エラー値、および証明書のダイジェストの検査に使用されます。コールバックから非 0 値が返される場合、ピアの証明書を処理するときにエラー状態が設定されます。ウォッチコールバックの追加のサポート機能は次のとおりです。

- ssl_SetWatchKeyCtx：ウォッチコールバックに渡されるカスタムユーザコンテキストを設定します。
- ssl_SetWatchKey_buffer：新しい DER 形式キーをサーバーセッションにロードします。
- ssl_SetWatchKey_file：ssl_SetWatchKey_buffer のファイルバージョン。

スニファを収集する統計は、マクロ WOLFSSL_SNIFFER_STATS を定義することでコンパイルできます。統計は SSLSTATS 構造体に保持され、ssl_ReadStatistics への呼び出しによってアプリケーション SSLSTATS 構造にコピーされます.Sniffer Statistics で使用する追加の API は ssl_ResetStatistics です (統計の収集をリセットします) と ssl_ReadResetStatistics(現在の統計値を読み込み、内部状態をリセットします)。以下は、オンになっているときに保存されている現在の統計です。

- sslStandardConns
- sslClientAuthConns
- sslResumedConns
- sslEphemeralMisses
- sslResumeMisses
- sslCiphersUnsupported
- sslKeysUnmatched
- sslKeyFails
- sslDecodeFails
- sslAlerts
- sslDecryptedBytes
- sslEncryptedBytes
- sslEncryptedPackets
- sslDecryptedPackets
- sslKeyMatches
- sslEncryptedConns

4.6 静的バッファ確保オプション

wolfSSL では**動的**メモリ管理が提供されていることを前提に処理が記述されています。すなわち、malloc/free 関数でバッファを確保/解放できることを前提としています。

wolfSSL が内部で使用している暗号化ライブラリ wolfCrypt では動的メモリを使用しない設定も可能です。これは動的メモリ管理機能を提供していない環境や、あるいは安全面の制約で動的メモリ管理機能の使用が制限されている環境では好ましいと言えます。

一方、組み込み機器では OS を使用しない（いわゆるベアメタル）環境で使用する場合、あるいはリアルタイム OS を使用している場合でも動的メモリ管理機能が提供されていない場合があります。このような環境で wolfSSL を使用する方法として**静的バッファ確保オプション**を提供しています。

4.6.1 静的バッファ確保の基本動作

先に説明した「動的メモリ管理」はバッファを「指定されたサイズ（可変長）」に動的に切り出して提供する管理方法です。バッファの使用効率は高いですが処理が比較的複雑です。一方、wolfSSL が提供する「静的バッファ確保機能」は、あらかじめ（静的に）用意した何種類かのバッファのなかから、要求したサイズに近いものを検索して提供するメモリ管理機能のことを言います。バッファの要求元には要求サイズ以上の大きさを持ったメモリブロックが割り当てられることがあります（そのため使用効率が低下します）が、バッファの確保と解放は malloc/free と同様に行え、割り当て処理は単純ですみます。任意サイズのメモリブロックを動的に確保する動的メモリ確保を模擬した機能となっています。

静的バッファ確保機能の使用は wolfSSL にとっては動的メモリ機能と等価に使用されます。この機能は wolfSSL がメモリの確保/解放を XMALLOC/XFREE 関数呼び出しに抽象化してあることで実現できています。一旦、静的バッファ確保機能が設定されると、以降 wolfSSL は内部で使用するバッファや他の構造体の確保に静的バッファ確保機能を使用します。この機能は WOLFSSL_CTX に対して設定するので、このオブジェクトの生存期間中は機能が継続します。

WOLFSSL_CTX に設定された静的バッファ確保機能はスレッドセーフとなっています。同一の WOLFSSL_CTX を異なるスレッドで共有しながら使用している場合でも、バッファの確保/解放は wolfSSL 内部で排他制御しながら利用します。

また、RTOS で使用されているメモリプールを使ったメモリ機能では未使用のメモリブロックが見つからない場合にはそのスレッド（タスク）は空きブロックが発生するまでサスペンドされますが、wolfSSL の静的バッファ確保機能にはそのような待ち合わせ機能はありません。

4.6.2 静的バッファの用途指定

静的バッファ確保機能では、２つの用途別にメモリを分けることが可能です。つまり、一般的な目的用と I/O 用に使用するバッファを分けて割り当て/解放を行うことが可能です。I/O に使用するバッファは TLS での典型的必要サイズから最大 2 ^ 16 バイトまでを扱えるように比較的大きく（17KB 程度) 設定されていて、それ以外の一般的な用途のバッファサイズと異なる点と、I/O を行うスレッド（タスク）とそれ以外のスレッドでのメモリ獲得/解放に伴う排他制御を排除したいという２つの理由で用途別に分けることを推奨しています。

さらに、バッファを設定した際に、同時に生成できる WOLFSSL オブジェクトの最大数を制限することができます。最大数を制限すると wolfSSL_new() 関数を使う度に生成できる WOLFSSL オブジェクトの数がチェックされ、上限を超えるとエラーとなります。

4.6.3 静的バッファ確保機能の有効化

wolfSSL のビルド時に静的バッファ確保オプションを有効化します。autoconf ツールを使用してビルドするシステムでは次のように”–enable-staticmemory“を指定します。

```
$./configure --enable-staticmemory
```

あるいは user_settings.h を使用している場合には次のマクロ定義を追加します：

user_settings.h

```
#define WOLFSSL_STATIC_MEMORY
```

さらに、静的バッファ確保機能は与えられたバッファから確保するメモリブロックが枯渇した際に、NULL を返さず標準関数の malloc() を追加で呼び出す実装となっています。もし、環境に動的メモリ管理機能が

提供されていない場合にはリンクエラーとなります。したがって、この機能をディセーブルにする為に、**WOLFSSL_NO_MALLOC** マクロも定義しておきます：

user_settings.h

```
    #define WOLFSSL_STATIC_MEMORY
    #define WOLFSSL_NO_MALLOC
```

4.6.4 静的バッファ確保機能の利用方法

上記設定で wolfSSL をビルドすることにより、静的バッファ確保機能が有効となります。

4.6.4.1 静的バッファの設定関数とその引数 さらにこの機能を利用するにあたってはアプリケーションは次の関数を呼び出してヒープとして使用するバッファを指定します：

```
int wolfSSL_CTX_load_static_memory(
        WOLFSSL_CTX** ctx,          /* 生成したWOLFSSL_CTXを受け取る為の変数へ
            のポインタ */
        wolfSSL_Methos_func method,/* メソッド */
        unsigned char* buf,         /* ヒープとして使用するバッファへのポインタ
            */
        unsigned int sz,            /* ヒープとして使用するバッファのサイズ */
        int flag,                   /* ヒープの使用用途 */
        int max);                   /* 許可する最大平行動作数 */
```

- 引数 **ctx** には生成された WOLFSSL_CTX 構造体へのポインタを受け取る変数のアドレスを指定します。
- 引数 **method** には wolfSSLv23_client_method_**ex**() などの”_ex” が付いた関数の戻り値を指定します。
- 引数 **buf**,**sz** にはそれぞれヒープに使用するバッファのアドレスとそのサイズを指定します。設定すべきバッファのサイズの決定については「必要バッファのサイズの取得」を参照してください。
- 引数 **flag** には一般用途あるいは I/O 用を指定する用途と静的バッファの確保状況のトラッキングを行うかどうかのフラグを組み合わせて指定します。一般用途を指定する場合には”0” あるいは WOLFMEM_GENERAL を指定します。I/O 用としての指定は WOLFMEM_IO_POO あるいは WOLFMEM_IO_POOL_FIXED を指定します。静的バッファの確保状況のトラッキングを行う場合には用途を指定する値に WOLFMEM_TRACK_STATS を OR して指定します。
- 引数 **max** は引数 flag で指定したバッファの用途に関係します。バッファの用途が一般用の場合には、生成する WOLFSSL オブジェクトの最大同時生成数（同時に存在できるオブジェクト数）を設定することになります。制限を行う必要がなければ 0 を指定します。0 以外の制限値を指定した場合には、その後の wolfSSL_new の呼び出しで生成する WOLFSSL オブジェクトの同時オブジェクト数が設定値を超える際には生成が失敗することになります。

4.6.4.2 静的バッファの設定関数の呼び出し方 静的バッファ確保機能を利用する際には、この **wolfSSL_CTX_load_static_memory 関数を 2 回呼び出します。最初は一般用途用にバッファを設定し、さらにそのバッファを使って WOLFSSL_CTX 構造体を確保します。2 回目の呼び出しでは、I/O 用バッファを設定します**:

```
WOLFSSL_CTX* ctx = NULL; /* WOLFSSL_CTXを生成する場合にはNULLを指定 */
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];

    /* 最初の呼び出しで一般用静的バッファを設定してそこからWOLFSSL_CTXを生成す
        る */
    ret = wolfSSL_CTX_load_static_memory(
            &ctx,                           /* ctx変数の内容にはNULLをセットして
                おく */
            wolfSSLv23_client_method_ex(), /* "_ex"の付いた関数を使う */
            GEN_MEM, GEN_MEM_SIZE,          /* 一般用途のバッファとそのサイズ */
            WOLFMEM_GENERAL,                /* 一般用途で使用 */
            MAX_CONCURRENT_TLS);            /* 最大許容同時接続数 */

    /* 次は生成したWOLFSSL_CTXにI/O用静的バッファをセットする */
    ret = wolfSSL_CTX_load_static_memory(
            &ctx,                           /* ctxには生成済み構造体が格納されて
                いる */
            NULL,                           /* 今度はctxの生成は行わないのでNULL
                をセット */
            IO_MEM, IO_MEM_SIZE,            /* I/O用途のバッファとそのサイズ */
            WOLFMEM_IO_FIXED,
            MAX_CONCURRENT_IO);
```

この後、WOLFSSL_CTX 構造体の使用を終了した後は、通常の wolfSSL_CTX_free() を使って解放してください。

4.6.5 静的バッファ確保機能の調整

wolfSSL で提供しているこの静的バッファ確保機能では指定されたバッファを次の図の様に複数の” バケット（bucket)” という領域に分けて管理します。バケット内には同一サイズの複数のメモリブロックがリンクされています。下図の中にはメモリブロックの管理のための構造は省略していますが、実際にはそれらの管理領域も含めて必要なサイズを持つバッファが与えられなければなりません。

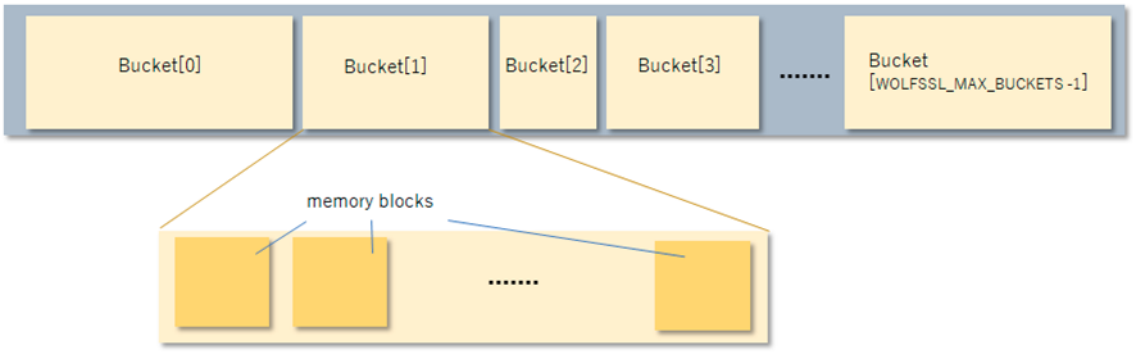

Figure 2: バケット

上記構造は一般用、I/O 用のいずれのバッファにも適用されますが、I/O 用バッファには Bucket は一種類しかありません。

4.6.5.1 一般用バッファの設定用マクロ定義 各バケットはバケット内に含むメモリブロックの数とそのサイズに応じて大きさが異なります。

使用する各領域のメモリブロックサイズとブロックの個数が/wolfssl/wolfcrypt/memory.h に次のようなマクロで定義してあります：

/wolfssl/wolfcrypt/memory.h

```
#define WOLFSSL_STATIC_ALIGN 16      /* 適用されるアライメント（デフォルト
    16バイト）*/
#define WOLFMEM_MAX_BUCKETS  9        /* バケット数 */
#define LARGEST_MEM_BUCKET   16128   /* 最大ブロックのサイズ */
#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** はバッファのアライメントサイズを指定します。デフォルトで 16 バイトです。御使用の MCU でのアライメントサイズに合わせて変更する必要があります。
- **WOLFMEM_MAX_BUCKETS** がバケットの数を示しています。9 種類のサイズのバケットを使用することを意味しています。
- **WOLFMEM_BUCKETS** が各バケット内のブロックのバイト数を小さいものから順にコンマ区切りで指定しています。この定義は一般用バッファに適用されます。
- **WOLFMEM_DIST** が各バケットに含まれる同一サイズのブロックの数を WOLFMEM_BUCKETS の各ブロックに対応するようにそれらの個数をコンマ区切りで指定しています。この定義は一般用バッファに適用されます。

上記の例でいえば、ブロックサイズ 64 バイトのバケットが最小のサイズであり、そのバケットには 49 個のメモリブロックを用意することになります。次に大きいバケットはブロックサイズ 128 バイトで 10 個のメモリブロックを用意することを意味します。

上記定義値はデフォルトの値として使用していただけますが、実際の環境での使用時には各バケットのサイズとそこに含まれるメモリブロック数は調整が必要かもしれません。

4.6.5.2 I/O 用バッファの設定用マクロ定義 I/O 用途のバッファは上記一般用途と管理方法は同じですが、バケット数は” 1”、バケット内のメモリブロックは１つだけです。またメモリブロックのサイズは **WOLFMEM_IO_SZ** で定義された値となっています。

この I/O バッファのサイズは TLS ハンドシェークで送受信される最大パケットサイズを考慮して設定されていますが、この最大パケットサイズを **wolfSSL_CTX_UseMaxFragment**() を使ってより小さい値に設定することが可能です。この関数を使って最大パケットサイズを小さくした場合には、その値（この例では 660 バイト）を WOLFMEM_IO_SZ として設定してください。

```
$ ./configure --enable-staticmemory C_EXTRA_FLAGS="-DWOLFMEM_IO_SZ=660"
```

4.6.5.3 必要バッファサイズの取得 静的バッファ確保機能に割り当てるバッファサイズ（すなわち wolfSSL_CTX_load_static_memory 関数に渡すバッファサイズ）の決定に有用な関数を用意してあります。この章の冒頭でバッファが内部でバケットと管理領域からなる構造に構成されて使用されることを説明しました。実際にバッファのサイズを決定する際には、メモリブロックのサイズと管理領域の占めるサイズとパディングによる余分に必要なサイズも含めて計算する必要があります。この計算を行い、必要なバッファサイズを返してくれる関数を用意してあります。

wolfSSL_StaticBufferSz 関数は、直前のセクションで紹介したマクロ定義値を基にに必要なバッファサイズを計算して返します。この関数が 0 より大きい値を返すまで、第 2 引数に与えるサイズは 1000 などの適当な値からスタートし、戻り値が正となるまでサイズを増やして何度も呼び出します。第 3 引数の flag には一般用バッファサイズを計算する場合には **WOLFMEM_GENERAL** を与え、I/O 用バッファサイズを計算する場合には **WOLFMEM_IO_FIXED** あるいは **WOLFMEM_IO_POOL** を指定してください。

```
int wolfSSL_StaticBufferSz(byte* buffer, /* バッファアドレス */
                           word32 sz,    /* バッファサイズ  */
                           int flag);    /* バッファの用途 */
```

この関数を使って一般用と､I/O 用に必要なバッファサイズを取得し､前述の wolfSSL_CTX_load_static_memory 関数に渡してください。

一旦バッファサイズが決定したら、上記 wolfSSL_StaticBufferSz 関数は呼び出す必要はありませんので、製品コードから呼び出し部分をコメントアウトするか削除していただけます。

4.6.6 静的バッファ利用状況のトラッキング

静的バッファ確保機能を利用する際に、バッファの確保、解放に関する使用状況を記録させることができます。この使用状況を記録する機能は静的バッファ確保機能にデフォルトで含まれていますからビルド時の有効化として別段のマクロ設定は必要ありません。機能の有効化は実行時に行います。

4.6.6.1 トラッキングの有効化 トラッキング機能の有効化は先に説明した wolfSSL_CTX_load_static_memory 関数の第 5 引数に **WOLFMEM_TRACK_STATS** を OR して指定します。wolfSSL 内部に **WOLFSSL_MEM_CONN_STATS 構造体**が確保されそこにメモリブロックの使用状況が記録されていきます。

wolfssl/wolfcrypt/memory.h

```
struct WOLFSSL_MEM_CONN_STATS {
        word32 peakMem;   /* メモリ使用量最大値（バイト数）   */
        word32 curMem;    /* 現在のメモリ使用量 */
        word32 peakAlloc; /* メモリ確保量最大値 */
        word32 curAlloc;  /* 現在のメモリ確保数 */
        word32 totalAlloc;/* 累計メモリ確保回数 */
        word32 totalFr;   /* 累計メモリ解放回数 */
};
```

4.6.6.2 メモリ使用状況の取得 トラッキングが有効になった時点からメモリブロックの使用状況は記録が有効となります。プログラム実行の任意の時点で次の関数を呼び出して、引数で渡した WOLFSSL_CTX あるいは WOLFSSL オブジェクトに静的バッファ確保機能が使用されているか否かを戻り値で返します。使用されている場合には戻り値として”1″を返します。さらに記録されているメモリの使用状況を取得することができます。

```
int wolfSSL_CTX_is_static_memory(WOLFSSL_CTX* ctx,
                                 WOLFSSL_MEM_STATS* mem_stats);

int wolfSSL_is_static_memory(WOLFSSL* ssl,
                             WOLFSSL_MEM_STATS* mem_stats);
```

上記関数の引数として渡した WOLFSSL_MEM_STATS 構造体は：

```
struct WOLFSSL_MEM_STATS {
        word32 curAlloc;  /* 現在のメモリ確保数 */
        word32 totalAlloc;/* 累計メモリ確保回数 */
        word32 totalFr;   /* 累計メモリ解放回数 */
        word32 totalUse;  /* N/A */
        word32 avaIO;     /* I/O用ブロックの空きブロック数 */
        word32 maxHa;     /* 一般用に設定した最大コネクション数 */
        word32 maxIO;     /* I/O用に設定した最大コネクション数 */
        word32 blockSz[WOLFMEM_MAX_BUCKETS]; /* ブロックサイズの配列 */
        word32 avaBlock[WOLFMEM_MAX_BUCKETS];/* 空きブロック数の配列 */
        word32 usedBlock[WOLFMEM_MAX_BUCKETS];
        int    flag;     /* 静的メモリ管理機能に設定したフラグ（バッファ用途
            等） */
```

```
};
```

この構造体に返却された値が呼び出した時点の静的バッファ管理状態を示します。この機能はメモリーリークの検出や無駄に多く割り当てたメモリブロック数の調査などに利用できます。

4.6.7 静的バッファ管理 API

ここで紹介する関数は wolfSSL_CTX_load_static_memory 関数の第 2 引数に指定する WOLFSSL_METHOD 構造体へのポインタを取得する為に使用する関数です。こららの関数はアプリケーションを TLS あるいは DTLS プロトコルのクライアントあるいはサーバーのいずれとして動作させるのかによって選択すべき関数が異なります。静的バッファ管理機能を使う際には必ず関数名の最後に"_ex" が付加された以下の関数を指定してください。こららの関数は"_ex" がつかない関数と機能は同じで、内部で WOLFSSL_METHOD 構造体を有効になった静的バッファ管理機能を使って確保する点だけが異なります。各関数の詳細は wolfSSL マニュアルの"wolfSSL API リファレンス >wolfSSL コンテクストの設定" を参照してください。

4.6.7.1 WOLFSSL_METHOD 構造体を返す関数群 TLS クライアント用関数

- wolfTLSv1_3_client_method_ex
- wolfTLSv1_2_client_method_ex
- wolfTLSv1_1_client_method_ex
- wolfSSLv23_client_method_ex

TLS サーバ用関数

- wolfTLSv1_3_server_method_ex
- wolfTLSv1_2_server_method_ex
- wolfTLSv1_1_server_method_ex
- wolfSSLv23_server_method_ex

DTLS クライアント用関数

- wolfDTLSv1_3_client_method_ex
- wolfTLSv1_2_client_method_ex
- wolfTLSv1_1_client_method_ex
- wolfSSLv23_client_method_ex

DTLS サーバ用関数

- wolfDTLSv1_3_server_method_ex
- wolfTLSv1_2_server_method_ex
- wolfTLSv1_1_server_method_ex
- wolfSSLv23_server_method_ex

4.6.7.2 静的バッファ管理機能 API API リファレンス

関数	機能
wolfSSL_CTX_load_static_memory	WOLFSSL_CTX に静的バッファ管理用のバッファを設定します。
wolfSSL_CTX_is_static_memory	静的バッファ管理が設定されているかと設定されている場合にはその使用状況を
wolfSSL_is_static_memory	静的バッファ管理が設定されているかと設定されている場合にはその使用状況を
wolfSSL_StaticBufferSz	静的バッファ管理で必要なバッファサイズを計算します。

4.7 圧縮

wolfSSL は、**zlib** ライブラリとのデータ圧縮をサポートしています。./configure ビルドシステムはこのライブラリの存在を検出しますが、他の方法でビルドしている場合は、定数 HAVE_LIBZ を定義し、Zlib.h へのパスを含めます。

特定の暗号ではデフォルトで圧縮がオフになっています。オンにするには、SSL が接続または受け入れる前に、関数wolfSSL_set_compression()を使用します。クライアントとサーバーの両方が、圧縮を使用するために圧縮をオンにする必要があります。

送信する前にデータを圧縮すると、送信されて受信されるメッセージの実際のサイズが減少しますが、圧縮によって保存されたデータの量は通常、最も遅いネットワークを除くすべてのもので生で送信するよりも分析に時間がかかることに注意してください。

4.8 事前共有鍵

wolfSSL は、静的事前共有キーを使用してこれらの暗号をサポートしています。

- 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

これらのスイートは、WOLFSSL_STATIC_PSK オンの wolfSSL に組み込まれています。すべての PSK スイートは、一定の NO_PSK でビルド時にオフにできます。これらの暗号を実行時にのみ使用するには、wolfSSL_CTX_set_cipher_list() を目的の暗号スイートとともに使用します。

wolfSSL は、Ephemeral Key PSK スイートをサポートしています。

- 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

クライアントでは、関数wolfSSL_CTX_set_psk_client_callback()を使用してコールバックをセットアップします。<wolfSSL_Home>/examples/client/client.c のクライアントの例は、クライアント

の ID とキーをセットアップするための使用例を示していますが、実際のコールバックは wolfssl/test.h に実装されています。

サーバー側では、2 つの追加のコールが必要です。

- wolfSSL_CTX_set_psk_server_callback()
- wolfSSL_CTX_use_psk_identity_hint()

サーバーは、「wolfSSL Server」のサーバーの例で、2 回目の呼び出しを使用してクライアントを支援するために ID ヒントを格納します。Server PSK コールバックの例は、wolfssl/test.h で my_psk_server_cb() にあります。

wolfSSL は、最大 128 オクテットの ID とヒント、および最大 64 オクテットの事前共有鍵をサポートします。

4.9 クライアント認証

クライアント認証は、クライアントが接続時に認証のためにサーバーに証明書を送信するように要求することによって、サーバーがクライアントを認証できるようにする機能です。クライアント認証には、CA からの X.509 クライアント証明書 (または、あなたまたは CA 以外の誰かによって生成された場合は自己署名) が必要です。

デフォルトでは、wolfSSL は受信したすべての証明書を検証します - これにはクライアントとサーバーの両方が含まれます。クライアント認証をセットアップするには、サーバーは、クライアント証明書を次のように確認するために使用される信頼できる CA 証明書のリストをロードする必要があります。

```
wolfSSL_CTX_load_verify_locations(ctx, caCert, 0);
```

クライアントの検証をオンにし、その動作を制御するために、wolfSSL_CTX_set_verify()のその他のオプションには、SSL_VERIFY_NONE および SSL_VERIFY_CLIENT_ONCE が含まれます。

```
wolfSSL_CTX_set_verify(ctx,SSL_VERIFY_PEER | ((usePskPlus)?
                        SSL_VERIFY_FAIL_EXCEPT_PSK :
                        SSL_VERIFY_FAIL_IF_NO_PEER_CERT),0);
```

クライアント認証の例は、wolfssl ダウンロード (/examples/server/server.c) に含まれるサンプルサーバー (server.c) にあります。

4.10 サーバー名の提示

SNI は、サーバーが単一の基礎となるネットワークアドレスで複数の「仮想」サーバーをホストする場合に役立ちます。クライアントが連絡しているサーバーの名前を提供することが望ましい場合があります。WolfsSL で SNI を有効にするには、簡単に実行できます。

```
./configure --enable-sni
```

クライアント側に SNI を使用するには、追加の関数呼び出しが必要です。これは、次の機能の 1 つである必要があります。

- wolfSSL_CTX_UseSNI()
- wolfSSL_UseSNI()

クライアントが同じサーバーに複数回連絡すると、wolfSSL_CTX_UseSNI()が最も推奨されます。コンテキストレベルでの SNI 拡張子を設定すると、呼び出しの瞬間から同じコンテキストから作成されたすべての SSL オブジェクトの SNI 使用法が可能になります。

wolfSSL_UseSNI()は 1 つの SSL オブジェクトのみの SNI 使用を有効にするため、サーバー名がセッション間で変更されたときにこの関数を使用することをお勧めします。

サーバー側では、同じ関数呼び出しの 1 つが必要です。wolfSSL Server は複数の「仮想」サーバーをホストしていないため、SNI の不一致の場合に接続の終了が必要な場合に SNI の使用が役立ちます。このシナリオでは、wolfSSL_CTX_UseSNI()がより効率的になります。サーバーは、同じコンテキストから SNI を使用して後続のすべての SSL オブジェクトを作成するコンテキストごとに 1 回しか設定しないためです。

4.11 ハンドシェイクの変更

4.11.1 ハンドシェイクメッセージのグループ化

wolfSSL には、ユーザーが望む場合、ハンドシェイクメッセージをグループ化する機能があります。これは、wolfSSL_CTX_set_group_messages(ctx);の SSL オブジェクトレベルで実行できます。

4.12 短縮された HMAC.

現在定義されている TLS 暗号スイートは、HMAC を使用してレコード層通信を認証します。TLS では、ハッシュ関数の出力全体が Mac タグとして使用されます。ただし、MAC タグを形成するときにハッシュ関数の出力を 80 ビットに切り捨てることにより、帯域幅を節約することは、制約付き環境で望ましい場合があります。wolfSSL で切り捨てられた HMAC の使用を可能にするには、簡単にできることを説明できます。

```
./configure --enable-truncatedhmac
```

クライアント側に切り捨てられた HMAC を使用するには、追加の関数呼び出しが必要です。これは、次の機能の 1 つである必要があります。

- wolfSSL_CTX_UseTruncatedHMAC()
- wolfSSL_UseTruncatedHMAC()

クライアントがすべてのセッションに対して切り捨てられた HMAC を有効にしたい場合に最も推奨されます。コンテキストレベルでの切り捨てられた HMAC 拡張子を設定すると、呼び出しの瞬間と同じコンテキストから作成されたすべての SSL オブジェクトでそれを有効にします。

wolfSSL_UseTruncatedHMAC()は 1 つの SSL オブジェクトのみを有効にします。そのため、すべてのセッションで Truncated HMAC を必要としない場合はこの機能を使用することをお勧めします。

サーバー側では、通話は不要です。サーバーは、クライアントの切り捨てられた HMAC の要求に自動的に注意を払っています。

すべての TLS 拡張機能を有効にすることもできます。

```
./configure --enable-tlsx
```

4.13 ユーザー暗号モジュール

ユーザー Crypto モジュールを使用すると、ユーザーがサポートされている操作中に使用したいカスタム暗号をプラグインできます (現在 RSA 操作がサポートされています)。モジュールの例は、IPP ライブラリを使用してディレクトリ root_wolfssl/wolfcrypt/user-crypto/にあります。Crypto モジュールを使用するために wolfSSL をビルドするときの構成オプションの例は次のとおりです。

```
./configure --with-user-crypto
```

また

```
./configure --with-user-crypto=/dir/to
```

RSA 操作を実行するユーザー暗号モジュールを作成するときは、user_rsa.h と呼ばれる RSA 用のヘッダーファイルがあることが必須です。すべてのユーザー暗号操作では、ユーザーライブラリが libusercrypto と呼ばれます。これらは wolfSSL AutoConf ツールの名前です。ユーザー暗号モジュールをリンクして使用するときに探してください。wolfSSL を備えた例では、ヘッダファイル user_rsa.h

はディレクトリ wolfcrypt/user-crypto/include/にあり、作成されたライブラリはディレクトリ wolfcrypt/user-crypto/lib/にあります。提供されているヘッダーファイルを参照してください。

例を作成するには、IPP ライブラリをインストールした後、ルート wolfSSL ディレクトリから次のコマンドを実行する必要があります。

```
cd wolfcrypt/user-crypto/
./autogen.sh
./configure
make
sudo make install
```

wolfSSL の添付の例では、プロジェクトをビルドする前にインストールする必要がある IPP の使用が必要です。ただし、例をビルドするための IPP ライブラリを持っていなくても、ファイル名選択と API インターフェイスの例をユーザーに提供することを目的としています。ライブラリ libusercrypto とヘッダーファイルの両方を作成してインストールしたら、wolfSSL を使用すると、Crypto モジュールは追加のステップを必要としません。Configure Flag --with-user-cryptoを使用するだけで、一般的な wolfSSL 暗号からのすべての関数呼び出しをユーザー暗号モジュールにマッピングします。

メモリの割り当ては、wolfSSL の XMALLOC を使用している場合は、DYNAMIC_TYPE_USER_CRYPTO でタグ付けする必要があります。モジュールで使用されるメモリ割り当てを分析できます。

ユーザーの暗号モジュールは、wolfssl の設定オプションの速い RSA および/または FIPS と組み合わせて使用することは**できません**。FIPS には、特定の認証コードを使用し、FAST-RSA を使用して RSA 操作を実行するために例ユーザー暗号モジュールを使用します。

4.14 Wolfssl のタイミング耐性

wolfSSL は、潜在的にリークタイミング情報を漏らす可能性のある比較操作を行うときに一定の時間を保証する関数 “ConstantCompare” を提供します。この API は、タイミングベースのサイドチャネル攻撃を阻止するために、wolfSSL の TLS レベルと暗号レベルの両方で使用されます。

wolfSSL ECC 実装には､ECC アルゴリズムのタイミング耐性を有効にするために､ECC_TIMING_RESISTANT を定義しています。同様に、定義 TFM_TIMING_RESISTANT は、RSA アルゴリズムのタイミング耐性の Fast Math ライブラリに提供されます。関数 exptmod は、タイミング耐性のモンゴメリーラダーを使用します。

参照：--disable-harden

処理時間一定化とキャッシュ抵抗は、--enable-harden で有効になっています。

- WOLFSSL_SP_CACHE_RESISTANT：使用するアドレスをマスクするロジックを有効にします。
- WC_RSA_BLINDING：タイミング攻撃を防ぐために、ブラインドモードを有効にします。
- ECC_TIMING_RESISTANT：ECC 固有の処理時間一定化。
- TFM_TIMING_RESISTANT：Fast Math 固有の処理時間一定化。

4.15 固定化された ABI

wolfSSL は、アプリケーションプログラミングインターフェイス (API) のサブセットに固定アプリケーションバイナリインターフェイス (ABI) を提供します。次の機能は、wolfSSL v4.3.0 以降、wolfSSL の将来のすべてのリリースにわたって互換性があります。

- 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 ポータビリティ

5.1 抽象化レイヤー

5.1.1 C 標準ライブラリ抽象化レイヤー

wolfSSL(以前の Cyassl) は、C 標準ライブラリなしでビルドして、開発者により高いレベルのポータビリティと柔軟性を提供することができます。ユーザーは、C 標準の関数の代わりに使用したい関数をマッピングする必要があります。

5.1.1.1 メモリ使用 ほとんどの C プログラムは、動的メモリ割り当てに malloc() および free() を使用します。wolfSSL は、代わりに XMALLOC() および XFREE() を使用します。デフォルトでは、これらは C ランタイムバージョンを指します。XMALLOC_USER を定義することにより、ユーザーは独自のフックを提供できます。各メモリ関数は、標準的なものについて 2 つの追加の引数、ヒープのヒント、および割り当てタイプを使用します。ユーザーは、これらを無視するか、好きな方法で使用できます。wolfSSL メモリ関数は wolfssl/wolfcrypt/types.h で見つけることができます。

wolfSSL は、コンパイル時ではなく実行時にメモリオーバーライド関数を登録する機能も提供します。wolfssl/wolfcrypt/memory.h はこの機能のヘッダーであり、ユーザーは次の関数を呼び出してメモリ関数をセットアップできます。

```
int wolfSSL_SetAllocators(wolfSSL_Malloc_cb  malloc_function,
                          wolfSSL_Free_cb    free_function,
                          wolfSSL_Realloc_cb realloc_function);
```

コールバックプロトタイプについては、ヘッダー wolfssl/wolfcrypt/memory.h、実装については memory.c を参照してください。

5.1.1.2 string.h wolfSSL は、string.h の memcpy()、memset()、および memcmp() のように振る舞ういくつかの機能を使用します。それらはそれぞれ XMEMCPY()、XMEMSET()、および XMEMCMP() に抽象化されています。デフォルトでは、C 標準ライブラリバージョンを指します。STRING_USER を定義することで、ユーザーは types.h で独自のフックを提供できます。たとえば、デフォルトでは XMEMCPY() は次のとおりです。

```
#define XMEMCPY(d,s,l)    memcpy((d),(s),(l))
```

STRING_USER を定義した後は、次のことができます。

```
#define XMEMCPY(d,s,l)    my_memcpy((d),(s),(l))
```

またはマクロを避けたい場合：

```
external void* my_memcpy(void* d, const void* s, size_t n);
```

あなたのバージョン my_memcpy() を指すように wolfssl の抽象化レイヤーを設定する。

5.1.1.3 Math.H wolfSSL は、math.h の pow() log() のように振る舞う 2 つの機能を使用しています。Difie-Hellman のみが必要とするため、ビルドから DH を除外すると、独自の DH を提供する必要はありません。それらは XPOW() および XLOG() として wolfcrypt/src/dh.c に定義されます。

5.1.1.4 ファイルシステムの使用 デフォルトでは、wolfSSL はキーと証明書をロードするためにシステムのファイルシステムを使用します。NO_FILESYSTEM を定義することでオフにすることができます。代わりに項目 V を参照してください。システムがて提供するファイル システムと異なるものを使用したい場合は、ssl.c の XFILE() レイヤーを使用して、ファイル システム呼び出しを使用したいシステムに向けることができます。Micrium Define によって提供される例を参照してください。

5.1.2 カスタム入力/出力抽象化レイヤー

wolfSSL は、SSL 接続の I/O をより高く制御したい、または TCP/IP 以外の異なるトランスポートメディアの上に SSL を実行したい方のためのカスタム I/O 抽象化レイヤを提供します。

ユーザーは 2 つの機能を定義する必要があります。

1. ネットワーク送信機能
2. ネットワーク受信機能

これらの 2 つの関数は、ssl.h の CallbackIOSend および CallbackIORecv によってプロトタイプ化されています。

```
typedef int (*CallbackIORecv)(WOLFSSL *ssl, char *buf, int sz, void *ctx);
typedef int (*CallbackIOSend)(WOLFSSL *ssl, char *buf, int sz, void *ctx);
```

ユーザーは WOLFSSL_CTX ごとに wolfSSL_SetIOSend() および wolfSSL_SetIORecv() に登録する必要があります。たとえば、デフォルトの場合は、CBIORecv() と CBIOSend() は io.c の下部に登録されています。

```
void wolfSSL_SetIORecv(WOLFSSL_CTX *ctx, CallbackIORecv CBIORecv)
{
     ctx->CBIORecv=CBIORecv;
}

void wolfSSL_SetIOSend(WOLFSSL_CTX *ctx, CallbackIOSend CBIOSend)
{
    ctx->CBIOSend=CBIOSend;
}
```

ユーザは、io.c の下部に示されているように、wolfSSL_SetIOWriteCtx() および wolfSSL_SetIOReadCtx() で wolfSSL オブジェクト (セッション) ごとにコンテキストを設定することができる。例えば、ユーザがメモリバッファを使用している場合、コンテキストはどこで説明を説明する構造へのポインタであり得る。メモリバッファにアクセスします。デフォルトの場合は、ユーザーが上書きしないで、ソケットをコンテキストとして登録します。

CBIORecv および CBIOSend 関数ポインターは、カスタム I/O 関数を指すことができます。io.c にあるデフォルトの Send() および Receive() 関数、EmbedSend() および EmbedReceive() は、テンプレートとガイドとして使用できます。

WOLFSSL_USER_IO は、デフォルトの I/O 関数 EmbedSend() および EmbedReceive() の自動設定を削除するために定義できます。

5.1.3 オペレーティングシステムの抽象化レイヤー

wolfSSL OS 抽象化レイヤーは、ユーザーのオペレーティングシステムへの wolfSSL の簡単な移植を容易にするのに役立ちます。wolfssl/wolfcrypt/settings.h ファイルには、OS レイヤーをトリガーする設定が含まれています。

OS 特有の定義は、WolfCrypt および Wolfssl の wolfssl/internal.h の wolfssl/wolfcrypt/types.h にあります。

5.2 サポートされているオペレーティングシステム

wolfSSL を定義する 1 つの要因は、新しいプラットフォームに簡単に移植される能力です。そのため、wolfssl は、out-of-box のオペレーティングシステムの多くをサポートしています。現在サポートされているオペレーティングシステムは次のとおりです。

- 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 と 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 サポートされたチップメーカー

wolfSSL は、ARM、Intel、Motorola、MBED、Freescale、Microchip(PIC32)、STMicro(STM32F2/F4)、NXP、Analog Devices、Texas Instruments、AMD などを含むチップセットをサポートしています。

5.4 C ＃ラッパー

wolfSSL は、制限付きですが C ＃での使用をサポートしています。ポートを含むビジュアルスタジオプロジェクトは、ディレクトリ `root_wolfSSL/wrapper/CSharp/`にあります。ビジュアルスタジオプロジェクトを開いた後、ビルド -> 構成マネージャーをクリックして「アクティブソリューション構成」と「アクティブソリューションプラットフォーム」を設定します。」は DLL デバッグと DLL リリースです。サポートされているプラットフォームは Win32 および X64 です。

ソリューションとプラットフォームを設定したら、プリプロセッサフラグ `HAVE_CSHARP` を追加する必要があります。これにより、C ＃ラッパーで使用され、サンプルプログラムで使用されるオプションがオンになります。

その後、ビルドするだけでビルドソリューションを選択します。これにより、`wolfssl.dll`、`wolfSSL_CSharp.dll` および例が作成されます。サンプルプログラムは、それらをエントリポイントとしてターゲットにして、Visual Studio でデバッグを実行することで実行できます。

作成された C ＃ラッパーを C ＃プロジェクトに追加することは、いくつかの方法で実行できます。1 つの方法は、作成された `wolfssl.dll` および `wolfSSL_CSharp.dll` をディレクトリ `C:/Windows/System/`にインストールすることです。これにより、Wolfssl C ＃ラッパーの呼び出しが可能になります。

```
using wolfSSL.CSharp

public some_class {

    public static main(){
    wolfssl.Init()
    ...
}
...
```

Wolfssl C ＃ラッパーに電話をかける。別の方法は、Visual Studio プロジェクトを作成し、wolfSSL のバンドル C ＃ラッパーソリューションを参照することです。

6 コールバック

6.1 ハンドシェイクコールバック

wolfSSL(以前の Cyassl) には、ハンドシェイクコールバックが接続または待受に設定できるようにする拡張子があります。これは、別のデバッガが利用できず、スニッフィングが実用的でない場合に、組み込みシステムでのデバックに役立ちます。wolfSSL ハンドシェイクコールバックを使用するには、拡張機能、wolfSSL_connect_ex()およびwolfSSL_accept_ex()を使用します。

```
int wolfSSL_connect_ex(WOLFSSL*, HandShakeCallBack, TimeoutCallBack,
                       Timeval)
int wolfSSL_accept_ex(WOLFSSL*, HandShakeCallBack, TimeoutCallBack,
                      Timeval)
```

HandShakeCallBack は次のように定義されています。

```
typedef int (*HandShakeCallBack)(HandShakeInfo*);
```

HandShakeInfo は wolfssl/callbacks.h で定義されています (標準以外のビルドに追加する必要があります)：

```
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;
```

ハンドシェーク中の SSL パケットの最大数がわかっているため、動的メモリは使用されません。パケット名には、packetNames[idx] から numberPackets までアクセスできます。コールバックは、ハンドシェイクエラーが発生したかどうかを呼び出します。使用法の例がクライアントのサンプルプログラムにあります。

6.2 タイムアウトコールバック

wolfSSL ハンドシェイクコールバックで使用されているのと同じ拡張機能を wolfSSL タイムアウトコールバックにも使用できます。これらの拡張機能は、コールバック (Handshake および/または Timeout) のいずれか、両方、またはどちらも使用せずに呼び出すことができます。TimeoutCallback は次のように定義されています。

```
typedef int (*TimeoutCallBack)(TimeoutInfo*);
```

TimeoutInfo は次のようになります。

```
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;
```

やはり、最大数の SSL パケットがハンドシェイクでわかっているので、この構造に動的メモリは使用されません。Timeval は、struct timeval の typedef です。

PacketInfo は次のように定義されています。

```
typedef struct packetInfo_st {
    char           packetName[MAX_PACKETNAME_SZ + 1]; /* SSL name */
    Timeval        timestamp;                  /* when it occured */
    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;
```

ここでは、動的メモリを用いてもよい。SSL パケットが value に収まることができる場合、それが配置されている場所です。valueSz は長さと bufferValue を保持します。パケットが value で大きすぎる場合は、**証明書**パケットのみがこれを引き起こすはずです.valueSz には SIZE を保持します。

証明書パケットにメモリが割り当てられている場合は、コールバックが返された後に回収されます。タイムアウトはシグナル、具体的には SIGALRM を使用して実装され、スレッドセーフです。以前のアラームがタイプ ITIMER_REAL のセットである場合、その後、正しいハンドラーとともにリセットされます。古いタイマーは、wolfSSL が処理に費やす時間に合わせて調整されます。既存のタイマーが渡されたタイマーよりも短い場合、既存のタイマー値が使用されます。その後はまだリセットされます。期限切れになる既存のタイマーは、それに関連付けられた間隔がある場合、リセットされます。コールバックは、タイムアウトが発生した場合にのみ発行されます。

使用法についてはクライアントの例を参照してください。

6.3 ユーザーアトミックレコードレイヤー処理

wolfSSL は、SSL/TLS 接続中に MAC/暗号化、復号化/検証などの機能をより制御したユーザーにアトミックレコード処理コールバックを提供します。

ユーザーは 2 つの関数を定義する必要があります。

1. Mac/暗号化コールバック関数
2. 復号化/検証コールバック関数

これらの 2 つの関数は、ssl.h の CallbackMacEncrypt および CallbackDecryptVerify によってプロトタイプ化されています。

```
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);
```

ユーザーは、wolfSSL_CTX_SetMacEncryptCb()および[wolfSSL_CTX_SetDecryptVerifyCb()](#function-wolfssl_ctx_setdecryptverifycb() で wolfSSL コンテキスト (WOLFSSL_CTX) ごとにこれらの機能を書いて登録する必要があります。

ユーザーは、wolfSSL_SetMacEncryptCtx()とwolfSSL_SetDecryptVerifyCtx()で wolfSSL オブジェクト (セッション) ごとにコンテキストを設定できます。このコンテキストは、ユーザー指定のコンテキストへのポインタであり得ます。これは次に Mac/Encrypt および Decrypt/Decrypt/Decrypt/Verify Callbacks に渡されます。void* ctx パラメータ。

1. コールバックの例は、-U コマンドラインオプションを使用する場合、wolfssl/test.h、myMacEncryptCb() および myDecryptVerifyCb() の下で、wolfSSL の例クライアント (examples/client/client.c) で使用できます。

Atomic Record Layer Callbacks を使用するには、wolfSSL を--enable-atomicuser Configure オプションを使用してコンパイルする必要があります。

6.4 公開キーのコールバック

wolfSSL は、SSL/TLS 接続中に RSA の署名/検証機能をもっと多くの制御を希望するユーザーのための公開鍵コールバックを提供します。

ユーザーはオプションで 7 つの関数を定義できます。

1. ECC サインコールバック
2. ECC の確認コールバック
3. ECC はシークレットコールバックを共有しました
4. RSA サインコールバック
5. RSA はコールバックを検証します
6. RSA 暗号化コールバック
7. RSA 復号化コールバック

これら 2 つの機能は ssl.h に CallbackRsaSign,CallbackEccVerify,CallbackRsaSign,CallbackRsaSign,Call
CallbackRsaEnc、CallbackRsaEnc、CallbackRsaSign、CallbackRsaEnc、CallbackRsaDec、
CallbackRsaDec、CallbackRsaDec

```
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);
```

ユーザーは wolfSSL コンテキストごとにこれらの機能を書いて登録する必要があります (WOLFSSL_CTX)。

- wolfSSL_CTX_SetEccSignCb()
- wolfSSL_CTX_SetEccVerifyCb()
- wolfSSL_CTX_SetEccSharedSecretCb()
- wolfSSL_CTX_SetRsaSignCb()
- wolfSSL_CTX_SetRsaVerifyCb()
- wolfSSL_CTX_SetRsaEncCb()
- wolfSSL_CTX_SetRsaDecCb()

ユーザーは、次のように WOLFSSL オブジェクト (セッション) ごとにコンテキストを設定できます。

- wolfSSL_SetEccSignCtx()
- wolfSSL_SetEccVerifyCtx()
- wolfSSL_SetEccSharedSecretCtx()
- wolfSSL_SetRsaSignCtx()
- wolfSSL_SetRsaVerifyCtx()
- wolfSSL_SetRsaEncCtx()
- wolfSSL_SetRsaDecCtx()

これらのコンテキストは、ユーザー指定のコンテキストへのポインターである場合があります。これにより、void* ctx パラメーターを介してそれぞれの公開キーコールバックに渡されます。

コールバックの例は、wolfssl/test.h、myEccSign()、myEccVerify()、myEccSharedSecret()、myRsaSign()、myRsaVerify()、myRsaEnc()、および myRsaDec() に基づいて見つけることができます。

Atomic Record Layer Callbacks を使用するには、wolfSSL を--enable-pkcallbacks Configure オプションを使用してコンパイルする必要があります。

7 キーと証明書

X.509 証明書の紹介と、SSL および TLS での使用方法については、付録 A を参照してください。

7.1 サポートされている形式とサイズ

wolfSSL(以前の Cyassl) は、** pem **、および** der ** 証明書とキーのフォーマット、および PKCS # 8 プライベートキー (PKCS # 5 または PKCS # 12 暗号化) をサポートしています。

** PEM **、または「プライバシー強化されたメール」は、証明書が証明書当局によって発行される最も一般的な形式です。PEM ファイルは、複数のサーバー証明書、中間証明書、およびプライベートキーを含むことができる Base64 エンコード ASCII ファイルであり、通常.pem、.crt、.cer、または.key ファイル拡張子を備えています。PEM ファイル内の証明書は、「-----BEGIN CERTIFICATE-----」および「-----END CERTIFICATE-----」ステートメントに包まれています。

** der **、または「区別されたエンコードルール」は、証明書のバイナリ形式です。der ファイル拡張子には.der および.cer を含めることができ、テキストエディターでは表示できません。

X.509 証明書は、ASN.1 形式を使用してエンコードされます。der 形式は ASN.1 エンコーディングです。PEM 形式は、Base64 エンコードされており、人間の読み取り可能なヘッダーとフッターでラップされています。TLS は DER 形式で証明書を送信します。

7.2 サポートされている証明書の拡張子

クリティカルとしてマークされたサポートされていない、または不明な拡張機能が見つかった場合、エラーメッセージが返されます。それ以外の場合は、サポートされていないか不明な拡張子が見つかりました は無視されます。証明書拡張子の解析では、少なくとも --enable-certext (マクロ WOLFSSL_CERT_EXT) が使用された場合 wolfSSL ライブラリのコンパイル。これは証明書のハイレベルなリストです **解析**可能な拡張子と、すべてではないにしても少なくとも一部の拡張子 使用されます。

RFC 5280 からの拡張	サポート
権限キー識別子	はい
サブジェクト キー識別子	はい
キーの使用法	はい
証明書ポリシー	はい
ポリシー マッピング	いいえ
サブジェクトの別名	はい
発行者の別名	いいえ
サブジェクト ディレクトリの属性	いいえ
基本的な制約	はい
名前の制約	はい
ポリシーの制約	はい
拡張キー使用法	はい
CRL 配布ポイント	はい
anyPolicy を禁止する	はい
最新の CRL	いいえ

追加拡張	サポート
ネットスケープ	はい
カスタム OID	はい

次の 2 つのセクションでは、個々の証明書拡張機能のサポートについて詳しく説明します。

7.2.0.1 認証/サブジェクト キー ID デフォルトでは、キー ID はキーの SHA1 ハッシュです。キーの SHA256 ハッシュは もサポートされています。

7.2.0.2 サブジェクトの別名

別名タイプ	サポート
電子メール	はい
DNS 名	はい
IP アドレス	はい
URI	はい

7.2.0.3 キーの使い方 キーの使用法は、証明書の解析が完了した後に解析および取得できます。

キーの使用法	サポート
デジタル署名	はい
否認防止	はい
キー暗号化	はい
データ暗号化	はい
キー契約	はい
keyCertSign	はい
cRL サイン	はい
暗号化のみ	はい
解読のみ	はい

7.2.0.4 拡張キー使用法 拡張キーの使用法が unknown/unsupported の場合は無視されます。3.15.5 より前のバージョンの場合は不明 拡張キー使用法 OID は解析エラーを引き起こします。

拡張キー使用法	サポート
任意の拡張キー使用法	はい
id-kp-serverAuth	はい
id-kp-clientAuth	はい
id-kp-codeSigning	はい
id-kp-emailProtection	はい
id-kp-timeStamping	はい
id-kp-OCSPSigning	はい

7.2.0.5 カスタム OID カスタム OID インジェクションと解析は、wolfSSL バージョン 5.3.0 で導入されました。有効化オプション –enable-certgen および –enable-asn=template は、マクロとともに使用する必要があります。カスタム拡張機能を操作するための WOLFSSL_CUSTOM_OID および HAVE_OID_ENCODING。これらの設定で wolfSSL を構築した後、関数 wc_SetCustomExtension を使用して証明書構造体にカスタム拡張を設定できます。

7.3 証明書の読み込み

証明書は通常、ファイルシステムを使用してロードされます (ただし、メモリバッファーからの読み込みもサポートされています - ファイルシステムがなく、証明書の使用を参照)。

7.3.1 CA 証明書をロードする

CA 証明書ファイルは、wolfSSL_CTX_load_verify_locations()関数を使用してロードできます。

```
int wolfSSL_CTX_load_verify_locations(WOLFSSL_CTX *ctx,
                                      const char *CAfile,
                                      const char *CApath);
```

CA Loading は、PEM 形式で CAfile をできるだけ多くの CERT で渡して、上記の機能を使用してファイルごとの複数の CA 証明書を解析することもできます。これにより、初期化が簡単になり、クライアントが起動時に複数のルート CA 証明書をロードする必要がある場合に役立ちます。これにより、wolfSSL は CA 証明書用の単一のファイルを使用できるようにするツールに移植するのを簡単にします。

注：ルーツのチェーンと中間証明書をロードする必要がある場合は、信頼の順にロードする必要があります。最初にルート CA をロードした後、中間体 1 に続いて中間体 2 などが続きます。CERT がロードされるごとにwolfSSL_CTX_load_verify_locations()を呼び出すか、CERT を順番に含むファイルを 1 回 (フファイルの先頭にあるルートと信頼チェーンによって順序付けられた証明書) を使用できます。

7.3.2 クライアントまたはサーバーの証明書を読み込みます

単一のクライアントまたはサーバー証明書の読み込みは、wolfSSL_CTX_use_certificate_file()関数で実行できます。この関数が証明書チェーンで使用される場合、実際の、つまり「最下位」の証明書のみが送信されます。

```
int wolfSSL_CTX_use_certificate_file(WOLFSSL_CTX *ctx,
                                     const char *CAfile,
                                     int type);
```

CAfile は CA 証明書ファイルであり、type は SSL_FILETYPE_PEM などの証明書の形式です。

サーバーとクライアントは、wolfSSL_CTX_use_certificate_chain_file()関数を使用して証明書チェーンを送信できます。証明書チェーンファイルは PEM 形式である必要があり、被験者の証明書 (実際のクライアントまたはサーバー証明書) から始まり、その後、中間証明書が続き、(オプションでは) 最上位のルート CA 証明書で終了する必要があります。サンプルサーバー (/examples/server/server.c) は、この機能を使用します。

注意：これは、検証のために証明書チェーンをロードするときに必要な順序とはまったく逆です! このシナリオでのファイルの内容は、ファイルの先頭にエンティティ証明書があり、その後にチェーンの次の証明書が続き、ファイルの末尾にルート CA が続きます。

```
int wolfSSL_CTX_use_certificate_chain_file(WOLFSSL_CTX *ctx,
                                   const char *file);
```

7.3.3 秘密鍵を読み込む

サーバープライベートキーは、wolfSSL_CTX_use_PrivateKey_file()関数を使用してロードできます。

```
int wolfSSL_CTX_use_PrivateKey_file(WOLFSSL_CTX *ctx,
                        const char *keyFile, int type);
```

拡張キー使用法	サポート
任意の拡張キー使用法	はい
id-kp-serverAuth	はい
id-kp-clientAuth	はい
id-kp-codeSigning	はい
id-kp-emailProtection	はい
id-kp-timeStamping	はい

拡張キー使用法	サポート
id-kp-OCSPSigning	はい

keyFile は秘密のキーファイルであり、type は秘密鍵の形式です (例：SSL_FILETYPE_PEM)。

7.3.4 信頼できるピア証明書を読み込み

拡張キー使用法	サポート
任意の拡張キー使用法	はい
id-kp-serverAuth	はい
id-kp-clientAuth	はい
id-kp-codeSigning	はい
id-kp-emailProtection	はい
id-kp-timeStamping	はい
id-kp-OCSPSigning	はい

使用する信頼できるピア証明書を読み込むことは、wolfSSL_CTX_trust_peer_cert()で行うことができます。

```
int wolfSSL_CTX_trust_peer_cert(WOLFSSL_CTX *ctx,
                  const char *trustCert, int type);
```

trustCert はロードする証明書ファイルであり、type は秘密鍵の形式 (つまり SSL_FILETYPE_PEM) です。

7.4 証明書チェーン検証

wolfSSL では、証明書チェーンを検証するために、チェーンのトップまたは「ルート」証明書のみを信頼できる証明書としてロードする必要があります。これは、証明書チェーン (A -> B -> C) がある場合、C は B によって署名され、B は A によって署名される場合、wolfSSL は、チェーン全体 (A->B->C) を検証するために、証明書 A を信頼できる証明書としてロードすることのみを必要とします。

たとえば、サーバー証明書チェーンが次のように見える場合

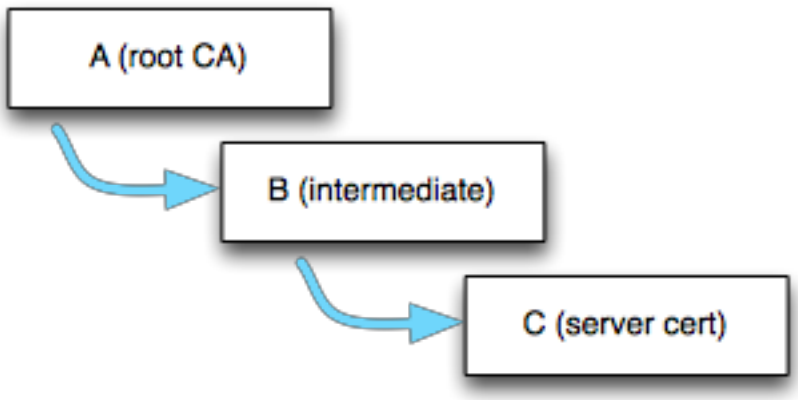

Figure 3: Certificate Chain

wolfSSL クライアントには、少なくともルート証明書 (a) が信頼できるルート (wolfSSL_CTX_load_verify_locations(
としてロードされている必要があります。クライアントがサーバー CERT チェーンを受信すると、A の署名

を使用して B を検証し、B が以前に信頼できるルートとして wolfSSL にロードされていなかった場合、B は wolfSSL の内部 CERT チェーンに保存されます (wolfSSL は必要なものを保存します証明書の確認：一般名ハッシュ、公開キー、キータイプなど)。B が有効な場合、C を確認するために使用されます。

このモデルに従って、ルート証明書「A」が wolfSSL サーバーに信頼できるルートとしてロードされている限り、サーバーが送信する場合、サーバー証明書チェーンを確認することができます (A -> B -> C)、または (b -> c)。サーバーが中間証明書ではなく (c) を送信するだけの場合、wolfSSL クライアントが既に B を信頼できるルートとしてロードしていない限り、チェーンを検証することはできません。

7.5 ドメイン名サーバー証明書のチェック

wolfSSL には、サーバー証明書のドメインを自動的にチェックするクライアントに拡張機能があります。OpenSSL モードでは、これを実行するにはほぼ 1 ダースの関数呼び出しが必要です。wolfSSL は、証明書の日付が範囲にあることをチェックし、署名を検証し、`wolfSSL_connect()`は DOMAIN_NAME_MISMATCH を返します。

証明書のドメイン名を確認することは、サーバーが実際にあると主張しているのかを確認する重要な手順です。この拡張は、チェックを実行する負担を軽減することを目的としています。

7.6 ファイルシステムがなく、証明書の使用

通常、ファイルシステムは、プライベートキー、証明書、および CA のロードに使用されます。wolfSSL は完全なファイルシステムのない環境で使用されることがあるため、代わりにメモリバッファーを使用するための拡張機能が提供されます。拡張機能を使用するには、定数 NO_FILESYSTEM を定義し、次の機能が利用可能になります。

- `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);`

これらの関数を *_buffer ではなく *_file という名前のカウンターパートとまったく同じように使用します。そして、ファイル名を指定する代わりにメモリバッファを提供します。使用上の詳細については API の資料を参照してください。

7.6.1 テスト証明書とキーバッファー

wolfSSL は、過去にテスト証明書とキーファイルのみがバンドルされていました。現在は、また、ファイルシステムが利用できない環境で使用するためのテスト証明書とキーバッファがバンドルされています。これらのバッファーは、USE_CERT_BUFFERS_1024、USE_CERT_BUFFERS_2048、または USE_CERT_BUFFERS_256 の 1 つ以上を定義する場合、certs_test.h で利用できます。

7.7 シリアル番号検索

X.509 証明書のシリアル番号は、`wolfSSL_X509_get_serial_number()`を使用して wolfSSL から抽出できます。

```
int wolfSSL_X509_get_serial_number(WOLFSSL_X509* x509,
            unsigned char* buffer, int* inOutSz)
```

buffer は、入力で最大 *inOutSz バイトで書き込まれます。コールの後、成功した場合 (0 の戻り)、*inOutSz は buffer に書き込まれた実際のバイト数を保持します。完全な例は wolfssl/test.h が含まれています。

7.8 RSA キー生成

wolfSSL は、最大 4096 ビットのさまざまな長さの RSA キー生成をサポートしています。キー生成はデフォルトでオフになっていますが、--enable-keygen の ./configure プロセス中に、または Windows または非標準の環境で WOLFSSL_KEY_GEN を定義することにより、オンにすることができます。キーを作成するのは簡単で、rsa.h から 1 つの関数のみが必要です。

```
int MakeRsaKey(RsaKey* key, int size, long e, RNG* rng);
```

size は BITS の長さで、e が公開指数です。

```
RsaKey genKey;
RNG    rng;
int    ret;

InitRng(&rng);
InitRsaKey(&genKey, 0);

ret=MakeRsaKey(&genKey, 1024, 65537, &rng);
if (ret != 0)
    /* ret contains error */;
```

RSAKEY genKey は、他の RSAKEY のように使用できるようになりました。キーをエクスポートする必要がある場合は、wolfssl は asn.h で DER と PEM フォーマットの両方を提供します。常にキーを最初に DER フォーマットに変換してから、PEM が必要な場合は汎用 DerToPem() 関数を使用する必要があります。

```
byte der[4096];
int  derSz=RsaKeyToDer(&genKey, der, sizeof(der));
if (derSz < 0)
    /* derSz contains error */;
```

バッファ der は、キーの DER フォーマットを保持する。DER バッファを PEM に変換するには、変換関数を使用します。

```
byte pem[4096];
int  pemSz=DerToPem(der, derSz, pem, sizeof(pem),
                    PRIVATEKEY_TYPE);
if (pemSz < 0)
    /* pemSz contains error */;
```

dertopem() の最後の引数は、通常 PRIVATEKEY_TYPE または CERT_TYPE のタイプパラメーターを取得します。これで、バッファ pem がキーの PEM 形式を保持します。サポートされているタイプは次のとおりです。

- 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 キー生成ノート

RSA 秘密鍵には公開鍵も含まれています。秘密鍵は、test.c で使用される wolfSSL によってプライベートキーと公開鍵の両方として使用できます。秘密鍵と公開鍵 (証明書の形式) は、SSL に通常必要なものすべてです。

必要に応じて、RsaPublicKeyDecode() 関数を使用して、別の公開キーを手動で wolfSSL にロードできます。さらに、wc_RsaKeyToPublicDer()関数を使用して、パブリック RSA キーをエクスポートできます。

7.9 証明書生成

wolfSSL は X.509 V3 証明書生成をサポートしています。証明書の生成はデフォルトでオフになっていますが、--enable-certgen の./configure プロセス中に、または Windows または非標準の環境で WOLFSSL_CERT_GEN を定義することにより、オンにすることができます。

証明書を生成する前に、ユーザーは証明書の主題に関する情報を提供する必要があります。この情報は、Cert という名前の wolfssl/wolfcrypt/asn_public.h の構造に含まれています。

```
/* 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;
```

CertName は次のようになります。

```
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;
```

サブジェクト情報を入力する前に、次のように初期化関数を呼び出す必要があります。

```
Cert myCert;
InitCert(&myCert);
```

InitCert( ) は、バージョンを **3** (0x02) に、シリアル番号を **0** (ランダムに生成) に、sigType を CTC_SHAwRSA に、daysValid を **500** に、selfSigned を **1** (TRUE) に設定するなど、いくつかの変数のデフォルトを設定します。

サポートされている署名タイプは次のとおりです。

- CTC_SHAwDSA
- CTC_MD2wRSA
- CTC_MD5wRSA
- CTC_SHAwRSA
- CTC_SHAwECDSA
- CTC_SHA256wRSA
- CTC_SHA256wECDSA
- CTC_SHA384wRSA
- CTC_SHA384wECDSA
- CTC_SHA512wRSA
- CTC_SHA512wECDSA

これで、ユーザーは wolfcrypt/test/test.c からこの例のようなサブジェクト情報を初期化できます。

```
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);
```

次に、上記のキー生成例からの変数 GenKey と RNG を使用して自己署名証明書を生成できます (もちろん、有効な RSAKEY または RNG を使用できます)：

```
byte derCert[4096];

int certSz=MakeSelfCert(&myCert, derCert, sizeof(derCert), &key, &rng);
if (certSz < 0)
  /* certSz contains the error */;
```

バッファ derCert には、証明書の DER フォーマットが含まれています。証明書の PEM フォーマットが必要な場合は、Generic DerToPem() 関数を使用して、このように CERT_TYPE になるタイプを指定できます。

```
byte* pem;

int pemSz=DerToPem(derCert, certSz, pem, sizeof(pemCert), CERT_TYPE);
if (pemCertSz < 0)
  /* pemCertSz contains error */;
```

サポートされているタイプは次のとおりです。

- 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

これで、バッファ pemCert< が証明書の PEM 形式を保持します。

CA 署名証明書を作成する場合は、いくつかの手順が必要です。以前のようにサブジェクト情報を記入した後、CA 証明書から発行者情報を設定する必要があります。これは、このように SetIssuer() で行うことができます。

```
ret=SetIssuer(&myCert, "ca-cert.pem");
if (ret < 0)
    /* ret contains error */;
```

その後、証明書を作成してから署名する 2 ステップのプロセスを実行する必要があります (MakeSelfCert() は両方とも 1 段階で行います)。発行者 (caKey) とサブジェクト (key) の両方から秘密鍵が必要です。完全な使用法については test.c の例をご覧ください。

```
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*/;
```

バッファ derCert には、CA 署名証明書の der 形式が含まれるようになりました。証明書の PEM 形式が必要な場合は、上記の自己署名の例を参照してください。MakeCert() および SignCert() は、使用する RSA または ECC キーの関数パラメーターを提供していることに注意してください。上記の例では、RSA キーを使用し、ECC キーパラメーターの null を渡します。

7.10 証明書署名要求 (CSR) 生成

wolfSSL は、X.509 v3 証明書署名要求 (CSR) 生成をサポートしています。CSR 生成はデフォルトでオフになっていますが、--enable-certreq --enable-certgen で ./configure プロセス中、または Windows または非標準の環境で WOLFSSL_CERT_GEN と WOLFSSL_CERT_REQ を定義することによってオンにすることができます。

CSR を生成する前に、ユーザーは証明書の主題に関する情報を提供する必要があります。この情報は、Cert という名前の wolfssl/wolfcrypt/asn_public.h の構造に含まれています。

CERT および CERTNAME 構造の詳細については、上記の証明書生成を参照してください。

サブジェクト情報に記入する前に、初期化機能を次のように呼び出す必要があります。

```
Cert request;
InitCert(&request);
```

InitCert() は、バージョンを **3** (0x02) に、シリアル番号を **0** (ランダムに生成) に、sigType を CTC_SHAwRSA に、daysValid を **500** に、selfSigned を **1** (TRUE) に設定するなど、いくつかの変数のデフォルトを設定します。

サポートされている署名タイプは次のとおりです。

- CTC_SHAwDSA
- CTC_MD2wRSA
- CTC_MD5wRSA
- CTC_SHAwRSA
- CTC_SHAwECDSA
- CTC_SHA256wRSA
- CTC_SHA256wECDSA
- CTC_SHA384wRSA
- CTC_SHA384wECDSA
- CTC_SHA512wRSA

- CTC_SHA512wECDSA

これで、ユーザーはhttps://github.com/wolfssl/wolfssl-examples/blob/master/certgen/csr_example.cからこの例のようなサブジェクト情報を初期化できます。

```
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);
```

次に、上記のキー生成例 (もちろん有効な ECC/RSA キーまたは RNG を使用できます) の変数キーを使用して、有効な署名された CSR を生成できます。

```
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 */
```

次に、リクエストに署名して有効にします。上記のキー生成例から RNG 変数を使用します。(もちろん、有効な ECC/RSA キーまたは RNG を使用できます)

```
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 */
```

最後に、CSR を PEM 形式に変換して、証明書の発行に使用する CA 権限に送信します。

```
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 制限

CSR で除外された証明書に必須のフィールドがあります。CSR には、証明書に載っているときに必須の「オプション」と見なされる他のフィールドがあります。このため、すべての証明書フィールドを厳密にチェックし、すべてのフィールドを必須と見なす wolfSSL 証明書解析エンジンは、現時点では CSR の使用をサポートしていません。したがって、CSR の生成と証明書の生成はゼロからサポートされていますが、wolfSSL は CSR からの証明書生成をサポートしていません。CSR を wolfSSL 解析エンジンに渡すと、現時点では障害が返されます。証明書生成で CSR の使用をサポートしたかについては更新を確認してください！

関連項目：証明書生成

7.11 Raw の ECC キーに変換

Raw ECC キーのインポートに対する最近追加されたサポートでは、ECC キーを PEM から DER に変換する機能が得られます。これを実行するには、指定された引数を次に使用します。

```
EccKeyToDer(ecc_key*, byte* output, word32 inLen);
```

7.11.1 例

```
#define FOURK_BUF 4096
byte  der[FOURK_BUF];
```

```
ecc_key userB;

EccKeyToDer(&userB, der, FOURK_BUF);
```

8 デバッグ

8.1 デバッグとロギング

wolfSSL(以前の Cyassl) は、デバッグが制限されている環境でログメッセージを介したデバッグをサポートしています。ロギングをオンにするには、機能wolfSSL_Debugging_ON()をオフにします。通常のビルド (リリースモード) では、これらの関数は効果がありません。デバッグビルドで、これらの機能がオンになっていることを確認するために DEBUG_WOLFSSL を定義します。

wolfSSL 2.0 以降、ロギングコールバック関数はランタイムに登録されて、ログ記録の完了方法をより柔軟に提供することができます。ロギングコールバックは機能wolfSSL_SetLoggingCb()に登録できます。

```
int wolfSSL_SetLoggingCb(wolfSSL_Logging_cb log_function);

typedef void (*wolfSSL_Logging_cb)(const int logLevel,
                   const char *const logMessage);
```

ログレベルは wolfssl/wolfcrypt/logging.h にあり、実装は logging.c にあります。デフォルトでは、wolfssl は stderr に fprintf でロギングします。

8.2 エラーコード

wolfSSL は、デバッグを支援するために、有益なエラーメッセージを提供します。

各wolfSSL_read()呼び出しは、read() と write() と同じように、成功時に 0、接続クロージャーの場合は 0、およびエラーの間のバイト数を返します。エラーが発生した場合は、2 つの API を使用して、エラーの情報を入手できます。

関数wolfSSL_get_error()の結果値を引数として取得し、対応するエラーコードを返します。

```
int err=wolfSSL_get_error(ssl, result);
```

より読み取り可能なエラーコードの説明を取得するには、wolfSSL_ERR_error_string()からのリターンコードとストレージバッファーを引数として取得し、対応するエラーの説明をストレージバッファー (以下の例の errorString) に配置します。

```
char errorString[80];
wolfSSL_ERR_error_string(err, errorString);
```

ノンブロッキングソケットを使用している場合は、EEAGAIN/EWOULDBLOCK をテストするか、SSL_ERROR_WANT_READ または SSL_ERROR_WANT_WRITE の特定のエラーコードをテストできます。

wolfSSL と WolfCrypt エラーコードのリストについては、付録 C(エラーコード) を参照してください。

9 ライブラリデザイン

9.1 ライブラリヘッダー

wolfSSL 2.0.0 RC3 のリリースにより、ライブラリヘッダーファイルは次の場所にあります。

- Wolfssl：`wolfssl/`
- wolfcrypt：`wolfssl/wolfcrypt/`
- wolfSSL OpenSSL 互換レイヤー：`wolfssl/openssl/`

OpenSSL 互換レイヤーを使用する場合 (OpenSSL 互換性参照)、`/wolfssl/openssl/ssl.h` ヘッダーを含める必要があります。

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

wolfSSL ネイティブ API のみを使用する場合は、`/wolfssl/ssl.h` ヘッダーのみを含める必要があります。

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

9.2 起動と終了

すべてのアプリケーションは、ライブラリを使用する前に`wolfSSL_Init()`を呼び出す必要があります。現在、これらの機能は、マルチユーザーモードでセッションキャッシュの共有ミューテックスを初期化して解放しますが、将来的にはより多くのことをするかもしれないので、それらを使用することは常に良い考えです。

9.3 構造の使用

ヘッダーファイルの場所の変更に加えて、wolfSSL 2.0.0 RC3 のリリースは、ネイティブの wolfSSL API と wolfSSL OpenSSL 互換層の間でより目に見える区別を作成しました。この区別により、ネイティブの wolfSSL API によって使用される主な SSL/TLS 構造は、名前が変更されました。新しい構造は次のとおりです。OpenSSL 互換層を使用するときは、前の名前はまだ使用されます (OpenSSL 互換性参照)。

- `WOLFSSL`(以前の SSL)
- `WOLFSSL_CTX`(以前は SSL_CTX)
- `WOLFSSL_METHOD`(以前は ssl_method)
- `WOLFSSL_SESSION`(以前は ssl_session)
- `WOLFSSL_X509`(以前は x509)
- `WOLFSSL_X509_NAME`(以前 x509_name)
- `WOLFSSL_X509_CHAIN`(以前は x509_chain)

9.4 スレッドの安全性

wolfssl(以前の Cyassl) は、設計によりスレッドセーフです。wolfSSL はグローバル データ、静的データ、およびオブジェクトの共有を回避するため、複数のスレッドが競合を発生させることなくライブラリに同時に入ることができます。ユーザーは、2 つの領域で潜在的な問題を回避するように注意する必要があります。

1. クライアントは、複数のスレッドにわたって wolfSSL オブジェクトを共有することができますが、アクセスは同期する必要があります。すなわち、同じ SSL ポインタを持つ 2 つの異なるスレッドから同時に Read/Write することはできません。

wolfSSL は、共有できない関数に入ったときにより積極的な（制限的な）スタンスを取り、他のユーザーをロックアウトする可能性がありますが、このレベルの粒度は直感に反しているようです。すべてのユーザー(シングルスレッドでさえ) はロックの負荷を負担し、マルチスレッドはスレッド間でオブジェクトを共有していなくてもライブラリーに再入力できません。このペナルティは高すぎるように思われが、wolfSSL は共有オブジェクトを同期する責任をユーザーの手に委ねます。

2. wolfSSL のポインターを共有するだけでなく、ユーザーは`wolfSSL_new()`の作成中にのみ読み取られます (または WOLFSSL_CTX の将来の変化 (または同時的な変化)WOLFSSL オブジェクトが作成されると、反映されません。
3. 一部の最適化では、スレッドごとにメモリが割り当てられます。固定小数点 ECC キャッシュが有効になっている場合 (FP_ECC)、スレッドは、スレッドが終了する前に `wc_ecc_fp_free()` を使用してキャッシュされたバッファを解放する必要があります。

繰り返しになりますが、複数のスレッドは WOLFSSL_CTX への書き込みアクセスを同期させる必要があり、単一のスレッドが WOLFSSL_CTX を初期化して上述の同期および更新の問題を回避することをお勧めします。

9.5 入力および出力バッファー

wolfSSL は、入力と出力に動的バッファーを使用するようになりました。デフォルトは 0 バイトになり、`wolfssl/internal.h` で RECORD_SIZE 定義によって制御されます。静的バッファーよりもサイズが大きい入力レコードを受信した場合、動的バッファーは一時的にリクエストを処理して解放されます。静的バッファーサイズを 2^16 または 16,384 の MAX_RECORD_SIZE に設定できます。

wolfssl が操作された前の方法を好む場合は、ダイナミックメモリを必要としない 16kb の静的バッファを使用して、LARGE_STATIC_BUFFERS を定義することでそのオプションを取得できます。

動的バッファーが使用され、ユーザーがバッファサイズよりも大きい`wolfSSL_write()`を要求する場合、最大 MAX_RECORD_SIZE までの動的ブロックを使用してデータを送信します。最大 RECORD_SIZE サイズのチャンクでデータを送信したいユーザーは、STATIC_CHUNKS_ONLY を定義することでこれを行うことができます。これにより、wolfSSL は、デフォルトで 128 バイトの RECORD_SIZE まで成長する I/O バッファーを使用します。

10 WolfCrypt の使用法

WolfCrypt は、主に wolfSSL が使用する暗号化ライブラリです。速度、小さなフットプリント、および移植性のために最適化されています。wolfSSL は、必要に応じて他の暗号ライブラリと交換します。

例で使用される型：

```
typedef unsigned char byte;
typedef unsigned int  word32;
```

10.1 ハッシュ関数

10.1.1 MD4

注：MD4 は古くて安全でないと考えられています。可能であれば、異なるハッシュ関数を使用してください。

MD4 を使用するには、MD4 ヘッダー wolfssl/wolfcrypt/md4.h を含みます。使用する構造は Md4 です。これは Typedef です。使用する前に、ハッシュの初期化はwc_InitMd4()を使用してください。

```
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 には、バッファーのハッシュデータのダイジェストが含まれています。

10.1.2 MD5

注：MD5 は時代遅れであり、不安定であると考えられています。可能であれば、異なるハッシュ関数を使用してください。

MD5 を使用するには、MD5 ヘッダー wolfssl/wolfcrypt/md5.h が含まれます。使用する構造は Md5 で、これは typedef です。使用する前に、ハッシュの初期化はwc_InitMd5()を更新して最終的なハッシュを取得します

```
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 には、バッファーのハッシュデータのダイジェストが含まれています。

10.1.3 SHA/SHA-224/SHA-256/SHA-384/SHA-512

SHA を使用するには、SHA ヘッダー wolfssl/wolfcrypt/sha.h を含めます。使用する構造は Sha で、typedef です。使用する前に、ハッシュの初期化はwc_InitSha()を更新して最終的なハッシュを取得します。

```
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 には、バッファーのハッシュデータのダイジェストが含まれています。

SHA-224、SHA-256、SHA-384、または SHA-512 を使用するには、上記と同じ手順に従いますが、wolfssl/wolfcrypt/sha256.h または wolfssl/wolfcrypt/sha512.h(SHA-384 と SHA-512 の両方) のいずれかを使用します。SHA-256、SHA-384、および SHA-512 機能は、SHA 関数と同様に命名されています。

** sha-224 ** の場合、関数wc_InitSha224()が構造 SHA224 で使用されます。

** sha-256 ** の場合、関数wc_InitSha256()が構造 SHA256 で使用されます。

** SHA-384 ** では、機能wc_InitSha384()は構造 SHA384 で使用されます。

** sha-512 ** の場合、関数wc_InitSha512()が構造 SHA512 で使用されます。

10.1.4 blake2b

Blake2B(SHA-3 ファイナリスト) を使用するには、Blake2B ヘッダー wolfssl/wolfcrypt/blake2.h を含みます。使用する構造は Blake2b です。これは Typedef です。使用する前に、ハッシュの初期化はwc_InitBlake2b()を更新して最終的なハッシュを取得します。

```
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);
```

wc_InitBlake2b()の 2 番目のパラメータは、最終ダイジェストサイズです。digest バッファ内のハッシュデータのダイジェストを含みます。

使用例使用例は、blake2b_test() 関数内で、WolfCrypt テストアプリケーション (wolfcrypt/test/test.c) にあります。

10.1.5 RIPEMD-160

RIPEMD-160 を使用するには、ヘッダー wolfssl/wolfcrypt/ripemd.h を含めます。使用する構造は RipeMd で、これは typedef です。使用する前に、ハッシュの初期化はwc_InitRipeMd()を更新して最終的なハッシュを取得します

```
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 には、バッファーのハッシュデータのダイジェストが含まれています。

10.2 キー付きハッシュ関数

10.2.1 HMAC

WolfCrypt は現在メッセージダイジェストニーズに HMAC を提供しています。構造 Hmac はヘッダ wolfssl/wolfcrypt/hmac.h にあります.HMAC 初期化はwc_HmacSetKey()で行われます.5 つの異なるタイプが HMAC でサポートされています.MD5、SHA、SHA-256、SHA-384、および SHA-512。これが SHA-256 の例です。

```
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 には、バッファーのハッシュデータのダイジェストが含まれています。

10.2.2 GMAC

WolfCrypt は、メッセージダイジェストのニーズにも GMAC を提供します。構造 Gmac は、アプリケーション AES-GCM であるため、ヘッダー wolfssl/wolfcrypt/aes.h にあります。GMAC 初期化はwc_GmacSetKey()で行われます。

```
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 には、バッファーのハッシュデータのダイジェストが含まれています。

10.2.3 poly1305

WolfCrypt は、メッセージダイジェストニーズに合わせて Poly1305 も提供しています。構造体 Poly1305 はヘッダ wolfssl/wolfcrypt/poly1305.h にあります.Poly1305 の初期化はwc_Poly1305SetKey()以降に Poly1305 を使用して次に Poly1305 を使用して実行する必要があります。

```
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 には、バッファーのハッシュデータのダイジェストが含まれています。

10.3 ブロック暗号

10.3.1 AES

WolfCrypt は、16 バイト (128 ビット)、24 バイト (192 ビット)、または 32 バイト (256 ビット) のキーサイズを持つ AES をサポートしています。サポートされている AES モードには、CBC、CTR、GCM、および CCM-8 が含まれます。

CBC モードは、暗号化と復号化の両方でサポートされており、wc_AesSetKey()の関数を介して提供されます。AES を使用するヘッダー wolfssl/wolfcrypt/aes.h を含めてください。AES には 16 バイトのブロックサイズがあり、IV も 16 バイトになります。関数の使用法は通常次のとおりです。

```
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 には、プレーンテキストの暗号文が含まれています。

```
/*decrypt*/
wc_AesSetKey(&dec, key, sizeof(key), iv, AES_DECRYPTION);
wc_AesCbcDecrypt(&dec, plain, cipher, sizeof(cipher));
```

plain CipherText から元の平文を含みます。

WolfCrypt はまた、CTR(Counter)、GCM(Galois/Counter)、および CCM-8(CBC-MAC の COUNTER) の AES の操作モードをサポートしています。CBC のようにこれらのモードを使用する場合は、wolfssl/wolfcrypt/aes.h ヘッダーを含めます。

GCM モードは、wc_AesGcmSetKey()関数を介した暗号化と復号化の両方で使用できます。使用例については、<wolfssl_root>/wolfcrypt/test/test.c の aesgcm_test() 関数を参照してください。

CCM-8 モードは、wc_AesCcmSetKey()の関数を介して暗号化と復号化の両方でサポートされています。使用例については、<wolfssl_root>/wolfcrypt/test/test.c の aesccm_test() 関数を参照してください。

CTR モードは、wc_AesCtrEncrypt()関数を介して暗号化と復号化の両方で使用できます。暗号化および復号化アクションは同一であるため、両方に同じ関数が使用されます。使用例については、ファイル wolfcrypt/test/test.c の関数 aes_test() を参照してください。

10.3.1.1 des and 3des WolfCrypt は DES と 3DES のサポートを提供します (3 は無効な先頭の C 識別子です)。これらを使用するには、ヘッダー wolfssl/wolfcrypt/des.h が含まれます。使用できる構造は Des および Des3 です。3DES の場合は 24)、ブロックサイズは 8 バイトであるため、機能を暗号化/復号化するために 8 バイトの増分だけを渡します。データがブロックサイズの増分になっていない場合は、必ずパディングを追加する必要があります。各 SetKey() はまた IV(キーサイズと同じサイズの初期化ベクトル) を取ります。使用法は通常次のようなものです。

```
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 には、プレーンテキストの暗号文が含まれています。

```
/*decrypt*/
wc_Des3_SetKey(&dec, key, iv, DES_DECRYPTION);
wc_Des3_CbcDecrypt(&dec, plain, cipher, sizeof(cipher));
```

plain CipherText から元の平文を含みます。

10.3.1.2 カメリア WolfCrypt は、Camellia ブロック暗号をサポートしています。Camellia を使用するには、ヘッダー wolfssl/wolfcrypt/camellia.h を含みます。使用できる構造は Camellia と呼ばれます。初期化はwc_CamelliaSetKey()まで提供されます。

使用例については、<wolfssl_root>/wolfcrypt/test/test.c の camellia_test() 関数を参照してください。

10.4 シップシンパー

10.4.1 ARC4

注：ARC4 は古くて安全でないと考えられています。別のストリーム暗号を使用することを検討してください。

インターネットで使用される最も一般的なストリーム暗号は ARC4 です。WolfCrypt は、ヘッダー wolfssl/wolfcrypt/arc4.h を通じてそれをサポートしています。ブロックサイズがなく、キーの長さが任意の長さであるため、使用法はブロック暗号よりも簡単です。以下は、ARC4 の典型的な使用法です。

```
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 には、プレーンテキストの暗号文が含まれています。

```
/*decrypt*/
wc_Arc4SetKey(&dec, key, sizeof(key));
wc_Arc4Process(&dec, plain, cipher, sizeof(cipher));
```

plain CipherText から元の平文を含みます。

10.4.2 Chacha

20 ラウンドのチャチャは、高レベルのセキュリティを維持しながら、ARC4 よりわずかに高速です。WolfCrypt で使用するには、ヘッダー wolfssl/wolfcrypt/chacha.h を含めてください。Chacha は通常 32 バイトキー (256 ビット) を使用しますが、16 バイトキー (128 ビット) を使用できます。

```
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 には、プレーンテキストの暗号文が含まれています。

```
/*decrypt*/
wc_Chacha_SetKey(&enc, key, keySz);
wc_Chacha_SetIV(&enc, iv, counter);
wc_Chacha_Process(&enc, plain, cipher, sizeof(cipher));
```

plain CipherText から元の平文を含みます。

wc_Chacha_SetKeyは 1 回だけ設定する必要がありますが、送信された情報のパケットごとに、New IV(Nonce) で呼び出す必要があります。Counter は、暗号化/復号化プロセスを実行するときに別のブロックから開始することによって、情報の部分的な復号化/暗号化を可能にする引数として設定されますが、ほとんどの場合、0 に設定されます。**ChaCha は、MAC アルゴリズムなしで使用しないでください (例 Poly1305、hmac).**

10.5 公開鍵暗号

10.5.1 RSA

WolfCrypt は、ヘッダー wolfssl/wolfcrypt/rsa.h を介して RSA のサポートを提供します。Public と Private の RSA キーには 2 つのタイプがあります。RSA 鍵には、公開鍵と秘密鍵の 2 種類があります。公開鍵を使用すると、秘密鍵の所有者だけが解読できるものを誰でも暗号化できます。また、秘密鍵の所有者が何かに署名することもでき、公開鍵を持っている人は誰でも、秘密鍵の所有者だけが実際に署名したことを確認できます。使用法は通常次のようなものです。

```
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);
```

現在、out は、プレーンテキスト in から暗号文を保持します。wc_RsaPublicEncrypt()を呼び出すことができます。

エラーが発生した場合、wc_RsaPublicEnrypt() を呼び出して、発生したエラーを説明する文字列を取得できます。

```
void wc_ErrorString(int error, char* buffer);
```

バッファーが少なくとも MAX_ERROR_SZ バイトであることを確認してください (80)。

復号化するために：

```
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);
```

これで、plain は plainSz バイトまたはエラーコードを保持します。wolfcrypt の各タイプの完全な例については、ファイル wolfcrypt/test/test.c を参照してください。wc_RsaPrivateKeyDecode関数は、raw DER 形式のキーのみを受け入れることに注意してください。

10.5.2 DH(diffie-hellman)

WolfCrypt は、Header wolfssl/wolfrypt/dh.h を通じて Diffie-Hellman のサポートを提供します。Diffie-Hellman キー Exchange アルゴリズムにより、2 つの関係者が共有秘密キーを確立できるようになります。使用は通常、次の例に似ています。ここで、** sideA ** and ** sideB ** は 2 つのパーティを指定します。

次の例では、dhPublicKey には、認証局によって署名された Diffie-Hellman パブリックパラメータが含まれています (または自己署名)。privA SIDEA の生成された秘密鍵を保持し、pubA は SIDEA の生成された公開鍵を保持し、agreeA は両側が合意したとの相互鍵を保持しています。

```
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()` DHPublickey の初期パラメータに基づいて、パブリックおよびプライベート DH キーを生成します。

```
wc_DhGenerateKeyPair(&dhPublicKey, &rng, privA, &privASz,
            pubA, &pubASz);
```

SIDEB が SIDEA に公開鍵 (pubB) を送信した後、SIDEA は`wc_DhAgree()`関数を使用して相互合意キー (agreeA) を生成できます。

```
wc_DhAgree(&dhPublicKey, agreeA, &agreeASz, privA, privASz,
           pubB, pubBSz);
```

現在、agreeA は、SIDEA の相互に生成されたキー (サイズ agreeASz バイト) を保持しています。同じプロセスが SideB で行われます。

WolfCrypt の Diffie-Hellman の完全な例については、ファイル `wolfcrypt/test/test.c` を参照してください。

10.5.3 EDH(Ephemeral DIFFIE-HELLMAN)

wolfSSL サーバーは、Ephemeral Diffie-Hellman を行うことができます。この機能を追加するためにビルドの変更は必要ありませんが、アプリケーションは EDH 暗号スイートを有効にするためにサーバー側に Ephemeral グループパラメーターを登録する必要があります。これを行うために新しい API を使用できます。

```
int wolfSSL_SetTmpDH(WOLFSSL* ssl, unsigned char* p,
                     int pSz,unsigned char* g,int gSz);
```

サンプルサーバーと echo サーバーは、この関数を SetDH() から使用しています。

10.5.4 DSA(デジタル署名アルゴリズム)

WolfCrypt は、ヘッダー `wolfssl/wolfcrypt/dsa.h` を介して DSA および DSS のサポートを提供します。DSA は、特定のデータハッシュに基づいてデジタル署名を作成できます。DSA は、SHA ハッシュアルゴリズムを使用してデータブロックのハッシュを生成し、署名者の秘密鍵を使用してハッシュする署名を生成します。標準的な使用法は、次のものと似ています。

最初に DSA キー構造 (key) を宣言し、署名される最初のメッセージ (message) を初期化し、DSA キーバッファ (dsaKeyBuffer) を初期化します (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*/ }
```

次に、SHA 構造 (sha)、乱数ジェネレーター (rng)、SHA ハッシュ (hash) を保存する配列、署名 (signature)、idx を保存する配列を宣言します (dsaKeyBuffer で読み始める場所をマークするために)、検証後の戻り値を保持するための int(answer) などがあります。

```
Sha    sha;
RNG    rng;
byte   hash[SHA_DIGEST_SIZE];
byte   signature[40];
word32 idx=0;
int    answer;
```

SHA ハッシュを設定して作成します。WolfCrypt の SHA アルゴリズムの詳細については、SHA/SHA-224/SHA-256/SHA-384/SHA-512を参照してください.message の SHA ハッシュは、変数 hash に格納されています。

```
wc_InitSha(&sha);
wc_ShaUpdate(&sha, message, sizeof(message));
wc_ShaFinal(&sha, hash);
```

DSA キー構造を初期化し、構造のキー値に入力し、乱数ジェネレーター (rng) を初期化します。

```
wc_InitDsaKey(&key);
wc_DsaPrivateKeyDecode(dsaKeyBuffer, &idx, &key,
                       sizeof(dsaKeyBuffer));
wc_InitRng(&rng);
```

wc_DsaSign()関数は、DSA 秘密キー、ハッシュ値、および乱数ジェネレーターを使用して署名 (signature) を作成します。

```
wc_DsaSign(hash, signature, &key, &rng);
```

署名を確認するには、wc_DsaVerify()を使用して DSA キー構造を解放します。

```
wc_DsaVerify(hash, signature, &key, &answer);
wc_FreeDsaKey(&key);
```

11 SSL チュートリアル

11.1 序章

wolfSSL(以前の Cyassl) の組み込み SSL ライブラリは、SSL と TLS を追加することで、通信セキュリティを強化するために既存のアプリケーションまたはデバイスに簡単に統合できます。wolfSSL は組み込み環境および RTOS 環境を対象としており、そのため優れた性能を維持しながら最小限のフットプリントを提供します。wolfSSL の最小ビルドサイズは、選択されたビルドオプションと使用されているプラットフォームに応じて 20~100KB の間の範囲です。

このチュートリアルの目標は、SSL と TLS の統合を簡単なアプリケーションに統合することです。このチュートリアルを通過するプロセスが、一般的に SSL をよりよく理解するプロセスにつながることを願っています。このチュートリアルでは、SSL サポートをアプリケーションに追加する一般的な手順を実証しながら、シンプルな EchoServer および EchoClient の例と組み合わせて wolfSSL を使用して、可能な限りシンプルなものを保持します。EchoServer と EchoClient の例は、リチャード・スティーブンス、ビル・フェナー、アンドリュー・ルドフによるUNIX ネットワークプログラミング、第 1 版、第 3 版というタイトルの人気の本から参考にしました。

このチュートリアルでは、読み手が GNU GCC コンパイラを使用して C コードを編集してコンパイルすることで、公開鍵暗号化の概念に精通していることを前提としています。UNIX ネットワークプログラミング本へのアクセスはこのチュートリアルには必要ありません。

11.1.1 このチュートリアルで使用されている例

- EchoClient - 図 5.4、124 ページ
- EchoServer - 図 5.12、139 ページ

11.2 SSL/TLS のクイックサマリー

** TLS ** (トランスポート層セキュリティ) と ** SSL ** (Secure Sockets Layer) は、さまざまなトランスポートプロトコルにわたって安全な通信を可能にする暗号プロトコルです。使用されるプライマリトランスポートプロトコルは TCP/IP です。SSL/TLS の最新バージョンは TLS 1.3 です。wolfSSL は、DTLS 1.0 と 1.2 に加えて、SSL 3.0、TLS 1.0,1.1,1.2,1.3 をサポートしています。

SSL と TLS は、OSI モデルの輸送層とアプリケーション層の間にあり、そこでは任意の数のプロトコル (TCP/IP、Bluetooth などを含む) が基礎となる輸送媒体として機能する場合があります。アプリケーションプロトコルは SSL の上に階層化されており、HTTP、FTP、SMTP などのプロトコルを含めることができます。SSL が OSI モデルにどのように適合するかの図と、SSL ハンドシェイクプロセスの簡単な図は、付録 A にあります。

11.3 ソースコードを取得します

このチュートリアルで使用されているすべてのソースコードは、特に次の場所から wolfSSL の Web サイトからダウンロードできます。このダウンロードには、このチュートリアルで使用されている EchoServer と EchoClient の両方の元のソースコードと完成したソースコードが含まれています。特定の内容がリンクの下にリストされています。

https://www.wolfssl.com/documentation/ssl-tutorial-2.3.zip

ダウンロードした zip ファイルには次のような構造があります。

```
/finished_src
    /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 基本変形例

このチュートリアルとそれに付随するソースコードは、プラットフォーム全体で可能な限りポータブルになるように設計されています。このため、また SSL と TLS をアプリケーションに追加する方法に焦点を合わせたいため、ベースの例は可能な限り単純に保たれています。不必要な複雑さを削除するか、サポートされているプラットフォームの範囲を増やすために、UNIX ネットワークプログラミングから取得した例にいくつかの変更が加えられています。このチュートリアルの移植性を高めるためにできることがあると思われる場合は、support@wolfssl.comまでお知らせください。

以下は、上記の本に記載されている元の EchoServer と EchoClient な例に加えられた修正のリストです。

11.4.1 EcheServer への変更 (tcpserv04.c)

- fork() が Windows でサポートされていないため、Fork() 関数への呼び出しを削除しました。この結果は、1 つのクライアントのみを同時に受け入れる EchoServer です。この削除に加えて、信号処理が削除されました。
- str_echo() 機能を str_echo.c ファイルから tcpserv04.c ファイルに移動しました
- Printf ステートメントを追加して、クライアントアドレスと私たちが接続したポートを表示します。

  ```
  printf("Connection from %s, port %d\n",
          inet_ntop(AF_INET, &cliaddr.sin_addr, buff, sizeof(buff)),
          ntohs(cliaddr.sin_port));
  ```

- リスニングソケットを作成した後、setsockopt() に呼び出しを追加して、「既に使用されているアドレス」バインドエラーを排除しました。
- 新しいコンパイラの警告をクリーンアップするためのマイナーな調整

11.4.2 EchoClient(tcpcli01.c) の変更

- str_cli() 機能を str_cli.c ファイルから tcpcli01.c ファイルに移動しました
- 新しいコンパイラの警告をクリーンアップするためのマイナーな調整

11.4.3 Unp.h ヘッダーへの変更

- このヘッダーは、この例に必要なもののみを含むように簡素化されました。

これらのソースコードの例では、特定の関数が大文字になっていることに注意してください。たとえば、Fputs() と Writen() の作者は、エラーチェックをきれいに処理するために、通常の関数にカスタムラッパー関数を書いています。これについてのより詳細な説明は、_unix ネットワークプログラミング _ 書籍の ** 1.4 **(11 ページ) を参照してください。

11.5 Wolfssl のビルドとインストール

開始する前に、上記のソースコードを取得しますセクションからサンプルコード (echoserver および echoclient) をダウンロードしてください。このセクションでは、システムに wolfSSL 埋め込み SSL ライブラリをダウンロード、構成、インストールする方法について説明します。

Wolfssl ダウンロードページから最新バージョンの wolfSSL をダウンロードしてインストールする必要があります。

利用可能なビルドオプションの完全なリストについては、「Building Wolfsslガイド」を参照してください。Wolfssl は移植性を念頭に置いて書かれており、ほとんどのシステムでビルドするのは一般的に簡単なべきです。Wolfssl をビルドするのが難しい場合は、Wolfssl 製品サポートフォーラムでのサポートをお気軽にお問い合わせください。

Linux、* BSD、OS X、Solaris、またはその他の * NIX システムの wolfSSL をビルドするときは、AutoConf システムを使用できます。Windows 固有の指示については、wolfSSL マニュアルのBuilding Wolfsslセクションを参照してください。wolfSSL を設定してビルドするには、端末から次の 2 つのコマンドを実行します。任意のビルドオプションを./configure に追加することができます (ex：./configure --enable-opensslextra)。

```
./configure
make
```

wolfssl をインストールするには、実行してください。

```
sudo make install
```

これにより、wolfSSL ヘッダーは/usr/local/include/wolfssl に、wolfSSL ライブラリはシステム上の/usr/local/lib にインストールされます。ビルドをテストするには、wolfSSL ルートディレクトリから TestSuite アプリケーションを実行します。

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

一連のテストが WolfCrypt と wolfSSL で実行され、正しくインストールされていることが確認されます。TestSuite アプリケーションの実行が成功した後、次のような出力が表示されるはずです。

```
MD5      test passed!
SHA      test passed!
SHA-224  test passed!
SHA-256  test passed!
SHA-384  test passed!
SHA-512  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!
GMAC     test passed!
Chacha   test passed!
POLY1305 test passed!
ChaCha20-Poly1305 AEAD test passed!
AES      test passed!
AES-GCM  test passed!
RANDOM   test passed!
RSA      test passed!
DH       test passed!
ECC      test passed!
SSL version is TLSv1.2
```

```
SSL cipher suite is TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
SSL version is TLSv1.2
SSL cipher suite is TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
Client message: hello wolfssl!
Server response: I hear you fa shizzle!
sending server shutdown command: quit!
client sent quit command: shutting down!
ciphers=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-
    N0Xq9c

All tests passed!
```

wolfSSL がインストールされたので、SSL 機能を追加するためにサンプルコードの変更を開始できます。まず、SSL を EchoClient に追加し、その後 EchoServer に移動することから始めます。

11.6 初期コンパイル

SSL チュートリアルソースバンドルから EchoClient および EchoServer コードの例をコンパイルして実行するには、含まれているメイクファイルを使用できます。ディレクトリ (CD) を EchoClient または EchoServer ディレクトリに変更して実行します。

```
make
```

これにより、サンプルコードがコンパイルされ、ビルドされているものに応じて、EchoServer または EchoClient という名前の実行可能ファイルが作成されます。MakeFile で使用される GCC コマンドを以下に示します。付属の MakeFile を使用せずに例のいずれかを作成する場合は、ディレクトリをサンプルディレクトリに変更し、次のコマンドで `tcpcli01.c`(echoclient) または `tcpserv04.c`(echoserver) を置き換えます。例の正しいソースファイルを使用して：

```
gcc -o echoserver ../lib/*.c tcpserv04.c -I ../include
```

これにより、現在の例が実行可能ファイルにコンパイルされ、「EchoServer」または「EchoClient」アプリケーションのいずれかが作成されます。コンパイルされた後に例の 1 つを実行するには、現在のディレクトリを目的の例ディレクトリに変更してアプリケーションを開始します。たとえば、Echoserver の使用を開始するには：

```
./echoserver
```

ローカルホストの EchoClient をテストするために 2 番目のターミナルウィンドウを開くことができ、アプリケーションを起動するときにサーバーの IP アドレスを指定する必要があります。これは私たちの場合は 127.0.0.1 になります。現在のディレクトリを “EchoClient” ディレクトリに変更して、次のコマンドを実行します。エコーサーバーはすでに実行されている必要があります。

```
./echoclient 127.0.0.1
```

EchoServer と EchoClient の両方の実行を行うと、EchoServer は EchoClient から受け取る入力をエコーバックする必要があります。EchoServer または EchoClient のいずれかを終了するには、_CTRL + C_ を使用してアプリケーションを終了します。現在、これら 2 つの例の間でやり取りされるデータは平文で送信

されており、少しのスキルがあれば誰でも簡単にクライアントとサーバーの間に自分自身を挿入して通信を聞くことができます。

11.7 ライブラリ

wolfSSL ライブラリは、一度コンパイルされると libwolfssl という名前であり、特に構成されていない限り、wolfSSL ビルドおよびインストールプロセスは、次のディレクトリの下に共有ライブラリのみを作成します。適切なビルドオプションを使用して、共有ライブラリと静的ライブラリの両方を有効または無効にすることができます。

/usr/local/lib

私たちがする必要がある最初のステップは、wolfSSL ライブラリを私たちのサンプルアプリケーションにリンクすることです。GCC コマンドの変更 (例として EchoServer を使用して)、次の新しいコマンドを参照してください。wolfSSL は標準的な場所にヘッダーファイルとライブラリをインストールするため、コンパイラは明示的な命令なしでそれらを見つけることができるはずです (-l または-L を使用)。-lwolfssl を使用することで、コンパイラは自動的に正しいタイプのライブラリ (静的または共有) を選択します。

```
gcc -o echoserver ../lib/*.c tcpserv04.c -I ../include -lm -lwolfssl
```

11.8 ヘッダー

最初に行う必要があるのは、クライアントとサーバーの両方に wolfSSL ネイティブ API ヘッダーを含めることです。クライアントの tcpcli01.c ファイルとサーバーの tcpserv04.c ファイルで、上部近くに次の行を追加します。

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

11.9 起動/シャットダウン

コードで wolfSSL を使用する前に、ライブラリーと WOLFSSL_CTX を初期化する必要があります.wolfSSL はwolfSSL_Init()を呼び出して初期化されます。

WOLFSSL_CTX 構造 (wolfSSL コンテキスト) には、証明書情報を含む各 SSL 接続のグローバル値が含まれています。単一の WOLFSSL_CTX は、作成された WOLFSSL オブジェクトの任意の数で使用できます。これにより、信頼できる CA 証明書のリストなど、特定の情報を 1 回だけ読み込むことができます。

新しい WOLFSSL_CTX を作成するには、wolfSSL_CTX_new()の引数として使用できる対応する関数があります。可能なクライアントとサーバーのプロトコルオプションを以下に示します。SSL 2.0 は、数年間不安定であるため、wolfSSL によってサポートされていません。

エコークライアント：

- 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
- wolfSSLv23_client_method(); - SSLv3 から可能な最高版を使用する - TLS 1.2
- wolfDTLSv1_client_method(); -DTLS 1.0
- wolfDTLSv1_2_client_method_ex(); -DTLS 1.2

エコーサーバー：

- wolfSSLv3_server_method(); - SSLv3

- wolfTLSv1_server_method(); - TLSv1
- wolfTLSv1_1_server_method(); -TLSV1.1
- wolfTLSv1_2_server_method(); -TLSV1.2
- wolfSSLv23_server_method();-クライアントが TLSV1+ に接続できるようにします
- wolfDTLSv1_server_method(); - DTLS.
- wolfDTLSv1_2_server_method(); -DTLS 1.2

EchoClient が EchoServer に接続すると、CA(認証局) 証明書を WOLFSSL_CTX にロードする必要があります。サーバーの身元を確認することができます。CA 証明書を WOLFSSL_CTX にロードするには、wolfSSL_CTX_load_verify_locations()関数は SSL_SUCCESS または SSL_FAILURE のいずれかを返します。

```
wolfSSL_CTX_load_verify_locations(WOLFSSL_CTX* ctx, const char* file, const
↪  char* path)
```

これらのものをまとめる (ライブラリの初期化、プロトコルの選択、CA 証明書)、次のことがあります。ここでは、TLS 1.2 を使用することを選択します。

エコーコイエント：

```
  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);
  }
```

エコーサーバー：

WOLFSSL_CTX に証明書をロードする場合、CA 証明書に加えてサーバー証明書とキーファイルをロードする必要があります。これにより、サーバーは識別検証のためにクライアントに証明書を送信できます。

```
  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 CYASSL_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);
    }
```

上記のコードは、変数定義の両方が IP アドレス (クライアント) を持つクライアントを起動したチェックの後、tcpcli01.c および tcpserv04.c の最初に追加されるべきです。完成したコードのバージョンは、参照用の SSL チュートリアル ZIP ファイルに含まれています。

WOLFSSL と WOLFSSL_CTX が初期化されているので、アプリケーションが SSL/TLS を使用して完全に行われたときに WOLFSSL_CTX オブジェクトと wolfSSL ライブラリが解放されていることを確認してください。クライアントとサーバーの両方で、main() 関数の最後に次の 2 行を配置する必要があります (クライアント内のクライアント内の exit() まで)。

```
wolfSSL_CTX_free(ctx);
wolfSSL_Cleanup();
```

11.10 wolfssl オブジェクト

11.10.1 ech

各 TCP Connect の後に wolfSSL オブジェクトを作成する必要があり、ソケットファイル記述子をセッションに関連付ける必要があります。EchoClient の例では、Connect() への呼び出し後にこれを行います。

```
/* Connect to socket file descriptor */
Connect(sockfd, (SA *) &servaddr, sizeof(servaddr));
```

接続直後に、wolfSSL_new()関数を使用して新しい WOLFSSL オブジェクトを作成します。ここの関数は、成功した場合は WOLFSSL オブジェクトへのポインタを返し、失敗した場合は NULL を返します。その後、ソケットファイル記述子 (sockfd) を新しい WOLFSSL オブジェクト (ssl) に関連付けることができます。

```
/* 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);
```

ここで注目すべきことの 1 つは、wolfSSL_connect()を呼びません。

11.10.2 EchoServer

メインメソッドの for ループの最後に、wolfSSL オブジェクトを挿入し、クライアントと同様に、ソケットファイル記述子 (connfd) を WOLFSSL オブジェクト (ssl) に関連付けます。

```
/* 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);
```

各 TCP Connect の後に wolfSSL オブジェクトを作成する必要があり、ソケットファイル記述子をセッションに関連付ける必要があります。

wolfSSL_new()関数を使用して、新しい wolfSSL オブジェクトを作成します。この関数は、成功した場合は WOLFSSL オブジェクトへのポインタを返し、失敗した場合は NULL を返します。その後、ソケットファイル記述子 (sockfd) を新しい WOLFSSL オブジェクト (ssl) に関連付けることができます。

```
/* 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);
```

11.11 データの送信/受信

11.11.1 EchoClient で送信します

次のステップは安全にデータの送信を開始することです。echoclient の例では、main() 関数が送受信作業を str_cli() に渡すことに注意してください。str_cli() 関数は、私たちの関数で置き換えが行われます。最初に str_cli() のオブジェクトにアクセスする必要があるため、別の引数を追加して SSL 変数を str_cli() に渡します.WOLFSSL オブジェクトは str_cli() 関数の内部で使用されるため、sockfd パラメーターを削除します。この変更後の新しい str_cli() 関数シグネチャを以下に示します。

```
void str_cli(FILE *fp, WOLFSSL* ssl)
```

main() 関数では、新しい引数 (ssl) が str_cli() に渡されます。

```
str_cli(stdin, ssl);
```

str_cli() の内部では、Writen() と Readline() の内側にwolfSSL_write()へのコールが成功したかどうかを確認する必要があることに注意してください。

UNIX プログラミングブックの著者は、それが交換された後に補う必要がある Writen() 関数にエラーチェックを書きました。New Int 変数 n を追加し、戻り値wolfSSL_readを監視し、バッファの内容を印刷する前に Recvline、Read データの終わりに\0 がマークされています。

```
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);
    }
}
```

最後に行うことは、WOLFSSL オブジェクトの使用が完全に終わったら、オブジェクトを解放することです。main() 関数では、WOLFSSL_CTX を解放する前の行の直前に、wolfSSL_free()を呼び出します。

```
str_cli(stdin, ssl);

wolfSSL_free(ssl);      /* Free WOLFSSL object */
wolfSSL_CTX_free(ctx);  /* Free WOLFSSL_CTX object */
wolfSSL_Cleanup();      /* Free wolfSSL */
```

11.11.2 EchoServer で受信します

エコー サーバーは読み書きを処理するために str_echo() を呼び出します (一方、クライアントは str_cli() を呼び出します)。クライアントと同様に、sockfd パラメーターを関数シグネチャの WOLFSSL オブジェクト (ssl) パラメーターに置き換えて、str_echo() を変更します。

```
void str_echo(WOLFSSL* ssl)
```

wolfSSL_read()への変更に対応するために、変数 n のタイプは ssize_t から int に変更されたことに注意してください。

```
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");
}
```

main() では、FOR ループの最後に str_echo() 関数を呼び出します (while ループに変更されます)。この関数の後、ループ内で、WOLFSSL オブジェクトを解放して CONNFD ソケットを閉じるように呼び出します。

```
str_echo(ssl);                    /* process the request */

wolfSSL_free(ssl);                /* Free WOLFSSL object */
Close(connfd);
```

呼び出しの前に ctx とクリーンアップを解放します。

11.12 シグナル処理

11.12.1 エコリエント /EchoServer

ecoclient および echeCoserver では、「Ctrl+C」を使用してユーザーがアプリを閉じるときの信号ハンドラーを追加する必要があります。Echo サーバーは、ループで継続的に実行されています。このため、ユーザーが「Ctrl+C」を押すと、そのループを抜け出す方法を提供する必要があります。これを行うには、最初に行う必要があることは、exit 変数 (クリーンアップ) が true に設定されたときに終了する場合に、ループを時間ループに変更することです。

まず、#include ステートメントの直後に、tcpserv04.c の上部にあるクリーンアップという新しい静的 INT 変数を定義します。

```
static int cleanup;  /* To handle shutdown */
```

EchoServer ループを for ループからしばらくのループに変更して、EchoServer ループを変更します。

```
while(cleanup != 1)
{
    /* echo server code here */
}
```

EchoServer の場合、ハンドラが終了した後に信号が処理される前に実行されていたコールの再起動からオペレーティングシステムを無効にする必要があります。これらを無効にすると、信号が処理された後、オペレーティングシステムは accept() への呼び出しを再開しません。これをしなかった場合は、EchoServer がリソースを整理して終了する前に、他のクライアントが接続して切断するのを待つ必要があります。シグナルハンドラを定義して SA_RESTART をオフにするには、最初に EchoServer の main() 関数内の ACT と OACT 構造を定義します。

```
struct sigaction     act, oact;
```

メイン関数のwolfSSL_Init()への呼び出しの前に、変数宣言の後に次のコードを挿入します。

```
/* 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         */
```

Echoserver の Sig_Handler 関数を以下に示します。

```
void sig_handler(const int sig)
{
    printf("\nSIGINT handled.\n");
    cleanup=1;
    return;
}
```

それだけです - EchoClient と EchoServer は、TLSV1.2 で有効になりました!!

我々のしたこと：

- wolfSSL のヘッダーをインクルードします
- wolfSSL を初期化します
- 使用したいプロトコルを選択した WOLFSSL_CTX 構造を作成しました
- データの送信と受信に使用する WOLFSSL オブジェクトを作成しました
- wolfSSL_write()に Writen() および Readline() に通話を置き換えました
- WOLFSSL 及び WOLFSSL_CTX を開放します
- シグナルハンドラでクライアントとサーバーのシャットダウンを処理していることを確認してください

SSL 接続の動作を設定および制御するための多くの局面や方法があります。詳細については、追加の wolfSSL のマニュアルとリソースを参照してください。

もう一度、完成したソースコードは、このセクションの上部にあるダウンロードした ZIP ファイルにあります。

11.13 証明書

テストのために、wolfSSL が提供する証明書を使用できます。これらは wolfSSL のダウンロードに記載されており、特にこのチュートリアルについては、finished_src フォルダーにあります。

生産アプリケーションの場合、信頼できる証明書当局から正しい正当な証明書を取得する必要があります。

11.14 結論

このチュートリアルは、wolfSSL 組み込み SL ライブラリを簡単なクライアントおよびサーバーアプリケーションに統合するプロセスを進めました。この例は単純ですが、SSL または TLS を独自のアプリケーションに追加するために同じ原則を適用することができます。wolfSSL 組み込み SSL ライブラリは、サイズと速度の両方で最適化されたコンパクトで効率的なパッケージで必要なすべての機能を提供します。

GPLV2 および標準的な商用ライセンスの下でデュアルライセンスを取得しているため、wolfSSL ソースコードを当社の Web サイトから直接ダウンロードできます。サポートフォーラム (https://www.wolfssl.com/forums) にお気軽に投稿してください。当社の製品の詳細については、info@wolfssl.comにお問い合わせください。

この SSL チュートリアルにあるフィードバックを歓迎します。より便利、理解しやすい、またはよりポータブルにするために、改善または強化できると思われる場合は、support@wolfssl.comでお知らせください。

12 組み込みデバイスのベストプラクティス

12.1 プライベートキーの作成

秘密キーをファームウェアに埋め込むことで、誰でもキーを抽出し、それ以外の場合は安全な接続を TCP よりも安全なものに変えることができます。

SSL 対応デバイス用のプライベートキーの作成について、いくつかのアイデアがあります。

1. サーバーとして機能する各デバイスは、非組み込みの世界のように、ユニークな秘密鍵を持つ必要があります。
2. 配信前にキーをデバイスに配置できない場合は、セットアップ中に生成されます。
3. セットアップ中にデバイスに独自のキーを生成する機能がない場合は、デバイスをセットアップしているクライアントにキーを生成させ、それをデバイスに送信してもらいます。
4. クライアントに秘密キーを生成する機能がない場合、クライアントに、既知の Web サイト (たとえば) から SSL/TLS 接続を介して一意の秘密キーを取得します。

wolfSSL(以前の Cyassl) は、これらすべてのステップで使用して、組み込みデバイスに安全な一意の秘密キーを確保するのに役立ちます。これらのステップを踏むことは、SSL 接続自体の保護に大いに役立ちます。

12.2 wolfSSL によるデジタル署名と認証

wolfSSL は、組み込みデバイス上でそれらをロードする前に、アプリケーション、ライブラリ、またはファイルにデジタル署名するための一般的なツールです。ほとんどのデスクトップおよびサーバー オペレーティング システムでは、システム ライブラリを介してこのタイプの機能を作成できますが、機能を取り除いた組み込みオペレーティング システムでは作成できません。Embedded RTOS 環境にデジタル署名機能が含まれていないため、歴史的にほとんどの組み込みアプリケーションが必要ではなかったためです。Connected Devices の世界では、セキュリティ上の懸念を高め、埋め込みまたはモバイルデバイスにロードされているものにデジタル署名しています。

この要件が過去に見つからなかった組み込み接続されたデバイスの例には、セットトップボックス、DVR、POIP システム、VoIP および携帯電話、接続ホーム、さらには自動車ベースのコンピューティングシステムが含まれます。wolfSSL は、主要な組み込みおよびリアルタイムオペレーティングシステム、暗号化標準、および認証機能をサポートしているため、デジタル署名機能を追加する際に組み込みシステム開発者が使用することは自然な選択です。

一般に、埋め込みデバイスでコードとファイルの署名を設定するプロセスは次のとおりです。

1. 組み込みシステム開発者は RSA キーペアを生成します。
2. サーバー側のスクリプトベースのツールが開発されています

1. サーバー側のツールは、デバイスにロードされるコードのハッシュを作成します (たとえば SHA-256 を使用)。2. その後、ハッシュがデジタルで署名され、RSA プライベート暗号化とも呼ばれます。3. デジタル署名とともにコードを含むパッケージが作成されます。

3. パッケージは、RSA の公開キーを取得する方法とともに、デバイスにロードされています。ハッシュは、デバイスに再作成され、既存のデジタル署名に対してデジタル検証 (RSA public Decrypt とも呼ばれます)。

デバイスでデジタル署名を有効にするための利点は次のとおりです。

1. サードパーティがファイルをデバイスにロードできるようにするための安全な方法を簡単に有効にします。
2. 悪意のあるファイルがデバイスに侵入するのを防ぎます。
3. デジタルで安全なファームウェア アップデート

4. 権限のない第三者によるファームウェアの更新を防ぐ

コード署名に関する一般情報：

https://en.wikipedia.org/wiki/code_signing

13 OpenSSL 互換性

13.1 OpenSSL との互換性

wolfSSL(以前の Cyassl) は、wolfSSL ネイティブ API に加えて、OpenSSL 互換性ヘッダー wolfssl/openssl/ssl.h を提供し、wolfSSL の使用を容易にしたり、既存の OpenSSL アプリケーションを wolfSSL に移植するのを支援します。OpenSSL 互換性レイヤーの概要については、以下をご覧ください。wolfSSL がサポートする OpenSSL 関数の完全なセットを表示するには、wolfssl/openssl/ssl.h ファイルを参照してください。

OpenSSL 互換性レイヤーは、wolfSSL のネイティブ API 関数に最も一般的に使用されている OpenSSL コマンドのサブセットをマップします。これにより、多くのコードを変更せずに、アプリケーションまたはプロジェクトで OpenSSL を OpenSSL に簡単に交換できるようになります。

OpenSSL 互換性のための私達のテストベッドは Stunnel と Lighttpd です。これは、OpenSSL 互換性 API をテストする方法として Wolfssl を使ってそれらの両方をビルドすることを意味します。

互換性レイヤーを備えた wolfSSL のビルド：

1. (--enable-opensslextra) またはマクロ OPENSSL_EXTRA を定義することによって有効にします。

   ```
   ./configure --enable-opensslextra
   ```

2. 最初の wolfSSL ヘッダーとして <wolfssl/options.h> を含めます
3. 移行のヘッダーファイルは以下にあります。

- ./wolfssl/openssl/*.h * 例：<wolfssl/openssl/ssl.h>

13.2 wolfSSL と OpenSSL の違い

多くの人々は、wolfSSL が OpenSSL とどのように比較され、組み込みプラットフォームで実行するように最適化された SSL/TLS ライブラリを使用する利点があるかについて興味があります。明らかに、OpenSSL は無料で、使用を開始するための最初のコストはありませんが、wolfSSL はあなたの柔軟性、SSL/TLS の既存のプラットフォームへのより簡単な統合、現在の標準サポート、その他の多くの機能を非常に使いやすいライセンスモデルで提供します。

以下の点は、wolfSSL と OpenSSL の間の主な違いのいくつかを概説します。

1. wolfSSL のビルドサイズは大体 20〜100kB で、条件がそろえば OpenSSL の 20 分の 1 ほどの大きさになります。wolfSSL はリソースに制約の厳しい環境に適した選択肢です。
2. Wolfssl は、DTLS を使用した TLS 1.3 の最新規格に対応しています。wolfSSL チームは、wolfSSL を現在の標準に合わせて継続的に最新の状態に保つことに専念しています。
3. wolfSSL は、ストリーミングメディアサポートのための暗号を含む、今日利用可能な最高の現在の暗号と標準を提供しています。さらに、最近導入された LIBOQS 統合により、Quantum 後の暗号化の実験を開始できます。
4. wolfSSL は、GPLV2 と商業ライセンスの両方でデュアルライセンスされており、OpenSSL は複数のソースからの独自のライセンスの下でのみ利用可能です。
5. wolfSSL は、そのユーザーを気にかけている優れた会社と彼らのセキュリティについて、そして常に助けを喜んでいます。チームは、wolfSSL の改善と拡張に積極的に取り組んでいます。wolfSSL チームは、主にモンタナ州ボーズマン、オレゴン州ポートランド、ワシントン州シアトルを拠点としており、世界中にいる他のチーム メンバーもいます。
6. wolfSSL は、実際のプラットフォームサポートと組み込み環境上の実装の成功により、リアルタイム、モバイル、および組み込みシステムのための主要な SSL/TLS ライブラリです。すでにあなたの環境に

移植されている可能性があります。そうでない場合はお知らせください。喜んでお手伝いさせていただきます。

7. wolfSSL は、できるだけ簡単に環境やプラットフォームに SSL を統合するためのいくつかの抽象化レイヤーを提供しています。OOS レイヤー、カスタム I/O レイヤー、および C 標準ライブラリ抽象化レイヤーにより、統合がこれまでになく簡単になりました。
8. wolfSSL は、wolfSSL 用のいくつかのサポートパッケージを提供しています。電話、電子メール、または wolfSSL 製品サポートフォーラムを介して直接利用できるように、あなたのプロジェクトをできるだけ早くあなたのプロジェクトを進歩させるのを助けるためにすばやく正確に答えられます。

13.3 サポートされている OpenSSL 構造

- SSL_METHOD は SSL バージョン情報を保持し、クライアントメソッドまたはサーバーのいずれかです。(ネイティブ wolfSSL API の WOLFSSL_METHOD と同じ)。
- SSL_CTX 証明書を含むコンテキスト情報を保持します。(ネイティブ wolfSSL API の WOLFSSL_CTX と同じ)。
- SSL 安全な接続のためにセッション情報を保持します。(ネイティブ wolfSSL API の WOLFSSL と同じ)。

13.4 サポートされている OpenSSL 関数

上記の 3 つの構造は、通常、次の方法で初期化されます。

```
SSL_METHOD* method=SSLv3_client_method();
SSL_CTX* ctx=SSL_CTX_new(method);
SSL* ssl=SSL_new(ctx);
```

これはクライアント側の SSL バージョン 3 メソッドを確立し、メソッドに基づいてコンテキストを作成し、コンテキストと SSL セッションを初期化します。SSL_METHOD が SSLv3_server_method()、または利用可能な機能の 1 つを使用して作成されることを除いて、サーバーサイドプログラムは変わりません。サポートされている機能のリストについては、プロトコルサポートセクションを参照してください。OpenSSL 互換層を使用する場合、このセクションの関数は “wolf” プレフィックスを削除することで変更する必要があります。たとえば、ネイティブの wolfSSL API 関数です。

```
wolfTLSv1_client_method()
```

次のようになります

```
TLSv1_client_method()
```

SSL 接続が不要になった場合、次の呼び出しは初期化中に作成された構造を無料で解放します。

```
SSL_CTX_free(ctx);
SSL_free(ssl);
```

SSL_CTX_free() には、関連する SSL_METHOD を解放するという追加の責任があります。XXX_free() 関数の使用に失敗すると、リソースリークが発生します。SSL のものの代わりにシステムの free() を使用すると、未定義の動作が生じます。

アプリケーションが SSL_new() から有効な SSL ポインタを持つと、SSL ハンドシェイクプロセスが開始されます。クライアントのビューから、SSL_connect() は安全な接続を確立しようとします。

```
SSL_set_fd(ssl, sockfd);
SSL_connect(ssl);
```

SSL_connect() を発行する前に、ユーザーは上記の例で有効なソケットファイル記述子、SOCKFD を wolfSSL に提供する必要があります。sockfd は通常、TCP socket() の結果であり、後に TCP connect()

を使用して確立されます。以下は、ポート 11111 でローカル wolfSSL サーバーで使用する有効なクライアントサイドソケット記述子を作成します。

```
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));
```

接続が確立されると、クライアントはサーバーに読み書きすることができます。TCP 関数 send() および receive() を使用する代わりに、wolfSSL および YASSL は SSL 関数 SSL_write() および SSL_read() を使用します。ここではクライアントデモからの簡単な例です。

```
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);
```

サーバーは同じ方法で接続しますが、TCP API に類似した SSL_connect() の代わりに SSL_accept() を使用することを除きます。完全なサーバーデモプログラムについては、サーバーの例を参照してください。

13.5 X509 証明書

サーバーとクライアントの両方が、wolfSSL に ** pem ** または ** der ** のいずれかの証明書を提供できます。

典型的な使用法は次のようなものです。

```
SSL_CTX_use_certificate_file(ctx, "certs/cert.pem",
SSL_FILETYPE_PEM);
SSL_CTX_use_PrivateKey_file(ctx, "certs/key.der",
SSL_FILETYPE_ASN1);
```

キーファイルは、どちらの形式でもコンテキストに表示することもできます。SSL_FILETYPE_PEM は、ファイルがフォーマットされていることを意味し、SSL_FILETYPE_ASN1 はファイルが DER 形式であると宣言します。キーファイルが証明書で使用するのに適していることを確認するために、次の関数を使用できます。

```
SSL_CTX_check_private_key(ctx);
```


14 ライセンス

14.1 オープンソース

wolfSSL、wolfCrypt、wolfMQTT、wolfTPM、wolfBoot、および wolfSentry は無料のソフトウェア ダウンロードであり、ユーザーが GPL ライセンスのバージョン 2 に準拠している限り、ユーザーのニーズに合わせて変更できます。GPLv2 ライセンスは、gnu.org の Web サイト https://www.gnu.org/licenses/old-licenses/gpl-2.0.html にあります。

wolfSSH ソフトウェアは無料のソフトウェアのダウンロードであり、ユーザーが GPL ライセンスの 3 つのバージョン 3 に準拠している限り、ユーザーのニーズに変更される可能性があります。GPLV3 ライセンスは、gnu.org の Web サイト (https://www.gnu.org/licenses/gpl.html) にあります。

14.2 商業用ライセンス

再配布のために wolfSSL 製品を独自のアプライアンスまたはその他の商用ソフトウェア製品に組み込みたい企業は、商用バージョンのライセンスを取得する必要があります。wolfSSL と wolfCrypt の商用ライセンスは、最終製品または SKU ごとに利用できます。通常、ライセンスは 1 つの製品に対して発行され、無制限のロイヤリティ フリーの配布が含まれます。カスタム ライセンス条項も利用できます。

wolfMQTT、wolfSSH、wolfTPM、wolfBoot、wolfSentry の商用ライセンスも利用できます。お問い合わせはlicensing@wolfssl.comまでお願いします。

14.3 FIPS 140-2/3 検証

wolfSSL は現在、組み込み FIPS 証明書のリーダーです。現在アクティブな FIPS 証明書と検証オプションの詳細については、wolfCrypt FIPS FAQ を参照するか、fips@wolfssl.com に連絡してください。wolfSSL.com)。

14.4 サポートパッケージ

wolfSSL 製品のサポート パッケージは、wolfSSL から直接年間ベースで入手できます。4 つの異なるパッケージ オプションを使用して、それらを並べて比較し、特定のニーズに最適なパッケージを選択できます。詳細については、サポート パッケージ ページ (https://www.wolfssl.com/products/support-and-maintenance) をご覧ください。

15　サポートとコンサルティング

15.1　サポートを受ける方法

一般的な製品サポートのために、wolfSSL(以前の Cyassl) は、wolfSSL 製品ファミリーのオンラインフォーラムを維持しています。フォーラムに投稿するか、ご不明な点が表示されます。

- wolfssl(yassl) フォーラム：https://www.wolfssl.com/forums
- 電子メールサポート：support@wolfssl.com

wolfSSL 製品に関する情報、ライセンスに関する質問、または一般的なコメントについては、info@wolfssl.comを E メールにすることで Wolfssl にお問い合わせください。サポートパッケージについては、ライセンスをご覧ください。

15.1.1　バグの報告とサポートの問題

バグレポートを提出したり、問題について尋ねている場合は、次の情報を提出に含めてください。

1. wolfSSL バージョン番号
2. オペレーティングシステムバージョン
3. コンパイラバージョン
4. 表示されている正確なエラー
5. この問題を再現または再現しようとする方法の説明

上記の情報を使用すると、問題を解決するために最善を尽くします。この情報がなければ、問題の原因を特定するのは非常に困難です。wolfSSL はあなたのフィードバックを値し、できるだけ早くあなたに戻るのが最優先事項になります。

15.2　コンサルティング

機能の追加、移植、競争力のあるアップグレードプログラム、およびデザインコンサルティングを提供します。

詳細は info@wolfssl.jp 宛にお問い合わせください。

15.2.1　機能追加と移植

現時点で、ご要望いただいているのに弊社製品で提供されていない機能を、契約または共同開発ベースで追加することができます。また、当社の製品を新しいホスト言語または新しい操作環境に移植するサービスも提供しています。

詳細は info@wolfssl.jp 宛にお問い合わせください。

15.2.2　デザインコンサルティング

あなたのアプリケーションまたはフレームワークを SSL/TLS で保護する必要があるが、あなたは保護されたシステムの最適設計がどのように構造化されるかについて不明なことがわかります。

wolfSSL を使用して、SSL/TLS セキュリティをデバイスにビルドするためのデザインコンサルティングを提供しています。当社のコンサルタントは、次のサービスを提供できます。

1. *Assessment*：現在の SSL/TLS 実装の評価。あなたの現在のセットアップについてアドバイスをすることができます。

2. *Design*：お客様のシステム要件とパラメータを確認し、最適なセキュリティを提供するようにwolfSSL をアプリケーションに実装する方法について、お客様と緊密に連携して推奨事項を作成します。

アプリケーションやデバイスに SSL をビルドするためのデザインコンサルティングについて詳しく知りたい場合は、info@wolfssl.comにお問い合わせください。

16 wolfSSL(以前の Cyassl) の更新

16.1 製品のリリース情報

更新情報を Twitter に定期的に投稿しています。追加のリリース情報については、GitHub でプロジェクトを追跡したり、Facebook でフォローしたり、毎日のブログをフォローしたりできます。

- Github の Wolfssl -https://www.github.com/wolfssl/wolfssl
- Twitter の wolfssl -https://twitter.com/wolfSSL
- Facebook の wolfssl -https://www.facebook.com/wolfSSL
- reddit の wolfssl -https://www.reddit.com/r/wolfssl/
- 毎日のブログ - https://www.wolfssl.com/blog

A　wolfSSL API リファレンス

A.1　CertManager API

A.1.1　Functions

	Name
WOLFSSL_CERT_MANAGER *	**wolfSSL_CertManagerNew_ex**(void * heap) 新しい証明書マネージャコンテキストを割り当てて初期化します。このコンテキストは、SSL のニーズとは無関係に使用できます。証明書をロードしたり、証明書を確認したり、失効状況を確認したりするために使用することができます。
WOLFSSL_CERT_MANAGER *	**wolfSSL_CertManagerNew**(void) 新しい証明書マネージャコンテキストを割り当てて初期化します。このコンテキストは、SSL のニーズとは無関係に使用できます。証明書をロードしたり、証明書を確認したり、失効状況を確認したりするために使用することができます。
void	**wolfSSL_CertManagerFree**(WOLFSSL_CERT_MANAGER *) 証明書マネージャのコンテキストに関連付けられているすべてのリソースを解放します。証明書マネージャを使用する必要がなくなるときにこれを呼び出します。
int	**wolfSSL_CertManagerLoadCA**(WOLFSSL_CERT_MANAGER * cm, const char * f, const char * d)Manager コンテキストへの CA 証明書のロードの場所を指定します。PEM 証明書カフェイルには、複数の信頼できる CA 証明書が含まれている可能性があります。capath が null でない場合、PEM 形式の CA 証明書を含むディレクトリを指定します。
int	**wolfSSL_CertManagerLoadCABuffer**(WOLFSSL_CERT_MANAGER * cm, const unsigned char * in, long sz, int format)wolfssl_ctx_load_verify_buffer を呼び出して、関数に渡された CM 内の情報を失うことなく一時的な CM を使用してその結果を返すことによって CA バッファをロードします。
int	**wolfSSL_CertManagerUnloadCAs**(WOLFSSL_CERT_MANAGER * cm) この関数は CA 署名者リストをアンロードします。
int	**wolfSSL_CertManagerUnload_trust_peers**(WOLFSSL_CERT_MA * cm) 関数は信頼できるピアリンクリストを解放し、信頼できるピアリストのロックを解除します。
int	**wolfSSL_CertManagerVerify**(WOLFSSL_CERT_MANAGER * cm, const char * f, int format) 証明書マネージャのコンテキストで確認する証明書を指定します。フォーマットは SSL_FILETYPE_PEM または SSL_FILETYPE_ASN1 にすることができます。

	Name
int	**wolfSSL_CertManagerVerifyBuffer**(WOLFSSL_CERT_MANAGER * cm, const unsigned char * buff, long sz, int format) 証明書マネージャのコンテキストを使用して確認する証明書バッファを指定します。フォーマットは SSL_FILETYPE_PEM または SSL_FILETYPE_ASN1 にすることができます。
void	**wolfSSL_CertManagerSetVerify**(WOLFSSL_CERT_MANAGER * cm, VerifyCallback vc) この関数は、証明書マネージャーの verifyCallback 関数を設定します。存在する場合、それはロードされた各 CERT に対して呼び出されます。検証エラーがある場合は、検証コールバックを使用してエラーを過度に乗り越えます。
int	**wolfSSL_CertManagerEnableCRL**(WOLFSSL_CERT_MANAGER * cm, int options) 証明書マネージャを使用して証明書を検証するときに証明書失効リストの確認をオンにします。デフォルトでは、CRL チェックはオフです。オプションには、wolfssl_crl_checkall が含まれます。これは、チェーン内の各証明書に対して CRL 検査を実行します。これはデフォルトであるリーフ証明書のみです。
int	**wolfSSL_CertManagerDisableCRL**(WOLFSSL_CERT_MANAGER *) 証明書マネージャを使用して証明書を検証するときに証明書失効リストの確認をオフにします。デフォルトでは、CRL チェックはオフです。この関数を使用して、この Certificate Manager コンテキストを使用して CRL 検査を一時的または恒久的に無効にして、以前は CRL 検査が有効になっていました。
int	**wolfSSL_CertManagerLoadCRL**(WOLFSSL_CERT_MANAGER * cm, const char * path, int type, int monitor) 証明書の失効確認のために証明書を CRL にロードする際にエラーチェックを行い、その後証明書を LoadCRL() へ渡します。
int	**wolfSSL_CertManagerLoadCRLBuffer**(WOLFSSL_CERT_MANAGE * cm, const unsigned char * buff, long sz, int type) この関数は、BufferLoadCRL を呼び出すことによって CRL ファイルをロードします。
int	**wolfSSL_CertManagerSetCRL_Cb**(WOLFSSL_CERT_MANAGER * cm, CbMissingCRL cb) この関数は CRL 証明書マネージャコールバックを設定します。LABE_CRL が定義されていて一致する CRL レコードが見つからない場合、CBMissingCRL は呼び出されます（WolfSSL_CertManagerSetCRL_CB を介して設定）。これにより、CRL を外部に検索してロードすることができます。
int	**wolfSSL_CertManagerFreeCRL**(WOLFSSL_CERT_MANAGER * cm) この関数は証明書マネジャーに保持されている CRL を解放します。アプリケーションは CRL を wolfSSL_CertManagerFreeCRL を呼び出して解放した後に、新しい CRL をロードすることができます。

	Name
int	**wolfSSL_CertManagerCheckOCSP**(WOLFSSL_CERT_MANAGER * cm, unsigned char * der, int sz) この機能により、OCSPENABLED OCSPENABLED が OCSP チェックオプションが有効になっていることを意味します。
int	**wolfSSL_CertManagerEnableOCSP**(WOLFSSL_CERT_MANAGER * cm, int options)OCSP がオフになっている場合は OCSP をオンにし、[設定] オプションを使用可能になっている場合。
int	**wolfSSL_CertManagerDisableOCSP**(WOLFSSL_CERT_MANAGER *)OCSP 証明書の失効を無効にします。
int	**wolfSSL_CertManagerSetOCSPOverrideURL**(WOLFSSL_CERT_M * cm, const char * url) この関数は、URL を wolfssl_cert_manager 構造体の OCSpoverrideURL メンバーにコピーします。
int	**wolfSSL_CertManagerSetOCSP_Cb**(WOLFSSL_CERT_MANAGER * cm, CbOCSPIO ioCb, CbOCSPRespFree respFreeCb, void * ioCbCtx) この関数は、wolfssl_cert_manager の OCSP コールバックを設定します。
int	**wolfSSL_CertManagerEnableOCSPStapling**(WOLFSSL_CERT_MA * cm) この関数は、オプションをオンにしないと OCSP ステープルをオンにします。オプションを設定します。

A.1.2 Functions Documentation

A.1.2.1 function wolfSSL_CertManagerNew_ex

```
WOLFSSL_CERT_MANAGER * wolfSSL_CertManagerNew_ex(
    void * heap
)
```

新しい証明書マネージャコンテキストを割り当てて初期化します。このコンテキストは、SSL のニーズとは無関係に使用できます。証明書をロードしたり、証明書を確認したり、失効状況を確認したりするために使用することができます。

See: wolfSSL_CertManagerFree

Return:

- WOLFSSL_CERT_MANAGER 正常にコールが有効な wolfssl_cert_manager ポインタを返します。
- NULL エラー状態に戻ります。

A.1.2.2 function wolfSSL_CertManagerNew

```
WOLFSSL_CERT_MANAGER * wolfSSL_CertManagerNew(
    void
)
```

新しい証明書マネージャコンテキストを割り当てて初期化します。このコンテキストは、SSL のニーズとは無関係に使用できます。証明書をロードしたり、証明書を確認したり、失効状況を確認したりするために使用することができます。

See: wolfSSL_CertManagerFree

Return:

- WOLFSSL_CERT_MANAGER 正常にコールが有効な wolfssl_cert_manager ポインタを返します。
- NULL エラー状態に戻ります。

Example

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

WOLFSSL_CERT_MANAGER* cm;
cm = wolfSSL_CertManagerNew();
if (cm == NULL) {
// error creating new cert manager
}
```

A.1.2.3 function wolfSSL_CertManagerFree

```
void wolfSSL_CertManagerFree(
    WOLFSSL_CERT_MANAGER *
)
```

証明書マネージャのコンテキストに関連付けられているすべてのリソースを解放します。証明書マネージャを使用する必要がなくなるときにこれを呼び出します。

See: wolfSSL_CertManagerNew

Return: none

Example

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

WOLFSSL_CERT_MANAGER* cm;
...
wolfSSL_CertManagerFree(cm);
```

A.1.2.4 function wolfSSL_CertManagerLoadCA

```
int wolfSSL_CertManagerLoadCA(
    WOLFSSL_CERT_MANAGER * cm,
    const char * f,
    const char * d
)
```

Manager コンテキストへの CA 証明書のロードの場所を指定します。PEM 証明書カフェイルには、複数の信頼できる CA 証明書が含まれている可能性があります。capath が null でない場合、PEM 形式の CA 証明書を含むディレクトリを指定します。

Parameters:

- **cm** wolfssl_certmanagernew() を使用して作成された wolfssl_cert_manager 構造体へのポインタ。
- **file** ロードする CA 証明書を含むファイルの名前へのポインタ。

See: wolfSSL_CertManagerVerify

Return:

- SSL_SUCCESS 成功した場合に返されます。、通話が戻ります。
- SSL_BAD_FILETYPE ファイルが間違った形式である場合に返されます。
- SSL_BAD_FILE ファイルが存在しない場合に返されます。読み込め、または破損していません。
- MEMORY_E メモリ不足状態が発生した場合に返されます。
- ASN_INPUT_E base16 デコードがファイルに対して失敗した場合に返されます。
- BAD_FUNC_ARG ポインタが提供されていない場合に返されるエラーです。

- SSL_FATAL_ERROR - 失敗時に返されます。

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
}
```

A.1.2.5 function wolfSSL_CertManagerLoadCABuffer

```
int wolfSSL_CertManagerLoadCABuffer(
    WOLFSSL_CERT_MANAGER * cm,
    const unsigned char * in,
    long sz,
    int format
)
```

wolfssl_ctx_load_verify_buffer を呼び出して、関数に渡された CM 内の情報を失うことなく一時的な CM を使用してその結果を返すことによって CA バッファをロードします。

Parameters:

- **cm** wolfssl_certmanagernew() を使用して作成された wolfssl_cert_manager 構造体へのポインタ。
- **in** CERT 情報用のバッファー。
- **sz** バッファの長さ。

See:

- wolfSSL_CTX_load_verify_buffer
- ProcessChainBuffer
- ProcessBuffer
- cm_pick_method

Return:

- SSL_FATAL_ERROR wolfssl_cert_manager 構造体が NULL の場合、または wolfSSL_CTX_new() が NULL を返す場合に返されます。
- SSL_SUCCESS 実行が成功するために返されます。

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.
}
```

A.1.2.6 function wolfSSL_CertManagerUnloadCAs

```
int wolfSSL_CertManagerUnloadCAs(
```

```
    WOLFSSL_CERT_MANAGER * cm
)
```

この関数は CA 署名者リストをアンロードします。

See:

- FreeSignerTable
- UnlockMutex

Return:

- SSL_SUCCESS 機能の実行に成功したことに戻ります。
- BAD_FUNC_ARG wolfssl_cert_manager が null の場合に返されます。
- BAD_MUTEX_E ミューテックスエラーが発生した場合に返されます。

Example

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

WOLFSSL_CTX* ctx = wolfSSL_CTX_new(protocol method);
WOLFSSL_CERT_MANAGER* cm = wolfSSL_CertManagerNew();
...
if(wolfSSL_CertManagerUnloadCAs(ctx->cm) != SSL_SUCCESS){
    Failure case.
}
```

A.1.2.7 function wolfSSL_CertManagerUnload_trust_peers

```
int wolfSSL_CertManagerUnload_trust_peers(
    WOLFSSL_CERT_MANAGER * cm
)
```

関数は信頼できるピアリンクリストを解放し、信頼できるピアリストのロックを解除します。

See: UnLockMutex

Return:

- SSL_SUCCESS 関数が正常に完了した場合
- BAD_FUNC_ARG wolfssl_cert_manager が null の場合
- BAD_MUTEX_E ミューテックスエラー TPLOCK では、WOLFSSL_CERT_MANAGER 構造体のメンバーは 0（ニル）です。

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.
}
```

A.1.2.8 function wolfSSL_CertManagerVerify

```
int wolfSSL_CertManagerVerify(
    WOLFSSL_CERT_MANAGER * cm,
    const char * f,
    int format
```

)

証明書マネージャのコンテキストで確認する証明書を指定します。フォーマットは SSL_FILETYPE_PEM または SSL_FILETYPE_ASN1 にすることができます。

Parameters:

- **cm** wolfssl_certmanagernew() を使用して作成された wolfssl_cert_manager 構造体へのポインタ。
- **fname** 検証する証明書を含むファイルの名前へのポインタ。

See:

- wolfSSL_CertManagerLoadCA
- wolfSSL_CertManagerVerifyBuffer

Return:

- SSL_SUCCESS 成功した場合に返されます。
- ASN_SIG_CONFIRM_E 署名が検証できなかった場合に返されます。
- ASN_SIG_OID_E 署名の種類がサポートされていない場合に返されます。
- CRL_CERT_REVOKED この証明書が取り消された場合に返されるエラーです。
- CRL_MISSING 現在の発行者 CRL が利用できない場合に返されるエラーです。
- ASN_BEFORE_DATE_E 現在の日付が前日の前にある場合に返されます。
- ASN_AFTER_DATE_E 現在の日付が後の日付の後の場合に返されます。
- SSL_BAD_FILETYPE ファイルが間違った形式である場合に返されます。
- SSL_BAD_FILE ファイルが存在しない場合に返されます。読み込め、または破損していません。
- MEMORY_E メモリ不足状態が発生した場合に返されます。
- ASN_INPUT_E base16 デコードがファイルに対して失敗した場合に返されます。
- BAD_FUNC_ARG ポインタが提供されていない場合に返されるエラーです。

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
}
```

A.1.2.9 function wolfSSL_CertManagerVerifyBuffer

```
int wolfSSL_CertManagerVerifyBuffer(
    WOLFSSL_CERT_MANAGER * cm,
    const unsigned char * buff,
    long sz,
    int format
)
```

証明書マネージャのコンテキストを使用して確認する証明書バッファを指定します。フォーマットは SSL_FILETYPE_PEM または SSL_FILETYPE_ASN1 にすることができます。

Parameters:

- **cm** wolfssl_certmanagernew() を使用して作成された wolfssl_cert_manager 構造体へのポインタ。
- **buff** 検証する証明書を含むバッファ。
- **sz** バッファのサイズ、BUF。

See:

- wolfSSL_CertManagerLoadCA
- wolfSSL_CertManagerVerify

Return:

- SSL_SUCCESS 成功した場合に返されます。
- ASN_SIG_CONFIRM_E 署名が検証できなかった場合に返されます。
- ASN_SIG_OID_E 署名の種類がサポートされていない場合に返されます。
- CRL_CERT_REVOKED この証明書が取り消された場合に返されるエラーです。
- CRL_MISSING 現在の発行者 CRL が利用できない場合に返されるエラーです。
- ASN_BEFORE_DATE_E 現在の日付が前日の前にある場合に返されます。
- ASN_AFTER_DATE_E 現在の日付が後の日付の後の場合に返されます。
- SSL_BAD_FILETYPE ファイルが間違った形式である場合に返されます。
- SSL_BAD_FILE ファイルが存在しない場合に返されます。読み込め、または破損していません。
- MEMORY_E メモリ不足状態が発生した場合に返されます。
- ASN_INPUT_E base16 デコードがファイルに対して失敗した場合に返されます。
- BAD_FUNC_ARG ポインタが提供されていない場合に返されるエラーです。

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
}
```

A.1.2.10 function wolfSSL_CertManagerSetVerify

```
void wolfSSL_CertManagerSetVerify(
    WOLFSSL_CERT_MANAGER * cm,
    VerifyCallback vc
)
```

この関数は、証明書マネージャーの verifyCallback 関数を設定します。存在する場合、それはロードされた各 CERT に対して呼び出されます。検証エラーがある場合は、検証コールバックを使用してエラーを過度に乗り越えます。

Parameters:

- **cm** wolfssl_certmanagernew() を使用して作成された wolfssl_cert_manager 構造体へのポインタ。

See: wolfSSL_CertManagerVerify

Return: none 返品不可。

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);
```

A.1.2.11 function wolfSSL_CertManagerEnableCRL

```
int wolfSSL_CertManagerEnableCRL(
    WOLFSSL_CERT_MANAGER * cm,
    int options
)
```

証明書マネージャを使用して証明書を検証するときに証明書失効リストの確認をオンにします。デフォルトでは、CRL チェックはオフです。オプションには、wolfssl_crl_checkall が含まれます。これは、チェーン内の各証明書に対して CRL 検査を実行します。これはデフォルトであるリーフ証明書のみです。

Parameters:

- **cm** wolfssl_certmanagernew() を使用して作成された wolfssl_cert_manager 構造体へのポインタ。

See: wolfSSL_CertManagerDisableCRL

Return:

- SSL_SUCCESS 成功した場合に返されます。、通話が戻ります。
- NOT_COMPILED_IN WolfSSL が CRL を有効にして構築されていない場合に返されます。
- MEMORY_E メモリ不足状態が発生した場合に返されます。
- BAD_FUNC_ARG ポインタが提供されていない場合に返されるエラーです。
- SSL_FAILURE CRL コンテキストを正しく初期化できない場合に返されます。

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
}

...
```

A.1.2.12 function wolfSSL_CertManagerDisableCRL

```
int wolfSSL_CertManagerDisableCRL(
    WOLFSSL_CERT_MANAGER *
)
```

証明書マネージャを使用して証明書を検証するときに証明書失効リストの確認をオフにします。デフォルトでは、CRL チェックはオフです。この関数を使用して、この Certificate Manager コンテキストを使用して CRL 検査を一時的または恒久的に無効にして、以前は CRL 検査が有効になっていました。

See: wolfSSL_CertManagerEnableCRL

Return:

- SSL_SUCCESS 成功した場合に返されます。、通話が戻ります。
- BAD_FUNC_ARG 関数ポインタが提供されていない場合に返されるエラーです。

Example

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

int ret = 0;
WOLFSSL_CERT_MANAGER* cm;
...
ret = wolfSSL_CertManagerDisableCRL(cm);
if (ret != SSL_SUCCESS) {
    error disabling cert manager
}
...
```

A.1.2.13 function wolfSSL_CertManagerLoadCRL

```
int wolfSSL_CertManagerLoadCRL(
    WOLFSSL_CERT_MANAGER * cm,
    const char * path,
    int type,
    int monitor
)
```

証明書の失効確認のために証明書を CRL にロードする際にエラーチェックを行い、その後証明書を LoadCRL() へ渡します。

Parameters:

- **cm** wolfSSL_CertManagerNew()を使用して作成された WOLFSSL_CERT_MANAGER 構造体へのポインタ。
- **path** CRL へのパスを保持しているバッファーへのポインタ。
- **type** ロードする証明書の種類。

See:

- wolfSSL_CertManagerEnableCRL
- wolfSSL_LoadCRL

Return:

- SSL_SUCCESS wolfSSL_CertManagerLoadCRL でエラーが発生せず、loadCRL が成功で戻る場合に返されます。
- BAD_FUNC_ARG WOLFSSL_CERT_MANAGER 構造体が NULL の場合
- SSL_FATAL_ERROR wolfSSL_CertManagerEnableCRL が SSL_SUCCESS 以外のを返す場合。
- BAD_PATH_ERROR path が NULL の場合
- MEMORY_E LOADCRL がヒープメモリの割り当てに失敗した場合。

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);
```

A.1.2.14 function wolfSSL_CertManagerLoadCRLBuffer

```
int wolfSSL_CertManagerLoadCRLBuffer(
    WOLFSSL_CERT_MANAGER * cm,
    const unsigned char * buff,
    long sz,
    int type
)
```

この関数は、BufferLoadCRL を呼び出すことによって CRL ファイルをロードします。

Parameters:

- **cm** wolfssl_cert_manager 構造体へのポインタ。
- **buff** 定数バイトタイプとバッファです。
- **sz** バッファのサイズを表す長い int。

See:

- BufferLoadCRL
- wolfSSL_CertManagerEnableCRL

Return:

- SSL_SUCCESS 関数がエラーなしで完了した場合に返されます。
- BAD_FUNC_ARG wolfssl_cert_manager が null の場合に返されます。
- SSL_FATAL_ERROR wolfssl_cert_manager に関連付けられているエラーがある場合に返されます。

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.
}
```

A.1.2.15 function wolfSSL_CertManagerSetCRL_Cb

```
int wolfSSL_CertManagerSetCRL_Cb(
    WOLFSSL_CERT_MANAGER * cm,
    CbMissingCRL cb
)
```

この関数は CRL 証明書マネージャコールバックを設定します。LABE_CRL が定義されていて一致する CRL レコードが見つからない場合、CBMissingCRL は呼び出されます（WolfSSL_CertManagerSetCRL_CB を介して設定）。これにより、CRL を外部に検索してロードすることができます。

Parameters:

- **cm** 証明書の情報を保持している WOLFSSL_CERT_MANAGER 構造。

See:

- CbMissingCRL

- wolfSSL_SetCRL_Cb

Return:

- SSL_SUCCESS 関数とサブルーチンの実行が成功したら返されます。
- BAD_FUNC_ARG wolfssl_cert_manager 構造体が NULL の場合に返されます。

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);
}
```

A.1.2.16 function wolfSSL_CertManagerFreeCRL

```
int wolfSSL_CertManagerFreeCRL(
    WOLFSSL_CERT_MANAGER * cm
)
```

この関数は証明書マネジャーに保持されている CRL を解放します。アプリケーションは CRL を wolfSSL_CertManagerFreeCRL を呼び出して解放した後に、新しい CRL をロードすることができます。

Parameters:

- **cm** wolfSSL_CertManagerNew()で生成された WOLFSSL_CERT_MANAGER 構造体へのポインター。

See: wolfSSL_CertManagerLoadCRL

Return:

- SSL_SUCCESS 関数の実行に成功した場合に返されます。
- BAD_FUNC_ARG WOLFSSL_CERT_MANAGER 構造体へのポインターが NULL で渡された場合に返されます。

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);
```

A.1.2.17 function wolfSSL_CertManagerCheckOCSP

```
int wolfSSL_CertManagerCheckOCSP(
    WOLFSSL_CERT_MANAGER * cm,
```

```
    unsigned char * der,
    int sz
)
```

この機能により、OCSPENABLED OCSPENABLED が OCSP チェックオプションが有効になっていることを意味します。

Parameters:

- **cm** wolfssl_certmanagernew() を使用して作成された wolfssl_cert_manager 構造体へのポインタ。
- **der** 証明書へのバイトポインタ。

See:

- ParseCertRelative
- CheckCertOCSP

Return:

- SSL_SUCCESS 機能の実行に成功したことに戻ります。wolfssl_cert_manager の OCSPENABLED メンバーが有効になっています。
- BAD_FUNC_ARG wolfssl_cert_manager 構造体が null の場合、または許可されていない引数値がサブルーチンに渡された場合に返されます。
- MEMORY_E この関数内にメモリを割り当てるエラーまたはサブルーチンがある場合に返されます。

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.
}
```

A.1.2.18 function wolfSSL_CertManagerEnableOCSP

```
int wolfSSL_CertManagerEnableOCSP(
    WOLFSSL_CERT_MANAGER * cm,
    int options
)
```

OCSP がオフになっている場合は OCSP をオンにし、[設定] オプションを使用可能になっている場合。

Parameters:

- **cm** wolfssl_certmanagernew() を使用して作成された wolfssl_cert_manager 構造体へのポインタ。

See: wolfSSL_CertManagerNew

Return:

- SSL_SUCCESS 関数呼び出しが成功した場合に返されます。
- BAD_FUNC_ARG cm 構造体が null の場合
- MEMORY_E wolfssl_ocsp struct 値が null の場合
- SSL_FAILURE WOLFSSL_OCSP 構造体の初期化は初期化に失敗します。
- NOT_COMPILED_IN 正しい機能を有効にしてコンパイルされていないビルド。

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.
}
```

A.1.2.19 function wolfSSL_CertManagerDisableOCSP

```
int wolfSSL_CertManagerDisableOCSP(
    WOLFSSL_CERT_MANAGER *
)
```

OCSP 証明書の失効を無効にします。

See: wolfSSL_DisableCRL

Return:

- SSL_SUCCESS WolfSSL_CertMangerDisableCRL は、WolfSSL_CERT_MANAGER 構造体の CRLEnabled メンバを無効にしました。
- BAD_FUNC_ARG WOLFSSL 構造はヌルでした。

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.
}
```

A.1.2.20 function wolfSSL_CertManagerSetOCSPOverrideURL

```
int wolfSSL_CertManagerSetOCSPOverrideURL(
    WOLFSSL_CERT_MANAGER * cm,
    const char * url
)
```

この関数は、URL を wolfssl_cert_manager 構造体の OCSpoverrideURL メンバーにコピーします。

See:

- ocspOverrideURL
- wolfSSL_SetOCSP_OverrideURL

Return:

- SSL_SUCCESS この機能は期待どおりに実行できました。
- BAD_FUNC_ARG wolfssl_cert_manager 構造体は null です。
- MEMEORY_E 証明書マネージャの OCSPoverRideURL メンバーにメモリを割り当てることができませんでした。

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.
}
```

A.1.2.21 function wolfSSL_CertManagerSetOCSP_Cb

```
int wolfSSL_CertManagerSetOCSP_Cb(
    WOLFSSL_CERT_MANAGER * cm,
    CbOCSPIO ioCb,
    CbOCSPRespFree respFreeCb,
    void * ioCbCtx
)
```

この関数は、wolfssl_cert_manager の OCSP コールバックを設定します。

Parameters:

- **cm** wolfssl_cert_manager 構造体へのポインタ。
- **ioCb** CBocSpio 型の関数ポインタ。
- **respFreeCb** - CBOCSPRESPFREAS 型の関数ポインタ。

See:

- wolfSSL_CertManagerSetOCSPOverrideURL
- wolfSSL_CertManagerCheckOCSP
- wolfSSL_CertManagerEnableOCSPStapling
- wolfSSL_ENableOCSP
- wolfSSL_DisableOCSP
- wolfSSL_SetOCSP_Cb

Return:

- SSL_SUCCESS 実行に成功したことに戻ります。引数は wolfssl_cert_manager 構造体に保存されます。
- BAD_FUNC_ARG wolfssl_cert_manager が null の場合に返されます。

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);
```

A.1.2.22 function wolfSSL_CertManagerEnableOCSPStapling

```
int wolfSSL_CertManagerEnableOCSPStapling(
    WOLFSSL_CERT_MANAGER * cm
)
```

この関数は、オプションをオンにしないと OCSP ステープルをオンにします。オプションを設定します。

See: wolfSSL_CTX_EnableOCSPStapling

Return:

- SSL_SUCCESS エラーがなく、関数が正常に実行された場合に返されます。
- BAD_FUNC_ARG wolfssl_cert_manager 構造体が NULL またはそうでない場合は、サブルーチンに渡された未解決の引数値があった場合に返されます。
- MEMORY_E メモリ割り当てがある問題が発生した場合に返されます。
- SSL_FAILURE OCSP 構造体の初期化が失敗した場合に返されます。
- NOT_COMPILED_IN wolfssl が haber_certificate_status_request オプションでコンパイルされていない場合に返されます。

Example

```
int wolfSSL_CTX_EnableOCSPStapling(WOLFSSL_CTX* ctx){
…
return wolfSSL_CertManagerEnableOCSPStapling(ctx->cm);
```

A.2 Memory Handling

A.2.1 Functions

	Name
void *	**wolfSSL_Malloc**(size_t size, void * heap, int type) この関数は malloc（）と似ていますが、WolfSSL が使用するように構成されているメモリ割り当て関数を呼び出します。デフォルトでは、WolfSSL は malloc（）を使用します。これは、WolfSSL メモリ抽象化レイヤを使用して変更できます - wolfssl_setAllocator（）を参照してください。注 WOLFSSL_MALLOC は、WOLFSSL によって直接呼び出されませんが、代わりに Macro XMalloc によって呼び出されます。デフォルトのビルドの場合、size 引数のみが存在します。wolfssl_static_memory ビルドを使用する場合は、ヒープとタイプ引数が含まれます。
void	**wolfSSL_Free**(void * ptr, void * heap, int type) この関数は free（）と似ていますが、WolfSSL が使用するように構成されているメモリフリー機能を呼び出します。デフォルトでは、WolfSSL は free（）を使用します。これは、WolfSSL メモリ抽象化レイヤを使用して変更できます - wolfssl_setAllocator（）を参照してください。注 WOLFSSL_FREE は WOLFSSL によって直接呼び出されませんが、代わりにマクロ XFree によって呼び出されます。デフォルトのビルドの場合、PTR 引数のみが存在します。wolfssl_static_memory ビルドを使用する場合は、ヒープとタイプ引数が含まれます。

	Name
void *	**wolfSSL_Realloc**(void * ptr, size_t size, void * heap, int type) この関数は REALLOC（）と似ていますが、WolfSSL が使用するように構成されているメモリ再割り当て機能を呼び出します。デフォルトでは、WolfSSL は RealLoc（）を使用します。これは、WolfSSL メモリ抽象化レイヤを使用して変更できます - wolfssl_setAllocator（）を参照してください。注 WOLFSSL_REALLOC は WOLFSSL によって直接呼び出されませんが、代わりにマクロ Xrealloc によって呼び出されます。デフォルトのビルドの場合、size 引数のみが存在します。wolfssl_static_memory ビルドを使用する場合は、ヒープとタイプ引数が含まれます。
int	**wolfSSL_SetAllocators**(wolfSSL_Malloc_cb , wolfSSL_Free_cb , wolfSSL_Realloc_cb) この機能は、WolfSSL が使用する割り当て関数を登録します。デフォルトでは、システムがそれをサポートしている場合、Malloc / Free と RealLoc が使用されます。この機能を使用すると、実行時にユーザーは独自のメモリハンドラをインストールできます。
int	**wolfSSL_StaticBufferSz**(byte * buffer, word32 sz, int flag) この機能は、静的メモリ機能が使用されている場合（–enable-staticMemory）の場合に使用できます。メモリの「バケット」に最適なバッファサイズを示します。これにより、パーティション化された後に追加の未使用のメモリが終了しないように、バッファサイズを計算する方法が可能になります。返された値は、正の場合、使用するコンピュータのバッファサイズです。
int	**wolfSSL_MemoryPaddingSz**(void) この機能は、静的メモリ機能が使用されている場合（–enable-staticMemory）の場合に使用できます。メモリの各パーティションに必要なパディングのサイズを示します。このパディングサイズは、メモリアライメントのために追加のメモリ管理構造を含む必要があるサイズになります。

	Name
void *	**XMALLOC**(size_t n, void * heap, int type) これは実際には関数ではなく、むしろプリプロセッサマクロであり、ユーザーは自分の Malloc、Realloc、および標準の C メモリ関数の代わりに自由な関数に置き換えることができます。外部メモリ機能を使用するには、xmalloc_user を定義します。これにより、メモリ機能をフォームの外部関数に置き換えます.extern void * xmalloc（size_t n、void * heap、int 型）; extern void * XrealLoc（void * p、size_t n、void ヒープ、*int* 型）。*extern void xfree*（*void* p、void * heap、int 型）; wolfssl_malloc、wolfssl_realloc、wolfssl_free の代わりに基本的な C メモリ機能を使用するには、NO_WOLFSSL_MEMORY を定義します。これにより、メモリ関数が次のものに置き換えられます。::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)）これらのオプションのどれも選択されていない場合、システムはデフォルトで使用されます。WolfSSL メモリ機能ユーザーはコールバックフックを介してカスタムメモリ機能を設定できます (Wolfssl_Malloc、WolfSSL_Realloc、wolfssl_free を参照)。このオプションは、メモリ関数を次のものに置き換えます。::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) これは実際には関数ではなく、むしろプリプロセッサマクロであり、ユーザーは自分の Malloc、Realloc、および標準の C メモリ関数の代わりに自由な関数に置き換えることができます。外部メモリ機能を使用するには、xmalloc_user を定義します。これにより、メモリ機能をフォームの外部関数に置き換えます.extern void * xmalloc（size_t n、void * heap、int 型）; extern void * XrealLoc（void * p、size_t n、void ヒープ、*int* 型）。*extern void xfree*（*void* p、void * heap、int 型）; wolfssl_malloc、wolfssl_realloc、wolfssl_free の代わりに基本的な C メモリ機能を使用するには、NO_WOLFSSL_MEMORY を定義します。これにより、メモリ関数が次のものに置き換えられます。::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)）これらのオプションのどれも選択されていない場合、システムはデフォルトで使用されます。WolfSSL メモリ機能ユーザーはコールバックフックを介してカスタムメモリ機能を設定できます（Wolfssl_Malloc、WolfSSL_Realloc、wolfssl_free を参照）。このオプションは、メモリ関数を次のものに置き換えます。::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) これは実際には関数ではなく、むしろプリプロセッサマクロであり、ユーザーは自分の Malloc、Realloc、および標準の C メモリ関数の代わりに自由な関数に置き換えることができます。外部メモリ機能を使用するには、xmalloc_user を定義します。これにより、メモリ機能をフォームの外部関数に置き換えます.extern void * xmalloc（size_t n、void * heap、int 型）; extern void * XrealLoc（void * p、size_t n、void ヒープ、*int* 型）。*extern void xfree*（*void* p、void * heap、int 型）; wolfssl_malloc、wolfssl_realloc、wolfssl_free の代わりに基本的な C メモリ機能を使用するには、NO_WOLFSSL_MEMORY を定義します。これにより、メモリ関数が次のものに置き換えられます。::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)）これらのオプションのどれも選択されていない場合、システムはデフォルトで使用されます。WolfSSL メモリ機能ユーザーはコールバックフックを介してカスタムメモリ機能を設定できます (Wolfssl_Malloc、WolfSSL_Realloc、wolfssl_free を参照)。このオプションは、メモリ関数を次のものに置き換えます。::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

A.2.2.1 function wolfSSL_Malloc

```
void * wolfSSL_Malloc(
    size_t size,
    void * heap,
    int type
)
```

この関数は malloc（）と似ていますが、WolfSSL が使用するように構成されているメモリ割り当て関数を呼び出します。デフォルトでは、WolfSSL は malloc（）を使用します。これは、WolfSSL メモリ抽象化レイヤを使用して変更できます - wolfssl_setAllocator（）を参照してください。注 WOLFSSL_MALLOC は、WOLFSSL によって直接呼び出されませんが、代わりに Macro XMalloc によって呼び出されます。デフォルトのビルドの場合、size 引数のみが存在します。wolfssl_static_memory ビルドを使用する場合は、ヒープとタイプ引数が含まれます。

Parameters:

- **size** 割り当てるメモリのサイズ（バイト）
- **heap** メモリに使用するヒントヒント。null になることができます *Example*

```
int* tenInts = (int*)wolfSSL_Malloc(sizeof(int)*10);
```

See:

- wolfSSL_Free
- wolfSSL_Realloc
- wolfSSL_SetAllocators
- XMALLOC
- XFREE
- XREALLOC

Return:

- pointer 成功した場合、この関数は割り当てられたメモリへのポインタを返します。
- error エラーがある場合は、NULL が返されます。

A.2.2.2 function wolfSSL_Free

```
void wolfSSL_Free(
    void * ptr,
    void * heap,
    int type
)
```

この関数は free（）と似ていますが、WolfSSL が使用するように構成されているメモリフリー機能を呼び出します。デフォルトでは、WolfSSL は free（）を使用します。これは、WolfSSL メモリ抽象化レイヤを使用して変更できます - wolfssl_setAllocator（）を参照してください。注 WOLFSSL_FREE は WOLFSSL によって直接呼び出されませんが、代わりにマクロ XFree によって呼び出されます。デフォルトのビルドの場合、PTR 引数のみが存在します。wolfssl_static_memory ビルドを使用する場合は、ヒープとタイプ引数が含まれます。

Parameters:

- **ptr** 解放されるメモリへのポインタ。
- **heap** メモリに使用するヒントヒント。null になることができます *Example*

```
int* tenInts = (int*)wolfSSL_Malloc(sizeof(int)*10);
// process data as desired
...
if(tenInts) {
    wolfSSL_Free(tenInts);
}
```

See:

- wolfSSL_Alloc
- wolfSSL_Realloc
- wolfSSL_SetAllocators
- XMALLOC
- XFREE
- XREALLOC

Return: none いいえ返します。

A.2.2.3 function wolfSSL_Realloc

```
void * wolfSSL_Realloc(
    void * ptr,
    size_t size,
    void * heap,
    int type
)
```

この関数は REALLOC（）と似ていますが、WolfSSL が使用するように構成されているメモリ再割り当て機能を呼び出します。デフォルトでは、WolfSSL は RealLoc（）を使用します。これは、WolfSSL メモリ抽象化レイヤを使用して変更できます - wolfssl_setAllocator（）を参照してください。注 WOLFSSL_REALLOC は WOLFSSL によって直接呼び出されませんが、代わりにマクロ Xrealloc によって呼び出されます。デフォルトのビルドの場合、size 引数のみが存在します。wolfssl_static_memory ビルドを使用する場合は、ヒープとタイプ引数が含まれます。

Parameters:

- **ptr** 再割り当てされているメモリへのポインタ。
- **size** 割り当てるバイト数。
- **heap** メモリに使用するヒントヒント。null になることができます *Example*

```
int* tenInts = (int*)wolfSSL_Malloc(sizeof(int)*10);
int* twentyInts = (int*)wolfSSL_Realloc(tenInts, sizeof(int)*20);
```

See:

- wolfSSL_Free
- wolfSSL_Malloc
- wolfSSL_SetAllocators
- XMALLOC
- XFREE
- XREALLOC

Return:

- pointer 成功した場合、この関数はマイポイントを再割り当てするためのポインタを返します。これは PTR と同じポインタ、または新しいポインタの場所であり得る。
- Null エラーがある場合は、NULL が返されます。

A.2.2.4 function wolfSSL_SetAllocators

```
int wolfSSL_SetAllocators(
    wolfSSL_Malloc_cb ,
    wolfSSL_Free_cb ,
    wolfSSL_Realloc_cb
)
```

この機能は、WolfSSL が使用する割り当て関数を登録します。デフォルトでは、システムがそれをサポートしている場合、Malloc / Free と RealLoc が使用されます。この機能を使用すると、実行時にユーザーは独自のメモリハンドラをインストールできます。

Parameters:

- **malloc_function** 使用する WolfSSL のメモリ割り当て機能関数署名は、上記の wolfssl_malloc_cb プロトタイプと一致する必要があります。
- **free_function** 使用する WolfSSL のメモリフリー機能関数シグネチャは、上記の wolfssl_free_cb プロトタイプと一致する必要があります。*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
}
```

See: none

Return:

- Success 成功した場合、この関数は 0 を返します。
- BAD_FUNC_ARG 関数ポインタが提供されていない場合に返されるエラーです。

A.2.2.5 function wolfSSL_StaticBufferSz

```
int wolfSSL_StaticBufferSz(
    byte * buffer,
    word32 sz,
    int flag
)
```

この機能は、静的メモリ機能が使用されている場合（–enable-staticMemory）の場合に使用できます。メモリの「バケット」に最適なバッファサイズを示します。これにより、パーティション化された後に追加の未使用のメモリが終了しないように、バッファサイズを計算する方法が可能になります。返された値は、正の場合、使用するコンピュータのバッファサイズです。

Parameters:

- **buffer** バッファへのポインタ
- **size** バッファのサイズ *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);
...
```

See:

- wolfSSL_Malloc
- wolfSSL_Free

Return:

- Success バッファサイズ計算を正常に完了すると、正の値が返されます。この返された値は最適なバッファサイズです。
- Failure すべての負の値はエラーの場合と見なされます。

A.2.2.6 function wolfSSL_MemoryPaddingSz

```
int wolfSSL_MemoryPaddingSz(
    void
)
```

この機能は、静的メモリ機能が使用されている場合（–enable-staticMemory）の場合に使用できます。メモリの各パーティションに必要なパディングのサイズを示します。このパディングサイズは、メモリアライメントのために追加のメモリ管理構造を含む必要があるサイズになります。

See:

- wolfSSL_Malloc
- wolfSSL_Free

Return:

- On 正常なメモリパディング計算戻り値は正の値になります
- All 負の値はエラーケースと見なされます。*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)
...
```

A.2.2.7 function XMALLOC

```
void * XMALLOC(
    size_t n,
    void * heap,
    int type
)
```

これは実際には関数ではなく、むしろプリプロセッサマクロであり、ユーザーは自分の Malloc、Realloc、および標準の C メモリ関数の代わりに自由な関数に置き換えることができます。外部メモリ機能を使用するには、xmalloc_user を定義します。これにより、メモリ機能をフォームの外部関数に置き換えます.extern void * xmalloc（size_t n、void * heap、int 型）; extern void * XrealLoc（void * p、size_t n、void ヒープ、*int* 型）。*extern void xfree*（*void* p、void * heap、int 型）; wolfssl_malloc、wolfssl_realloc、wolfssl_free の代わりに基本的な C メモリ機能を使用するには、NO_WOLFSSL_MEMORY を定義します。これにより、メモリ関数が次のものに置き換えられます。::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））これらのオプションのどれも選択されていない場合、システムはデフォルトで使用されます。WolfSSL メモリ機能ユーザーはコールバックフックを介してカスタムメモリ機能を設定できます（Wolfssl_Malloc、WolfSSL_Realloc、wolfssl_free を参照）。このオプションは、メモリ関数を次のものに置き換えます。::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** 割り当てるメモリのサイズ
- **h** （カスタム XMalloc 関数で使用されています）使用するヒープへのポインタ *Example*

```
int* tenInts = XMALLOC(sizeof(int)*10, NULL, DYNAMIC_TYPE_TMP_BUFFER);
if (tenInts == NULL) {
    // error allocating space
    return MEMORY_E;
}
```

See:

- wolfSSL_Malloc

- wolfSSL_Realloc
- wolfSSL_Free
- wolfSSL_SetAllocators

Return:

- pointer 成功したメモリへのポインタを返します
- NULL 失敗した

A.2.2.8 function XREALLOC

```
void * XREALLOC(
    void * p,
    size_t n,
    void * heap,
    int type
)
```

これは実際には関数ではなく、むしろプリプロセッサマクロであり、ユーザーは自分の Malloc、Realloc、および標準の C メモリ関数の代わりに自由な関数に置き換えることができます。外部メモリ機能を使用するには、xmalloc_user を定義します。これにより、メモリ機能をフォームの外部関数に置き換えます.extern void * xmalloc（size_t n、void * heap、int 型）; extern void * XrealLoc（void * p、size_t n、void ヒープ、*int* 型）。*extern void xfree*（*void* p、void * heap、int 型）; wolfssl_malloc、wolfssl_realloc、wolfssl_free の代わりに基本的な C メモリ機能を使用するには、NO_WOLFSSL_MEMORY を定義します。これにより、メモリ関数が次のものに置き換えられます。::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））これらのオプションのどれも選択されていない場合、システムはデフォルトで使用されます。WolfSSL メモリ機能ユーザーはコールバックフックを介してカスタムメモリ機能を設定できます（Wolfssl_Malloc、WolfSSL_Realloc、wolfssl_free を参照）。このオプションは、メモリ関数を次のものに置き換えます。::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** Reallocate へのアドレスへのポインタ
- **n** 割り当てるメモリのサイズ
- **h** （カスタム Xrealloc 関数で使用されています）使用するヒープへのポインタ *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);
```

See:

- wolfSSL_Malloc
- wolfSSL_Realloc
- wolfSSL_Free
- wolfSSL_SetAllocators

Return:

- Return 成功したメモリを割り当てるポインタ
- NULL 失敗した

A.2.2.9 function XFREE

```
void XFREE(
    void * p,
    void * heap,
```

```
    int type
)
```

これは実際には関数ではなく、むしろプリプロセッサマクロであり、ユーザーは自分の Malloc、Realloc、および標準の C メモリ関数の代わりに自由な関数に置き換えることができます。外部メモリ機能を使用するには、xmalloc_user を定義します。これにより、メモリ機能をフォームの外部関数に置き換えます.extern void * xmalloc（size_t n、void * heap、int 型）; extern void * XrealLoc（void * p、size_t n、void ヒープ、*int* 型）。*extern void xfree*（*void* p、void * heap、int 型）; wolfssl_malloc、wolfssl_realloc、wolfssl_free の代わりに基本的な C メモリ機能を使用するには、NO_WOLFSSL_MEMORY を定義します。これにより、メモリ関数が次のものに置き換えられます。::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））これらのオプションのどれも選択されていない場合、システムはデフォルトで使用されます。WolfSSL メモリ機能ユーザーはコールバックフックを介してカスタムメモリ機能を設定できます（Wolfssl_Malloc、WolfSSL_Realloc、wolfssl_free を参照）。このオプションは、メモリ関数を次のものに置き換えます。::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** 無料のアドレスへのポインタ
- **h** 使用するヒープへの（カスタム XFree 関数で使用されています）。*Example*

```
int* tenInts = XMALLOC(sizeof(int) * 10, NULL, DYNAMIC_TYPE_TMP_BUFFER);
if (tenInts == NULL) {
    // error allocating space
    return MEMORY_E;
}
```

See:

- wolfSSL_Malloc
- wolfSSL_Realloc
- wolfSSL_Free
- wolfSSL_SetAllocators

Return: none いいえ返します。

A.3 OpenSSL API

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) この関数は、次の数学「R =（A ^ P）%M」を実行します。
const WOLFSSL_EVP_CIPHER *	**wolfSSL_EVP_des_ede3_ecb**(void) それぞれの wolfssl_evp_cipher ポインタのゲッター関数。最初にプログラム内で wolfssl_evp_init（）を 1 回呼び出す必要があります。wolfssl_des_ecb マクロは、wolfssl_evp_des_ede3_ecb（）に対して定義する必要があります。

	Name
const WOLFSSL_EVP_CIPHER *	**wolfSSL_EVP_des_cbc**(void) それぞれの wolfssl_evp_cipher ポインタのゲッター関数。最初にプログラム内で wolfssl_evp_init （） を 1 回呼び出す必要があります。wolfssl_des_ecb マクロは、wolfssl_evp_des_ecb （） に対して定義する必要があります。
int	**wolfSSL_EVP_DigestInit_ex**(WOLFSSL_EVP_MD_CTX * ctx, const WOLFSSL_EVP_MD * type, WOLFSSL_ENGINE * impl)wolfssl_evp_md_ctx を初期化する機能。この関数は wolfssl_engine が wolfssl_engine を使用しないため、wolfssl_evp_digestinit （） のラッパーです。
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)wolfssl_evp_cipher_ctx を初期化する機能。この関数は wolfssl_engine が wolfssl_engine を使用しないため、wolfssl_ciphinit （） のラッパーです。
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)wolfssl_evp_cipher_ctx を初期化する機能。WolfSSL は WOLFSSL_ENGINE を使用しないため、この関数は wolfssl_evp_ciphinit （） のラッパーです。暗号化フラグを暗号化するように設定します。
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)wolfssl_evp_cipher_ctx を初期化する機能。WolfSSL は WOLFSSL_ENGINE を使用しないため、この関数は wolfssl_evp_ciphinit （） のラッパーです。暗号化フラグを復号化するように設定します。
int	**wolfSSL_EVP_CipherUpdate**(WOLFSSL_EVP_CIPHER_CTX * ctx, unsigned char * out, int * outl, const unsigned char * in, int inl) データを暗号化/復号化する機能。バッファ内では暗号化または復号化され、OUT バッファが結果を保持します。OUTOR は暗号化/復号化された情報の長さになります。

	Name
int	**wolfSSL_EVP_CipherFinal**(WOLFSSL_EVP_CIPHER_CTX * ctx, unsigned char * out, int * outl) この関数は、パディングを追加する最終暗号化操作を実行します。wolfssl_evp_ciph_no_padding フラグが wolfssl_evp_cipher_ctx 構造に設定されている場合、1 が返され、暗号化/復号化は行われません。PADDING FLAG が SETI パディングを追加して暗号化すると、暗号化に CTX が設定されていると、復号化されたときにパディング値がチェックされます。
int	**wolfSSL_EVP_CIPHER_CTX_set_key_length**(WOLFSSL_EVP_CIPHE * ctx, int keylen)WolfSSL EVP_CIPHER_CTX 構造キー長の設定機能
int	**wolfSSL_EVP_CIPHER_CTX_block_size**(const WOLFSSL_EVP_CIPHER_CTX * ctx) これは CTX ブロックサイズの Getter 関数です。
int	**wolfSSL_EVP_CIPHER_block_size**(const WOLFSSL_EVP_CIPHER * cipher) これは暗号のブロックサイズのゲッター関数です。
void	**wolfSSL_EVP_CIPHER_CTX_set_flags**(WOLFSSL_EVP_CIPHER_CTX * ctx, int flags)WolfSSL evp_cipher_ctx 構造の設定機能
void	**wolfSSL_EVP_CIPHER_CTX_clear_flags**(WOLFSSL_EVP_CIPHER_C * ctx, int flags)WolfSSL evp_cipher_ctx 構造のクリア機能
int	**wolfSSL_EVP_CIPHER_CTX_set_padding**(WOLFSSL_EVP_CIPHER_ * c, int pad)wolfssl_evp_cipher_ctx 構造のためのセッター機能パディングを使用する。
unsigned long	**wolfSSL_EVP_CIPHER_CTX_flags**(const WOLFSSL_EVP_CIPHER_CTX * ctx)wolfssl_evp_cipher_ctx 構造のゲッター関数廃止予定の 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) この関数は、PEM 形式の wolfssl_bio 構造体にキーを書き込みます。
int	**wolfSSL_CTX_use_RSAPrivateKey_file**(WOLFSSL_CTX * ctx, const char * file, int format) この関数は、SSL 接続で使用されている RSA 秘密鍵を SSL コンテキスト (WOLFSSL_CTX) にロードします。この関数は、wolfSSL が OpenSSL 互換 API が有効（–enable_openSSLExtra、#define OPENSSL_EXTRA）でコンパイルされている場合にのみ利用可能で、より一般的に使用されている wolfSSL_CTX_use_PrivateKey_file() 関数と同じです。ファイル引数には、RSA 秘密鍵ファイルへのポインタが、引数 format で指定された形式で含まれています。

	Name
int	**wolfSSL_use_certificate_file**(WOLFSSL * ssl, const char * file, int format) この関数は証明書ファイルを SSL セッション（WOLFSSL 構造体）にロードします。証明書ファイルはファイル引数によって提供されます。引数 format は、ファイルのフォーマットタイプ（SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM）を指定します。
int	**wolfSSL_use_PrivateKey_file**(WOLFSSL * ssl, const char * file, int format) この関数は、秘密鍵ファイルを SSL セッション（WOLFSSL 構造体）にロードします。鍵ファイルは引数 file によって提供されます。引数 format は、ファイルのタイプ（SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM が指定可）を指定します。外部キーストアを使用し、秘密鍵を持っていない場合は、代わりに公開鍵を入力して CryPro コールバックを登録して署名を処理することができます。このためには、Crypto コールバックまたは PK コールバックを使用したコンフィグレーションでビルドします。Crypto コールバックを有効にするには、–enable-cryptocb または WOLF_CRYPTO_CB マクロを使用してビルドし、wc_CryptoCb_RegisterDevice を使用して暗号コールバックを登録し、wolfSSL_SetDevId を使用して関連する devId を設定します。
int	**wolfSSL_use_certificate_chain_file**(WOLFSSL * ssl, const char * file) この関数は、証明書チェーンを SSL セッション WOLFSSL 構造体）にロードします。証明書チェーンを含むファイルは引数 file によって提供され、PEM 形式の証明書を含める必要があります。この関数は、MAX_CHAIN_DEPTH（既定で 9、internal.h で定義されている）証明書に加えて、サブジェクト証明書を処理します。
int	**wolfSSL_use_RSAPrivateKey_file**(WOLFSSL * ssl, const char * file, int format) この関数は、SSL 接続で使用されている RSA 秘密鍵を SSL セッション（WOLFSSL 構造体）にロードします。この関数は、wolfSSL が OpenSSL 互換 API を有効（–enable_openSSlExtra、#define OPENSSL_EXTRA）でビルドされている場合にのみ利用可能で、より一般的に使用される wolfSSL_use_PrivateKey_file() 関数と同じです。引数 file には、RSA 秘密鍵ファイルへのポインタが、フォーマットで指定された形式で含まれています。
long	**wolfSSL_set_tlsext_status_type**(WOLFSSL * s, int type) この関数は、サーバが OCSP ステータス応答（OCSP ステイプルとも呼ばれる）を送受信するクライアントアプリケーションが要求されたときに呼び出されます。
WOLFSSL_X509_CHAIN *	**wolfSSL_get_peer_chain**(WOLFSSL * ssl) ピアの証明書チェーンを取得します。

	Name
int	**wolfSSL_get_chain_count**(WOLFSSL_X509_CHAIN * chain) ピアの証明書チェーン数を取得します。
int	**wolfSSL_get_chain_length**(WOLFSSL_X509_CHAIN * chain, int idx)Index（IDX）のピアの ASN1.DER 証明書長をバイト単位で取得します。
unsigned char *	**wolfSSL_get_chain_cert**(WOLFSSL_X509_CHAIN * chain, int idx) インデックス（IDX）でピアの ASN1.DER 証明書を取得します。
int	**wolfSSL_get_chain_cert_pem**(WOLFSSL_X509_CHAIN * chain, int idx, unsigned char * buf, int inLen, int * outLen) インデックス（IDX）でピアの PEM 証明書を取得します。
const unsigned char *	**wolfSSL_get_sessionID**(const WOLFSSL_SESSION * s) セッションの ID を取得します。セッション ID は常に 32 バイトの長さです。
int	**wolfSSL_X509_get_serial_number**(WOLFSSL_X509 * x509, unsigned char * in, int * inOutSz) ピアの証明書のシリアル番号を取得します。シリアル番号バッファ（IN）は少なくとも 32 バイト以上であり、入力として * INOUTSZ 引数として提供されます。関数を呼び出した後 * INOUTSZ は IN バッファに書き込まれた実際の長さをバイト単位で保持します。
WC_PKCS12 *	**wolfSSL_d2i_PKCS12_bio**(WOLFSSL_BIO * bio, WC_PKCS12 ** pkcs12)WOLFSSL_D2I_PKCS12_BIO（D2I_PKCS12_BIO）は、WOLFSSL_BIO から構造 WC_PKCS12 への PKCS12 情報にコピーされます。この情報は、オプションの MAC 情報を保持するための構造とともにコンテンツに関する情報のリストとして構造内に分割されています。構造体 WC_PKCS12 で情報がチャンク（ただし復号化されていない）に分割された後、それはその後、呼び出しによって解析および復号化され得る。
WC_PKCS12 *	**wolfSSL_i2d_PKCS12_bio**(WOLFSSL_BIO * bio, WC_PKCS12 * pkcs12)WOLFSSL_I2D_PKCS12_BIO（I2D_PKCS12_BIO）は、構造 WC_PKCS12 から WOLFSSL_BIO への証明書情報にコピーされます。

	Name
int	**wolfSSL_PKCS12_parse(WOLFSSL_X509)** ca)pkcs12 は、configure コマンドへの-enable-openSSLAXTRA を追加することで有効にできます。それは復号化のためにトリプル DES と RC4 を使うことができるので、OpenSSlextra（–enable-des3 -enable-arc4）を有効にするときにもこれらの機能を有効にすることをお勧めします。wolfssl は現在 RC2 をサポートしていませんので、RC2 での復号化は現在利用できません。これは、.p12 ファイルを作成するために OpenSSL コマンドラインで使用されるデフォルトの暗号化方式では注目すかもしれません。WOLFSSL_PKCS12_PARSE（PKCS12_PARSE）。この関数が最初に行っているのは、存在する場合は Mac が正しいチェックです。MAC が失敗した場合、関数は返され、保存されているコンテンツ情報のいずれかを復号化しようとしません。この関数は、バッグタイプを探している各コンテンツ情報を介して解析します。バッグタイプがわかっている場合は、必要に応じて復号化され、構築されている証明書のリストに格納されているか、見つかったキーとして保存されます。すべてのバッグを介して解析した後、見つかったキーは、一致するペアが見つかるまで証明書リストと比較されます。この一致するペアはキーと証明書として返され、オプションで見つかった証明書リストは stack_of 証明書として返されます。瞬間、CRL、秘密または安全なバッグがスキップされ、解析されません。デバッグプリントアウトを見ることで、これらまたは他の「不明」バッグがスキップされているかどうかがわかります。フレンドリー名などの追加の属性は、PKCS12 ファイルを解析するときにスキップされます。

A.3.2 Functions Documentation

A.3.2.1 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
)
```

この関数は、次の数学「R =（A ^ P）%M」を実行します。

Parameters:

- **r** 結果を保持するための構造。
- **a** 電力で上げられる値。
- **p** によって上げる力。
- **m** 使用率 *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
```

See:

- wolfSSL_BN_new
- wolfSSL_BN_free

Return:

- SSL_SUCCESS 数学操作をうまく実行します。
- SSL_FAILURE エラーケースに遭遇した場合

A.3.2.2 function wolfSSL_EVP_des_ede3_ecb

```
const WOLFSSL_EVP_CIPHER * wolfSSL_EVP_des_ede3_ecb(
    void
)
```

それぞれの wolfssl_evp_cipher ポインタのゲッター関数。最初にプログラム内で wolfssl_evp_init（）を 1 回呼び出す必要があります。wolfssl_des_ecb マクロは、wolfssl_evp_des_ede3_ecb（）に対して定義する必要があります。

See: wolfSSL_EVP_CIPHER_CTX_init

Return: pointer DES EDE3 操作のための wolfssl_evp_cipher ポインタを返します。*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()));
```

A.3.2.3 function wolfSSL_EVP_des_cbc

```
const WOLFSSL_EVP_CIPHER * wolfSSL_EVP_des_cbc(
    void
)
```

それぞれの wolfssl_evp_cipher ポインタのゲッター関数。最初にプログラム内で wolfssl_evp_init（）を 1 回呼び出す必要があります。wolfssl_des_ecb マクロは、wolfssl_evp_des_ecb（）に対して定義する必要があります。

See: wolfSSL_EVP_CIPHER_CTX_init

Return: pointer DES 操作のための wolfssl_evp_cipher ポインタを返します。*Example*

```
WOLFSSL_EVP_CIPHER* cipher;
cipher = wolfSSL_EVP_des_cbc();
…
```

A.3.2.4 function wolfSSL_EVP_DigestInit_ex

```
int wolfSSL_EVP_DigestInit_ex(
    WOLFSSL_EVP_MD_CTX * ctx,
    const WOLFSSL_EVP_MD * type,
    WOLFSSL_ENGINE * impl
)
```

wolfssl_evp_md_ctx を初期化する機能。この関数は wolfssl_engine が wolfssl_engine を使用しないため、wolfssl_evp_digestinit（）のラッパーです。

Parameters:

- **ctx** 初期化する構造
- **type** SHA などのハッシュの種類。*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
```

See:

- wolfSSL_EVP_MD_CTX_new
- wolfCrypt_Init
- wolfSSL_EVP_MD_CTX_free

Return:

- SSL_SUCCESS 正常に設定されている場合。
- SSL_FAILURE 成功しなかった場合

A.3.2.5 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
)
```

wolfssl_evp_cipher_ctx を初期化する機能。この関数は wolfssl_engine が wolfssl_engine を使用しないため、wolfssl_ciphinit（）のラッパーです。

Parameters:

- **ctx** 初期化する構造
- **type** AES などの暗号化/復号化の種類。
- **impl** 使用するエンジン。wolfssl の n / a は、null になることができます。
- **key** 設定するキー
- **iv** アルゴリズムで必要な場合は IV。*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
```

See:

- wolfSSL_EVP_CIPHER_CTX_new
- wolfCrypt_Init
- wolfSSL_EVP_CIPHER_CTX_free

Return:

- SSL_SUCCESS 正常に設定されている場合。
- SSL_FAILURE 成功しなかった場合

A.3.2.6 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
)
```

wolfssl_evp_cipher_ctx を初期化する機能。WolfSSL は WOLFSSL_ENGINE を使用しないため、この関数は wolfssl_evp_ciphinit（）のラッパーです。暗号化フラグを暗号化するように設定します。

Parameters:

- **ctx** 初期化する構造
- **type** AES などの暗号化の種類。
- **impl** 使用するエンジン。wolfssl の n / a は、null になることができます。
- **key** 使用する鍵 *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
```

See:

- wolfSSL_EVP_CIPHER_CTX_new
- wolfCrypt_Init
- wolfSSL_EVP_CIPHER_CTX_free

Return:

- SSL_SUCCESS 正常に設定されている場合。

- SSL_FAILURE 成功しなかった場合

A.3.2.7 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
)
```

wolfssl_evp_cipher_ctx を初期化する機能。WolfSSL は WOLFSSL_ENGINE を使用しないため、この関数は wolfssl_evp_ciphinit（）のラッパーです。暗号化フラグを復号化するように設定します。

Parameters:

- **ctx** 初期化する構造
- **type** AES などの暗号化/復号化の種類。
- **impl** 使用するエンジン。wolfssl の n / a は、null になることができます。
- **key** 設定するキー
- **iv** アルゴリズムで必要な場合は IV。*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
```

See:

- wolfSSL_EVP_CIPHER_CTX_new
- wolfCrypt_Init
- wolfSSL_EVP_CIPHER_CTX_free

Return:

- SSL_SUCCESS 正常に設定されている場合。
- SSL_FAILURE 成功しなかった場合

A.3.2.8 function wolfSSL_EVP_CipherUpdate

```
int wolfSSL_EVP_CipherUpdate(
    WOLFSSL_EVP_CIPHER_CTX * ctx,
    unsigned char * out,
    int * outl,
```

```
    const unsigned char * in,
    int inl
)
```

データを暗号化/復号化する機能。バッファ内では暗号化または復号化され、OUT バッファが結果を保持します。OUTOR は暗号化/復号化された情報の長さになります。

Parameters:

- **ctx** から暗号化の種類を取得するための構造。
- **out** 出力を保持するためのバッファ。
- **outl** 出力のサイズになるように調整しました。
- **in** 操作を実行するためのバッファー。*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
```

See:

- wolfSSL_EVP_CIPHER_CTX_new
- wolfCrypt_Init
- wolfSSL_EVP_CIPHER_CTX_free

Return:

- SSL_SUCCESS 成功した場合
- SSL_FAILURE 成功しなかった場合

A.3.2.9 function wolfSSL_EVP_CipherFinal

```
int wolfSSL_EVP_CipherFinal(
    WOLFSSL_EVP_CIPHER_CTX * ctx,
    unsigned char * out,
    int * outl
)
```

この関数は、パディングを追加する最終暗号化操作を実行します。wolfssl_evp_ciph_no_padding フラグが wolfssl_evp_cipher_ctx 構造に設定されている場合、1 が返され、暗号化/復号化は行われません。PADDING FLAG が SETI パディングを追加して暗号化すると、暗号化に CTX が設定されていると、復号化されたときにパディング値がチェックされます。

Parameters:

- **ctx** 復号化/暗号化する構造。
- **out** 最後の復号化/暗号化のためのバッファ。*Example*

```
WOLFSSL_EVP_CIPHER_CTX* ctx;
int out1;
unsigned char out[64];
```

```
// create ctx
wolfSSL_EVP_CipherFinal(ctx, out, &out1);
```

See: wolfSSL_EVP_CIPHER_CTX_new

Return:

- 1 成功に戻りました。
- 0 失敗に遭遇した場合

A.3.2.10 function wolfSSL_EVP_CIPHER_CTX_set_key_length

```
int wolfSSL_EVP_CIPHER_CTX_set_key_length(
    WOLFSSL_EVP_CIPHER_CTX * ctx,
    int keylen
)
```

WolfSSL EVP_CIPHER_CTX 構造キー長の設定機能

Parameters:

- **ctx** キーの長さを設定する構造 *Example*

```
WOLFSSL_EVP_CIPHER_CTX* ctx;
int keylen;
// create ctx
wolfSSL_EVP_CIPHER_CTX_set_key_length(ctx, keylen);
```

See: wolfSSL_EVP_CIPHER_flags

Return:

- SSL_SUCCESS 正常に設定されている場合。
- SSL_FAILURE キーの長さを設定できなかった場合。

A.3.2.11 function wolfSSL_EVP_CIPHER_CTX_block_size

```
int wolfSSL_EVP_CIPHER_CTX_block_size(
    const WOLFSSL_EVP_CIPHER_CTX * ctx
)
```

これは CTX ブロックサイズの Getter 関数です。

See: wolfSSL_EVP_CIPHER_block_size

Return: size ctx-> block_size を返します。*Example*

```
const WOLFSSL_CVP_CIPHER_CTX* ctx;
//set up ctx
printf("block size = %d\n", wolfSSL_EVP_CIPHER_CTX_block_size(ctx));
```

A.3.2.12 function wolfSSL_EVP_CIPHER_block_size

```
int wolfSSL_EVP_CIPHER_block_size(
    const WOLFSSL_EVP_CIPHER * cipher
)
```

これは暗号のブロックサイズのゲッター関数です。

See: wolfSSL_EVP_aes_256_ctr

Return: size ブロックサイズを返します。*Example*

```
printf(“block size = %d\n”,
wolfSSL_EVP_CIPHER_block_size(wolfSSL_EVP_aes_256_ecb()));
```

A.3.2.13 function wolfSSL_EVP_CIPHER_CTX_set_flags

```
void wolfSSL_EVP_CIPHER_CTX_set_flags(
    WOLFSSL_EVP_CIPHER_CTX * ctx,
    int flags
)
```

WolfSSL evp_cipher_ctx 構造の設定機能

Parameters:

- **ctx** フラグを設定する構造 *Example*

```
WOLFSSL_EVP_CIPHER_CTX* ctx;
int flag;
// create ctx
wolfSSL_EVP_CIPHER_CTX_set_flags(ctx, flag);
```

See:

- wolfSSL_EVP_CIPHER_flags
- wolfSSL_EVP_CIPHER_CTX_flags

Return: none いいえ返します。

A.3.2.14 function wolfSSL_EVP_CIPHER_CTX_clear_flags

```
void wolfSSL_EVP_CIPHER_CTX_clear_flags(
    WOLFSSL_EVP_CIPHER_CTX * ctx,
    int flags
)
```

WolfSSL evp_cipher_ctx 構造のクリア機能

Parameters:

- **ctx** フラグをクリアするための構造 *Example*

```
WOLFSSL_EVP_CIPHER_CTX* ctx;
int flag;
// create ctx
wolfSSL_EVP_CIPHER_CTX_clear_flags(ctx, flag);
```

See:

- wolfSSL_EVP_CIPHER_flags
- wolfSSL_EVP_CIPHER_CTX_flags

Return: none いいえ返します。

A.3.2.15 function wolfSSL_EVP_CIPHER_CTX_set_padding

```
int wolfSSL_EVP_CIPHER_CTX_set_padding(
    WOLFSSL_EVP_CIPHER_CTX * c,
    int pad
)
```

wolfssl_evp_cipher_ctx 構造のためのセッター機能パディングを使用する。

Parameters:

- **ctx** パディングフラグを設定する構造 *Example*

```
WOLFSSL_EVP_CIPHER_CTX* ctx;
// create ctx
wolfSSL_EVP_CIPHER_CTX_set_padding(ctx, 1);
```

See: wolfSSL_EVP_CIPHER_CTX_new

Return:

- SSL_SUCCESS 正常に設定されている場合。
- BAD_FUNC_ARG NULL 引数が渡された場合。

A.3.2.16 function wolfSSL_EVP_CIPHER_CTX_flags

```
unsigned long wolfSSL_EVP_CIPHER_CTX_flags(
    const WOLFSSL_EVP_CIPHER_CTX * ctx
)
```

wolfssl_evp_cipher_ctx 構造のゲッター関数廃止予定の V1.1.0

See:

- wolfSSL_EVP_CIPHER_CTX_new
- wolfSSL_EVP_CIPHER_flags

Return: unsigned フラグ/モードの長い。*Example*

```
WOLFSSL_EVP_CIPHER_CTX* ctx;
unsigned long flags;
ctx = wolfSSL_EVP_CIPHER_CTX_new()
flags = wolfSSL_EVP_CIPHER_CTX_flags(ctx);
```

A.3.2.17 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
)
```

この関数は、PEM 形式の wolfssl_bio 構造体にキーを書き込みます。

Parameters:

- **bio** wolfssl_bio 構造体から PEM バッファを取得します。
- **key** PEM 形式に変換するためのキー。
- **cipher** EVP 暗号構造
- **passwd** パスワード。
- **len** パスワードの長さ
- **cb** パスワードコールバック *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
```

See: wolfSSL_PEM_read_bio_X509_AUX

Return:

- SSL_SUCCESS 成功すると。
- SSL_FAILURE 失敗すると。

A.3.2.18 function wolfSSL_CTX_use_RSAPrivateKey_file

```
int wolfSSL_CTX_use_RSAPrivateKey_file(
    WOLFSSL_CTX * ctx,
    const char * file,
    int format
)
```

この関数は、SSL 接続で使用されている RSA 秘密鍵を SSL コンテキスト (WOLFSSL_CTX) にロードします。この関数は、wolfSSL が OpenSSL 互換 API が有効（–enable-openSSLExtra、#define OPENSSL_EXTRA）でコンパイルされている場合にのみ利用可能で、より一般的に使用されている wolfSSL_CTX_use_PrivateKey_file() 関数と同じです。ファイル引数には、RSA 秘密鍵ファイルへのポインタが、引数 format で指定された形式で含まれています。

Parameters:

- **ctx** wolfSSL_CTX_new()を使用して作成された WOLFSSL_CTX 構造体へのポインタ
- **file** フォーマットで指定された形式で、WolfSSL SSL コンテキストにロードされる RSA 秘密鍵を含むファイルの名前へのポインタ。
- **format** RSA 秘密鍵のエンコード形式を指定します。指定可能なフォーマット値は：SSL_FILETYPE_PEM と 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 成功時に返されます。
- SSL_FAILURE 関数呼び出しが失敗した場合に返されます。失敗の原因には次が考えられます：入力鍵ファイルが誤った形式である、または引数 format を使用して誤った形式が与えられている場合、ファイルが存在しない、読み込めない、または破損してる、メモリ不足状態が発生。

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
}
...
```

A.3.2.19 function wolfSSL_use_certificate_file

```
int wolfSSL_use_certificate_file(
    WOLFSSL * ssl,
    const char * file,
    int format
)
```

この関数は証明書ファイルを SSL セッション（WOLFSSL 構造体）にロードします。証明書ファイルはファイル引数によって提供されます。引数 format は、ファイルのフォーマットタイプ（SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM）を指定します。

Parameters:

- **ssl** wolfSSL_new()で作成された WOLFSSL 構造体へのポインタ。
- **file** WOLFSSL 構造体にロードされる証明書を含むファイルの名前へのポインタ
- **format** 証明書ファイルのエンコード形式を指定します。指定可能なフォーマット値は：SSL_FILETYPE_PEM と SSL_FILETYPE_ASN1

See:

- wolfSSL_CTX_use_certificate_buffer
- wolfSSL_CTX_use_certificate_file
- wolfSSL_use_certificate_buffer

Return:

- SSL_SUCCESS 成功時に返されます。
- SSL_FAILURE 関数呼び出しが失敗した場合に返されます。可能な原因には次のようなものがあります。ファイルが誤った形式、または引数 format を使用して誤った形式が与えられた、メモリ不足状態が発生した、ファイルで Base16 のデコードが失敗した

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
}
...
```

A.3.2.20 function wolfSSL_use_PrivateKey_file

```
int wolfSSL_use_PrivateKey_file(
    WOLFSSL * ssl,
    const char * file,
    int format
)
```

この関数は、秘密鍵ファイルを SSL セッション（WOLFSSL 構造体）にロードします。鍵ファイルは引数 file によって提供されます。引数 format は、ファイルのタイプ（SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM が指定可）を指定します。外部キーストアを使用し、秘密鍵を持っていない場合は、代わりに公開鍵を入力して CryPro コールバックを登録して署名を処理することができます。このためには、Crypto コールバックまたは PK コールバックを使用したコンフィグレーションでビルドします。Crypto コールバックを有効にするには、–enable-cryptocb または WOLF_CRYPTO_CB マクロを使用してビルドし、wc_CryptoCb_RegisterDevice を使用して暗号コールバックを登録し、wolfSSL_SetDevId を使用して関連する devId を設定します。

Parameters:

- **ssl** wolfSSL_new()で作成された WOLFSSL 構造体へのポインタ。
- **file** WOLFSSL 構造体にロードされる証明書を含むファイルの名前へのポインタ
- **format** 秘密鍵ファイルのエンコード形式を指定します。指定可能なフォーマット値は：SSL_FILETYPE_PEM と 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 成功時に返されます。
- SSL_FAILURE 関数呼び出しが失敗した場合に返されます。可能な原因には次のようなものがあります。ファイルが誤った形式、または引数 format を使用して誤った形式が与えられた、メモリ不足状態が発生した、ファイルで Base16 のデコードが失敗した

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
}
...
```

A.3.2.21 function wolfSSL_use_certificate_chain_file

```
int wolfSSL_use_certificate_chain_file(
    WOLFSSL * ssl,
    const char * file
)
```

この関数は、証明書チェーンを SSL セッション WOLFSSL 構造体）にロードします。証明書チェーンを含むファイルは引数 file によって提供され、PEM 形式の証明書を含める必要があります。この関数は、MAX_CHAIN_DEPTH（既定で 9、internal.h で定義されている）証明書に加えて、サブジェクト証明書を処理します。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ
- **file** WOLFSSL 構造体にロードされる証明書を含むファイルの名前へのポインタ。証明書は PEM 形式でなければなりません。

See:

- wolfSSL_CTX_use_certificate_chain_file
- wolfSSL_CTX_use_certificate_chain_buffer
- wolfSSL_use_certificate_chain_buffer

Return:

- SSL_SUCCESS 成功時に返されます。

- SSL_FAILURE 関数呼び出しが失敗した場合に返されます。可能な原因には次のようなものがあります：ファイルが誤った形式、または引数 format を使用して誤った形式が与えられた、メモリ不足状態が発生した、ファイルで base16 のデコードが失敗した

Example

```
int ret = 0;
WOLFSSL* ctx;
...
ret = wolfSSL_use_certificate_chain_file(ssl, "./cert-chain.pem");
if (ret != SSL_SUCCESS) {
    // error loading cert file
}
...
```

A.3.2.22 function wolfSSL_use_RSAPrivateKey_file

```
int wolfSSL_use_RSAPrivateKey_file(
    WOLFSSL * ssl,
    const char * file,
    int format
)
```

この関数は、SSL 接続で使用されている RSA 秘密鍵を SSL セッション（WOLFSSL 構造体）にロードします。この関数は、wolfSSL が OpenSSL 互換 API を有効（–enable-openSSlExtra、#define OPENSSL_EXTRA）でビルドされている場合にのみ利用可能で、より一般的に使用される wolfSSL_use_PrivateKey_file() 関数と同じです。引数 file には、RSA 秘密鍵ファイルへのポインタが、フォーマットで指定された形式で含まれています。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ

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 成功時に返されます。
- SSL_FAILURE 関数呼び出しが失敗した場合に返されます。可能な原因には次のようなものがあります：ファイルが誤った形式、または引数 format を使用して誤った形式が与えられた、メモリ不足状態が発生した、ファイルで Base16 のデコードが失敗した

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
}
...
```

A.3.2.23 function wolfSSL_set_tlsext_status_type

```
long wolfSSL_set_tlsext_status_type(
    WOLFSSL * s,
    int type
)
```

この関数は、サーバが OCSP ステータス応答（OCSP ステイプルとも呼ばれる）を送受信するクライアントアプリケーションが要求されたときに呼び出されます。

Parameters:

- **s** ssl_new() 関数によって作成された WOLFSSL 構造体へのポインタ
- **type** ssl 拡張タイプ。TLSEXT_STATUSTYPE_ocsp のみ指定可。

See:

- wolfSSL_new
- wolfSSL_CTX_new
- wolfSSL_free
- wolfSSL_CTX_free

Return:

- 1 成功時に返されます。
- 0 エラー時に返されます。

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);
```

A.3.2.24 function wolfSSL_get_peer_chain

```
WOLFSSL_X509_CHAIN * wolfSSL_get_peer_chain(
    WOLFSSL * ssl
)
```

ピアの証明書チェーンを取得します。

See:

- wolfSSL_get_chain_count
- wolfSSL_get_chain_length
- wolfSSL_get_chain_cert
- wolfSSL_get_chain_cert_pem

Return:

- chain 正常にコールがピアの証明書チェーンを返します。
- 0 無効な WolfSSL ポインタが関数に渡されると返されます。

Example

none

A.3.2.25 function wolfSSL_get_chain_count

```
int wolfSSL_get_chain_count(
    WOLFSSL_X509_CHAIN * chain
)
```

ピアの証明書チェーン数を取得します。

See:

- wolfSSL_get_peer_chain
- wolfSSL_get_chain_length
- wolfSSL_get_chain_cert
- wolfSSL_get_chain_cert_pem

Return:

- Success 正常にコールがピアの証明書チェーン数を返します。
- 0 無効なチェーンポインタが関数に渡されると返されます。

Example

none

A.3.2.26 function wolfSSL_get_chain_length

```
int wolfSSL_get_chain_length(
    WOLFSSL_X509_CHAIN * chain,
    int idx
)
```

Index（IDX）のピアの ASN1.DER 証明書長をバイト単位で取得します。

Parameters:

- **chain** 有効な wolfssl_x509_chain 構造へのポインタ。

See:

- wolfSSL_get_peer_chain
- wolfSSL_get_chain_count
- wolfSSL_get_chain_cert
- wolfSSL_get_chain_cert_pem

Return:

- Success 正常にコールがインデックス別にピアの証明書長をバイト単位で返します。
- 0 無効なチェーンポインタが関数に渡されると返されます。

Example

none

A.3.2.27 function wolfSSL_get_chain_cert

```
unsigned char * wolfSSL_get_chain_cert(
    WOLFSSL_X509_CHAIN * chain,
    int idx
)
```

インデックス（IDX）でピアの ASN1.DER 証明書を取得します。

Parameters:

- **chain** 有効な wolfssl_x509_chain 構造へのポインタ。

See:

- wolfSSL_get_peer_chain
- wolfSSL_get_chain_count
- wolfSSL_get_chain_length
- wolfSSL_get_chain_cert_pem

Return:

- Success 正常にコールがインデックスでピアの証明書を返します。
- 0 無効なチェーンポインタが関数に渡されると返されます。

Example

none

A.3.2.28 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
)
```

インデックス（IDX）でピアの PEM 証明書を取得します。

Parameters:

- **chain** 有効な wolfssl_x509_chain 構造へのポインタ。

See:

- wolfSSL_get_peer_chain
- wolfSSL_get_chain_count
- wolfSSL_get_chain_length
- wolfSSL_get_chain_cert

Return:

- Success 正常にコールがインデックスでピアの証明書を返します。
- 0 無効なチェーンポインタが関数に渡されると返されます。

Example

none

A.3.2.29 function wolfSSL_get_sessionID

```
const unsigned char * wolfSSL_get_sessionID(
    const WOLFSSL_SESSION * s
)
```

セッションの ID を取得します。セッション ID は常に 32 バイトの長さです。

See: SSL_get_session

Return: id セッション ID。

Example

none

A.3.2.30 function wolfSSL_X509_get_serial_number

```
int wolfSSL_X509_get_serial_number(
    WOLFSSL_X509 * x509,
    unsigned char * in,
    int * inOutSz
)
```

ピアの証明書のシリアル番号を取得します。シリアル番号バッファ（IN）は少なくとも 32 バイト以上であり、入力として * INOUTSZ 引数として提供されます。関数を呼び出した後 * INOUTSZ は IN バッファに書き込まれた実際の長さをバイト単位で保持します。

Parameters:

- **in** シリアル番号バッファは少なくとも 32 バイトの長さであるべきです

See: SSL_get_peer_certificate

Return:

- SSL_SUCCESS 成功時に返されます。
- BAD_FUNC_ARG 関数の不良引数が見つかった場合に返されます。

Example

none

A.3.2.31 function wolfSSL_d2i_PKCS12_bio

```
WC_PKCS12 * wolfSSL_d2i_PKCS12_bio(
    WOLFSSL_BIO * bio,
    WC_PKCS12 ** pkcs12
)
```

WOLFSSL_D2I_PKCS12_BIO（D2I_PKCS12_BIO）は、WOLFSSL_BIO から構造 WC_PKCS12 への PKCS12 情報にコピーされます。この情報は、オプションの MAC 情報を保持するための構造とともにコンテンツに関する情報のリストとして構造内に分割されています。構造体 WC_PKCS12 で情報がチャンク（ただし復号化されていない）に分割された後、それはその後、呼び出しによって解析および復号化され得る。

Parameters:

- **bio** PKCS12 バッファを読み取るための WOLFSSL_BIO 構造。

See:

- wolfSSL_PKCS12_parse
- wc_PKCS12_free

Return:

- WC_PKCS12 WC_PKCS12 構造へのポインタ。
- Failure 関数に失敗した場合は NULL を返します。

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.3.2.32 function wolfSSL_i2d_PKCS12_bio

```
WC_PKCS12 * wolfSSL_i2d_PKCS12_bio(
    WOLFSSL_BIO * bio,
    WC_PKCS12 * pkcs12
)
```

WOLFSSL_I2D_PKCS12_BIO（I2D_PKCS12_BIO）は、構造 WC_PKCS12 から WOLFSSL_BIO への証明書情報にコピーされます。

Parameters:

- **bio** PKCS12 バッファを書き込むための WOLFSSL_BIO 構造。

See:

- wolfSSL_PKCS12_parse
- wc_PKCS12_free

Return:

- 1 成功のために。
- 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
```

A.3.2.33 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 は、configure コマンドへの-enable-openSSLAXTRA を追加することで有効にできます。それは復号化のためにトリプル DES と RC4 を使うことができるので、OpenSSlextra（–enable-des3 -enable-arc4）を

有効にするときにもこれらの機能を有効にすることをお勧めします。wolfssl は現在 RC2 をサポートしていませんので、RC2 での復号化は現在利用できません。これは、.p12 ファイルを作成するために OpenSSL コマンドラインで使用されるデフォルトの暗号化方式では注目すかもしれません。WOLFSSL_PKCS12_PARSE（PKCS12_PARSE）。この関数が最初に行っているのは、存在する場合は Mac が正しいチェックです。MAC が失敗した場合、関数は返され、保存されているコンテンツ情報のいずれかを復号化しようとしません。この関数は、バッグタイプを探している各コンテンツ情報を介して解析します。バッグタイプがわかっている場合は、必要に応じて復号化され、構築されている証明書のリストに格納されているか、見つかったキーとして保存されます。すべてのバッグを介して解析した後、見つかったキーは、一致するペアが見つかるまで証明書リストと比較されます。この一致するペアはキーと証明書として返され、オプションで見つかった証明書リストは stack_of 証明書として返されます。瞬間、CRL、秘密または安全なバッグがスキップされ、解析されません。デバッグプリントアウトを見ることで、これらまたは他の「不明」バッグがスキップされているかどうかがわかります。フレンドリー名などの追加の属性は、PKCS12 ファイルを解析するときにスキップされます。

Parameters:

- **pkcs12** wc_pkcs12 解析する構造
- **paswd** PKCS12 を復号化するためのパスワード。
- **pkey** PKCS12 からデコードされた秘密鍵を保持するための構造。
- **cert** PKCS12 から復号された証明書を保持する構造

See:

- wolfSSL_d2i_PKCS12_bio
- wc_PKCS12_free

Return:

- SSL_SUCCESS PKCS12 の解析に成功しました。
- SSL_FAILURE エラーケースに遭遇した場合

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.4.1 Functions

	Name
int	**wc_KeyPemToDer**(const unsigned char * pem, int pemSz, unsigned char * buff, int buffSz, const char * pass)PEM 形式の鍵を DER 形式に変換します。

	Name
int	**wc_CertPemToDer**(const unsigned char * pem, int pemSz, unsigned char * buff, int buffSz, int type) この関数は PEM 形式の証明書を DER 形式に変換します。内部では OpenSSL 互換 API の PemToDer を呼び出します。
int	**wc_GetPubKeyDerFromCert**(struct DecodedCert * cert, byte * derKey, word32 * derKeySz) この関数は公開鍵を DER 形式で DecodedCert 構造体から取り出します。wc_InitDecodedCert() と wc_ParseCert() を事前に呼び出しておく必要があります。wc_InitDecodedCert() は DER/ASN.1 エンコードされた証明書を受け付けます。PEM 形式の鍵を DER 形式で取得する場合には、wc_InitDecodedCert() より先に wc_CertPemToDer() を呼び出してください。
int	**wolfSSL_CTX_use_certificate_file**(WOLFSSL_CTX * ctx, const char * file, int format) この関数は証明書ファイルを SSL コンテキスト (WOLFSSL_CTX) にロードします。ファイルは引数 file によって提供されます。引数 format は、ファイルのフォーマットタイプ（SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM）を指定します。適切な使用法の例をご覧ください。
int	**wolfSSL_CTX_use_PrivateKey_file**(WOLFSSL_CTX * ctx, const char * file, int format) この関数は、秘密鍵ファイルを SSL コンテキスト (WOLFSSL_CTX) にロードします。ファイルは引数 file によって提供されます。引数 format は、次のファイルのフォーマットタイプを指定します：SSL_FILETYPE_ASN1　あるいは SSL_FILETYPE_PEM。適切な使用法の例をご覧ください。外部キーストアを使用し、秘密鍵を持っていない場合は、代わりに公開鍵を入力して crypto コールバックを登録して署名を処理することができます。このためには、crypto コールバックまたは PK コールバックを使用したコンフィギュレーションでビルドします。crypto コールバックを有効にするには、–enable-cryptocb または WOLF_CRYPTO_CB マクロを使用し、wc_CryptoCb_RegisterDevice を使用して暗号コールバックを登録し、wolfSSL_CTX_SetDevId を使用して関連する devid を設定します。

	Name
int	**wolfSSL_CTX_load_verify_locations**(WOLFSSL_CTX * ctx, const char * file, const char * path) この関数は、PEM 形式の CA 証明書ファイルを SSL コンテキスト (WOLFSSL_CTX) にロードします。これらの証明書は、信頼できるルート証明書として扱われ、SSL ハンドシェイク中にピアから受信した証明書を検証するために使用されます。引数 file によって提供されるルート証明書ファイルは、単一の証明書または複数の証明書を含むファイルでの場合があります。複数の CA 証明書が同じファイルに含まれている場合、wolfSSL はファイルに表示されているのと同じ順序でそれらをロードします。引数 path は、信頼できるルート CA の証明書を含むディレクトリの名前へのポインタです。引数 file が NULL ではない場合、パスが必要でない場合は NULL として指定できます。引数 path が指定されていてかつ NO_WOLFSSL_DIR が定義されていない場合には、wolfSSL ライブラリは指定されたディレクトリに存在するすべての CA 証明書をロードします。この関数はディレクトリ内のすべてのファイルをロードしようとします。この関数は、ヘッダーに “—–BEGIN CERTIFICATE—–” を持つ PEM フォーマットされた CERT_TYPE ファイルを期待しています。
const char **	**wolfSSL_get_system_CA_dirs**(word32 * num) この関数は、wolfSSL_CTX_load_system_CA_certs が呼び出されたときに、wolfSSL がシステム CA 証明書を検索するディレクトリを表す文字列の配列へのポインタを返します。
int	**wolfSSL_CTX_load_system_CA_certs**(WOLFSSL_CTX * ctx) この関数は、CA 証明書を OS 依存の CA 証明書ストアから WOLFSSL_CTX にロードしようとします。ロードされた証明書は信頼されます。サポートおよびテストされているプラットフォームは、Linux(Debian、Ubuntu、Gentoo、Fedora、RHEL)、Windows 10/11、Android、Apple OS X、iOS です。
int	**wolfSSL_CTX_use_certificate_chain_file**(WOLFSSL_CTX * ctx, const char * file) この関数は、証明書チェーンを SSL コンテキスト (WOLFSSL_CTX) にロードします。証明書チェーンを含むファイルは引数 file によって提供され、PEM 形式の証明書を含める必要があります。この関数は、最大 MAX_CHAIN_DEPTH（既定で 9、internal.h で定義されている）数の証明書を処理します。この数にはサブジェクト証明書を含みます。

	Name
int	**wolfSSL_CTX_der_load_verify_locations**(WOLFSSL_CTX * ctx, const char * file, int format) この関数は wolfSSL_CTX_load_verify_locations と似ていますが、DER フォーマットされた CA ファイルを SSL コンテキスト（WOLFSSL_CTX）にロードすることを許可します。それはまだ PEM 形式の CA ファイルをロードするためにも使用されるかもしれません。これらの証明書は、信頼できるルート証明書として扱われ、SSL ハンドシェイク中にピアから受信した証明書を検証するために使用されます。ファイル引数によって提供されるルート証明書ファイルは、単一の証明書または複数の証明書を含むファイルでも可能。複数の CA 証明書が同じファイルに含まれている場合、wolfSSL はファイルに表示されているのと同じ順序でそれらをロードします。引数 format は、証明書が SSL_FILETYPE_PEM または SSL_FILETYPE_ASN1（DER）のいずれかにある形式を指定します。wolfSSL_CTX_load_verify_locations とは異なり、この関数は特定のディレクトリパスからの CA 証明書のロードを許可しません。この関数は、wolfSSL ライブラリが WOLFSSL_DER_LOAD マクロが定義された状態でビルドされたときにのみ利用可能です。
void	**wolfSSL_SetCertCbCtx**(WOLFSSL * ssl, void * ctx) この関数は、検証コールバックのためのユーザー CTX オブジェクト情報を格納します。
void	**wolfSSL_CTX_SetCertCbCtx**(WOLFSSL_CTX * ctx, void * userCtx) この関数は、検証コールバックのためのユーザー CTX オブジェクト情報を格納します。
int	**wolfSSL_CTX_save_cert_cache**(WOLFSSL_CTX * ctx, const char * fname) この関数は Cert キャッシュをメモリからファイルに書き込みます。
int	**wolfSSL_CTX_restore_cert_cache**(WOLFSSL_CTX * ctx, const char * fname) この関数はファイルから証明書キャッシュを担当します。
int	**wolfSSL_CTX_memsave_cert_cache**(WOLFSSL_CTX * ctx, void * mem, int sz, int * used) この関数は証明書キャッシュをメモリに持続します。
int	**wolfSSL_CTX_get_cert_cache_memsize**(WOLFSSL_CTX * ctx)Certificate Cache Save バッファが必要なサイズを返します。
char *	**wolfSSL_X509_NAME_oneline**(WOLFSSL_X509_NAME * name, char * in, int sz) この関数は X509 の名前をバッファにコピーします。
WOLFSSL_X509_NAME *	**wolfSSL_X509_get_issuer_name**(WOLFSSL_X509 * cert) この関数は証明書発行者の名前を返します。
WOLFSSL_X509_NAME *	**wolfSSL_X509_get_subject_name**(WOLFSSL_X509 * cert) この関数は、wolfssl_x509 構造の件名メンバーを返します。

	Name
int	**wolfSSL_X509_get_isCA**(WOLFSSL_X509 * cert)WOLFSSL_X509 構造体の isCa メンバーをチェックして値を返します。
int	**wolfSSL_X509_NAME_get_text_by_NID**(WOLFSSL_X509_NAME * name, int nid, char * buf, int len) この関数は、渡された NID 値に関連するテキストを取得します。
int	**wolfSSL_X509_get_signature_type**(WOLFSSL_X509 * cert) この関数は、WOLFSSL_X509 構造体の sigOID メンバーに格納されている値を返します。
int	**wolfSSL_X509_get_signature**(WOLFSSL_X509 * x509, unsigned char * buf, int * bufSz)x509 署名を取得し、それをバッファに保存します。
int	**wolfSSL_X509_STORE_add_cert**(WOLFSSL_X509_STORE * store, WOLFSSL_X509 * x509) この関数は、WOLFSSL_X509_STRE 構造体に証明書を追加します。
WOLFSSL_STACK *	**wolfSSL_X509_STORE_CTX_get_chain**(WOLFSSL_X509_STORE_CT * ctx) この関数は、WOLFSSL_X509_STORE_CTX 構造体のチェーン変数の getter 関数です。現在チェーンは取り込まれていません。
int	**wolfSSL_X509_STORE_set_flags**(WOLFSSL_X509_STORE * store, unsigned long flag) この関数は、渡された WOLFSSL_X509_STORE 構造体の動作を変更するためのフラグを取ります。使用されるフラグの例は WOLFSSL_CRL_CHECK です。
const byte *	**wolfSSL_X509_notBefore**(WOLFSSL_X509 * x509) この関数は BYTE アレイとして符号化された “not before” 要素を返します。
const byte *	**wolfSSL_X509_notAfter**(WOLFSSL_X509 * x509) この関数は、BYTE 配列として符号化された “not after” 要素を返します。
const char *	**wolfSSL_get_psk_identity_hint**(const WOLFSSL *) この関数は PSK アイデンティティヒントを返します。
const char *	**wolfSSL_get_psk_identity**(const WOLFSSL *) 関数は、配列構造の Client_Identity メンバーへの定数ポインタを返します。
int	**wolfSSL_CTX_use_psk_identity_hint**(WOLFSSL_CTX * ctx, const char * hint) この関数は、WOLFSSL_CTX 構造体の server_hint メンバーに HINT 引数を格納します。
int	**wolfSSL_use_psk_identity_hint**(WOLFSSL * ssl, const char * hint) この関数は、wolfssl 構造内の配列構造の server_hint メンバーに HINT 引数を格納します。
WOLFSSL_X509 *	**wolfSSL_get_peer_certificate**(WOLFSSL * ssl) この関数はピアの証明書を取得します。
WOLFSSL_X509 *	**wolfSSL_get_chain_X509**(WOLFSSL_X509_CHAIN * chain, int idx) この関数は、証明書のチェーンからのピアの WOLFSSL_X509 構造体をインデックス（IDX）で取得します。

	Name
char *	**wolfSSL_X509_get_subjectCN**(WOLFSSL_X509 *) 証明書から件名の共通名を返します。
const unsigned char *	**wolfSSL_X509_get_der**(WOLFSSL_X509 * x509, int * outSz) この関数は、wolfssl_x509 構造体の DER エンコードされた証明書を取得します。
WOLFSSL_ASN1_TIME *	**wolfSSL_X509_get_notAfter**(WOLFSSL_X509 *) この関数は、x509 が null のかどうかを確認し、そうでない場合は、x509 構造体のノッカスメンバーを返します。
int	**wolfSSL_X509_version**(WOLFSSL_X509 *) この関数は X509 証明書のバージョンを取得します。
WOLFSSL_X509 *	**wolfSSL_X509_d2i_fp**(WOLFSSL_X509 ** x509, FILE * file)no_stdio_filesystem が定義されている場合、この関数はヒープメモリを割り当て、wolfssl_x509 構造を初期化してそれにポインタを返します。
WOLFSSL_X509 *	**wolfSSL_X509_load_certificate_file**(const char * fname, int format) 関数は X509 証明書をメモリにロードします。
unsigned char *	**wolfSSL_X509_get_device_type**(WOLFSSL_X509 * x509, unsigned char * in, int * inOutSz) この関数は、デバイスの種類を X509 構造からバッファにコピーします。
unsigned char *	**wolfSSL_X509_get_hw_type**(WOLFSSL_X509 * x509, unsigned char * in, int * inOutSz) この関数は、wolfssl_x509 構造の HWType メンバーをバッファにコピーします。
unsigned char *	**wolfSSL_X509_get_hw_serial_number**(WOLFSSL_X509 * x509, unsigned char * in, int * inOutSz) この関数は X509 オブジェクトの hwserialNum メンバを返します。
int	**wolfSSL_SetTmpDH**(WOLFSSL * ssl, const unsigned char * p, int pSz, const unsigned char * g, int gSz) サーバー DIFFIE-HELLMAN エフェメラルパラメータ設定。この関数は、サーバーが DHE を使用する暗号スイートをネゴシエートしている場合に使用するグループパラメータを設定します。
int	**wolfSSL_SetTmpDH_buffer**(WOLFSSL * ssl, const unsigned char * b, long sz, int format) 関数は wolfssl_settmph_buffer_wrapper を呼び出します。これは Diffie-Hellman パラメータのラッパーです。
int	**wolfSSL_SetTmpDH_file**(WOLFSSL * ssl, const char * f, int format) この関数は、wolfssl_settmph_file_wrapper を呼び出してサーバ diffie-hellman パラメータを設定します。
int	**wolfSSL_CTX_SetTmpDH**(WOLFSSL_CTX * ctx, const unsigned char * p, int pSz, const unsigned char * g, int gSz) サーバー CTX Diffie-Hellman のパラメータを設定します。

	Name
int	**wolfSSL_CTX_SetTmpDH_buffer**(WOLFSSL_CTX * ctx, const unsigned char * b, long sz, int format)wolfssl_settmph_buffer_wrapper を呼び出すラッパー関数
int	**wolfSSL_CTX_SetTmpDH_file**(WOLFSSL_CTX * ctx, const char * f, int format) この関数は、wolfssl_settmph_file_wrapper を呼び出してサーバー Diffie-Hellman パラメータを設定します。
int	**wolfSSL_CTX_SetMinDhKey_Sz**(WOLFSSL_CTX * ctx, word16) この関数は、WOLFSSL_CTX 構造体の minkkeysz メンバーにアクセスして、Diffie Hellman 鍵サイズの最小サイズ（ビット単位）を設定します。
int	**wolfSSL_SetMinDhKey_Sz**(WOLFSSL * ssl, word16 keySz_bits)WolfSSL 構造の Diffie-Hellman 鍵の最小サイズ（ビット単位）を設定します。
int	**wolfSSL_CTX_SetMaxDhKey_Sz**(WOLFSSL_CTX * ctx, word16 keySz_bits) この関数は、WOLFSSL_CTX 構造体の maxdhkeysz メンバーにアクセスして、Diffie Hellman 鍵サイズの最大サイズ（ビット単位）を設定します。
int	**wolfSSL_SetMaxDhKey_Sz**(WOLFSSL * ssl, word16 keySz_bits)WolfSSL 構造の Diffie-Hellman 鍵の最大サイズ（ビット単位）を設定します。
int	**wolfSSL_GetDhKey_Sz**(WOLFSSL *) オプション構造のメンバーである DHKEYSZ（ビット内）の値を返します。この値は、Diffie-Hellman 鍵サイズをバイト単位で表します。
int	**wolfSSL_CTX_SetMinRsaKey_Sz**(WOLFSSL_CTX * ctx, short keySz)WOLFSSL_CTX 構造体と wolfssl_cert_manager 構造の両方で最小 RSA 鍵サイズを設定します。
int	**wolfSSL_SetMinRsaKey_Sz**(WOLFSSL * ssl, short keySz)WolfSSL 構造にある RSA のためのビットで最小許容鍵サイズを設定します。
int	**wolfSSL_CTX_SetMinEccKey_Sz**(WOLFSSL_CTX * ssl, short keySz)wolf_ctx 構造体と wolfssl_cert_manager 構造体の ECC 鍵の最小サイズをビット単位で設定します。
int	**wolfSSL_SetMinEccKey_Sz**(WOLFSSL * ssl, short keySz) オプション構造の MineCckeysz メンバーの値を設定します。オプション構造体は、WolfSSL 構造のメンバーであり、SSL パラメータを介してアクセスされます。
int	**wolfSSL_make_eap_keys**(WOLFSSL * ssl, void * key, unsigned int len, const char * label) この関数は、eap_tls と eap-ttls によって、マスターシークレットからキーイングマテリアルを導出します。

	Name
int	**wolfSSL_CTX_load_verify_buffer**(WOLFSSL_CTX * ctx, const unsigned char * in, long sz, int format) この関数は CA 証明書バッファを WolfSSL コンテキストにロードします。バッファ以外のバージョンのように動作し、ファイルの代わりに入力としてバッファと呼ばれる機能が異なるだけです。バッファはサイズ SZ の引数によって提供されます。形式バッファのフォーマットタイプを指定します。SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM。フォーマットが PEM 内にある限り、バッファあたり複数の CA 証明書をロードすることができます。適切な使用法の例をご覧ください。
int	**wolfSSL_CTX_load_verify_buffer_ex**(WOLFSSL_CTX * ctx, const unsigned char * in, long sz, int format, int userChain, word32 flags) この関数は CA 証明書バッファを WolfSSL コンテキストにロードします。バッファ以外のバージョンのように動作し、ファイルの代わりに入力としてバッファと呼ばれる機能が異なるだけです。バッファはサイズ SZ の引数によって提供されます。形式バッファのフォーマットタイプを指定します。SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM。フォーマットが PEM 内にある限り、バッファあたり複数の CA 証明書をロードすることができます。_EX バージョンは PR 2413 に追加され、UserChain と Flags の追加の引数をサポートします。
int	**wolfSSL_CTX_load_verify_chain_buffer_format**(WOLFSSL_CTX * ctx, const unsigned char * in, long sz, int format) この関数は、CA 証明書チェーンバッファを WolfSSL コンテキストにロードします。バッファ以外のバージョンのように動作し、ファイルの代わりに入力としてバッファと呼ばれる機能が異なるだけです。バッファはサイズ SZ の引数によって提供されます。形式バッファのフォーマットタイプを指定します。SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM。フォーマットが PEM 内にある限り、バッファあたり複数の CA 証明書をロードすることができます。適切な使用法の例をご覧ください。
int	**wolfSSL_CTX_use_certificate_buffer**(WOLFSSL_CTX * ctx, const unsigned char * in, long sz, int format) この関数は証明書バッファを WolfSSL コンテキストにロードします。バッファ以外のバージョンのように動作し、ファイルの代わりに入力としてバッファと呼ばれる機能が異なるだけです。バッファはサイズ SZ の引数によって提供されます。形式バッファのフォーマットタイプを指定します。SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM。適切な使用法の例をご覧ください。

	Name
int	**wolfSSL_CTX_use_PrivateKey_buffer**(WOLFSSL_CTX * ctx, const unsigned char * in, long sz, int format) この関数は、秘密鍵バッファを SSL コンテキストにロードします。バッファ以外のバージョンのように動作し、ファイルの代わりに入力としてバッファと呼ばれる機能が異なるだけです。バッファはサイズ SZ の引数によって提供されます。形式バッファのフォーマットタイプを指定します。SSL_FILETYPE_ASN1OR SSL_FILETYPE_PEM。適切な使用法の例をご覧ください。
int	**wolfSSL_CTX_use_certificate_chain_buffer**(WOLFSSL_CTX * ctx, const unsigned char * in, long sz) この関数は、証明書チェーンバッファを WolfSSL コンテキストにロードします。バッファ以外のバージョンのように動作し、ファイルの代わりに入力としてバッファと呼ばれる機能が異なるだけです。バッファはサイズ SZ の引数によって提供されます。バッファは PEM 形式で、ルート証明書で終わる対象の証明書から始めてください。適切な使用法の例をご覧ください。
int	**wolfSSL_use_certificate_buffer**(WOLFSSL * ssl, const unsigned char * in, long sz, int format) この関数は、証明書バッファを WolfSSL オブジェクトにロードします。バッファ以外のバージョンのように動作し、ファイルの代わりに入力としてバッファと呼ばれる機能が異なるだけです。バッファはサイズ SZ の引数によって提供されます。形式バッファのフォーマットタイプを指定します。SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM。適切な使用法の例をご覧ください。
int	**wolfSSL_use_PrivateKey_buffer**(WOLFSSL * ssl, const unsigned char * in, long sz, int format) この関数は、秘密鍵バッファを WolfSSL オブジェクトにロードします。バッファ以外のバージョンのように動作し、ファイルの代わりに入力としてバッファと呼ばれる機能が異なるだけです。バッファはサイズ SZ の引数によって提供されます。形式バッファのフォーマットタイプを指定します。SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM。適切な使用法の例をご覧ください。

	Name
int	**wolfSSL_use_certificate_chain_buffer**(WOLFSSL * ssl, const unsigned char * in, long sz) この関数は、証明書チェーンバッファを WolfSSL オブジェクトにロードします。バッファ以外のバージョンのように動作し、ファイルの代わりに入力としてバッファと呼ばれる機能が異なるだけです。バッファはサイズ SZ の引数によって提供されます。バッファは PEM 形式で、ルート証明書で終わる対象の証明書から始めてください。適切な使用法の例をご覧ください。
int	**wolfSSL_UnloadCertsKeys**(WOLFSSL *) この関数は、SSL が所有する証明書または鍵をアンロードします。
int	**wolfSSL_GetIVSize**(WOLFSSL *)WolfSSL 構造体に保持されている Specs 構造体の IV_SIZE メンバーを返します。
void	**wolfSSL_KeepArrays**(WOLFSSL *) 通常、SSL ハンドシェイクの最後に、WolfSSL は一時的なアレイを解放します。ハンドシェイクが始まる前にこの関数を呼び出すと、WolfSSL は一時的な配列を解放するのを防ぎます。Wolfssl_get_keys() または PSK のヒントなどのものには、一時的な配列が必要になる場合があります。ユーザが一時的な配列で行われると、wolfssl_freearray() のいずれかが即座にリソースを解放することができ、あるいは、関連する SSL オブジェクトが解放されたときにリソースが解放されるようになる可能性がある。
void	**wolfSSL_FreeArrays**(WOLFSSL *) 通常、SSL ハンドシェイクの最後に、WolfSSL は一時的なアレイを解放します。wolfssl_keeparrays() がハンドシェイクの前に呼び出された場合、WolfSSL は一時的な配列を解放しません。この関数は一時的な配列を明示的に解放し、ユーザーが一時的な配列で行われたときに呼び出されるべきであり、SSL オブジェクトがこれらのリソースを解放するのを待ったくない。
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)TLS キーを導き出すための外部のラッパー。
int	**wolfSSL_X509_get_ext_by_NID**(const WOLFSSL_X509 * x509, int nid, int lastPos) この機能は、渡された NID 値に一致する拡張索引を探して返します。
void *	**wolfSSL_X509_get_ext_d2i**(const WOLFSSL_X509 * x509, int nid, int * c, int * idx) この関数は、渡された NID 値に合った拡張子を探して返します。

	Name
int	**wolfSSL_X509_digest**(const WOLFSSL_X509 * x509, const WOLFSSL_EVP_MD * digest, unsigned char * buf, unsigned int * len) この関数は DER 証明書のハッシュを返します。
int	**wolfSSL_use_PrivateKey**(WOLFSSL * ssl, WOLFSSL_EVP_PKEY * pkey) これは WolfSSL 構造の秘密鍵を設定するために使用されます。
int	**wolfSSL_use_PrivateKey_ASN1**(int pri, WOLFSSL * ssl, unsigned char * der, long derSz) これは WolfSSL 構造の秘密鍵を設定するために使用されます。DER フォーマットのキーバッファが予想されます。
int	**wolfSSL_use_RSAPrivateKey_ASN1**(WOLFSSL * ssl, unsigned char * der, long derSz) これは WolfSSL 構造の秘密鍵を設定するために使用されます。DER フォーマットの RSA キーバッファが予想されます。
WOLFSSL_DH *	**wolfSSL_DSA_dup_DH**(const WOLFSSL_DSA * r) この関数は、DSA のパラメータを新しく作成された WOLFSSL_DH 構造体に重複しています。
WOLFSSL_X509 *	**wolfSSL_d2i_X509_bio**(WOLFSSL_BIO * bio, WOLFSSL_X509 ** x509) この関数は BIO から DER バッファを取得し、それを WolfSSL_X509 構造に変換します。
WOLFSSL_X509 *	**wolfSSL_PEM_read_bio_X509_AUX**(WOLFSSL_BIO * bp, WOLFSSL_X509 ** x, wc_pem_password_cb * cb, void * u) この関数は wolfssl_pem_read_bio_x509 と同じように動作します。AUX は、信頼できる/拒否されたユースケースや人間の読みやすさのためのフレンドリーな名前などの追加情報を含むことを意味します。
long	**wolfSSL_CTX_set_tmp_dh**(WOLFSSL_CTX * ctx, WOLFSSL_DH * dh)WOLFSSL_CTX 構造体の DH メンバーを diffie-hellman パラメータで初期化します。
WOLFSSL_DSA *	**wolfSSL_PEM_read_bio_DSAparams**(WOLFSSL_BIO * bp, WOLFSSL_DSA ** x, wc_pem_password_cb * cb, void * u) この関数は、BIO の PEM バッファから DSA パラメータを取得します。
	WOLF_STACK_OF(WOLFSSL_X509) const この関数はピアの証明書チェーンを取得します。
char *	**wolfSSL_X509_get_next_altname**(WOLFSSL_X509 * x509) この関数は、存在する場合は、ピア証明書から altname を返します。
WOLFSSL_ASN1_TIME *	**wolfSSL_X509_get_notBefore**(WOLFSSL_X509 * x509) 関数は、x509 が null のかどうかを確認し、そうでない場合は、WOLFSSL_X509 構造体の NotBefore メンバーを返します。

A.4.2 Functions Documentation

A.4.2.1 function wc_KeyPemToDer

```
int wc_KeyPemToDer(
    const unsigned char * pem,
    int pemSz,
    unsigned char * buff,
    int buffSz,
    const char * pass
)
```

PEM 形式の鍵を DER 形式に変換します。

Parameters:

- **pem** PEM 形式の証明書データへのポインタ
- **pemSz** PEM 形式の証明書データのサイズ
- **buff** DerBuffer 構造体の buffer メンバーのコピーへのポインタ
- **buffSz** DerBuffer 構造体の buffer メンバーへ確保されたバッファのサイズ
- **pass** パスワード

See: wc_PemToDer

Return:

- 変換に成功した際には出力バッファに書き込んだデータサイズを返します。
- エラー発生時には負の整数値を返します。

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.
}
```

A.4.2.2 function wc_CertPemToDer

```
int wc_CertPemToDer(
    const unsigned char * pem,
    int pemSz,
    unsigned char * buff,
    int buffSz,
    int type
)
```

この関数は PEM 形式の証明書を DER 形式に変換します。内部では OpenSSL 互換 API の PemToDer を呼び出します。

Parameters:

- **pem** PEM 形式の証明書を含むバッファへのポインタ
- **pemSz** PEM 形式の証明書を含むバッファのサイズ
- **buff** DER 形式に変換した証明書データの出力先バッファへのポインタ

- **buffSz** 出力先バッファのサイズ
- **type** 証明書のタイプ。asn_public.h で定義の enum CertType の値。

See: wc_PemToDer

Return: バッファに出力したサイズを返します。

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
}
```

A.4.2.3 function wc_GetPubKeyDerFromCert

```
int wc_GetPubKeyDerFromCert(
    struct DecodedCert * cert,
    byte * derKey,
    word32 * derKeySz
)
```

この関数は公開鍵を DER 形式で DecodedCert 構造体から取り出します。wc_InitDecodedCert() と wc_ParseCert() を事前に呼び出しておく必要があります。wc_InitDecodedCert() は DER/ASN.1 エンコードされた証明書を受け付けます。PEM 形式の鍵を DER 形式で取得する場合には、wc_InitDecodedCert() より先に wc_CertPemToDer() を呼び出してください。

Parameters:

- **cert** X.509 証明書を保持した DecodedCert 構造体へのポインタ
- **derKey** DER 形式の公開鍵を出力する先のバッファへのポインタ
- **derKeySz** [IN/OUT] 入力時には derKey で与えられるバッファのサイズ, 出力時には公開鍵のサイズを保持します。もし、derKey が NULL で渡された場合には, derKeySz には必要なバッファサイズが格納され、LENGTH_ONLY_E が戻り値として返されます。

See: wc_GetPubKeyDerFromCert

Return:

- 成功時に 0 を返します。エラー発生時には負の整数を返します。
- LENGTH_ONLY_E derKey が NULL の際に返されます。

A.4.2.4 function wolfSSL_CTX_use_certificate_file

```
int wolfSSL_CTX_use_certificate_file(
    WOLFSSL_CTX * ctx,
    const char * file,
    int format
)
```

この関数は証明書ファイルを SSL コンテキスト (WOLFSSL_CTX) にロードします。ファイルは引数 file によって提供されます。引数 format は、ファイルのフォーマットタイプ（SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM）を指定します。適切な使用法の例をご覧ください。

Parameters:

- **ctx** wolfSSL_CTX_new()を使用して作成された WOLFSSL_CTX 構造体へのポインタ
- **file** ロードする証明書を含むファイルパス文字列。
- **format** ロードする証明書のフォーマット：SSL_FILETYPE_ASN1 あるいは SSL_FILETYPE_PEM

See:

- wolfSSL_CTX_use_certificate_buffer
- wolfSSL_use_certificate_file
- wolfSSL_use_certificate_buffer

Return:

- SSL_SUCCESS 成功した場合に返されます。に返されます。
- SSL_FAILURE 失敗時に返されます。失敗した場合の可能な原因としては、ファイルが誤った形式の場合、または引数 format を使用して誤ったフォーマットが指定されている、あるいはファイルが存在しない、あるいは読み取ることができない、または破損している、メモリ不足が発生、Base16 のデコードに失敗しているなどの原因が考えられます。

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
}
...
```

A.4.2.5 function wolfSSL_CTX_use_PrivateKey_file

```
int wolfSSL_CTX_use_PrivateKey_file(
    WOLFSSL_CTX * ctx,
    const char * file,
    int format
)
```

この関数は、秘密鍵ファイルを SSL コンテキスト (WOLFSSL_CTX) にロードします。ファイルは引数 file によって提供されます。引数 format は、次のファイルのフォーマットタイプを指定します：SSL_FILETYPE_ASN1 あるいは SSL_FILETYPE_PEM。適切な使用法の例をご覧ください。外部キーストアを使用し、秘密鍵を持っていない場合は、代わりに公開鍵を入力して crypto コールバックを登録して署名を処理することができます。このためには、crypto コールバックまたは PK コールバックを使用したコンフィギュレーションでビルドします。crypto コールバックを有効にするには、–enable-cryptocb または WOLF_CRYPTO_CB マクロを使用し、wc_CryptoCb_RegisterDevice を使用して暗号コールバックを登録し、wolfSSL_CTX_SetDevId を使用して関連する devid を設定します。

Parameters:

- **なし** *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
}
...
```

See:

- wolfSSL_CTX_use_PrivateKey_buffer
- wolfSSL_use_PrivateKey_file
- wolfSSL_use_PrivateKey_buffer
- wc_CryptoCb_RegisterDevice
- wolfSSL_CTX_SetDevId

Return:

- SSL_SUCCESS 成功した場合に返されます。に返されます。
- SSL_FAILURE 関数呼び出しが失敗した場合の可能な原因としては、ファイルが誤った形式の場合、または引数 format を使用して誤ったフォーマットが指定されている、あるいはファイルが存在しない、あるいは読み取ることができない、または破損している、メモリ不足が発生、Base16 のデコードに失敗しているなどの原因が考えられます

A.4.2.6 function wolfSSL_CTX_load_verify_locations

```
int wolfSSL_CTX_load_verify_locations(
    WOLFSSL_CTX * ctx,
    const char * file,
    const char * path
)
```

この関数は、PEM 形式の CA 証明書ファイルを SSL コンテキスト (WOLFSSL_CTX) にロードします。これらの証明書は、信頼できるルート証明書として扱われ、SSL ハンドシェイク中にピアから受信した証明書を検証するために使用されます。引数 file によって提供されるルート証明書ファイルは、単一の証明書または複数の証明書を含むファイルでの場合があります。複数の CA 証明書が同じファイルに含まれている場合、wolfSSL はファイルに表示されているのと同じ順序でそれらをロードします。引数 path は、信頼できるルート CA の証明書を含むディレクトリの名前へのポインタです。引数 file が NULL ではない場合、パスが必要でない場合は NULL として指定できます。引数 path が指定されていてかつ NO_WOLFSSL_DIR が定義されていない場合には、wolfSSL ライブラリは指定されたディレクトリに存在するすべての CA 証明書をロードします。この関数はディレクトリ内のすべてのファイルをロードしようとします。この関数は、ヘッダーに “—–BEGIN CERTIFICATE—–” を持つ PEM フォーマットされた CERT_TYPE ファイルを期待しています。

Parameters:

- **ctx** wolfSSL_CTX_new()で作成された SSL コンテキストへのポインタ。
- **file** PEM 形式の CA 証明書を含むファイルの名前へのポインタ。
- **path** CA 証明書を含んでいるディレクトリのディレクトリの名前へのポインタ。

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 成功した場合に返されます。に返されます。
- SSL_FAILURE CTX が NULL の場合、またはファイルとパスの両方が NULL の場合に返されます。
- SSL_BAD_FILETYPE ファイルが間違った形式である場合に返されます。
- SSL_BAD_FILE ファイルが存在しない場合、読み込めない場合、または破損している場合に返されます。

- MEMORY_E メモリ不足状態が発生した場合に返されます。
- ASN_INPUT_E base16 デコードがファイルに対して失敗した場合に返されます。
- ASN_BEFORE_DATE_E 現在の日付が使用開始日より前の場合に返されます。
- ASN_AFTER_DATE_E 現在の日付が使用期限後より後の場合に返されます。
- BUFFER_E チェーンバッファが受信バッファよりも大きい場合に返されます。
- BAD_PATH_ERROR opendir() がパスを開こうとして失敗した場合に返されます。

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
}
...
```

A.4.2.7 function wolfSSL_get_system_CA_dirs

```
const char ** wolfSSL_get_system_CA_dirs(
    word32 * num
)
```

この関数は、wolfSSL_CTX_load_system_CA_certs が呼び出されたときに、wolfSSL がシステム CA 証明書を検索するディレクトリを表す文字列の配列へのポインタを返します。

Parameters:

- **num** word32 型変数へのポインタ。文字列配列の長さを格納します。

See:

- wolfSSL_CTX_load_system_CA_certs
- wolfSSL_CTX_load_verify_locations
- wolfSSL_CTX_load_verify_locations_ex

Return:

- 成功時には文字列配列へのポインタを返します。
- NULL 失敗時に返します。

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]);
}
...
```

A.4.2.8 function wolfSSL_CTX_load_system_CA_certs

```
int wolfSSL_CTX_load_system_CA_certs(
    WOLFSSL_CTX * ctx
)
```

この関数は、CA 証明書を OS 依存の CA 証明書ストアから WOLFSSL_CTX にロードしようとします。ロードされた証明書は信頼されます。サポートおよびテストされているプラットフォームは、Linux(Debian、Ubuntu、Gentoo、Fedora、RHEL)、Windows 10/11、Android、Apple OS X、iOS です。

Parameters:

- **ctx** wolfSSL_CTX_new()で生成された WOLFSSL_CTX 構造体へのポインタ。

See:

- wolfSSL_get_system_CA_dirs
- wolfSSL_CTX_load_verify_locations
- wolfSSL_CTX_load_verify_locations_ex

Return:

- WOLFSSL_SUCCESS 成功時に返されます。
- WOLFSSL_BAD_PATH システム CA 証明書がロードできなかった場合に返されます。
- WOLFSSL_FAILURE そのほかのエラー発生時（Windows 証明書ストアが正常にクローズされない等）

Example

```
int ret = 0;
WOLFSSL_CTX* ctx;
...
ret = wolfSSL_CTX_load_system_CA_certs(ctx,);
if (ret != WOLFSSL_SUCCESS) {
    // error loading system CA certs
}
...
```

A.4.2.9 function wolfSSL_CTX_use_certificate_chain_file

```
int wolfSSL_CTX_use_certificate_chain_file(
    WOLFSSL_CTX * ctx,
    const char * file
)
```

この関数は、証明書チェーンを SSL コンテキスト (WOLFSSL_CTX) にロードします。証明書チェーンを含むファイルは引数 file によって提供され、PEM 形式の証明書を含める必要があります。この関数は、最大 MAX_CHAIN_DEPTH（既定で 9、internal.h で定義されている）数の証明書を処理します。この数にはサブジェクト証明書を含みます。

Parameters:

- **ctx** wolfSSL_CTX_new()で生成された WOLFSSL_CTX 構造体へのポインタ。

See:

- wolfSSL_CTX_use_certificate_file
- wolfSSL_CTX_use_certificate_buffer
- wolfSSL_use_certificate_file
- wolfSSL_use_certificate_buffer

Return:

- SSL_SUCCESS 成功時に返されます。
- SSL_FAILURE 関数呼び出しが失敗した場合、可能な原因としては：誤った形式である場合、または「format」引数を使用して誤ったフォーマットが指定されている場合、ファイルが存在しない、読み取れない、または破損している、メモリ枯渇などが考えられます。

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
}
...
```

A.4.2.10 function wolfSSL_CTX_der_load_verify_locations

```
int wolfSSL_CTX_der_load_verify_locations(
    WOLFSSL_CTX * ctx,
    const char * file,
    int format
)
```

この関数は wolfSSL_CTX_load_verify_locations と似ていますが、DER フォーマットされた CA ファイルを SSL コンテキスト（WOLFSSL_CTX）にロードすることを許可します。それはまだ PEM 形式の CA ファイルをロードするためにも使用されるかもしれません。これらの証明書は、信頼できるルート証明書として扱われ、SSL ハンドシェイク中にピアから受信した証明書を検証するために使用されます。ファイル引数によって提供されるルート証明書ファイルは、単一の証明書または複数の証明書を含むファイルでも可能。複数の CA 証明書が同じファイルに含まれている場合、wolfSSL はファイルに表示されているのと同じ順序でそれらをロードします。引数 format は、証明書が SSL_FILETYPE_PEM または SSL_FILETYPE_ASN1（DER）のいずれかにある形式を指定します。wolfSSL_CTX_load_verify_locations とは異なり、この関数は特定のディレクトリパスからの CA 証明書のロードを許可しません。この関数は、wolfSSL ライブラリが WOLFSSL_DER_LOAD マクロが定義された状態でビルドされたときにのみ利用可能です。

Parameters:

- **ctx** wolfSSL_CTX_new()を使用して作成された WOLFSSL_CTX 構造体へのポインタ
- **file** wolfssl SSL コンテキストにロードされる CA 証明書を含むファイルの名前をフォーマットで指定された形式で指定します。

See:

- wolfSSL_CTX_load_verify_locations
- wolfSSL_CTX_load_verify_buffer

Return:

- SSL_SUCCESS 成功時に返されます。
- SSL_FAILURE 失敗すると返されます。

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
}
...
```

A.4.2.11 function wolfSSL_SetCertCbCtx

```
void wolfSSL_SetCertCbCtx(
```

```
    WOLFSSL * ssl,
    void * ctx
)
```

この関数は、検証コールバックのためのユーザー CTX オブジェクト情報を格納します。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ。
- **ctx** ボイドポインタ。WOLFSSL 構造体の verifyCbCtx メンバーにセットされます。

See:

- wolfSSL_CTX_save_cert_cache
- wolfSSL_CTX_restore_cert_cache
- wolfSSL_CTX_set_verify

Return: なし

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.
}
```

A.4.2.12 function wolfSSL_CTX_SetCertCbCtx

```
void wolfSSL_CTX_SetCertCbCtx(
    WOLFSSL_CTX * ctx,
    void * userCtx
)
```

この関数は、検証コールバックのためのユーザー CTX オブジェクト情報を格納します。

Parameters:

- **ctx** WOLFSSL_CTX 構造体へのポインタ。
- **ctx** ボイドポインタ。WOLFSSL_CTX 構造体の verifyCbCtx メンバーにセットされます。

See:

- wolfSSL_CTX_save_cert_cache
- wolfSSL_CTX_restore_cert_cache
- wolfSSL_CTX_set_verify

Return: なし

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.
}
```

A.4.2.13 function wolfSSL_CTX_save_cert_cache

```
int wolfSSL_CTX_save_cert_cache(
    WOLFSSL_CTX * ctx,
    const char * fname
)
```

この関数は Cert キャッシュをメモリからファイルに書き込みます。

Parameters:

- **ctx** WOLFSSL_CTX 構造体へのポインタ、証明書情報を保持します。
- **fname** 出力先ファイル名へのポインタ

See:

- CM_SaveCertCache
- DoMemSaveCertCache

Return:

- SSL_SUCCESS CM_SaveCertCache が正常に終了した場合。
- BAD_FUNC_ARG 引数のいずれかの引数が NULL の場合に返されます。
- SSL_BAD_FILE 証明書キャッシュ保存ファイルを開くことができなかった場合。
- BAD_MUTEX_E ロックミューテックスが失敗した場合
- MEMORY_E メモリの割り当てに失敗しました。
- FWRITE_ERROR 証明書キャッシュファイルの書き込みに失敗しました。

Example

```
WOLFSSL_CTX* ctx = WOLFSSL_CTX_new( protocol def );
const char* fname;
...
if(wolfSSL_CTX_save_cert_cache(ctx, fname)){
    // file was written.
}
```

A.4.2.14 function wolfSSL_CTX_restore_cert_cache

```
int wolfSSL_CTX_restore_cert_cache(
    WOLFSSL_CTX * ctx,
    const char * fname
)
```

この関数はファイルから証明書キャッシュを担当します。

Parameters:

- **ctx** WOLFSSL_CTX 構造体へのポインタ、証明書情報を保持します。
- **fname** 証明書キャッシュを読み取るファイル名へのポインタ。

See:

- CM_RestoreCertCache
- XFOPEN

Return:

- SSL_SUCCESS 正常に実行された場合に返されます。

- SSL_BAD_FILE XFOPEN が XBADFILE を返すと返されます。ファイルが破損しています。
- MEMORY_E TEMP バッファの割り当てられたメモリが失敗した場合に返されます。
- BAD_FUNC_ARG 引数 fname または引数 ctx が NULL である場合に返されます。

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
}
```

A.4.2.15 function wolfSSL_CTX_memsave_cert_cache

```
int wolfSSL_CTX_memsave_cert_cache(
    WOLFSSL_CTX * ctx,
    void * mem,
    int sz,
    int * used
)
```

この関数は証明書キャッシュをメモリに持続します。

Parameters:

- **ctx** wolfSSL_CTX_new()を使用して作成された WOLFSSL_CTX 構造体へのポインタ。
- **mem** 宛先への void ポインタ（出力バッファ）。
- **sz** 出力バッファのサイズ。
- **used** 証明書キャッシュヘッダーのサイズを格納する変数へのポインタ。

See:

- DoMemSaveCertCache
- GetCertCacheMemSize
- CM_MemRestoreCertCache
- CM_GetCertCacheMemSize

Return:

- SSL_SUCCESS 機能の実行に成功したことに戻ります。エラーが投げられていません。
- BAD_MUTEX_E WOLFSSL_CERT_MANAGER 構造体の caLock メンバー 0（ゼロ）ではなかった。
- BAD_FUNC_ARG 引数 ctx、mem が NULL の場合、または sz が 0 以下の場合に返されます。
- BUFFER_E 出力バッファ MEM が小さすぎました。

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
}
```

A.4.2.16 function wolfSSL_CTX_get_cert_cache_memsize

```
int wolfSSL_CTX_get_cert_cache_memsize(
```

```
    WOLFSSL_CTX * ctx
)
```

Certificate Cache Save バッファが必要なサイズを返します。

Parameters:

- **ctx** wolfSSL_CTX_new()で作成された SSL コンテキストへのポインタ。

See: CM_GetCertCacheMemSize

Return:

- メモリサイズを返します。
- BAD_FUNC_ARG WOLFSSL_CTX 構造体が NULL の場合に返されます。
- BAD_MUTEX_E ミューテックスロックエラーが発生した場合に返されます。

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.
}
```

A.4.2.17 function wolfSSL_X509_NAME_oneline

```
char * wolfSSL_X509_NAME_oneline(
    WOLFSSL_X509_NAME * name,
    char * in,
    int sz
)
```

この関数は X509 の名前をバッファにコピーします。

Parameters:

- **name** wolfssl_x509 構造へのポインタ。
- **in** WOLFSSL_X509_NAME 構造体からコピーされた名前を保持するためのバッファ。
- **sz** バッファの最大サイズ

See:

- wolfSSL_X509_get_subject_name
- wolfSSL_X509_get_issuer_name
- wolfSSL_X509_get_isCA
- wolfSSL_get_peer_certificate
- wolfSSL_X509_version

Return: A WOLFSSL_X509_NAME 構造名メンバーのデータが正常に実行された場合、name メンバーのデータが返されます。

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.
}
```

A.4.2.18 function wolfSSL_X509_get_issuer_name

```
WOLFSSL_X509_NAME * wolfSSL_X509_get_issuer_name(
    WOLFSSL_X509 * cert
)
```

この関数は証明書発行者の名前を返します。

Parameters:

- **cert** WOLFSSL_X509 構造体へのポインタ

See:

- wolfSSL_X509_get_subject_name
- wolfSSL_X509_get_isCA
- wolfSSL_get_peer_certificate
- wolfSSL_X509_NAME_oneline

Return:

- point WOLFSSL_X509 構造体の発行者メンバーへのポインタが返されます。
- NULL 渡された証明書が NULL の場合

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.
}
```

A.4.2.19 function wolfSSL_X509_get_subject_name

```
WOLFSSL_X509_NAME * wolfSSL_X509_get_subject_name(
    WOLFSSL_X509 * cert
)
```

この関数は、wolfssl_x509 構造の件名メンバーを返します。

Parameters:

- **cert** WOLFSSL_X509 構造体へのポインタ

See:

- wolfSSL_X509_get_issuer_name
- wolfSSL_X509_get_isCA
- wolfSSL_get_peer_certificate

Return: pointer wolfssl_x509_name 構造へのポインタ。WOLFSSL_X509 構造体が NULL の場合、または構造体の件名メンバーが NULL の場合、ポインタは NULL になることがあります。

Example

```
WOLFSSL_X509* cert;
WOLFSSL_X509_NAME name;
...
name = wolfSSL_X509_get_subject_name(cert);
if(name == NULL){
    // Deal with the NULL cacse
}
```

A.4.2.20 function wolfSSL_X509_get_isCA

```
int wolfSSL_X509_get_isCA(
    WOLFSSL_X509 * cert
)
```

WOLFSSL_X509 構造体の isCa メンバーをチェックして値を返します。

Parameters:

- **cert** WOLFSSL_X509 構造体へのポインタ

See:

- wolfSSL_X509_get_issuer_name
- wolfSSL_X509_get_isCA

Return:

- isCA WOLFSSL_X509 構造体の isCa メンバーの値を返します。
- 0 有効な WOLFSSL_X509 構造体が渡されない場合に返されます。

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
}
```

A.4.2.21 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
)
```

この関数は、渡された NID 値に関連するテキストを取得します。

Parameters:

- **name** wolfssl_x509_name テキストを検索する。
- **nid** 検索する NID。
- **buf** 見つかったときにテキストを保持するためのバッファー。
- **len** バッファのサイズ

See: none

Return: int テキストバッファのサイズを返します。

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
```

A.4.2.22 function wolfSSL_X509_get_signature_type

```
int wolfSSL_X509_get_signature_type(
    WOLFSSL_X509 * cert
)
```

この関数は、WOLFSSL_X509 構造体の sigOID メンバーに格納されている値を返します。

Parameters:

- **cert** WOLFSSL_X509 構造体へのポインタ

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 WOLFSSL_X509 構造体が NULL の場合に返されます。
- int x509 オブジェクトから取得された整数値が返されます。

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
}
```

A.4.2.23 function wolfSSL_X509_get_signature

```
int wolfSSL_X509_get_signature(
    WOLFSSL_X509 * x509,
    unsigned char * buf,
    int * bufSz
)
```

x509 署名を取得し、それをバッファに保存します。

Parameters:

- **x509** wolfssl_x509 構造へのポインタ。
- **buf** バッファへの文字ポインタ。
- **bufSz** バッファサイズを格納する int 型変数へのポインタ

See:

- wolfSSL_X509_get_serial_number
- wolfSSL_X509_get_signature_type
- wolfSSL_X509_get_device_type

Return:

- SSL_SUCCESS 関数が正常に実行された場合に返されます。署名がバッファにロードされます。
- SSL_FATAL_ERRROR X509 構造体または BUFSZ メンバーが NULL の場合に返します。SIG 構造の長さメンバのチェックもある（SIG は X509 のメンバーである）。

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.
}
```

A.4.2.24 function wolfSSL_X509_STORE_add_cert

```
int wolfSSL_X509_STORE_add_cert(
    WOLFSSL_X509_STORE * store,
    WOLFSSL_X509 * x509
)
```

この関数は、WOLFSSL_X509_STRE 構造体に証明書を追加します。

Parameters:

- **str** 証明書を追加する証明書ストア。
- **x509** 追加する WOLFSSL_X509 構造体へのポインタ

See: wolfSSL_X509_free

Return:

- SSL_SUCCESS 証明書が正常に追加された場合。
- SSL_FATAL_ERROR: 証明書が正常に追加されない場合

Example

```
WOLFSSL_X509_STORE* str;
WOLFSSL_X509* x509;
int ret;
ret = wolfSSL_X509_STORE_add_cert(str, x509);
//check ret value
```

A.4.2.25 function wolfSSL_X509_STORE_CTX_get_chain

```
WOLFSSL_STACK * wolfSSL_X509_STORE_CTX_get_chain(
    WOLFSSL_X509_STORE_CTX * ctx
)
```

この関数は、WOLFSSL_X509_STORE_CTX 構造体のチェーン変数の getter 関数です。現在チェーンは取り込まれていません。

Parameters:

- **ctx** WOLFSSL_X509_STORE_CTX 構造体へのポインタ

See: wolfSSL_sk_X509_free

Return:

- pointer 成功した場合 WOLFSSL_STACK（STACK_OF(WOLFSSL_X509)）ポインタと同じ
- Null 失敗した場合に返されます。

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.
```

A.4.2.26 function wolfSSL_X509_STORE_set_flags

```
int wolfSSL_X509_STORE_set_flags(
    WOLFSSL_X509_STORE * store,
    unsigned long flag
)
```

この関数は、渡された WOLFSSL_X509_STORE 構造体の動作を変更するためのフラグを取ります。使用されるフラグの例は WOLFSSL_CRL_CHECK です。

Parameters:

- **str** フラグを設定する証明書ストア。
- **flag** フラグ

See:

- wolfSSL_X509_STORE_new
- wolfSSL_X509_STORE_free

Return:

- SSL_SUCCESS フラグを設定するときにエラーが発生しなかった場合。
- <0 障害の際に負の値が返されます。

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
}
```

A.4.2.27 function wolfSSL_X509_notBefore

```
const byte * wolfSSL_X509_notBefore(
    WOLFSSL_X509 * x509
)
```

この関数は BYTE アレイとして符号化された “not before” 要素を返します。

Parameters:

- **x509** WOLFSSL_X509 構造体へのポインタ。

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 WOLFSSL_X509 構造体が NULL の場合に返されます。
- byte NetBeforEdata を含むバッファへのポインタが返されます。

Example

```
WOLFSSL_X509* x509 = (WOLFSSL_X509*)XMALLOC(sizeof(WOLFSSL_X509), NULL,
                        DYNAMIC_TYPE_X509);
...
byte* notBeforeData = wolfSSL_X509_notBefore(x509);
```

A.4.2.28 function wolfSSL_X509_notAfter

```
const byte * wolfSSL_X509_notAfter(
    WOLFSSL_X509 * x509
)
```

この関数は、BYTE 配列として符号化された “not after” 要素を返します。

Parameters:

- **x509** WOLFSSL_X509 構造体へのポインタ。

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 WOLFSSL_X509 構造体が NULL の場合に返されます。
- byte notAfterData を含むバッファへのポインタが返されます。

Example

```
WOLFSSL_X509* x509 = (WOLFSSL_X509*)XMALLOC(sizeof(WOLFSSL_X509), NULL,
                        DYNAMIC_TYPE_X509);
```

```
...
byte* notAfterData = wolfSSL_X509_notAfter(x509);
```

A.4.2.29 function wolfSSL_get_psk_identity_hint

```
const char * wolfSSL_get_psk_identity_hint(
    const WOLFSSL *
)
```

この関数は PSK アイデンティティヒントを返します。

See: wolfSSL_get_psk_identity

Return:

- pointer WolfSSL 構造の配列メンバーに格納されている値への const char ポインタが返されます。
- NULL WOLFSSL または配列構造が NULL の場合に返されます。

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
}
```

A.4.2.30 function wolfSSL_get_psk_identity

```
const char * wolfSSL_get_psk_identity(
    const WOLFSSL *
)
```

関数は、配列構造の Client_Identity メンバーへの定数ポインタを返します。

See:

- wolfSSL_get_psk_identity_hint
- wolfSSL_use_psk_identity_hint

Return:

- string 配列構造の client_identity メンバの文字列値。
- NULL WOLFSSL 構造が NULL の場合、または WOLFSSL 構造の配列メンバーが NULL の場合。

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
}
```

A.4.2.31 function wolfSSL_CTX_use_psk_identity_hint

```
int wolfSSL_CTX_use_psk_identity_hint(
    WOLFSSL_CTX * ctx,
    const char * hint
)
```

この関数は、WOLFSSL_CTX 構造体の server_hint メンバーに HINT 引数を格納します。

Parameters:

- **ctx** wolfSSL_CTX_new()を使用して作成された WOLFSSL_CTX 構造体へのポインタ。

See: wolfSSL_use_psk_identity_hint

Return: SSL_SUCCESS 機能の実行が成功したために返されます。

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.
}
```

A.4.2.32 function wolfSSL_use_psk_identity_hint

```
int wolfSSL_use_psk_identity_hint(
    WOLFSSL * ssl,
    const char * hint
)
```

この関数は、wolfssl 構造内の配列構造の server_hint メンバーに HINT 引数を格納します。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WolfSSL 構造体へのポインタ

See: wolfSSL_CTX_use_psk_identity_hint

Return:

- SSL_SUCCESS ヒントが WolfSSL 構造に正常に保存された場合に返されます。
- SSL_FAILURE WOLFSSL または配列構造が NULL の場合に返されます。

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
const char* hint;
...
if(wolfSSL_use_psk_identity_hint(ssl, hint) != SSL_SUCCESS){
    // Handle failure case.
}
```

A.4.2.33 function wolfSSL_get_peer_certificate

```
WOLFSSL_X509 * wolfSSL_get_peer_certificate(
    WOLFSSL * ssl
)
```

この関数はピアの証明書を取得します。

See:

- wolfSSL_X509_get_issuer_name
- wolfSSL_X509_get_subject_name
- wolfSSL_X509_get_isCA

Return:

- pointer WOLFSSL_X509 構造の PECRERT メンバーへのポインタが存在する場合は。
- 0 ピア証明書発行者サイズが定義されていない場合に返されます。

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
}
```

A.4.2.34 function wolfSSL_get_chain_X509

```
WOLFSSL_X509 * wolfSSL_get_chain_X509(
    WOLFSSL_X509_CHAIN * chain,
    int idx
)
```

この関数は、証明書のチェーンからのピアの WOLFSSL_X509 構造体をインデックス（IDX）で取得します。

Parameters:

- **chain** 動的メモリ session_cache の場合に使用される WOLFSSL_X509_CHAIN へのポインタ。

See:

- InitDecodedCert
- ParseCertRelative
- CopyDecodedToX509

Return: pointer WOLFSSL_X509 構造体へのポインタを返します。

注意：本関数から返された構造体を wolfSSL_FreeX509() を呼び出して解放するのはユーザーの責任です。

Example

```
WOLFSSL_X509_CHAIN* chain = &session->chain;
int idx = 999; // set idx
...
WOLFSSL_X509* 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
}
```

A.4.2.35 function wolfSSL_X509_get_subjectCN

```
char * wolfSSL_X509_get_subjectCN(
    WOLFSSL_X509 *
)
```

証明書から件名の共通名を返します。

See:

- wolfSSL_X509_Name_get_entry
- wolfSSL_X509_get_next_altname
- wolfSSL_X509_get_issuer_name
- wolfSSL_X509_get_subject_name

Return:

- NULL X509 構造が NULL の場合に返されます
- string サブジェクトの共通名の文字列表現は成功に返されます

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
}
```

A.4.2.36 function wolfSSL_X509_get_der

```
const unsigned char * wolfSSL_X509_get_der(
    WOLFSSL_X509 * x509,
    int * outSz
)
```

この関数は、wolfssl_x509 構造体の DER エンコードされた証明書を取得します。

Parameters:

- **x509** 証明書情報を含む WolfSSL_X509 構造へのポインタ。

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 この関数は Derbuffer 構造体のバッファメンバーを返します。これはバイト型です。

- NULL x509 または outsz パラメーターが null の場合に返されます。

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
}
```

A.4.2.37 function wolfSSL_X509_get_notAfter

```
WOLFSSL_ASN1_TIME * wolfSSL_X509_get_notAfter(
    WOLFSSL_X509 *
)
```

この関数は、x509 が null のかどうかを確認し、そうでない場合は、x509 構造体のノッカスメンバーを返します。

See: wolfSSL_X509_get_notBefore

Return:

- pointer ASN1_TIME を使用して X509 構造体のノカフターメンバーに構造体を表明します。
- NULL X509 オブジェクトが NULL の場合に返されます。

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.
}
```

A.4.2.38 function wolfSSL_X509_version

```
int wolfSSL_X509_version(
    WOLFSSL_X509 *
)
```

この関数は X509 証明書のバージョンを取得します。

See:

- wolfSSL_X509_get_subject_name
- wolfSSL_X509_get_issuer_name
- wolfSSL_X509_get_isCA
- wolfSSL_get_peer_certificate

Return:

- 0 X509 構造が NULL の場合に返されます。
- version X509 構造に保存されているバージョンが返されます。

Example

```
WOLFSSL_X509* x509;
int version;
...
version = wolfSSL_X509_version(x509);
if(!version){
    // The function returned 0, failure case.
}
```

A.4.2.39 function wolfSSL_X509_d2i_fp

```
WOLFSSL_X509 * wolfSSL_X509_d2i_fp(
    WOLFSSL_X509 ** x509,
    FILE * file
)
```

no_stdio_filesystem が定義されている場合、この関数はヒープメモリを割り当て、wolfssl_x509 構造を初期化してそれにポインタを返します。

Parameters:

- **x509** wolfssl_x509 ポインタへのポインタ。

See:

- wolfSSL_X509_d2i
- XFTELL
- XREWIND
- XFSEEK

Return:

- *WOLFSSL_X509 関数が正常に実行された場合、WolfSSL_X509 構造ポインタが返されます。
- NULL Xftell マクロの呼び出しが負の値を返す場合。

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
}
```

A.4.2.40 function wolfSSL_X509_load_certificate_file

```
WOLFSSL_X509 * wolfSSL_X509_load_certificate_file(
    const char * fname,
    int format
)
```

関数は X509 証明書をメモリにロードします。

Parameters:

- **fname** ロードする証明書ファイル。

See:

- InitDecodedCert
- PemToDer
- wolfSSL_get_certificate
- AssertNotNull

Return:

- pointer 実行された実行は、wolfssl_x509 構造へのポインタを返します。
- NULL 証明書が書き込まれなかった場合に返されます。

Example

```
#define cliCert    “certs/client-cert.pem”
…
X509* x509;
…
x509 = wolfSSL_X509_load_certificate_file(cliCert, SSL_FILETYPE_PEM);
AssertNotNull(x509);
```

A.4.2.41 function wolfSSL_X509_get_device_type

```
unsigned char * wolfSSL_X509_get_device_type(
    WOLFSSL_X509 * x509,
    unsigned char * in,
    int * inOutSz
)
```

この関数は、デバイスの種類を X509 構造からバッファにコピーします。

Parameters:

- **x509** wolfssl_x509_new() で作成された wolfssl_x509 構造へのポインタ。
- **in** デバイスタイプ（バッファ）を保持するバイトタイプへのポインタ。

See:

- wolfSSL_X509_get_hw_type
- wolfSSL_X509_get_hw_serial_number
- wolfSSL_X509_d2i

Return:

- pointer X509 構造からデバイスの種類を保持するバイトポインタを返します。
- NULL バッファサイズが NULL の場合に返されます。

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.
}
```

A.4.2.42 function wolfSSL_X509_get_hw_type

```
unsigned char * wolfSSL_X509_get_hw_type(
    WOLFSSL_X509 * x509,
    unsigned char * in,
    int * inOutSz
)
```

この関数は、wolfssl_x509 構造の HWType メンバーをバッファにコピーします。

Parameters:

- **x509** 証明書情報を含む WolfSSL_X509 構造へのポインタ。
- **in** バッファを表すバイトを入力するポインタ。

See:

- wolfSSL_X509_get_hw_serial_number
- wolfSSL_X509_get_device_type

Return:

- byte この関数は、wolfssl_x509 構造の HWType メンバーに以前に保持されているデータのバイトタイプを返します。
- NULL inoutsz が null の場合に返されます。

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.
}
```

A.4.2.43 function wolfSSL_X509_get_hw_serial_number

```
unsigned char * wolfSSL_X509_get_hw_serial_number(
    WOLFSSL_X509 * x509,
    unsigned char * in,
    int * inOutSz
)
```

この関数は X509 オブジェクトの hwserialNum メンバを返します。

Parameters:

- **x509** 証明書情報を含む WOLFSSL_X509 構造へのポインタ。
- **in** コピーされるバッファへのポインタ。

See:

- wolfSSL_X509_get_subject_name
- wolfSSL_X509_get_issuer_name
- wolfSSL_X509_get_isCA
- wolfSSL_get_peer_certificate
- wolfSSL_X509_version

Return: pointer この関数は、X509 オブジェクトからロードされたシリアル番号を含む IN バッファへのバイトポインタを返します。

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
}
```

A.4.2.44 function wolfSSL_SetTmpDH

```
int wolfSSL_SetTmpDH(
    WOLFSSL * ssl,
    const unsigned char * p,
    int pSz,
    const unsigned char * g,
    int gSz
)
```

サーバー DIFFIE-HELLMAN エフェメラルパラメータ設定。この関数は、サーバーが DHE を使用する暗号スイートをネゴシエートしている場合に使用するグループパラメータを設定します。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WolfSSL 構造へのポインタ。
- **p** Diffie-Hellman 素数パラメータ。
- **pSz** p のサイズ。
- **g** Diffie-Hellman “Generator” パラメータ。

See: SSL_accept

Return:

- SSL_SUCCESS 成功時に返されます。
- MEMORY_ERROR メモリエラーが発生した場合に返されます。
- SIDE_ERROR この関数が SSL サーバではなく SSL クライアントで呼び出されると返されます。

Example

```
WOLFSSL* ssl;
static unsigned char p[] = {...};
static unsigned char g[] = {...};
...
wolfSSL_SetTmpDH(ssl, p, sizeof(p), g, sizeof(g));
```

A.4.2.45 function wolfSSL_SetTmpDH_buffer

```
int wolfSSL_SetTmpDH_buffer(
    WOLFSSL * ssl,
    const unsigned char * b,
    long sz,
    int format
)
```

関数は wolfssl_settmph_buffer_wrapper を呼び出します。これは Diffie-Hellman パラメータのラッパーです。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WolfSSL 構造へのポインタ。
- **buf** wolfssl_settmph_file_wrapper から渡された割り当てバッファー。
- **sz** ファイルのサイズ（wolfssl_settmph_file_wrapper 内の fname）を保持するロング int。

See:

- wolfSSL_SetTmpDH_buffer_wrapper
- wc_DhParamsLoad
- wolfSSL_SetTmpDH
- PemToDer
- wolfSSL_CTX_SetTmpDH
- wolfSSL_CTX_SetTmpDH_file

Return:

- SSL_SUCCESS 実行に成功した場合。
- SSL_BAD_FILETYPE ファイルの種類が pem ではなく、asn.1 ではない場合 WC_DHParamSLOAD が正常に戻っていない場合は、も返されます。
- SSL_NO_PEM_HEADER PEM ヘッダーがない場合は PemToder から返します。
- SSL_BAD_FILE PemToder にファイルエラーがある場合に返されます。
- SSL_FATAL_ERROR コピーエラーが発生した場合は PemToder から返されました。
- MEMORY_E - メモリ割り当てエラーが発生した場合
- BAD_FUNC_ARG wolfssl 構造体が null の場合、またはそうでない場合はサブルーチンに渡された場合に返されます。
- DH_KEY_SIZE_E wolfssl_settmph() または WOLFSSL_CTX_settmph() の鍵サイズエラーがある場合に返されます。
- SIDE_ERROR wolfssl_settmph のサーバー側ではない場合に返されます。

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);
```

A.4.2.46 function wolfSSL_SetTmpDH_file

```
int wolfSSL_SetTmpDH_file(
    WOLFSSL * ssl,
    const char * f,
    int format
)
```

この関数は、wolfssl_settmph_file_wrapper を呼び出してサーバ diffie-hellman パラメータを設定します。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ
- **fname** 証明書を保持している定数の文字ポインタ。

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 この機能の正常な完了とそのサブルーチンの完了に戻りました。
- MEMORY_E この関数またはサブルーチンにメモリ割り当てが失敗した場合に返されます。
- SIDE_ERROR WolfSSL 構造体にあるオプション構造のサイドメンバーがサーバー側ではない場合。
- SSL_BAD_FILETYPE 証明書が一連のチェックに失敗した場合は返します。
- DH_KEY_SIZE_E DH パラメーターの鍵サイズが WolfSSL 構造体の MinkKeysz メンバーの値より小さい場合に返されます。
- DH_KEY_SIZE_E DH パラメータの鍵サイズが wolfssl 構造体の MAXDHKEYSZ メンバーの値よりも大きい場合に返されます。
- BAD_FUNC_ARG wolfssl 構造など、引数値が null の場合に返します。

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
const char* dhParam;
…
AssertIntNE(SSL_SUCCESS,
wolfSSL_SetTmpDH_file(ssl, dhParam, SSL_FILETYPE_PEM));
```

A.4.2.47 function wolfSSL_CTX_SetTmpDH

```
int wolfSSL_CTX_SetTmpDH(
    WOLFSSL_CTX * ctx,
    const unsigned char * p,
    int pSz,
    const unsigned char * g,
    int gSz
)
```

サーバー CTX Diffie-Hellman のパラメータを設定します。

Parameters:

- **ctx** wolfSSL_CTX_new()を使用して作成された WOLFSSL_CTX 構造体へのポインタ。
- **p** ServerDH_P 構造体のバッファメンバーにロードされた定数の符号なし文字ポインタ。
- **pSz** p のサイズを表す int 型は、max_dh_size に初期化されます。
- **g** ServerDh_g 構造体のバッファメンバーにロードされた定数の符号なし文字ポインタ。

See:

- wolfSSL_SetTmpDH
- wc_DhParamsLoad

Return:

- SSL_SUCCESS 関数とすべてのサブルーチンがエラーなしで戻った場合に返されます。
- BAD_FUNC_ARG CTX、P、または G パラメーターが NULL の場合に返されます。
- DH_KEY_SIZE_E DH パラメータの鍵サイズが WOLFSSL_CTX 構造体の MindHKEYSZ メンバーの値より小さい場合に返されます。
- DH_KEY_SIZE_E DH パラメータの鍵サイズが WOLFSSL_CTX 構造体の MaxDhkeySZ メンバーの値よりも大きい場合に返されます。
- MEMORY_E この関数またはサブルーチンにメモリの割り当てが失敗した場合に返されます。

Example

```
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
}
```

A.4.2.48 function wolfSSL_CTX_SetTmpDH_buffer

```
int wolfSSL_CTX_SetTmpDH_buffer(
    WOLFSSL_CTX * ctx,
    const unsigned char * b,
    long sz,
    int format
)
```

wolfssl_settmph_buffer_wrapper を呼び出すラッパー関数

Parameters:

- **ctx** wolfSSL_CTX_new()を使用して作成された WolfSSL 構造へのポインタ。
- **buf** バッファとして割り当てられ、wolfssl_settmpdh_buffer_wrapper に渡された定数の符号なし文字型へのポインタ。
- **sz** wolfssl_settmph_file_wrapper() の FNAME パラメータから派生した長い整数型。

See:

- wolfSSL_SetTmpDH_buffer_wrapper
- wolfSSL_SetTMpDH_buffer
- wolfSSL_SetTmpDH_file_wrapper
- wolfSSL_CTX_SetTmpDH_file

Return:

- 0 実行が成功するために返されました。
- BAD_FUNC_ARG CTX パラメータまたは BUF パラメータが NULL の場合に返されます。
- MEMORY_E メモリ割り当てエラーがある場合
- SSL_BAD_FILETYPE フォーマットが正しくない場合に返されます。

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);
}
```

A.4.2.49 function wolfSSL_CTX_SetTmpDH_file

```
int wolfSSL_CTX_SetTmpDH_file(
    WOLFSSL_CTX * ctx,
    const char * f,
    int format
)
```

この関数は、wolfssl_settmph_file_wrapper を呼び出してサーバー Diffie-Hellman パラメータを設定します。

Parameters:

- **ctx** wolfSSL_CTX_new()を使用して作成された WOLFSSL_CTX 構造体へのポインタ。
- **fname** 証明書ファイルへの定数文字ポインタ。

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 wolfssl_settmph_file_wrapper またはそのサブルーチンのいずれかが正常に戻った場合に返されます。
- MEMORY_E 動的メモリの割り当てがサブルーチンで失敗した場合に返されます。
- BAD_FUNC_ARG CTX または FNAME パラメータが NULL またはサブルーチンが NULL 引数に渡された場合に返されます。
- SSL_BAD_FILE 証明書ファイルが開くことができない場合、またはファイルの一連のチェックが wolfssl_settmpdh_file_wrapper から失敗した場合に返されます。
- SSL_BAD_FILETYPE フォーマットが wolfssl_settmph_buffer_wrapper() から PEM または ASN.1 ではない場合に返されます。
- DH_KEY_SIZE_E DH パラメータの鍵サイズが WOLFSSL_CTX 構造体の MindHKEYSZ メンバーの値より小さい場合に返されます。
- DH_KEY_SIZE_E DH パラメータの鍵サイズが WOLFSSL_CTX 構造体の MaxDhkeySZ メンバーの値よりも大きい場合に返されます。
- SIDE_ERROR wolfssl_settmph() で返されたサイドがサーバー終了ではない場合。
- SSL_NO_PEM_HEADER PEM ヘッダーがない場合は PemToder から返されます。
- SSL_FATAL_ERROR メモリコピーの失敗がある場合は PemToder から返されます。

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));
```

A.4.2.50 function wolfSSL_CTX_SetMinDhKey_Sz

```
int wolfSSL_CTX_SetMinDhKey_Sz(
    WOLFSSL_CTX * ctx,
    word16
)
```

この関数は、WOLFSSL_CTX 構造体の minkkeysz メンバーにアクセスして、Diffie Hellman 鍵サイズの最小サイズ（ビット単位）を設定します。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WolfSSL 構造へのポインタ。

See:

- wolfSSL_SetMinDhKey_Sz
- wolfSSL_CTX_SetMaxDhKey_Sz
- wolfSSL_SetMaxDhKey_Sz
- wolfSSL_GetDhKey_Sz
- wolfSSL_CTX_SetTMpDH_file

Return:

- SSL_SUCCESS 関数が正常に完了した場合に返されます。
- BAD_FUNC_ARG WOLFSSL_CTX 構造体が null の場合、またはキー z_BITS が 16,000 を超えるか、または 8 によって割り切れない場合に返されます。

Example

```
public static int CTX_SetMinDhKey_Sz(IntPtr ctx, short minDhKey){
…
return wolfSSL_CTX_SetMinDhKey_Sz(local_ctx, minDhKeyBits);
```

A.4.2.51 function wolfSSL_SetMinDhKey_Sz

```
int wolfSSL_SetMinDhKey_Sz(
    WOLFSSL * ssl,
    word16 keySz_bits
)
```

WolfSSL 構造の Diffie-Hellman 鍵の最小サイズ（ビット単位）を設定します。

Parameters:

- **ssl** wolfssl_new() を使用して作成された WolfSSL 構造へのポインタ。

See:

- wolfSSL_CTX_SetMinDhKey_Sz
- wolfSSL_GetDhKey_Sz

Return:

- SSL_SUCCESS 最小サイズは正常に設定されました。
- BAD_FUNC_ARG wolfssl 構造は NULL、または Keysz_BITS が 16,000 を超えるか、または 8 によって割り切れない場合

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
word16 keySz_bits;
...
if(wolfSSL_SetMinDhKey_Sz(ssl, keySz_bits) != SSL_SUCCESS){
```

```
    // Failed to set.
}
```

A.4.2.52 function wolfSSL_CTX_SetMaxDhKey_Sz

```
int wolfSSL_CTX_SetMaxDhKey_Sz(
    WOLFSSL_CTX * ctx,
    word16 keySz_bits
)
```

この関数は、WOLFSSL_CTX 構造体の maxdhkeysz メンバーにアクセスして、Diffie Hellman 鍵サイズの最大サイズ（ビット単位）を設定します。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ

See:

- wolfSSL_SetMinDhKey_Sz
- wolfSSL_CTX_SetMinDhKey_Sz
- wolfSSL_SetMaxDhKey_Sz
- wolfSSL_GetDhKey_Sz
- wolfSSL_CTX_SetTMpDH_file

Return:

- SSL_SUCCESS 関数が正常に完了した場合に返されます。
- BAD_FUNC_ARG WOLFSSL_CTX 構造体が null の場合、またはキー z_BITS が 16,000 を超えるか、または 8 によって割り切れない場合に返されます。

Example

```
public static int CTX_SetMaxDhKey_Sz(IntPtr ctx, short maxDhKey){
…
return wolfSSL_CTX_SetMaxDhKey_Sz(local_ctx, keySz_bits);
```

A.4.2.53 function wolfSSL_SetMaxDhKey_Sz

```
int wolfSSL_SetMaxDhKey_Sz(
    WOLFSSL * ssl,
    word16 keySz_bits
)
```

WolfSSL 構造の Diffie-Hellman 鍵の最大サイズ（ビット単位）を設定します。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ

See:

- wolfSSL_CTX_SetMaxDhKey_Sz
- wolfSSL_GetDhKey_Sz

Return:

- SSL_SUCCESS 最大サイズは正常に設定されました。
- BAD_FUNC_ARG WOLFSSL 構造は NULL または KEYSZ パラメータは許容サイズより大きかったか、または 8 によって割り切れませんでした。

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
word16 keySz;
...
if(wolfSSL_SetMaxDhKey(ssl, keySz) != SSL_SUCCESS){
    // Failed to set.
}
```

A.4.2.54 function wolfSSL_GetDhKey_Sz

```
int wolfSSL_GetDhKey_Sz(
    WOLFSSL *
)
```

オプション構造のメンバーである DHKEYSZ（ビット内）の値を返します。この値は、Diffie-Hellman 鍵サイズをバイト単位で表します。

See:

- wolfSSL_SetMinDhKey_sz
- wolfSSL_CTX_SetMinDhKey_Sz
- wolfSSL_CTX_SetTmpDH
- wolfSSL_SetTmpDH
- wolfSSL_CTX_SetTmpDH_file

Return:

- dhKeySz サイズを表す整数値である ssl-> options.dhkeysz で保持されている値を返します。
- BAD_FUNC_ARG wolfssl 構造体が NULL の場合に返します。

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.
}
```

A.4.2.55 function wolfSSL_CTX_SetMinRsaKey_Sz

```
int wolfSSL_CTX_SetMinRsaKey_Sz(
    WOLFSSL_CTX * ctx,
    short keySz
)
```

WOLFSSL_CTX 構造体と wolfssl_cert_manager 構造の両方で最小 RSA 鍵サイズを設定します。

Parameters:

- **ctx** wolfSSL_CTX_new()を使用して作成された WOLFSSL_CTX 構造体へのポインタ。

See: wolfSSL_SetMinRsaKey_Sz

Return:

- SSL_SUCCESS 機能の実行に成功したことに戻ります。

- BAD_FUNC_ARG CTX 構造が NULL の場合、または KEYSZ がゼロより小さいか、または 8 によって割り切れない場合に返されます。

Example

```
WOLFSSL_CTX* ctx = SSL_CTX_new(method);
(void)minDhKeyBits;
ourCert = myoptarg;
…
minDhKeyBits = atoi(myoptarg);
…
if(wolfSSL_CTX_SetMinRsaKey_Sz(ctx, minRsaKeyBits) != SSL_SUCCESS){
…
```

A.4.2.56 function wolfSSL_SetMinRsaKey_Sz

```
int wolfSSL_SetMinRsaKey_Sz(
    WOLFSSL * ssl,
    short keySz
)
```

WolfSSL 構造にある RSA のためのビットで最小許容鍵サイズを設定します。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ

See: wolfSSL_CTX_SetMinRsaKey_Sz

Return:

- SSL_SUCCESS 最小値が正常に設定されました。
- BAD_FUNC_ARG SSL 構造が NULL の場合、または KSYSZ がゼロより小さい場合、または 8 によって割り切れない場合に返されます。

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
short keySz;
…

int isSet =  wolfSSL_SetMinRsaKey_Sz(ssl, keySz);
if(isSet != SSL_SUCCESS){
    Failed to set.
}
```

A.4.2.57 function wolfSSL_CTX_SetMinEccKey_Sz

```
int wolfSSL_CTX_SetMinEccKey_Sz(
    WOLFSSL_CTX * ssl,
    short keySz
)
```

wolf_ctx 構造体と wolfssl_cert_manager 構造体の ECC 鍵の最小サイズをビット単位で設定します。

Parameters:

- **ctx** wolfSSL_CTX_new()を使用して作成された WOLFSSL_CTX 構造体へのポインタ。

See: wolfSSL_SetMinEccKey_Sz

Return:

- SSL_SUCCESS 実行が成功したために返され、MineCkeysz メンバーが設定されます。
- BAD_FUNC_ARG WOLFSSL_CTX 構造体が null の場合、または鍵が負の場合、または 8 によって割り切れない場合に返されます。

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
}
```

A.4.2.58 function wolfSSL_SetMinEccKey_Sz

```
int wolfSSL_SetMinEccKey_Sz(
    WOLFSSL * ssl,
    short keySz
)
```

オプション構造の MineCckeysz メンバーの値を設定します。オプション構造体は、WolfSSL 構造のメンバーであり、SSL パラメータを介してアクセスされます。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ

See:

- wolfSSL_CTX_SetMinEccKey_Sz
- wolfSSL_CTX_SetMinRsaKey_Sz
- wolfSSL_SetMinRsaKey_Sz

Return:

- SSL_SUCCESS 関数がオプション構造の MineCckeysz メンバーを正常に設定した場合。
- BAD_FUNC_ARG WOLFSSL_CTX 構造体が null の場合、または鍵サイズ（keysz）が 0（ゼロ）未満の場合、または 8 で割り切れない場合。

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.
}
```

A.4.2.59 function wolfSSL_make_eap_keys

```
int wolfSSL_make_eap_keys(
    WOLFSSL * ssl,
    void * key,
    unsigned int len,
    const char * label
)
```

この関数は、eap_tls と eap-ttls によって、マスターシークレットからキーイングマテリアルを導出します。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ
- **msk** p_hash 関数の結果を保持する void ポインタ変数。
- **len** MSK 変数の長さを表す符号なし整数。

See:

- wc_PRF
- wc_HmacFinal
- wc_HmacUpdate

Return:

- BUFFER_E バッファの実際のサイズが許容最大サイズを超える場合に返されます。
- MEMORY_E メモリ割り当てにエラーがある場合に返されます。

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);;
void* msk;
unsigned int len;
const char* label;
…
return wolfSSL_make_eap_keys(ssl, msk, len, label);
```

A.4.2.60 function wolfSSL_CTX_load_verify_buffer

```
int wolfSSL_CTX_load_verify_buffer(
    WOLFSSL_CTX * ctx,
    const unsigned char * in,
    long sz,
    int format
)
```

この関数は CA 証明書バッファを WolfSSL コンテキストにロードします。バッファ以外のバージョンのように動作し、ファイルの代わりに入力としてバッファと呼ばれる機能が異なるだけです。バッファはサイズ SZ の引数によって提供されます。形式バッファのフォーマットタイプを指定します。SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM。フォーマットが PEM 内にある限り、バッファあたり複数の CA 証明書をロードすることができます。適切な使用法の例をご覧ください。

Parameters:

- **ctx** wolfSSL_CTX_new()で作成された SSL コンテキストへのポインタ。
- **in** CA 証明書バッファへのポインタ。
- **sz** 入力 CA 証明書バッファのサイズ、IN。

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 成功すると
- SSL_BAD_FILETYPE ファイルが間違った形式である場合に返されます。
- SSL_BAD_FILE ファイルが存在しない場合に返されます。読み込め、または破損していません。
- MEMORY_E メモリ不足状態が発生した場合に返されます。

- ASN_INPUT_E base16 デコードがファイルに対して失敗した場合に返されます。
- BUFFER_E チェーンバッファが受信バッファよりも大きい場合に返されます。

Example

```
int ret = 0;
int sz = 0;
WOLFSSL_CTX* ctx;
byte certBuff[...];
...

ret = wolfSSL_CTX_load_verify_buffer(ctx, certBuff, sz, SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
    // error loading CA certs from buffer
}
...
```

A.4.2.61 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
)
```

この関数は CA 証明書バッファを WolfSSL コンテキストにロードします。バッファ以外のバージョンのように動作し、ファイルの代わりに入力としてバッファと呼ばれる機能が異なるだけです。バッファはサイズ SZ の引数によって提供されます。形式バッファのフォーマットタイプを指定します。SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM。フォーマットが PEM 内にある限り、バッファあたり複数の CA 証明書をロードすることができます。_EX バージョンは PR 2413 に追加され、UserChain と Flags の追加の引数をサポートします。

Parameters:

- **ctx** wolfSSL_CTX_new()で作成された SSL コンテキストへのポインタ。
- **in** CA 証明書バッファへのポインタ。
- **sz** 入力 CA 証明書バッファのサイズ、IN。
- **format** バッファ証明書の形式、SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM。
- **userChain** フォーマット wolfssl_filetype_asn1 を使用する場合、このセットはゼロ以外のセットを示しています.Der のチェーンが表示されています。

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 成功すると
- SSL_BAD_FILETYPE ファイルが間違った形式である場合に返されます。

- SSL_BAD_FILE ファイルが存在しない場合に返されます。読み込め、または破損していません。
- MEMORY_E メモリ不足状態が発生した場合に返されます。
- ASN_INPUT_E base16 デコードがファイルに対して失敗した場合に返されます。
- BUFFER_E チェーンバッファが受信バッファよりも大きい場合に返されます。

Example

```
int ret = 0;
int sz = 0;
WOLFSSL_CTX* ctx;
byte 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
}
...
```

A.4.2.62 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
)
```

この関数は、CA 証明書チェーンバッファを WolfSSL コンテキストにロードします。バッファ以外のバージョンのように動作し、ファイルの代わりに入力としてバッファと呼ばれる機能が異なるだけです。バッファはサイズ SZ の引数によって提供されます。形式バッファのフォーマットタイプを指定します。SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM。フォーマットが PEM 内にある限り、バッファあたり複数の CA 証明書をロードすることができます。適切な使用法の例をご覧ください。

Parameters:

- **ctx** wolfSSL_CTX_new()で作成された SSL コンテキストへのポインタ。
- **in** CA 証明書バッファへのポインタ。
- **sz** 入力 CA 証明書バッファのサイズ、IN。

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 成功すると
- SSL_BAD_FILETYPE ファイルが間違った形式である場合に返されます。
- SSL_BAD_FILE ファイルが存在しない場合に返されます。読み込め、または破損していません。
- MEMORY_E メモリ不足状態が発生した場合に返されます。
- ASN_INPUT_E base16 デコードがファイルに対して失敗した場合に返されます。

- BUFFER_E チェーンバッファが受信バッファよりも大きい場合に返されます。

Example

```
int ret = 0;
int sz = 0;
WOLFSSL_CTX* ctx;
byte 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
}
...
```

A.4.2.63 function wolfSSL_CTX_use_certificate_buffer

```
int wolfSSL_CTX_use_certificate_buffer(
    WOLFSSL_CTX * ctx,
    const unsigned char * in,
    long sz,
    int format
)
```

この関数は証明書バッファを WolfSSL コンテキストにロードします。バッファ以外のバージョンのように動作し、ファイルの代わりに入力としてバッファと呼ばれる機能が異なるだけです。バッファはサイズ SZ の引数によって提供されます。形式バッファのフォーマットタイプを指定します。SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM。適切な使用法の例をご覧ください。

Parameters:

- **ctx** wolfSSL_CTX_new()で作成された SSL コンテキストへのポインタ。
- **in** ロードする証明書を含む入力バッファ。
- **sz** 入力バッファのサイズ。

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 成功すると
- SSL_BAD_FILETYPE ファイルが間違った形式である場合に返されます。
- SSL_BAD_FILE ファイルが存在しない場合に返されます。読み込め、または破損していません。
- MEMORY_E メモリ不足状態が発生した場合に返されます。
- ASN_INPUT_E base16 デコードがファイルに対して失敗した場合に返されます。

Example

```
int ret = 0;
int sz = 0;
WOLFSSL_CTX* ctx;
byte certBuff[...];
```

```
...
ret = wolfSSL_CTX_use_certificate_buffer(ctx, certBuff, sz, SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
    // error loading certificate from buffer
}
...
```

A.4.2.64 function wolfSSL_CTX_use_PrivateKey_buffer

```
int wolfSSL_CTX_use_PrivateKey_buffer(
    WOLFSSL_CTX * ctx,
    const unsigned char * in,
    long sz,
    int format
)
```

この関数は、秘密鍵バッファを SSL コンテキストにロードします。バッファ以外のバージョンのように動作し、ファイルの代わりに入力としてバッファと呼ばれる機能が異なるだけです。バッファはサイズ SZ の引数によって提供されます。形式バッファのフォーマットタイプを指定します。SSL_FILETYPE_ASN1OR SSL_FILETYPE_PEM。適切な使用法の例をご覧ください。

Parameters:

- **ctx** wolfSSL_CTX_new()で作成された SSL コンテキストへのポインタ。
- **in** ロードする秘密鍵を含む入力バッファ。
- **sz** 入力バッファのサイズ。

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 成功すると
- SSL_BAD_FILETYPE ファイルが間違った形式である場合に返されます。
- SSL_BAD_FILE ファイルが存在しない場合に返されます。読み込め、または破損していません。
- MEMORY_E メモリ不足状態が発生した場合に返されます。
- ASN_INPUT_E base16 デコードがファイルに対して失敗した場合に返されます。
- NO_PASSWORD 鍵ファイルが暗号化されているがパスワードが提供されていない場合に返されます。

Example

```
int ret = 0;
int sz = 0;
WOLFSSL_CTX* ctx;
byte keyBuff[...];
...
ret = wolfSSL_CTX_use_PrivateKey_buffer(ctx, keyBuff, sz, SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
    // error loading private key from buffer
}
...
```

A.4.2.65 function wolfSSL_CTX_use_certificate_chain_buffer

```
int wolfSSL_CTX_use_certificate_chain_buffer(
    WOLFSSL_CTX * ctx,
    const unsigned char * in,
    long sz
)
```

この関数は、証明書チェーンバッファを WolfSSL コンテキストにロードします。バッファ以外のバージョンのように動作し、ファイルの代わりに入力としてバッファと呼ばれる機能が異なるだけです。バッファはサイズ SZ の引数によって提供されます。バッファは PEM 形式で、ルート証明書で終わる対象の証明書から始めてください。適切な使用法の例をご覧ください。

Parameters:

- **ctx** wolfSSL_CTX_new()で作成された SSL コンテキストへのポインタ。
- **in** ロードされる PEM 形式の証明書チェーンを含む入力バッファ。

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 成功すると
- SSL_BAD_FILETYPE ファイルが間違った形式である場合に返されます。
- SSL_BAD_FILE ファイルが存在しない場合に返されます。読み込め、または破損していません。
- MEMORY_E メモリ不足状態が発生した場合に返されます。
- ASN_INPUT_E base16 デコードがファイルに対して失敗した場合に返されます。
- BUFFER_E チェーンバッファが受信バッファよりも大きい場合に返されます。

Example

```
int ret = 0;
int sz = 0;
WOLFSSL_CTX* ctx;
byte certChainBuff[...];
...
ret = wolfSSL_CTX_use_certificate_chain_buffer(ctx, certChainBuff, sz);
if (ret != SSL_SUCCESS) {
    // error loading certificate chain from buffer
}
...
```

A.4.2.66 function wolfSSL_use_certificate_buffer

```
int wolfSSL_use_certificate_buffer(
    WOLFSSL * ssl,
    const unsigned char * in,
    long sz,
    int format
)
```

この関数は、証明書バッファを WolfSSL オブジェクトにロードします。バッファ以外のバージョンのように動作し、ファイルの代わりに入力としてバッファと呼ばれる機能が異なるだけです。バッファはサイズ

SZ の引数によって提供されます。形式バッファのフォーマットタイプを指定します。SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM。適切な使用法の例をご覧ください。

Parameters:

- **ssl** wolfSSL_new()で作成された SSL セッションへのポインタ。
- **in** ロードする証明書を含むバッファ。
- **sz** バッファにある証明書のサイズ。

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 成功時に返されます。
- SSL_BAD_FILETYPE ファイルが間違った形式である場合に返されます。
- SSL_BAD_FILE ファイルが存在しない場合に返されます。読み込め、または破損していません。
- MEMORY_E メモリ不足状態が発生した場合に返されます。
- ASN_INPUT_E base16 デコードがファイルに対して失敗した場合に返されます。

Example

```
int buffSz;
int ret;
byte certBuff[...];
WOLFSSL* ssl = 0;
...

ret = wolfSSL_use_certificate_buffer(ssl, certBuff, buffSz, SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
    // failed to load certificate from buffer
}
```

A.4.2.67 function wolfSSL_use_PrivateKey_buffer

```
int wolfSSL_use_PrivateKey_buffer(
    WOLFSSL * ssl,
    const unsigned char * in,
    long sz,
    int format
)
```

この関数は、秘密鍵バッファを WolfSSL オブジェクトにロードします。バッファ以外のバージョンのように動作し、ファイルの代わりに入力としてバッファと呼ばれる機能が異なるだけです。バッファはサイズ SZ の引数によって提供されます。形式バッファのフォーマットタイプを指定します。SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM。適切な使用法の例をご覧ください。

Parameters:

- **ssl** wolfssl_new() で作成された SSL セッションへのポインタ。
- **in** ロードする秘密鍵を含むバッファ。
- **sz** バッファにある秘密鍵のサイズ。

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 成功時に返されます。
- SSL_BAD_FILETYPE ファイルが間違った形式である場合に返されます。
- SSL_BAD_FILE ファイルが存在しない場合に返されます。読み込め、または破損していません。
- MEMORY_E メモリ不足状態が発生した場合に返されます。
- ASN_INPUT_E base16 デコードがファイルに対して失敗した場合に返されます。
- NO_PASSWORD 鍵ファイルが暗号化されているがパスワードが提供されていない場合に返されます。

Example

```
int buffSz;
int ret;
byte keyBuff[...];
WOLFSSL* ssl = 0;
...
ret = wolfSSL_use_PrivateKey_buffer(ssl, keyBuff, buffSz, SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
    // failed to load private key from buffer
}
```

A.4.2.68 function wolfSSL_use_certificate_chain_buffer

```
int wolfSSL_use_certificate_chain_buffer(
    WOLFSSL * ssl,
    const unsigned char * in,
    long sz
)
```

この関数は、証明書チェーンバッファを WolfSSL オブジェクトにロードします。バッファ以外のバージョンのように動作し、ファイルの代わりに入力としてバッファと呼ばれる機能が異なるだけです。バッファはサイズ SZ の引数によって提供されます。バッファは PEM 形式で、ルート証明書で終わる対象の証明書から始めてください。適切な使用法の例をご覧ください。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ
- **in** ロードする証明書を含むバッファ。

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 成功時に返されます。
- SSL_BAD_FILETYPE ファイルが間違った形式である場合に返されます。

- SSL_BAD_FILE ファイルが存在しない場合に返されます。読み込め、または破損していません。
- MEMORY_E メモリ不足状態が発生した場合に返されます。
- ASN_INPUT_E base16 デコードがファイルに対して失敗した場合に返されます。
- BUFFER_E チェーンバッファが受信バッファよりも大きい場合に返されます。

Example

```
int buffSz;
int ret;
byte certChainBuff[...];
WOLFSSL* ssl = 0;
...
ret = wolfSSL_use_certificate_chain_buffer(ssl, certChainBuff, buffSz);
if (ret != SSL_SUCCESS) {
    // failed to load certificate chain from buffer
}
```

A.4.2.69 function wolfSSL_UnloadCertsKeys

```
int wolfSSL_UnloadCertsKeys(
    WOLFSSL *
)
```

この関数は、SSL が所有する証明書または鍵をアンロードします。

See: wolfSSL_CTX_UnloadCAs

Return:

- SSL_SUCCESS - 関数が正常に実行された場合に返されます。
- BAD_FUNC_ARG - wolfssl オブジェクトが null の場合に返されます。

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
...
int unloadKeys = wolfSSL_UnloadCertsKeys(ssl);
if(unloadKeys != SSL_SUCCESS){
    // Failure case.
}
```

A.4.2.70 function wolfSSL_GetIVSize

```
int wolfSSL_GetIVSize(
    WOLFSSL *
)
```

WolfSSL 構造体に保持されている Specs 構造体の IV_SIZE メンバーを返します。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ

See:

- wolfSSL_GetKeySize
- wolfSSL_GetClientWriteIV
- wolfSSL_GetServerWriteIV

Return:

- iv_size ssl-> specs.iv_size で保持されている値を返します。

- BAD_FUNC_ARG WolfSSL 構造が NULL の場合に返されます。

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.
}
```

A.4.2.71 function wolfSSL_KeepArrays

```
void wolfSSL_KeepArrays(
    WOLFSSL *
)
```

通常、SSL ハンドシェイクの最後に、WolfSSL は一時的なアレイを解放します。ハンドシェイクが始まる前にこの関数を呼び出すと、WolfSSL は一時的な配列を解放するのを防ぎます。Wolfssl_get_keys() または PSK のヒントなどのものには、一時的な配列が必要になる場合があります。ユーザが一時的な配列で行われると、wolfssl_freearray() のいずれかが即座にリソースを解放することができ、あるいは、関連する SSL オブジェクトが解放されたときにリソースが解放されるようになる可能性がある。

See: wolfSSL_FreeArrays

Return: none 返品不可。

Example

```
WOLFSSL* ssl;
...
wolfSSL_KeepArrays(ssl);
```

A.4.2.72 function wolfSSL_FreeArrays

```
void wolfSSL_FreeArrays(
    WOLFSSL *
)
```

通常、SSL ハンドシェイクの最後に、WolfSSL は一時的なアレイを解放します。wolfssl_keeparrays() がハンドシェイクの前に呼び出された場合、WolfSSL は一時的な配列を解放しません。この関数は一時的な配列を明示的に解放し、ユーザーが一時的な配列で行われたときに呼び出されるべきであり、SSL オブジェクトがこれらのリソースを解放するのを待ったくない。

See: wolfSSL_KeepArrays

Return: none 返品不可。

Example

```
WOLFSSL* ssl;
...
wolfSSL_FreeArrays(ssl);
```

A.4.2.73 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
)
```

TLS キーを導き出すための外部のラッパー。

Parameters:

- **key_data** DeriveTlSkeys に割り当てられ、最終ハッシュを保持するために WC_PRF に渡されたバイトポインタ。
- **keyLen** WOLFSSL 構造体のスペックメンバーからの Derivetlskeys で派生した Word32 タイプ。
- **ms** WolfSSL 構造内でアレイ構造に保持されているマスターシークレットを保持する定数ポインタ型。
- **msLen** 列挙された定義で、マスターシークレットの長さを保持する Word32 タイプ。
- **sr** WOLFSSL 構造内の配列構造の ServerRandom メンバーへの定数バイトポインタ。
- **cr** WolfSSL 構造内の配列構造の ClientRandom メンバーへの定数バイトポインタ。
- **tls1_2** ISATLEASTLSV1_2() から返された整数型。

See:

- wc_PRF
- DeriveTlsKeys
- IsAtLeastTLSv1_2

Return:

- 0 成功に戻りました。
- BUFFER_E LABLEN と SEADLEN の合計（合計サイズを計算）が最大サイズを超えると返されます。
- MEMORY_E メモリの割り当てが失敗した場合に返されます。

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);
…
}
```

A.4.2.74 function wolfSSL_X509_get_ext_by_NID

```
int wolfSSL_X509_get_ext_by_NID(
    const WOLFSSL_X509 * x509,
    int nid,
    int lastPos
)
```

この機能は、渡された NID 値に一致する拡張索引を探して返します。

Parameters:

- **x509** 拡張のために解析する証明書。
- **nid** 見つかる拡張 OID。

Return:

- = 0 拡張インデックスが成功した場合に返されます。
- -1 拡張が見つからないかエラーが発生した場合

Example

```
const WOLFSSL_X509* x509;
int lastPos = -1;
int idx;

idx = wolfSSL_X509_get_ext_by_NID(x509, NID_basic_constraints, lastPos);
```

A.4.2.75 function wolfSSL_X509_get_ext_d2i

```
void * wolfSSL_X509_get_ext_d2i(
    const WOLFSSL_X509 * x509,
    int nid,
    int * c,
    int * idx
)
```

この関数は、渡された NID 値に合った拡張子を探して返します。

Parameters:

- **x509** 拡張のために解析する証明書。
- **nid** 見つかる拡張 OID。
- **c** not null が複数の拡張子に-2 に設定されていない場合は-1 が見つかりませんでした。

See: wolfSSL_sk_ASN1_OBJECT_free

Return:

- pointer STACK_OF（wolfssl_asn1_object）ポインタが成功した場合に返されます。
- NULL 拡張が見つからないかエラーが発生した場合

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.
```

A.4.2.76 function wolfSSL_X509_digest

```
int wolfSSL_X509_digest(
    const WOLFSSL_X509 * x509,
    const WOLFSSL_EVP_MD * digest,
    unsigned char * buf,
    unsigned int * len
)
```

この関数は DER 証明書のハッシュを返します。

Parameters:

- **x509** ハッシュを得るための証明書。
- **digest** 使用するハッシュアルゴリズム
- **buf** ハッシュを保持するためのバッファ。

See: none

Return:

- SSL_SUCCESS ハッシュの作成に成功しました。
- SSL_FAILURE 不良入力または失敗したハッシュに戻りました。

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
```

A.4.2.77 function wolfSSL_use_PrivateKey

```
int wolfSSL_use_PrivateKey(
    WOLFSSL * ssl,
    WOLFSSL_EVP_PKEY * pkey
)
```

これは WolfSSL 構造の秘密鍵を設定するために使用されます。

Parameters:

- **ssl** 引数を設定するための WolfSSL 構造。

See:

- wolfSSL_new
- wolfSSL_free

Return:

- SSL_SUCCESS 設定の成功した引数について。
- SSL_FAILURE NULL SSL が渡された場合。すべてのエラーケースは負の値になります。

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
```

A.4.2.78 function wolfSSL_use_PrivateKey_ASN1

```
int wolfSSL_use_PrivateKey_ASN1(
    int pri,
    WOLFSSL * ssl,
    unsigned char * der,
    long derSz
)
```

これは WolfSSL 構造の秘密鍵を設定するために使用されます。DER フォーマットのキーバッファが予想されます。

Parameters:

- **pri** 秘密鍵の種類。
- **ssl** 引数を設定するための WolfSSL 構造。
- **der** バッファー保持 DER キー。

See:

- wolfSSL_new
- wolfSSL_free
- wolfSSL_use_PrivateKey

Return:

- SSL_SUCCESS 秘密鍵の構文解析と設定に成功した場合。
- SSL_FAILURE NULL SSL が渡された場合。すべてのエラーケースは負の値になります。

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
```

A.4.2.79 function wolfSSL_use_RSAPrivateKey_ASN1

```
int wolfSSL_use_RSAPrivateKey_ASN1(
    WOLFSSL * ssl,
    unsigned char * der,
    long derSz
)
```

これは WolfSSL 構造の秘密鍵を設定するために使用されます。DER フォーマットの RSA キーバッファが予想されます。

Parameters:

- **ssl** 引数を設定するための WolfSSL 構造。
- **der** バッファー保持 DER キー。

See:

- wolfSSL_new
- wolfSSL_free
- wolfSSL_use_PrivateKey

Return:

- SSL_SUCCESS 秘密鍵の構文解析と設定に成功した場合。
- SSL_FAILURE NULL SSL が渡された場合。すべてのエラーケースは負の値になります。

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
```

A.4.2.80 function wolfSSL_DSA_dup_DH

```
WOLFSSL_DH * wolfSSL_DSA_dup_DH(
    const WOLFSSL_DSA * r
)
```

この関数は、DSA のパラメータを新しく作成された WOLFSSL_DH 構造体に重複しています。

See: none

Return:

- WOLFSSL_DH 重複した場合は WolfSSL_DH 構造体を返す場合
- NULL 失敗すると

Example

```
WOLFSSL_DH* dh;
WOLFSSL_DSA* dsa;
// set up dsa
dh = wolfSSL_DSA_dup_DH(dsa);

// check dh is not null
```

A.4.2.81 function wolfSSL_d2i_X509_bio

```
WOLFSSL_X509 * wolfSSL_d2i_X509_bio(
    WOLFSSL_BIO * bio,
    WOLFSSL_X509 ** x509
)
```

この関数は BIO から DER バッファを取得し、それを WolfSSL_X509 構造に変換します。

Parameters:

- **bio** DER 証明書バッファを持つ WOLFSSL_BIO 構造体体へのポインタ。

See: none

Return:

- pointer 成功した wolfssl_x509 構造ポインタを返します。
- Null 失敗時に NULL を返します

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)
```

A.4.2.82 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
)
```

この関数は wolfssl_pem_read_bio_x509 と同じように動作します。AUX は、信頼できる/拒否されたユースケースや人間の読みやすさのためのフレンドリーな名前などの追加情報を含むことを意味します。

Parameters:

- **bp** WOLFSSL_BIO 構造体体から PEM バッファを取得します。
- **x** wolfssl_x509 を機能副作用で設定する場合
- **cb** パスワードコールバック

See: wolfSSL_PEM_read_bio_X509

Return:

- WOLFSSL_X509 PEM バッファの解析に成功した場合、wolfssl_x509 構造が返されます。
- Null PEM バッファの解析に失敗した場合。

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
```

A.4.2.83 function wolfSSL_CTX_set_tmp_dh

```
long wolfSSL_CTX_set_tmp_dh(
    WOLFSSL_CTX * ctx,
    WOLFSSL_DH * dh
)
```

WOLFSSL_CTX 構造体の DH メンバーを diffie-hellman パラメータで初期化します。

Parameters:

- **ctx** wolfSSL_CTX_new()を使用して作成された WOLFSSL_CTX 構造体へのポインタ。

See: wolfSSL_BN_bn2bin

Return:

- SSL_SUCCESS 関数が正常に実行された場合に返されます。
- BAD_FUNC_ARG CTX または DH 構造体が NULL の場合に返されます。
- SSL_FATAL_ERROR 構造値を設定するエラーが発生した場合に返されます。
- MEMORY_E メモリを割り当てることができなかった場合に返されます。

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
WOLFSSL_DH* dh;
…
return wolfSSL_CTX_set_tmp_dh(ctx, dh);
```

A.4.2.84 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
)
```

この関数は、BIO の PEM バッファから DSA パラメータを取得します。

Parameters:

- **bio** PEM メモリポインタを取得するための WOLFSSL_BIO 構造体体へのポインタ。
- **x** 新しい WolfSSL_DSA 構造に設定するポインタ。
- **cb** パスワードコールバック関数

See: none

Return:

- WOLFSSL_DSA PEM バッファの解析に成功した場合、WolfSSL_DSA 構造が作成され、返されます。
- Null PEM バッファの解析に失敗した場合。

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
```

A.4.2.85 function WOLF_STACK_OF

```
WOLF_STACK_OF(
    WOLFSSL_X509
) const
```

この関数はピアの証明書チェーンを取得します。

See:

- wolfSSL_X509_get_issuer_name
- wolfSSL_X509_get_subject_name
- wolfSSL_X509_get_isCA

Return:

- pointer ピアの証明書スタックへのポインタを返します。
- NULL ピア証明書がない場合に返されます。

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
}
```

A.4.2.86 function wolfSSL_X509_get_next_altname

```
char * wolfSSL_X509_get_next_altname(
    WOLFSSL_X509 * x509
)
```

この関数は、存在する場合は、ピア証明書から altname を返します。

See:

- wolfSSL_X509_get_issuer_name
- wolfSSL_X509_get_subject_name

Return:

- NULL 次の AltName がない場合。
- cert->altNamesNext->name wolfssl_x509 から、AltName リストからの文字列値である構造が存在する場合に返されます。

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
}
```

A.4.2.87 function wolfSSL_X509_get_notBefore

```
WOLFSSL_ASN1_TIME * wolfSSL_X509_get_notBefore(
    WOLFSSL_X509 * x509
)
```

関数は、x509 が null のかどうかを確認し、そうでない場合は、WOLFSSL_X509 構造体の NotBefore メンバーを返します。

Parameters:

- **x509** WOLFSSL_X509 構造体へのポインタ

See: wolfSSL_X509_get_notAfter

Return:

- pointer WOLFSSL_ASN1_TIME へのポインタ（WOLFSSL_X509 構造体の NotBefore メンバーへのポインタ）を返します。
- NULL WOLFSSL_X509 構造体が NULL の場合に返されます。

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.5.1 Functions

	Name
long	**wolfSSL_get_verify_depth**(WOLFSSL * ssl) この関数は、有効なセッション（NULL 以外の引数 ssl）が指定された場合に、デフォルトで 9 の最大チェーン深度を返します。
char *	**wolfSSL_get_cipher_list**(int priority) この関数は引数で渡された優先順位の暗号名 (Cipher) 文字列へのポインタを返します。
int	**wolfSSL_get_ciphers**(char * buf, int len) この関数は wolfSS で有効化されている暗号名 (Cipher) を取得します。
const char *	**wolfSSL_get_cipher_name**(WOLFSSL * ssl) この関数は、引数を wolfSSL_get_cipher_name_internal に渡すことによって、DHE-RSA の形式の暗号名を取得します。
int	**wolfSSL_get_fd**(const WOLFSSL *) この関数は、SSL 接続の入出力機能として使用されるファイル記述子 (fd) を返します。通常これはソケットファイル記述子になります。
int	**wolfSSL_get_using_nonblock**(WOLFSSL *) この機能により、wolfSSL がノンブロッキング I/O を使用しているかどうかをアプリケーションが判断できます。wolfSSL がノンブロッキング I/O を使用している場合、この関数は 1 を返します。アプリケーションが WOLFSSL オブジェクトを生成した後に wolfSSL_set_using_nonblock() を呼び出してノンブロッキングソケットを使うとこの関数は１を返します。これにより、WOLFSSL オブジェクトは、recevfrom がタイムアウトせず代わりに EWOULDBLOCK を受信するようになります。

	Name
int	**wolfSSL_write**(WOLFSSL * ssl, const void * data, int sz) この関数は、バッファあるいはデータから、SSL 接続に対して、sz バイトを書き込みます。必要に応じて、wolfSSL_write() の呼び出し時点ではまだ wolfSSL_connect() または wolfSSL_accept() がまだ呼び出されていない場合、SSL/TLS セッションをネゴシエートします。wolfSSL_write() は、ブロックとノンブロッキング I/O の両方で動作します。基礎となる入出力がノンブロッキングに設定されている場合、wolfSSL_write() が要求を満たすことができなかったら wolfSSL_write() は関数呼び出しからすぐに戻ります。この場合、wolfSSL_get_error() の呼び出しは SSL_ERROR_WANT_READ または SSL_ERROR_WANT_WRITE のいずれかを返します。その結果、基礎となる I/O が準備ができたら、呼び出し側プロセスは wolfssl_write() への呼び出しを繰り返す必要があります。基礎となる入出力がブロックされている場合、WolfSSL_WRITE() は、サイズ SZ のバッファデータが完全に書かれたかエラーが発生したら、戻るだけです。
int	**wolfSSL_read**(WOLFSSL * ssl, void * data, int sz) この関数は、SSL セッション (ssl) の内部読み取りバッファから sz バイトをバッファデータに読み出します。読み取られたバイトは内部受信バッファから削除されます。必要に応じて、wolfSSL_read() の呼び出し時点ではまだ wolfSSL_connect() または wolfSSL_accept() がまだ呼び出されていない場合、SSL/TLS セッションをネゴシエートします。SSL/TLS プロトコルは、最大サイズの SSL レコードを使用します（最大レコードサイズは /wolfssl/internal.h）。そのため、wolfSSL は、レコードを処理および復号することができる前に、SSL レコード全体を内部的に読み取る必要があります。このため、wolfSSL_read() への呼び出しは、呼び出し時に復号された最大バッファサイズを返すことができます。検索され、次回の wolfSSL_read() への呼び出しで復号される内部 wolfSSL 受信バッファで待機していない追加の復号データがあるかもしれません。sz が内部読み取りバッファ内のバイト数より大きい場合、wolfSSL_read() は内部読み取りバッファで使用可能なバイトを返します。BYTES が内部読み取りバッファにバッファされていない場合は、wolfSSL_read() への呼び出しは次のレコードの処理をトリガーします。

	Name
int	**wolfSSL_peek**(WOLFSSL * ssl, void * data, int sz) この関数は SSL セッション（SSL）内部読み取りバッファから SZ バイトをバッファデータにコピーします。この関数は、内部 SSL セッション受信バッファ内のデータが削除されていないか変更されていないことを除いて、wolfssl_read() と同じです。必要に応じて、wolfssl_read() のように、wolfssl_peek() はまだ wolfssl_connect() または wolfssl_accept() によってまだ実行されていない場合、wolfssl_peek() は SSL / TLS セッションをネゴシエートします。SSL/TLS プロトコルは、最大サイズの SSL レコードを使用します（最大レコードサイズは /wolfssl/internal.h）。そのため、WolfSSL は、レコードを処理および復号化することができる前に、SSL レコード全体を内部的に読み取る必要があります。このため、wolfssl_peek() への呼び出しは、呼び出し時に復号化された最大バッファサイズを返すことができます。wolfssl_peek()/ wolfssl_read() への次の呼び出しで検索および復号化される内部 WolfSSL 受信バッファ内で待機していない追加の復号化データがあるかもしれません。SZ が内部読み取りバッファ内のバイト数よりも大きい場合、SSL_PEEK() は内部読み取りバッファで使用可能なバイトを返します。バイトが内部読み取りバッファにバッファされていない場合、Wolfssl_peek() への呼び出しは次のレコードの処理をトリガーします。
int	**wolfSSL_accept**(WOLFSSL *) この関数はサーバー側で呼び出され、SSL クライアントが SSL/TLS ハンドシェイクを開始するのを待ちます。この関数が呼び出されると、基礎となる通信チャネルはすでに設定されています。wolfSSL_accept() は、ブロックとノンブロッキング I/O の両方で動作します。基礎となる入出力がノンブロッキングである場合、wolfSSL_accept() は、基礎となる I/O が wolfSSL_accept の要求を満たすことができなかったときに戻ります。この場合、wolfSSL_get_error() への呼び出しは SSL_ERROR_WANT_READ または SSL_ERROR_WANT_WRITE のいずれかを生成します。呼び出しプロセスは、読み取り可能なデータが使用可能であり、wolfSSL が停止した場所を拾うときに、wolfSSL_accept の呼び出しを繰り返す必要があります。ノンブロッキングソケットを使用する場合は、何も実行する必要がありますが、select() を使用して必要な条件を確認できます。基礎となる I/O がブロックされている場合、wolfSSL_accept() はハンドシェイクが終了したら、またはエラーが発生したら戻ります。

	Name
int	**wolfSSL_send**(WOLFSSL * ssl, const void * data, int sz, int flags) この関数は、書き込み操作のために指定されたフラグを使用してバッファあるいはデータから、SSL 接続に対して、sz バイトを書き込みます。必要に応じて、wolfSSL_send() の呼び出し時点ではまだ wolfSSL_connect() または wolfSSL_accept() がまだ呼び出されていない場合、SSL/TLS セッションをネゴシエートします。wolfSSL_send() は、ブロックとノンブロッキング I/O の両方で動作します。基礎となる入出力がノンブロッキングに設定されている場合、wolfSSL_send() が要求を満たすことができなかったら wolfSSL_send() は関数呼び出しからすぐに戻ります。この場合、wolfSSL_get_error() の呼び出しは SSL_ERROR_WANT_READ または SSL_ERROR_WANT_WRITE のいずれかを返します。その結果、基礎となる I/O が準備ができたら、呼び出し側プロセスは wolfSSL_send() への呼び出しを繰り返す必要があります。基礎となる入出力がブロックされている場合、wolfSSL_send() は、サイズ SZ のバッファデータが完全に書かれたかエラーが発生したら、戻るだけです。
int	**wolfSSL_recv**(WOLFSSL * ssl, void * data, int sz, int flags) この関数は、基礎となる RECV 動作のために指定されたフラグを使用して、SSL セッション（ssl）内部読み取りバッファから sz バイトをバッファデータに読み出します。読み取られたバイトは内部受信バッファから削除されます。この関数は wolfssl_read() と同じです。ただし、アプリケーションが基礎となる読み取り操作の RECV フラグを設定できることを許可します。必要に応じて wolfssl_recv() が wolfssl_connect() または wolfssl_accept() によってハンドシェイクがまだ実行されていない場合は、SSL/TLS セッションをネゴシエートします。SSL/TLS プロトコルは、最大サイズの SSL レコードを使用します（最大レコードサイズは /wolfssl/internal.h）。そのため、wolfSSL は、レコードを処理および復号することができる前に、SSL レコード全体を内部的に読み取る必要があります。このため、wolfSSL_recv() への呼び出しは、呼び出し時に復号された最大バッファサイズを返すことができるだけです。wolfSSL_recv() への次の呼び出しで検索および復号される内部 wolfSSL 受信バッファで待機していない追加の復号化されたデータがあるかもしれません。引数 sz が内部読み取りバッファ内のバイト数よりも大きい場合、wolfSSL_recv() は内部読み取りバッファで使用可能なバイトを返します。バイトが内部読み取りバッファにバッファされていない場合は、wolfSSL_recv() への呼び出しは次のレコードの処理をトリガーします。

	Name
int	**wolfSSL_get_alert_history**(WOLFSSL * ssl, WOLFSSL_ALERT_HISTORY * h) この関数はアラート履歴を取得します。
WOLFSSL_SESSION *	**wolfSSL_get_session**(WOLFSSL * ssl)NO_SESSION_CACHE_REF が定義されている場合、この関数は SSL で使用されている現在のセッション（WOLFSSL_SESSION）へのポインタを返します。この関数は、WOLFSSL_SESSION オブジェクトへの永続的なポインタを返します。返されるポインタは、wolfSSL_free が呼び出されたときに解放されます。この呼び出しは、現在のセッションを検査または変更するためにのみ使用されます。セッション再開に使用する場合は、wolfSSL_get1_session() を使用することをお勧めします。NO_SESSION_CACHE_REF が定義されていない場合の後方互換性のために、この関数はローカルキャッシュに格納されている永続セッションオブジェクトポインタを返します。キャッシュサイズは有限であり、アプリケーションが wolfSSL_set_session() を呼び出す時までにセッションオブジェクトが別の SSL 接続によって上書きされる危険性があります。アプリケーションに NO_SESSION_CACHE_REF を定義し、セッション再開に wolfSSL_get1_session() を使用することをお勧めします。
void	**wolfSSL_flush_sessions**(WOLFSSL_CTX * ctx, long tm) この機能は、期限切れになったセッションキャッシュからセッションをフラッシュします。時間比較には引数 tm が使用されます。wolfSSL は現在セッションに静的テーブルを使用しているため、フラッシングは不要です。そのため、この機能は現在スタブとして存在しています。この関数は、wolfssl が OpenSSL 互換層でコンパイルされているときの OpenSSL 互換性（ssl_flush_sessions）を提供します。
int	**wolfSSL_GetSessionIndex**(WOLFSSL * ssl) この関数は、WOLFSSL 構造体の指定セッションインデックス値を取得します。
int	**wolfSSL_GetSessionAtIndex**(int index, WOLFSSL_SESSION * session) この関数はセッションキャッシュの指定されたインデックスのセッションを取得し、それをメモリにコピーします。WOLFSSL_SESSION 構造体はセッション情報を保持します。
WOLFSSL_X509_CHAIN *	**wolfSSL_SESSION_get_peer_chain**(WOLFSSL_SESSION * session)WOLFSSL_SESSION 構造体からピア証明書チェーンを返します。
int	**wolfSSL_pending**(WOLFSSL *) この関数は、wolfSSL_read() によって読み取られる WOLFSSL オブジェクトでバッファされているバイト数を返します。

	Name
int	**wolfSSL_save_session_cache**(const char * fname) この関数はセッションキャッシュをファイルに持続します。追加のメモリ使用のため、memsave は使用されません。
int	**wolfSSL_restore_session_cache**(const char * fname) この関数はファイルから永続セッションキャッシュを復元します。追加のメモリ使用のため、memstore は使用しません。
int	**wolfSSL_memsave_session_cache**(void * mem, int sz) この関数はセッションキャッシュをメモリに保持します。
int	**wolfSSL_memrestore_session_cache**(const void * mem, int sz) この関数はメモリから永続セッションキャッシュを復元します。
int	**wolfSSL_get_session_cache_memsize**(void) この関数は、セッションキャッシュ保存バッファをどのように大きくするかを返します。
int	**wolfSSL_session_reused**(WOLFSSL * ssl) この関数は、オプション構造体の再開メンバを返します。フラグはセッションを再利用するかどうかを示します。そうでなければ、新しいセッションを確立する必要があります。
const char *	**wolfSSL_get_version**(WOLFSSL * ssl) 文字列として使用されている SSL バージョンを返します。
int	**wolfSSL_get_current_cipher_suite**(WOLFSSL * ssl)SSL セッションで現在の暗号スイートを返します。
WOLFSSL_CIPHER *	**wolfSSL_get_current_cipher**(WOLFSSL * ssl) この関数は、SSL セッションの現在の暗号へのポインタを返します。
const char *	**wolfSSL_CIPHER_get_name**(const WOLFSSL_CIPHER * cipher) この関数は、SSL オブジェクト内の Cipher Suite と使用可能なスイートと一致し、文字列表現を返します。
const char *	**wolfSSL_get_cipher**(WOLFSSL * ssl) この関数は、SSL オブジェクト内の暗号スイートと使用可能なスイートと一致します。
int	**wolfSSL_BIO_get_mem_data**(WOLFSSL_BIO * bio, void * p) この関数は、内部メモリバッファの先頭へのバイトポインタを設定するために使用されます。
long	**wolfSSL_BIO_set_fd**(WOLFSSL_BIO * b, int fd, int flag) 使用する BIO のファイル記述子を設定します。
int	**wolfSSL_BIO_set_close**(WOLFSSL_BIO * b, long flag)BIO が解放されたときに I/O ストリームを閉じる必要があることを示すために使用されるクローズフラグを設定します。
WOLFSSL_BIO_METHOD *	**wolfSSL_BIO_s_socket**(void) この関数は BIO_SOCKET タイプの WOLFSSL_BIO_METHOD を取得するために使用されます。

	Name
int	**wolfSSL_BIO_set_write_buf_size**(WOLFSSL_BIO * b, long size) この関数は、WOLFSSL_BIO のライトバッファのサイズを設定するために使用されます。書き込みバッファが以前に設定されている場合、この関数はサイズをリセットするときに解放されます。読み書きインデックスを 0 にリセットするという点で、wolfSSL_BIO_reset に似ています。
int	**wolfSSL_BIO_make_bio_pair**(WOLFSSL_BIO * b1, WOLFSSL_BIO * b2) これは 2 つの BIOS を一緒にペアリングするために使用されます。一対の BIOS は、2 つの方法パイプと同様に、他方で読み取られることができ、その逆も同様である。BIOS の両方が同じスレッド内にあることが予想されます。この機能はスレッドセーフではありません。2 つの BIOS のうちの 1 つを解放すると、両方ともペアになっています。書き込みバッファサイズが以前に設定されていない場合、それはペアになる前に 17000（wolfssl_bio_size）のデフォルトサイズに設定されます。
int	**wolfSSL_BIO_ctrl_reset_read_request**(WOLFSSL_BIO * bio) この関数は、読み取り要求フラグを 0 に戻すために使用されます。
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) 関数によって返される数のバイトを書き込むためにバッファーへのポインタを取得します。返されるポインタに追加のバイトを書き込んだ場合、返された値は範囲外の書き込みにつながる可能性があります。
int	**wolfSSL_BIO_reset**(WOLFSSL_BIO * bio) バイオを初期状態にリセットします。タイプ BIO_BIO の例として、これは読み書きインデックスをリセットします。
int	**wolfSSL_BIO_seek**(WOLFSSL_BIO * bio, int ofs) この関数は、指定されたオフセットへファイルポインタを調整します。これはファイルの先頭からのオフセットです。
int	**wolfSSL_BIO_write_filename**(WOLFSSL_BIO * bio, char * name) これはファイルに設定および書き込むために使用されます。現在ファイル内のデータを上書きし、BIO が解放されたときにファイルを閉じるように設定されます。
long	**wolfSSL_BIO_set_mem_eof_return**(WOLFSSL_BIO * bio, int v) これはファイル値の終わりを設定するために使用されます。一般的な値は予想される正の値と混同されないように-1 です。

	Name
long	**wolfSSL_BIO_get_mem_ptr**(WOLFSSL_BIO * bio, WOLFSSL_BUF_MEM ** m) これは WolfSSL_BIO メモリポインタのゲッター関数です。
const char *	**wolfSSL_lib_version**(void) この関数は現在のライブラリーバージョンを返します。
word32	**wolfSSL_lib_version_hex**(void) この関数は、現在のライブラリーのバージョンを 16 進表記で返します。
int	**wolfSSL_negotiate**(WOLFSSL * ssl)SSL メソッドの側面に基づいて、実際の接続または承認を実行します。クライアント側から呼び出された場合、サーバ側から呼び出された場合に wolfssl_accept() が実行されている間に wolfssl_connect() が行われる。
int	**wolfSSL_connect_cert**(WOLFSSL * ssl) この関数はクライアント側で呼び出され、ピアの証明書チェーンを取得するのに十分な長さだけサーバーを持つ SSL / TLS ハンドシェイクを開始します。この関数が呼び出されると、基礎となる通信チャネルはすでに設定されています。 wolfssl_connect_cert() は、ブロックと非ブロック I / O の両方で動作します。基礎となる I / O がノンブロッキングである場合、 wolfsl_connect_cert() は、 wolfssl_connect_cert_cert() のニーズを満たすことができなかったときに戻ります。ハンドシェイクを続けます。この場合、wolfSSL_get_error() への呼び出しは SSL_ERROR_WANT_READ または SSL_ERROR_WANT_WRITE のいずれかを生成します。通話プロセスは、基礎となる I / O が準備ができて、wolfssl がオフになっているところを拾うときに、wolfssl_connect_cert() への呼び出しを繰り返す必要があります。ノンブロッキングソケットを使用する場合は、何も実行する必要がありますが、select() を使用して必要な条件を確認できます。基礎となる入出力がブロックされている場合、wolfssl_connect_cert() はピアの証明書チェーンが受信されたらのみ返されます。
int	**wolfSSL_writev**(WOLFSSL * ssl, const struct iovec * iov, int iovcnt)Writev Semantics をシミュレートしますが、SSL_Write() の動作のために実際にはブロックしないため、フロント追加が小さくなる可能性があるため Writev を使いやすいソフトウェアに移植する。
unsigned char	**wolfSSL_SNI_Status**(WOLFSSL * ssl, unsigned char type) この関数は SNI オブジェクトのステータスを取得します。
int	**wolfSSL_UseSecureRenegotiation**(WOLFSSL * ssl) この関数は、供給された WOLFSSL 構造の安全な再交渉を強制します。これはお勧めできません。

	Name
int	**wolfSSL_Rehandshake**(WOLFSSL * ssl) この関数は安全な再交渉ハンドシェイクを実行します。これは、WolfSSL がこの機能を妨げるように強制されます。
int	**wolfSSL_UseSessionTicket**(WOLFSSL * ssl) セッションチケットを使用するように WolfSSL 構造を強制します。定数 hous_session_ticket を定義し、定数 NO_WOLFSSL_CLIENT をこの関数を使用するように定義しないでください。
int	**wolfSSL_get_SessionTicket**(WOLFSSL * ssl, unsigned char * buf, word32 * bufSz) この機能は、セッション構造のチケットメンバーをバッファにコピーします。
int	**wolfSSL_set_SessionTicket**(WOLFSSL * ssl, const unsigned char * buf, word32 bufSz) この関数は、WolfSSL 構造体内の wolfssl_session 構造体のチケットメンバーを設定します。関数に渡されたバッファはメモリにコピーされます。
int	**wolfSSL_PrintSessionStats**(void) この関数はセッションから統計を印刷します。
int	**wolfSSL_get_session_stats**(unsigned int * active, unsigned int * total, unsigned int * peak, unsigned int * maxSessions) この関数はセッションの統計を取得します。
long	**wolfSSL_BIO_set_fp**(WOLFSSL_BIO * bio, XFILE fp, int c) これは BIO の内部ファイルポインタを設定するために使用されます。
size_t	**wolfSSL_BIO_ctrl_pending**(WOLFSSL_BIO * b) 保留中のバイト数を読み取る数を取得します。BIO タイプが BIO_BIO の場合、ペアから読み取る番号です。BIO に SSL オブジェクトが含まれている場合は、SSL オブジェクトからのデータを保留中です（WolfSSL_Pending（SSL））。bio_memory タイプがある場合は、メモリバッファのサイズを返します。
int	**wolfSSL_set_jobject**(WOLFSSL * ssl, void * objPtr) この関数は、WolfSSL 構造の jobjectref メンバーを設定します。
void *	**wolfSSL_get_jobject**(WOLFSSL * ssl) この関数は、wolfssl 構造の jobjectref メンバーを返します。

	Name
int	**wolfSSL_connect**(WOLFSSL * ssl) この関数はクライアント側で呼び出され、サーバーとのSSL/TLS ハンドシェイクを開始します。この関数が呼び出されるまでに下層の通信チャネルはすでに設定されている必要があります。wolfSSL_connect() は、ブロッキングとノンブロッキング I/O の両方で動作します。下層の I/O がノンブロッキングの場合、wolfSSL_connect() は、下層の I/O が wolfSSL_connect の要求（送信データ、受信データ）を満たすことができなかったときには即戻ります。この場合、wolfSSL_get_error() の呼び出しでSSL_ERROR_WANT_READ またはSSL_ERROR_WANT_WRITE のいずれかが返されます。呼び出したプロセスは、下層の I/O ががREADY になった時点で、WOLFSSL が停止したときから再開できるように wolfSSL_connect() への呼び出しを繰り返す必要があります。これにはselect() を使用して必要な条件が整ったかどうかを確認できます。ブロッキング I/O を使用する場合は、ハンドシェークが終了するかエラーが発生するまで戻ってきません。wolfSSL は OpenSSLと比べて証明書検証に異なるアプローチを取ります。クライアントのデフォルトポリシーはサーバーを認証することです。これは、CA 証明書を読み込まない場合、サーバーを確認することができず”_155” のエラーコードが返されます。OpenSSLと同じ振る舞い（つまり、CA 証明書のロードなしでサーバー認証を成功させる）を取らせたい場合には、セキュリティ面でお勧めはしませんが、SSL_CTX_SET_VERIFY（ctx、SSL_VERIFY_NONE、0) を呼び出すことで可能となります。
int	**wolfSSL_update_keys**(WOLFSSL * ssl) この関数は、TLS v1.3 クライアントまたはサーバーのwolfssl で呼び出されて、キーのロールオーバーを強制します。KeyUpdate メッセージがピアに送信され、新しいキーが暗号化のために計算されます。ピアは KeyUpdate メッセージを送り、新しい復号化キー WIL を計算します。この機能は、ハンドシェイクが完了した後にのみ呼び出すことができます。
int	**wolfSSL_key_update_response**(WOLFSSL * ssl, int * required) この関数は、TLS v1.3 クライアントまたはサーバーの wolfssl で呼び出され、キーのロールオーバーが進行中かどうかを判断します。wolfssl_update_keys() が呼び出されると、KeyUpdate メッセージが送信され、暗号化キーが更新されます。復号化キーは、応答が受信されたときに更新されます。

	Name
int	**wolfSSL_request_certificate**(WOLFSSL * ssl) この関数は、TLS v1.3 クライアントからクライアント証明書を要求します。これは、Web サーバーがクライアント認証やその他のものを必要とするページにサービスを提供している場合に役立ちます。接続で最大 256 の要求を送信できます。
int	**wolfSSL_connect_TLSv13は、ブロックとノンブロック I/O の両方で動作します。下層 I/O がノンブロッキングの場合、wolfSSL_connect() は、下層 I/O が wolfssl_connect の要求を満たすことができなかったときに戻ります。この場合、wolfSSL_get_error() への呼び出しは SSL_ERROR_WANT_READ または SSL_ERROR_WANT_WRITE のいずれかを生成します。通話プロセスは、下層 I/O が READY および WOLFSSL が停止したときに wolfssl_connect() への呼び出しを繰り返す必要があります。ノンブロッキングソケットを使用する場合は、何も実行する必要がありますが、select() を使用して必要な条件を確認できます。基礎となる入出力がブロックされている場合、wolfssl_connect() はハンドシェイクが終了したら、またはエラーが発生したらのみ戻ります。WolfSSL は OpenSSL よりも証明書検証に異なるアプローチを取ります。クライアントのデフォルトポリシーはサーバーを確認することです。これは、CAS を読み込まない場合、サーバーを確認することができ、確認できません（_155）。SSL_CONNECT を持つことの OpenSSL の動作が成功した場合は、サーバーを検証してセキュリティを抑えることができます。SSL_CTX_SET_VERIFY（CTX、SSL_VERIFY_NONE、0）。ssl_new() を呼び出す前に。お勧めできませんが。

	Name
	**wolfSSL_accept_TLSv13は、ブロックとノンブロッキング I/O の両方で動作します。下層の入出力がノンブロッキングである場合、wolfSSL_accept() は、下層の I/O が wolfSSL_accept の要求を満たすことができなかったときに戻ります。この場合、wolfSSL_get_error() への呼び出しは SSL_ERROR_WANT_READ または SSL_ERROR_WANT_WRITE のいずれかを生成します。通話プロセスは、読み取り可能なデータが使用可能であり、wolfssl が停止した場所を拾うときに、wolfssl_accept の呼び出しを繰り返す必要があります。ノンブロッキングソケットを使用する場合は、何も実行する必要がありますが、select() を使用して必要な条件を確認できます。下層の I/O がブロックされている場合、wolfssl_accept() はハンドシェイクが終了したら、またはエラーが発生したら戻ります。古いバージョンの ClientHello メッセージがサポートされていますが、TLS v1.3 接続を期待するときにこの関数を呼び出します。
int	**wolfSSL_write_early_dataまたは wolfSSL_connect_tlsv13() の代わりにこの関数を呼び出して、サーバーに接続してハンドシェイクにデータを送ります。この機能はクライアントでのみ使用されます。
int	**wolfSSL_read_early_data**(WOLFSSL * ssl, void * data, int sz, int * outSz) この関数は、再開時にクライアントからの早期データを読み取ります。wolfssl_accept() または wolfssl_accept_tlsv13() の代わりにこの関数を呼び出して、クライアントを受け入れ、ハンドシェイク内の早期データを読み取ります。ハンドシェイクよりも早期データがない場合は、通常として処理されます。この機能はサーバーでのみ使用されます。
void *	**wolfSSL_GetIOReadCtx**(WOLFSSL * ssl) この関数は、WolfSSL 構造体の IOCB_READCTX メンバーを返します。
void *	**wolfSSL_GetIOWriteCtx**(WOLFSSL * ssl) この関数は、WolfSSL 構造の IOCB_WRITECTX メンバーを返します。
void	**wolfSSL_SetIO_NetX**(WOLFSSL * ssl, NX_TCP_SOCKET * nxsocket, ULONG waitoption) この関数は、wolfssl 構造内の nxctx 構造体の NxSocket メンバーと NXWAIT メンバーを設定します。

A.5.2 Functions Documentation

A.5.2.1 function wolfSSL_get_verify_depth

```
long wolfSSL_get_verify_depth(
    WOLFSSL * ssl
```

```
)
```

この関数は、有効なセッション（NULL 以外の引数 ssl）が指定された場合に、デフォルトで 9 の最大チェーン深度を返します。

See: wolfSSL_CTX_get_verify_depth

Return:

- MAX_CHAIN_DEPTH WOLFSSL 構造体が NULL ではない場合に返されます。デフォルトでは値は 9 です。
- BAD_FUNC_ARG WOLFSSL 構造体が NULL の場合に返されます。

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
}
```

A.5.2.2 function wolfSSL_get_cipher_list

```
char * wolfSSL_get_cipher_list(
    int priority
)
```

この関数は引数で渡された優先順位の暗号名 (Cipher) 文字列へのポインタを返します。

Parameters:

- **priority** 整数値で指定する優先順位

See:

- wolfSSL_CIPHER_get_name
- wolfSSL_get_current_cipher

Return:

- 成功時には暗号名 (Cipher) 文字列へのポインタを返します。
- 0 引数で渡された優先順位が範囲外かあるいは無効な値であった場合に返されます。

Example

```
printf("The cipher at 1 is %s", wolfSSL_get_cipher_list(1));
```

A.5.2.3 function wolfSSL_get_ciphers

```
int wolfSSL_get_ciphers(
    char * buf,
    int len
)
```

この関数は wolfSS で有効化されている暗号名 (Cipher) を取得します。

Parameters:

- **buf** 文字列を格納するバッファへのポインタ。
- **len** バッファのサイズ

See:

- GetCipherNames
- wolfSSL_get_cipher_list
- ShowCiphers

Return:

- SSL_SUCCESS 関数がエラーなしで実行された場合に返されます。
- BAD_FUNC_ARG 引数 buf が NULL の場合、または引数 len がゼロ以下の場合に返されます。
- BUFFER_E バッファが十分に大きくなく、オーバーフローする可能性がある場合に返されます。

Example

```
static void ShowCiphers(void){
char* ciphers;
int ret = wolfSSL_get_ciphers(ciphers, (int)sizeof(ciphers));

if(ret == SSL_SUCCES){
        printf(“%s\n”, ciphers);
    }
}
```

A.5.2.4 function wolfSSL_get_cipher_name

```
const char * wolfSSL_get_cipher_name(
    WOLFSSL * ssl
)
```

この関数は、引数を wolfSSL_get_cipher_name_internal に渡すことによって、DHE-RSA の形式の暗号名を取得します。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ

See:

- wolfSSL_CIPHER_get_name
- wolfSSL_get_current_cipher
- wolfSSL_get_cipher_name_internal

Return:

- 成功時には一致した暗号スイートの文字列表現を返します。
- NULL エラーまたは暗号が見つからない場合に返されます。

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);
}
```

A.5.2.5 function wolfSSL_get_fd

```
int wolfSSL_get_fd(
    const WOLFSSL *
)
```

この関数は、SSL 接続の入出力機能として使用されるファイル記述子 (fd) を返します。通常これはソケットファイル記述子になります。

See: wolfSSL_set_fd

Return: fd 成功時には SSL セッションに関連つけられているファイル記述子を返します。

Example

```
int sockfd;
WOLFSSL* ssl = 0;
...
sockfd = wolfSSL_get_fd(ssl);
...
```

A.5.2.6 function wolfSSL_get_using_nonblock

```
int wolfSSL_get_using_nonblock(
    WOLFSSL *
)
```

この機能により、wolfSSL がノンブロッキング I/O を使用しているかどうかをアプリケーションが判断できます。wolfSSL がノンブロッキング I/O を使用している場合、この関数は 1 を返します。アプリケーションが WOLFSSL オブジェクトを生成した後に wolfSSL_set_using_nonblock() を呼び出してノンブロッキングソケットを使うとこの関数は 1 を返します。これにより、WOLFSSL オブジェクトは、recevfrom がタイムアウトせず代わりに EWOULDBLOCK を受信するようになります。

See: wolfSSL_set_session

Return:

- 0 基礎となる I/O がブロックされています。
- 1 基礎となる I/O は非ブロッキングです。

Example

```
int ret = 0;
WOLFSSL* ssl = 0;
...
ret = wolfSSL_get_using_nonblock(ssl);
if (ret == 1) {
    // underlying I/O is non-blocking
}
...
```

A.5.2.7 function wolfSSL_write

```
int wolfSSL_write(
    WOLFSSL * ssl,
    const void * data,
    int sz
)
```

この関数は、バッファあるいはデータから、SSL 接続に対して、sz バイトを書き込みます。必要に応じて、wolfSSL_write() の呼び出し時点ではまだ wolfSSL_connect() または wolfSSL_accept() がまだ呼び出されていない場合、SSL/TLS セッションをネゴシエートします。wolfSSL_write() は、ブロックとノンブロッキング I/O の両方で動作します。基礎となる入出力がノンブロッキングに設定されている場合、wolfSSL_write() が要求を満たすことができなかったら wolfSSL_write() は関数呼び出しからすぐに戻ります。この場合、wolfSSL_get_error() の呼び出しは SSL_ERROR_WANT_READ または SSL_ERROR_WANT_WRITE のいずれかを返します。その結果、基礎となる I/O が準備ができたら、呼び出し側プロセスは wolfssl_write() への呼び出しを繰り返す必要があります。基礎となる入出力がブロックされている場合、WolfSSL_WRITE() は、サイズ SZ のバッファデータが完全に書かれたかエラーが発生したら、戻るだけです。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ
- **data** ピアに送信されるデータを含んでいるバッファへのポインタ。
- **sz** 送信データを含んでいるバッファのサイズ

See:

- wolfSSL_send
- wolfSSL_read
- wolfSSL_recv

Return:

- 成功時には書き込んだバイト数（1 以上）を返します。
- 0 失敗したときに返されます。特定のエラーコードについて wolfSSL_get_error() を呼び出します。
- SSL_FATAL_ERROR エラーが発生したとき、または非ブロッキングソケットを使用するときには、SSL_ERROR_WANT_READ または SSL_ERROR_WANT_WRITE エラーが受信され、再度 WOLFSSL_WRITE() を呼び出す必要がある場合は、障害が発生します。特定のエラーコードを取得するには、wolfSSL_get_error() を使用してください。

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()
}
```

A.5.2.8 function wolfSSL_read

```
int wolfSSL_read(
    WOLFSSL * ssl,
    void * data,
    int sz
)
```

この関数は、SSL セッション (ssl) の内部読み取りバッファから sz バイトをバッファデータに読み出します。読み取られたバイトは内部受信バッファから削除されます。必要に応じて、wolfSSL_read() の呼び出し時点ではまだ wolfSSL_connect() または wolfSSL_accept() がまだ呼び出されていない場合、SSL/TLS セッションをネゴシエートします。SSL/TLS プロトコルは、最大サイズの SSL レコードを使用します（最大レコードサイズは /wolfssl/internal.h）。そのため、wolfSSL は、レコードを処理および復号することができる前に、SSL レコード全体を内部的に読み取る必要があります。このため、wolfSSL_read() への呼び出しは、呼び

出し時に復号された最大バッファサイズを返すことができます。検索され、次回の wolfSSL_read() への呼び出しで復号される内部 wolfSSL 受信バッファで待機していない追加の復号データがあるかもしれません。sz が内部読み取りバッファ内のバイト数より大きい場合、wolfSSL_read() は内部読み取りバッファで使用可能なバイトを返します。BYTES が内部読み取りバッファにバッファされていない場合は、wolfSSL_read() への呼び出しは次のレコードの処理をトリガーします。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ
- **data** wolfSSL_read()が読み取るデータを格納するバッファへのポインタ。
- **sz** バッファに読み取るデータのサイズ

See:

- wolfSSL_recv
- wolfSSL_write
- wolfSSL_peek
- wolfSSL_pending

Return:

- 成功時には読み取られたバイト数（1 以上）を返します。
- 0 失敗したときに返されます。これは、クリーン（通知アラートを閉じる）シャットダウンまたはピアが接続を閉じただけであることによって発生する可能性があります。特定のエラーコードについて wolfSSL_get_error() を呼び出します。
- SSL_FATAL_ERROR エラーが発生したとき、またはノンブロッキングソケットを使用するときに、SSL_ERROR_WANT_READ または SSL_ERROR_WANT_WRITE エラーが受信され、再度 wolfSSL_read() を呼び出す必要がある場合は、障害が発生します。特定のエラーコードを取得するには、wolfSSL_get_error() を使用してください。

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().
```

A.5.2.9 function wolfSSL_peek

```
int wolfSSL_peek(
    WOLFSSL * ssl,
    void * data,
    int sz
)
```

この関数は SSL セッション（SSL）内部読み取りバッファから SZ バイトをバッファデータにコピーします。この関数は、内部 SSL セッション受信バッファ内のデータが削除されていないか変更されていないことを除いて、wolfssl_read() と同じです。必要に応じて、wolfssl_read() のように、wolfssl_peek() はまだ wolfssl_connect() または wolfssl_accept() によってまだ実行されていない場合、wolfssl_peek() は SSL / TLS セッションをネゴシエートします。SSL/TLS プロトコルは、最大サイズの SSL レコードを使用します（最大レコードサイズは /wolfssl/internal.h）。そのため、WolfSSL は、レコードを処理および復号化することができる前に、SSL レコード全体を内部的に読み取る必要があります。このため、wolfssl_peek()

への呼び出しは、呼び出し時に復号化された最大バッファサイズを返すことができます。wolfssl_peek()/wolfssl_read() への次の呼び出しで検索および復号化される内部 WolfSSL 受信バッファ内で待機していない追加の復号化データがあるかもしれません。SZ が内部読み取りバッファ内のバイト数よりも大きい場合、SSL_PEEK() は内部読み取りバッファで使用可能なバイトを返します。バイトが内部読み取りバッファにバッファされていない場合、Wolfssl_peek() への呼び出しは次のレコードの処理をトリガーします。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ
- **data** wolfSSL_peek()がデータを読み取るバッファー。
- **sz** バッファに読み取るデータのサイズ

See: wolfSSL_read

Return:

- 成功時には読み取られたバイト数（1 以上）を返します。
- 0 失敗したときに返されます。これは、クリーン（通知アラートを閉じる）シャットダウンまたはピアが接続を閉じただけであることによって発生する可能性があります。特定のエラーコードについて wolfSSL_get_error() を呼び出します。
- SSL_FATAL_ERROR エラーが発生したとき、またはノンブロッキングソケットを使用するときに、SSL_ERROR_WANT_READ または SSL_ERROR_WANT_WRITE エラーが受信され、再度 wolfSSL_peek() を呼び出す必要がある場合は、障害が発生します。特定のエラーコードを取得するには、wolfSSL_get_error() を使用してください。

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”
}
```

A.5.2.10 function wolfSSL_accept

```
int wolfSSL_accept(
    WOLFSSL *
)
```

この関数はサーバー側で呼び出され、SSL クライアントが SSL/TLS ハンドシェイクを開始するのを待ちます。この関数が呼び出されると、基礎となる通信チャネルはすでに設定されています。wolfSSL_accept() は、ブロックとノンブロッキング I/O の両方で動作します。基礎となる入出力がノンブロッキングである場合、wolfSSL_accept() は、基礎となる I/O が wolfSSL_accept の要求を満たすことができなかったときに戻ります。この場合、wolfSSL_get_error() への呼び出しは SSL_ERROR_WANT_READ または SSL_ERROR_WANT_WRITE のいずれかを生成します。呼び出しプロセスは、読み取り可能なデータが使用可能であり、wolfSSL が停止した場所を拾うときに、wolfSSL_accept の呼び出しを繰り返す必要があります。ノンブロッキングソケットを使用する場合は、何も実行する必要がありますが、select() を使用して必要な条件を確認できます。基礎となる I/O がブロックされている場合、wolfSSL_accept() はハンドシェイクが終了したら、またはエラーが発生したら戻ります。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ

See:

- wolfSSL_get_error

- wolfSSL_connect

Return:

- SSL_SUCCESS 成功時に返されます。
- SSL_FATAL_ERROR エラーが発生した場合に返されます。より詳細なエラーコードを取得するには、wolfSSL_get_error() を呼び出します。

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));
}
```

A.5.2.11 function wolfSSL_send

```
int wolfSSL_send(
    WOLFSSL * ssl,
    const void * data,
    int sz,
    int flags
)
```

この関数は、書き込み操作のために指定されたフラグを使用してバッファあるいはデータから、SSL 接続に対して、sz バイトを書き込みます。必要に応じて、wolfSSL_send() の呼び出し時点ではまだ wolfSSL_connect() または wolfSSL_accept() がまだ呼び出されていない場合、SSL/TLS セッションをネゴシエートします。wolfSSL_send() は、ブロックとノンブロッキング I/O の両方で動作します。基礎となる入出力がノンブロッキングに設定されている場合、wolfSSL_send() が要求を満たすことができなかったら wolfSSL_send() は関数呼び出しからすぐに戻ります。この場合、wolfSSL_get_error() の呼び出しは SSL_ERROR_WANT_READ または SSL_ERROR_WANT_WRITE のいずれかを返します。その結果、基礎となる I/O が準備ができたら、呼び出し側プロセスは wolfSSL_send() への呼び出しを繰り返す必要があります。基礎となる入出力がブロックされている場合、wolfSSL_send() は、サイズ SZ のバッファデータが完全に書かれたかエラーが発生したら、戻るだけです。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ
- **data** ピアに送信されるデータを含んでいるバッファへのポインタ。
- **sz** 送信データを含んでいるバッファのサイズ
- **flags** 下層の I/O の send に対して指定するフラグ

See:

- wolfSSL_write
- wolfSSL_read
- wolfSSL_recv

Return:

- 成功時には書き込んだバイト数（1 以上）を返します。
- 0 失敗したときに返されます。特定のエラーコードについて wolfSSL_get_error() を呼び出します。

- SSL_FATAL_ERROR エラーが発生したとき、または非ブロッキングソケットを使用するときには、SSL_ERROR_WANT_READ または SSL_ERROR_WANT_WRITE エラーが受信され、再度 WOLFSSL_WRITE() を呼び出す必要がある場合は、障害が発生します。特定のエラーコードを取得するには、wolfSSL_get_error() を使用してください。

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
}
```

A.5.2.12 function wolfSSL_recv

```
int wolfSSL_recv(
    WOLFSSL * ssl,
    void * data,
    int sz,
    int flags
)
```

この関数は、基礎となる RECV 動作のために指定されたフラグを使用して、SSL セッション（ssl）内部読み取りバッファから sz バイトをバッファデータに読み出します。読み取られたバイトは内部受信バッファから削除されます。この関数は wolfssl_read() と同じです。ただし、アプリケーションが基礎となる読み取り操作の RECV フラグを設定できることを許可します。必要に応じて wolfssl_recv() が wolfssl_connect() または wolfssl_accept() によってハンドシェイクがまだ実行されていない場合は、SSL/TLS セッションをネゴシエートします。SSL/TLS プロトコルは、最大サイズの SSL レコードを使用します（最大レコードサイズは /wolfssl/internal.h）。そのため、wolfSSL は、レコードを処理および復号することができる前に、SSL レコード全体を内部的に読み取る必要があります。このため、wolfSSL_recv() への呼び出しは、呼び出し時に復号された最大バッファサイズを返すことができるだけです。wolfSSL_recv() への次の呼び出しで検索および復号される内部 wolfSSL 受信バッファで待機していない追加の復号化されたデータがあるかもしれません。引数 sz が内部読み取りバッファ内のバイト数よりも大きい場合、wolfSSL_recv() は内部読み取りバッファで使用可能なバイトを返します。バイトが内部読み取りバッファにバッファされていない場合は、wolfSSL_recv() への呼び出しは次のレコードの処理をトリガーします。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ
- **data** wolfSSL_recv()がデータを読み取るバッファー。
- **sz** データを読み込むためのバイト数。

See:

- wolfSSL_read
- wolfSSL_write
- wolfSSL_peek
- wolfSSL_pending

Return:

- 成功時には読み取られたバイト数 (1 以上) を返します。

- 0 失敗したときに返されます。これは、クリーン（通知アラートを閉じる）シャットダウンまたはピアが接続を閉じただけであることによって発生する可能性があります。特定のエラーコードについてwolfSSL_get_error() を呼び出します。
- SSL_FATAL_ERROR エラーが発生した場合、または非ブロッキングソケットを使用するときには、SSL_ERROR_WANT_READ または SSL_ERROR_WANT_WRITE エラーが発生し、アプリケーションが再び WOLFSSL_RECV() を呼び出す必要があります。特定のエラーコードを取得するには、wolfSSL_get_error() を使用してください。

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”
}
```

A.5.2.13 function wolfSSL_get_alert_history

```
int wolfSSL_get_alert_history(
    WOLFSSL * ssl,
    WOLFSSL_ALERT_HISTORY * h
)
```

この関数はアラート履歴を取得します。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WolfSSL 構造へのポインタ。
- **h** WOLFSSL 構造体の “alert_history member” の値が格納される、WOLFSSL_ALERT_HISTORY 構造体へのポインタ。

See: wolfSSL_get_error

Return: SSL_SUCCESS 関数が正常に完了したときに返されます。警告履歴があったか、またはいずれにも、戻り値は SSL_SUCCESS です。

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
```

A.5.2.14 function wolfSSL_get_session

```
WOLFSSL_SESSION * wolfSSL_get_session(
    WOLFSSL * ssl
)
```

NO_SESSION_CACHE_REF が定義されている場合、この関数は SSL で使用されている現在のセッション（WOLFSSL_SESSION）へのポインタを返します。この関数は、WOLFSSL_SESSION オブジェクトへの永続的なポインタを返します。返されるポインタは、wolfSSL_free が呼び出されたときに解放されます。この呼び出しは、現在のセッションを検査または変更するためにのみ使用されます。セッション再開に使用する場

合は、wolfSSL_get1_session() を使用することをお勧めします。NO_SESSION_CACHE_REF が定義されていない場合の後方互換性のために、この関数はローカルキャッシュに格納されている永続セッションオブジェクトポインタを返します。キャッシュサイズは有限であり、アプリケーションが wolfSSL_set_session() を呼び出す時までにセッションオブジェクトが別の SSL 接続によって上書きされる危険性があります。アプリケーションに NO_SESSION_CACHE_REF を定義し、セッション再開に wolfSSL_get1_session() を使用することをお勧めします。

See:

- wolfSSL_get1_session
- wolfSSL_set_session

Return:

- 現在の SSL セッションオブジェクトへのポインタを返します。
- NULL ssl が NULL の場合、SSL セッションキャッシュが無効になっている場合、wolfSSL はセッション ID を使用できない、またはミューテックス関数が失敗した場合に返されます。

Example

```
WOLFSSL* ssl;
WOLFSSL_SESSION* session;
...
session = wolfSSL_get_session(ssl);
if (session == NULL) {
    // failed to get session pointer
}
...
```

A.5.2.15 function wolfSSL_flush_sessions

```
void wolfSSL_flush_sessions(
    WOLFSSL_CTX * ctx,
    long tm
)
```

この機能は、期限切れになったセッションキャッシュからセッションをフラッシュします。時間比較には引数 tm が使用されます。wolfSSL は現在セッションに静的テーブルを使用しているため、フラッシングは不要です。そのため、この機能は現在スタブとして存在しています。この関数は、wolfssl が OpenSSL 互換層でコンパイルされているときの OpenSSL 互換性（ssl_flush_sessions）を提供します。

Parameters:

- **ctx** wolfSSL_CTX_new()を使用して作成された WOLFSSL_CTX 構造体へのポインタ。
- **tm** セッションの有効期限の比較で使用される時間

See:

- wolfSSL_get1_session
- wolfSSL_set_session

Return: なし

Example

```
WOLFSSL_CTX* ssl;
...
wolfSSL_flush_sessions(ctx, time(0));
```

A.5.2.16 function wolfSSL_GetSessionIndex

```
int wolfSSL_GetSessionIndex(
    WOLFSSL * ssl
)
```

この関数は、WOLFSSL 構造体の指定セッションインデックス値を取得します。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ。

See: wolfSSL_GetSessionAtIndex

Return: この関数は、WOLFSSL 構造体内の SessionIndex を表す int 型の値を返します。

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.
}
```

A.5.2.17 function wolfSSL_GetSessionAtIndex

```
int wolfSSL_GetSessionAtIndex(
    int index,
    WOLFSSL_SESSION * session
)
```

この関数はセッションキャッシュの指定されたインデックスのセッションを取得し、それをメモリにコピーします。WOLFSSL_SESSION 構造体はセッション情報を保持します。

Parameters:

- **idx** セッションインデックス値
- **session** WOLFSSL_SESSION 構造体へのポインタ

See:

- UnLockMutex
- LockMutex
- wolfSSL_GetSessionIndex

Return:

- SSL_SUCCESS 関数が正常に実行され、エラーがスローされなかった場合に返されます。
- BAD_MUTEX_E アンロックまたはロックミューテックスエラーが発生した場合に返されます。
- SSL_FAILURE 関数が正常に実行されなかった場合に返されます。

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.
}
```

A.5.2.18 function wolfSSL_SESSION_get_peer_chain

```
WOLFSSL_X509_CHAIN * wolfSSL_SESSION_get_peer_chain(
    WOLFSSL_SESSION * session
)
```

WOLFSSL_SESSION 構造体からピア証明書チェーンを返します。

Parameters:

- **session** WOLFSSL_SESSION 構造体へのポインタ

See:

- wolfSSL_GetSessionAtIndex
- wolfSSL_GetSessionIndex
- AddSession

Example

```
WOLFSSL_SESSION* session;
WOLFSSL_X509_CHAIN* chain;
...
chain = wolfSSL_SESSION_get_peer_chain(session);
if(!chain){
    // There was no chain. Failure case.
}
```

A.5.2.19 function wolfSSL_pending

```
int wolfSSL_pending(
    WOLFSSL *
)
```

この関数は、wolfSSL_read() によって読み取られる WOLFSSL オブジェクトでバッファされているバイト数を返します。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ

See:

- wolfSSL_recv
- wolfSSL_read
- wolfSSL_peek

Return: この関数は、保留中のバイト数を返します。

Example

```
int pending = 0;
WOLFSSL* ssl = 0;
...

pending = wolfSSL_pending(ssl);
printf("There are %d bytes buffered and available for reading", pending);
```

A.5.2.20 function wolfSSL_save_session_cache

```
int wolfSSL_save_session_cache(
    const char * fname
)
```

この関数はセッションキャッシュをファイルに持続します。追加のメモリ使用のため、memsave は使用されません。

Parameters:

- **fname** 書き込み対象ファイル名へのポインタ。

See:

- XFWRITE
- wolfSSL_restore_session_cache
- wolfSSL_memrestore_session_cache

Return:

- SSL_SUCCESS 関数がエラーなしで実行された場合に返されます。セッションキャッシュはファイルに書き込まれました。
- SSL_BAD_FILE FNAME を開くことができないか、それ以外の場合は破損した場合に返されます。
- FWRITE_ERROR XfWrite がファイルへの書き込みに失敗した場合に返されます。
- BAD_MUTEX_E ミューテックスロック障害が発生した場合に返されます。

Example

```
const char* fname;
...
if(wolfSSL_save_session_cache(fname) != SSL_SUCCESS){
    // Fail to write to file.
}
```

A.5.2.21 function wolfSSL_restore_session_cache

```
int wolfSSL_restore_session_cache(
    const char * fname
)
```

この関数はファイルから永続セッションキャッシュを復元します。追加のメモリ使用のため、memstore は使用しません。

Parameters:

- **fname** キャッシュを読み取るためのファイル名へのポインタ。

See:

- XFREAD
- XFOPEN

Return:

- SSL_SUCCESS 関数がエラーなしで実行された場合に返されます。
- SSL_BAD_FILE 関数に渡されたファイルが破損していて XFOPEN によって開くことができなかった場合に返されます。
- FREAD_ERROR ファイルに XFREAD から読み取りエラーが発生した場合に返されます。
- CACHE_MATCH_ERROR セッションキャッシュヘッダの一致が失敗した場合に返されます。
- BAD_MUTEX_E ミューテックスロック障害が発生した場合に返されます。

Example

```
const char *fname;
...
if(wolfSSL_restore_session_cache(fname) != SSL_SUCCESS){
    // Failure case. The function did not return SSL_SUCCESS.
}
```

A.5.2.22 function wolfSSL_memsave_session_cache

```
int wolfSSL_memsave_session_cache(
    void * mem,
    int sz
)
```

この関数はセッションキャッシュをメモリに保持します。

Parameters:

- **mem** セッションキャッシュのコピー先バッファへのポインタ
- **sz** コピー先バッファのサイズ

See:

- XMEMCPY
- wolfSSL_get_session_cache_memsize

Return:

- SSL_SUCCESS 関数がエラーなしで実行された場合に返されます。セッションキャッシュはメモリに正常に永続化されました。
- BAD_MUTEX_E ミューテックスロックエラーが発生した場合に返されます。
- BUFFER_E バッファサイズが小さすぎると返されます。

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
}
```

A.5.2.23 function wolfSSL_memrestore_session_cache

```
int wolfSSL_memrestore_session_cache(
    const void * mem,
    int sz
)
```

この関数はメモリから永続セッションキャッシュを復元します。

Parameters:

- **mem** セッションキャッシュを保持しているバッファへのポインタ。
- **sz** バッファのサイズ

See: wolfSSL_save_session_cache

Return:

- SSL_SUCCESS 関数がエラーなしで実行された場合に返されます。
- BUFFER_E メモリバッファが小さすぎると返されます。
- BAD_MUTEX_E セッションキャッシュミューテックスロックが失敗した場合に返されます。
- CACHE_MATCH_ERROR セッションキャッシュヘッダの一致が失敗した場合に返されます。

Example

```
const void* memoryFile;
int szMf;
...
```

```
if(wolfSSL_memrestore_session_cache(memoryFile, szMf) != SSL_SUCCESS){
    // Failure case. SSL_SUCCESS was not returned.
}
```

A.5.2.24 function wolfSSL_get_session_cache_memsize

```
int wolfSSL_get_session_cache_memsize(
    void
)
```

この関数は、セッションキャッシュ保存バッファをどのように大きくするかを返します。

See: wolfSSL_memrestore_session_cache

Return: この関数は、セッションキャッシュ保存バッファのサイズを表す整数を返します。

Example

```
int sz = // Minimum size for error checking;
...
if(sz < wolfSSL_get_session_cache_memsize()){
    // Memory buffer is too small
}
```

A.5.2.25 function wolfSSL_session_reused

```
int wolfSSL_session_reused(
    WOLFSSL * ssl
)
```

この関数は、オプション構造体の再開メンバを返します。フラグはセッションを再利用するかどうかを示します。そうでなければ、新しいセッションを確立する必要があります。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ

See:

- wolfSSL_SESSION_free
- wolfSSL_GetSessionIndex
- wolfSSL_memsave_session_cache

Return: This 関数セッションの再利用のフラグを表すオプション構造に保持されている int 型を返します。

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
...
if(!wolfSSL_session_reused(sslResume)){
    // No session reuse allowed.
}
```

A.5.2.26 function wolfSSL_get_version

```
const char * wolfSSL_get_version(
    WOLFSSL * ssl
)
```

文字列として使用されている SSL バージョンを返します。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ

See: wolfSSL_lib_version

Return:

- “SSLv3” SSLv3 を使う
- “TLSv1” TLSV1 を使用する
- “TLSv1.1” TLSV1.1 を使用する
- “TLSv1.2” TLSV1.2 を使用する
- “TLSv1.3” TLSV1.3 を使用する
- “DTLS”: DTLS を使う
- “DTLSv1.2” DTLSV1.2 を使用する
- “unknown” どのバージョンの TLS が使用されているかを判断するという問題がありました。

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));
```

A.5.2.27 function wolfSSL_get_current_cipher_suite

```
int wolfSSL_get_current_cipher_suite(
    WOLFSSL * ssl
)
```

SSL セッションで現在の暗号スイートを返します。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ

See:

- wolfSSL_CIPHER_get_name
- wolfSSL_get_current_cipher
- wolfSSL_get_cipher_list

Return:

- ssl->options.cipherSuite 現在の暗号スイートを表す整数。
- 0 提供されている SSL セッションは NULL です。

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
}
```

A.5.2.28 function wolfSSL_get_current_cipher

```
WOLFSSL_CIPHER * wolfSSL_get_current_cipher(
    WOLFSSL * ssl
)
```

この関数は、SSL セッションの現在の暗号へのポインタを返します。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ

See:

- wolfSSL_get_cipher
- wolfSSL_get_cipher_name_internal
- wolfSSL_get_cipher_name

Return:

- The 関数 WolfSSL 構造体の暗号メンバーのアドレスを返します。これは wolfssl_icipher 構造へのポインタです。
- NULL WolfSSL 構造が NULL の場合に返されます。

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
}
```

A.5.2.29 function wolfSSL_CIPHER_get_name

```
const char * wolfSSL_CIPHER_get_name(
    const WOLFSSL_CIPHER * cipher
)
```

この関数は、SSL オブジェクト内の Cipher Suite と使用可能なスイートと一致し、文字列表現を返します。

Parameters:

- **cipher** WOLFSSL_CIPHER 構造体へのポインタ

See:

- wolfSSL_get_cipher
- wolfSSL_get_current_cipher
- wolfSSL_get_cipher_name_internal
- wolfSSL_get_cipher_name

Return:

- string この関数は、一致した暗号スイートの文字列表現を返します。
- none スイートが一致していない場合は「なし」を返します。

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
}
```

A.5.2.30 function wolfSSL_get_cipher

```
const char * wolfSSL_get_cipher(
    WOLFSSL * ssl
)
```

この関数は、SSL オブジェクト内の暗号スイートと使用可能なスイートと一致します。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ

See:

- wolfSSL_CIPHER_get_name
- wolfSSL_get_current_cipher

Return: This 関数 Suite が一致させた String 値を返します。スイートが一致していない場合は「なし」を返します。

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
```

A.5.2.31 function wolfSSL_BIO_get_mem_data

```
int wolfSSL_BIO_get_mem_data(
    WOLFSSL_BIO * bio,
    void * p
)
```

この関数は、内部メモリバッファの先頭へのバイトポインタを設定するために使用されます。

Parameters:

- **bio** のメモリバッファを取得するための WOLFSSL_BIO 構造体。
- **p** メモリバッファへのポインタ。

See:

- wolfSSL_BIO_new

- wolfSSL_BIO_s_mem
- wolfSSL_BIO_set_fp
- wolfSSL_BIO_free

Return:

- size 成功すると、バッファのサイズが返されます
- SSL_FATAL_ERROR エラーケースに遭遇した場合

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
```

A.5.2.32 function wolfSSL_BIO_set_fd

```
long wolfSSL_BIO_set_fd(
    WOLFSSL_BIO * b,
    int fd,
    int flag
)
```

使用する BIO のファイル記述子を設定します。

Parameters:

- **bio** FD を設定するための WOLFSSL_BIO 構造。
- **fd** 使用するファイル記述子。
- **closeF** fd をクローズする際のふるまいを指定するフラグ

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_free

Return: SSL_SUCCESS(1) 成功時に返されます。

Example

```
WOLFSSL_BIO* bio;
int fd;
// setup bio
wolfSSL_BIO_set_fd(bio, fd, BIO_NOCLOSE);
```

A.5.2.33 function wolfSSL_BIO_set_close

```
int wolfSSL_BIO_set_close(
    WOLFSSL_BIO * b,
    long flag
)
```

BIO が解放されたときに I/O ストリームを閉じる必要があることを示すために使用されるクローズフラグを設定します。

Parameters:

- **bio** WOLFSSL_BIO 構造体。
- **flag** I/O ストリームを閉じる必要があることを示すために使用されるクローズフラグ

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_free

Return: SSL_SUCCESS(1) 成功時に返されます。

Example

```
WOLFSSL_BIO* bio;
// setup bio
wolfSSL_BIO_set_close(bio, BIO_NOCLOSE);
```

A.5.2.34 function wolfSSL_BIO_s_socket

```
WOLFSSL_BIO_METHOD * wolfSSL_BIO_s_socket(
    void
)
```

この関数は BIO_SOCKET タイプの WOLFSSL_BIO_METHOD を取得するために使用されます。

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_s_mem

Return: WOLFSSL_BIO_METHOD ソケットタイプである WOLFSSL_BIO_METHOD 構造体へのポインタ

Example

```
WOLFSSL_BIO* bio;
bio = wolfSSL_BIO_new(wolfSSL_BIO_s_socket);
```

A.5.2.35 function wolfSSL_BIO_set_write_buf_size

```
int wolfSSL_BIO_set_write_buf_size(
    WOLFSSL_BIO * b,
    long size
)
```

この関数は、WOLFSSL_BIO のライトバッファのサイズを設定するために使用されます。書き込みバッファが以前に設定されている場合、この関数はサイズをリセットするときに解放されます。読み書きインデックスを 0 にリセットするという点で、wolfSSL_BIO_reset に似ています。

Parameters:

- **bio** FD を設定するための WOLFSSL_BIO 構造。
- **size** バッファサイズ

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_s_mem
- wolfSSL_BIO_free

Return:

- SSL_SUCCESS 書き込みバッファの設定に成功しました。
- SSL_FAILURE エラーケースに遭遇した場合

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
```

A.5.2.36 function wolfSSL_BIO_make_bio_pair

```
int wolfSSL_BIO_make_bio_pair(
    WOLFSSL_BIO * b1,
    WOLFSSL_BIO * b2
)
```

これは 2 つの BIOS を一緒にペアリングするために使用されます。一対の BIOS は、2 つの方法パイプと同様に、他方で読み取られることができ、その逆も同様である。BIOS の両方が同じスレッド内にあることが予想されます。この機能はスレッドセーフではありません。2 つの BIOS のうちの 1 つを解放すると、両方ともペアになっています。書き込みバッファサイズが以前に設定されていない場合、それはペアになる前に 17000（wolfssl_bio_size）のデフォルトサイズに設定されます。

Parameters:

- **b1** ペアを設定するための第一の WOLFSSL_BIO 構造体へのポインタ。
- **b2** 第二のの WOLFSSL_BIO 構造体へのポインタ。

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_s_mem
- wolfSSL_BIO_free

Return:

- SSL_SUCCESS 2 つの BIOS をうまくペアリングします。
- SSL_FAILURE エラーケースに遭遇した場合

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
```

A.5.2.37 function wolfSSL_BIO_ctrl_reset_read_request

```
int wolfSSL_BIO_ctrl_reset_read_request(
    WOLFSSL_BIO * bio
)
```

この関数は、読み取り要求フラグを 0 に戻すために使用されます。

Parameters:

- **bio** WOLFSSL_BIO 構造体へのポインタ。

See:

- wolfSSL_BIO_new, wolfSSL_BIO_s_mem
- wolfSSL_BIO_new, wolfSSL_BIO_free

Return:

- SSL_SUCCESS 値を正常に設定します。
- SSL_FAILURE エラーケースに遭遇した場合

Example

```
WOLFSSL_BIO* bio;
int ret;
...
ret = wolfSSL_BIO_ctrl_reset_read_request(bio);
// check ret value
```

A.5.2.38 function wolfSSL_BIO_nread0

```
int wolfSSL_BIO_nread0(
    WOLFSSL_BIO * bio,
    char ** buf
)
```

Parameters:

- **bio** WOLFSSL_BIO 構造体へのポインタ。
- **buf** 読み取り用バッファへのポインタのポインタ

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_nwrite0

Return: >=0 成功すると、読み取るバイト数を返します

bri f この関数は、読み取り用のバッファポインタを取得するために使用されます。wolfSSL_BIO_nread とは異なり、内部読み取りインデックスは関数呼び出しから返されたサイズ分進みません。返される値を超えて読み取ると、アレイの境界から読み出される可能性があります。*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
```

A.5.2.39 function wolfSSL_BIO_nread

```
int wolfSSL_BIO_nread(
    WOLFSSL_BIO * bio,
    char ** buf,
    int num
)
```

Parameters:

- **bio** WOLFSSL_BIO 構造体へのポインタ。
- **buf** 読み取り配列の先頭に設定するポインタ。
- **num** 読み取りサイズ

See:

- wolfSSL_BIO_new

- wolfSSL_BIO_nwrite

Return:

- =0 成功すると、読み取るバイト数を返します
- WOLFSSL_BIO_ERROR(-1) Return -1 を読むものではないエラーケースについて

biie れは、この関数は、読み取り用のバッファポインタを取得するために使用されます。内部読み取りインデックスは、読み取り元のバッファの先頭に指されている BUF を使用して、関数呼び出しから返されるサイズ分進みます。数 num で要求された値よりもバイトが少ない場合、より少ない値が返されます。返される値を超えて読み取ると、アレイの境界から読み出される可能性があります。*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
```

A.5.2.40 function wolfSSL_BIO_nwrite

```
int wolfSSL_BIO_nwrite(
    WOLFSSL_BIO * bio,
    char ** buf,
    int num
)
```

関数によって返される数のバイトを書き込むためにバッファーへのポインタを取得します。返されるポインタに追加のバイトを書き込んだ場合、返された値は範囲外の書き込みにつながる可能性があります。

Parameters:

- **bio** WOLFSSL_BIO 構造に書き込む構造。
- **buf** 書き込むためのバッファへのポインタ。
- **num** 書き込みたいサイズ

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_free
- wolfSSL_BIO_nread

Return:

- int 返されたバッファポインタに書き込むことができるバイト数を返します。
- WOLFSSL_BIO_UNSET(-2) バイオペアの一部ではない場合
- WOLFSSL_BIO_ERROR(-1) に書くべき部屋がこれ以上ない場合

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
```

A.5.2.41 function wolfSSL_BIO_reset

```
int wolfSSL_BIO_reset(
    WOLFSSL_BIO * bio
)
```

バイオを初期状態にリセットします。タイプ BIO_BIO の例として、これは読み書きインデックスをリセットします。

Parameters:

- **bio** WOLFSSL_BIO 構造体へのポインタ。

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_free

Return:

- 0 バイオのリセットに成功しました。
- WOLFSSL_BIO_ERROR(-1) 不良入力または失敗したリセットで返されます。

Example

```
WOLFSSL_BIO* bio;
// setup bio
wolfSSL_BIO_reset(bio);
//use pt
```

A.5.2.42 function wolfSSL_BIO_seek

```
int wolfSSL_BIO_seek(
    WOLFSSL_BIO * bio,
    int ofs
)
```

この関数は、指定されたオフセットへファイルポインタを調整します。これはファイルの先頭からのオフセットです。

Parameters:

- **bio** 設定する WOLFSSL_BIO 構造体へのポインタ。
- **ofs** ファイルの先頭からのオフセット

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_s_mem
- wolfSSL_BIO_set_fp
- wolfSSL_BIO_free

Return:

- 0 正常に探しています。
- -1 エラーケースに遭遇した場合

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
```

A.5.2.43 function wolfSSL_BIO_write_filename

```
int wolfSSL_BIO_write_filename(
    WOLFSSL_BIO * bio,
    char * name
)
```

これはファイルに設定および書き込むために使用されます。現在ファイル内のデータを上書きし、BIO が解放されたときにファイルを閉じるように設定されます。

Parameters:

- **bio** ファイルを設定する WOLFSSL_BIO 構造体体。
- **name** 書き込み先ファイル名へのポインタ

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_s_file
- wolfSSL_BIO_set_fp
- wolfSSL_BIO_free

Return:

- SSL_SUCCESS ファイルの開きと設定に成功しました。
- SSL_FAILURE エラーケースに遭遇した場合

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
```

A.5.2.44 function wolfSSL_BIO_set_mem_eof_return

```
long wolfSSL_BIO_set_mem_eof_return(
    WOLFSSL_BIO * bio,
    int v
)
```

これはファイル値の終わりを設定するために使用されます。一般的な値は予想される正の値と混同されないように-1 です。

Parameters:

- **bio** ファイル値の終わりを設定するための WOLFSSL_BIO 構造体体。
- **v** bio にセットする値。

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_s_mem
- wolfSSL_BIO_set_fp
- wolfSSL_BIO_free

Return: 0 完了に戻りました

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
```

A.5.2.45 function wolfSSL_BIO_get_mem_ptr

```
long wolfSSL_BIO_get_mem_ptr(
    WOLFSSL_BIO * bio,
    WOLFSSL_BUF_MEM ** m
)
```

これは WolfSSL_BIO メモリポインタのゲッター関数です。

Parameters:

- **bio** メモリポインタを取得するための WOLFSSL_BIO 構造体へのポインタ。
- **ptr** WOLFSSL_BUF_MEM 構造体へのポインタ（現在は char* となっている）

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_s_mem

Return:

- SSL_SUCCESS ポインタ SSL_SUCCESS を返す正常に（現在 1 の値）。
- SSL_FAILURE null 引数が渡された場合（現在 0 の値）に渡された場合に返されます。

Example

```
WOLFSSL_BIO* bio;
WOLFSSL_BUF_MEM* pt;
// setup bio
wolfSSL_BIO_get_mem_ptr(bio, &pt);
//use pt
```

A.5.2.46 function wolfSSL_lib_version

```
const char * wolfSSL_lib_version(
    void
)
```

この関数は現在のライブラリーバージョンを返します。

See: word32_wolfSSL_lib_version_hex

Return: LIBWOLFSSL_VERSION_STRING バージョンを定義する const char ポインタ。

Example

```
char version[MAXSIZE];
version = wolfSSL_KeepArrays();
…
if(version != ExpectedVersion){
    // Handle the mismatch case
}
```

A.5.2.47 function wolfSSL_lib_version_hex

```
word32 wolfSSL_lib_version_hex(
    void
)
```

この関数は、現在のライブラリーのバージョンを 16 進表記で返します。

See: wolfSSL_lib_version

Return: LILBWOLFSSL_VERSION_HEX wolfssl / version.h で定義されている 16 進数バージョンを返します。

Example

```
word32 libV;
libV = wolfSSL_lib_version_hex();

if(libV != EXPECTED_HEX){
    // How to handle an unexpected value
} else {
    // The expected result for libV
}
```

A.5.2.48 function wolfSSL_negotiate

```
int wolfSSL_negotiate(
    WOLFSSL * ssl
)
```

SSL メソッドの側面に基づいて、実際の接続または承認を実行します。クライアント側から呼び出された場合、サーバ側から呼び出された場合に wolfssl_accept() が実行されている間に wolfssl_connect() が行われる。

See:

- SSL_connect
- SSL_accept

Return:

- SSL_SUCCESS 成功した場合に返されます。に返却されます。(注意、古いバージョンは 0 を返します)
- SSL_FATAL_ERROR 基礎となる呼び出しがエラーになった場合に返されます。特定のエラーコードを取得するには、wolfSSL_get_error() を使用してください。

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);
...
}
...
```

A.5.2.49 function wolfSSL_connect_cert

```
int wolfSSL_connect_cert(
    WOLFSSL * ssl
)
```

この関数はクライアント側で呼び出され、ピアの証明書チェーンを取得するのに十分な長さだけサーバーを持つ SSL / TLS ハンドシェイクを開始します。この関数が呼び出されると、基礎となる通信チャネルはすでに設定されています。wolfssl_connect_cert() は、ブロックと非ブロック I / O の両方で動作します。基礎となる I / O がノンブロッキングである場合、wolfsl_connect_cert() は、wolfssl_connect_cert_cert() のニーズを満たすことができなかったときに戻ります。ハンドシェイクを続けます。この場合、wolfSSL_get_error() への呼び出しは SSL_ERROR_WANT_READ または SSL_ERROR_WANT_WRITE のいずれかを生成します。通話プロセスは、基礎となる I / O が準備ができて、wolfssl がオフになっているところを拾うときに、wolfssl_connect_cert() への呼び出しを繰り返す必要があります。ノンブロッキングソケットを使用する場合は、何も実行する必要がありますが、select() を使用して必要な条件を確認できます。基礎となる入出力がブロックされている場合、wolfssl_connect_cert() はピアの証明書チェーンが受信されたらのみ返されます。

See:

- wolfSSL_get_error
- wolfSSL_connect
- wolfSSL_accept

Return:

- SSL_SUCCESS 成功時に返されます。
- SSL_FAILURE SSL セッションパラメータが NULL の場合、返されます。
- SSL_FATAL_ERROR エラーが発生した場合に返されます。より詳細なエラーコードを取得するには、wolfSSL_get_error() を呼び出します。

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));
}
```

A.5.2.50 function wolfSSL_writev

```
int wolfSSL_writev(
    WOLFSSL * ssl,
    const struct iovec * iov,
    int iovcnt
)
```

Writev Semantics をシミュレートしますが、SSL_Write() の動作のために実際にはブロックしないため、フロント追加が小さくなる可能性があるため Writev を使いやすいソフトウェアに移植する。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ
- **iov** 書き込みへの I / O ベクトルの配列

See: wolfSSL_write

Return:

- 0 成功時に書かれたバイト数。
- 0 失敗したときに返されます。特定のエラーコードについて wolfSSL_get_error() を呼び出します。
- MEMORY_ERROR メモリエラーが発生した場合に返されます。
- SSL_FATAL_ERROR エラーが発生したとき、または非ブロッキングソケットを使用するときには、SSL_ERROR_WANT_READ または SSL_ERROR_WANT_WRITE エラーが受信され、再度 WOLFSSL_WRITE() を呼び出す必要がある場合は、障害が発生します。特定のエラーコードを取得するには、wolfSSL_get_error() を使用してください。

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.
```

A.5.2.51 function wolfSSL_SNI_Status

```
unsigned char wolfSSL_SNI_Status(
    WOLFSSL * ssl,
    unsigned char type
)
```

この関数は SNI オブジェクトのステータスを取得します。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WolfSSL 構造へのポインタ。

See:

- TLSX_SNI_Status
- TLSX_SNI_find
- TLSX_Find

Return:

- value SNI が NULL でない場合、この関数は SNI 構造体のステータスメンバーのバイト値を返します。
- 0 SNI オブジェクトが NULL の場合

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));
...
```

A.5.2.52 function wolfSSL_UseSecureRenegotiation

```
int wolfSSL_UseSecureRenegotiation(
    WOLFSSL * ssl
)
```

この関数は、供給された WOLFSSL 構造の安全な再交渉を強制します。これはお勧めできません。

See:

- TLSX_Find
- TLSX_UseSecureRenegotiation

Return:

- SSL_SUCCESS 安全な再ネゴシエーションを正常に設定します。
- BAD_FUNC_ARG ssl が NULL の場合、エラーを返します。
- MEMORY_E 安全な再交渉のためにメモリを割り当てることができない場合、エラーを返します。

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
}
```

A.5.2.53 function wolfSSL_Rehandshake

```
int wolfSSL_Rehandshake(
    WOLFSSL * ssl
)
```

この関数は安全な再交渉ハンドシェイクを実行します。これは、WolfSSL がこの機能を妨げるように強制されます。

See:

- wolfSSL_negotiate
- wc_InitSha512
- wc_InitSha384
- wc_InitSha256
- wc_InitSha
- wc_InitMd5

Return:

- SSL_SUCCESS 関数がエラーなしで実行された場合に返されます。

- BAD_FUNC_ARG wolfssl 構造が null またはそうでなければ、許容できない引数がサブルーチンに渡された場合に返されます。
- SECURE_RENEGOTIATION_E ハンドシェイクを再ネゴシエーションすることにエラーが発生した場合に返されます。
- SSL_FATAL_ERROR サーバーまたはクライアント構成にエラーが発生した場合は、再ネゴシエーションが完了できなかった場合に返されます。wolfssl_negotiate() を参照してください。

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
...
if(wolfSSL_Rehandshake(ssl) != SSL_SUCCESS){
    // There was an error and the rehandshake is not successful.
}
```

A.5.2.54 function wolfSSL_UseSessionTicket

```
int wolfSSL_UseSessionTicket(
    WOLFSSL * ssl
)
```

セッションチケットを使用するように WolfSSL 構造を強制します。定数 hous_session_ticket を定義し、定数 NO_WOLFSSL_CLIENT をこの関数を使用するように定義しないでください。

See: TLSX_UseSessionTicket

Return:

- SSL_SUCCESS セッションチケットを使用したセットに成功しました。
- BAD_FUNC_ARG ssl が NULL の場合に返されます。
- MEMORY_E セッションチケットを設定するためのメモリの割り当て中にエラーが発生しました。

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
}
```

A.5.2.55 function wolfSSL_get_SessionTicket

```
int wolfSSL_get_SessionTicket(
    WOLFSSL * ssl,
    unsigned char * buf,
    word32 * bufSz
)
```

この機能は、セッション構造のチケットメンバーをバッファにコピーします。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WolfSSL 構造へのポインタ。
- **buf** メモリバッファを表すバイトポインタ。

See:

- wolfSSL_UseSessionTicket
- wolfSSL_set_SessionTicket

Return:

- SSL_SUCCESS 関数がエラーなしで実行された場合に返されます。
- BAD_FUNC_ARG 引数の 1 つが NULL の場合、または bufsz 引数が 0 の場合に返されます。

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
}
```

A.5.2.56 function wolfSSL_set_SessionTicket

```
int wolfSSL_set_SessionTicket(
    WOLFSSL * ssl,
    const unsigned char * buf,
    word32 bufSz
)
```

この関数は、WolfSSL 構造体内の wolfssl_session 構造体のチケットメンバーを設定します。関数に渡されたバッファはメモリにコピーされます。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WolfSSL 構造へのポインタ。
- **buf** セッション構造のチケットメンバーにロードされるバイトポインタ。

See: wolfSSL_set_SessionTicket_cb

Return:

- SSL_SUCCESS 機能の実行に成功したことに戻ります。関数はエラーなしで返されました。
- BAD_FUNC_ARG WolfSSL 構造が NULL の場合に返されます。BUF 引数が NULL の場合は、これはスローされますが、bufsz 引数はゼロではありません。

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.
}
```

A.5.2.57 function wolfSSL_PrintSessionStats

```
int wolfSSL_PrintSessionStats(
    void
)
```

この関数はセッションから統計を印刷します。

See: wolfSSL_get_session_stats

Return:

- SSL_SUCCESS 関数とサブルーチンがエラーなしで戻った場合に返されます。セッション統計は正常に取得され印刷されました。
- BAD_FUNC_ARG サブルーチン wolfssl_get_session_stats() が許容できない引数に渡された場合に返されます。
- BAD_MUTEX_E サブルーチンにミューテックスエラーがあった場合に返されます。

Example

```
// You will need to have a session object to retrieve stats from.
if(wolfSSL_PrintSessionStats(void) != SSL_SUCCESS   ){
    // Did not print session stats
}
```

A.5.2.58 function wolfSSL_get_session_stats

```
int wolfSSL_get_session_stats(
    unsigned int * active,
    unsigned int * total,
    unsigned int * peak,
    unsigned int * maxSessions
)
```

この関数はセッションの統計を取得します。

Parameters:

- **active** 現在のセッションの合計を表す Word32 ポインタ。
- **total** 総セッションを表す Word32 ポインタ。
- **peak** ピークセッションを表す Word32 ポインタ。

See: wolfSSL_PrintSessionStats

Return:

- SSL_SUCCESS 関数とサブルーチンがエラーなしで戻った場合に返されます。セッション統計は正常に取得され印刷されました。
- BAD_FUNC_ARG サブルーチン wolfssl_get_session_stats() が許容できない引数に渡された場合に返されます。
- BAD_MUTEX_E サブルーチンにミューテックスエラーがあった場合に返されます。

Example

```
int wolfSSL_PrintSessionStats(void){
…
ret = wolfSSL_get_session_stats(&totalSessionsNow,
&totalSessionsSeen, &peak, &maxSessions);
…
return ret;
```

A.5.2.59 function wolfSSL_BIO_set_fp

```
long wolfSSL_BIO_set_fp(
    WOLFSSL_BIO * bio,
    XFILE fp,
    int c
)
```

これは BIO の内部ファイルポインタを設定するために使用されます。

Parameters:

- **bio** ペアを設定するための WOLFSSL_BIO 構造体。
- **fp** バイオで設定するファイルポインタ。

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_s_mem
- wolfSSL_BIO_get_fp
- wolfSSL_BIO_free

Return:

- SSL_SUCCESS ファイルポインタを正常に設定します。
- SSL_FAILURE エラーケースに遭遇した場合

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
```

A.5.2.60 function wolfSSL_BIO_ctrl_pending

```
size_t wolfSSL_BIO_ctrl_pending(
    WOLFSSL_BIO * b
)
```

保留中のバイト数を読み取る数を取得します。BIO タイプが BIO_BIO の場合、ペアから読み取る番号です。BIO に SSL オブジェクトが含まれている場合は、SSL オブジェクトからのデータを保留中です（WolfSSL_Pending（SSL））。bio_memory タイプがある場合は、メモリバッファのサイズを返します。

See:

- wolfSSL_BIO_make_bio_pair
- wolfSSL_BIO_new

Return: >=0 保留中のバイト数。

Example

```
WOLFSSL_BIO* bio;
int pending;
bio = wolfSSL_BIO_new();
…
pending = wolfSSL_BIO_ctrl_pending(bio);
```

A.5.2.61 function wolfSSL_set_jobject

```
int wolfSSL_set_jobject(
    WOLFSSL * ssl,
    void * objPtr
)
```

この関数は、WolfSSL 構造の jobjectref メンバーを設定します。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WolfSSL 構造へのポインタ。

See: wolfSSL_get_jobject

Return:

- SSL_SUCCESS jobjectref が objptr に正しく設定されている場合に返されます。
- SSL_FAILURE 関数が正しく実行されず、jobjectref が設定されていない場合に返されます。

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
}
```

A.5.2.62 function wolfSSL_get_jobject

```
void * wolfSSL_get_jobject(
    WOLFSSL * ssl
)
```

この関数は、wolfssl 構造の jobjectref メンバーを返します。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WolfSSL 構造へのポインタ。

See: wolfSSL_set_jobject

Return:

- value wolfssl 構造体が null でない場合、関数は jobjectref 値を返します。
- NULL wolfssl 構造体が NULL の場合に返されます。

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
WOLFSSL* ssl = wolfSSL(ctx);
...
void* jobject = wolfSSL_get_jobject(ssl);

if(jobject != NULL){
    // Success case
}
```

A.5.2.63 function wolfSSL_connect

```
int wolfSSL_connect(
    WOLFSSL * ssl
)
```

この関数はクライアント側で呼び出され、サーバーとの SSL/TLS ハンドシェイクを開始します。この関数が呼び出されるまでに下層の通信チャネルはすでに設定されている必要があります。wolfSSL_connect() は、ブロッキングとノンブロッキング I/O の両方で動作します。下層の I/O がノンブロッキングの場合、wolfSSL_connect() は、下層の I/O が wolfSSL_connect の要求（送信データ、受信データ）を満たすことができなかったときには即戻ります。この場合、wolfSSL_get_error() の呼び出しで SSL_ERROR_WANT_READ または SSL_ERROR_WANT_WRITE のいずれかが返されます。呼び出したプロセスは、下層の I/O がが READY になった時点で、WOLFSSL が停止したときから再開できるように wolfSSL_connect() への呼び出しを繰り返す必要があります。これには select() を使用して必要な条件が整ったかどうかを確認できます。ブロッキング I/O を使用する場合は、ハンドシェークが終了するかエラーが発生するまで戻ってきません。wolfSSL は OpenSSL と比べて証明書検証に異なるアプローチを取ります。クライアントのデフォルトポリシーはサーバーを認証することです。これは、CA 証明書を読み込まない場合、サーバーを確認することができず”-155” のエラーコードが返されます。OpenSSL と同じ振る舞い（つまり、CA 証明書のロードなしでサーバー認証を成功させる）を取らせたい場合には、セキュリティ面でお勧めはしませんが、SSL_CTX_SET_VERIFY（ctx、SSL_VERIFY_NONE、0) を呼び出すことで可能となります。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WolfSSL 構造へのポインタ。

See:

- wolfSSL_get_error
- wolfSSL_accept

Return:

- SSL_SUCCESS 成功した場合に返されます。
- SSL_FATAL_ERROR エラーが発生した場合に返されます。より詳細なエラーコードを取得するには、wolfSSL_get_error() を呼び出します。

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));
}
```

A.5.2.64 function wolfSSL_update_keys

```
int wolfSSL_update_keys(
    WOLFSSL * ssl
)
```

この関数は、TLS v1.3 クライアントまたはサーバーの wolfssl で呼び出されて、キーのロールオーバーを強制します。KeyUpdate メッセージがピアに送信され、新しいキーが暗号化のために計算されます。ピアは KeyUpdate メッセージを送り、新しい復号化キー WIL を計算します。この機能は、ハンドシェイクが完了した後にのみ呼び出すことができます。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ。

See: wolfSSL_write

Return:

- BAD_FUNC_ARG ssl が NULL の場合、または TLS v1.3 を使用していない場合。
- WANT_WRITE 書き込みが準備ができていない場合

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
}
```

A.5.2.65 function wolfSSL_key_update_response

```
int wolfSSL_key_update_response(
    WOLFSSL * ssl,
    int * required
)
```

この関数は、TLS v1.3 クライアントまたはサーバーの wolfssl で呼び出され、キーのロールオーバーが進行中かどうかを判断します。wolfssl_update_keys() が呼び出されると、KeyUpdate メッセージが送信され、暗号化キーが更新されます。復号化キーは、応答が受信されたときに更新されます。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ。
- **キー更新応答が必要ない場合は必須 0。1 キー更新応答が必要ない場合。**

See: wolfSSL_update_keys

Return: 0 成功した。

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
}
```

A.5.2.66 function wolfSSL_request_certificate

```
int wolfSSL_request_certificate(
    WOLFSSL * ssl
)
```

この関数は、TLS v1.3 クライアントからクライアント証明書を要求します。これは、Web サーバーがクライアント認証やその他のものを必要とするページにサービスを提供している場合に役立ちます。接続で最大 256 の要求を送信できます。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ。

See:

- wolfSSL_allow_post_handshake_auth
- wolfSSL_write

Return:

- BAD_FUNC_ARG ssl が NULL の場合、または TLS v1.3 を使用していない場合。
- WANT_WRITE 書き込みが準備ができていない場合
- SIDE_ERROR クライアントで呼び出された場合。
- NOT_READY_ERROR ハンドシェイクが終了していないときに呼び出された場合。
- POST_HAND_AUTH_ERROR 送付後認証が許可されていない場合。
- MEMORY_E 動的メモリ割り当てが失敗した場合

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
}
```

A.5.2.67 function wolfSSL_connect_TLSv13

```
int wolfSSL_connect_TLSv13(
    WOLFSSL * ssl
)
```

この関数はクライアント側で呼び出され、サーバーとの TLS v1.3 ハンドシェイクを開始します。この関数が呼び出されると、下層の通信チャネルはすでに設定されています。wolfSSL_connect()は、ブロックとノンブロック I/O の両方で動作します。下層 I/O がノンブロッキングの場合、wolfSSL_connect() は、下層 I/O が wolfssl_connect の要求を満たすことができなかったときに戻ります。この場合、wolfSSL_get_error() への呼び出しは SSL_ERROR_WANT_READ または SSL_ERROR_WANT_WRITE のいずれかを生成します。通話プロセスは、下層 I/O が READY および WOLFSSL が停止したときに wolfssl_connect() への呼び出しを繰り返す必要があります。ノンブロッキングソケットを使用する場合は、何も実行する必要がありますが、select() を使用して必要な条件を確認できます。基礎となる入出力がブロックされている場合、wolfssl_connect() はハンドシェイクが終了したら、またはエラーが発生したらのみ戻ります。WolfSSL は OpenSSL よりも証明書検証に異なるアプローチを取ります。クライアントのデフォルトポリシーはサーバーを確認することです。これは、CAS を読み込まない場合、サーバーを確認することができ、確認できません（_155）。SSL_CONNECT を持つことの OpenSSL の動作が成功した場合は、サーバーを検証してセキュリティを抑えることができます。SSL_CTX_SET_VERIFY（CTX、SSL_VERIFY_NONE、0）。ssl_new() を呼び出す前に。お勧めできませんが。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ。

See:

- wolfSSL_get_error
- wolfSSL_connect
- wolfSSL_accept_TLSv13
- wolfSSL_accept

Return:

- SSL_SUCCESS 成功時に返されます。
- SSL_FATAL_ERROR エラーが発生した場合に返されます。より詳細なエラーコードを取得するには、wolfSSL_get_error() を呼び出します。

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));
}
```

A.5.2.68 function wolfSSL_accept_TLSv13

```
wolfSSL_accept_TLSv13(
    WOLFSSL * ssl
)
```

この関数はサーバー側で呼び出され、SSL/TLS クライアントが SSL/TLS ハンドシェイクを開始するのを待ちうけます。この関数が呼び出されると、下層の通信チャネルはすでに設定されています。wolfSSL_accept()は、ブロックとノンブロッキング I/O の両方で動作します。下層の入出力がノンブロッキングである場合、wolfSSL_accept() は、下層の I/O が wolfSSL_accept の要求を満たすことができなかったときに戻ります。この場合、wolfSSL_get_error() への呼び出しは SSL_ERROR_WANT_READ または SSL_ERROR_WANT_WRITE のいずれかを生成します。通話プロセスは、読み取り可能なデータが使用可能であり、wolfssl が停止した場所を拾うときに、wolfssl_accept の呼び出しを繰り返す必要があります。ノンブロッキングソケットを使用する場合は、何も実行する必要がありますが、select() を使用して必要な条件を確認できます。下層の I/O がブロックされている場合、wolfssl_accept() はハンドシェイクが終了したら、またはエラーが発生したら戻ります。古いバージョンの ClientHello メッセージがサポートされていますが、TLS v1.3 接続を期待するときにこの関数を呼び出します。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ。

See:

- wolfSSL_get_error
- wolfSSL_connect_TLSv13
- wolfSSL_connect
- wolfSSL_accept_TLSv13
- wolfSSL_accept

Return:

- SSL_SUCCESS 成功時に返されます。

- SSL_FATAL_ERROR エラーが発生した場合に返されます。より詳細なエラーコードを取得するには、wolfSSL_get_error() を呼び出します。

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));
}
```

A.5.2.69 function wolfSSL_write_early_data

```
int wolfSSL_write_early_data(
    WOLFSSL * ssl,
    const void * data,
    int sz,
    int * outSz
)
```

この関数は、セッション再開時にサーバーにアーリーデータを書き込みます。wolfSSL_connect()またはwolfSSL_connect_tlsv13() の代わりにこの関数を呼び出して、サーバーに接続してハンドシェイクにデータを送ります。この機能はクライアントでのみ使用されます。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ。
- **data** アーリーデータを保持しているバッファへのポインタ。
- **sz** 書き込むアーリーデータのサイズ
- **outSz** 書き込んだアーリーデータのサイズ

See:

- wolfSSL_read_early_data
- wolfSSL_connect
- wolfSSL_connect_TLSv13

Return:

- BAD_FUNC_ARG ポインタパラメータが NULL の場合に返されます。sz は 0 未満または TLSV1.3 を使用しない場合にも返されます。
- SIDE_ERROR サーバーで呼び出された場合に返されます。
- WOLFSSL_FATAL_ERROR 接続が行われていない場合に返されます。

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));
}
```

A.5.2.70 function wolfSSL_read_early_data

```
int wolfSSL_read_early_data(
    WOLFSSL * ssl,
    void * data,
    int sz,
    int * outSz
)
```

この関数は、再開時にクライアントからの早期データを読み取ります。wolfssl_accept() または wolfssl_accept_tlsv13() の代わりにこの関数を呼び出して、クライアントを受け入れ、ハンドシェイク内の早期データを読み取ります。ハンドシェイクよりも早期データがない場合は、通常として処理されます。この機能はサーバーでのみ使用されます。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ。
- **データはクライアントから読み込まれた早期データを保持するためのバッファ。**
- **バッファの SZ サイズバイト数。**
- **OUTSZ 初期データのバイト数。**

See:

- wolfSSL_write_early_data
- wolfSSL_accept
- wolfSSL_accept_TLSv13

Return:

- BAD_FUNC_ARG ポインタパラメータが NULL の場合、SZ は 0 未満または TLSV1.3 を使用しない。
- SIDE_ERROR クライアントで呼び出された場合。
- WOLFSSL_FATAL_ERROR 接続を受け入れると失敗した場合

Example

```
int ret = 0;
int err = 0;
WOLFSSL* ssl;
byte earlyData[128];
int outSz;
char buffer[80];
...

ret = wolfSSL_read_early_data(ssl, earlyData, sizeof(earlyData), &outSz);
```

```
if (ret != SSL_SUCCESS) {
    err = wolfSSL_get_error(ssl, ret);
    printf("error = %d, %s\n", err, wolfSSL_ERR_error_string(err, buffer));
}
if (outSz > 0) {
    // early data available
}
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));
}
```

A.5.2.71 function wolfSSL_GetIOReadCtx

```
void * wolfSSL_GetIOReadCtx(
    WOLFSSL * ssl
)
```

この関数は、WolfSSL 構造体の IOCB_READCTX メンバーを返します。

See:

- wolfSSL_GetIOWriteCtx
- wolfSSL_SetIOReadFlags
- wolfSSL_SetIOWriteCtx
- wolfSSL_SetIOReadCtx
- wolfSSL_CTX_SetIOSend

Return:

- pointer この関数は、wolfssl 構造体の iocb_readctx メンバーへの void ポインタを返します。
- NULL wolfssl 構造体が NULL の場合に返されます。*Example*

```
WOLFSSL* ssl = wolfSSL_new(ctx);
void* ioRead;
...
ioRead = wolfSSL_GetIOReadCtx(ssl);
if(ioRead == NULL){
    // Failure case. The ssl object was NULL.
}
```

A.5.2.72 function wolfSSL_GetIOWriteCtx

```
void * wolfSSL_GetIOWriteCtx(
    WOLFSSL * ssl
)
```

この関数は、WolfSSL 構造の IOCB_WRITECTX メンバーを返します。

See:

- wolfSSL_GetIOReadCtx
- wolfSSL_SetIOWriteCtx
- wolfSSL_SetIOReadCtx
- wolfSSL_CTX_SetIOSend

Return:

- pointer この関数は、WolfSSL 構造の IOCB_WRITECTX メンバーへの void ポインタを返します。

- NULL wolfssl 構造体が NULL の場合に返されます。*Example*

```
WOLFSSL* ssl;
void* ioWrite;
...
ioWrite = wolfSSL_GetIOWriteCtx(ssl);
if(ioWrite == NULL){
    // The function returned NULL.
}
```

A.5.2.73 function wolfSSL_SetIO_NetX

```
void wolfSSL_SetIO_NetX(
    WOLFSSL * ssl,
    NX_TCP_SOCKET * nxsocket,
    ULONG waitoption
)
```

この関数は、wolfssl 構造内の nxctx 構造体の NxSocket メンバーと NXWAIT メンバーを設定します。

Parameters:

- **ssl** wolfssl_new（）を使用して作成された WolfSSL 構造へのポインタ。
- **nxSocket** NXCTX 構造の NXSOCTOCK メンバーに設定されている NX_TCP_SOCKET を入力するためのポインタ。*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.
}
```

See:

- set_fd
- NetX_Send
- NetX_Receive

Return: none いいえ返します。

A.6 wolfSSL Context and Session Set Up

A.6.1 Functions

	Name
WOLFSSL_METHOD *	**wolfSSLv23_method**(void) この関数は、wolfSSLv23_client_method と同様に WOLFSSL_METHOD を返します（サーバー/クライアント）。

	Name
WOLFSSL_METHOD *	**wolfSSLv3_server_method**(void)wolfSSLv3_server_method() 関数は、アプリケーションがサーバーであることを示すために使用され、SSL3.0 プロトコルのみをサポートします。この関数は、wolfSSL_CTX_new() を使用して SSL/TLS コンテキストを作成するときに使用される新しい WOLFSSL_METHOD 構造体のメモリを割り当てて初期化します。
WOLFSSL_METHOD *	**wolfSSLv3_client_method**(void)wolfSSLv3_client_method() 関数は、アプリケーションがクライアントであり、SSL 3.0 プロトコルのみをサポートすることを示すために使用されます。この関数は、wolfSSL_CTX_new() を使用して SSL/TLS コンテキストを作成するときに使用される新しい WOLFSSL_METHOD 構造体のメモリを割り当てて初期化します。
WOLFSSL_METHOD *	**wolfTLSv1_server_method**(void)wolfTLSv1_server_method() 関数は、アプリケーションがサーバーであることを示すために使用され、TLS 1.0 プロトコルのみをサポートします。この関数は、wolfSSL_ctx_new() を使用して SSL/TLS コンテキストを作成するときに使用される新しい WOLFSSL_METHOD 構造体のメモリを割り当てて初期化します。
WOLFSSL_METHOD *	**wolfTLSv1_client_method**(void)wolftlsv1_client_method() 関数は、アプリケーションがクライアントであり、TLS 1.0 プロトコルのみをサポートすることを示すために使用されます。この関数は、wolfSSL_ctx_new() を使用して SSL/TLS コンテキストを作成するときに使用される新しい WOLFSSL_METHOD 構造体のメモリを割り当てて初期化します。
WOLFSSL_METHOD *	**wolfTLSv1_1_server_method**(void)wolfTLSv1_1_server_method() 関数は、アプリケーションがサーバーであることを示すために使用され、TLS 1.1 プロトコルのみをサポートします。この関数は、wolfSSL_ctx_new() を使用して SSL/TLS コンテキストを作成するときに使用される新しい WOLFSSL_METHOD 構造体のメモリを割り当てて初期化します。
WOLFSSL_METHOD *	**wolfTLSv1_1_client_method**(void)wolfTLSv1_1_client_method() 関数は、アプリケーションがクライアントであり、TLS 1.0 プロトコルのみをサポートすることを示すために使用されます。この関数は、wolfSSL_ctx_new() を使用して SSL/TLS コンテキストを作成するときに使用される新しい WOLFSSL_METHOD 構造体のメモリを割り当てて初期化します。

	Name
WOLFSSL_METHOD *	**wolfTLSv1_2_server_method**(void)wolfTLSv1_2_server_method() 関数は、アプリケーションがサーバーであることを示すために使用され、TLS 1.2 プロトコルのみをサポートします。この関数は、wolfSSL_ctx_new() を使用して SSL/TLS コンテキストを作成するときに使用される新しい WOLFSSL_METHOD 構造体のメモリを割り当てて初期化します。
WOLFSSL_METHOD *	**wolfTLSv1_2_client_method**(void)wolfTLSv1_2_client_method() 関数は、アプリケーションがクライアントであり、TLS 1.2 プロトコルのみをサポートすることを示すために使用されます。この関数は、wolfSSL_ctx_new() を使用して SSL/TLS コンテキストを作成するときに使用される新しい Wolfssl_method 構造体のメモリを割り当てて初期化します。
WOLFSSL_METHOD *	**wolfDTLSv1_client_method**(void)wolfdtlsv1_client_method() 関数は、アプリケーションがクライアントであり、DTLS 1.0 プロトコルのみをサポートすることを示すために使用されます。この関数は、wolfSSL_ctx_new() を使用して SSL/TLS コンテキストを作成するときに使用される新しい WOLFSSL_METHOD 構造体のメモリを割り当てて初期化します。この関数は、WolfSSL が DTLS サポート（–enable-dtls、または WOLFSSL_DTLS を定義することによって）ビルドされている場合にのみ使用できます。
WOLFSSL_METHOD *	**wolfDTLSv1_server_method**(void)wolfDTLSv1_server_method() 関数は、アプリケーションがサーバーであることを示すために使用され、DTLS 1.0 プロトコルのみをサポートします。この関数は、wolfSSL_ctx_new() を使用して SSL/TLS コンテキストを作成するときに使用される新しい WOLFSSL_METHOD 構造体のメモリを割り当てて初期化します。この関数は、WolfSSL が DTLS サポート（–enable-dtls、または WOLFSSL_DTLS マクロを定義することによって）ビルドされている場合にのみ使用できます。
WOLFSSL_METHOD *	**wolfDTLSv1_3_server_method**(void)wolfDTLSv1_3_server_method() 関数はアプリケーションがサーバーであることを示すために使用され、DTLS 1.3 プロトコルのみをサポートします。この関数は、wolfSSL_ctx_new() を使用して SSL/TLS コンテキストを作成するときに使用される新しい WOLFSSL_METHOD 構造体のメモリを割り当てて初期化します。この関数は、WolfSSL が DTLS サポート（–enable-dtls13、または WOLFSSL_DTLS13 を定義することによって）ビルドされている場合にのみ使用できます。

	Name
WOLFSSL_METHOD *	**wolfDTLSv1_3_client_method**(void)wolfDTLSv1_3_client_method() 関数はアプリケーションがクライアントであることを示すために使用され、DTLS 1.3 プロトコルのみをサポートします。この関数は、wolfSSL_ctx_new() を使用して SSL/TLS コンテキストを作成するときに使用される新しい WOLFSSL_METHOD 構造体のメモリを割り当てて初期化します。この関数は、WolfSSL が DTLS サポート（–enable-dtls13、または WOLFSSL_DTLS13 を定義することによって）ビルドされている場合にのみ使用できます。
WOLFSSL_METHOD *	**wolfDTLS_server_method**(void)wolfDTLS_server_method() 関数はアプリケーションがサーバーであることを示すために使用され、可能な限り高いバージョン最小バージョンの DTLS プロトコルをサポートします。デフォルトの最小バージョンは WOLFSSL_MIN_DTLS_DOWNGRADE マクロでの指定をもとにしていて、実行時に wolfSSL_SetMinVersion() で変更することができます。この関数は、wolfSSL_ctx_new() を使用して SSL/TLS コンテキストを作成するときに使用される新しい WOLFSSL_METHOD 構造体のメモリを割り当てて初期化します。この関数は、WolfSSL が DTLS サポート（–enable-dtls、または WOLFSSL_DTLS を定義することによって）ビルドされている場合にのみ使用できます。
WOLFSSL_METHOD *	**wolfDTLS_client_method**(void)wolfDTLS_client_method() 関数は アプリケーションがクライアントであることを示すために使用され、可能な限り高いバージョン最小バージョンの DTLS プロトコルをサポートします。デフォルトの最小バージョンは WOLFSSL_MIN_DTLS_DOWNGRADE マクロでの指定をもとにしていて、実行時に wolfSSL_SetMinVersion() で変更することができます。この関数は、wolfSSL_ctx_new() を使用して SSL/TLS コンテキストを作成するときに使用される新しい WOLFSSL_METHOD 構造体のメモリを割り当てて初期化します。この関数は、wolfSSL が DTLS サポート（–enable-dtls、または WOLFSSL_DTLS を定義することによって）ビルドされている場合にのみ使用できます。
int	**wolfSSL_use_old_poly**(WOLFSSL * ssl, int value)Chacha-Poly Aead Construction の最初のリリースと新しいバージョンの間にいくつかの違いがあるため、古いバージョンを使用してサーバー/クライアントと通信するオプションを追加しました。デフォルトでは、wolfSSL は新しいバージョンを使用します。

	Name
int	**wolfSSL_CTX_trust_peer_cert**(WOLFSSL_CTX * ctx, const char * file, int type) この関数は、TLS/SSL ハンドシェイクを実行するときにピアを検証するために使用する証明書をロードします。ハンドシェイク中に送信されたピア証明書は、この関数で指定された証明書の SKID と署名を比較することによって検証されます。これら 2 つのことが一致しない場合は、ピア証明書の検証にはロードされた CA 証明書が使用されます。この機能は WOLFSSL_TRUST_PEER_CERT マクロを定義することで機能を有効にできます。適切な使用法は例をご覧ください。
long	**wolfSSL_CTX_get_verify_depth**(WOLFSSL_CTX * ctx) この関数は、WOLFSSL_CTX 構造体構造を使用して証明書チェーン深度を取得します。
WOLFSSL_CTX *	**wolfSSL_CTX_new**(WOLFSSL_METHOD *) この関数は、所望の SSL/TLS プロトコル用メソッド構造体を引数に取って、新しい SSL コンテキストを作成します。
WOLFSSL *	**wolfSSL_new**(WOLFSSL_CTX *) この関数はすでに作成された SSL コンテキスト (WOLFSSL_CTX) を入力として、新しい SSL セッション (WOLFSSL) を作成します。
int	**wolfSSL_set_fd**(WOLFSSL * ssl, int fd) この関数は、SSL 接続の入出力機能としてファイル記述子 (fd) を割り当てます。通常これはソケットファイル記述子になります。
int	**wolfSSL_set_dtls_fd_connected**(WOLFSSL * ssl, int fd) この関数はファイルディスクリプタ (fd) を SSL コネクションの入出力手段として設定します。通常はソケットファイルディスクリプタが指定されます。この関数は DTLS 専用の API であり、ソケットは接続済みとマークされます。したがって、与えられた fd に対する recvfrom と sendto 呼び出しでの addr と addr_len は NULL に設定されます。
int	**wolfDTLS_SetChGoodCb**(WOLFSSL * ssl, ClientHelloGoodCb cb, void * user_ctx) この関数は DTLS ClientHello メッセージが正しく処理できた際に呼び出されるコールバック関数を設定します。クッキー交換メカニズムを使用する場合 (DTLS1.2 の HelloVerifyRequest か DTLS1.3 のクッキー拡張を伴った HelloRetryRequest のいずれかを使用する場合) には、クッキー交換が成功した時点でこのコールバック関数が呼び出されます。この機能はひとつの WOLFSSL オブジェクトを新たな接続を待ち受けるリスナーとして使い, ClientHello が検証された WOLFSSL オブジェクトから絶縁させることができます。この場合の検証はクッキー交換か ClientHello が正しいフォーマットになっているかのチェックによってなされます。

	Name
void	**wolfSSL_set_using_nonblock**(WOLFSSL * ssl, int nonblock) この関数は、WOLFSSL オブジェクトに基礎となる I/O がノンブロックであることを通知します。アプリケーションが WOLFSSL オブジェクトを作成した後、ブロッキング以外のソケットで使用する場合は、wolfssl_set_using_nonblock() を呼び出します。これにより、wolfssl オブジェクトは、EWOULDBLOCK を受信することを意味します。
void	**wolfSSL_CTX_free**(WOLFSSL_CTX *) この関数は、割り当てられた WOLFSSL_CTX オブジェクトを解放します。この関数は CTX 参照数を減らし、参照カウントが 0 に達したときにのみコンテキストを解放します。
void	**wolfSSL_free**(WOLFSSL *) この関数は割り当てられた WOLFSSL オブジェクトを解放します。
int	**wolfSSL_set_session**(WOLFSSL * ssl, WOLFSSL_SESSION * session) この関数は、SSL オブジェクト SSL が SSL/TLS 接続を確立する目的で使用するセッションを設定します。セッション再開を行う場合、wolfSSL_shutdown() を呼び出す前に wolfSSL_get1_session() を呼び出してセッションオブジェクトを取得し、セッション ID を保存しておく必要があります。後で、アプリケーションは新しい WOLFSSL オブジェクトを作成し、保存したセッションを wolfSSL_set_session() に渡す必要があります。その後アプリケーションは wolfSSL_connect() を呼び出し、wolfSSL はセッション再開を試みます。wolfSSL サーバーコードでは、デフォルトでセッション再開を許可します。wolfSSL_get1_session() によって返されたオブジェクトは、アプリケーションが使用後に解放する必要があります。

	Name
void	**wolfSSL_CTX_set_verify**(WOLFSSL_CTX * ctx, int mode, VerifyCallback verify_callback) この関数はリモートピアの検証方法を設定し、また証明書検証コールバック関数を SSL コンテキストに登録することもできます。検証コールバックは、検証障害が発生した場合にのみ呼び出されます。検証コールバックが必要な場合は、NULL ポインタを verify_callback に使用できます。ピア証明書の検証モードは、論理的またはフラグのリストです。可能なフラグ値は次のとおりです: SSL_VERIFY_NONE -クライアントモード：クライアントはサーバーから受信した証明書を検証せず、ハンドシェイクは通常どおり続きます。-サーバーモード：サーバーはクライアントに証明書要求を送信しません。そのため、クライアント検証は有効になりません。SSL_VERIFY_PEER -クライアントモード：クライアントはハンドシェイク中にサーバーから受信した証明書を検証します。これは wolfSSL ではデフォルトでオンにされます。したがって、このオプションを使用すると効果がありません。-サーバーモード：サーバーは証明書要求をクライアントに送信し、受信したクライアント証明書を確認します。 SSL_VERIFY_FAIL_IF_NO_PEER_CERT -クライアントモード：クライアント側で使用されていない場合は効果がありません。-サーバーモード：要求されたときにクライアントが証明書の送信に失敗した場合は、サーバー側で検証が失敗します（SSL サーバーの SSL_VERIFY_PEER を使用する場合）。 SSL_VERIFY_FAIL_EXCEPT_PSK -クライアントモード：クライアント側で使用されていない場合は効果がありません。-サーバーモード：PSK 接続の場合を除き、検証は SSL_VERIFY_FAIL_IF_NO_PEER_CERT と同じです。PSK 接続が行われている場合、接続はピア証明書なしで通過します。

	Name
void	**wolfSSL_set_verify**(WOLFSSL * ssl, int mode, VerifyCallback verify_callback) この関数はリモートピアの検証方法を設定し、また証明書検証コールバック関数を WOLFSSL オブジェクトに登録することもできます。検証コールバックは、検証障害が発生した場合にのみ呼び出されます。検証コールバックが必要な場合は、NULL ポインタを verify_callback に使用できます。ピア証明書の検証モードは、論理的またはフラグのリストです。可能なフラグ値は次のとおりです: SSL_VERIFY_NONE -クライアントモード：クライアントはサーバーから受信した証明書を検証せず、ハンドシェイクは通常どおり続きます。-サーバーモード：サーバーはクライアントに証明書要求を送信しません。そのため、クライアント検証は有効になりません。SSL_VERIFY_PEER -クライアントモード：クライアントはハンドシェイク中にサーバーから受信した証明書を検証します。これは wolfSSL ではデフォルトでオンにされます。したがって、このオプションを使用すると効果がありません。-サーバーモード：サーバーは証明書要求をクライアントに送信し、受信したクライアント証明書を確認します。 SSL_VERIFY_FAIL_IF_NO_PEER_CERT -クライアントモード：クライアント側で使用されていない場合は効果がありません。-サーバーモード：要求されたときにクライアントが証明書の送信に失敗した場合は、サーバー側で検証が失敗します（SSL サーバーの SSL_VERIFY_PEER を使用する場合）。 SSL_VERIFY_FAIL_EXCEPT_PSK -クライアントモード：クライアント側で使用されていない場合は効果がありません。-サーバーモード：PSK 接続の場合を除き、検証は SSL_VERIFY_FAIL_IF_NO_PEER_CERT と同じです。PSK 接続が行われている場合、接続はピア証明書なしで通過します。
long	**wolfSSL_CTX_set_session_cache_mode**(WOLFSSL_CTX * ctx, long mode) この関数は SSL セッションキャッシュ機能を有効または無効にします。動作はモードに使用される値によって異なります。モードの値は次のとおりです：SSL_SESS_CACHE_OFF - セッションキャッシングを無効にします。デフォルトでセッションキャッシングがオンになっています。SSL_SESS_CACHE_NO_AUTO_CLEAR - セッションキャッシュのオートフラッシュを無効にします。デフォルトで自動フラッシングはオンになっています。
int	**wolfSSL_CTX_memrestore_cert_cache**(WOLFSSL_CTX * ctx, const void * mem, int sz) この関数は証明書キャッシュをメモリから復元します。

	Name
int	**wolfSSL_CTX_set_cipher_list**(WOLFSSL_CTX * ctx, const char * list) この関数は、与えられた WOLFSSL_CTX に暗号スイートリストを設定します。この暗号スイートリストは、このコンテキストを使用して作成された新しい SSL セッション（WolfSSL）のデフォルトリストになります。リスト内の暗号は、優先度の高いものの順に順にソートされるべきです。wolfSSL_CTX_set_cipher_list() が呼び出される都度、特定の SSL コンテキストの暗号スイートリストを提供されたリストにリセットします。暗号スイートリストはヌル終端されたコロン区切りリストです。たとえば、リストの値が「DHE-RSA-AES256-SHA256：DHE-RSA-AES128-SHA256：AES256-SHA256」有効な暗号値は、src/internal.c の cipher_names [] 配列のフルネーム値です。（有効な暗号化値の明確なリストの場合は src/internal.c をチェックしてください）
int	**wolfSSL_set_cipher_listが呼び出される都度、特定の SSL コンテキストの暗号スイートリストを提供されたリストにリセットします。暗号スイートリストはヌル終端されたコロン区切りリストです。たとえば、リストの値が「DHE-RSA-AES256-SHA256：DHE-RSA-AES128-SHA256：AES256-SHA256」有効な暗号値は、src/internal.c の cipher_names [] 配列のフルネーム値です。（有効な暗号化値の明確なリストの場合は src/internal.c をチェックしてください）
void	**wolfSSL_dtls13_set_send_more_acks**(WOLFSSL * ssl, int value) この関数は、ライブラリが中断を検出したときにすぐに他のピアに ACK を送信するかどうかを設定します。ACK をすぐに送信すると、遅延は最小限に抑えられますが、必要以上に多くの帯域幅が消費される可能性があります。アプリケーションが独自にタイマーを管理しており、このオプションが 0 に設定されている場合、アプリケーションコードは wolfSSL_dtls13_use_quick_timeout() を使用して、遅延した ACK を送信するためにより速いタイムアウトを設定する必要があるかどうかを判断できます。
int	**wolfSSL_dtls_set_timeout_init**(WOLFSSL * ssl, int) この関数は DTLS タイムアウトを設定します。

	Name
WOLFSSL_SESSION *	**wolfSSL_get1_session**(WOLFSSL * ssl) この関数は、WOLFSSL 構造体から WOLFSSL_SESSION を参照型として返します。これには、wolfSSL_SESSION_free を呼び出してセッション参照を解除する必要があります。WOLFSSL_SESSION は、セッションの再開を実行するために必要なすべての必要な情報を含み、新しいハンドシェイクなしで接続を再確立します。セッションの再開の場合、wolfSSL_shutdown() をセッションオブジェクトに呼び出す前に、アプリケーションはオブジェクトから wolfssl_get1_session() を呼び出して保存する必要があります。これはセッションへのポインタを返します。その後、アプリケーションは新しい WOLFSSL オブジェクトを作成し、保存したセッションを wolfssl_set_session() に割り当てる必要があります。この時点で、アプリケーションは wolfssl_connect() を呼び出し、WolfSSL はセッションを再開しようとします。WolfSSL サーバーコードでは、デフォルトでセッションの再開を許可します。wolfssl_get1_session() によって返されたオブジェクトは、アプリケーションが使用後は解放される必要があります。
WOLFSSL_METHOD *	**wolfSSLv23_client_method**(void)wolfsslv23_client_method() 関数は、アプリケーションがクライアントであることを示すために使用され、SSL 3.0~TLS 1.3 の間でサーバーでサポートされている最高のプロトコルバージョンをサポートします。この関数は、wolfSSL_CTX_new() を使用して SSL / TLS コンテキストを作成するときに使用される新しい Wolfssl_method 構造体のメモリを割り当てて初期化します。WolfSSL クライアントとサーバーの両方が堅牢なバージョンのダウングレード機能を持っています。特定のプロトコルバージョンメソッドがどちらの側で使用されている場合は、そのバージョンのみがネゴシエートされたり、エラーが返されます。たとえば、TLSV1 を使用し、SSLv3 のみに接続しようとするクライアントは、TLSV1.1 に接続しても失敗します。この問題を解決するために、wolfsslv23_client_method() 関数を使用するクライアントは、サーバーでサポートされている最高のプロトコルバージョンを使用し、必要に応じて SSLv3 にダウングレードします。この場合、クライアントは SSLv3 - TLSv1.3 を実行しているサーバーに接続できるようになります。
WOLFSSL_BIGNUM *	**wolfSSL_ASN1_INTEGER_to_BN**(const WOLFSSL_ASN1_INTEGER * ai, WOLFSSL_BIGNUM * bn) この関数は、WOLFSSL_ASN1_INTEGER 値を WOLFSSL_BIGNUM 構造体にコピーするために使用されます。

	Name
long	**wolfSSL_CTX_add_extra_chain_cert**(WOLFSSL_CTX * ctx, WOLFSSL_X509 * x509) この関数は、WOLFSSL_CTX 構造で構築されている内部チェーンに証明書を追加します。
int	**wolfSSL_CTX_get_read_ahead**(WOLFSSL_CTX * ctx) この関数は、WOLFSSL_CTX 構造から Get Read Hape フラグを返します。
int	**wolfSSL_CTX_set_read_ahead**(WOLFSSL_CTX * ctx, int v) この関数は、WOLFSSL_CTX 構造内の読み出し先のフラグを設定します。
long	**wolfSSL_CTX_set_tlsext_status_arg**(WOLFSSL_CTX * ctx, void * arg) この関数は OCSP で使用するオプション引数を設定します。
long	**wolfSSL_CTX_set_tlsext_opaque_prf_input_callback_arg**(WOLF * ctx, void * arg) この関数は、PRF コールバックに渡すオプションの引数を設定します。
long	**wolfSSL_set_options**(WOLFSSL * s, long op) この関数は、SSL のオプションマスクを設定します。いくつかの有効なオプションは、ssl_op_all、ssl_op_cookie_exchange、ssl_op_no_sslv2、ssl_op_no_sslv3、ssl_op_no_tlsv1_1、ssl_op_no_tlsv1_2、ssl_op_no_compression です。
long	**wolfSSL_get_options**(const WOLFSSL * ssl) この関数は現在のオプションマスクを返します。
long	**wolfSSL_set_tlsext_debug_arg**(WOLFSSL * ssl, void * arg) この関数は、渡されたデバッグ引数を設定するために使用されます。
long	**wolfSSL_get_verify_result**(const WOLFSSL * ssl)
int	**wolfSSL_CTX_allow_anon_cipher**(WOLFSSL_CTX *) この機能により、CTX 構造の HAVAnon メンバーがコンパイル中に定義されている場合は、CTX 構造の HABANON メンバーを有効にします。
WOLFSSL_METHOD *	**wolfSSLv23_server_method**(void)wolfsslv23_server_method() 関数は、アプリケーションがサーバーであることを示すために使用され、SSL 3.0 _ TLS 1.3 からプロトコルバージョンと接続するクライアントをサポートします。この関数は、wolfSSL_CTX_new() を使用して SSL / TLS コンテキストを作成するときに使用される新しい Wolfssl_method 構造体のメモリを割り当てて初期化します。
int	**wolfSSL_state**(WOLFSSL * ssl)
int	**wolfSSL_check_domain_name**(WOLFSSL * ssl, const char * dn)wolfssl デフォルトでは、有効な日付範囲と検証済みの署名のためにピア証明書をチェックします。wolfssl_connect() または wolfssl_accept() の前にこの関数を呼び出すと、実行するチェックのリストにドメイン名チェックが追加されます。DN 受信時にピア証明書を確認するためのドメイン名を保持します。

	Name
int	**wolfSSL_set_compression**(WOLFSSL * ssl)SSL 接続に圧縮を使用する機能をオンにします。両側には圧縮がオンになっている必要があります。そうでなければ圧縮は使用されません。ZLIB ライブラリは実際のデータ圧縮を実行します。ライブラリにコンパイルするには、システムの設定システムに–with-libz を使用し、そうでない場合は hand_libz を定義します。送受信されるメッセージの実際のサイズを減らす前にデータを圧縮している間に、圧縮によって保存されたデータの量は通常、ネットワークの遅いすべてのネットワークを除いたものよりも分析に時間がかかります。
int	**wolfSSL_set_timeout**(WOLFSSL * ssl, unsigned int to) この関数は SSL セッションタイムアウト値を秒単位で設定します。
int	**wolfSSL_CTX_set_timeout**(WOLFSSL_CTX * ctx, unsigned int to) この関数は、指定された SSL コンテキストに対して、SSL セッションのタイムアウト値を秒単位で設定します。
int	**wolfSSL_CTX_UnloadCAs**(WOLFSSL_CTX *) この関数は CA 署名者リストをアンロードし、署名者全体のテーブルを解放します。
int	**wolfSSL_CTX_Unload_trust_peers**(WOLFSSL_CTX *) この関数は、以前にロードされたすべての信頼できるピア証明書をアンロードするために使用されます。マクロ wolfssl_trust_peer_cert を定義することで機能が有効になっています。
int	**wolfSSL_CTX_trust_peer_buffer**(WOLFSSL_CTX * ctx, const unsigned char * in, long sz, int format) この関数は、TLS / SSL ハンドシェイクを実行するときにピアを検証するために使用する証明書をロードします。ハンドシェイク中に送信されたピア証明書は、使用可能なときにスキッドを使用することによって比較されます。これら 2 つのことが一致しない場合は、ロードされた CAS が使用されます。ファイルの代わりにバッファーの場合は、wolfssl_ctx_trust_peer_cert と同じ機能です。特徴はマクロ wolfssl_trust_peer_cert を定義することによって有効になっています適切な使用法の例を参照してください。
int	**wolfSSL_CTX_set_group_messages**(WOLFSSL_CTX *) この機能は、可能な限りハンドシェイクメッセージのグループ化をオンにします。
int	**wolfSSL_set_group_messages**(WOLFSSL *) この機能は、可能な限りハンドシェイクメッセージのグループ化をオンにします。
int	**wolfSSL_CTX_SetMinVersion**(WOLFSSL_CTX * ctx, int version) この関数は、許可されている最小のダウングレードバージョンを設定します。接続が（wolfsslv23_client_method または wolfsslv23_server_method）を使用して、接続がダウングレードできる場合にのみ適用されます。

	Name
int	**wolfSSL_SetVersion**(WOLFSSL * ssl, int version) この関数は、バージョンで指定されたバージョンを使用して、指定された SSL セッション（WolfSSL オブジェクト）の SSL/TLS プロトコルバージョンを設定します。これにより、SSL セッション（SSL）のプロトコル設定が最初に定義され、SSL コンテキスト（wolfSSL_CTX_new()）メソッドの種類によって上書きされます。
int	**wolfSSL_UseALPN**(WOLFSSL * ssl, char * protocol_name_list, unsigned int protocol_name_listSz, unsigned char options)wolfssl セッションに ALPN を設定します。
int	**wolfSSL_CTX_UseSessionTicket**(WOLFSSL_CTX * ctx) この関数は、セッションチケットを使用するように WolfSSL コンテキストを設定します。
int	**wolfSSL_check_private_key**(const WOLFSSL * ssl) この関数は、秘密鍵が使用されている証明書との一致であることを確認します。
int	**wolfSSL_use_certificate**(WOLFSSL * ssl, WOLFSSL_X509 * x509) ハンドシェイク中に使用するために、WolfSSL 構造の証明書を設定するために使用されます。
int	**wolfSSL_use_certificate_ASN1**(WOLFSSL * ssl, unsigned char * der, int derSz)
int	**wolfSSL_SESSION_get_master_key**(const WOLFSSL_SESSION * ses, unsigned char * out, int outSz) これはハンドシェイクを完了した後にマスターキーを取得するために使用されます。
int	**wolfSSL_SESSION_get_master_key_length**(const WOLFSSL_SESSION * ses) これはマスター秘密鍵の長さを取得するために使用されます。
void	**wolfSSL_CTX_set_cert_store**(WOLFSSL_CTX * ctx, WOLFSSL_X509_STORE * str)
WOLFSSL_X509_STORE *	**wolfSSL_CTX_get_cert_store**(WOLFSSL_CTX * ctx)
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) これは CTX で設定されたパスワードコールバックのゲッター関数です。
void *	**wolfSSL_CTX_get_default_passwd_cb_userdata**(WOLFSSL_CTX * ctx)
long	**wolfSSL_CTX_clear_options**(WOLFSSL_CTX * ctx, long opt) この関数は、WOLFSSL_CTX オブジェクトのオプションビットをリセットします。

	Name
int	**wolfSSL_set_msg_callback**(WOLFSSL * ssl, SSL_Msg_Cb cb) この関数は SSL 内のコールバックを設定します。コールバックはハンドシェイクメッセージを観察することです。CB の NULL 値はコールバックをリセットします。
int	**wolfSSL_set_msg_callback_arg**(WOLFSSL * ssl, void * arg) この関数は、SSL 内の関連コールバックコンテキスト値を設定します。値はコールバック引数に渡されます。
int	**wolfSSL_send_hrr_cookie**(WOLFSSL * ssl, const unsigned char * secret, unsigned int secretSz) この関数はサーバー側で呼び出されて、HellORetryRequest メッセージに Cookie を含める必要があることを示します。Cookie は現在のトランスクリプトのハッシュを保持しているので、別のサーバープロセスは応答で ClientHello を処理できます。秘密は Cookie データの整合性チェックを Genertingするときに使用されます。
int	**wolfSSL_disable_hrr_cookie**(WOLFSSL * ssl) この関数はサーバー側で呼び出され、HelloRetryRequest メッセージがクッキーを含んではならないこと、DTLSv1.3 が使用されている場合にはクッキーの交換がハンドシェークに含まれないことを表明します。DTLSv1.3 ではクッキー交換を行わないとサーバーが DoS/Amplification 攻撃を受けやすくなる可能性があることに留意してください。
int	**wolfSSL_CTX_no_ticket_TLSv13**(WOLFSSL_CTX * ctx) この関数はサーバー上で呼び出され、ハンドシェイク完了時にセッション再開のためのセッションチケットの送信を行わないようにします。
int	**wolfSSL_no_ticket_TLSv13**(WOLFSSL * ssl) ハンドシェイクが完了すると、この関数はサーバー上で再開セッションチケットの送信を停止するように呼び出されます。
int	**wolfSSL_CTX_no_dhe_psk**(WOLFSSL_CTX * ctx) この関数は、Authentication にプリシェアキーを使用している場合、DIFFIE-HELLMAN（DH）スタイルのキー交換を許可する TLS V1.3 WolfSSL コンテキストで呼び出されます。
int	**wolfSSL_no_dhe_psk**(WOLFSSL * ssl) この関数は、事前共有鍵を使用している TLS V1.3 クライアントまたはサーバーで、に Diffie-Hellman（DH）スタイルの鍵交換を許可しないように設定します。
int	**wolfSSL_CTX_allow_post_handshake_auth**(WOLFSSL_CTX * ctx) この関数は、TLS v1.3 クライアントの WolfSSL コンテキストで呼び出され、クライアントはサーバーからの要求に応じて Post Handshake を送信できるようにします。これは、クライアント認証などを必要としないページを持つ Web サーバーに接続するときに役立ちます。

	Name
int	**wolfSSL_allow_post_handshake_auth**(WOLFSSL * ssl) この関数は、TLS V1.3 クライアント WolfSSL で呼び出され、クライアントはサーバーからの要求に応じてハンドシェイクを送ります。handshake クライアント認証拡張機能は ClientHello で送信されます。これは、クライアント認証などを必要としないページを持つ Web サーバーに接続するときに役立ちます。
int	**wolfSSL_CTX_set1_groups_list**(WOLFSSL_CTX * ctx, char * list) この関数は楕円曲線グループのリストを設定して、WolfSSL コンテキストを希望の順に設定します。リストはヌル終了したテキスト文字列、およびコロン区切りリストです。この関数を呼び出して、TLS v1.3 接続で使用する鍵交換楕円曲線パラメータを設定します。
int	**wolfSSL_set1_groups_list**(WOLFSSL * ssl, char * list) この関数は楕円曲線グループのリストを設定して、WolfSSL を希望の順に設定します。リストはヌル終了したテキスト文字列、およびコロン区切りリストです。この関数を呼び出して、TLS v1.3 接続で使用する鍵交換楕円曲線パラメータを設定します。
int	**wolfSSL_CTX_set_groups**(WOLFSSL_CTX * ctx, int * groups, int count) この関数は楕円曲線グループのリストを設定して、WolfSSL コンテキストを希望の順に設定します。リストは、Count で指定された識別子の数を持つグループ識別子の配列です。この関数を呼び出して、TLS v1.3 接続で使用する鍵交換楕円曲線パラメータを設定します。
int	**wolfSSL_set_groups**(WOLFSSL * ssl, int * groups, int count) この関数は、wolfssl を許すために楕円曲線グループのリストを設定します。リストは、Count で指定された識別子の数を持つグループ識別子の配列です。この関数を呼び出して、TLS v1.3 接続で使用する鍵交換楕円曲線パラメータを設定します。
int	**wolfSSL_CTX_set_max_early_data**(WOLFSSL_CTX * ctx, unsigned int sz) この関数は、WolfSSL コンテキストを使用して TLS V1.3 サーバーによって受け入れられるアーリーデータの最大量を設定します。この関数を呼び出して、再生攻撃を軽減するためのプロセスへのアーリーデータの量を制限します。初期のデータは、セッションチケットが送信されたこと、したがってセッションチケットが再開されるたびに同じ接続の鍵から派生した鍵によって保護されます。値は再開のためにセッションチケットに含まれています。ゼロの値は、セッションチケットを使用してクライアントによってアーリーデータを送信することを示します。アーリーデータバイト数をアプリケーションで実際には可能な限り低く保つことをお勧めします。

	Name
int	**wolfSSL_set_max_early_data**(WOLFSSL * ssl, unsigned int sz) この関数は、WolfSSL コンテキストを使用して TLS V1.3 サーバーによって受け入れられるアーリーデータの最大量を設定します。この関数を呼び出して、再生攻撃を軽減するためプロセスへのアーリーデータの量を制限します。初期のデータは、セッションチケットが送信されたこと、したがってセッションチケットが再開されるたびに同じ接続の鍵から派生した鍵によって保護されます。値は再開のためにセッションチケットに含まれています。ゼロの値は、セッションチケットを使用してクライアントによってアーリーデータを送信することを示します。アーリーデータバイト数をアプリケーションで実際には可能な限り低く保つことをお勧めします。
void	**wolfSSL_CTX_set_psk_client_tls13_callback**(WOLFSSL_CTX * ctx, wc_psk_client_tls13_callback cb) この関数は、TLS v1.3 接続のプレシェア鍵（PSK）クライアント側コールバックを設定します。コールバックは PSK アイデンティティを見つけ、そのキーと、ハンドシェイクに使用する暗号の名前を返します。この関数は、WOLFSSL_CTX 構造体の client_psk_tls13_cb メンバーを設定します。
void	**wolfSSL_set_psk_client_tls13_callback**(WOLFSSL * ssl, wc_psk_client_tls13_callback cb) この関数は、TLS v1.3 接続のプレシェアキー（PSK）クライアント側コールバックを設定します。コールバックは PSK アイデンティティを見つけ、そのキーと、ハンドシェイクに使用する暗号の名前を返します。この関数は、wolfssl 構造体の Options フィールドの client_psk_tls13_cb メンバーを設定します。
void	**wolfSSL_CTX_set_psk_server_tls13_callback**(WOLFSSL_CTX * ctx, wc_psk_server_tls13_callback cb) この関数は、TLS v1.3 接続用の事前共有鍵（PSK）サーバ側コールバックを設定します。コールバックは PSK アイデンティティを見つけ、そのキーと、ハンドシェイクに使用する暗号の名前を返します。この関数は、wolfssl_ctx 構造体の server_psk_tls13_cb メンバーを設定します。
void	**wolfSSL_set_psk_server_tls13_callback**(WOLFSSL * ssl, wc_psk_server_tls13_callback cb) この関数は、TLS v1.3 接続用の事前共有鍵（PSK）サーバ側コールバックを設定します。コールバックは PSK アイデンティティを見つけ、そのキーと、ハンドシェイクに使用する暗号の名前を返します。この関数は、wolfssl 構造体のオプションフィールドの server_psk_tls13_cb メンバーを設定します。

	Name
int	**wolfSSL_UseKeyShare**(WOLFSSL * ssl, word16 group) この関数は、キーペアの生成を含むグループからキーシェアエントリを作成します。Keyshare エクステンションには、鍵交換のための生成されたすべての公開鍵が含まれています。この関数が呼び出されると、指定されたグループのみが含まれます。優先グループがサーバーに対して以前に確立されているときにこの関数を呼び出します。
int	**wolfSSL_NoKeyShares**(WOLFSSL * ssl) この関数は、ClientHello で鍵共有が送信されないように呼び出されます。これにより、ハンドシェイクに鍵交換が必要な場合は、サーバーが HelloretryRequest で応答するように強制します。予想される鍵交換グループが知られておらず、キーの生成を不必要に回避するときにこの機能を呼び出します。鍵交換が必要なときにハンドシェイクを完了するために追加の往復が必要になることに注意してください。
WOLFSSL_METHOD *	**wolfTLSv1_3_server_method_ex**(void * heap) この関数は、アプリケーションがサーバーであることを示すために使用され、TLS 1.3 プロトコルのみをサポートします。この関数は、wolfSSL_CTX_new() を使用して SSL / TLS コンテキストを作成するときに使用される新しい Wolfssl_method 構造体のメモリを割り当てて初期化します。
WOLFSSL_METHOD *	**wolfTLSv1_3_client_method_ex**(void * heap) この関数は、アプリケーションがクライアントであることを示すために使用され、TLS 1.3 プロトコルのみをサポートします。この関数は、wolfSSL_CTX_new() を使用して SSL / TLS コンテキストを作成するときに使用される新しい Wolfssl_method 構造体のメモリを割り当てて初期化します。
WOLFSSL_METHOD *	**wolfTLSv1_3_server_method**(void) この関数は、アプリケーションがサーバーであることを示すために使用され、TLS 1.3 プロトコルのみをサポートします。この関数は、wolfSSL_CTX_new() を使用して SSL / TLS コンテキストを作成するときに使用される新しい Wolfssl_method 構造体のメモリを割り当てて初期化します。
WOLFSSL_METHOD *	**wolfTLSv1_3_client_method**(void) この関数は、アプリケーションがクライアントであることを示すために使用され、TLS 1.3 プロトコルのみをサポートします。この関数は、wolfSSL_CTX_new() を使用して SSL / TLS コンテキストを作成するときに使用される新しい Wolfssl_method 構造体のメモリを割り当てて初期化します。

	Name
WOLFSSL_METHOD *	**wolfTLSv1_3_method_ex**(void * heap) この関数は、まだどちらの側（サーバ/クライアント）を決定していないことを除いて、Wolftlsv1_3_client_method と同様の wolfssl_method を返します。
WOLFSSL_METHOD *	**wolfTLSv1_3_method**(void) この関数は、まだどちらの側（サーバ/クライアント）を決定していないことを除いて、Wolftlsv1_3_client_method と同様の wolfssl_method を返します。
int	**wolfSSL_CTX_set_client_cert_type**(WOLFSSL_CTX * ctx, const char * buf, int len) この関数はクライアント側で呼び出される場合には、サーバー側に Certificate メッセージで送信できる証明書タイプを設定します。サーバー側で呼び出される場合には、受入れ可能なクライアント証明書タイプを設定します。Raw Public Key 証明書を送受信したい場合にはこの関数を使って証明書タイプを設定しなければなりません。設定する証明書タイプは優先度順に格納したバイト配列として渡します。設定するバッファアドレスに NULL を渡すか、あるいはバッファサイズに 0 を渡すと規定値にもどすことができます。規定値は X509 証明書（WOLFSSL_CERT_TYPE_X509）のみを扱う設定となっています。
int	**wolfSSL_CTX_set_server_cert_type**(WOLFSSL_CTX * ctx, const char * buf, int len) この関数はサーバー側で呼び出される場合には、クライアント側に Certificate メッセージで送信できる証明書タイプを設定します。クライアント側で呼び出される場合には、受入れ可能なサーバー証明書タイプを設定します。Raw Public Key 証明書を送受信したい場合にはこの関数を使って証明書タイプを設定しなければなりません。設定する証明書タイプは優先度順に格納したバイト配列として渡します。設定するバッファアドレスに NULL を渡すか、あるいはバッファサイズに 0 を渡すと規定値にもどすことができます。規定値は X509 証明書（WOLFSSL_CERT_TYPE_X509）のみを扱う設定となっています。

	Name
int	**wolfSSL_set_client_cert_type**(WOLFSSL * ssl, const char * buf, int len) この関数はクライアント側で呼び出される場合には、サーバー側にCertificate メッセージで送信できる証明書タイプを設定します。サーバー側で呼び出される場合には、受入れ可能なクライアント証明書タイプを設定します。Raw Public Key 証明書を送受信したい場合にはこの関数を使って証明書タイプを設定しなければなりません。設定する証明書タイプは優先度順に格納したバイト配列として渡します。設定するバッファアドレスに NULL を渡すか、あるいはバッファサイズに 0 を渡すと規定値にもどすことができます。規定値は X509 証明書（WOLFSSL_CERT_TYPE_X509）のみを扱う設定となっています。
int	**wolfSSL_set_server_cert_type**(WOLFSSL * ssl, const char * buf, int len) この関数はサーバー側で呼び出される場合には、クライアント側にCertificate メッセージで送信できる証明書タイプを設定します。クライアント側で呼び出される場合には、受入れ可能なサーバー証明書タイプを設定します。Raw Public Key 証明書を送受信したい場合にはこの関数を使って証明書タイプを設定しなければなりません。設定する証明書タイプは優先度順に格納したバイト配列として渡します。設定するバッファアドレスに NULL を渡すか、あるいはバッファサイズに 0 を渡すと規定値にもどすことができます。規定値は X509 証明書（WOLFSSL_CERT_TYPE_X509）のみを扱う設定となっています。
void *	**wolfSSL_GetCookieCtx**(WOLFSSL * ssl) この関数は、WolfSSL 構造の IOCB_COOKIECTX メンバーを返します。
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) この関数は、WolfSSL が WolfSSL_ISOTP でコンパイルされている場合に使用する場合は、WolfSSL の場合は ISO-TP コンテキストを設定します。

A.6.2 Functions Documentation

A.6.2.1 function wolfSSLv23_method

```
WOLFSSL_METHOD * wolfSSLv23_method(
    void
)
```

この関数は、wolfSSLv23_client_method と同様に WOLFSSL_METHOD を返します（サーバー/クライアント）。

See:

- wolfSSL_new
- wolfSSL_free

Return:

- 作成に成功した場合は、WOLFSSL_METHOD ポインタを返します。
- メモリ割り当てエラーまたはメソッドの作成の失敗の場合は NULL を返します。

Example

```
WOLFSSL* ctx;
ctx  = wolfSSL_CTX_new(wolfSSLv23_method());
// check ret value
```

A.6.2.2 function wolfSSLv3_server_method

```
WOLFSSL_METHOD * wolfSSLv3_server_method(
    void
)
```

wolfSSLv3_server_method() 関数は、アプリケーションがサーバーであることを示すために使用され、SSL3.0 プロトコルのみをサポートします。この関数は、wolfSSL_CTX_new() を使用して SSL/TLS コンテキストを作成するときに使用される新しい WOLFSSL_METHOD 構造体のメモリを割り当てて初期化します。

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:

- 成功した場合、新しく作成された WOLFSSL_METHOD 構造体へのポインタを返します。
- XMALLOC を呼び出すときにメモリ割り当てが失敗した場合、基礎となる Malloc() 実装の失敗値が返されます（通常は errno が ENOMEM に設定されます）。

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);
...
```

A.6.2.3 function wolfSSLv3_client_method

```
WOLFSSL_METHOD * wolfSSLv3_client_method(
    void
)
```

wolfSSLv3_client_method() 関数は、アプリケーションがクライアントであり、SSL 3.0 プロトコルのみをサポートすることを示すために使用されます。この関数は、wolfSSL_CTX_new() を使用して SSL/TLS コンテキストを作成するときに使用される新しい WOLFSSL_METHOD 構造体のメモリを割り当てて初期化します。

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:

- 成功した場合、新しく作成された WOLFSSL_METHOD 構造体へのポインタを返します。
- XMALLOC を呼び出すときにメモリ割り当てが失敗した場合、基礎となる Malloc() 実装の失敗値が返されます（通常は errno が ENOMEM に設定されます）。

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);
...
```

A.6.2.4 function wolfTLSv1_server_method

```
WOLFSSL_METHOD * wolfTLSv1_server_method(
    void
)
```

wolfTLSv1_server_method() 関数は、アプリケーションがサーバーであることを示すために使用され、TLS 1.0 プロトコルのみをサポートします。この関数は、wolfSSL_ctx_new() を使用して SSL/TLS コンテキストを作成するときに使用される新しい WOLFSSL_METHOD 構造体のメモリを割り当てて初期化します。

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:

- 成功した場合、新しく作成された WOLFSSL_METHOD 構造体へのポインタを返します。

- XMALLOC を呼び出すときにメモリ割り当てが失敗した場合、基礎となる Malloc() 実装の失敗値が返されます（通常は errno が ENOMEM に設定されます）。

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);
...
```

A.6.2.5 function wolfTLSv1_client_method

```
WOLFSSL_METHOD * wolfTLSv1_client_method(
    void
)
```

wolftlsv1_client_method() 関数は、アプリケーションがクライアントであり、TLS 1.0 プロトコルのみをサポートすることを示すために使用されます。この関数は、wolfSSL_ctx_new() を使用して SSL/TLS コンテキストを作成するときに使用される新しい WOLFSSL_METHOD 構造体のメモリを割り当てて初期化します。

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:

- 成功した場合、新しく作成された WOLFSSL_METHOD 構造体へのポインタを返します。
- XMALLOC を呼び出すときにメモリ割り当てが失敗した場合、基礎となる Malloc() 実装の失敗値が返されます（通常は errno が ENOMEM に設定されます）。

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);
...
```

A.6.2.6 function wolfTLSv1_1_server_method

```
WOLFSSL_METHOD * wolfTLSv1_1_server_method(
    void
)
```

wolfTLSv1_1_server_method() 関数は、アプリケーションがサーバーであることを示すために使用され、TLS 1.1 プロトコルのみをサポートします。この関数は、wolfSSL_ctx_new() を使用して SSL/TLS コンテキストを作成するときに使用される新しい WOLFSSL_METHOD 構造体のメモリを割り当てて初期化します。

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:

- 成功した場合、新しく作成された WOLFSSL_METHOD 構造体へのポインタを返します。
- XMALLOC を呼び出すときにメモリ割り当てが失敗した場合、基礎となる Malloc() 実装の失敗値が返されます（通常は errno が ENOMEM に設定されます）。

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);
...
```

A.6.2.7 function wolfTLSv1_1_client_method

```
WOLFSSL_METHOD * wolfTLSv1_1_client_method(
    void
)
```

wolfTLSv1_1_client_method() 関数は、アプリケーションがクライアントであり、TLS 1.0 プロトコルのみをサポートすることを示すために使用されます。この関数は、wolfSSL_ctx_new() を使用して SSL/TLS コンテキストを作成するときに使用される新しい WOLFSSL_METHOD 構造体のメモリを割り当てて初期化します。

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:

- 成功した場合、新しく作成された WOLFSSL_METHOD 構造体へのポインタを返します。
- XMALLOC を呼び出すときにメモリ割り当てが失敗した場合、基礎となる Malloc() 実装の失敗値が返されます（通常は errno が ENOMEM に設定されます）。

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);
...
```

A.6.2.8 function wolfTLSv1_2_server_method

```
WOLFSSL_METHOD * wolfTLSv1_2_server_method(
    void
)
```

wolfTLSv1_2_server_method() 関数は、アプリケーションがサーバーであることを示すために使用され、TLS 1.2 プロトコルのみをサポートします。この関数は、wolfSSL_ctx_new() を使用して SSL/TLS コンテキストを作成するときに使用される新しい WOLFSSL_METHOD 構造体のメモリを割り当てて初期化します。

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:

- 成功した場合、新しく作成された WOLFSSL_METHOD 構造体へのポインタを返します。
- XMALLOC を呼び出すときにメモリ割り当てが失敗した場合、基礎となる Malloc() 実装の失敗値が返されます（通常は errno が ENOMEM に設定されます）。

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);
...
```

A.6.2.9 function wolfTLSv1_2_client_method

```
WOLFSSL_METHOD * wolfTLSv1_2_client_method(
    void
)
```

wolfTLSv1_2_client_method() 関数は、アプリケーションがクライアントであり、TLS 1.2 プロトコルのみをサポートすることを示すために使用されます。この関数は、wolfSSL_ctx_new() を使用して SSL/TLS コンテキストを作成するときに使用される新しい Wolfssl_method 構造体のメモリを割り当てて初期化します。

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:

- 成功した場合、新しく作成された WOLFSSL_METHOD 構造体へのポインタを返します。
- XMALLOC を呼び出すときにメモリ割り当てが失敗した場合、基礎となる Malloc() 実装の失敗値が返されます（通常は errno が ENOMEM に設定されます）。

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);
...
```

A.6.2.10 function wolfDTLSv1_client_method

```
WOLFSSL_METHOD * wolfDTLSv1_client_method(
    void
)
```

wolfdtlsv1_client_method() 関数は、アプリケーションがクライアントであり、DTLS 1.0 プロトコルのみをサポートすることを示すために使用されます。この関数は、wolfSSL_ctx_new() を使用して SSL/TLS コンテキストを作成するときに使用される新しい WOLFSSL_METHOD 構造体のメモリを割り当てて初期化します。この関数は、WolfSSL が DTLS サポート（–enable-dtls、または WOLFSSL_DTLS を定義することによって）ビルドされている場合にのみ使用できます。

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:

- 成功した場合、新しく作成された WOLFSSL_METHOD 構造体へのポインタを返します。
- XMALLOC を呼び出すときにメモリ割り当てが失敗した場合、基礎となる Malloc() 実装の失敗値が返されます（通常は errno が ENOMEM に設定されます）。

Example

```
WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfDTLSv1_client_method();
if (method == NULL) {
    // unable to get method
}

ctx = wolfSSL_CTX_new(method);
...
```

A.6.2.11 function wolfDTLSv1_server_method

```
WOLFSSL_METHOD * wolfDTLSv1_server_method(
    void
)
```

wolfDTLSv1_server_method() 関数は、アプリケーションがサーバーであることを示すために使用され、DTLS 1.0 プロトコルのみをサポートします。この関数は、wolfSSL_ctx_new() を使用して SSL/TLS コンテキストを作成するときに使用される新しい WOLFSSL_METHOD 構造体のメモリを割り当てて初期化します。この関数は、WolfSSL が DTLS サポート（–enable-dtls、または WOLFSSL_DTLS マクロを定義することによって）ビルドされている場合にのみ使用できます。

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:

- 成功した場合、新しく作成された WOLFSSL_METHOD 構造体へのポインタを返します。
- XMALLOC を呼び出すときにメモリ割り当てが失敗した場合、基礎となる Malloc() 実装の失敗値が返されます（通常は errno が ENOMEM に設定されます）。

Example

```
WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfDTLSv1_server_method();
```

```
if (method == NULL) {
    // unable to get method
}

ctx = wolfSSL_CTX_new(method);
...
```

A.6.2.12 function wolfDTLSv1_3_server_method

```
WOLFSSL_METHOD * wolfDTLSv1_3_server_method(
    void
)
```

wolfDTLSv1_3_server_method() 関数はアプリケーションがサーバーであることを示すために使用され、DTLS 1.3 プロトコルのみをサポートします。この関数は、wolfSSL_ctx_new() を使用して SSL/TLS コンテキストを作成するときに使用される新しい WOLFSSL_METHOD 構造体のメモリを割り当てて初期化します。この関数は、WolfSSL が DTLS サポート（–enable-dtls13、または WOLFSSL_DTLS13 を定義することによって）ビルドされている場合にのみ使用できます。

Parameters:

- **なし** *Example*

```
WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfDTLSv1_3_server_method();
if (method == NULL) {
    // unable to get method
}

ctx = wolfSSL_CTX_new(method);
...
```

See: wolfDTLSv1_3_client_method

Return:

- 成功した場合、新しく作成された WOLFSSL_METHOD 構造体へのポインタを返します。
- XMALLOC を呼び出すときにメモリ割り当てが失敗した場合、基礎となる Malloc() 実装の失敗値が返されます（通常は errno が ENOMEM に設定されます）。

A.6.2.13 function wolfDTLSv1_3_client_method

```
WOLFSSL_METHOD * wolfDTLSv1_3_client_method(
    void
)
```

wolfDTLSv1_3_client_method() 関数はアプリケーションがクライアントであることを示すために使用され、DTLS 1.3 プロトコルのみをサポートします。この関数は、wolfSSL_ctx_new() を使用して SSL/TLS コンテキストを作成するときに使用される新しい WOLFSSL_METHOD 構造体のメモリを割り当てて初期化します。この関数は、WolfSSL が DTLS サポート（–enable-dtls13、または WOLFSSL_DTLS13 を定義することによって）ビルドされている場合にのみ使用できます。

Parameters:

- **なし**

See: wolfDTLSv1_3_server_method

Return:

- 成功した場合、新しく作成された WOLFSSL_METHOD 構造体へのポインタを返します。
- XMALLOC を呼び出すときにメモリ割り当てが失敗した場合、基礎となる Malloc() 実装の失敗値が返されます（通常は errno が ENOMEM に設定されます）。

Example

```
WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfDTLSv1_3_client_method();
if (method == NULL) {
    // unable to get method
}

ctx = wolfSSL_CTX_new(method);
...
```

A.6.2.14 function wolfDTLS_server_method

```
WOLFSSL_METHOD * wolfDTLS_server_method(
    void
)
```

wolfDTLS_server_method() 関数はアプリケーションがサーバーであることを示すために使用され、可能な限り高いバージョン最小バージョンの DTLS プロトコルをサポートします。デフォルトの最小バージョンは WOLFSSL_MIN_DTLS_DOWNGRADE マクロでの指定をもとにしていて、実行時に wolfSSL_SetMinVersion() で変更することができます。この関数は、wolfSSL_ctx_new() を使用して SSL/TLS コンテキストを作成するときに使用される新しい WOLFSSL_METHOD 構造体のメモリを割り当てて初期化します。この関数は、WolfSSL が DTLS サポート（–enable-dtls、または WOLFSSL_DTLS を定義することによって）ビルドされている場合にのみ使用できます。

Parameters:

- **なし** *Example*

```
WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfDTLS_server_method();
if (method == NULL) {
    // unable to get method
}

ctx = wolfSSL_CTX_new(method);
...
```

See:

- wolfDTLS_client_method
- wolfSSL_SetMinVersion

Return:

- 成功した場合、新しく作成された WOLFSSL_METHOD 構造体へのポインタを返します。
- XMALLOC を呼び出すときにメモリ割り当てが失敗した場合、基礎となる Malloc() 実装の失敗値が返されます（通常は errno が ENOMEM に設定されます）。

A.6.2.15 function wolfDTLS_client_method

```
WOLFSSL_METHOD * wolfDTLS_client_method(
```

```
    void
)
```

wolfDTLS_client_method() 関数は アプリケーションがクライアントであることを示すために使用され、可能な限り高いバージョン最小バージョンの DTLS プロトコルをサポートします。デフォルトの最小バージョンは WOLFSSL_MIN_DTLS_DOWNGRADE マクロでの指定をもとにしていて、実行時に wolfSSL_SetMinVersion() で変更することができます。この関数は、wolfSSL_ctx_new() を使用して SSL/TLS コンテキストを作成するときに使用される新しい WOLFSSL_METHOD 構造体のメモリを割り当てて初期化します。この関数は、wolfSSL が DTLS サポート（–enable-dtls、または WOLFSSL_DTLS を定義することによって）ビルドされている場合にのみ使用できます。

Parameters:

- **なし**

See:

- wolfDTLS_server_method
- wolfSSL_SetMinVersion

Return:

- 成功した場合、新しく作成された WOLFSSL_METHOD 構造体へのポインタを返します。
- XMALLOC を呼び出すときにメモリ割り当てが失敗した場合、基礎となる Malloc() 実装の失敗値が返されます（通常は errno が ENOMEM に設定されます）。

Example

```
WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfDTLS_client_method();
if (method == NULL) {
    // unable to get method
}

ctx = wolfSSL_CTX_new(method);
...
```

A.6.2.16 function wolfSSL_use_old_poly

```
int wolfSSL_use_old_poly(
    WOLFSSL * ssl,
    int value
)
```

Chacha-Poly Aead Construction の最初のリリースと新しいバージョンの間にいくつかの違いがあるため、古いバージョンを使用してサーバー/クライアントと通信するオプションを追加しました。デフォルトでは、wolfSSL は新しいバージョンを使用します。

Parameters:

- **ssl** wolfSSL_new()を使用して作成した WOLFSSL 構造体へのポインタ。

See: none

Return: 0 成功の場合に返されます。

Example

```
int ret = 0;
WOLFSSL* ssl;
...

ret = wolfSSL_use_old_poly(ssl, 1);
if (ret != 0) {
    // failed to set poly1305 AEAD version
}
```

A.6.2.17 function wolfSSL_CTX_trust_peer_cert

```
int wolfSSL_CTX_trust_peer_cert(
    WOLFSSL_CTX * ctx,
    const char * file,
    int type
)
```

この関数は、TLS/SSL ハンドシェイクを実行するときにピアを検証するために使用する証明書をロードします。ハンドシェイク中に送信されたピア証明書は、この関数で指定された証明書の SKID と署名を比較することによって検証されます。これら 2 つのことが一致しない場合は、ピア証明書の検証にはロードされた CA 証明書が使用されます。この機能は WOLFSSL_TRUST_PEER_CERT マクロを定義することで機能を有効にできます。適切な使用法は例をご覧ください。

Parameters:

- **ctx** wolfSSL_CTX_new()で生成された WOLFSSL_CTX 構造体へのポインタ。
- **file** 証明書を含むファイルの名前へのポインタ

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 成功時に返されます。
- SSL_FAILURE CTX が NULL の場合、または両方のファイルと種類が無効な場合に返されます。
- SSL_BAD_FILETYPE ファイルが間違った形式である場合に返されます。
- SSL_BAD_FILE ファイルが存在しない場合に返されます。読み込め、または破損していません。
- MEMORY_E メモリ不足状態が発生した場合に返されます。
- ASN_INPUT_E base16 デコードがファイルに対して失敗した場合に返されます。

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
```

```
}
...
```

A.6.2.18 function wolfSSL_CTX_get_verify_depth

```
long wolfSSL_CTX_get_verify_depth(
    WOLFSSL_CTX * ctx
)
```

この関数は、WOLFSSL_CTX 構造体構造を使用して証明書チェーン深度を取得します。

See:

- wolfSSL_CTX_use_certificate_chain_file
- wolfSSL_get_verify_depth

Return:

- MAX_CHAIN_DEPTH WOLFSSL_CTX 構造体が NULL ではない場合に返されます。最大証明書チェーンピア深度の定数表現。
- BAD_FUNC_ARG WOLFSSL_CTX 構造体が NULL の場合に返されます。

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
}
```

A.6.2.19 function wolfSSL_CTX_new

```
WOLFSSL_CTX * wolfSSL_CTX_new(
    WOLFSSL_METHOD *
)
```

この関数は、所望の SSL/TLS プロトコル用メソッド構造体を引数に取って、新しい SSL コンテキストを作成します。

See: wolfSSL_new

Return:

- pointer 成功した場合、新しく作成された WOLFSSL_CTX 構造体へのポインタを返します。
- NULL 失敗時に返されます。

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
}
```

A.6.2.20 function wolfSSL_new

```
WOLFSSL * wolfSSL_new(
    WOLFSSL_CTX *
)
```

この関数はすでに作成された SSL コンテキスト (WOLFSSL_CTX) を入力として、新しい SSL セッション (WOLFSSL) を作成します。

See: wolfSSL_CTX_new

Return:

- 成功した場合、新しく作成された WOLFSSL 構造体へのポインタを返します。
- NULL 失敗時に返されます。

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
}
```

A.6.2.21 function wolfSSL_set_fd

```
int wolfSSL_set_fd(
    WOLFSSL * ssl,
    int fd
)
```

この関数は、SSL 接続の入出力機能としてファイル記述子 (fd) を割り当てます。通常これはソケットファイル記述子になります。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ
- **fd** SSL/TLS 接続に使用するファイルディスクリプタ

See:

- wolfSSL_CTX_SetIOSend
- wolfSSL_CTX_SetIORecv
- wolfSSL_SetIOReadCtx
- wolfSSL_SetIOWriteCtx

Return:

- SSL_SUCCESS 成功時に返されます。
- BAD_FUNC_ARG 失敗時に返されます。

Example

```
int sockfd;
WOLFSSL* ssl = 0;
...

ret = wolfSSL_set_fd(ssl, sockfd);
if (ret != SSL_SUCCESS) {
    // failed to set SSL file descriptor
}
```

A.6.2.22 function wolfSSL_set_dtls_fd_connected

```
int wolfSSL_set_dtls_fd_connected(
    WOLFSSL * ssl,
    int fd
)
```

この関数はファイルディスクリプタ (fd) を SSL コネクションの入出力手段として設定します。通常はソケットファイルディスクリプタが指定されます。この関数は DTLS 専用の API であり、ソケットは接続済みとマークされます。したがって、与えられた fd に対する recvfrom と sendto 呼び出しでの addr と addr_len は NULL に設定されます。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ
- **fd** SSL/TLS コネクションに使用するファイルディスクリプタ。

See:

- wolfSSL_CTX_SetIOSend
- wolfSSL_CTX_SetIORecv
- wolfSSL_SetIOReadCtx
- wolfSSL_SetIOWriteCtx
- wolfDTLS_SetChGoodCb

Return:

- SSL_SUCCESS 成功時に返されます。
- BAD_FUNC_ARG 失敗時に返されます。

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
}
```

A.6.2.23 function wolfDTLS_SetChGoodCb

```
int wolfDTLS_SetChGoodCb(
    WOLFSSL * ssl,
    ClientHelloGoodCb cb,
    void * user_ctx
)
```

この関数は DTLS ClientHello メッセージが正しく処理できた際に呼び出されるコールバック関数を設定します。クッキー交換メカニズムを使用する場合 (DTLS1.2 の HelloVerifyRequest か DTLS1.3 のクッキー拡張を伴った HelloRetryRequest のいずれかを使用する場合) には、クッキー交換が成功した時点でこのコールバック関数が呼び出されます。この機能はひとつの WOLFSSL オブジェクトを新たな接続を待ち受けるリスナーとして使い, ClientHello が検証された WOLFSSL オブジェクトから絶縁させることができます。この場合の検証はクッキー交換か ClientHello が正しいフォーマットになっているかのチェックによってなされます。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ
- **fd** SSL/TLS コネクションに使用するファイルディスクリプタ。

See: wolfSSL_set_dtls_fd_connected

Return:

- SSL_SUCCESS 成功時に返されます。
- BAD_FUNC_ARG 失敗時に返されます。

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

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
}
```

A.6.2.24 function wolfSSL_set_using_nonblock

```
void wolfSSL_set_using_nonblock(
    WOLFSSL * ssl,
    int nonblock
)
```

この関数は、WOLFSSL オブジェクトに基礎となる I/O がノンブロックであることを通知します。アプリケーションが WOLFSSL オブジェクトを作成した後、ブロッキング以外のソケットで使用する場合は、wolfssl_set_using_nonblock() を呼び出します。これにより、wolfssl オブジェクトは、EWOULDBLOCK を受信することを意味します。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ
- **nonblock** WOLFSSL オブジェクトにノンブロッキング I/O を使用することを通知するフラグ。１を指定することでノンブロッキング I/O を使用することを指定する。

See:

- wolfSSL_get_using_nonblock
- wolfSSL_dtls_got_timeout
- wolfSSL_dtls_get_current_timeout

Return: なし

Example

```
WOLFSSL* ssl = 0;
...
wolfSSL_set_using_nonblock(ssl, 1);
```

A.6.2.25 function wolfSSL_CTX_free

```
void wolfSSL_CTX_free(
    WOLFSSL_CTX *
)
```

この関数は、割り当てられた WOLFSSL_CTX オブジェクトを解放します。この関数は CTX 参照数を減らし、参照カウントが 0 に達したときにのみコンテキストを解放します。

Parameters:

- **ctx** wolfSSL_CTX_new()を使用して作成された WOLFSSL_CTX 構造体へのポインタ

See:

- wolfSSL_CTX_new
- wolfSSL_new
- wolfSSL_free

Return: なし

Example

```
WOLFSSL_CTX* ctx = 0;
...
wolfSSL_CTX_free(ctx);
```

A.6.2.26 function wolfSSL_free

```
void wolfSSL_free(
    WOLFSSL *
)
```

この関数は割り当てられた WOLFSSL オブジェクトを解放します。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ

See:

- wolfSSL_CTX_new
- wolfSSL_new
- wolfSSL_CTX_free

Return: なし

Example

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

WOLFSSL* ssl = 0;
...
wolfSSL_free(ssl);
```

A.6.2.27 function wolfSSL_set_session

```
int wolfSSL_set_session(
    WOLFSSL * ssl,
    WOLFSSL_SESSION * session
)
```

この関数は、SSL オブジェクト SSL が SSL/TLS 接続を確立する目的で使用するセッションを設定します。セッション再開を行う場合、wolfSSL_shutdown() を呼び出す前に wolfSSL_get1_session() を呼び出してセッションオブジェクトを取得し、セッション ID を保存しておく必要があります。後で、アプリケーションは新しい WOLFSSL オブジェクトを作成し、保存したセッションを wolfSSL_set_session() に渡す必要があります。その後アプリケーションは wolfSSL_connect() を呼び出し、wolfSSL はセッション再開を試みます。wolfSSL サーバーコードでは、デフォルトでセッション再開を許可します。wolfSSL_get1_session() によって返されたオブジェクトは、アプリケーションが使用後に解放する必要があります。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ
- **session** WOLFSSL_SESSION 構造体へのポインタ。

See: wolfSSL_get1_session

Return:

- SSL_SUCCESS セッションを正常に設定すると返されます。
- SSL_FAILURE 失敗した場合に返されます。これはセッションキャッシュが無効になっている、またはセッションがタイムアウトした場合によって発生する可能性があります。
- OPENSSL_EXTRA と WOLFSSL_ERROR_CODE_OPENSSL が定義されている場合には、セッションがタイムアウトしていても SSL_SUCCESS が返されます。

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);
...
```

A.6.2.28 function wolfSSL_CTX_set_verify

```
void wolfSSL_CTX_set_verify(
    WOLFSSL_CTX * ctx,
    int mode,
```

```
    VerifyCallback verify_callback
)
```

この関数はリモートピアの検証方法を設定し、また証明書検証コールバック関数を SSL コンテキストに登録することもできます。検証コールバックは、検証障害が発生した場合にのみ呼び出されます。検証コールバックが必要な場合は、NULL ポインタを verify_callback に使用できます。ピア証明書の検証モードは、論理的またはフラグのリストです。可能なフラグ値は次のとおりです: SSL_VERIFY_NONE -クライアントモード：クライアントはサーバーから受信した証明書を検証せず、ハンドシェイクは通常どおり続きます。-サーバーモード：サーバーはクライアントに証明書要求を送信しません。そのため、クライアント検証は有効になりません。SSL_VERIFY_PEER -クライアントモード：クライアントはハンドシェイク中にサーバーから受信した証明書を検証します。これは wolfSSL ではデフォルトでオンにされます。したがって、このオプションを使用すると効果がありません。-サーバーモード：サーバーは証明書要求をクライアントに送信し、受信したクライアント証明書を確認します。SSL_VERIFY_FAIL_IF_NO_PEER_CERT -クライアントモード：クライアント側で使用されていない場合は効果がありません。-サーバーモード：要求されたときにクライアントが証明書の送信に失敗した場合は、サーバー側で検証が失敗します（SSL サーバーの SSL_VERIFY_PEER を使用する場合）。SSL_VERIFY_FAIL_EXCEPT_PSK -クライアントモード：クライアント側で使用されていない場合は効果がありません。-サーバーモード：PSK 接続の場合を除き、検証は SSL_VERIFY_FAIL_IF_NO_PEER_CERT と同じです。PSK 接続が行われている場合、接続はピア証明書なしで通過します。

Parameters:

- **ctx** wolfSSL_CTX_new()で作成された SSL コンテキストへのポインタ。
- **mode** ピアの証明書をどのように検証するかを示すフラグ値
- **verify_callback** 証明書検証が失敗した際に呼び出されるコールバック関数。必要がないなら NULL を指定すること。

See: wolfSSL_set_verify

Return: なし

Example

```
WOLFSSL_CTX*    ctx    = 0;
...
wolfSSL_CTX_set_verify(ctx, (WOLFSSL_VERIFY_PEER |
                        WOLFSSL_VERIFY_FAIL_IF_NO_PEER_CERT), NULL);
```

A.6.2.29 function wolfSSL_set_verify

```
void wolfSSL_set_verify(
    WOLFSSL * ssl,
    int mode,
    VerifyCallback verify_callback
)
```

この関数はリモートピアの検証方法を設定し、また証明書検証コールバック関数を WOLFSSL オブジェクトに登録することもできます。検証コールバックは、検証障害が発生した場合にのみ呼び出されます。検証コールバックが必要な場合は、NULL ポインタを verify_callback に使用できます。ピア証明書の検証モードは、論理的またはフラグのリストです。可能なフラグ値は次のとおりです: SSL_VERIFY_NONE -クライアントモード：クライアントはサーバーから受信した証明書を検証せず、ハンドシェイクは通常どおり続きます。-サーバーモード：サーバーはクライアントに証明書要求を送信しません。そのため、クライアント検証は有効になりません。SSL_VERIFY_PEER -クライアントモード：クライアントはハンドシェイク中にサーバーから受信した証明書を検証します。これは wolfSSL ではデフォルトでオンにされます。したがって、このオプションを使用すると効果がありません。-サーバーモード：サーバーは証明書要求をクライアントに送信し、受信したクライアント証明書を確認します。SSL_VERIFY_FAIL_IF_NO_PEER_CERT -クライアントモード：クライアント側で使用されていない場合は効果がありません。-サーバーモード：要求されたときにクライアントが証明書の送信に失敗した場合は、サーバー側で検証が失敗します（SSL サー

バーの SSL_VERIFY_PEER を使用する場合)。SSL_VERIFY_FAIL_EXCEPT_PSK -クライアントモード：クライアント側で使用されていない場合は効果がありません。-サーバーモード：PSK 接続の場合を除き、検証は SSL_VERIFY_FAIL_IF_NO_PEER_CERT と同じです。PSK 接続が行われている場合、接続はピア証明書なしで通過します。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ
- **mode** ピアの証明書をどのように検証するかを示すフラグ値
- **verify_callback** 証明書検証が失敗した際に呼び出されるコールバック関数。必要がないなら NULL を指定すること。

See: wolfSSL_CTX_set_verify

Return: なし

Example

```
WOLFSSL* ssl = 0;
...
wolfSSL_set_verify(ssl, SSL_VERIFY_PEER | SSL_VERIFY_FAIL_IF_NO_PEER_CERT, 0);
```

A.6.2.30 function wolfSSL_CTX_set_session_cache_mode

```
long wolfSSL_CTX_set_session_cache_mode(
    WOLFSSL_CTX * ctx,
    long mode
)
```

この関数は SSL セッションキャッシュ機能を有効または無効にします。動作はモードに使用される値によって異なります。モードの値は次のとおりです：SSL_SESS_CACHE_OFF - セッションキャッシングを無効にします。デフォルトでセッションキャッシングがオンになっています。SSL_SESS_CACHE_NO_AUTO_CLEAR - セッションキャッシュのオートフラッシュを無効にします。デフォルトで自動フラッシングはオンになっています。

Parameters:

- **ctx** wolfSSL_CTX_new()で作成された SSL コンテキストへのポインタ。
- **mode** セッションキャッシュの振る舞いを変更する為に使用します。

See:

- wolfSSL_flush_sessions
- wolfSSL_get1_session
- wolfSSL_set_session
- wolfSSL_get_sessionID
- wolfSSL_CTX_set_timeout

Return: SSL_SUCCESS 成功に戻ります。

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
}
```

A.6.2.31 function wolfSSL_CTX_memrestore_cert_cache

```
int wolfSSL_CTX_memrestore_cert_cache(
    WOLFSSL_CTX * ctx,
    const void * mem,
    int sz
)
```

この関数は証明書キャッシュをメモリから復元します。

Parameters:

- **ctx** wolfSSL_CTX_new()を使用して作成された WOLFSSL_CTX 構造体へのポインタ。
- **mem** 証明書キャッシュに復元される値を保持しているバッファへのポインタ。
- **sz** バッファのサイズ

See: CM_MemRestoreCertCache

Return:

- SSL_SUCCESS 関数とサブルーチンがエラーなしで実行された場合に返されます。
- BAD_FUNC_ARG CTX または MEM パラメータが NULL または SZ パラメータがゼロ以下の場合に返されます。
- BUFFER_E CERT キャッシュメモリバッファが小さすぎると戻ります。
- CACHE_MATCH_ERROR CERT キャッシュヘッダーの不一致があった場合に返されます。
- BAD_MUTEX_E ロックミューテックスが失敗した場合に返されます。

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
}
```

A.6.2.32 function wolfSSL_CTX_set_cipher_list

```
int wolfSSL_CTX_set_cipher_list(
    WOLFSSL_CTX * ctx,
    const char * list
)
```

この関数は、与えられた WOLFSSL_CTX に暗号スイートリストを設定します。この暗号スイートリストは、このコンテキストを使用して作成された新しい SSL セッション（WolfSSL）のデフォルトリストになります。リスト内の暗号は、優先度の高いものの順に順にソートされるべきです。wolfSSL_CTX_set_cipher_list()が呼び出される都度、特定の SSL コンテキストの暗号スイートリストを提供されたリストにリセットします。暗号スイートリストはヌル終端されたコロン区切りリストです。たとえば、リストの値が「DHE-RSA-AES256-SHA256：DHE-RSA-AES128-SHA256：AES256-SHA256」有効な暗号値は、src/internal.c の cipher_names [] 配列のフルネーム値です。（有効な暗号化値の明確なリストの場合は src/internal.c をチェックしてください）

Parameters:

- **ctx** wolfSSL_CTX_new()で作成された SSL コンテキストへのポインタ。
- **list** ヌル終端されたコロン区切りの暗号スイートリスト文字列へのポインタ。

See:

- wolfSSL_set_cipher_list
- wolfSSL_CTX_new

Return:

- SSL_SUCCESS 成功時に返されます。
- SSL_FAILURE 失敗した場合に返されます。

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
}
```

A.6.2.33 function wolfSSL_set_cipher_list

```
int wolfSSL_set_cipher_list(
    WOLFSSL * ssl,
    const char * list
)
```

この関数は、特定の WolfSSL オブジェクト（SSL セッション）の暗号スイートリストを設定します。この暗号スイートリストは、このコンテキストを使用して作成された新しい SSL セッション（WolfSSL）のデフォルトリストになります。リスト内の暗号は、優先度の高いものの順に順にソートされるべきです。wolfSSL_CTX_set_cipher_list()が呼び出される都度、特定の SSL コンテキストの暗号スイートリストを提供されたリストにリセットします。暗号スイートリストはヌル終端されたコロン区切りリストです。たとえば、リストの値が「DHE-RSA-AES256-SHA256：DHE-RSA-AES128-SHA256：AES256-SHA256」有効な暗号値は、src/internal.c の cipher_names [] 配列のフルネーム値です。（有効な暗号化値の明確なリストの場合は src/internal.c をチェックしてください）

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ
- **list** ヌル終端されたコロン区切りの暗号スイートリスト文字列へのポインタ。

See:

- wolfSSL_CTX_set_cipher_list
- wolfSSL_new

Return:

- SSL_SUCCESS 機能完了に成功したときに返されます。
- SSL_FAILURE 失敗した場合に返されます。

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
}
```

A.6.2.34 function wolfSSL_dtls13_set_send_more_acks

```
void wolfSSL_dtls13_set_send_more_acks(
    WOLFSSL * ssl,
    int value
)
```

この関数は、ライブラリが中断を検出したときにすぐに他のピアに ACK を送信するかどうかを設定します。ACK をすぐに送信すると、遅延は最小限に抑えられますが、必要以上に多くの帯域幅が消費される可能性があります。アプリケーションが独自にタイマーを管理しており、このオプションが 0 に設定されている場合、アプリケーションコードは wolfSSL_dtls13_use_quick_timeout() を使用して、遅延した ACK を送信するためにより速いタイムアウトを設定する必要があるかどうかを判断できます。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ
- **value** 設定を行う場合には 1 を行わない場合には 0 を設定します。

See:

- wolfSSL_dtls
- wolfSSL_dtls_get_peer
- wolfSSL_dtls_got_timeout
- wolfSSL_dtls_set_peer
- wolfSSL_dtls13_use_quick_timeout

A.6.2.35 function wolfSSL_dtls_set_timeout_init

```
int wolfSSL_dtls_set_timeout_init(
    WOLFSSL * ssl,
    int
)
```

この関数は DTLS タイムアウトを設定します。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ
- **value** タイムアウトオプションを有効にする場合には 1 を指定し、無効にする場合には 0 を指定します。

See:

- wolfSSL_dtls_set_timeout_max
- wolfSSL_dtls_got_timeout

Return:

- SSL_SUCCESS 関数がエラーなしで実行された場合に返されます。SSL の DTLS_TIMEOUT_INIT と DTLS_TIMEOUT メンバーが設定されています。
- BAD_FUNC_ARG 引数 ssl が NULL の場合、またはタイムアウトが 0 以下の場合に返されます。タイムアウト引数が許可されている最大値を超えている場合にも返されます。

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.
}
```

A.6.2.36 function wolfSSL_get1_session

```
WOLFSSL_SESSION * wolfSSL_get1_session(
    WOLFSSL * ssl
)
```

この関数は、WOLFSSL 構造体から WOLFSSL_SESSION を参照型として返します。これには、wolfSSL_SESSION_free を呼び出してセッション参照を解除する必要があります。WOLFSSL_SESSION は、セッションの再開を実行するために必要なすべての必要な情報を含み、新しいハンドシェイクなしで接続を再確立します。セッションの再開の場合、wolfSSL_shutdown() をセッションオブジェクトに呼び出す前に、アプリケーションはオブジェクトから wolfssl_get1_session() を呼び出して保存する必要があります。これはセッションへのポインタを返します。その後、アプリケーションは新しい WOLFSSL オブジェクトを作成し、保存したセッションを wolfssl_set_session() に割り当てる必要があります。この時点で、アプリケーションは wolfssl_connect() を呼び出し、WolfSSL はセッションを再開しようとします。WolfSSL サーバーコードでは、デフォルトでセッションの再開を許可します。wolfssl_get1_session() によって返されたオブジェクトは、アプリケーションが使用後は解放される必要があります。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ

See:

- wolfSSL_new
- wolfSSL_free
- wolfSSL_SESSION_free

Return:

- WOLFSSL_SESSION 成功の場合はセッションポインタを返します。
- NULL ssl が NULL の場合、SSL セッションキャッシュが無効になっている場合、WolfSSL はセッション ID を使用できない、またはミューテックス関数が失敗します。

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);
```

A.6.2.37 function wolfSSLv23_client_method

```
WOLFSSL_METHOD * wolfSSLv23_client_method(
    void
)
```

wolfsslv23_client_method() 関数は、アプリケーションがクライアントであることを示すために使用され、SSL 3.0~TLS 1.3 の間でサーバーでサポートされている最高のプロトコルバージョンをサポートします。この関数は、wolfSSL_CTX_new() を使用して SSL / TLS コンテキストを作成するときに使用される新しい Wolfssl_method 構造体のメモリを割り当てて初期化します。WolfSSL クライアントとサーバーの両方が堅

牢なバージョンのダウングレード機能を持っています。特定のプロトコルバージョンメソッドがどちらの側で使用されている場合は、そのバージョンのみがネゴシエートされたり、エラーが返されます。たとえば、TLSV1 を使用し、SSLv3 のみに接続しようとするクライアントは、TLSV1.1 に接続しても失敗します。この問題を解決するために、wolfsslv23_client_method() 関数を使用するクライアントは、サーバーでサポートされている最高のプロトコルバージョンを使用し、必要に応じて SSLv3 にダウングレードします。この場合、クライアントは SSLv3 - TLSv1.3 を実行しているサーバーに接続できるようになります。

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 成功すると、wolfssl_method へのポインタが返されます。
- Failure xmalloc を呼び出すときにメモリ割り当てが失敗した場合、基礎となる Malloc() 実装の失敗値が返されます（通常は errno が ENOMEM に設定されます）。

Example

```
WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;
method = wolfSSLv23_client_method();
if (method == NULL) {
// unable to get method
}

ctx = wolfSSL_CTX_new(method);
...
```

A.6.2.38 function wolfSSL_ASN1_INTEGER_to_BN

```
WOLFSSL_BIGNUM * wolfSSL_ASN1_INTEGER_to_BN(
    const WOLFSSL_ASN1_INTEGER * ai,
    WOLFSSL_BIGNUM * bn
)
```

この関数は、WOLFSSL_ASN1_INTEGER 値を WOLFSSL_BIGNUM 構造体にコピーするために使用されます。

Parameters:

- **ai** WOLFSSL_ASN1_INTEGER 構造体へのポインタ
- **bn** もし、既存の WOLFSSL_BIGNUM 構造体にコピーしたい場合そのポインタをこの引数で指定します。NULL を指定すると新たに WOLFSSL_BIGNUM 構造体が生成されて使用されます。

See: none

Return:

- pointer WOLFSSL_ASN1_INTEGER 値を正常にコピーすると、WOLFSSL_BIGNUM ポインタが返されます。
- Null 失敗時に返されます。

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
```

A.6.2.39 function wolfSSL_CTX_add_extra_chain_cert

```
long wolfSSL_CTX_add_extra_chain_cert(
    WOLFSSL_CTX * ctx,
    WOLFSSL_X509 * x509
)
```

この関数は、WOLFSSL_CTX 構造で構築されている内部チェーンに証明書を追加します。

Parameters:

- **ctx** 証明書を追加するための WOLFSSL_CTX 構造。
- **x509** WOLFSSL_X509 構造体へのポインタ。

See:

- wolfSSL_CTX_new
- wolfSSL_CTX_free

Return:

- SSL_SUCCESS 証明書の追加に成功したら。
- SSL_FAILURE チェーンに証明書を追加することが失敗した場合。

Example

```
WOLFSSL_CTX* ctx;
WOLFSSL_X509* x509;
int ret;
// create ctx
ret = wolfSSL_CTX_add_extra_chain_cert(ctx, x509);
// check ret value
```

A.6.2.40 function wolfSSL_CTX_get_read_ahead

```
int wolfSSL_CTX_get_read_ahead(
    WOLFSSL_CTX * ctx
)
```

この関数は、WOLFSSL_CTX 構造から Get Read Hape フラグを返します。

Parameters:

- **ctx** WOLFSSL_CTX 構造体へのポインタ

See:

- wolfSSL_CTX_new
- wolfSSL_CTX_free
- wolfSSL_CTX_set_read_ahead

Return:

- flag 成功すると、読み取り先のフラグを返します。
- SSL_FAILURE ctx が null の場合、ssl_failure が返されます。

Example

```
WOLFSSL_CTX* ctx;
int flag;
// setup ctx
flag = wolfSSL_CTX_get_read_ahead(ctx);
//check flag
```

A.6.2.41 function wolfSSL_CTX_set_read_ahead

```
int wolfSSL_CTX_set_read_ahead(
    WOLFSSL_CTX * ctx,
    int v
)
```

この関数は、WOLFSSL_CTX 構造内の読み出し先のフラグを設定します。

Parameters:

- **ctx** WOLFSSL_CTX 構造体へのポインタ
- **v** 先読みフラグ

See:

- wolfSSL_CTX_new
- wolfSSL_CTX_free
- wolfSSL_CTX_get_read_ahead

Return:

- SSL_SUCCESS ctx が先読みフラグを設定した場合。
- SSL_FAILURE ctx が NULL の場合に返されます。

Example

```
WOLFSSL_CTX* ctx;
int flag;
int ret;
// setup ctx
ret = wolfSSL_CTX_set_read_ahead(ctx, flag);
// check return value
```

A.6.2.42 function wolfSSL_CTX_set_tlsext_status_arg

```
long wolfSSL_CTX_set_tlsext_status_arg(
    WOLFSSL_CTX * ctx,
    void * arg
)
```

この関数は OCSP で使用するオプション引数を設定します。

Parameters:

- **ctx** WOLFSSL_CTX 構造へのポインタ
- **arg** ユーザー引数

See:

- wolfSSL_CTX_new

- wolfSSL_CTX_free

Return:

- SSL_FAILURE CTX または IT の CERT Manager が NULL の場合。
- SSL_SUCCESS 正常に設定されている場合。

Example

```
WOLFSSL_CTX* ctx;
void* data;
int ret;
// setup ctx
ret = wolfSSL_CTX_set_tlsext_status_arg(ctx, data);

//check ret value
```

A.6.2.43 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
)
```

この関数は、PRF コールバックに渡すオプションの引数を設定します。

Parameters:

- **ctx** WOLFSSL_CTX 構造へのポインタ
- **arg** ユーザー引数

See:

- wolfSSL_CTX_new
- wolfSSL_CTX_free

Return:

- SSL_FAILURE CTX が NULL の場合
- SSL_SUCCESS 正常に設定されている場合。

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
```

A.6.2.44 function wolfSSL_set_options

```
long wolfSSL_set_options(
    WOLFSSL * s,
    long op
)
```

この関数は、SSL のオプションマスクを設定します。いくつかの有効なオプションは、ssl_op_all、ssl_op_cookie_exchange、ssl_op_no_sslv2、ssl_op_no_sslv3、ssl_op_no_tlsv1_1、ssl_op_no_tlsv1_2、ssl_op_no_compression です。

Parameters:

- **s** オプションマスクを設定するための WolfSSL 構造。
- **op** オプションマスク。以下の値が指定可能です：

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 SSL に格納されている更新されたオプションマスク値を返します。

Example

```
WOLFSSL* ssl;
unsigned long mask;
mask = SSL_OP_NO_TLSv1
mask  = wolfSSL_set_options(ssl, mask);
// check mask
```

A.6.2.45 function wolfSSL_get_options

```
long wolfSSL_get_options(
    const WOLFSSL * ssl
)
```

この関数は現在のオプションマスクを返します。

Parameters:

- **ssl** WOLFSSL 構造体へのポインタ

See:

- wolfSSL_new
- wolfSSL_free
- wolfSSL_set_options

Return: val SSL に格納されているマスク値を返します。

Example

```
WOLFSSL* ssl;
unsigned long mask;
mask  = wolfSSL_get_options(ssl);
// check mask
```

A.6.2.46 function wolfSSL_set_tlsext_debug_arg

```
long wolfSSL_set_tlsext_debug_arg(
    WOLFSSL * ssl,
    void * arg
)
```

この関数は、渡されたデバッグ引数を設定するために使用されます。

Parameters:

- **ssl** 引数を設定するための WolfSSL 構造。
- **arg** デバッグ引数

See:

- wolfSSL_new
- wolfSSL_free

Return:

- SSL_SUCCESS 成功時に返されます。
- SSL_FAILURE NULL SSL が渡された場合。

Example

```
WOLFSSL* ssl;
void* args;
int ret;
// create ssl object
ret  = wolfSSL_set_tlsext_debug_arg(ssl, args);
// check ret value
```

A.6.2.47 function wolfSSL_get_verify_result

```
long wolfSSL_get_verify_result(
    const WOLFSSL * ssl
)
```

Parameters:

- **ssl** WOLFSSL 構造体へのポインタ

See:

- wolfSSL_new
- wolfSSL_free

Return:

- X509_V_OK 成功した検証について
- SSL_FAILURE NULL SSL が渡された場合。

bri f この関数は、れは、ピアの証明書を確認しようとした後に結果を取得するために使用されます。*Example*

```
WOLFSSL* ssl;
long ret;
// attempt/complete handshake
ret  = wolfSSL_get_verify_result(ssl);
// check ret value
```

A.6.2.48 function wolfSSL_CTX_allow_anon_cipher

```
int wolfSSL_CTX_allow_anon_cipher(
    WOLFSSL_CTX *
)
```

この機能により、CTX 構造の HAVAnon メンバーがコンパイル中に定義されている場合は、CTX 構造の HABANON メンバーを有効にします。

See: none

Return:

- SSL_SUCCESS 機能が正常に実行され、CTX の Haveannon メンバーが 1 に設定されている場合に返されます。
- SSL_FAILURE CTX 構造が NULL の場合に返されます。

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
```

A.6.2.49 function wolfSSLv23_server_method

```
WOLFSSL_METHOD * wolfSSLv23_server_method(
    void
)
```

wolfsslv23_server_method() 関数は、アプリケーションがサーバーであることを示すために使用され、SSL 3.0 - TLS 1.3 からプロトコルバージョンと接続するクライアントをサポートします。この関数は、wolfSSL_CTX_new() を使用して SSL / TLS コンテキストを作成するときに使用される新しい Wolfssl_method 構造体のメモリを割り当てて初期化します。

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 成功した場合、呼び出しは新しく作成された wolfssl_method 構造へのポインタを返します。
- Failure xmalloc を呼び出すときにメモリ割り当てが失敗した場合、基礎となる Malloc() 実装の失敗値が返されます（通常は errno が enomeem に設定されます）。

Example

```
WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfSSLv23_server_method();
if (method == NULL) {
    // unable to get method
}

ctx = wolfSSL_CTX_new(method);
...
```

A.6.2.50 function wolfSSL_state

```
int wolfSSL_state(
    WOLFSSL * ssl
)
```

See:

- wolfSSL_new
- wolfSSL_free

Return:

- wolfssl_error SSL エラー状態、通常はマイナスを返します
- BAD_FUNC_ARG ssl が NULL の場合

bri f この関数は、れは、WolfSSL 構造体の内部エラー状態を取得するために使用されます。*Example*

```
WOLFSSL* ssl;
int ret;
// create ssl object
ret  = wolfSSL_state(ssl);
// check ret value
```

A.6.2.51 function wolfSSL_check_domain_name

```
int wolfSSL_check_domain_name(
    WOLFSSL * ssl,
    const char * dn
)
```

wolfssl デフォルトでは、有効な日付範囲と検証済みの署名のためにピア証明書をチェックします。wolfssl_connect() または wolfssl_accept() の前にこの関数を呼び出すと、実行するチェックのリストにドメイン名チェックが追加されます。DN 受信時にピア証明書を確認するためのドメイン名を保持します。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WolfSSL 構造体へのポインタ

See: none

Return:

- SSL_SUCCESS 成功時に返されます。
- SSL_FAILURE メモリエラーが発生した場合に返されます。

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
}
```

A.6.2.52 function wolfSSL_set_compression

```
int wolfSSL_set_compression(
    WOLFSSL * ssl
)
```

SSL 接続に圧縮を使用する機能をオンにします。両側には圧縮がオンになっている必要があります。そうでなければ圧縮は使用されません。ZLIB ライブラリは実際のデータ圧縮を実行します。ライブラリにコンパイルするには、システムの設定システムに–with-libz を使用し、そうでない場合は hand_libz を定義します。送受信されるメッセージの実際のサイズを減らす前にデータを圧縮している間に、圧縮によって保存されたデータの量は通常、ネットワークの遅いすべてのネットワークを除いたものよりも分析に時間がかかります。

See: none

Return:

- SSL_SUCCESS 成功時に返されます。
- NOT_COMPILED_IN 圧縮サポートがライブラリに組み込まれていない場合に返されます。

Example

```
int ret = 0;
WOLFSSL* ssl = 0;
...
ret = wolfSSL_set_compression(ssl);
if (ret == SSL_SUCCESS) {
    // successfully enabled compression for SSL session
}
```

A.6.2.53 function wolfSSL_set_timeout

```
int wolfSSL_set_timeout(
    WOLFSSL * ssl,
    unsigned int to
)
```

この関数は SSL セッションタイムアウト値を秒単位で設定します。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WolfSSL 構造体へのポインタ

See:

- wolfSSL_get1_session
- wolfSSL_set_session

Return:

- SSL_SUCCESS セッションを正常に設定すると返されます。
- BAD_FUNC_ARG ssl が NULL の場合に返されます。

Example

```
int ret = 0;
WOLFSSL* ssl = 0;
...

ret = wolfSSL_set_timeout(ssl, 500);
if (ret != SSL_SUCCESS) {
    // failed to set session timeout value
}
...
```

A.6.2.54 function wolfSSL_CTX_set_timeout

```
int wolfSSL_CTX_set_timeout(
    WOLFSSL_CTX * ctx,
    unsigned int to
)
```

この関数は、指定された SSL コンテキストに対して、SSL セッションのタイムアウト値を秒単位で設定します。

Parameters:

- **ctx** wolfSSL_CTX_new()で作成された SSL コンテキストへのポインタ。

See:

- wolfSSL_flush_sessions
- wolfSSL_get1_session
- wolfSSL_set_session
- wolfSSL_get_sessionID
- wolfSSL_CTX_set_session_cache_mode

Return:

- the wolfssl_error_code_openssl の場合、以前のタイムアウト値
- defined 成功しています。定義されていない場合、SSL_SUCCESS は返されます。
- BAD_FUNC_ARG 入力コンテキスト（CTX）が NULL のときに返されます。

Example

```
WOLFSSL_CTX*    ctx    = 0;
...
ret = wolfSSL_CTX_set_timeout(ctx, 500);
if (ret != SSL_SUCCESS) {
    // failed to set session timeout value
}
```

A.6.2.55 function wolfSSL_CTX_UnloadCAs

```
int wolfSSL_CTX_UnloadCAs(
    WOLFSSL_CTX *
)
```

この関数は CA 署名者リストをアンロードし、署名者全体のテーブルを解放します。

See:

- wolfSSL_CertManagerUnloadCAs
- LockMutex

- FreeSignerTable
- UnlockMutex

Return:

- SSL_SUCCESS 機能の実行に成功したことに戻ります。
- BAD_FUNC_ARG WOLFSSL_CTX 構造体が null の場合、または他の方法では未解決の引数値がサブルーチンに渡された場合に返されます。
- BAD_MUTEX_E ミューテックスエラーが発生した場合に返されます。lockmutex() は 0 を返しませんでした。

Example

```
WOLFSSL_METHOD method = wolfTLSv1_2_client_method();
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(method);
...
if(!wolfSSL_CTX_UnloadCAs(ctx)){
    // The function did not unload CAs
}
```

A.6.2.56 function wolfSSL_CTX_Unload_trust_peers

```
int wolfSSL_CTX_Unload_trust_peers(
    WOLFSSL_CTX *
)
```

この関数は、以前にロードされたすべての信頼できるピア証明書をアンロードするために使用されます。マクロ wolfssl_trust_peer_cert を定義することで機能が有効になっています。

See:

- wolfSSL_CTX_trust_peer_buffer
- wolfSSL_CTX_trust_peer_cert

Return:

- SSL_SUCCESS 成功時に返されます。
- BAD_FUNC_ARG CTX が NULL の場合に返されます。
- SSL_BAD_FILE ファイルが存在しない場合に返されます。読み込め、または破損していません。
- MEMORY_E メモリ不足状態が発生した場合に返されます。

Example

```
int ret = 0;
WOLFSSL_CTX* ctx;
...
ret = wolfSSL_CTX_Unload_trust_peers(ctx);
if (ret != SSL_SUCCESS) {
    // error unloading trusted peer certs
}
...
```

A.6.2.57 function wolfSSL_CTX_trust_peer_buffer

```
int wolfSSL_CTX_trust_peer_buffer(
    WOLFSSL_CTX * ctx,
    const unsigned char * in,
    long sz,
    int format
)
```

この関数は、TLS / SSL ハンドシェイクを実行するときにピアを検証するために使用する証明書をロードします。ハンドシェイク中に送信されたピア証明書は、使用可能なときにスキッドを使用することによって比較されます。これら 2 つのことが一致しない場合は、ロードされた CAS が使用されます。ファイルの代わりにバッファーの場合は、wolfssl_ctx_trust_peer_cert と同じ機能です。特徴はマクロ wolfssl_trust_peer_cert を定義することによって有効になっています適切な使用法の例を参照してください。

Parameters:

- **ctx** wolfSSL_CTX_new()で作成された SSL コンテキストへのポインタ。
- **buffer** 証明書を含むバッファへのポインタ。
- **sz** バッファ入力の長さ。

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 成功すると
- SSL_FAILURE CTX が NULL の場合、または両方のファイルと種類が無効な場合に返されます。
- SSL_BAD_FILETYPE ファイルが間違った形式である場合に返されます。
- SSL_BAD_FILE ファイルが存在しない場合に返されます。読み込め、または破損していません。
- MEMORY_E メモリ不足状態が発生した場合に返されます。
- ASN_INPUT_E base16 デコードがファイルに対して失敗した場合に返されます。

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
}
...
```

A.6.2.58 function wolfSSL_CTX_set_group_messages

```
int wolfSSL_CTX_set_group_messages(
    WOLFSSL_CTX *
)
```

この機能は、可能な限りハンドシェイクメッセージのグループ化をオンにします。

See:

- wolfSSL_set_group_messages
- wolfSSL_CTX_new

Return:

- SSL_SUCCESS 成功に戻ります。
- BAD_FUNC_ARG 入力コンテキストが NULL の場合、返されます。

Example

```
WOLFSSL_CTX* ctx = 0;
...
ret = wolfSSL_CTX_set_group_messages(ctx);
if (ret != SSL_SUCCESS) {
    // failed to set handshake message grouping
}
```

A.6.2.59 function wolfSSL_set_group_messages

```
int wolfSSL_set_group_messages(
    WOLFSSL *
)
```

この機能は、可能な限りハンドシェイクメッセージのグループ化をオンにします。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ

See:

- wolfSSL_CTX_set_group_messages
- wolfSSL_new

Return:

- SSL_SUCCESS 成功に戻ります。
- BAD_FUNC_ARG 入力コンテキストが NULL の場合、返されます。

Example

```
WOLFSSL* ssl = 0;
...
ret = wolfSSL_set_group_messages(ssl);
if (ret != SSL_SUCCESS) {
// failed to set handshake message grouping
}
```

A.6.2.60 function wolfSSL_CTX_SetMinVersion

```
int wolfSSL_CTX_SetMinVersion(
    WOLFSSL_CTX * ctx,
    int version
)
```

この関数は、許可されている最小のダウングレードバージョンを設定します。接続が（wolfsslv23_client_method または wolfsslv23_server_method）を使用して、接続がダウングレードできる場合にのみ適用されます。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ

See: SetMinVersionHelper

Return:

- SSL_SUCCESS エラーなしで返された関数と最小バージョンが設定されている場合に返されます。

- BAD_FUNC_ARG WOLFSSL_CTX 構造が NULL の場合、または最小バージョンがサポートされていない場合に返されます。

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
}
```

A.6.2.61 function wolfSSL_SetVersion

```
int wolfSSL_SetVersion(
    WOLFSSL * ssl,
    int version
)
```

この関数は、バージョンで指定されたバージョンを使用して、指定された SSL セッション（WolfSSL オブジェクト）の SSL/TLS プロトコルバージョンを設定します。これにより、SSL セッション（SSL）のプロトコル設定が最初に定義され、SSL コンテキスト（wolfSSL_CTX_new()）メソッドの種類によって上書きされます。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WolfSSL 構造へのポインタ。

See: wolfSSL_CTX_new

Return:

- SSL_SUCCESS 成功時に返されます。
- BAD_FUNC_ARG 入力 SSL オブジェクトが NULL または誤ったプロトコルバージョンがバージョンで指定されている場合に返されます。

Example

```
int ret = 0;
WOLFSSL* ssl;
...

ret = wolfSSL_SetVersion(ssl, WOLFSSL_TLSV1);
if (ret != SSL_SUCCESS) {
    // failed to set SSL session protocol version
}
```

A.6.2.62 function wolfSSL_UseALPN

```
int wolfSSL_UseALPN(
    WOLFSSL * ssl,
    char * protocol_name_list,
    unsigned int protocol_name_listSz,
    unsigned char options
)
```

wolfssl セッションに ALPN を設定します。

Parameters:

- **ssl** 使用する WolfSSL セッション。

- **protocol_name_list** 使用するプロトコル名のリスト。カンマ区切り文字列が必要です。
- **protocol_name_listSz** プロトコル名のリストのサイズ。

See: TLSX_UseALPN

Return:

- WOLFSSL_SUCCESS: 成功時に返されます。
- BAD_FUNC_ARG SSL または PROTOCOL_NAME_LIST が NULL または PROTOCOL_NAME_LISTSZ が大きすぎたり、オプションがサポートされていないものを含みます。
- MEMORY_ERROR プロトコルリストのメモリの割り当て中にエラーが発生しました。
- SSL_FAILURE 失敗時に返されます。

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
}
```

A.6.2.63 function wolfSSL_CTX_UseSessionTicket

```
int wolfSSL_CTX_UseSessionTicket(
    WOLFSSL_CTX * ctx
)
```

この関数は、セッションチケットを使用するように WolfSSL コンテキストを設定します。

See: TLSX_UseSessionTicket

Return:

- SSL_SUCCESS 関数は正常に実行されます。
- BAD_FUNC_ARG CTX が NULL の場合に返されます。
- MEMORY_E 内部関数内のメモリの割り当て中にエラーが発生しました。

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
}
```

A.6.2.64 function wolfSSL_check_private_key

```
int wolfSSL_check_private_key(
    const WOLFSSL * ssl
)
```

この関数は、秘密鍵が使用されている証明書との一致であることを確認します。

See:

- wolfSSL_new
- wolfSSL_free

Return:

- SSL_SUCCESS うまく一致します。
- SSL_FAILURE エラーケースに遭遇した場合
- <0 ssl_failure 以外のすべてのエラーケースは負の値です。

Example

```
WOLFSSL* ssl;
int ret;
// create and set up ssl
ret  = wolfSSL_check_private_key(ssl);
// check ret value
```

A.6.2.65 function wolfSSL_use_certificate

```
int wolfSSL_use_certificate(
    WOLFSSL * ssl,
    WOLFSSL_X509 * x509
)
```

ハンドシェイク中に使用するために、WolfSSL 構造の証明書を設定するために使用されます。

Parameters:

- **ssl** 証明書を設定するための WolfSSL 構造。

See:

- wolfSSL_new
- wolfSSL_free

Return:

- SSL_SUCCESS 設定の成功した引数について。
- SSL_FAILURE NULL 引数が渡された場合。

Example

```
 WOLFSSL* ssl;
WOLFSSL_X509* x509
int ret;
// create ssl object and x509
ret  = wolfSSL_use_certificate(ssl, x509);
// check ret value
```

A.6.2.66 function wolfSSL_use_certificate_ASN1

```
int wolfSSL_use_certificate_ASN1(
    WOLFSSL * ssl,
    unsigned char * der,
    int derSz
)
```

Parameters:

- **ssl** 証明書を設定するための WolfSSL 構造。
- **der** 使用する証明書。

See:

- wolfSSL_new
- wolfSSL_free

Return:

- SSL_SUCCESS 設定の成功した引数について。
- SSL_FAILURE NULL 引数が渡された場合。

bii こ f は、この関数は、handshake の間に使用するために WolfSSL 構造の証明書を設定するために使用されます。DER フォーマットバッファが予想されます。*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
```

A.6.2.67 function wolfSSL_SESSION_get_master_key

```
int wolfSSL_SESSION_get_master_key(
    const WOLFSSL_SESSION * ses,
    unsigned char * out,
    int outSz
)
```

これはハンドシェイクを完了した後にマスターキーを取得するために使用されます。

Parameters:

- **ses** マスターシークレットバッファを取得するための WolfSSL_SESSION 構造。
- **out** データを保持するためのバッファ。

See:

- wolfSSL_new
- wolfSSL_free

Return:

- 0 データの取得に成功した場合、0 より大きい値を返します。
- 0 ランダムなデータバッファまたはエラー状態が返されない場合は 0
- max 渡された OUTSZ が 0 の場合、必要な最大バッファサイズが返されます。

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
```

A.6.2.68 function wolfSSL_SESSION_get_master_key_length

```
int wolfSSL_SESSION_get_master_key_length(
    const WOLFSSL_SESSION * ses
)
```

これはマスター秘密鍵の長さを取得するために使用されます。

See:

- wolfSSL_new
- wolfSSL_free

Return: size マスターシークレットキーサイズを返します。

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
```

A.6.2.69 function wolfSSL_CTX_set_cert_store

```
void wolfSSL_CTX_set_cert_store(
    WOLFSSL_CTX * ctx,
    WOLFSSL_X509_STORE * str
)
```

Parameters:

- **ctx** Cert Store ポインタを設定するための WolfSSL_CTX 構造体へのポインタ。

See:

- wolfSSL_CTX_new
- wolfSSL_CTX_free

Return: none 返品不可。

bri f この関数は、れは、CTX の WOLFSSL_X509_STORE 構造の設定機能です。*Example*

```
WOLFSSL_CTX ctx;
WOLFSSL_X509_STORE* st;
// setup ctx and st
st = wolfSSL_CTX_set_cert_store(ctx, st);
//use st
```

A.6.2.70 function wolfSSL_CTX_get_cert_store

```
WOLFSSL_X509_STORE * wolfSSL_CTX_get_cert_store(
    WOLFSSL_CTX * ctx
)
```

See:

- wolfSSL_CTX_new
- wolfSSL_CTX_free
- wolfSSL_CTX_set_cert_store

Return:

- WOLFSSL_X509_STORE* ポインタを正常に入手します。
- NULL NULL 引数が渡された場合に返されます。

bri f この関数は、れは、CTX の WOLFSSL_X509_STORE 構造のゲッター関数です。*Example*

```
WOLFSSL_CTX ctx;
WOLFSSL_X509_STORE* st;
// setup ctx
st = wolfSSL_CTX_get_cert_store(ctx);
//use st
```

A.6.2.71 function wolfSSL_get_server_random

```
size_t wolfSSL_get_server_random(
    const WOLFSSL * ssl,
    unsigned char * out,
    size_t outlen
)
```

Parameters:

- **ssl** クライアントのランダムデータバッファを取得するための WolfSSL 構造。
- **out** ランダムデータを保持するためのバッファ。

See:

- wolfSSL_new
- wolfSSL_free

Return:

- 0 データの取得に成功した場合、0 より大きい値を返します。
- 0 ランダムなデータバッファまたはエラー状態が返されない場合は 0
- max 渡された OUTSZ が 0 の場合、必要な最大バッファサイズが返されます。

biief は、この関数は、ハンドシェイク中にサーバーによって送信されたランダムなデータを取得するために使用されます。*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
```

A.6.2.72 function wolfSSL_get_client_random

```
size_t wolfSSL_get_client_random(
    const WOLFSSL * ssl,
    unsigned char * out,
    size_t outSz
)
```

Parameters:

- **ssl** クライアントのランダムデータバッファを取得するための WolfSSL 構造。
- **out** ランダムデータを保持するためのバッファ。

See:

- wolfSSL_new
- wolfSSL_free

Return:

- 0 データの取得に成功した場合、0 より大きい値を返します。
- 0 ランダムなデータバッファまたはエラー状態が返されない場合は 0
- max 渡された OUTSZ が 0 の場合、必要な最大バッファサイズが返されます。

biief は、この関数は、ハンドシェイク中にクライアントによって送信されたランダムなデータを取得するために使用されます。*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
```

A.6.2.73 function wolfSSL_CTX_get_default_passwd_cb

```
wc_pem_password_cb * wolfSSL_CTX_get_default_passwd_cb(
    WOLFSSL_CTX * ctx
)
```

これは CTX で設定されたパスワードコールバックのゲッター関数です。

See:

- wolfSSL_CTX_new
- wolfSSL_CTX_free

Return:

- func 成功すると、コールバック関数を返します。
- NULL CTX が NULL の場合、NULL が返されます。

Example

```
WOLFSSL_CTX* ctx;
wc_pem_password_cb cb;
// setup ctx
cb = wolfSSL_CTX_get_default_passwd_cb(ctx);
//use cb
```

A.6.2.74 function wolfSSL_CTX_get_default_passwd_cb_userdata

```
void * wolfSSL_CTX_get_default_passwd_cb_userdata(
    WOLFSSL_CTX * ctx
)
```

See:

- wolfSSL_CTX_new
- wolfSSL_CTX_free

Return:

- pointer 成功すると、ユーザーデータポインタを返します。
- NULL CTX が NULL の場合、NULL が返されます。

bri f この関数は、れは、CTX で設定されているパスワードコールバックユーザーデータの取得機能です。
Example

```
WOLFSSL_CTX* ctx;
void* data;
// setup ctx
data = wolfSSL_CTX_get_default_passwd_cb(ctx);
//use data
```

A.6.2.75 function wolfSSL_CTX_clear_options

```
long wolfSSL_CTX_clear_options(
    WOLFSSL_CTX * ctx,
    long opt
)
```

この関数は、WOLFSSL_CTX オブジェクトのオプションビットをリセットします。

See:

- wolfSSL_CTX_new
- wolfSSL_new
- wolfSSL_free

Return: option 新しいオプションビット

Example

```
WOLFSSL_CTX* ctx = 0;
...
wolfSSL_CTX_clear_options(ctx, SSL_OP_NO_TLSv1);
```

A.6.2.76 function wolfSSL_set_msg_callback

```
int wolfSSL_set_msg_callback(
    WOLFSSL * ssl,
    SSL_Msg_Cb cb
)
```

この関数は SSL 内のコールバックを設定します。コールバックはハンドシェイクメッセージを観察することです。CB の NULL 値はコールバックをリセットします。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WolfSSL 構造へのポインタ。

See: wolfSSL_set_msg_callback_arg

Return:

- SSL_SUCCESS 成功しています。
- SSL_FAILURE NULL SSL が渡された場合。

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
```

A.6.2.77 function wolfSSL_set_msg_callback_arg

```
int wolfSSL_set_msg_callback_arg(
    WOLFSSL * ssl,
    void * arg
)
```

この関数は、SSL 内の関連コールバックコンテキスト値を設定します。値はコールバック引数に渡されます。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WolfSSL 構造へのポインタ。

See: wolfSSL_set_msg_callback

Return: none 返品不可。

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);
```

A.6.2.78 function wolfSSL_send_hrr_cookie

```
int wolfSSL_send_hrr_cookie(
    WOLFSSL * ssl,
    const unsigned char * secret,
    unsigned int secretSz
)
```

この関数はサーバー側で呼び出されて、HellORetryRequest メッセージに Cookie を含める必要があることを示します。Cookie は現在のトランスクリプトのハッシュを保持しているので、別のサーバープロセスは応答で ClientHello を処理できます。秘密は Cookie データの整合性チェックを Generting するときに使用されます。

Parameters:

- **ssl** l wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ。
- **秘密を保持しているバッファへのポインタを秘密にします。渡す NULL は、新しいランダムシークレットを生成することを示します。**

- **シークスのサイズをバイト単位でサイズ。0 を渡すと、デフォルトのサイズを使用することを示します.WC_SHA256_DIGEST_SIZE（または SHA-256 が使用できない場合は WC_SHA_DIGEST_SIZE）。**

See: wolfSSL_new

Return:

- BAD_FUNC_ARG ssl が NULL の場合、または TLS v1.3 を使用していない場合。
- SIDE_ERROR クライアントで呼び出された場合。
- WOLFSSL_SUCCESS 成功した場合に返されます。
- MEMORY_ERROR 秘密を保存するために動的メモリを割り当てる場合に失敗しました。

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
}
```

A.6.2.79 function wolfSSL_disable_hrr_cookie

```
int wolfSSL_disable_hrr_cookie(
    WOLFSSL * ssl
)
```

この関数はサーバー側で呼び出され、HelloRetryRequest メッセージがクッキーを含んではならないこと、DTLSv1.3 が使用されている場合にはクッキーの交換がハンドシェークに含まれないことを表明します。DTLSv1.3 ではクッキー交換を行わないとサーバーが DoS/Amplification 攻撃を受けやすくなる可能性があることに留意してください。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ。

See: wolfSSL_send_hrr_cookie

Return:

- WOLFSSL_SUCCESS 成功時に返されます。
- BAD_FUNC_ARG ssl が NULL あるいは TLS v1.3 を使用していない場合に返されます。
- SIDE_ERROR クライアント側でこの関数が呼び出された場合に返されます。

A.6.2.80 function wolfSSL_CTX_no_ticket_TLSv13

```
int wolfSSL_CTX_no_ticket_TLSv13(
    WOLFSSL_CTX * ctx
)
```

この関数はサーバー上で呼び出され、ハンドシェイク完了時にセッション再開のためのセッションチケットの送信を行わないようにします。

Parameters:

- **ctx** wolfSSL_CTX_new()で作成された WOLFSSL_CTX 構造体へのポインタ。

See: wolfSSL_no_ticket_TLSv13

Return:

- BAD_FUNC_ARG CTX が NULL の場合、または TLS v1.3 を使用していない場合。
- SIDE_ERROR クライアントで呼び出された場合。

Example

```
int ret;
WOLFSSL_CTX* ctx;
...
ret = wolfSSL_CTX_no_ticket_TLSv13(ctx);
if (ret != 0) {
    // failed to set no ticket
}
```

A.6.2.81 function wolfSSL_no_ticket_TLSv13

```
int wolfSSL_no_ticket_TLSv13(
    WOLFSSL * ssl
)
```

ハンドシェイクが完了すると、この関数はサーバー上で再開セッションチケットの送信を停止するように呼び出されます。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ。

See: wolfSSL_CTX_no_ticket_TLSv13

Return:

- BAD_FUNC_ARG ssl が NULL の場合、または TLS v1.3 を使用していない場合。
- SIDE_ERROR クライアントで呼び出された場合。

Example

```
int ret;
WOLFSSL* ssl;
...
ret = wolfSSL_no_ticket_TLSv13(ssl);
if (ret != 0) {
    // failed to set no ticket
}
```

A.6.2.82 function wolfSSL_CTX_no_dhe_psk

```
int wolfSSL_CTX_no_dhe_psk(
    WOLFSSL_CTX * ctx
)
```

この関数は、Authentication にプリシェアキーを使用している場合、DIFFIE-HELLMAN（DH）スタイルのキー交換を許可する TLS V1.3 WolfSSL コンテキストで呼び出されます。

Parameters:

- **ctx** wolfSSL_CTX_new()で作成された WOLFSSL_CTX 構造体へのポインタ。

See: wolfSSL_no_dhe_psk

Return: BAD_FUNC_ARG CTX が NULL の場合、または TLS v1.3 を使用していない場合。

Example

```
int ret;
WOLFSSL_CTX* ctx;
...
ret = wolfSSL_CTX_no_dhe_psk(ctx);
if (ret != 0) {
    // failed to set no DHE for PSK handshakes
}
```

A.6.2.83 function wolfSSL_no_dhe_psk

```
int wolfSSL_no_dhe_psk(
    WOLFSSL * ssl
)
```

この関数は、事前共有鍵を使用している TLS V1.3 クライアントまたはサーバーで、に Diffie-Hellman（DH）スタイルの鍵交換を許可しないように設定します。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ。

See: wolfSSL_CTX_no_dhe_psk

Return: BAD_FUNC_ARG ssl が NULL の場合、または TLS v1.3 を使用していない場合。

Example

```
int ret;
WOLFSSL* ssl;
...
ret = wolfSSL_no_dhe_psk(ssl);
if (ret != 0) {
    // failed to set no DHE for PSK handshakes
}
```

A.6.2.84 function wolfSSL_CTX_allow_post_handshake_auth

```
int wolfSSL_CTX_allow_post_handshake_auth(
    WOLFSSL_CTX * ctx
)
```

この関数は、TLS v1.3 クライアントの WolfSSL コンテキストで呼び出され、クライアントはサーバーからの要求に応じて Post Handshake を送信できるようにします。これは、クライアント認証などを必要としないページを持つ Web サーバーに接続するときに役立ちます。

Parameters:

- **ctx** wolfSSL_CTX_new()で作成された WOLFSSL_CTX 構造体へのポインタ。

See:

- wolfSSL_allow_post_handshake_auth
- wolfSSL_request_certificate

Return:

- BAD_FUNC_ARG CTX が NULL の場合、または TLS v1.3 を使用していない場合。
- SIDE_ERROR サーバーで呼び出された場合。

Example

```
int ret;
WOLFSSL_CTX* ctx;
...
ret = wolfSSL_allow_post_handshake_auth(ctx);
if (ret != 0) {
    // failed to allow post handshake authentication
}
```

A.6.2.85 function wolfSSL_allow_post_handshake_auth

```
int wolfSSL_allow_post_handshake_auth(
    WOLFSSL * ssl
)
```

この関数は、TLS V1.3 クライアント WolfSSL で呼び出され、クライアントはサーバーからの要求に応じてハンドシェイクを送ります。handshake クライアント認証拡張機能は ClientHello で送信されます。これは、クライアント認証などを必要としないページを持つ Web サーバーに接続するときに役立ちます。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ。

See:

- wolfSSL_CTX_allow_post_handshake_auth
- wolfSSL_request_certificate

Return:

- BAD_FUNC_ARG ssl が NULL の場合、または TLS v1.3 を使用していない場合。
- SIDE_ERROR サーバーで呼び出された場合。

Example

```
int ret;
WOLFSSL* ssl;
...
ret = wolfSSL_allow_post_handshake_auth(ssl);
if (ret != 0) {
    // failed to allow post handshake authentication
}
```

A.6.2.86 function wolfSSL_CTX_set1_groups_list

```
int wolfSSL_CTX_set1_groups_list(
    WOLFSSL_CTX * ctx,
    char * list
)
```

この関数は楕円曲線グループのリストを設定して、WolfSSL コンテキストを希望の順に設定します。リストはヌル終了したテキスト文字列、およびコロン区切りリストです。この関数を呼び出して、TLS v1.3 接続で使用する鍵交換楕円曲線パラメータを設定します。

Parameters:

- **ctx** wolfSSL_CTX_new()で作成された WOLFSSL_CTX 構造体へのポインタ。
- **list** 楕円曲線グループのコロン区切りリストである文字列をリストします。

See:

- wolfSSL_set1_groups_list

- wolfSSL_CTX_set_groups
- wolfSSL_set_groups
- wolfSSL_UseKeyShare
- wolfSSL_preferred_group

Return: WOLFSSL_FAILURE ポインタパラメータが NULL の場合、wolfssl_max_group_count グループが多い場合は、グループ名が認識されないか、TLS v1.3 を使用していません。

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
}
```

A.6.2.87 function wolfSSL_set1_groups_list

```
int wolfSSL_set1_groups_list(
    WOLFSSL * ssl,
    char * list
)
```

この関数は楕円曲線グループのリストを設定して、WolfSSL を希望の順に設定します。リストはヌル終了したテキスト文字列、およびコロン区切りリストです。この関数を呼び出して、TLS v1.3 接続で使用する鍵交換楕円曲線パラメータを設定します。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ。
- **list** 鍵交換グループのコロン区切りリストである文字列をリストします。

See:

- wolfSSL_CTX_set1_groups_list
- wolfSSL_CTX_set_groups
- wolfSSL_set_groups
- wolfSSL_UseKeyShare
- wolfSSL_preferred_group

Return: WOLFSSL_FAILURE ポインタパラメータが NULL の場合、wolfssl_max_group_count グループが多い場合は、グループ名が認識されないか、TLS v1.3 を使用していません。

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
}
```

A.6.2.88 function wolfSSL_CTX_set_groups

```
int wolfSSL_CTX_set_groups(
```

```
    WOLFSSL_CTX * ctx,
    int * groups,
    int count
)
```

この関数は楕円曲線グループのリストを設定して、WolfSSL コンテキストを希望の順に設定します。リストは、Count で指定された識別子の数を持つグループ識別子の配列です。この関数を呼び出して、TLS v1.3 接続で使用する鍵交換楕円曲線パラメータを設定します。

Parameters:

- **ctx** wolfSSL_CTX_new()で作成された WOLFSSL_CTX 構造体へのポインタ。
- **groups** 識別子によって鍵交換グループのリストをグループ化します。
- **count** グループ内の鍵交換グループの数を数えます。

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 ポインタパラメータが NULL の場合、グループ数は wolfssl_max_group_count を超えているか、TLS v1.3 を使用していません。

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
}
```

A.6.2.89 function wolfSSL_set_groups

```
int wolfSSL_set_groups(
    WOLFSSL * ssl,
    int * groups,
    int count
)
```

この関数は、wolfssl を許すために楕円曲線グループのリストを設定します。リストは、Count で指定された識別子の数を持つグループ識別子の配列です。この関数を呼び出して、TLS v1.3 接続で使用する鍵交換楕円曲線パラメータを設定します。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ。
- **groups** 識別子によって鍵交換グループのリストをグループ化します。
- **count** グループ内の鍵交換グループの数を数えます。

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 ポインタパラメータが NULL の場合､グループ数が WolfSSL_MAX_GROUP_COUNT を超えている場合、任意の識別子は認識されないか、TLS v1.3 を使用していません。

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
}
```

A.6.2.90 function wolfSSL_CTX_set_max_early_data

```
int wolfSSL_CTX_set_max_early_data(
    WOLFSSL_CTX * ctx,
    unsigned int sz
)
```

この関数は、WolfSSL コンテキストを使用して TLS V1.3 サーバーによって受け入れられるアーリーデータの最大量を設定します。この関数を呼び出して、再生攻撃を軽減するためのプロセスへのアーリーデータの量を制限します。初期のデータは、セッションチケットが送信されたこと、したがってセッションチケットが再開されるたびに同じ接続の鍵から派生した鍵によって保護されます。値は再開のためにセッションチケットに含まれています。ゼロの値は、セッションチケットを使用してクライアントによってアーリーデータを送信することを示します。アーリーデータバイト数をアプリケーションで実際には可能な限り低く保つことをお勧めします。

Parameters:

- **ctx** wolfSSL_CTX_new()で作成された WOLFSSL_CTX 構造体へのポインタ。
- **sz** バイト単位で受け入れるアーリーデータのサイズ。

See:

- wolfSSL_set_max_early_data
- wolfSSL_write_early_data
- wolfSSL_read_early_data

Return:

- BAD_FUNC_ARG CTX が NULL の場合、または TLS v1.3 を使用していない場合。
- SIDE_ERROR クライアントで呼び出された場合。

Example

```
int ret;
WOLFSSL_CTX* ctx;
...
ret = wolfSSL_CTX_set_max_early_data(ctx, 128);
```

```
if (ret != WOLFSSL_SUCCESS) {
    // failed to set group list
}
```

A.6.2.91 function wolfSSL_set_max_early_data

```
int wolfSSL_set_max_early_data(
    WOLFSSL * ssl,
    unsigned int sz
)
```

この関数は、WolfSSL コンテキストを使用して TLS V1.3 サーバーによって受け入れられるアーリーデータの最大量を設定します。この関数を呼び出して、再生攻撃を軽減するためプロセスへのアーリーデータの量を制限します。初期のデータは、セッションチケットが送信されたこと、したがってセッションチケットが再開されるたびに同じ接続の鍵から派生した鍵によって保護されます。値は再開のためにセッションチケットに含まれています。ゼロの値は、セッションチケットを使用してクライアントによってアーリーデータを送信することを示します。アーリーデータバイト数をアプリケーションで実際には可能な限り低く保つことをお勧めします。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ。
- **SZ クライアントからバイト単位で受け入れるアーリーデータのサイズ。**

See:

- wolfSSL_CTX_set_max_early_data
- wolfSSL_write_early_data
- wolfSSL_read_early_data

Return:

- BAD_FUNC_ARG ssl が NULL の場合、または TLS v1.3 を使用していない場合。
- SIDE_ERROR クライアントで呼び出された場合。

Example

```
int ret;
WOLFSSL* ssl;
...
ret = wolfSSL_set_max_early_data(ssl, 128);
if (ret != WOLFSSL_SUCCESS) {
    // failed to set group list
}
```

A.6.2.92 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
)
```

この関数は、TLS v1.3 接続のプレシェア鍵（PSK）クライアント側コールバックを設定します。コールバックは PSK アイデンティティを見つけ、そのキーと、ハンドシェイクに使用する暗号の名前を返します。この関数は、WOLFSSL_CTX 構造体の client_psk_tls13_cb メンバーを設定します。

Parameters:

- **ctx** wolfSSL_CTX_new()で作成された WOLFSSL_CTX 構造体へのポインタ。

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);
```

A.6.2.93 function wolfSSL_set_psk_client_tls13_callback

```
void wolfSSL_set_psk_client_tls13_callback(
    WOLFSSL * ssl,
    wc_psk_client_tls13_callback cb
)
```

この関数は、TLS v1.3 接続のプレシェアキー（PSK）クライアント側コールバックを設定します。コールバックは PSK アイデンティティを見つけ、そのキーと、ハンドシェイクに使用する暗号の名前を返します。この関数は、wolfssl 構造体の Options フィールドの client_psk_tls13_cb メンバーを設定します。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ。

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);
```

A.6.2.94 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
)
```

この関数は、TLS v1.3 接続用の事前共有鍵（PSK）サーバ側コールバックを設定します。コールバックは PSK アイデンティティを見つけ、そのキーと、ハンドシェイクに使用する暗号の名前を返します。この関数は、wolfssl_ctx 構造体の server_psk_tls13_cb メンバーを設定します。

Parameters:

- **ctx** wolfSSL_CTX_new()で作成された WOLFSSL_CTX 構造体へのポインタ。

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);
```

A.6.2.95 function wolfSSL_set_psk_server_tls13_callback

```
void wolfSSL_set_psk_server_tls13_callback(
    WOLFSSL * ssl,
    wc_psk_server_tls13_callback cb
)
```

この関数は、TLS v1.3 接続用の事前共有鍵（PSK）サーバ側コールバックを設定します。コールバックはPSK アイデンティティを見つけ、そのキーと、ハンドシェイクに使用する暗号の名前を返します。この関数は、wolfssl 構造体のオプションフィールドの server_psk_tls13_cb メンバーを設定します。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ。

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);
```

A.6.2.96 function wolfSSL_UseKeyShare

```
int wolfSSL_UseKeyShare(
    WOLFSSL * ssl,
    word16 group
)
```

この関数は、キーペアの生成を含むグループからキーシェアエントリを作成します。Keyshare エクステンションには、鍵交換のための生成されたすべての公開鍵が含まれています。この関数が呼び出されると、指定されたグループのみが含まれます。優先グループがサーバーに対して以前に確立されているときにこの関数を呼び出します。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ。
- **キー交換グループ識別子をグループ化します。**

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 ssl が NULL の場合に返されます。
- MEMORY_E 動的メモリ割り当てに失敗すると返されます。

Example

```
int ret;
WOLFSSL* ssl;
...
```

```
ret = wolfSSL_UseKeyShare(ssl, WOLFSSL_ECC_X25519);
if (ret != WOLFSSL_SUCCESS) {
    // failed to set key share
}
```

A.6.2.97 function wolfSSL_NoKeyShares

```
int wolfSSL_NoKeyShares(
    WOLFSSL * ssl
)
```

この関数は、ClientHello で鍵共有が送信されないように呼び出されます。これにより、ハンドシェイクに鍵交換が必要な場合は、サーバーが HelloretryRequest で応答するように強制します。予想される鍵交換グループが知られておらず、キーの生成を不必要に回避するときにこの機能を呼び出します。鍵交換が必要なときにハンドシェイクを完了するために追加の往復が必要になることに注意してください。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ。

See: wolfSSL_UseKeyShare

Return:

- BAD_FUNC_ARG ssl が NULL の場合に返されます。
- SIDE_ERROR サーバーで呼び出された場合。

Example

```
int ret;
WOLFSSL* ssl;
...
ret = wolfSSL_NoKeyShares(ssl);
if (ret != WOLFSSL_SUCCESS) {
    // failed to set no key shares
}
```

A.6.2.98 function wolfTLSv1_3_server_method_ex

```
WOLFSSL_METHOD * wolfTLSv1_3_server_method_ex(
    void * heap
)
```

この関数は、アプリケーションがサーバーであることを示すために使用され、TLS 1.3 プロトコルのみをサポートします。この関数は、wolfSSL_CTX_new() を使用して SSL / TLS コンテキストを作成するときに使用される新しい Wolfssl_method 構造体のメモリを割り当てて初期化します。

Parameters:

- **ヒープ静的メモリ割り当て中に静的メモリ割り当て器が使用するバッファへのポインタを使用します。**

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: 新しく作成された wWOLFSSL_METHOS 構造体へのポインタを返します。

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);
...
```

A.6.2.99 function wolfTLSv1_3_client_method_ex

```
WOLFSSL_METHOD * wolfTLSv1_3_client_method_ex(
    void * heap
)
```

この関数は、アプリケーションがクライアントであることを示すために使用され、TLS 1.3 プロトコルのみをサポートします。この関数は、wolfSSL_CTX_new() を使用して SSL / TLS コンテキストを作成するときに使用される新しい Wolfssl_method 構造体のメモリを割り当てて初期化します。

Parameters:

- **ヒープ静的メモリ割り当て中に静的メモリ割り当て器が使用するバッファへのポインタを使用します。**

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: 新しく作成された wWOLFSSL_METHOS 構造体へのポインタを返します。

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);
...
```

A.6.2.100 function wolfTLSv1_3_server_method

```
WOLFSSL_METHOD * wolfTLSv1_3_server_method(
    void
)
```

この関数は、アプリケーションがサーバーであることを示すために使用され、TLS 1.3 プロトコルのみをサポートします。この関数は、wolfSSL_CTX_new() を使用して SSL / TLS コンテキストを作成するときに使用される新しい Wolfssl_method 構造体のメモリを割り当てて初期化します。

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: 新しく作成された wWOLFSSL_METHOS 構造体へのポインタを返します。

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);
...
```

A.6.2.101 function wolfTLSv1_3_client_method

```
WOLFSSL_METHOD * wolfTLSv1_3_client_method(
    void
)
```

この関数は、アプリケーションがクライアントであることを示すために使用され、TLS 1.3 プロトコルのみをサポートします。この関数は、wolfSSL_CTX_new() を使用して SSL / TLS コンテキストを作成するときに使用される新しい Wolfssl_method 構造体のメモリを割り当てて初期化します。

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: 新しく作成された wWOLFSSL_METHOS 構造体へのポインタを返します。

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);
...
```

A.6.2.102 function wolfTLSv1_3_method_ex

```
WOLFSSL_METHOD * wolfTLSv1_3_method_ex(
    void * heap
)
```

この関数は、まだどちらの側(サーバ/クライアント)を決定していないことを除いて、Wolftlsv1_3_client_method と同様の wolfssl_method を返します。

Parameters:

- **ヒープ静的メモリ割り当て中に静的メモリ割り当て器が使用するバッファへのポインタを使用します。**

See:

- wolfSSL_new
- wolfSSL_free

Return: WOLFSSL_METHOD 成功した作成では、wolfssl_method ポインタを返します

Example

```
WOLFSSL* ctx;
ctx  = wolfSSL_CTX_new(wolfTLSv1_3_method_ex(NULL));
// check ret value
```

A.6.2.103 function wolfTLSv1_3_method

```
WOLFSSL_METHOD * wolfTLSv1_3_method(
    void
)
```

この関数は、まだどちらの側(サーバ/クライアント)を決定していないことを除いて、Wolftlsv1_3_client_method と同様の wolfssl_method を返します。

See:

- wolfSSL_new
- wolfSSL_free

Return: WOLFSSL_METHOD 成功した作成では、wolfssl_method ポインタを返します

Example

```
WOLFSSL* ctx;
ctx  = wolfSSL_CTX_new(wolfTLSv1_3_method());
// check ret value
```

A.6.2.104 function wolfSSL_CTX_set_client_cert_type

```
int wolfSSL_CTX_set_client_cert_type(
    WOLFSSL_CTX * ctx,
    const char * buf,
    int len
)
```

この関数はクライアント側で呼び出される場合には、サーバー側に Certificate メッセージで送信できる証明書タイプを設定します。サーバー側で呼び出される場合には、受入れ可能なクライアント証明書タイプを設定します。Raw Public Key 証明書を送受信したい場合にはこの関数を使って証明書タイプを設定しなければなりません。設定する証明書タイプは優先度順に格納したバイト配列として渡します。設定するバッファアドレスに NULL を渡すか、あるいはバッファサイズに 0 を渡すと規定値にもどすことができます。規定値は X509 証明書（WOLFSSL_CERT_TYPE_X509）のみを扱う設定となっています。

Parameters:

- **ctx** wolfssl_ctx コンテキストポインタ
- **ctype** 証明書タイプを格納したバッファへのポインタ
- **len** 証明書タイプを格納したバッファのサイズ（バイト数）*Example*

```
int ret;
WOLFSSL_CTX* ctx;
char ctype[] = {WOLFSSL_CERT_TYPE_RPK, WOLFSSL_CERT_TYPE_X509};
int len = sizeof(ctype)/sizeof(byte);
...

ret = wolfSSL_CTX_set_client_cert_type(ctx, ctype, 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 成功
- BAD_FUNC_ARG ctx として NULL を渡した、あるいは不正な証明書タイプを指定した、あるいは MAX_CLIENT_CERT_TYPE_CNT 以上のバッファサイズを指定した、あるいは指定の証明書タイプに重複がある

A.6.2.105 function wolfSSL_CTX_set_server_cert_type

```
int wolfSSL_CTX_set_server_cert_type(
    WOLFSSL_CTX * ctx,
    const char * buf,
    int len
)
```

この関数はサーバー側で呼び出される場合には、クライアント側に Certificate メッセージで送信できる証明書タイプを設定します。クライアント側で呼び出される場合には、受入れ可能なサーバー証明書タイプを設定します。Raw Public Key 証明書を送受信したい場合にはこの関数を使って証明書タイプを設定しなければなりません。設定する証明書タイプは優先度順に格納したバイト配列として渡します。設定するバッファアドレスに NULL を渡すか、あるいはバッファサイズに 0 を渡すと規定値にもどすことができます。規定値は X509 証明書（WOLFSSL_CERT_TYPE_X509）のみを扱う設定となっています。

Parameters:

- **ctx** wolfssl_ctx コンテキストポインタ
- **ctype** 証明書タイプを格納したバッファへのポインタ
- **len** 証明書タイプを格納したバッファのサイズ（バイト数）*Example*

```
int ret;
WOLFSSL_CTX* ctx;
char ctype[] = {WOLFSSL_CERT_TYPE_RPK, WOLFSSL_CERT_TYPE_X509};
int len = sizeof(ctype)/sizeof(byte);
...

ret = wolfSSL_CTX_set_server_cert_type(ctx, ctype, 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 成功
- BAD_FUNC_ARG ctx として NULL を渡した、あるいは不正な証明書タイプを指定した、あるいは MAX_SERVER_CERT_TYPE_CNT 以上のバッファサイズを指定した、あるいは指定の証明書タイプに重複がある

A.6.2.106 function wolfSSL_set_client_cert_type

```
int wolfSSL_set_client_cert_type(
    WOLFSSL * ssl,
    const char * buf,
    int len
)
```

この関数はクライアント側で呼び出される場合には、サーバー側に Certificate メッセージで送信できる証明書タイプを設定します。サーバー側で呼び出される場合には、受入れ可能なクライアント証明書タイプを設定します。Raw Public Key 証明書を送受信したい場合にはこの関数を使って証明書タイプを設定しなければなりません。設定する証明書タイプは優先度順に格納したバイト配列として渡します。設定するバッファアドレスに NULL を渡すか、あるいはバッファサイズに 0 を渡すと規定値にもどすことができます。規定値は X509 証明書（WOLFSSL_CERT_TYPE_X509）のみを扱う設定となっています。

Parameters:

- **ssl** WOLFSSL 構造体へのポインタ
- **ctype** 証明書タイプを格納したバッファへのポインタ
- **len** 証明書タイプを格納したバッファのサイズ（バイト数）*Example*

```
int ret;
WOLFSSL* ssl;
char ctype[] = {WOLFSSL_CERT_TYPE_RPK, WOLFSSL_CERT_TYPE_X509};
```

```
int len = sizeof(ctype)/sizeof(byte);
...

ret = wolfSSL_set_client_cert_type(ssl, ctype, 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 成功
- BAD_FUNC_ARG ssl として NULL を渡した、あるいは不正な証明書タイプを指定した、あるいは MAX_CLIENT_CERT_TYPE_CNT 以上のバッファサイズを指定した、あるいは指定の証明書タイプに重複がある

A.6.2.107 function wolfSSL_set_server_cert_type

```
int wolfSSL_set_server_cert_type(
    WOLFSSL * ssl,
    const char * buf,
    int len
)
```

この関数はサーバー側で呼び出される場合には、クライアント側に Certificate メッセージで送信できる証明書タイプを設定します。クライアント側で呼び出される場合には、受入れ可能なサーバー証明書タイプを設定します。Raw Public Key 証明書を送受信したい場合にはこの関数を使って証明書タイプを設定しなければなりません。設定する証明書タイプは優先度順に格納したバイト配列として渡します。設定するバッファアドレスに NULL を渡すか、あるいはバッファサイズに 0 を渡すと規定値にもどすことができます。規定値は X509 証明書（WOLFSSL_CERT_TYPE_X509）のみを扱う設定となっています。

Parameters:

- **ssl** WOLFSSL 構造体へのポインタ
- **ctype** 証明書タイプを格納したバッファへのポインタ
- **len** 証明書タイプを格納したバッファのサイズ（バイト数）*Example*

```
int ret;
WOLFSSL* ssl;
char ctype[] = {WOLFSSL_CERT_TYPE_RPK, WOLFSSL_CERT_TYPE_X509};
int len = sizeof(ctype)/sizeof(byte);
...

ret = wolfSSL_set_server_cert_type(ssl, ctype, 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 成功

- BAD_FUNC_ARG ctx として NULL を渡した、あるいは不正な証明書タイプを指定した、あるいは MAX_SERVER_CERT_TYPE_CNT 以上のバッファサイズを指定した、あるいは指定の証明書タイプに重複がある

A.6.2.108 function wolfSSL_GetCookieCtx

```
void * wolfSSL_GetCookieCtx(
    WOLFSSL * ssl
)
```

この関数は、WolfSSL 構造の IOCB_COOKIECTX メンバーを返します。

See:

- wolfSSL_SetCookieCtx
- wolfSSL_CTX_SetGenCookie

Return:

- pointer この関数は、iocb_cookiectx に格納されている void ポインタ値を返します。
- NULL WolfSSL 構造体が NULL の場合 *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
}
```

A.6.2.109 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
)
```

この関数は、WolfSSL が WolfSSL_ISOTP でコンパイルされている場合に使用する場合は、WolfSSL の場合は ISO-TP コンテキストを設定します。

Parameters:

- **ssl** wolfssl コンテキスト
- **ctx** ユーザーはこの関数が初期化される ISOTP コンテキストを作成しました
- **recv_fn** ユーザーはバスを受信できます
- **send_fn** ユーザーはバスを送ることができます
- **delay_fn** ユーザーマイクロ秒の粒度遅延関数
- **receive_delay** 各 CAN バスパケットを遅らせるためのマイクロ秒のセット数
- **receive_buffer** ユーザーがデータを受信するためのバッファーが提供され、ISOTP_DEFAULT_BUFFER_SIZE バイトに割り当てられていることをお勧めします。
- **receive_buffer_size** - receive_buffer のサイズ *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);
```

Return: 0 成功すると、故障の wolfssl_cbio_err_general

A.7 wolfSSL Error Handling and Reporting

A.7.1 Functions

	Name
int	**wolfSSL_Debugging_ON**(void) ビルド時にロギングが有効になっている場合、この関数は実行時にロギングをオンにします。ビルド時にログ記録を有効にするには–enable-debug または debug_wolfssl を定義します。
void	**wolfSSL_Debugging_OFF**(void) この関数はランタイムロギングメッセージをオフにします。彼らがすでに消えている場合は、行動はとられません。
int	**wolfSSL_get_error**(WOLFSSL * ssl, int ret) この関数は、直前の API 関数呼び出し（wolfssl_connect、wolfssl_accept、wolfssl_read、wolfssl_write など）がエラーコード（SSL_FAILURE）を呼び出した理由を表す一意のエラーコードを返します。直前の関数の戻り値は、ret を介して wolfSSL_get_error に渡されます。wolfSSL_get_error は一意のエラーコードを返します。wolfSSL_err_error_string() を呼び出して人間が読めるエラー文字列を取得することができます。詳細については、wolfSSL_err_error_string() を参照してください。
void	**wolfSSL_load_error_strings**(void) この機能は OpenSSL API（SSL_load_error_string）との互換性の目的みで提供してあり処理は行いません。
char *	**wolfSSL_ERR_error_string**(unsigned long errNumber, char * data) この関数は、wolfSSL_get_error() によって返されたエラーコードをより人間が読めるエラー文字列に変換します。引数 errNumber は、wolfSSL_get_error() によって返され、引数 data はエラー文字列が配置されるバッファへのポインタです。MAX_ERROR_SZ で定義されているように、データの最大長はデフォルトで 80 文字です。これは wolfssl/wolfcrypt/error.h で定義されています。

	Name
void	**wolfSSL_ERR_error_string_n**(unsigned long e, char * buf, unsigned long sz) この関数は、wolfssl_err_error_string() のバッファのサイズを指定するバージョンです。ここで、引数 len は引数 buf に書き込まれ得る最大文字数を指定します。wolfSSL_err_error_string() と同様に、この関数は wolfSSL_get_error() から返されたエラーコードをより人間が読めるエラー文字列に変換します。人間が読める文字列は buf に置かれます。
void	**wolfSSL_ERR_print_errors_fp**(XFILE fp, int err) この関数は、wolfSSL_get_error() によって返されたエラーコードをより多くの人間が読めるエラー文字列に変換し、その文字列を出力ファイルに印刷します。ERR は、WOLFSSL_GET_ERROR() によって返され、FP がエラー文字列が配置されるファイルであるエラーコードです。
void	**wolfSSL_ERR_print_errors_cb**(int(*)(const char str, size_t len, void *u) cb, void* u) この関数は提供されたコールバックを使用してエラー報告を処理します。コールバック関数はエラー回線ごとに実行されます。文字列、長さ、および userdata はコールバックパラメータに渡されます。
int	**wolfSSL_want_read**(WOLFSSL *) この関数は、wolfSSL_get_error() を呼び出して ssl_error_want_read を取得するのと似ています。基礎となるエラー状態が SSL_ERROR_WANT_READ の場合、この関数は 1 を返しますが、それ以外の場合は 0 です。
int	**wolfSSL_want_write**(WOLFSSL *) この関数は、wolfSSL_get_error() を呼び出し、RETURS の SSL_ERROR_WANT_WRITE を取得するのと同じです。基礎となるエラー状態が SSL_ERROR_WANT_WRITE の場合、この関数は 1 を返しますが、それ以外の場合は 0 です。
unsigned long	**wolfSSL_ERR_peek_last_error**(void) この関数は、wolfssl_Error に遭遇した最後のエラーの絶対値を返します。

A.7.2 Functions Documentation

A.7.2.1 function wolfSSL_Debugging_ON

```
int wolfSSL_Debugging_ON(
    void
)
```

ビルド時にロギングが有効になっている場合、この関数は実行時にロギングをオンにします。ビルド時にログ記録を有効にするには–enable-debug または debug_wolfssl を定義します。

See:

- wolfSSL_Debugging_OFF
- wolfSSL_SetLoggingCb

Return:

- 0 成功すると。
- NOT_COMPILED_IN このビルドに対してロギングが有効になっていない場合は返されるエラーです。
  Example

```
wolfSSL_Debugging_ON();
```

A.7.2.2 function wolfSSL_Debugging_OFF

```
void wolfSSL_Debugging_OFF(
    void
)
```

この関数はランタイムロギングメッセージをオフにします。彼らがすでに消えている場合は、行動はとられません。

See:

- wolfSSL_Debugging_ON
- wolfSSL_SetLoggingCb

Return: none いいえ返します。*Example*

```
wolfSSL_Debugging_OFF();
```

A.7.2.3 function wolfSSL_get_error

```
int wolfSSL_get_error(
    WOLFSSL * ssl,
    int ret
)
```

この関数は、直前の API 関数呼び出し（wolfssl_connect、wolfssl_accept、wolfssl_read、wolfssl_write など）がエラーコード（SSL_FAILURE）を呼び出した理由を表す一意のエラーコードを返します。直前の関数の戻り値は、ret を介して wolfSSL_get_error に渡されます。wolfSSL_get_error は一意のエラーコードを返します。wolfSSL_err_error_string() を呼び出して人間が読めるエラー文字列を取得することができます。詳細については、wolfSSL_err_error_string() を参照してください。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ

See:

- wolfSSL_ERR_error_string
- wolfSSL_ERR_error_string_n
- wolfSSL_ERR_print_errors_fp
- wolfSSL_load_error_strings

Return:

- 呼び出し成功時、この関数は、直前の関数が失敗した理由を説明する固有のエラーコードを返します。
- SSL_ERROR_NONE 引数 ret が 0 より大きい場合に返されます。ret が 0 以下の場合、直前の API がエラーコードを返すが実際に発生しなかった場合にこの値を返す場合があります。例としては、引数 sz に 0 を渡して wolfSSL_read() を呼び出す場合に発生します。wolfssl_read() が 0 を戻した場合は通常エラーを示しますが、この場合はエラーは発生していません。従って、wolfSSL_get_error() がその後呼び出された場合、ssl_error_none が返されます。

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);
```

A.7.2.4 function wolfSSL_load_error_strings

```
void wolfSSL_load_error_strings(
    void
)
```

この機能は OpenSSL API（SSL_load_error_string）との互換性の目的みで提供してあり処理は行いません。

Parameters:

- **なし** *Example*

```
wolfSSL_load_error_strings();
```

See:

- wolfSSL_get_error
- wolfSSL_ERR_error_string
- wolfSSL_ERR_error_string_n
- wolfSSL_ERR_print_errors_fp
- wolfSSL_load_error_strings

Return: なし

A.7.2.5 function wolfSSL_ERR_error_string

```
char * wolfSSL_ERR_error_string(
    unsigned long errNumber,
    char * data
)
```

この関数は、wolfSSL_get_error() によって返されたエラーコードをより人間が読めるエラー文字列に変換します。引数 errNumber は、wolfSSL_get_error() によって返され、引数 data はエラー文字列が配置されるバッファへのポインタです。MAX_ERROR_SZ で定義されているように、データの最大長はデフォルトで 80 文字です。これは wolfssl/wolfcrypt/error.h で定義されています。

Parameters:

- **errNumber** wolfSSL_get_error()によって返されたエラーコード。
- **data** 人間が読めるエラー文字列を格納したバッファへのポインタ

See:

- wolfSSL_get_error
- wolfSSL_ERR_error_string_n
- wolfSSL_ERR_print_errors_fp
- wolfSSL_load_error_strings

Return:

- success 正常に完了すると、この関数は data に返されるのと同じ文字列を返します。
- failure 失敗すると、この関数は適切な障害理由、MSG を持つ文字列を返します。

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);
```

A.7.2.6 function wolfSSL_ERR_error_string_n

```
void wolfSSL_ERR_error_string_n(
    unsigned long e,
    char * buf,
    unsigned long sz
)
```

この関数は、wolfssl_err_error_string() のバッファのサイズを指定するバージョンです。ここで、引数 len は引数 buf に書き込まれ得る最大文字数を指定します。wolfSSL_err_error_string() と同様に、この関数は wolfSSL_get_error() から返されたエラーコードをより人間が読めるエラー文字列に変換します。人間が読める文字列は buf に置かれます。

Parameters:

- **e** wolfSSL_get_error()によって返されたエラーコード。
- **buff** e と一致する人間が読めるエラー文字列を含む出力バッファ。
- **len** 出力バッファのサイズ

See:

- wolfSSL_get_error
- wolfSSL_ERR_error_string
- wolfSSL_ERR_print_errors_fp
- wolfSSL_load_error_strings

Return: なし

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);
```

A.7.2.7 function wolfSSL_ERR_print_errors_fp

```
void wolfSSL_ERR_print_errors_fp(
    XFILE fp,
    int err
)
```

この関数は、wolfSSL_get_error() によって返されたエラーコードをより多くの人間が読めるエラー文字列に変換し、その文字列を出力ファイルに印刷します。ERR は、WOLFSSL_GET_ERROR() によって返され、FP がエラー文字列が配置されるファイルであるエラーコードです。

Parameters:

- **fp** に書き込まれる人間が読めるエラー文字列の出力ファイル。
- **err** wolfSSL_get_error()で返されるエラーコード。

See:

- wolfSSL_get_error
- wolfSSL_ERR_error_string
- wolfSSL_ERR_error_string_n
- wolfSSL_load_error_strings

Return: なし

Example

```
int err = 0;
WOLFSSL* ssl;
FILE* fp = ...
...
err = wolfSSL_get_error(ssl, 0);
wolfSSL_ERR_print_errors_fp(fp, err);
```

A.7.2.8 function wolfSSL_ERR_print_errors_cb

```
void wolfSSL_ERR_print_errors_cb(
    int(*)(const char *str, size_t len, void *u) cb,
    void * u
)
```

この関数は提供されたコールバックを使用してエラー報告を処理します。コールバック関数はエラー回線ごとに実行されます。文字列、長さ、および userdata はコールバックパラメータに渡されます。

Parameters:

- **cb** コールバック関数
- **u** コールバック関数に渡される userdata

See:

- wolfSSL_get_error
- wolfSSL_ERR_error_string
- wolfSSL_ERR_error_string_n
- wolfSSL_load_error_strings

Return: なし

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);
```

A.7.2.9 function wolfSSL_want_read

```
int wolfSSL_want_read(
    WOLFSSL *
)
```

この関数は、wolfSSL_get_error() を呼び出して ssl_error_want_read を取得するのと似ています。基礎となるエラー状態が SSL_ERROR_WANT_READ の場合、この関数は 1 を返しますが、それ以外の場合は 0 です。

See:

- wolfSSL_want_write
- wolfSSL_get_error

Return:

- 1 WOLFSSL_GET_ERROR() は SSL_ERROR_WANT_READ を返し、基礎となる I / O には読み取り可能なデータがあります。
- 0 SSL_ERROR_WANT_READ エラー状態はありません。

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)
}
```

A.7.2.10 function wolfSSL_want_write

```
int wolfSSL_want_write(
    WOLFSSL *
)
```

この関数は、wolfSSL_get_error() を呼び出し、RETURS の SSL_ERROR_WANT_WRITE を取得するのと同じです。基礎となるエラー状態が SSL_ERROR_WANT_WRITE の場合、この関数は 1 を返しますが、それ以外の場合は 0 です。

See:

- wolfSSL_want_read
- wolfSSL_get_error

Return:

- 1 WOLFSSL_GET_ERROR() は SSL_ERROR_WANT_WRITE を返します。基礎となる I / O は、基礎となる SSL 接続で進行状況を行うために書き込まれるデータを必要とします。
- 0 ssl_error_want_write エラー状態はありません。

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)
}
```

A.7.2.11 function wolfSSL_ERR_peek_last_error

```
unsigned long wolfSSL_ERR_peek_last_error(
    void
)
```

この関数は、wolfssl_Error に遭遇した最後のエラーの絶対値を返します。

See: wolfSSL_ERR_print_errors_fp

Return: error 最後のエラーの絶対値を返します。

Example

```
unsigned long err;
...
err = wolfSSL_ERR_peek_last_error();
// inspect err value
```

A.8 wolfSSL Initialization/Shutdown

A.8.1 Functions

	Name
int	**wolfSSL_shutdown**(WOLFSSL *) この関数は、引数 ssl の SSL セッションに対してアクティブな SSL/TLS 接続をシャットダウンします。この関数は、ピアに “Close Notify” アラートを送信しようとします。呼び出し側アプリケーションは、Peer がその “Close Notify” アラートを応答として送信してくるのを待つか、または wolfSSL_shutdown から呼び出しが戻った時点で（リソースを保存するために）下層の接続を切断するのを待つことができます。どちらのオプションも TLS 仕様で許されています。シャットダウンした後に下層の接続を再び別のセッションで使用する予定ならば、ピア間で同期を保つために完全な 2 方向のシャットダウン手順を実行する必要があります。wolfSSL_shutdown() は、ブロックとノンブロッキング I/O の両方で動作します。下層の I/O がノンブロッキングの場合、wolfSSL_shutdown() が要求を満たすことができなかった場合、wolfSSL_shutdown() はエラーを返します。この場合、wolfSSL_get_error() への呼び出しは SSL_ERROR_WANT_READ または SSL_ERROR_WANT_WRITE のいずれかを生成します。その結果、下層の I/O が準備ができたら、呼び出し側プロセスは wolfSSL_shutdown() への呼び出しを繰り返す必要があります。
int	**wolfSSL_SetServerID**(WOLFSSL * ssl, const unsigned char * id, int len, int newSession) この関数はクライアントセッションをサーバー ID と関連付けます。引数 newSession がオンの場合、既存のセッションは再利用されません。
int	**wolfSSL_library_init**(void) この関数は wolfSSL_CTX_new() 内で内部的に呼び出されます。この関数は wolfSSL_Init() のラッパーで、wolfSSL が OpenSSL 互換層でコンパイルされたときの OpenSSL API（ssl_library_init）との互換性の為に存在します。wolfSSL_init() は、より一般的に使用されている wolfSSL 初期化機能です。

	Name
int	**wolfSSL_get_shutdown**(const WOLFSSL * ssl) この関数は、Options 構造体の closeNotify または connReset または sentNotify メンバーのシャットダウン条件をチェックします。Options 構造体は WOLFSSL 構造体内にあります。
int	**wolfSSL_is_init_finished**(WOLFSSL * ssl) この関数は、接続が確立されているかどうかを確認します。
int	**wolfSSL_Init**(void) 使用するために WolfSSL ライブラリを初期化します。アプリケーションごとに 1 回、その他のライブラリへの呼び出しの前に呼び出す必要があります。
int	**wolfSSL_Cleanup**(void) さらなる使用から WOLFSSL ライブラリを初期化します。ライブラリによって使用されるリソースを解放しますが、呼び出される必要はありません。
int	**wolfSSL_SetMinVersion**(WOLFSSL * ssl, int version) この関数は、許可されている最小のダウングレードバージョンを設定します。接続が（wolfsslv23_client_method または wolfsslv23_server_method）を使用して、接続がダウングレードできる場合にのみ適用されます。
int	**wolfSSL_ALPN_GetProtocol**(WOLFSSL * ssl, char ** protocol_name, unsigned short * size) この関数は、サーバーによって設定されたプロトコル名を取得します。
int	**wolfSSL_ALPN_GetPeerProtocol**(WOLFSSL * ssl, char ** list, unsigned short * listSz) この関数は、alpn_client_list データを SSL オブジェクトからバッファにコピーします。
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) この関数は CR と SR の値をコピーしてから WC_PRF（疑似ランダム関数）に渡し、その値を返します。
int	**wolfSSL_preferred_group**(WOLFSSL * ssl) この関数は、クライアントが TLS v1.3 ハンドシェイクで使用することを好む鍵交換グループを返します。この情報を完了した後にこの機能を呼び出して、サーバーがどのグループが予想されるようにこの情報が将来の接続で使用できるようになるかを決定するために、この情報が将来の接続で鍵交換のための鍵ペアを事前生成することができます。

A.8.2 Functions Documentation

A.8.2.1 function wolfSSL_shutdown

```
int wolfSSL_shutdown(
    WOLFSSL *
)
```

この関数は、引数 ssl の SSL セッションに対してアクティブな SSL/TLS 接続をシャットダウンします。この関数は、ピアに “Close Notify” アラートを送信しようとします。呼び出し側アプリケーションは、Peer がその “Close Notify” アラートを応答として送信してくるのを待つか、または wolfSSL_shutdown から呼び出しが戻った時点で（リソースを保存するために）下層の接続を切断するのを待つことができます。どちらのオプションも TLS 仕様で許されています。シャットダウンした後に下層の接続を再び別のセッションで使用する予定ならば、ピア間で同期を保つために完全な 2 方向のシャットダウン手順を実行する必要があります。wolfSSL_shutdown() は、ブロックとノンブロッキング I/O の両方で動作します。下層の I/O がノンブロッキングの場合、wolfSSL_shutdown() が要求を満たすことができなかった場合、wolfSSL_shutdown() はエラーを返します。この場合、wolfSSL_get_error() への呼び出しは SSL_ERROR_WANT_READ または SSL_ERROR_WANT_WRITE のいずれかを生成します。その結果、下層の I/O が準備ができたら、呼び出し側プロセスは wolfSSL_shutdown() への呼び出しを繰り返す必要があります。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ

See:

- wolfSSL_free
- wolfSSL_CTX_free

Return:

- SSL_SUCCESS 成功時に返されます。
- SSL_SHUTDOWN_NOT_DONE シャットダウンが終了していない場合に返され、関数を再度呼び出す必要があります。
- SSL_FATAL_ERROR 失敗したときに返されます。より具体的なエラーコードは wolfSSL_get_error() を呼び出します。

Example

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

int ret = 0;
WOLFSSL* ssl = 0;
...
ret = wolfSSL_shutdown(ssl);
if (ret != 0) {
    // failed to shut down SSL connection
}
```

A.8.2.2 function wolfSSL_SetServerID

```
int wolfSSL_SetServerID(
    WOLFSSL * ssl,
    const unsigned char * id,
    int len,
    int newSession
)
```

この関数はクライアントセッションをサーバー ID と関連付けます。引数 newSession がオンの場合、既存のセッションは再利用されません。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ。
- **id** WOLFSSL_SESSION 構造体の ServerID メンバーにコピーされるサーバー ID データへのポインタ。
- **len** サーバー ID データのサイズ
- **newSession** セッションを再利用するか否かを指定するフラグ。オンの場合、既存のセッションは再利用されません。

See: wolfSSL_set_session

Return:

- SSL_SUCCESS 関数がエラーなしで実行された場合に返されます。
- BAD_FUNC_ARG 引数 ssl または引数 id が NULL の場合、または引数 len がゼロ以下の場合に返されます。

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
}
```

A.8.2.3 function wolfSSL_library_init

```
int wolfSSL_library_init(
    void
)
```

この関数は wolfSSL_CTX_new() 内で内部的に呼び出されます。この関数は wolfSSL_Init() のラッパーで、wolfSSL が OpenSSL 互換層でコンパイルされたときの OpenSSL API（ssl_library_init）との互換性の為に存在します。wolfSSL_init() は、より一般的に使用されている wolfSSL 初期化機能です。

See:

- wolfSSL_Init
- wolfSSL_Cleanup

Return:

- SSL_SUCCESS 成功した場合に返されます。に返されます。
- SSL_FATAL_ERROR 失敗したときに返されます。

Example

```
int ret = 0;
ret = wolfSSL_library_init();
if (ret != SSL_SUCCESS) {
    failed to initialize wolfSSL
}
...
```

A.8.2.4 function wolfSSL_get_shutdown

```
int wolfSSL_get_shutdown(
    const WOLFSSL * ssl
)
```

この関数は、Options 構造体の closeNotify または connReset または sentNotify メンバーのシャットダウン条件をチェックします。Options 構造体は WOLFSSL 構造体内にあります。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ

See: wolfSSL_SESSION_free

Return:

- 1 SSL_SENT_SHUTDOWN が返されます。
- 2 SSL_RECEIVED_SHUTDOWN が返されます。

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.
}
```

A.8.2.5 function wolfSSL_is_init_finished

```
int wolfSSL_is_init_finished(
    WOLFSSL * ssl
)
```

この関数は、接続が確立されているかどうかを確認します。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ

See:

- wolfSSL_set_accept_state
- wolfSSL_get_keys
- wolfSSL_set_shutdown

Return:

- 0 接続が確立されていない場合、すなわち WolfSSL 構造体が NULL またはハンドシェイクが行われていない場合に返されます。
- 1 接続が確立されていない場合は返されます.WolfSSL 構造体は NULL またはハンドシェイクが行われていません。

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
}
```

A.8.2.6 function wolfSSL_Init

```
int wolfSSL_Init(
    void
)
```

使用するために WolfSSL ライブラリを初期化します。アプリケーションごとに 1 回、その他のライブラリへの呼び出しの前に呼び出す必要があります。

See: wolfSSL_Cleanup

Return:

- SSL_SUCCESS 成功した場合に返されます。、通話が戻ります。
- BAD_MUTEX_E 返される可能性があるエラーです。

Example

```
int ret = 0;
ret = wolfSSL_Init();
if (ret != SSL_SUCCESS) {
    failed to initialize wolfSSL library
}
```

A.8.2.7 function wolfSSL_Cleanup

```
int wolfSSL_Cleanup(
    void
)
```

さらなる使用から WOLFSSL ライブラリを初期化します。ライブラリによって使用されるリソースを解放しますが、呼び出される必要はありません。

See: wolfSSL_Init

Return: SSL_SUCCESS エラーを返しません。

Example

```
wolfSSL_Cleanup();
```

A.8.2.8 function wolfSSL_SetMinVersion

```
int wolfSSL_SetMinVersion(
    WOLFSSL * ssl,
    int version
)
```

この関数は、許可されている最小のダウングレードバージョンを設定します。接続が（wolfsslv23_client_method または wolfsslv23_server_method）を使用して、接続がダウングレードできる場合にのみ適用されます。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ

See: SetMinVersionHelper

Return:

- SSL_SUCCESS この関数とそのサブルーチンがエラーなしで実行された場合に返されます。
- BAD_FUNC_ARG SSL オブジェクトが NULL の場合に返されます。サブルーチンでは、良いバージョンが一致しない場合、このエラーはスローされます。

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
}
```

A.8.2.9 function wolfSSL_ALPN_GetProtocol

```
int wolfSSL_ALPN_GetProtocol(
    WOLFSSL * ssl,
    char ** protocol_name,
    unsigned short * size
)
```

この関数は、サーバーによって設定されたプロトコル名を取得します。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WolfSSL 構造へのポインタ。
- **protocol_name** プロトコル名を表す CHAR へのポインタは、ALPN 構造に保持されます。

See:

- TLSX_ALPN_GetRequest
- TLSX_Find

Return:

- SSL_SUCCESS エラーが投げられていない正常な実行に戻りました。
- SSL_FATAL_ERROR 拡張子が見つからなかった場合、またはピアとプロトコルが一致しなかった場合に返されます。2 つ以上のプロトコル名が受け入れられている場合は、スローされたエラーもあります。
- SSL_ALPN_NOT_FOUND ピアとプロトコルの一致が見つからなかったことを示す返されました。
- BAD_FUNC_ARG 関数に渡された null 引数があった場合に返されます。

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
}
```

A.8.2.10 function wolfSSL_ALPN_GetPeerProtocol

```
int wolfSSL_ALPN_GetPeerProtocol(
```

```
    WOLFSSL * ssl,
    char ** list,
    unsigned short * listSz
)
```

この関数は、alpn_client_list データを SSL オブジェクトからバッファにコピーします。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WolfSSL 構造へのポインタ。
- **list** バッファへのポインタ。SSL オブジェクトからのデータがコピーされます。

See: wolfSSL_UseALPN

Return:

- SSL_SUCCESS 関数がエラーなしで実行された場合に返されます。SSL オブジェクトの ALPN_CLIENT_LIST メンバーが LIST パラメータにコピーされました。
- BAD_FUNC_ARG list または listsz パラメーターが null の場合に返されます。
- BUFFER_ERROR リストバッファに問題がある場合は（NULL またはサイズが 0 の場合）に問題がある場合に返されます。
- MEMORY_ERROR メモリを動的に割り当てる問題がある場合に返されます。

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
}
```

A.8.2.11 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
)
```

この関数は CR と SR の値をコピーしてから WC_PRF（疑似ランダム関数）に渡し、その値を返します。

Parameters:

- **ms** マスターシークレットはアレイ構造に保持されています。
- **msLen** マスターシークレットの長さ。

- **pms** マスター前の秘密はアレイ構造に保持されています。
- **pmsLen** マスタープレマスターシークレットの長さ。
- **cr** クライアントのランダム
- **sr** サーバーのランダムです。
- **tls1_2** バージョンが少なくとも TLS バージョン 1.2 であることを意味します。

See:

- wc_PRF
- MakeTlsMasterSecret

Return:

- 0 成功した
- BUFFER_E バッファのサイズにエラーが発生した場合に返されます。
- MEMORY_E サブルーチンが動的メモリを割り当てることができなかった場合に返されます。

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;

}
```

A.8.2.12 function wolfSSL_preferred_group

```
int wolfSSL_preferred_group(
    WOLFSSL * ssl
)
```

この関数は、クライアントが TLS v1.3 ハンドシェイクで使用することを好む鍵交換グループを返します。この情報を完了した後にこの機能を呼び出して、サーバーがどのグループが予想されるようにこの情報が将来の接続で使用できるようになるかを決定するために、この情報が将来の接続で鍵交換のための鍵ペアを事前生成することができます。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ。

See:

- wolfSSL_UseKeyShare
- wolfSSL_CTX_set_groups
- wolfSSL_set_groups
- wolfSSL_CTX_set1_groups_list
- wolfSSL_set1_groups_list

Return:

- BAD_FUNC_ARG ssl が NULL の場合、または TLS v1.3 を使用していない場合。

- SIDE_ERROR サーバーで呼び出された場合。
- NOT_READY_ERROR ハンドシェイクが完了する前に呼び出された場合。

Example

```
int ret;
int group;
WOLFSSL* ssl;
...
ret = wolfSSL_CTX_set1_groups_list(ssl)
if (ret < 0) {
    // failed to get group
}
group = ret;
```

B WolfCrypt API リファレンス

B.1 ASN.1

B.1.1 Functions

	Name
int	**wc_InitCert**(Cert *) この関数は Cert 構造体をデフォルトの値で初期化します。デフォルトのオプション：version = 3（0x2）、sigtype = sha_with_rsa、issuer = 空白、dayValid = 500、selfsigned = 1（true）発行者としての件名 = 空白
Cert *	**wc_CertNew**(void * heap) この関数は証明書操作の為に新たな Cert 構造体を割り当てます。割り当てた Cert 構造体はこの関数内で初期化されるので、wc_InitCert() を呼び出す必要はありません。アプリケーションがこの Cert 構造体の使用を終了する際には wc_CertFree() を呼び出す必要があります。
void	**wc_CertFree**(Cert * cert) この関数は wc_CertNew() で確保された Cert 構造体を解放します。
int	**wc_MakeCert**(Cert * cert, byte * derBuffer, word32 derSz, RsaKey * rsaKey, ecc_key * eccKey, WC_RNG * rng)CA 署名付き証明書を作成するために使用されます。サブジェクト情報を入力した後に呼び出す必要があります。この関数は、証明書入力から X.509v3 RSA または ECC 証明書を作成し derBuffer に書き込みます。証明書を生成するための RsaKey または EccKey のいずれかを引数として取ります。この関数が呼び出される前に、証明書を wc_InitCert で初期化する必要があります。
int	**wc_MakeCertReq**(Cert * cert, byte * derBuffer, word32 derSz, RsaKey * rsaKey, ecc_key * eccKey) この関数は、入力された Cert 構造体を使用して証明書署名要求を作成し derBuffer に書き込みます。証明書要求の生成には RsaKey または EccKey のいずれかの鍵を受け取り使用します。この関数の後に、署名するために wc_SignCert() を呼び出す必要があります。この関数の使用例については、wolfCrypt テストアプリケーション (./wolfcrypt/test/test.c) を参照してください。
int	**wc_SignCert**(int requestSz, int sigType, byte * derBuffer, word32 derSz, RsaKey * rsaKey, ecc_key * eccKey, WC_RNG * rng) この関数はバッファーの内容に署名し、署名をバッファの最後に追加します。署名の種類を取ります。CA 署名付き証明書を作成する場合は、wc_MakeCert() または wc_MakeCertReq() の後に呼び出す必要があります。

	Name
int	**wc_MakeSelfCert**(Cert * cert, byte * derBuffer, word32 derSz, RsaKey * key, WC_RNG * rng) この関数は、以前の 2 つの関数、wc_MakeCert、および自己署名のための wc_SignCert の組み合わせです（前の関数は CA 要求に使用される場合があります）。証明書を作成してから、それに署名し、自己署名証明書を生成します。
int	**wc_SetIssuer**(Cert * cert, const char * issuerFile) この関数は PEM 形式の issureFile で与えられた発行者を証明書の発行者として設定します。また、その際に、証明書の自己署名プロパティを false に変更します。発行者は証明書の発行者として設定される前に検証されます。この関数は証明書への署名に先立ち呼び出される必要があります。
int	**wc_SetSubject**(Cert * cert, const char * subjectFile) この関数は PEM 形式の subjectFile で与えられた主体者を証明書の主体者として設定します。この関数は証明書への署名に先立ち呼び出される必要があります。
int	**wc_SetSubjectRaw**(Cert * cert, const byte * der, int derSz) この関数は DER 形式でバッファに格納されている Raw-Subject 情報を証明書の Raw-Subject 情報として設定します。この関数は証明書への署名に先立ち呼び出される必要があります。
int	**wc_GetSubjectRaw**(byte ** subjectRaw, Cert * cert) この関数は Cert 構造体から Raw-Subject 情報を取り出します。
int	**wc_SetAltNames**(Cert * cert, const char * file) この関数は引数で与えられた PEM 形式の証明書の主体者の別名を Cert 構造体に設定します。複数のドメインで同一の証明書を使用する際には主体者の別名を付与する機能は有用です。この関数は証明書への署名に先立ち呼び出される必要があります。
int	**wc_SetIssuerBuffer**(Cert * cert, const byte * der, int derSz) この関数は DER 形式でバッファに格納されている発行者を証明書の発行者として設定します。加えて、証明書の事故署名プロパティを false に設定します。この関数は証明書への署名に先立ち呼び出される必要があります。
int	**wc_SetIssuerRaw**(Cert * cert, const byte * der, int derSz) この関数は DER 形式でバッファに格納されている Raw-Issuer 情報を証明書の Raw-Issuer 情報として設定します。この関数は証明書への署名に先立ち呼び出される必要があります。

	Name
int	**wc_SetSubjectBuffer**(Cert * cert, const byte * der, int derSz) この関数は DER 形式でバッファに格納されている主体者を証明書の主体者として設定します。この関数は証明書への署名に先立ち呼び出される必要があります。
int	**wc_SetAltNamesBuffer**(Cert * cert, const byte * der, int derSz) この関数は DER 形式でバッファに格納されている「別名情報」を証明書の「別名情報」として設定します。この機能は複数ドメインを一つの証明書を使ってセキュアにする際に有用です。この関数は証明書への署名に先立ち呼び出される必要があります。
int	**wc_SetDatesBuffer**(Cert * cert, const byte * der, int derSz) この関数は DER 形式でバッファに格納されている「有効期間」情報を証明書の「有効期間」情報として設定します。この関数は証明書への署名に先立ち呼び出される必要があります。
int	**wc_SetAuthKeyIdFromPublicKey**(Cert * cert, RsaKey * rsakey, ecc_key * eckey) この関数は指定された RSA あるいは ECC 公開鍵の一方から得た AKID（認証者鍵 ID）を証明書の AKID として設定します。
int	**wc_SetAuthKeyIdFromCert**(Cert * cert, const byte * der, int derSz) この関数は DER 形式でバッファに格納された証明書から得た AKID(認証者鍵 ID) を証明書の AKID として設定します。
int	**wc_SetAuthKeyId**(Cert * cert, const char * file) この関数は PEM 形式の証明書から得た AKID(認証者鍵 ID) を証明書の AKID として設定します。
int	**wc_SetSubjectKeyIdFromPublicKey**(Cert * cert, RsaKey * rsakey, ecc_key * eckey) この関数は指定された RSA あるいは ECC 公開鍵の一方から得た SKID（主体者鍵 ID）を証明書の SKID として設定します。
int	**wc_SetSubjectKeyId**(Cert * cert, const char * file) この関数は PEM 形式の証明書から得た SKID(主体者鍵 ID) を証明書の SKID として設定します。引数は両方が与えられることが必要です。
int	**wc_PemPubKeyToDer**(const char * fileName, unsigned char * derBuf, int derSz)PEM 形式の鍵ファイルをロードし DER 形式に変換してバッファに出力します。
int	**wc_PubKeyPemToDer**(const unsigned char * pem, int pemSz, unsigned char * buff, int buffSz)PEM 形式の鍵データを DER 形式に変換してバッファに出力し、出力バイト数あるいは負のエラー値を返します。
int	**wc_PemCertToDer**(const char * fileName, unsigned char * derBuf, int derSz) この関数は PEM 形式の証明書を DER 形式に変換し、与えられたバッファに出力します。

	Name
int	**wc_DerToPem**(const byte * der, word32 derSz, byte * output, word32 outputSz, int type) この関数はバッファで与えられた DER 形式の証明書を PEM 形式に変換し、与えられた出力用バッファに出力します。この関数は入力バッファと出力バッファを共用することはできません。両バッファは必ず別のものを用意してください。
int	**wc_DerToPemEx**(const byte * der, word32 derSz, byte * output, word32 outputSz, byte * cipherIno, int type) この関数は DER 形式証明書を入力バッファから読み出し、PEM 形式に変換して出力バッファに出力します。この関数は入力バッファと出力バッファを共用することはできません。両バッファは必ず別のものを用意してください。追加の暗号情報を指定することができます。
int	**wc_EccPrivateKeyDecode**(const byte * input, word32 * inOutIdx, ecc_key * key, word32 inSz) この関数は ECC 秘密鍵を入力バッファから読み込み、解析の後 ecc_key 構造体を作成してそこに鍵を格納します。
int	**wc_EccKeyToDer**(ecc_key * key, byte * output, word32 inLen) この関数は ECC 秘密鍵を DER 形式でバッファに出力します。
int	**wc_EccPublicKeyDecode**(const byte * input, word32 * inOutIdx, ecc_key * key, word32 inSz) この関数は入力バッファの ECC 公開鍵を ASN シーケンスをデコードして取り出します。
int	**wc_EccPublicKeyToDer**(ecc_key * key, byte * output, word32 inLen, int with_AlgCurve) この関数は ECC 公開鍵を DER 形式に変換します。処理したバッファのサイズを返します。変換して得られる DER 形式の ECC 公開鍵は出力バッファに格納されます。AlgCurve フラグの指定により、アルゴリズムと曲線情報をヘッダーに含めることができます。
int	**wc_EccPublicKeyToDer_ex**(ecc_key * key, byte * output, word32 inLen, int with_AlgCurve, int comp) この関数は ECC 公開鍵を DER 形式に変換します。処理したバッファサイズを返します。変換された DER 形式の ECC 公開鍵は出力バッファに格納されます。AlgCurve フラグの指定により、アルゴリズムと曲線情報をヘッダーに含めることができます。comp パラメータは公開鍵を圧縮して出力するか否かを指定します。
word32	**wc_EncodeSignature**(byte * out, const byte * digest, word32 digSz, int hashOID) この関数はデジタル署名をエンコードして出力バッファに出力し、生成された署名のサイズを返します。

	Name
int	**wc_GetCTC_HashOID**(int type) この関数はハッシュタイプに対応したハッシュ OID を返します。例えば、ハッシュタイプが “WC_SHA512” の場合、この関数は “SHA512h” を対応するハッシュ OID として返します。
void	**wc_SetCert_Free**(Cert * cert) この関数はキャッシュされていた Cert 構造体で使用されたメモリとリソースをクリーンアップします。WOLFSSL_CERT_GEN_CACHE が定義されている場合には DecodedCert 構造体が Cert 構造体内部にキャッシュされ、後続する set 系関数の呼び出しの都度 DecodedCert 構造体がパースされることを防ぎます。
int	**wc_GetPkcs8TraditionalOffset**(byte * input, word32 * inOutIdx, word32 sz) この関数は PKCS#8 の暗号化されていないバッファ内部の従来の秘密鍵の開始位置を検出して返します。
int	**wc_CreatePKCS8Key**(byte * out, word32 * outSz, byte * key, word32 keySz, int algoID, const byte * curveOID, word32 oidSz) この関数は DER 形式の秘密鍵を入力とし、RKCS#8 形式に変換します。また、PKCS#12 のシュロ — ディットキーバッグの作成にも使用できます。RFC5208 を参照のこと。
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) この関数は暗号化されていない PKCS#8 の DER 形式の鍵 (例えば wc_CreatePKCS8Key で生成された鍵) を受け取り、PKCS#8 暗号化形式に変換します。結果として得られた暗号化鍵は wc_DecryptPKCS8Key を使って復号できます。RFC5208 を参照してください。
int	**wc_DecryptPKCS8Key**(byte * input, word32 sz, const char * password, int passwordSz) この関数は暗号化された PKCS#8 の DER 形式の鍵を受け取り、復号して PKCS#8 DER 形式に変換します。wc_EncryptPKCS8Key によって行われた暗号化を元に戻します。RFC5208 を参照してください。入力データは復号データによって上書きされます。
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) この関数は従来の DER 形式の鍵を PKCS#8 フォーマットに変換し、暗号化を行います。この処理には wc_CreatePKCS8Key と wc_EncryptPKCS8Key を使用します。

	Name
void	**wc_InitDecodedCert**(struct DecodedCert * cert, const byte * source, word32 inSz, void * heap) この関数は cert 引数で与えられた DecodedCert 構造体を初期化します。DER 形式の証明書を含んでいる source 引数の指すポインタから証明書サイズ inSz の長さを内部に保存します。この関数の後に呼び出される wc_ParseCert によって証明書が解析されます。
int	**wc_ParseCert**(DecodedCert * cert, int type, int verify, void * cm) この関数は DecodedCert 構造体に保存されている DER 形式の証明書を解析し、その構造体に各種フィールドを設定します。DecodedCert 構造体は wc_InitDecodedCert を呼び出して初期化しておく必要があります。この関数はオプションで CertificateManager 構造体へのポインタを受け取り、CA が証明書マネジャーで検索できた場合には、その CA に関する情報も DecodedCert 構造体に追加設定します。
void	**wc_FreeDecodedCert**(struct DecodedCert * cert) この関数は wc_InitDecodedCert で初期化済みの DecodedCert 構造体を解放します。
int	**wc_SetTimeCb**(wc_time_cb f) この関数はタイムコールバック関数を登録します。wolfSSL が現在時刻を必要としたタイミングでこのコールバックを呼び出します。このタイムコールバック関数のプロトタイプ（シグネチャ）は C 標準ライブラリの “time” 関数と同一です。
time_t	**wc_Time**(time_t * t) この関数は現在時刻を取得します。デフォルトで XTIME マクロ関数を使います。このマクロ関数はプラットフォーム依存です。ユーザーはこのマクロの代わりに wc_SetTimeCb でタイムコールバック関数を使うように設定することができます
int	**wc_SetCustomExtension**(Cert * cert, int critical, const char * oid, const byte * der, word32 derSz) この関数は X.509 証明書にカスタム拡張を追加します。注: この関数に渡すポインタ引数が保持する内容は証明書が生成されるまで変更されてはいけません。この関数ではポインタが指す先の内容は別のバッファには複製しません。
int	**wc_SetUnknownExtCallback**(DecodedCert * cert, wc_UnknownExtCallback cb) この関数は wolfSSL が証明書の解析中に未知の X.509 拡張に遭遇した際に呼び出すコールバック関数を登録します。コールバック関数のプロトタイプは使用例を参照してください。
int	**wc_CheckCertSigPubKey**(const byte * cert, word32 certSz, void * heap, const byte * pubKey, word32 pubKeySz, int pubKeyOID) この関数は DER 形式の X.509 証明書の署名を与えられた公開鍵を使って検証します。公開鍵は DER 形式で全公開鍵情報を含んだものが求められます。

	Name
int	**wc_Asn1PrintOptions_Init**(Asn1PrintOptions * opts) この関数は Asn1PrintOptions 構造体を初期化します。
int	**wc_Asn1PrintOptions_Set**(Asn1PrintOptions * opts, enum Asn1PrintOpt opt, word32 val) この関数は Asn1PrintOptions 構造体にプリント情報を設定します。
int	**wc_Asn1_Init**(Asn1 * asn1) この関数は Asn1 構造体を初期化します。
int	**wc_Asn1_SetFile**(Asn1 * asn1, XFILE file) この関数は出力先として使用するファイルを Asn1 構造体にセットします。
int	**wc_Asn1_PrintAll**(Asn1 * asn1, Asn1PrintOptions * opts, unsigned char * data, word32 len)ASN.1 アイテムをプリントします。

B.1.2 Functions Documentation

B.1.2.1 function wc_InitCert

```
int wc_InitCert(
    Cert *
)
```

この関数は Cert 構造体をデフォルトの値で初期化します。デフォルトのオプション：version = 3（0x2）、sigtype = sha_with_rsa、issuer = 空白、dayValid = 500、selfsigned = 1（true）発行者としての件名 = 空白

See:

- wc_MakeCert
- wc_MakeCertReq

Return: 成功した場合 0 を返します。

Example

```
Cert myCert;
wc_InitCert(&myCert);
```

B.1.2.2 function wc_CertNew

```
Cert * wc_CertNew(
    void * heap
)
```

この関数は証明書操作の為に新たな Cert 構造体を割り当てます。割り当てた Cert 構造体はこの関数内で初期化されるので、wc_InitCert() を呼び出す必要はありません。アプリケーションがこの Cert 構造体の使用を終了する際には wc_CertFree() を呼び出す必要があります。

Parameters:

- **メモリの動的確保で使用されるヒープへのポインタ。NULL の指定も可。** *Example*

```
Cert*   myCert;

myCert = wc_CertNew(NULL);
if (myCert == NULL) {
```

```
    // Cert creation failure
}
```

See:

- wc_InitCert
- wc_MakeCert
- wc_CertFree

Return:

- 処理が成功した際には新に割り当てられた Cert 構造体へのポインタを返します。
- メモリ確保に失敗した場合には NULL を返します。

B.1.2.3 function wc_CertFree

```
void wc_CertFree(
    Cert * cert
)
```

この関数は wc_CertNew() で確保された Cert 構造体を解放します。

Parameters:

- **解放すべき Cert 構造体へのポインタ** *Example*

```
Cert*   myCert;

myCert = wc_CertNew(NULL);

// Perform cert operations.

wc_CertFree(myCert);
```

See:

- wc_InitCert
- wc_MakeCert
- wc_CertNew

Return: 無し

B.1.2.4 function wc_MakeCert

```
int wc_MakeCert(
    Cert * cert,
    byte * derBuffer,
    word32 derSz,
    RsaKey * rsaKey,
    ecc_key * eccKey,
    WC_RNG * rng
)
```

CA 署名付き証明書を作成するために使用されます。サブジェクト情報を入力した後に呼び出す必要があります。この関数は、証明書入力から X.509v3 RSA または ECC 証明書を作成し derBuffer に書き込みます。証明書を生成するための RsaKey または EccKey のいずれかを引数として取ります。この関数が呼び出される前に、証明書を wc_InitCert で初期化する必要があります。

Parameters:

- **cert** 初期化された Cert 構造体へのポインタ

- **derBuffer** 生成された証明書を保持するバッファへのポインタ
- **derSz** 証明書を保存するバッファのサイズ
- **rsaKey** 証明書の生成に使用される RSA 鍵を含む RsaKey 構造体へのポインタ
- **eccKey** 証明書の生成に使用される ECC 鍵を含む EccKey 構造体へのポインタ

See:

- wc_InitCert
- wc_MakeCertReq

Return:

- 指定された入力証明書から X509 証明書が正常に生成された場合、生成された証明書のサイズを返します。
- MEMORY_E xmalloc でのメモリ割り当てでエラーが発生した場合に返ります。
- BUFFER_E 提供された derBuffer が生成された証明書を保存するには小さすぎる場合に返されます
- Others 証明書の生成が成功しなかった場合、追加のエラーメッセージが返される可能性があります。

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);
```

B.1.2.5 function wc_MakeCertReq

```
int wc_MakeCertReq(
    Cert * cert,
    byte * derBuffer,
    word32 derSz,
    RsaKey * rsaKey,
    ecc_key * eccKey
)
```

この関数は、入力された Cert 構造体を使用して証明書署名要求を作成し derBuffer に書き込みます。証明書要求の生成には RsaKey または EccKey のいずれかの鍵を受け取り使用します。この関数の後に、署名するために wc_SignCert() を呼び出す必要があります。この関数の使用例については、wolfCrypt テストアプリケーション (./wolfcrypt/test/test.c) を参照してください。

Parameters:

- **cert** 初期化された Cert 構造体へのポインタ
- **derBuffer** 生成された証明書署名要求を保持するバッファへのポインタ
- **derSz** 証明書署名要求を保存するバッファのサイズ
- **rsaKey** 証明書署名要求を生成するために使用される RSA 鍵を含む RsaKey 構造体へのポインタ
- **eccKey** 証明書署名要求を生成するために使用される RECC 鍵を含む EccKey 構造体へのポインタ

See:

- wc_InitCert
- wc_MakeCert

Return:

- 証明書署名要求が正常に生成されると、生成された証明書署名要求のサイズを返します。

- MEMORY_E xmalloc でのメモリ割り当てでエラーが発生した場合
- BUFFER_E 提供された derBuffer が生成された証明書を保存するには小さすぎる場合
- Other 証明書署名要求の生成が成功しなかった場合、追加のエラーメッセージが返される可能性があります。

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);
```

B.1.2.6 function wc_SignCert

```
int wc_SignCert(
    int requestSz,
    int sigType,
    byte * derBuffer,
    word32 derSz,
    RsaKey * rsaKey,
    ecc_key * eccKey,
    WC_RNG * rng
)
```

この関数はバッファーの内容に署名し、署名をバッファの最後に追加します。署名の種類を取ります。CA 署名付き証明書を作成する場合は、wc_MakeCert() または wc_MakeCertReq() の後に呼び出す必要があります。

Parameters:

- **requestSz** 署名対象の証明書本文のサイズ
- **sigType** 作成する署名の種類。有効なオプションは次のとおりです:CTC_MD5WRSA、CTC_SHAWRSA、CTC_SHAWECDSA、CTC_SHA256WECDSA、ANDCTC_SHA256WRSA
- **derBuffer** 署名対象の証明書を含むバッファへのポインタ。関数の処理成功時には署名が付加された証明書を保持します。
- **derSz** 新たに署名された証明書を保存するバッファの（合計）サイズ
- **rsaKey** 証明書に署名するために使用される RSA 鍵を含む RsaKey 構造体へのポインタ
- **eccKey** 証明書に署名するために使用される ECC 鍵を含む EccKey 構造体へのポインタ
- **rng** 署名に使用する乱数生成器 (WC_RNG 構造体) へのポインタ

See:

- wc_InitCert
- wc_MakeCert

Return:

- 証明書への署名に成功した場合は、証明書の新しいサイズ (署名を含む) を返します。
- MEMORY_E xmalloc でのメモリを割り当てでエラーがある場合
- BUFFER_E 提供された証明書を保存するには提供されたバッファが小さすぎる場合に返されます。
- Other 証明書の生成が成功しなかった場合、追加のエラーメッセージが返される可能性があります。

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);
```

B.1.2.7 function wc_MakeSelfCert

```
int wc_MakeSelfCert(
    Cert * cert,
    byte * derBuffer,
    word32 derSz,
    RsaKey * key,
    WC_RNG * rng
)
```

この関数は、以前の 2 つの関数、wc_MakeCert、および自己署名のための wc_SignCert の組み合わせです（前の関数は CA 要求に使用される場合があります）。証明書を作成してから、それに署名し、自己署名証明書を生成します。

Parameters:

- **cert** 署名する対象の Cert 構造体へのポインタ
- **derBuffer** 署名付き証明書を保持するためのバッファへのポインタ
- **derSz** 署名付き証明書を保存するバッファのサイズ
- **key** 証明書に署名するために使用される RSA 鍵を含む RsaKey 構造体へのポインタ
- **rng** 署名に使用する乱数生成器 (WC_RNG 構造体) へのポインタ

See:

- wc_InitCert
- wc_MakeCert
- wc_SignCert

Return:

- 証明書への署名が成功した場合は、証明書の新しいサイズを返します。
- MEMORY_E xmalloc でのメモリを割り当てでエラーがある場合
- BUFFER_E 提供された証明書を保存するには提供されたバッファが小さすぎる場合に返されます。
- Other 証明書の生成が成功しなかった場合、追加のエラーメッセージが返される可能性があります。

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);
```

B.1.2.8 function wc_SetIssuer

```
int wc_SetIssuer(
    Cert * cert,
    const char * issuerFile
)
```

この関数は PEM 形式の issureFile で与えられた発行者を証明書の発行者として設定します。また、その際に、証明書の自己署名プロパティを false に変更します。発行者は証明書の発行者として設定される前に検証されます。この関数は証明書への署名に先立ち呼び出される必要があります。

Parameters:

- **cert** 発行者を設定する対象の Cert 構造体へのポインタ
- **issuerFile** PEM 形式の証明書ファイルへのファイルパス

See:

- wc_InitCert
- wc_SetSubject
- wc_SetIssuerBuffer

Return:

- 0 証明書の発行者の設定に成功した場合に返されます。
- MEMORY_E XMALLOC でメモリの確保に失敗した際に返されます。
- ASN_PARSE_E 証明書のヘッダーファイルの解析に失敗した際に返されます。
- ASN_OBJECT_ID_E 証明書の暗号タイプの解析でエラーが発生した際に返されます。
- ASN_EXPECT_0_E 証明書の暗号化仕様にフォーマットエラーが検出された際に返されます。
- ASN_BEFORE_DATE_E 証明書の使用開始日より前であった場合に返されます。
- ASN_AFTER_DATE_E 証明書の有効期限日より後であった場合に返されます。
- ASN_BITSTR_E 証明書のビットストリング要素の解析でエラーが発生した際に返されます。
- ECC_CURVE_OID_E 証明書の ECC 鍵の解析でエラーが発生した際に返されます。
- ASN_UNKNOWN_OID_E 証明書が未知のオブジェクト ID を使用していた際に返されます。
- ASN_VERSION_E ALLOW_V1_EXTENSIONS マクロが定義されていないのに証明書が V1 あるいは V2 形式であった場合に返されます。
- BAD_FUNC_ARG 証明書の拡張情報の解析でエラーが発生した際に返されます。
- ASN_CRIT_EXT_E 証明書の解析中に未知のクリティカル拡張に遭遇した際に返されます。
- ASN_SIG_OID_E 署名暗号化タイプが引数で渡された証明書のタイプと異なる場合に返されます。
- ASN_SIG_CONFIRM_E 証明書の署名の検証に失敗した際に返されます。
- ASN_NAME_INVALID_E 証明書の名前が CA の名前に関数制限によって許されていない場合に返されます。
- ASN_NO_SIGNER_E CA 証明書の発行者を検証することができない場合に返されます。

Example

```
Cert myCert;
// initialize myCert
if(wc_SetIssuer(&myCert, ”./path/to/ca-cert.pem”) != 0) {
    // error setting issuer
}
```

B.1.2.9 function wc_SetSubject

```
int wc_SetSubject(
    Cert * cert,
    const char * subjectFile
)
```

この関数は PEM 形式の subjectFile で与えられた主体者を証明書の主体者として設定します。この関数は証明書への署名に先立ち呼び出される必要があります。

Parameters:

- **主体者を設定する対象の Cert 構造体へのポインタ**
- **subjectFile** PEM 形式の証明書ファイルへのファイルパス

See:

- wc_InitCert
- wc_SetIssuer

Return:

- 0 証明書の主体者の設定に成功した場合に返されます。
- MEMORY_E XMALLOC でメモリの確保に失敗した際に返されます。
- ASN_PARSE_E 証明書のヘッダーファイルの解析に失敗した際に返されます。
- ASN_OBJECT_ID_E 証明書の暗号タイプの解析でエラーが発生した際に返されます。
- ASN_EXPECT_0_E 証明書の暗号化仕様にフォーマットエラーが検出された際に返されます。
- ASN_BEFORE_DATE_E 証明書の使用開始日より前であった場合に返されます。
- ASN_AFTER_DATE_E 証明書の有効期限日より後であった場合に返されます。
- ASN_BITSTR_E 証明書のビットストリング要素の解析でエラーが発生した際に返されます。
- ECC_CURVE_OID_E 証明書の ECC 鍵の解析でエラーが発生した際に返されます。
- ASN_UNKNOWN_OID_E 証明書が未知のオブジェクト ID を使用していた際に返されます。
- ASN_VERSION_E ALLOW_V1_EXTENSIONS マクロが定義されていないのに証明書が V1 あるいは V2 形式であった場合に返されます。
- BAD_FUNC_ARG 証明書の拡張情報の解析でエラーが発生した際に返されます。
- ASN_CRIT_EXT_E 証明書の解析中に未知のクリティカル拡張に遭遇した際に返されます。
- ASN_SIG_OID_E 署名暗号化タイプが引数で渡された証明書のタイプと異なる場合に返されます。
- ASN_SIG_CONFIRM_E 証明書の署名の検証に失敗した際に返されます。
- ASN_NAME_INVALID_E 証明書の名前が CA の名前に関数制限によって許されていない場合に返されます。
- ASN_NO_SIGNER_E CA 証明書の主体者を検証することができない場合に返されます。

Example

```
Cert myCert;
// initialize myCert
if(wc_SetSubject(&myCert, ”./path/to/ca-cert.pem”) != 0) {
    // error setting subject
}
```

B.1.2.10 function wc_SetSubjectRaw

```
int wc_SetSubjectRaw(
    Cert * cert,
    const byte * der,
    int derSz
)
```

この関数は DER 形式でバッファに格納されている Raw-Subject 情報を証明書の Raw-Subject 情報として設定します。この関数は証明書への署名に先立ち呼び出される必要があります。

Parameters:

- **cert** Raw-Subject 情報を設定する対象の Cert 構造体へのポインタ
- **der** DER 形式の証明書を格納しているバッファへのポインタ。この証明書の Raw-Subject 情報が取り出されて cert に設定されます。
- **derSz** DER 形式の証明書を格納しているバッファのサイズ

See:

- wc_InitCert
- wc_SetSubject

Return:

- 0 証明書の Raw-Subject 情報の設定に成功した場合に返されます。
- MEMORY_E XMALLOC でメモリの確保に失敗した際に返されます。
- ASN_PARSE_E 証明書のヘッダーファイルの解析に失敗した際に返されます。
- ASN_OBJECT_ID_E 証明書の暗号タイプの解析でエラーが発生した際に返されます。
- ASN_EXPECT_0_E 証明書の暗号化仕様にフォーマットエラーが検出された際に返されます。
- ASN_BEFORE_DATE_E 証明書の使用開始日より前であった場合に返されます。
- ASN_AFTER_DATE_E 証明書の有効期限日より後であった場合に返されます。
- ASN_BITSTR_E 証明書のビットストリング要素の解析でエラーが発生した際に返されます。
- ECC_CURVE_OID_E 証明書の ECC 鍵の解析でエラーが発生した際に返されます。
- ASN_UNKNOWN_OID_E 証明書が未知のオブジェクト ID を使用していた際に返されます。
- ASN_VERSION_E ALLOW_V1_EXTENSIONS マクロが定義されていないのに証明書が V1 あるいは V2 形式であった場合に返されます。
- BAD_FUNC_ARG 証明書の拡張情報の解析でエラーが発生した際に返されます。
- ASN_CRIT_EXT_E 証明書の解析中に未知のクリティカル拡張に遭遇した際に返されます。
- ASN_SIG_OID_E 署名暗号化タイプが引数で渡された証明書のタイプと異なる場合に返されます。
- ASN_SIG_CONFIRM_E 証明書の署名の検証に失敗した際に返されます。
- ASN_NAME_INVALID_E 証明書の名前が CA の名前に関数制限によって許されていない場合に返されます。
- ASN_NO_SIGNER_E CA 証明書の主体者を検証することができない場合に返されます。
- ASN_NO_SIGNER_E CA 証明書の主体者を検証することができない場合に返されます。

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
}
```

B.1.2.11 function wc_GetSubjectRaw

```
int wc_GetSubjectRaw(
    byte ** subjectRaw,
    Cert * cert
)
```

この関数は Cert 構造体から Raw-Subject 情報を取り出します。

Parameters:

- **subjectRaw** 処理が成功した際に返される Raw-Subject 情報を格納するバッファへのポインタのポインタ
- **cert** Raw-Subject 情報を保持する Cert 構造体へのポインタ

See:

- wc_InitCert
- wc_SetSubjectRaw

Return:

- 0 証明書の Raw-Subject 情報の取得に成功した場合に返されます。
- BAD_FUNC_ARG 証明書の拡張情報の解析でエラーが発生した際に返されます。

Example

```
Cert myCert;
byte *subjRaw;
// initialize myCert

if(wc_GetSubjectRaw(&subjRaw, &myCert) != 0) {
    // error setting subject
}
```

B.1.2.12 function wc_SetAltNames

```
int wc_SetAltNames(
    Cert * cert,
    const char * file
)
```

この関数は引数で与えられた PEM 形式の証明書の主体者の別名を Cert 構造体に設定します。複数のドメインで同一の証明書を使用する際には主体者の別名を付与する機能は有用です。この関数は証明書への署名に先立ち呼び出される必要があります。

Parameters:

- **cert** 主体者の別名を設定する対象の Cert 構造体へのポインタ
- **file** PEM 形式の証明書のファイルパス

See:

- wc_InitCert
- wc_SetIssuer

Return:

- 0 証明書の主体者の設定に成功した場合に返されます。
- MEMORY_E XMALLOC でメモリの確保に失敗した際に返されます。
- ASN_PARSE_E 証明書のヘッダーファイルの解析に失敗した際に返されます。
- ASN_OBJECT_ID_E 証明書の暗号タイプの解析でエラーが発生した際に返されます。
- ASN_EXPECT_0_E 証明書の暗号化仕様にフォーマットエラーが検出された際に返されます。
- ASN_BEFORE_DATE_E 証明書の使用開始日より前であった場合に返されます。
- ASN_AFTER_DATE_E 証明書の有効期限日より後であった場合に返されます。
- ASN_BITSTR_E 証明書のビットストリング要素の解析でエラーが発生した際に返されます。
- ECC_CURVE_OID_E 証明書の ECC 鍵の解析でエラーが発生した際に返されます。
- ASN_UNKNOWN_OID_E 証明書が未知のオブジェクト ID を使用していた際に返されます。
- ASN_VERSION_E ALLOW_V1_EXTENSIONS マクロが定義されていないのに証明書が V1 あるいは V2 形式であった場合に返されます。
- BAD_FUNC_ARG 証明書の拡張情報の解析でエラーが発生した際に返されます。
- ASN_CRIT_EXT_E 証明書の解析中に未知のクリティカル拡張に遭遇した際に返されます。
- ASN_SIG_OID_E 署名暗号化タイプが引数で渡された証明書のタイプと異なる場合に返されます。
- ASN_SIG_CONFIRM_E 証明書の署名の検証に失敗した際に返されます。
- ASN_NAME_INVALID_E 証明書の名前が CA の名前に関数制限によって許されていない場合に返されます。
- ASN_NO_SIGNER_E CA 証明書の主体者を検証することができない場合に返されます。
- ASN_NO_SIGNER_E CA 証明書の主体者を検証することができない場合に返されます。

Example

```
Cert myCert;
// initialize myCert
if(wc_SetSubject(&myCert, ”./path/to/ca-cert.pem”) != 0) {
    // error setting alt names
}
```

B.1.2.13 function wc_SetIssuerBuffer

```
int wc_SetIssuerBuffer(
    Cert * cert,
    const byte * der,
    int derSz
)
```

この関数は DER 形式でバッファに格納されている発行者を証明書の発行者として設定します。加えて、証明書の事故署名プロパティを false に設定します。この関数は証明書への署名に先立ち呼び出される必要があります。

Parameters:

- **cert** 発行者を設定する対象の Cert 構造体へのポインタ
- **der** DER 形式の証明書を格納しているバッファへのポインタ。この証明書の発行者情報が取り出されて cert に設定されます。
- **derSz** DER 形式の証明書を格納しているバッファのサイズ

See:

- wc_InitCert
- wc_SetIssuer

Return:

- 0 証明書の発行者の設定に成功した場合に返されます。
- MEMORY_E XMALLOC でメモリの確保に失敗した際に返されます。
- ASN_PARSE_E 証明書のヘッダーファイルの解析に失敗した際に返されます。
- ASN_OBJECT_ID_E 証明書の暗号タイプの解析でエラーが発生した際に返されます。
- ASN_EXPECT_0_E 証明書の暗号化仕様にフォーマットエラーが検出された際に返されます。
- ASN_BEFORE_DATE_E 証明書の使用開始日より前であった場合に返されます。
- ASN_AFTER_DATE_E 証明書の有効期限日より後であった場合に返されます。
- ASN_BITSTR_E 証明書のビットストリング要素の解析でエラーが発生した際に返されます。
- ECC_CURVE_OID_E 証明書の ECC 鍵の解析でエラーが発生した際に返されます。
- ASN_UNKNOWN_OID_E 証明書が未知のオブジェクト ID を使用していた際に返されます。
- ASN_VERSION_E ALLOW_V1_EXTENSIONS マクロが定義されていないのに証明書が V1 あるいは V2 形式であった場合に返されます。
- BAD_FUNC_ARG 証明書の拡張情報の解析でエラーが発生した際に返されます。
- ASN_CRIT_EXT_E 証明書の解析中に未知のクリティカル拡張に遭遇した際に返されます。
- ASN_SIG_OID_E 署名暗号化タイプが引数で渡された証明書のタイプと異なる場合に返されます。
- ASN_SIG_CONFIRM_E 証明書の署名の検証に失敗した際に返されます。
- ASN_NAME_INVALID_E 証明書の名前が CA の名前に関数制限によって許されていない場合に返されます。
- ASN_NO_SIGNER_E CA 証明書の主体者を検証することができない場合に返されます。
- ASN_NO_SIGNER_E CA 証明書の主体者を検証することができない場合に返されます。

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
}
```

B.1.2.14 function wc_SetIssuerRaw

```
int wc_SetIssuerRaw(
    Cert * cert,
    const byte * der,
    int derSz
)
```

この関数は DER 形式でバッファに格納されている Raw-Issuer 情報を証明書の Raw-Issuer 情報として設定します。この関数は証明書への署名に先立ち呼び出される必要があります。

Parameters:

- **cert** Raw-Issuer 情報を設定する対象の Cert 構造体へのポインタ
- **der** DER 形式の証明書を格納しているバッファへのポインタ。この証明書の Raw-Issuer 情報が取り出されて cert に設定されます。
- **derSz** DER 形式の証明書を格納しているバッファのサイズ

See:

- wc_InitCert
- wc_SetIssuer

Return:

- 0 証明書の Raw-Issuer 情報の設定に成功した場合に返されます。
- MEMORY_E XMALLOC でメモリの確保に失敗した際に返されます。
- ASN_PARSE_E 証明書のヘッダーファイルの解析に失敗した際に返されます。
- ASN_OBJECT_ID_E 証明書の暗号タイプの解析でエラーが発生した際に返されます。
- ASN_EXPECT_0_E 証明書の暗号化仕様にフォーマットエラーが検出された際に返されます。
- ASN_BEFORE_DATE_E 証明書の使用開始日より前であった場合に返されます。
- ASN_AFTER_DATE_E 証明書の有効期限日より後であった場合に返されます。
- ASN_BITSTR_E 証明書のビットストリング要素の解析でエラーが発生した際に返されます。
- ECC_CURVE_OID_E 証明書の ECC 鍵の解析でエラーが発生した際に返されます。
- ASN_UNKNOWN_OID_E 証明書が未知のオブジェクト ID を使用していた際に返されます。
- ASN_VERSION_E ALLOW_V1_EXTENSIONS マクロが定義されていないのに証明書が V1 あるいは V2 形式であった場合に返されます。
- BAD_FUNC_ARG 証明書の拡張情報の解析でエラーが発生した際に返されます。
- ASN_CRIT_EXT_E 証明書の解析中に未知のクリティカル拡張に遭遇した際に返されます。
- ASN_SIG_OID_E 署名暗号化タイプが引数で渡された証明書のタイプと異なる場合に返されます。
- ASN_SIG_CONFIRM_E 証明書の署名の検証に失敗した際に返されます。
- ASN_NAME_INVALID_E 証明書の名前が CA の名前に関数制限によって許されていない場合に返されます。
- ASN_NO_SIGNER_E CA 証明書の主体者を検証することができない場合に返されます。
- ASN_NO_SIGNER_E CA 証明書の主体者を検証することができない場合に返されます。

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
}
```

B.1.2.15 function wc_SetSubjectBuffer

```
int wc_SetSubjectBuffer(
    Cert * cert,
    const byte * der,
    int derSz
)
```

この関数は DER 形式でバッファに格納されている主体者を証明書の主体者として設定します。この関数は証明書への署名に先立ち呼び出される必要があります。

Parameters:

- **cert** 主体者を設定する対象の Cert 構造体へのポインタ
- **der** DER 形式の証明書を格納しているバッファへのポインタ。この証明書の主体者が取り出されて cert に設定されます。
- **derSz** DER 形式の証明書を格納しているバッファのサイズ

See:

- wc_InitCert
- wc_SetSubject

Return:

- 0 証明書の主体者の設定に成功した場合に返されます。
- MEMORY_E XMALLOC でメモリの確保に失敗した際に返されます。
- ASN_PARSE_E 証明書のヘッダーファイルの解析に失敗した際に返されます。
- ASN_OBJECT_ID_E 証明書の暗号タイプの解析でエラーが発生した際に返されます。
- ASN_EXPECT_0_E 証明書の暗号化仕様にフォーマットエラーが検出された際に返されます。
- ASN_BEFORE_DATE_E 証明書の使用開始日より前であった場合に返されます。
- ASN_AFTER_DATE_E 証明書の有効期限日より後であった場合に返されます。
- ASN_BITSTR_E 証明書のビットストリング要素の解析でエラーが発生した際に返されます。
- ECC_CURVE_OID_E 証明書の ECC 鍵の解析でエラーが発生した際に返されます。
- ASN_UNKNOWN_OID_E 証明書が未知のオブジェクト ID を使用していた際に返されます。
- ASN_VERSION_E ALLOW_V1_EXTENSIONS マクロが定義されていないのに証明書が V1 あるいは V2 形式であった場合に返されます。
- BAD_FUNC_ARG 証明書の拡張情報の解析でエラーが発生した際に返されます。
- ASN_CRIT_EXT_E 証明書の解析中に未知のクリティカル拡張に遭遇した際に返されます。
- ASN_SIG_OID_E 署名暗号化タイプが引数で渡された証明書のタイプと異なる場合に返されます。
- ASN_SIG_CONFIRM_E 証明書の署名の検証に失敗した際に返されます。
- ASN_NAME_INVALID_E 証明書の名前が CA の名前に関数制限によって許されていない場合に返されます。
- ASN_NO_SIGNER_E CA 証明書の主体者を検証することができない場合に返されます。
- ASN_NO_SIGNER_E CA 証明書の主体者を検証することができない場合に返されます。

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
}
```

B.1.2.16 function wc_SetAltNamesBuffer

```
int wc_SetAltNamesBuffer(
    Cert * cert,
    const byte * der,
    int derSz
)
```

この関数は DER 形式でバッファに格納されている「別名情報」を証明書の「別名情報」として設定します。この機能は複数ドメインを一つの証明書を使ってセキュアにする際に有用です。この関数は証明書への署名に先立ち呼び出される必要があります。

Parameters:

- **cert** 別名情報を設定する対象の Cert 構造体へのポインタ
- **der** DER 形式の証明書を格納しているバッファへのポインタ。この証明書の別名情報が取り出されて cert に設定されます。
- **derSz** DER 形式の証明書を格納しているバッファのサイズ

See:

- wc_InitCert
- wc_SetAltNames

Return:

- 0 証明書の別名情報の設定に成功した場合に返されます。
- MEMORY_E XMALLOC でメモリの確保に失敗した際に返されます。
- ASN_PARSE_E 証明書のヘッダーファイルの解析に失敗した際に返されます。
- ASN_OBJECT_ID_E 証明書の暗号タイプの解析でエラーが発生した際に返されます。
- ASN_EXPECT_0_E 証明書の暗号化仕様にフォーマットエラーが検出された際に返されます。
- ASN_BEFORE_DATE_E 証明書の使用開始日より前であった場合に返されます。
- ASN_AFTER_DATE_E 証明書の有効期限日より後であった場合に返されます。
- ASN_BITSTR_E 証明書のビットストリング要素の解析でエラーが発生した際に返されます。
- ECC_CURVE_OID_E 証明書の ECC 鍵の解析でエラーが発生した際に返されます。
- ASN_UNKNOWN_OID_E 証明書が未知のオブジェクト ID を使用していた際に返されます。
- ASN_VERSION_E ALLOW_V1_EXTENSIONS マクロが定義されていないのに証明書が V1 あるいは V2 形式であった場合に返されます。
- BAD_FUNC_ARG 証明書の拡張情報の解析でエラーが発生した際に返されます。
- ASN_CRIT_EXT_E 証明書の解析中に未知のクリティカル拡張に遭遇した際に返されます。
- ASN_SIG_OID_E 署名暗号化タイプが引数で渡された証明書のタイプと異なる場合に返されます。
- ASN_SIG_CONFIRM_E 証明書の署名の検証に失敗した際に返されます。
- ASN_NAME_INVALID_E 証明書の名前が CA の名前に関数制限によって許されていない場合に返されます。
- ASN_NO_SIGNER_E CA 証明書の主体者を検証することができない場合に返されます。
- ASN_NO_SIGNER_E CA 証明書の主体者を検証することができない場合に返されます。

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
}
```

B.1.2.17 function wc_SetDatesBuffer

```
int wc_SetDatesBuffer(
    Cert * cert,
    const byte * der,
    int derSz
)
```

この関数は DER 形式でバッファに格納されている「有効期間」情報を証明書の「有効期間」情報として設定します。この関数は証明書への署名に先立ち呼び出される必要があります。

Parameters:

- **cert** 有効期間情報を設定する対象の Cert 構造体へのポインタ
- **der** DER 形式の証明書を格納しているバッファへのポインタ。この証明書の有効期間情報が取り出されて cert に設定されます。
- **derSz** DER 形式の証明書を格納しているバッファのサイズ

See: wc_InitCert

Return:

- 0 証明書の有効期間情報の設定に成功した場合に返されます。
- MEMORY_E XMALLOC でメモリの確保に失敗した際に返されます。
- ASN_PARSE_E 証明書のヘッダーファイルの解析に失敗した際に返されます。
- ASN_OBJECT_ID_E 証明書の暗号タイプの解析でエラーが発生した際に返されます。
- ASN_EXPECT_0_E 証明書の暗号化仕様にフォーマットエラーが検出された際に返されます。
- ASN_BEFORE_DATE_E 証明書の使用開始日より前であった場合に返されます。
- ASN_AFTER_DATE_E 証明書の有効期限日より後であった場合に返されます。
- ASN_BITSTR_E 証明書のビットストリング要素の解析でエラーが発生した際に返されます。
- ECC_CURVE_OID_E 証明書の ECC 鍵の解析でエラーが発生した際に返されます。
- ASN_UNKNOWN_OID_E 証明書が未知のオブジェクト ID を使用していた際に返されます。
- ASN_VERSION_E ALLOW_V1_EXTENSIONS マクロが定義されていないのに証明書が V1 あるいは V2 形式であった場合に返されます。
- BAD_FUNC_ARG 証明書の拡張情報の解析でエラーが発生した際に返されます。
- ASN_CRIT_EXT_E 証明書の解析中に未知のクリティカル拡張に遭遇した際に返されます。
- ASN_SIG_OID_E 署名暗号化タイプが引数で渡された証明書のタイプと異なる場合に返されます。
- ASN_SIG_CONFIRM_E 証明書の署名の検証に失敗した際に返されます。
- ASN_NAME_INVALID_E 証明書の名前が CA の名前に関数制限によって許されていない場合に返されます。
- ASN_NO_SIGNER_E CA 証明書の主体者を検証することができない場合に返されます。
- ASN_NO_SIGNER_E CA 証明書の主体者を検証することができない場合に返されます。

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
}
```

B.1.2.18 function wc_SetAuthKeyIdFromPublicKey

```
int wc_SetAuthKeyIdFromPublicKey(
    Cert * cert,
    RsaKey * rsakey,
```

```
    ecc_key * eckey
)
```

この関数は指定された RSA あるいは ECC 公開鍵の一方から得た AKID（認証者鍵 ID）を証明書の AKID として設定します。

Parameters:

- **cert** AKID を設定する対象の Cert 構造体へのポインタ
- **rsakey** RsaKey 構造体へのポインタ
- **eckey** ecc_key 構造体へのポインタ

See:

- wc_SetSubjectKeyId
- wc_SetAuthKeyId
- wc_SetAuthKeyIdFromCert

Return:

- 0 証明書の AKID の設定に成功した場合に返されます。
- BAD_FUNC_ARG Cert 構造体へのポインタ (cert) が NULL か RsaKey 構造体へのポインタ (rsakey) と ecc_key 構造体へのポインタ (eckey) の両方が NULL である場合に返されます。
- MEMORY_E メモリの確保に失敗した際に返されます。
- PUBLIC_KEY_E 公開鍵の取得に失敗した際に返されます。

Example

```
Cert myCert;
RsaKey keypub;

wc_InitRsaKey(&keypub, 0);

if (wc_SetAuthKeyIdFromPublicKey(&myCert, &keypub, NULL) != 0)
{
    // Handle error
}
```

B.1.2.19 function wc_SetAuthKeyIdFromCert

```
int wc_SetAuthKeyIdFromCert(
    Cert * cert,
    const byte * der,
    int derSz
)
```

この関数は DER 形式でバッファに格納された証明書から得た AKID(認証者鍵 ID) を証明書の AKID として設定します。

Parameters:

- **cert** AKID を設定する対象の Cert 構造体へのポインタ。
- **der** DER 形式の証明書を格納しているバッファへのポインタ。
- **derSz** DER 形式の証明書を格納しているバッファのサイズ。

See:

- wc_SetAuthKeyIdFromPublicKey
- wc_SetAuthKeyId

Return:

- 0 証明書の AKID の設定に成功した場合に返されます。
- BAD_FUNC_ARG 引数のいずれかが NULL, あるいは derSz が 0 より小さい場合に返されます。
- MEMORY_E メモリの確保に失敗した際に返されます。
- ASN_NO_SKID 認証者鍵 ID が見つからない場合に返されます。

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
}
```

B.1.2.20 function wc_SetAuthKeyId

```
int wc_SetAuthKeyId(
    Cert * cert,
    const char * file
)
```

この関数は PEM 形式の証明書から得た AKID(認証者鍵 ID) を証明書の AKID として設定します。

Parameters:

- **cert** AKID を設定する対象の Cert 構造体へのポインタ。
- **file** PEM 形式の証明書ファイルへのファイルパス

See:

- wc_SetAuthKeyIdFromPublicKey
- wc_SetAuthKeyIdFromCert

Return:

- 0 証明書の AKID の設定に成功した場合に返されます。
- BAD_FUNC_ARG 引数のいずれかが NULL の場合に返されます。
- MEMORY_E メモリの確保に失敗した際に返されます。

Example

```
char* file_name = "/path/to/file";
cert some_cert;
wc_InitCert(&some_cert);

if(wc_SetAuthKeyId(&some_cert, file_name) != 0)
{
    // Handle Error
}
```

B.1.2.21 function wc_SetSubjectKeyIdFromPublicKey

```
int wc_SetSubjectKeyIdFromPublicKey(
    Cert * cert,
    RsaKey * rsakey,
    ecc_key * eckey
)
```

この関数は指定された RSA あるいは ECC 公開鍵の一方から得た SKID（主体者鍵 ID）を証明書の SKID として設定します。

Parameters:

- **cert** SKID を設定する対象の Cert 構造体へのポインタ
- **rsakey** RsaKey 構造体へのポインタ
- **eckey** ecc_key 構造体へのポインタ

See: wc_SetSubjectKeyId

Return:

- 0 証明書の SKID の設定に成功した場合に返されます。
- BAD_FUNC_ARG Cert 構造体へのポインタ (cert) が NULL か RsaKey 構造体へのポインタ (rsakey) と ecc_key 構造体へのポインタ (eckey) の両方が NULL である場合に返されます。
- MEMORY_E メモリの確保に失敗した際に返されます。
- PUBLIC_KEY_E 公開鍵の取得に失敗した際に返されます。

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
}
```

B.1.2.22 function wc_SetSubjectKeyId

```
int wc_SetSubjectKeyId(
    Cert * cert,
    const char * file
)
```

この関数は PEM 形式の証明書から得た SKID(主体者鍵 ID) を証明書の SKID として設定します。引数は両方が与えられることが必要です。

Parameters:

- **cert** SKID を設定する対象の Cert 構造体へのポインタ。
- **file** PEM 形式の証明書ファイルへのファイルパス

See: wc_SetSubjectKeyIdFromPublicKey

Return:

- 0 証明書の SKID の設定に成功した場合に返されます。
- BAD_FUNC_ARG 引数のいずれかが NULL の場合に返されます。
- MEMORY_E メモリの確保に失敗した際に返されます。
- PUBLIC_KEY_E 公開鍵のデコードに失敗した際に返されます。

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
}
```

B.1.2.23 function wc_PemPubKeyToDer

```
int wc_PemPubKeyToDer(
    const char * fileName,
    unsigned char * derBuf,
    int derSz
)
```

PEM 形式の鍵ファイルをロードし DER 形式に変換してバッファに出力します。

Parameters:

- **fileName** PEM 形式のファイルパス
- **derBuf** DER 形式鍵を出力する先のバッファ
- **derSz** 出力先バッファのサイズ

See: wc_PubKeyPemToDer

Return:

- 0 処理成功時に返されます。
- <0 エラー発生時に返されます。
- SSL_BAD_FILE ファイルのオープンに問題が生じた際に返されます。
- MEMORY_E メモリの確保に失敗した際に返されます。
- BUFFER_E 与えられた出力バッファ derBuf が結果を保持するのに十分な大きさがない場合に返されます。

Example

```
char* some_file = "filename";
unsigned char der[];

if(wc_PemPubKeyToDer(some_file, der, sizeof(der)) != 0)
{
    //Handle Error
}
```

B.1.2.24 function wc_PubKeyPemToDer

```
int wc_PubKeyPemToDer(
    const unsigned char * pem,
    int pemSz,
    unsigned char * buff,
    int buffSz
)
```

PEM 形式の鍵データを DER 形式に変換してバッファに出力し、出力バイト数あるいは負のエラー値を返します。

Parameters:

- **pem** PEM 形式の鍵を含んだバッファへのポインタ
- **pemSz** PEM 形式の鍵を含んだバッファのサイズ
- **buff** 出力先バッファへのポインタ
- **buffSz** 出力先バッファのサイズ

See: wc_PemPubKeyToDer

Return:

- 0 処理成功時には出力したバイト数が返されます。
- BAD_FUNC_ARG 引数の pem, buff, あるいは buffSz のいずれかば NULL の場合に返されます。
- <0 エラーが発生した際に返されます。

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
}
```

B.1.2.25 function wc_PemCertToDer

```
int wc_PemCertToDer(
    const char * fileName,
    unsigned char * derBuf,
    int derSz
)
```

この関数は PEM 形式の証明書を DER 形式に変換し、与えられたバッファに出力します。

Parameters:

- **fileName** PEM 形式のファイルパス
- **derBuf** DER 形式証明書を出力する先のバッファへのポインタ
- **derSz** DER 形式証明書を出力する先のバッファのサイズ

See: none

Return:

- 処理成功時には出力したバイト数が返されます。
- BUFFER_E 与えられた出力バッファ derBuf が結果を保持するのに十分な大きさがない場合に返されます。
- MEMORY_E メモリの確保に失敗した際に返されます。

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
}
```

B.1.2.26 function wc_DerToPem

```
int wc_DerToPem(
    const byte * der,
    word32 derSz,
```

```
    byte * output,
    word32 outputSz,
    int type
)
```

この関数はバッファで与えられた DER 形式の証明書を PEM 形式に変換し、与えられた出力用バッファに出力します。この関数は入力バッファと出力バッファを共用することはできません。両バッファは必ず別のものを用意してください。

Parameters:

- **der** DER 形式証明書データを保持するバッファへのポインタ
- **derSz** DER 形式証明書データのサイズ
- **output** PEM 形式証明書データを出力する先のバッファへのポインタ
- **outSz** PEM 形式証明書データを出力する先のバッファのサイズ
- **type** 変換する証明書のタイプ。次のタイプが指定可: CERT_TYPE, PRIVATEKEY_TYPE, ECC_PRIVATEKEY_TYPE, and CERTREQ_TYPE.

See: wc_PemCertToDer

Return:

- 処理成功時には変換後の PEM 形式データのサイズを返します。
- BAD_FUNC_ARG DER 形式証明書データの解析中にエラーが発生した際、あるいは PEM 形式に変換の際にエラーが発生した際に返されます。
- MEMORY_E メモリの確保に失敗した際に返されます。
- ASN_INPUT_E Base64 エンコーディングエラーが検出された際に返されます。
- BUFFER_E 与えられた出力バッファが結果を保持するのに十分な大きさがない場合に返されます。

Example

```
byte* der;
// initialize der with certificate
byte* pemFormatted[FOURK_BUF];

word32 pemSz;
pemSz = wc_DerToPem(der, derSz,pemFormatted,FOURK_BUF, CERT_TYPE);
```

B.1.2.27 function wc_DerToPemEx

```
int wc_DerToPemEx(
    const byte * der,
    word32 derSz,
    byte * output,
    word32 outputSz,
    byte * cipherIno,
    int type
)
```

この関数は DER 形式証明書を入力バッファから読み出し、PEM 形式に変換して出力バッファに出力します。この関数は入力バッファと出力バッファを共用することはできません。両バッファは必ず別のものを用意してください。追加の暗号情報を指定することができます。

Parameters:

- **der** DER 形式証明書データを保持するバッファへのポインタ
- **derSz** DER 形式証明書データのサイズ
- **output** PEM 形式証明書データを出力する先のバッファへのポインタ
- **outSz** PEM 形式証明書データを出力する先のバッファのサイズ

- **cipher_inf** 追加の暗号情報
- **type** 生成する証明書タイプ。指定可能なタイプ: CERT_TYPE, PRIVATEKEY_TYPE, ECC_PRIVATEKEY_TYPE と CERTREQ_TYPE

See: wc_PemCertToDer

Return:

- 処理成功時には変換後の PEM 形式データのサイズを返します。
- BAD_FUNC_ARG Returned DER 形式証明書データの解析中にエラーが発生した際、あるいは PEM 形式に変換の際にエラーが発生した際に返されます。
- MEMORY_E メモリの確保に失敗した際に返されます。
- ASN_INPUT_E Base64 エンコーディングエラーが検出された際に返されます。
- BUFFER_E 与えられた出力バッファが結果を保持するのに十分な大きさがない場合に返されます。

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);
```

B.1.2.28 function wc_EccPrivateKeyDecode

```
int wc_EccPrivateKeyDecode(
    const byte * input,
    word32 * inOutIdx,
    ecc_key * key,
    word32 inSz
)
```

この関数は ECC 秘密鍵を入力バッファから読み込み、解析の後 ecc_key 構造体を作成してそこに鍵を格納します。

Parameters:

- **input** 入力となる秘密鍵データを含んでいるバッファへのポインタ
- **inOutIdx** word32 型変数で内容として入力バッファの処理開始位置を先頭からのインデクス値として保持している。
- **key** デコードされた秘密鍵が格納される初期化済みの ecc_key 構造体へのポインタ
- **inSz** 秘密鍵を含んでいる入力バッファのサイズ

See: wc_RSA_PrivateKeyDecode

Return:

- 0 秘密鍵のデコードと結果の ecc_key 構造体への格納成功時に返されます。
- ASN_PARSE_E 入力バッファの解析あるいは結果の格納時にエラーが発生した場合に返されます。
- MEMORY_E メモリの確保に失敗した際に返されます。
- BUFFER_E 入力された証明書が最大証明書サイズより大きかった場合に返されます。
- ASN_OBJECT_ID_E 証明書が無効なオブジェクト ID を含んでいる場合に返されます。
- ECC_CURVE_OID_E 与えられた秘密鍵の ECC 曲線がサポートされていない場合に返されます。
- ECC_BAD_ARG_E ECC 秘密鍵のフォーマットにエラーがある場合に返されます。
- NOT_COMPILED_IN 秘密鍵が圧縮されていて圧縮鍵が提供されていない場合に返されます。
- MP_MEM 秘密鍵の解析で使用される数学ライブラリがエラーを検出した場合に返されます。
- MP_VAL 秘密鍵の解析で使用される数学ライブラリがエラーを検出した場合に返されます。

- MP_RANGE 秘密鍵の解析で使用される数学ライブラリがエラーを検出した場合に返されます。

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
}
```

B.1.2.29 function wc_EccKeyToDer

```
int wc_EccKeyToDer(
    ecc_key * key,
    byte * output,
    word32 inLen
)
```

この関数は ECC 秘密鍵を DER 形式でバッファに出力します。

Parameters:

- **key** 入力となる ECC 秘密鍵データを含んでいるバッファへのポインタ
- **output** DER 形式の ECC 秘密鍵を出力する先のバッファへのポインタ
- **inLen** DER 形式の ECC 秘密鍵を出力する先のバッファのサイズ

See: wc_RsaKeyToDer

Return:

- ECC 秘密鍵を DER 形式での出力に成功した場合にはバッファへ出力したサイズを返します。
- BAD_FUNC_ARG 出力バッファ output が NULL あるいは inLen がゼロの場合に返します。
- MEMORY_E メモリの確保に失敗した際に返されます。
- BUFFER_E 出力バッファが必要量より小さい
- ASN_UNKNOWN_OID_E ECC 秘密鍵が未知のタイプの場合に返します。
- MP_MEM 秘密鍵の解析で使用される数学ライブラリがエラーを検出した場合に返されます。
- MP_VAL 秘密鍵の解析で使用される数学ライブラリがエラーを検出した場合に返されます。
- MP_RANGE 秘密鍵の解析で使用される数学ライブラリがエラーを検出した場合に返されます。

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
}
```

B.1.2.30 function wc_EccPublicKeyDecode

```
int wc_EccPublicKeyDecode(
    const byte * input,
    word32 * inOutIdx,
    ecc_key * key,
    word32 inSz
)
```

この関数は入力バッファの ECC 公開鍵を ASN シーケンスをデコードして取り出します。

Parameters:

- **input** DER 形式の公開鍵を含んだバッファへのポインタ
- **inOutIdx** バッファの読み出し位置インデクス値を保持している変数へのポインタ (入力時)。出力時にはこの変数に解析済みのバッファのインデクス値が格納されます。
- **key** ecc_key 構造体へのポインタ
- **inSz** 入力バッファのサイズ

See: wc_ecc_import_x963

Return:

- 0 処理成功時に返します。
- BAD_FUNC_ARG Returns いずれかの引数が NULL の場合に返します。
- ASN_PARSE_E 解析中にエラーが発生した場合に返します。
- ASN_ECC_KEY_E 鍵のインポートでエラーが発生した場合に返します。発生理由については wc_ecc_import_x963() を参照のこと。

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
}
```

B.1.2.31 function wc_EccPublicKeyToDer

```
int wc_EccPublicKeyToDer(
    ecc_key * key,
    byte * output,
    word32 inLen,
    int with_AlgCurve
)
```

この関数は ECC 公開鍵を DER 形式に変換します。処理したバッファのサイズを返します。変換して得られる DER 形式の ECC 公開鍵は出力バッファに格納されます。AlgCurve フラグの指定により、アルゴリズムと曲線情報をヘッダーに含めることができます。

Parameters:

- **key** ecc_key 構造体へのポインタ
- **output** 出力バッファへのポインタ

- **inLen** 出力バッファのサイズ
- **with_AlgCurve** アルゴリズムと曲線情報をヘッダーに含める際には１を指定

See:

- wc_EccKeyToDer
- wc_EccPrivateKeyDecode

Return:

- 成功時には処理したバッファのサイズを返します。
- BAD_FUNC_ARG 出力バッファ output あるいは ecc_key 構造体 key が NULL の場合に返します。
- LENGTH_ONLY_E ECC 公開鍵のサイズ取得に失敗した場合に返します。
- BUFFER_E 出力バッファが必要量より小さい場合に返します。

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
}
```

B.1.2.32 function wc_EccPublicKeyToDer_ex

```
int wc_EccPublicKeyToDer_ex(
    ecc_key * key,
    byte * output,
    word32 inLen,
    int with_AlgCurve,
    int comp
)
```

この関数は ECC 公開鍵を DER 形式に変換します。処理したバッファサイズを返します。変換された DER 形式の ECC 公開鍵は出力バッファに格納されます。AlgCurve フラグの指定により、アルゴリズムと曲線情報をヘッダーに含めることができます。comp パラメータは公開鍵を圧縮して出力するか否かを指定します。

Parameters:

- **key** ecc_key 構造体へのポインタ
- **output** 出力バッファへのポインタ
- **inLen** 出力バッファのサイズ
- **with_AlgCurve** アルゴリズムと曲線情報をヘッダーに含める際には１を指定
- **comp** 非ゼロ値の指定時には ECC 公開鍵は圧縮形式で出力されます。ゼロが指定された場合には非圧縮で出力されます。

See:

- wc_EccKeyToDer
- wc_EccPublicKeyDecode

Return:

- 0 成功時には処理したバッファのサイズを返します。

- BAD_FUNC_ARG 出力バッファ output あるいは ecc_key 構造体 key が NULL の場合に返します。
- LENGTH_ONLY_E ECC 公開鍵のサイズ取得に失敗した場合に返します。
- BUFFER_E 出力バッファが必要量より小さい場合に返します。

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
}
```

B.1.2.33 function wc_EncodeSignature

```
word32 wc_EncodeSignature(
    byte * out,
    const byte * digest,
    word32 digSz,
    int hashOID
)
```

この関数はデジタル署名をエンコードして出力バッファに出力し、生成された署名のサイズを返します。

Parameters:

- **out** エンコードした署名データを出力する先のバッファへのポインタ
- **digest** 署名データのエンコードに使用するダイジェストへのポインタ
- **digSz** ダイジェストを含んでいるバッファのサイズ
- **hashOID** ハッシュタイプを示すオブジェクト ID。有効な値は: SHAh, SHA256h, SHA384h, SHA512h, MD2h, MD5h, DESb, DES3b, CTC_MD5wRSA, CTC_SHAwRSA, CTC_SHA256wRSA, CTC_SHA384wRSA, CTC_SHA512wRSA, CTC_SHAwECDSA, CTC_SHA256wECDSA, CTC_SHA384wECDSA, と CTC_SHA512wECDSA。

See: none

Return: 成功時には署名を出力バッファに出力し、出力したサイズを返します。

```
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);
```

B.1.2.34 function wc_GetCTC_HashOID

```
int wc_GetCTC_HashOID(
    int type
)
```

この関数はハッシュタイプに対応したハッシュ OID を返します。例えば、ハッシュタイプが “WC_SHA512” の場合、この関数は “SHA512h” を対応するハッシュ OID として返します。

Parameters:

- **type** ハッシュタイプ。指定可能なタイプ: WC_MD5, WC_SHA, WC_SHA256, WC_SHA384, WC_SHA512, WC_SHA3_224, WC_SHA3_256, WC_SHA3_384, WC_SHA3_512

See: none

Return:

- 成功時には指定されたハッシュタイプと対応するハッシュ OID を返します。
- 0 認識できないハッシュタイプが引数として指定された場合に返します。

Example

```
int hashOID;

hashOID = wc_GetCTC_HashOID(WC_SHA512);
if (hashOID == 0) {
    // WOLFSSL_SHA512 not defined
}
```

B.1.2.35 function wc_SetCert_Free

```
void wc_SetCert_Free(
    Cert * cert
)
```

この関数はキャッシュされていた Cert 構造体で使用されたメモリとリソースをクリーンアップします。WOLFSSL_CERT_GEN_CACHE が定義されている場合には DecodedCert 構造体が Cert 構造体内部にキャッシュされ、後続する set 系関数の呼び出しの都度 DecodedCert 構造体がパースされることを防ぎます。

Parameters:

- **cert** 未初期化の Cert 構造体へのポインタ

See:

- wc_SetAuthKeyIdFromCert
- wc_SetIssuerBuffer
- wc_SetSubjectBuffer
- wc_SetSubjectRaw
- wc_SetIssuerRaw
- wc_SetAltNamesBuffer
- wc_SetDatesBuffer

Return:

- 0 成功時に返されます。
- BAD_FUNC_ARG 引数として無効な値が渡された場合に返されます。

Example

```
Cert cert; // Initialized certificate structure

wc_SetCert_Free(&cert);
```

B.1.2.36 function wc_GetPkcs8TraditionalOffset

```
int wc_GetPkcs8TraditionalOffset(
    byte * input,
    word32 * inOutIdx,
    word32 sz
)
```

この関数は PKCS#8 の暗号化されていないバッファ内部の従来の秘密鍵の開始位置を検出して返します。

Parameters:

- **input** PKCS#8 の暗号化されていない秘密鍵を保持するバッファへのポインタ
- **inOutIdx** バッファのインデクス位置を保持する変数へのポインタ。入力時にはこの変数の内容はバッファ内部の PKCS#8 の開始位置を示します。出力時には、秘密鍵の先頭位置を保持します。
- **sz** 入力バッファのサイズ

See:

- wc_CreatePKCS8Key
- wc_EncryptPKCS8Key
- wc_DecryptPKCS8Key
- wc_CreateEncryptedPKCS8Key

Return:

- 成功時には従来の秘密鍵の長さを返します。
- エラー時には負の整数値を返します。

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.
```

B.1.2.37 function wc_CreatePKCS8Key

```
int wc_CreatePKCS8Key(
    byte * out,
    word32 * outSz,
    byte * key,
    word32 keySz,
    int algoID,
    const byte * curveOID,
    word32 oidSz
)
```

この関数は DER 形式の秘密鍵を入力とし、RKCS#8 形式に変換します。また、PKCS#12 のシュロ — ディットキーバッグの作成にも使用できます。RFC5208 を参照のこと。

Parameters:

- **out** 結果の出力先バッファへのポインタ。NULL の場合には必要な出力先バッファのサイズが outSz に格納されます。
- **outSz** 出力先バッファのサイズ
- **key** 従来の DER 形式の秘密鍵を含むバッファへのポインタ
- **keySz** 秘密鍵を含むバッファのサイズ
- **algoID** アルゴリズム ID (RSAk 等の)

- **curveOID** ECC 曲線 OID。RSA 鍵を使用する場合には NULL にすること。
- **oidSz** ECC 曲線 OID のサイズ。curveOID が NULL の場合には 0 にすること。

See:

- wc_GetPkcs8TraditionalOffset
- wc_EncryptPKCS8Key
- wc_DecryptPKCS8Key
- wc_CreateEncryptedPKCS8Key

Return:

- 成功時には出力された PKCS#8 鍵のサイズを返します。
- LENGTH_ONLY_E 出力先バッファ out が NULL として渡された場合にはこのエラーコードが返され、outSz に必要な出力バッファのサイズが格納されます。
- エラー時には負の整数値が返されます。

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);
```

B.1.2.38 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
)
```

この関数は暗号化されていない PKCS#8 の DER 形式の鍵 (例えば wc_CreatePKCS8Key で生成された鍵) を受け取り、PKCS#8 暗号化形式に変換します。結果として得られた暗号化鍵は wc_DecryptPKCS8Key を使って復号できます。RFC5208 を参照してください。

Parameters:

- **key** 従来の DER 形式の鍵を含んだバッファへのポインタ
- **keySz** 鍵を含んだバッファのサイズ
- **out** 出力結果を格納する先のバッファへのポインタ。NULL の場合には必要な出力先バッファのサイズが outSz に格納されます。
- **outSz** 出力先バッファのサイズ
- **password** パスワードベース暗号化アルゴリズムに使用されるパスワード
- **passwordSz** パスワードのサイズ (NULL 終端文字は含まない)
- **vPKCS** 使用する PKCS のバージョン番号。1 は PKCS12 か PKCS5。
- **pbeOid** パスワードベース暗号化スキームの OID(PBES2 あるいは RFC2898 A.3 にある OID の一つ)
- **encAlgId** 暗号化アルゴリズム ID(例えば AES256CBCb)。
- **salt** ソルト。NULL の場合はランダムに選定したソルトが使用されます。
- **saltSz** ソルトサイズ。salt に NULL を渡した場合には 0 を指定できます。
- **itt** 鍵導出のための繰り返し回数
- **rng** 初期化済みの WC_RNG 構造体へのポインタ
- **heap** 動的メモリ確保のためのヒープ。NULL 指定も可。

See:

- wc_GetPkcs8TraditionalOffset
- wc_CreatePKCS8Key
- wc_DecryptPKCS8Key
- wc_CreateEncryptedPKCS8Key

Return:

- 成功時には出力先バッファに出力された暗号化鍵のサイズを返します。
- LENGTH_ONLY_E 出力先バッファ out が NULL として渡された場合にはこのエラーコードが返され、outSz に必要な出力バッファのサイズが格納されます。
- エラー時には負の整数値が返されます。

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);
```

B.1.2.39 function wc_DecryptPKCS8Key

```
int wc_DecryptPKCS8Key(
    byte * input,
    word32 sz,
```

```
    const char * password,
    int passwordSz
)
```

この関数は暗号化された PKCS#8 の DER 形式の鍵を受け取り、復号して PKCS#8 DER 形式に変換します。wc_EncryptPKCS8Key によって行われた暗号化を元に戻します。RFC5208 を参照してください。入力データは復号データによって上書きされます。

Parameters:

- **input** 入力時には暗号化された PKCS#8 鍵データを含みます。出力時には復号された PKCS#8 鍵データを含みます。
- **sz** 入力バッファのサイズ
- **password** 鍵を暗号化する際のパスワード
- **passwordSz** パスワードのサイズ (NULL 終端文字は含まない)

See:

- wc_GetPkcs8TraditionalOffset
- wc_CreatePKCS8Key
- wc_EncryptPKCS8Key
- wc_CreateEncryptedPKCS8Key

Return:

- 成功時には復号データの長さを返します。
- エラー発生時には負の整数値を返します。

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);
```

B.1.2.40 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
)
```

この関数は従来の DER 形式の鍵を PKCS#8 フォーマットに変換し、暗号化を行います。この処理には wc_CreatePKCS8Key と wc_EncryptPKCS8Key を使用します。

Parameters:

- **key** 従来の DER 形式の鍵を含んだバッファへのポインタ
- **keySz** 鍵を含んだバッファのサイズ
- **out** 結果を出力する先のバッファへのポインタ。NULL が指定された場合には、必要なバッファサイズが outSz に格納されます。
- **outSz** 結果を出力する先のバッファのサイズ
- **password** パスワードベース暗号アルゴリズムに使用されるパスワード
- **passwordSz** パスワードのサイズ (NULL 終端文字は含まない)
- **vPKCS** 使用する PKCS のバージョン番号。1 は PKCS12 か PKCS5。
- **pbeOid** パスワードベース暗号化スキームの OID(PBES2 あるいは RFC2898 A.3 にある OID の一つ)
- **encAlgId** 暗号化アルゴリズム ID(例えば AES256CBCb)。
- **salt** ソルト。NULL の場合はランダムに選定したソルトが使用されます。
- **saltSz** ソルトサイズ。salt に NULL を渡した場合には 0 を指定できます。
- **itt** 鍵導出のための繰り返し回数
- **rng** 初期化済みの WC_RNG 構造体へのポインタ
- **heap** 動的メモリ確保のためのヒープ。NULL 指定も可。

See:

- wc_GetPkcs8TraditionalOffset
- wc_CreatePKCS8Key
- wc_EncryptPKCS8Key
- wc_DecryptPKCS8Key

Return:

- 成功時には出力した暗号化鍵のサイズを返します。
- LENGTH_ONLY_E もし出力用バッファ out に NULL が渡された場合に返されます。その際には outSz 変数に必要な出力用バッファサイズを格納します。
- エラー発生時には負の整数値を返します。

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);
```

B.1.2.41 function wc_InitDecodedCert

```
void wc_InitDecodedCert(
    struct DecodedCert * cert,
    const byte * source,
    word32 inSz,
    void * heap
)
```

この関数は cert 引数で与えられた DecodedCert 構造体を初期化します。DER 形式の証明書を含んでいる source 引数の指すポインタから証明書サイズ inSz の長さを内部に保存します。この関数の後に呼び出される wc_ParseCert によって証明書が解析されます。

Parameters:

- **cert** DecodedCert 構造体へのポインタ
- **source** DER 形式の証明書データへのポインタ
- **inSz** 証明書データのサイズ（バイト数）
- **heap** 動的メモリ確保のためのヒープ。NULL 指定も可。

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);
```

B.1.2.42 function wc_ParseCert

```
int wc_ParseCert(
    DecodedCert * cert,
    int type,
    int verify,
    void * cm
)
```

この関数は DecodedCert 構造体に保存されている DER 形式の証明書を解析し、その構造体に各種フィールドを設定します。DecodedCert 構造体は wc_InitDecodedCert を呼び出して初期化しておく必要があります。この関数はオプションで CertificateManager 構造体へのポインタを受け取り、CA が証明書マネジャーで検索できた場合には、その CA に関する情報も DecodedCert 構造体に追加設定します。

Parameters:

- **cert** 初期化済みの DecodedCert 構造体へのポインタ。
- **type** 証明書タイプ。タイプの設定値については asn_public.h の CertType enum 定義を参照してください。
- **verify** 呼び出し側が証明書の検証を求めていることを指示すフラグです。
- **cm** CertificateManager 構造体へのポインタ。オプションで指定可。NULL でも可。

See:

- wc_InitDecodedCert
- wc_FreeDecodedCert

Return:

- 0 成功時に返します。
- エラー発生時には負の整数値を返します。

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");
}
```

B.1.2.43 function wc_FreeDecodedCert

```
void wc_FreeDecodedCert(
    struct DecodedCert * cert
)
```

この関数は wc_InitDecodedCert で初期化済みの DecodedCert 構造体を解放します。

Parameters:

- **cert** 初期化済みの DecodedCert 構造体へのポインタ。

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);
```

B.1.2.44 function wc_SetTimeCb

```
int wc_SetTimeCb(
    wc_time_cb f
)
```

この関数はタイムコールバック関数を登録します。wolfSSL が現在時刻を必要としたタイミングでこのコールバックを呼び出します。このタイムコールバック関数のプロトタイプ（シグネチャ）は C 標準ライブラリの “time” 関数と同一です。

Parameters:

- **f** タイムコールバック関数ポインタ

See: wc_Time

Return: 0 成功時に返します。

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
}
```

B.1.2.45 function wc_Time

```
time_t wc_Time(
    time_t * t
)
```

この関数は現在時刻を取得します。デフォルトで XTIME マクロ関数を使います。このマクロ関数はプラットフォーム依存です。ユーザーはこのマクロの代わりに wc_SetTimeCb でタイムコールバック関数を使うように設定することができます

Parameters:

- **t** 現在時刻を返却するオプションの time_t 型変数。

See: wc_SetTimeCb

Return: 成功時には現在時刻を返します。

Example

```
time_t currentTime = 0;
currentTime = wc_Time(NULL);
wc_Time(&currentTime);
```

B.1.2.46 function wc_SetCustomExtension

```
int wc_SetCustomExtension(
    Cert * cert,
    int critical,
    const char * oid,
    const byte * der,
    word32 derSz
)
```

この関数は X.509 証明書にカスタム拡張を追加します。注: この関数に渡すポインタ引数が保持する内容は証明書が生成されるまで変更されてはいけません。この関数ではポインタが指す先の内容は別のバッファには複製しません。

Parameters:

- **cert** 初期化済みの DecodedCert 構造体へのポインタ。
- **critical** 0 が指定された場合には追加する拡張はクリティカルとはマークされません。0 以外が指定された場合にはクリティカルとマークされます。
- **oid** ドット区切りの oid 文字列。例えば、“1.2.840.10045.3.1.7”
- **der** 拡張情報の DER エンコードされた内容を含むバッファへのポインタ。
- **derSz** DER エンコードされた内容を含むバッファのサイズ

See:

- wc_InitCert

- wc_SetUnknownExtCallback

Return:

- 0 成功時に返します。
- エラー発生時には負の整数値を返します。

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.
```

B.1.2.47 function wc_SetUnknownExtCallback

```
int wc_SetUnknownExtCallback(
    DecodedCert * cert,
    wc_UnknownExtCallback cb
)
```

この関数は wolfSSL が証明書の解析中に未知の X.509 拡張に遭遇した際に呼び出すコールバック関数を登録します。コールバック関数のプロトタイプは使用例を参照してください。

Parameters:

- **cert** コールバック関数を登録する対象の DecodedCert 構造体へのポインタ。
- **cb** 登録されるコールバック関数ポインタ

See:

- ParseCert
- wc_SetCustomExtension

Return:

- 0 成功時に返します。
- エラー発生時には負の整数値を返します。

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: OID を構成するドット区切りの数を格納した配列
// oidSz: oid 内の値の数
// crit: 拡張がクリティカルとマークされているか
// der: DER エンコードされている拡張の内容
// derSz: 拡張の内容のサイズ
int myCustomExtCallback(const word16* oid, word32 oidSz, int crit,
                        const unsigned char* der, word32 derSz) {

    // 拡張を解析するロジックはここに記述します

    // NOTE: コールバック関数から 0 を返すと wolfSSL に対してこの拡張を受け入れ可能と
    // 表明することになります。この拡張を処理できると判断できない場合にはエラーを
    // 返してください。クリティカルとマークされている未知の拡張に遭遇した際の標準的
    // な振る舞いは ASN_CRIT_EXT_E を返すことです。
    // 簡潔にするためにこの例ではすべての拡張情報を受け入れ可としていますが、実際には実情に沿
    ↪  うようにロジックを追加してください。

    return 0;
}
```

B.1.2.48 function wc_CheckCertSigPubKey

```
int wc_CheckCertSigPubKey(
    const byte * cert,
    word32 certSz,
    void * heap,
    const byte * pubKey,
    word32 pubKeySz,
    int pubKeyOID
)
```

この関数は DER 形式の X.509 証明書の署名を与えられた公開鍵を使って検証します。公開鍵は DER 形式で全公開鍵情報を含んだものが求められます。

Parameters:

- **cert** DER 形式の X.509 証明書を含んだバッファへのポインタ
- **certSz** 証明書を含んだバッファのサイズ
- **heap** 動的メモリ確保のためのヒープ。NULL 指定も可。
- **pubKey** DER 形式の公開鍵を含んだバッファへのポインタ
- **pubKeySz** 公開鍵を含んだバッファのサイズ
- **pubKeyOID** 公開鍵のアルゴリズムを特定する OID(すなわち: ECDSAk, DSAk や RSAk)

Return:

- 0 成功時に返します。
- エラー発生時には負の整数値を返します。

B.1.2.49 function wc_Asn1PrintOptions_Init

```
int wc_Asn1PrintOptions_Init(
```

```
    Asn1PrintOptions * opts
)
```

この関数は Asn1PrintOptions 構造体を初期化します。

Parameters:

- **opts** プリントのための Asn1PrintOptions 構造体へのポインタ

See:

- wc_Asn1PrintOptions_Set
- wc_Asn1_PrintAll

Return:

- 0 成功時に返します。
- BAD_FUNC_ARG asn1 が NULL の場合に返されます。

Example

```
Asn1PrintOptions opt;

// Initialize ASN.1 print options before use.
wc_Asn1PrintOptions_Init(&opt);
```

B.1.2.50 function wc_Asn1PrintOptions_Set

```
int wc_Asn1PrintOptions_Set(
    Asn1PrintOptions * opts,
    enum Asn1PrintOpt opt,
    word32 val
)
```

この関数は Asn1PrintOptions 構造体にプリント情報を設定します。

Parameters:

- **opts** Asn1PrintOptions 構造体へのポインタ
- **opt** 設定する情報へのポインタ
- **val** 設定値

See:

- wc_Asn1PrintOptions_Init
- wc_Asn1_PrintAll

Return:

- 0 成功時に返します。
- BAD_FUNC_ARG asn1 が NULL の場合に返されます。
- BAD_FUNC_ARG val が範囲外の場合に返されます。

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);
```

B.1.2.51 function wc_Asn1_Init

```
int wc_Asn1_Init(
    Asn1 * asn1
)
```

この関数は Asn1 構造体を初期化します。

Parameters:

- **asn1** Asn1 構造体へのポインタ

See:

- wc_Asn1_SetFile
- wc_Asn1_PrintAll

Return:

- 0 成功時に返します。
- BAD_FUNC_ARG asn1 が NULL の場合に返されます。

Example

```
Asn1 asn1;

// Initialize ASN.1 parse object before use.
wc_Asn1_Init(&asn1);
```

B.1.2.52 function wc_Asn1_SetFile

```
int wc_Asn1_SetFile(
    Asn1 * asn1,
    XFILE file
)
```

この関数は出力先として使用するファイルを Asn1 構造体にセットします。

Parameters:

- **asn1** Asn1 構造体へのポインタ
- **file** プリント先のファイル

See:

- wc_Asn1_Init
- wc_Asn1_PrintAll

Return:

- 0 成功時に返します。
- BAD_FUNC_ARG asn1 が NULL の場合に返されます。
- BAD_FUNC_ARG file が XBADFILE の場合に返されます。.

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);
```

B.1.2.53 function wc_Asn1_PrintAll

```
int wc_Asn1_PrintAll(
    Asn1 * asn1,
    Asn1PrintOptions * opts,
    unsigned char * data,
    word32 len
)
```

ASN.1 アイテムをプリントします。

Parameters:

- **asn1** Asn1 構造体へのポインタ
- **opts** Asn1PrintOptions 構造体へのポインタ
- **data** BER/DER 形式のプリント対象データへのポインタ
- **len** プリント対象データのサイズ（バイト数）

See:

- wc_Asn1_Init
- wc_Asn1_SetFile

Return:

- 0 成功時に返します。
- BAD_FUNC_ARG asn1 か opts が NULL の場合に返されます。
- ASN_LEN_E ASN.1 アイテムが長すぎる場合に返されます。
- ASN_DEPTH_E 終了オフセットが無効の場合に返されます。
- ASN_PARSE_E 全の ASN.1 アイテムの解析が完了できなかった場合に返されます。

```
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.2.1 Functions

	Name
int	**Base64_Decode**(const byte * in, word32 inLen, byte * out, word32 * outLen) この機能は、与えられた BASS64 符号化入力、IN、および出力バッファを出力バッファ OUT に格納します。また、変数 outlen 内の出力バッファに書き込まれたサイズも設定します。
int	**Base64_Encode**(const byte * in, word32 inLen, byte * out, word32 * outLen) この機能は与えられた入力を符号化し、符号化結果を出力バッファ OUT に格納します。エスケープ%0A 行末の代わりに、従来の' N' 行の終わりを持つデータを書き込みます。正常に完了すると、この機能はまた、出力バッファに書き込まれたバイト数に統一されます。
int	**Base64_EncodeEsc**(const byte * in, word32 inLen, byte * out, word32 * outLen) この機能は与えられた入力を符号化し、符号化結果を出力バッファ OUT に格納します。それは' n " 行の終わりではなく、%0a エスケープ行の終わりを持つデータを書き込みます。正常に完了すると、この機能はまた、出力バッファに書き込まれたバイト数に統一されます。
int	**Base64_Encode_NoNl**(const byte * in, word32 inLen, byte * out, word32 * outLen) この機能は与えられた入力を符号化し、符号化結果を出力バッファ OUT に格納します。それは新しい行なしでデータを書き込みます。正常に完了すると、この関数はまた、出力バッファに書き込まれたバイト数に統一されたものを設定します
int	**Base16_Decode**(const byte * in, word32 inLen, byte * out, word32 * outLen) この機能は、与えられた BASE16 符号化入力、IN、および出力バッファへの結果を記憶する。また、変数 outlen 内の出力バッファに書き込まれたサイズも設定します。
int	**Base16_Encode**(const byte * in, word32 inLen, byte * out, word32 * outLen)BASE16 出力へのエンコード入力。

B.2.2 Functions Documentation

B.2.2.1 function Base64_Decode

```
int Base64_Decode(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 * outLen
)
```

この機能は、与えられた BASS64 符号化入力、IN、および出力バッファを出力バッファ OUT に格納します。また、変数 outlen 内の出力バッファに書き込まれたサイズも設定します。

Parameters:

- **in** デコードする入力バッファへのポインタ
- **inLen** デコードする入力バッファの長さ
- **out** デコードされたメッセージを保存する出力バッファへのポインタ *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
}
```

See:

- Base64_Encode
- Base16_Decode

Return:

- 0 Base64 エンコード入力の復号化に成功したときに返されます
- BAD_FUNC_ARG 復号化された入力を保存するには、出力バッファが小さすぎる場合は返されます。
- ASN_INPUT_E 入力バッファ内の文字が BASE64 範囲（[A-ZA-Z0-9 + / =]）の外側にある場合、または BASE64 エンコード入力に無効な行が終了した場合

B.2.2.2 function Base64_Encode

```
int Base64_Encode(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 * outLen
)
```

この機能は与えられた入力を符号化し、符号化結果を出力バッファ OUT に格納します。エスケープ%0A 行末の代わりに、従来の' N' 行の終わりを持つデータを書き込みます。正常に完了すると、この機能はまた、出力バッファに書き込まれたバイト数に統一されます。

Parameters:

- **in** エンコードする入力バッファへのポインタ
- **inLen** エンコードする入力バッファの長さ
- **out** エンコードされたメッセージを保存する出力バッファへのポインタ *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
}
```

See:

- Base64_EncodeEsc
- Base64_Decode

Return:

- 0 Base64 エンコード入力の復号化に成功したときに返されます
- BAD_FUNC_ARG 出力バッファが小さすぎてエンコードされた入力を保存する場合は返されます。
- BUFFER_E 出力バッファがエンコード中に部屋の外に実行された場合に返されます。

B.2.2.3 function Base64_EncodeEsc

```
int Base64_EncodeEsc(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 * outLen
)
```

この機能は与えられた入力を符号化し、符号化結果を出力バッファ OUT に格納します。それは' n " 行の終わりではなく、%0a エスケープ行の終わりを持つデータを書き込みます。正常に完了すると、この機能はまた、出力バッファに書き込まれたバイト数に統一されます。

Parameters:

- **in** エンコードする入力バッファへのポインタ
- **inLen** エンコードする入力バッファの長さ
- **out** エンコードされたメッセージを保存する出力バッファへのポインタ *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
}
```

See:

- Base64_Encode
- Base64_Decode

Return:

- 0 Base64 エンコード入力の復号化に成功したときに返されます
- BAD_FUNC_ARG 出力バッファが小さすぎてエンコードされた入力を保存する場合は返されます。
- BUFFER_E 出力バッファがエンコード中に部屋の外に実行された場合に返されます。
- ASN_INPUT_E 入力メッセージのデコードの処理中にエラーが発生した場合

B.2.2.4 function Base64_Encode_NoNl

```
int Base64_Encode_NoNl(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 * outLen
)
```

この機能は与えられた入力を符号化し、符号化結果を出力バッファ OUT に格納します。それは新しい行なしでデータを書き込みます。正常に完了すると、この関数はまた、出力バッファに書き込まれたバイト数に統一されたものを設定します

Parameters:

- **in** エンコードする入力バッファへのポインタ

- **inLen** エンコードする入力バッファの長さ
- **out** エンコードされたメッセージを保存する出力バッファへのポインタ *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
}
```

See:

- Base64_Encode
- Base64_Decode

Return:

- 0 Base64 エンコード入力の復号化に成功したときに返されます
- BAD_FUNC_ARG 出力バッファが小さすぎてエンコードされた入力を保存する場合は返されます。
- BUFFER_E 出力バッファがエンコード中に部屋の外に実行された場合に返されます。
- ASN_INPUT_E 入力メッセージのデコードの処理中にエラーが発生した場合

B.2.2.5 function Base16_Decode

```
int Base16_Decode(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 * outLen
)
```

この機能は、与えられた BASE16 符号化入力、IN、および出力バッファへの結果を記憶する。また、変数 outlen 内の出力バッファに書き込まれたサイズも設定します。

Parameters:

- **in** デコードする入力バッファへのポインタ
- **inLen** デコードする入力バッファの長さ
- **out** デコードされたメッセージを保存する出力バッファへのポインタ *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
}
```

See:

- Base64_Encode
- Base64_Decode
- Base16_Encode

Return:

- 0 Base16 エンコード入力の復号にうまく復号化したときに返されます
- BAD_FUNC_ARG 出力バッファが復号化された入力を保存するにも小さすぎる場合、または入力長が 2 つの倍数でない場合に返されます。
- ASN_INPUT_E 入力バッファ内の文字が BASE16 の範囲外にある場合は返されます（[0-9a-f]）

B.2.2.6 function Base16_Encode

```
int Base16_Encode(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 * outLen
)
```

BASE16 出力へのエンコード入力。

Parameters:

- **in** エンコードされる入力バッファへのポインタ。
- **inLen** 入力バッファの長さ
- **out** 出力バッファへのポインタ。*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
}
```

See:

- Base64_Encode
- Base64_Decode
- Base16_Decode

Return:

- 0 成功
- BAD_FUNC_ARG IN、OUT、または outlen が NULL の場合、または outlen が Inlen Plus 1 を超えている場合は返します。

B.3 Compression

B.3.1 Functions

	Name
int	**wc_Compress**(byte * out, word32 outSz, const byte * in, word32 inSz, word32 flags) この関数は、ハフマン符号化を用いて与えられた入力データを圧縮し、出力を OUT に格納する。出力バッファは、圧縮が可能でないことが存在するため、出力バッファが入力バッファよりも大きいはずです。これはまだルックアップテーブルを必要とします。出力バッファに対して SRCSZ + 0.1%+ 12 を割り当てることをお勧めします。
int	**wc_DeCompress**(byte * out, word32 outSz, const byte * in, word32 inSz) この関数は、ハフマン符号化を用いて所定の圧縮データを解凍し、出力を OUT に格納する。

B.3.2 Functions Documentation

B.3.2.1 function wc_Compress

```
int wc_Compress(
    byte * out,
    word32 outSz,
    const byte * in,
    word32 inSz,
    word32 flags
)
```

この関数は、ハフマン符号化を用いて与えられた入力データを圧縮し、出力を OUT に格納する。出力バッファは、圧縮が可能でないことが存在するため、出力バッファが入力バッファよりも大きいはずです。これはまだルックアップテーブルを必要とします。出力バッファに対して SRCSZ + 0.1%+ 12 を割り当てることをお勧めします。

Parameters:

- **out** 圧縮データを格納する出力バッファへのポインタ
- **outSz** 出力バッファで保存されているサイズ
- **in** 圧縮するメッセージを含むバッファへのポインタ
- **inSz** 圧縮する入力メッセージのサイズ *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
}
```

See: wc_DeCompress

Return:

- On 入力データの圧縮に成功し、出力バッファに格納されているバイト数を返します。
- COMPRESS_INIT_E 圧縮のためにストリームの初期化中にエラーがある場合
- COMPRESS_E 圧縮中にエラーが発生した場合は返されます

B.3.2.2 function wc_DeCompress

```
int wc_DeCompress(
    byte * out,
    word32 outSz,
    const byte * in,
    word32 inSz
)
```

この関数は、ハフマン符号化を用いて所定の圧縮データを解凍し、出力を OUT に格納する。

Parameters:

- **out** 解凍されたデータを格納する出力バッファへのポインタ
- **outSz** 出力バッファで保存されているサイズ
- **in** 解凍するメッセージを含むバッファへのポインタ *Example*

```
byte compressed[] = { // initialize compressed message };
byte decompressed[MAX_MESSAGE_SIZE];
```

```
if( wc_DeCompress(decompressed, sizeof(decompressed),
compressed, sizeof(compressed)) != 0 ) {
    // error decompressing data
}
```

See: wc_Compress

Return:

- Success 入力データの解凍に成功した場合は、出力バッファに格納されているバイト数を返します。
- COMPRESS_INIT_E: 圧縮のためにストリームの初期化中にエラーがある場合
- COMPRESS_E: 圧縮中にエラーが発生した場合は返されます

B.4 Error Reporting

B.4.1 Functions

	Name
void	**wc_ErrorString**(int err, char * buff) この関数は、特定のバッファ内の特定のエラーコードのエラー文字列を格納します。
const char *	**wc_GetErrorString**(int error) この関数は、特定のエラーコードのエラー文字列を返します。

B.4.2 Functions Documentation

B.4.2.1 function wc_ErrorString

```
void wc_ErrorString(
    int err,
    char * buff
)
```

この関数は、特定のバッファ内の特定のエラーコードのエラー文字列を格納します。

Parameters:

- **error** 文字列を取得するためのエラーコード *Example*

```
char errorMsg[WOLFSSL_MAX_ERROR_SZ];
int err = wc_some_function();

if( err != 0) { // error occurred
    wc_ErrorString(err, errorMsg);
}
```

See: wc_GetErrorString

Return: none いいえ返します。

B.4.2.2 function wc_GetErrorString

```
const char * wc_GetErrorString(
    int error
)
```

この関数は、特定のエラーコードのエラー文字列を返します。

See: wc_ErrorString

Return: string エラーコードのエラー文字列を文字列リテラルとして返します。*Example*

```
char * errorMsg;
int err = wc_some_function();

if( err != 0) { // error occurred
    errorMsg = wc_GetErrorString(err);
}
```

B.5 IoT-Safe Module

More...

B.5.1 Functions

	Name
int	**wolfSSL_CTX_iotsafe_enable**(WOLFSSL_CTX * ctx) この関数は与えられたコンテキストでの IoT セーフサポートを有効にします。
int	**wolfSSL_iotsafe_on**(WOLFSSL * ssl, byte privkey_id, byte ecdh_keypair_slot, byte peer_pubkey_slot, byte peer_cert_slot) この関数は、IOT-SAFE TLS コールバックを特定の SSL セッションに接続します。
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) この関数は、IOT-SAFE TLS コールバックを特定の SSL セッションに接続します。これは、IOT セーフスロットの ID を参照で渡すことができ、ID フィールドの長さをパラメータ “id_size” で指定できます。
void	**wolfIoTSafe_SetCSIM_read_cb**(wolfSSL_IOTSafe_CSIM_read_cb rf)AT + CSIM コマンドのリードコールバックを関連付けます。この入力関数は通常、モデムと通信する UART チャネルの読み取りイベントに関連付けられています。読み取りコールバックが関連付けられているのは、同時に IoT-Safe サポートを使用するすべてのコンテキストのグローバルと変更です。*Example*
void	**wolfIoTSafe_SetCSIM_write_cb**(wolfSSL_IOTSafe_CSIM_write_cb wf)AT + CSIM コマンドの書き込みコールバックを関連付けます。この出力関数は通常、モデムと通信する UART チャネル上のライトイベントに関連付けられています。Write Callback が関連付けられているのは、同時に IoT-Safe サポートを使用するすべてのコンテキストのグローバルと変更です。*Example*

	Name
int	**wolfIoTSafe_GetRandom**(unsigned char * out, word32 sz)IOT セーフ機能 getrandom を使用して、指定されたサイズのランダムなバッファを生成します。この関数は、WolfCrypt RNG オブジェクトによって自動的に使用されます。
int	**wolfIoTSafe_GetCert**(uint8_t id, unsigned char * output, unsigned long sz)IOT-Safe アプレット上のファイルに保存されている証明書をインポートし、ローカルにメモリに保存します。1 バイトのファイル ID フィールドで動作します。
int	**wolfIoTSafe_GetCert_ex**(uint8_t * id, uint16_t id_sz, unsigned char * output, unsigned long sz)IOT-Safe アプレット上のファイルに保存されている証明書をインポートし、ローカルにメモリに保存します。 ref wolfiotsafe_getcert “wolfiotsafe_getcert” と同等です。ただし、2 バイト以上のファイル ID で呼び出すことができます。
int	**wc_iotsafe_ecc_import_public**(ecc_key * key, byte key_id)IOT セーフアプレットに格納されている ECC 256 ビットの公開鍵を ECC_Key オブジェクトにインポートします。
int	**wc_iotsafe_ecc_export_public**(ecc_key * key, byte key_id)ECC_KEY オブジェクトから IOT-SAFE アプレットへの書き込み可能なパブリックキースロットに ECC 256 ビット公開鍵をエクスポートします。
int	**wc_iotsafe_ecc_import_public_ex**(ecc_key * key, byte * key_id, word16 id_size)ECC_KEY オブジェクトから IOT-SAFE アプレットへの書き込み可能なパブリックキースロットに ECC 256 ビット公開鍵をエクスポートします。 ref WC_IOTSAFE_ECC_IMPORT_PUBLIC「WC_IOTSAFE_ECC_IMPORT_PUBLIC」と同等のものは、2 バイト以上のキー ID で呼び出すことができる点を除きます。
int	**wc_iotsafe_ecc_export_private**(ecc_key * key, byte key_id)ECC 256 ビットキーを ECC_KEY オブジェクトから IOT セーフアプレットに書き込み可能なプライベートキースロットにエクスポートします。
int	**wc_iotsafe_ecc_export_private_ex**(ecc_key * key, byte * key_id, word16 id_size)ECC 256 ビットキーを ECC_KEY オブジェクトから IOT セーフアプレットに書き込み可能なプライベートキースロットにエクスポートします。 ref WC_IOTSAFE_ECC_EXPORT_PRIVATE「WC_IOTSAFE_ECC_EXPORT_PRIVATE」を除き、2 バイト以上のキー ID を呼び出すことができる点を除き、

	Name
int	**wc_iotsafe_ecc_sign_hash**(byte * in, word32 inlen, byte * out, word32 * outlen, byte key_id) 事前計算された 256 ビットハッシュに署名して、IOT-SAFE アプレットに、以前に保存されたプライベートキー、またはプリプロビジョニングされています。
int	**wc_iotsafe_ecc_sign_hash_ex**(byte * in, word32 inlen, byte * out, word32 * outlen, byte * key_id, word16 id_size) 事前計算された 256 ビットハッシュに署名して、IOT-SAFE アプレットに、以前に保存されたプライベートキー、またはプリプロビジョニングされています。 ref wc_iotsafe_ecc_sign_hash “wc_iotsafe_ecc_sign_hash” と同等です。ただし、2 バイト以上のキー ID で呼び出すことができます。
int	**wc_iotsafe_ecc_verify_hash**(byte * sig, word32 siglen, byte * hash, word32 hashlen, int * res, byte key_id) 予め計算された 256 ビットハッシュに対する ECC シグネチャを、IOT-SAFE アプレット内のプリプロビジョニング、またはプロビジョニングされたプリプロビジョニングを使用します。結果は RES に書き込まれます。1 が有効で、0 が無効です。注：有効なテストに戻り値を使用しないでください。Res のみを使用してください。
int	**wc_iotsafe_ecc_verify_hash_ex**(byte * sig, word32 siglen, byte * hash, word32 hashlen, int * res, byte * key_id, word16 id_size) 予め計算された 256 ビットハッシュに対する ECC シグネチャを、IOT-SAFE アプレット内のプリプロビジョニング、またはプロビジョニングされたプリプロビジョニングを使用します。結果は RES に書き込まれます。1 が有効で、0 が無効です。注：有効なテストに戻り値を使用しないでください。Res のみを使用してください。 ref WC_IOTSAFE_ECC_VERIFY_HASH “WC_IOTSAFE_ECC_VERIFY_HASH” を除き、2 バイト以上のキー ID で呼び出すことができる点を除きます。
int	**wc_iotsafe_ecc_gen_k**(byte key_id)ECC 256 ビットのキーペアを生成し、それを（書き込み可能な）スロットに IOT セーフなアプレットに保存します。

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

B.5.3.1 function wolfSSL_CTX_iotsafe_enable

```
int wolfSSL_CTX_iotsafe_enable(
    WOLFSSL_CTX * ctx
)
```

この関数は与えられたコンテキストでの IoT セーフサポートを有効にします。

Parameters:

- **ctx** IOT セーフサポートを有効にする必要がある WOLFSSL_CTX オブジェクトへのポインタ

See:

- wolfSSL_iotsafe_on
- wolfIoTSafe_SetCSIM_read_cb
- wolfIoTSafe_SetCSIM_write_cb

Return: 0 成功した *Example*

```
WOLFSSL_CTX *ctx;
ctx = wolfSSL_CTX_new(wolfTLSv1_2_client_method());
if (!ctx)
    return NULL;
wolfSSL_CTX_iotsafe_enable(ctx);
```

B.5.3.2 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
)
```

この関数は、IOT-SAFE TLS コールバックを特定の SSL セッションに接続します。

Parameters:

- **ssl** コールバックが有効になる WolfSSL オブジェクトへのポインタ
- **privkey_id** ホストの秘密鍵を含む IOT セーフなアプレットスロットの ID
- **ecdh_keypair_slot** ECDH 鍵ペアを保存するための IoT 安全アプレットスロットの ID
- **peer_pubkey_slot** ECDH 用の他のエンドポイントの公開鍵を保存するための IOT-SAFE アプレットスロットの ID
- **peer_cert_slot** 検証のための他のエンドポイントの公開鍵を保存するための IOT セーフなアプレットスロットの ID

See:

- wolfSSL_iotsafe_on_ex
- wolfSSL_CTX_iotsafe_enable

Return:

- 0 成功すると
- NOT_COMPILED_IN habe_pk_callbacks が無効になっている場合 *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);
}
```

スロットの ID が 1 バイトの長さの場合、SSL セッションを IoT-Safe アプレットに接続するように呼び出す必要があります。IOT セーフスロットの ID が 2 バイト以上の場合、 REF WOLFSSL_IOTSAFE_ON_EX「WOLFSSL_IOTSAFE_ON_EX ()」を使用する必要があります。

B.5.3.3 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
)
```

この関数は、IOT-SAFE TLS コールバックを特定の SSL セッションに接続します。これは、IOT セーフスロットの ID を参照で渡すことができ、ID フィールドの長さをパラメータ “id_size” で指定できます。

Parameters:

- **ssl** コールバックが有効になる WolfSSL オブジェクトへのポインタ
- **privkey_id** ホストの秘密鍵を含む IoT セーフアプレットスロットの ID へのポインタ
- **ecdh_keypair_slot** ECDH 鍵ペアを保存する IOT-Safe アプレットスロットの ID へのポインタ
- **peer_pubkey_slot** ECDH 用の他のエンドポイントの公開鍵を保存する IOT セーフアプレットスロットの ID へのポインタ
- **peer_cert_slot** 検証のために他のエンドポイントの公開鍵を保存するための IOT-SAFE アプレットスロットの ID へのポインタ
- **id_size** 各スロット ID のサイズ

See:

- wolfSSL_iotsafe_on
- wolfSSL_CTX_iotsafe_enable

Return:

- 0 成功すると
- NOT_COMPILED_IN habe_pk_callbacks が無効になっている場合 *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);
}
```

B.5.3.4 function wolfIoTSafe_SetCSIM_read_cb

```
void wolfIoTSafe_SetCSIM_read_cb(
    wolfSSL_IOTSafe_CSIM_read_cb rf
)
```

AT + CSIM コマンドのリードコールバックを関連付けます。この入力関数は通常、モデムと通信する UART チャネルの読み取りイベントに関連付けられています。読み取りコールバックが関連付けられているのは、同時に IoT-Safe サポートを使用するすべてのコンテキストのグローバルと変更です。*Example*

See: wolfIoTSafe_SetCSIM_write_cb

```
// USART read function, defined elsewhere
int usart_read(char *buf, int len);

wolfIoTSafe_SetCSIM_read_cb(usart_read);
```

B.5.3.5 function wolfIoTSafe_SetCSIM_write_cb

```
void wolfIoTSafe_SetCSIM_write_cb(
    wolfSSL_IOTSafe_CSIM_write_cb wf
)
```

AT + CSIM コマンドの書き込みコールバックを関連付けます。この出力関数は通常、モデムと通信する UART チャネル上のライトイベントに関連付けられています。Write Callback が関連付けられているのは、同時に IoT-Safe サポートを使用するすべてのコンテキストのグローバルと変更です。*Example*

See: wolfIoTSafe_SetCSIM_read_cb

```
// USART write function, defined elsewhere
int usart_write(const char *buf, int len);
wolfIoTSafe_SetCSIM_write_cb(usart_write);
```

B.5.3.6 function wolfIoTSafe_GetRandom

```
int wolfIoTSafe_GetRandom(
    unsigned char * out,
    word32 sz
)
```

IOT セーフ機能 getrandom を使用して、指定されたサイズのランダムなバッファを生成します。この関数は、WolfCrypt RNG オブジェクトによって自動的に使用されます。

Parameters:

- **out** ランダムなバイトシーケンスが格納されているバッファ。
- **sz** 生成するランダムシーケンスのサイズ（バイト単位）

B.5.3.7 function wolfIoTSafe_GetCert

```
int wolfIoTSafe_GetCert(
    uint8_t id,
    unsigned char * output,
    unsigned long sz
)
```

IOT-Safe アプレット上のファイルに保存されている証明書をインポートし、ローカルにメモリに保存します。1 バイトのファイル ID フィールドで動作します。

Parameters:

- **id** 証明書が保存されている IOT セーフ・アプレットのファイル ID
- **output** 証明書がインポートされるバッファー
- **sz** バッファ出力で使用可能な最大サイズ

Return: the 輸入された証明書の長さ *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");
```

B.5.3.8 function wolfIoTSafe_GetCert_ex

```
int wolfIoTSafe_GetCert_ex(
    uint8_t * id,
    uint16_t id_sz,
    unsigned char * output,
```

```
    unsigned long sz
)
```

IOT-Safe アプレット上のファイルに保存されている証明書をインポートし、ローカルにメモリに保存します。 ref wolfiotsafe_getcert “wolfiotsafe_getcert” と同等です。ただし、2 バイト以上のファイル ID で呼び出すことができます。

Parameters:

- **id** 証明書が保存されている IOT-SAFE アプレットのファイル ID へのポインタ
- **id_sz** ファイル ID のサイズ：バイト数
- **output** 証明書がインポートされるバッファー
- **sz** バッファ出力で使用可能な最大サイズ

Return: the 輸入された証明書の長さ *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");
```

B.5.3.9 function wc_iotsafe_ecc_import_public

```
int wc_iotsafe_ecc_import_public(
    ecc_key * key,
    byte key_id
)
```

IOT セーフアプレットに格納されている ECC 256 ビットの公開鍵を ECC_Key オブジェクトにインポートします。

Parameters:

- **key** IOT-SAFE アプレットからインポートされたキーを含む ECC_KEY オブジェクト
- **id** 公開鍵が保存されている IOT セーフアプレットのキー ID

See:

- wc_iotsafe_ecc_export_public

- wc_iotsafe_ecc_export_private

Return: 0 成功すると

B.5.3.10 function wc_iotsafe_ecc_export_public

```
int wc_iotsafe_ecc_export_public(
    ecc_key * key,
    byte key_id
)
```

ECC_KEY オブジェクトから IOT-SAFE アプレットへの書き込み可能なパブリックキースロットに ECC 256 ビット公開鍵をエクスポートします。

Parameters:

- **key** エクスポートする鍵を含む ecc_key オブジェクト
- **id** 公開鍵が保存されている IOT セーフアプレットのキー ID

See:

- wc_iotsafe_ecc_import_public_ex
- wc_iotsafe_ecc_export_private

Return: 0 成功すると

B.5.3.11 function wc_iotsafe_ecc_import_public_ex

```
int wc_iotsafe_ecc_import_public_ex(
    ecc_key * key,
    byte * key_id,
    word16 id_size
)
```

ECC_KEY オブジェクトから IOT-SAFE アプレットへの書き込み可能なパブリックキースロットに ECC 256 ビット公開鍵をエクスポートします。 ref WC_IOTSAFE_ECC_IMPORT_PUBLIC「WC_IOTSAFE_ECC_IMPORT_PUBLIC」と同等のものは、2 バイト以上のキー ID で呼び出すことができる点を除きます。

Parameters:

- **key** エクスポートする鍵を含む ecc_key オブジェクト
- **id** 公開鍵が保存される IOT セーフアプレットのキー ID へのポインタ
- **id_size** キー ID サイズ

See:

- wc_iotsafe_ecc_import_public
- wc_iotsafe_ecc_export_private

Return: 0 成功すると

B.5.3.12 function wc_iotsafe_ecc_export_private

```
int wc_iotsafe_ecc_export_private(
    ecc_key * key,
    byte key_id
)
```

ECC 256 ビットキーを ECC_KEY オブジェクトから IOT セーフアプレットに書き込み可能なプライベートキースロットにエクスポートします。

Parameters:

- **key** エクスポートする鍵を含む ecc_key オブジェクト
- **id** 秘密鍵が保存される IOT セーフアプレットのキー ID

See:

- wc_iotsafe_ecc_export_private_ex
- wc_iotsafe_ecc_import_public
- wc_iotsafe_ecc_export_public

Return: 0 成功すると

B.5.3.13 function wc_iotsafe_ecc_export_private_ex

```
int wc_iotsafe_ecc_export_private_ex(
    ecc_key * key,
    byte * key_id,
    word16 id_size
)
```

ECC 256 ビットキーを ECC_KEY オブジェクトから IOT セーフアプレットに書き込み可能なプライベートキースロットにエクスポートします。 ref WC_IOTSAFE_ECC_EXPORT_PRIVATE「WC_IOTSAFE_ECC_EXPORT_PRIVATE」を除き、2 バイト以上のキー ID を呼び出すことができる点を除き、

Parameters:

- **key** エクスポートする鍵を含む ecc_key オブジェクト
- **id** 秘密鍵が保存される IOT セーフアプレットのキー ID へのポインタ
- **id_size** キー ID サイズ

See:

- wc_iotsafe_ecc_export_private
- wc_iotsafe_ecc_import_public
- wc_iotsafe_ecc_export_public

Return: 0 成功すると

B.5.3.14 function wc_iotsafe_ecc_sign_hash

```
int wc_iotsafe_ecc_sign_hash(
    byte * in,
    word32 inlen,
    byte * out,
    word32 * outlen,
    byte key_id
)
```

事前計算された 256 ビットハッシュに署名して、IOT-SAFE アプレットに、以前に保存されたプライベートキー、またはプリプロビジョニングされています。

Parameters:

- **in** サインするメッセージハッシュを含むバッファへのポインタ
- **inlen** 署名するメッセージの長さ
- **out** 生成された署名を保存するためのバッファ
- **outlen** 出力バッファの最大長。バイトを保存します
- **id** メッセージ署名の生成に成功したときに書き込まれたペイロードに署名するための秘密鍵を含むスロットの IOT セーフアプレットのキー ID

See:

- wc_iotsafe_ecc_sign_hash_ex
- wc_iotsafe_ecc_verify_hash
- wc_iotsafe_ecc_gen_k

Return: 0 成功すると

B.5.3.15 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
)
```

事前計算された 256 ビットハッシュに署名して、IOT-SAFE アプレットに、以前に保存されたプライベートキー、またはプリプロビジョニングされています。ref wc_iotsafe_ecc_sign_hash “wc_iotsafe_ecc_sign_hash” と同等です。ただし、2 バイト以上のキー ID で呼び出すことができます。

Parameters:

- **in** サインするメッセージハッシュを含むバッファへのポインタ
- **inlen** 署名するメッセージの長さ
- **out** 生成された署名を保存するためのバッファ
- **outlen** 出力バッファの最大長。バイトを保存します
- **id** 秘密鍵を含むスロットの IOT-SAFE アプレットのキー ID へのポインタメッセージ署名の生成に成功したときに書き込まれるペイロードに署名する
- **id_size** キー ID サイズ

See:

- wc_iotsafe_ecc_sign_hash
- wc_iotsafe_ecc_verify_hash
- wc_iotsafe_ecc_gen_k

Return: 0 成功すると

B.5.3.16 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
)
```

予め計算された 256 ビットハッシュに対する ECC シグネチャを、IOT-SAFE アプレット内のプリプロビジョニング、またはプロビジョニングされたプリプロビジョニングを使用します。結果は RES に書き込まれます。1 が有効で、0 が無効です。注：有効なテストに戻り値を使用しないでください。Res のみを使用してください。

Parameters:

- **sig** 検証する署名を含むバッファ

- **hash** 署名されたハッシュ（メッセージダイジェスト）
- **hashlen** ハッシュの長さ（オクテット）
- **res** 署名の結果、1 == 有効、0 == 無効

See:

- wc_iotsafe_ecc_verify_hash_ex
- wc_iotsafe_ecc_sign_hash
- wc_iotsafe_ecc_gen_k

Return:

- 0 成功すると（署名が無効であっても）
- < 故障の場合は 0

B.5.3.17 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
)
```

予め計算された 256 ビットハッシュに対する ECC シグネチャを、IOT-SAFE アプレット内のプリプロビジョニング、またはプロビジョニングされたプリプロビジョニングを使用します。結果は RES に書き込まれます。1 が有効で、0 が無効です。注：有効なテストに戻り値を使用しないでください。Res のみを使用してください。ref WC_IOTSAFE_ECC_VERIFY_HASH “WC_IOTSAFE_ECC_VERIFY_HASH” を除き、2 バイト以上のキー ID で呼び出すことができる点を除きます。

Parameters:

- **sig** 検証する署名を含むバッファ
- **hash** 署名されたハッシュ（メッセージダイジェスト）
- **hashlen** ハッシュの長さ（オクテット）
- **res** 署名の結果、1 == 有効、0 == 無効
- **key_id** パブリック ECC キーが IOT セーフアプレットに保存されているスロットの ID

See:

- wc_iotsafe_ecc_verify_hash
- wc_iotsafe_ecc_sign_hash
- wc_iotsafe_ecc_gen_k

Return:

- 0 成功すると（署名が無効であっても）
- < 故障の場合は 0

B.5.3.18 function wc_iotsafe_ecc_gen_k

```
int wc_iotsafe_ecc_gen_k(
    byte key_id
)
```

ECC 256 ビットのキーペアを生成し、それを（書き込み可能な）スロットに IOT セーフなアプレットに保存します。

Parameters:

- **key_id** ECC キーペアが IOT セーフアプレットに格納されているスロットの ID。
- **key_id** ECC キーペアが IOT セーフアプレットに格納されているスロットの ID。
- **id_size** キー ID サイズ

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 成功すると
- 0 成功すると

ECC 256 ビットのキーペアを生成し、それを（書き込み可能な）スロットに IOT セーフなアプレットに保存します。 ref wc_iotsafe_ecc_gen_k “wc_iotsafe_ecc_gen_k” と同等です。ただし、2 バイト以上のキー ID で呼び出すことができます。

B.6 Key and Cert Conversion

B.7 Logging

B.7.1 Functions

	Name
int	**wolfSSL_SetLoggingCb**(wolfSSL_Logging_cb log_function) この関数は、WolfSSL ログメッセージを処理するために使用されるロギングコールバックを登録します。デフォルトでは、システムが IT fprintf（）を STDERR にサポートしている場合は、この関数を使用することによって、ユーザーによって何でも実行できます。

B.7.2 Functions Documentation

B.7.2.1 function wolfSSL_SetLoggingCb

```
int wolfSSL_SetLoggingCb(
    wolfSSL_Logging_cb log_function
)
```

この関数は、WolfSSL ログメッセージを処理するために使用されるロギングコールバックを登録します。デフォルトでは、システムが IT fprintf（）を STDERR にサポートしている場合は、この関数を使用することによって、ユーザーによって何でも実行できます。

See:

- wolfSSL_Debugging_ON
- wolfSSL_Debugging_OFF

Return:

- Success 成功した場合、この関数は 0 を返します。
- BAD_FUNC_ARG 関数ポインタが提供されていない場合に返されるエラーです。*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.8.1 Functions

	Name
word32	**CheckRunTimeFastMath**(void) この関数は、整数の最大サイズのランタイム FastMath 設定をチェックします。FP_SIZE が正しく機能するために、FP_SIZE が各ライブラリーに一致しなければならないため、ユーザーが WolfCrypt ライブラリを独立して使用している場合に重要です。このチェックは CheckFastMathSettings () として定義されています。これは、CheckRuntimeFastMath と FP_SIZE を比較するだけで、ミスマッチがある場合は 0 を返します。
word32	**CheckRunTimeSettings**(void) この関数はコンパイル時クラスの設定をチェックします。設定が正しく機能するためのライブラリ間のライブラリ間で一致する必要があるため、ユーザーが WolfCrypt ライブラリを独立して使用している場合は重要です。このチェックは CheckCtcSettings () として定義されています。これは、CheckRuntimeSettings と CTC_Settings を比較するだけで、ミスマッチがある場合は 0、または 1 が一致した場合は 1 を返します。

B.8.2 Functions Documentation

B.8.2.1 function CheckRunTimeFastMath

```
word32 CheckRunTimeFastMath(
    void
)
```

この関数は、整数の最大サイズのランタイム FastMath 設定をチェックします。FP_SIZE が正しく機能するために、FP_SIZE が各ライブラリーに一致しなければならないため、ユーザーが WolfCrypt ライブラリを独立して使用している場合に重要です。このチェックは CheckFastMathSettings () として定義されていま

す。これは、CheckRuntimeFastMath と FP_SIZE を比較するだけで、ミスマッチがある場合は 0 を返します。

See: CheckRunTimeSettings

Return: FP_SIZE 数学ライブラリで利用可能な最大サイズに対応する FP_SIZE を返します。*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
```

B.8.2.2 function CheckRunTimeSettings

```
word32 CheckRunTimeSettings(
    void
)
```

この関数はコンパイル時クラスの設定をチェックします。設定が正しく機能するためのライブラリ間のライブラリ間で一致する必要があるため、ユーザーが WolfCrypt ライブラリを独立して使用している場合は重要です。このチェックは CheckCtcSettings（）として定義されています。これは、CheckRuntimeSettings と CTC_Settings を比較するだけで、ミスマッチがある場合は 0、または 1 が一致した場合は 1 を返します。

See: CheckRunTimeFastMath

Return: settings 実行時 CTC_SETTINGS（コンパイル時設定）を返します。*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.9.1 Functions

	Name
int	**wc_InitNetRandom**(const char * configFile, wnr_hmac_key hmac_cb, int timeout)Init Global WhiteWood Netrandom のコンテキスト
int	**wc_FreeNetRandom**(void) 無料の Global WhiteWood Netrandom コンテキスト。
int	**wc_InitRng**(WC_RNG *)RNG のシード（OS から）とキー暗号を取得します。割り当てられた RNG-> DRBG（決定論的ランダムビットジェネレータ）が割り当てられます（WC_FREERNG で割り当てられている必要があります）。これはブロッキング操作です。

	Name
int	**wc_RNG_GenerateBlock**(WC_RNG * rng, byte * b, word32 sz) 疑似ランダムデータの SZ バイトを出力にコピーします。必要に応じて RNG（ブロッキング）します。
WC_RNG *	**wc_rng_new**(byte * nonce, word32 nonceSz, void * heap) 新しい WC_RNG 構造を作成します。
int	**wc_FreeRng**(WC_RNG *)RNG が DRGB を安全に解放するために必要なときに呼び出されるべきです。ゼロと Xfrees RNG-DRBG。
WC_RNG *	**wc_rng_free**(WC_RNG * rng)RNG を安全に自由に解放するために RNG が不要になったときに呼び出されるべきです。*Example*
int	**wc_RNG_HealthTest**(int reseed, const byte * entropyA, word32 entropyASz, const byte * entropyB, word32 entropyBSz, byte * output, word32 outputSz)DRBG の機能を作成しテストします。

B.9.2 Functions Documentation

B.9.2.1 function wc_InitNetRandom

```
int wc_InitNetRandom(
    const char * configFile,
    wnr_hmac_key hmac_cb,
    int timeout
)
```

Init Global WhiteWood Netrandom のコンテキスト

Parameters:

- **configFile** 設定ファイルへのパス
- **hmac_cb** HMAC コールバックを作成するにはオプションです。*Example*

```
char* config = "path/to/config/example.conf";
int time = // Some sufficient timeout value;

if (wc_InitNetRandom(config, NULL, time) != 0)
{
    // Some error occurred
}
```

See: wc_FreeNetRandom

Return:

- 0 成功
- BAD_FUNC_ARG configfile が null またはタイムアウトのどちらかが否定的です。
- RNG_FAILURE_E RNG の初期化に失敗しました。

B.9.2.2 function wc_FreeNetRandom

```
int wc_FreeNetRandom(
    void
)
```

無料の Global WhiteWood Netrandom コンテキスト。

See: wc_InitNetRandom

Return:

- 0 成功
- BAD_MUTEX_E Wnr_Mutex でミューテックスをロックするエラー *Example*

```
int ret = wc_FreeNetRandom();
if(ret != 0)
{
    // Handle the error
}
```

B.9.2.3 function wc_InitRng

```
int wc_InitRng(
    WC_RNG *
)
```

RNG のシード（OS から）とキー暗号を取得します。割り当てられた RNG-> DRBG（決定論的ランダムビットジェネレータ）が割り当てられます（WC_FREERNG で割り当てられている必要があります）。これはブロッキング操作です。

See:

- wc_InitRngCavium
- wc_RNG_GenerateBlock
- wc_RNG_GenerateByte
- wc_FreeRng
- wc_RNG_HealthTest

Return:

- 0 成功しています。
- MEMORY_E XMalloc に失敗しました
- WINCRYPT_E WC_GENERATSEED：コンテキストの取得に失敗しました
- CRYPTGEN_E WC_GENERATSEED：ランダムになりました
- BAD_FUNC_ARG WC_RNG_GenerateBlock 入力は NULL または SZ が MAX_REQUEST_LEN を超えています
- DRBG_CONT_FIPS_E wc_rng_generateblock：hash_gen は drbg_cont_failure を返しました
- RNG_FAILURE_E wc_rng_generateBlock：デフォルトエラーです。RNG のステータスはもともと OK ではなく、drbg_failed に設定されています *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;
}
```

B.9.2.4 function wc_RNG_GenerateBlock

```
int wc_RNG_GenerateBlock(
    WC_RNG * rng,
    byte * b,
    word32 sz
)
```

疑似ランダムデータの SZ バイトを出力にコピーします。必要に応じて RNG（ブロッキング）します。

Parameters:

- **rng** 乱数発生器は WC_INITRNG で初期化された
- **output** ブロックがコピーされるバッファ *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!
}
```

See:

- wc_InitRngCavium, wc_InitRng
- wc_RNG_GenerateByte
- wc_FreeRng
- wc_RNG_HealthTest

Return:

- 0 成功した
- BAD_FUNC_ARG 入力は NULL または SZ が MAX_REQUEST_LEN を超えています
- DRBG_CONT_FIPS_E hash_gen は drbg_cont_failure を返しました
- RNG_FAILURE_E デフォルトのエラー RNG のステータスはもともと OK ではなく、drbg_failed に設定されています

B.9.2.5 function wc_rng_new

```
WC_RNG * wc_rng_new(
    byte * nonce,
    word32 nonceSz,
    void * heap
)
```

新しい WC_RNG 構造を作成します。

Parameters:

- **heap** ヒープ識別子へのポインタ

- **nonce** nonce を含むバッファへのポインタ *Example*

```
RNG  rng;
byte nonce[] = { initialize nonce };
word32 nonceSz = sizeof(nonce);

wc_rng_new(&nonce, nonceSz, &heap);
```

- **rng** 乱数発生器は WC_INITRNG で初期化された *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!
}
```

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 成功の構造
- NULL 誤りに
- 0 成功した
- BAD_FUNC_ARG 入力は NULL または SZ が MAX_REQUEST_LEN を超えています
- DRBG_CONT_FIPS_E hash_gen は drbg_cont_failure を返しました
- RNG_FAILURE_E デフォルトのエラー RNG のステータスはもともと OK ではなく、drbg_failed に設定されています

wc_rng_generateBlock を呼び出して、疑似ランダムデータのバイトを b にコピーします。必要に応じて RNG が再販されます。

B.9.2.6 function wc_FreeRng

```
int wc_FreeRng(
    WC_RNG *
)
```

RNG が DRGB を安全に解放するために必要なときに呼び出されるべきです。ゼロと Xfrees RNG-DRBG。

See:

- wc_InitRngCavium

- wc_InitRng
- wc_RNG_GenerateBlock
- wc_RNG_GenerateByte,
- wc_RNG_HealthTest

Return:

- 0 成功した
- BAD_FUNC_ARG RNG または RNG-> DRGB NULL
- RNG_FAILURE_E DRBG の割り当て解除に失敗しました *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!
}
```

B.9.2.7 function wc_rng_free

```
WC_RNG * wc_rng_free(
    WC_RNG * rng
)
```

RNG を安全に自由に解放するために RNG が不要になったときに呼び出されるべきです。*Example*

See:

- wc_InitRng
- wc_rng_new
- wc_FreeRng
- wc_RNG_HealthTest

```
RNG  rng;
byte nonce[] = { initialize nonce };
word32 nonceSz = sizeof(nonce);

rng = wc_rng_new(&nonce, nonceSz, &heap);

// use rng

wc_rng_free(&rng);
```

B.9.2.8 function wc_RNG_HealthTest

```
int wc_RNG_HealthTest(
    int reseed,
    const byte * entropyA,
    word32 entropyASz,
    const byte * entropyB,
    word32 entropyBSz,
    byte * output,
    word32 outputSz
)
```

DRBG の機能を作成しテストします。

Parameters:

- **int** RESEED：設定されている場合は、Reseed 機能をテストします
- **entropyA** DRGB をインスタンス化するエントロピー
- **entropyASz** バイト数のエントロピヤのサイズ
- **entropyB** Reseed Set を設定した場合、DRBG は Entropyb でリサイードされます
- **entropyBSz** バイト単位の Entropyb のサイズ
- **output** SEADRANDOM が設定されている場合は、Entropyb に播種されたランダムなデータに初期化され、それ以外の場合は Entropya *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
```

See:

- wc_InitRngCavium
- wc_InitRng
- wc_RNG_GenerateBlock
- wc_RNG_GenerateByte
- wc_FreeRng

Return:

- 0 成功した
- BAD_FUNC_ARG ELTOPYA と出力は NULL にしないでください。Reseed Set Entropyb が NULL でなければならない場合
- -1 テスト失敗

B.10 Signature API

B.10.1 Functions

	Name
int	**wc_SignatureGetSize**(enum wc_SignatureType sig_type, const void * key, word32 key_len) この関数は、結果のシグネチャの最大サイズを返します。
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) この関数は、データをハッシュし、結果のハッシュとキーを使用して署名を使用して署名を使用して署名を検証します。
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) この関数は、キーを使用してデータから署名を生成します。まずデータのハッシュを作成し、キーを使用してハッシュに署名します。

B.10.2 Functions Documentation

B.10.2.1 function wc_SignatureGetSize

```
int wc_SignatureGetSize(
    enum wc_SignatureType sig_type,
    const void * key,
    word32 key_len
)
```

この関数は、結果のシグネチャの最大サイズを返します。

Parameters:

- **sig_type** wc_signature_type_ecc または wc_signature_type_rsa などの署名型列挙型値。
- **key** ECC_KEY や RSAKEY などのキー構造へのポインタ。*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
}
```

See:

- wc_HashGetDigestSize
- wc_SignatureGenerate
- wc_SignatureVerify

Return: Returns sig_type_e sig_type がサポートされていない場合 sig_type が無効な場合は bad_func_arg を返します。正の戻り値は、署名の最大サイズを示します。

B.10.2.2 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
)
```

この関数は、データをハッシュし、結果のハッシュとキーを使用して署名を使用して署名を使用して署名を検証します。

Parameters:

- **hash_type** “wc_hash_type_sha256” などの “enum wc_hashtype” からのハッシュ型。
- **sig_type** wc_signature_type_ecc または wc_signature_type_rsa などの署名型列挙型値。
- **data** ハッシュへのデータを含むバッファへのポインタ。
- **data_len** データバッファの長さ。
- **sig** 署名を出力するためのバッファへのポインタ。
- **sig_len** シグネチャ出力バッファの長さ。
- **key** ECC_KEY や RSAKEY などのキー構造へのポインタ。*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);
```

See:

- wc_SignatureGetSize
- wc_SignatureGenerate

Return:

- 0 成功
- SIG_TYPE_E -231、署名タイプが有効/利用可能です
- BAD_FUNC_ARG -173、関数の不良引数が提供されています
- BUFFER_E -132、出力バッファが小さすぎたり入力が大きすぎたりします。

B.10.2.3 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
)
```

この関数は、キーを使用してデータから署名を生成します。まずデータのハッシュを作成し、キーを使用してハッシュに署名します。

Parameters:

- **hash_type** “wc_hash_type_sha256” などの “enum wc_hashtype” からのハッシュ型。
- **sig_type** wc_signature_type_ecc または wc_signature_type_rsa などの署名型列挙型値。
- **data** ハッシュへのデータを含むバッファへのポインタ。
- **data_len** データバッファの長さ。
- **sig** 署名を出力するためのバッファへのポインタ。
- **sig_len** シグネチャ出力バッファの長さ。
- **key** ECC_KEY や RSAKEY などのキー構造へのポインタ。
- **key_len** キー構造のサイズ *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);
```

See:

- wc_SignatureGetSize
- wc_SignatureVerify

Return:

- 0 成功
- SIG_TYPE_E -231、署名タイプが有効/利用可能です
- BAD_FUNC_ARG -173、関数の不良引数が提供されています
- BUFFER_E -132、出力バッファが小さすぎたり入力が大きすぎたりします。

B.11 wolfCrypt Init and Cleanup

B.11.1 Functions

	Name
int	**wc_HashGetOID**(enum wc_HashType hash_type) この関数は提供された wc_hashtype の OID を返します。
int	**wc_HashGetDigestSize**(enum wc_HashType hash_type) この関数は、hash_type のダイジェスト（出力）のサイズを返します。返品サイズは、WC_HASH に提供される出力バッファが十分に大きいことを確認するために使用されます。
int	**wc_Hash**(enum wc_HashType hash_type, const byte * data, word32 data_len, byte * hash, word32 hash_len) この関数は、提供されたデータバッファ上にハッシュを実行し、提供されたハッシュバッファにそれを返します。
int	**wolfCrypt_Init**(void)WolfCrypt によって使用されるリソースを初期化するために使用されます。
int	**wolfCrypt_Cleanup**(void)WolfCrypt によって使用されるリソースをクリーンアップするために使用されます。

B.11.2 Functions Documentation

B.11.2.1 function wc_HashGetOID

```
int wc_HashGetOID(
    enum wc_HashType hash_type
)
```

この関数は提供された wc_hashtype の OID を返します。

See:

- wc_HashGetDigestSize
- wc_Hash

Return:

- OID 戻り値 0 を超えてください
- HASH_TYPE_E ハッシュ型はサポートされていません。
- BAD_FUNC_ARG 提供された引数の 1 つが正しくありません。*Example*

```
enum wc_HashType hash_type = WC_HASH_TYPE_SHA256;
int oid = wc_HashGetOID(hash_type);
if (oid > 0) {
    // Success
}
```

B.11.2.2 function wc_HashGetDigestSize

```
int wc_HashGetDigestSize(
    enum wc_HashType hash_type
)
```

この関数は、hash_type のダイジェスト（出力）のサイズを返します。返品サイズは、WC_HASH に提供される出力バッファが十分に大きいことを確認するために使用されます。

See: wc_Hash

Return:

- Success 正の戻り値は、ハッシュのダイジェストサイズを示します。
- Error hash_type がサポートされていない場合は hash_type_e を返します。
- Failure 無効な hash_type が使用された場合、bad_func_arg を返します。*Example*

```
int hash_len = wc_HashGetDigestSize(hash_type);
if (hash_len <= 0) {
WOLFSSL_MSG("Invalid hash type/len");
return BAD_FUNC_ARG;
}
```

B.11.2.3 function wc_Hash

```
int wc_Hash(
    enum wc_HashType hash_type,
    const byte * data,
    word32 data_len,
    byte * hash,
    word32 hash_len
)
```

この関数は、提供されたデータバッファ上にハッシュを実行し、提供されたハッシュバッファにそれを返します。

Parameters:

- **hash_type** “wc_hash_type_sha256” などの “enum wc_hashtype” からのハッシュ型。
- **data** ハッシュへのデータを含むバッファへのポインタ。
- **data_len** データバッファの長さ。
- **hash** 最後のハッシュを出力するために使用されるバッファへのポインタ。*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
    }
}
```

See: wc_HashGetDigestSize

Return: 0 そうでなければ、それ以外の誤り（bad_func_arg や buffer_e など）。

B.11.2.4 function wolfCrypt_Init

```
int wolfCrypt_Init(
    void
)
```

WolfCrypt によって使用されるリソースを初期化するために使用されます。

See: wolfCrypt_Cleanup

Return:

- 0 成功すると。
- <0 init リソースが失敗すると。*Example*

```
...
if (wolfCrypt_Init() != 0) {
    WOLFSSL_MSG("Error with wolfCrypt_Init call");
}
```

B.11.2.5 function wolfCrypt_Cleanup

```
int wolfCrypt_Cleanup(
    void
)
```

WolfCrypt によって使用されるリソースをクリーンアップするために使用されます。

See: wolfCrypt_Init

Return:

- 0 成功すると。
- <0 リソースのクリーンアップが失敗したとき。*Example*

```
...
if (wolfCrypt_Cleanup() != 0) {
    WOLFSSL_MSG("Error with wolfCrypt_Cleanup call");
}
```

B.12 Algorithms - 3DES

B.12.1 Functions

	Name
int	**wc_Des_SetKey**(Des * des, const byte * key, const byte * iv, int dir) この関数は、引数として与えられた DES 構造体のキーと初期化ベクトル（IV）を設定します。また、これらがまだ初期化されていない場合は、暗号化と復号化に必要なバッファーのスペースを初期化して割り当てます。注：IV が指定されていない場合（i.e.iv == null）初期化ベクトルは、デフォルトの IV 0 になります。
void	**wc_Des_SetIV**(Des * des, const byte * iv) この関数は、引数として与えられた DES 構造体の初期化ベクトル（IV）を設定します。NULL IV を渡したら、初期化ベクトルを 0 に設定します。
int	**wc_Des_CbcEncrypt**(Des * des, byte * out, const byte * in, word32 sz) この関数は入力メッセージを暗号化し、結果を出力バッファーに格納します。暗号ブロックチェーンチェーン（CBC）モードで DES 暗号化を使用します。

	Name
int	**wc_Des_CbcDecrypt**(Des * des, byte * out, const byte * in, word32 sz) この関数は入力暗号文を復号化し、結果の平文を出力バッファーに出力します。暗号ブロックチェーンチェーン（CBC）モードで DES 暗号化を使用します。
int	**wc_Des_EcbEncrypt**(Des * des, byte * out, const byte * in, word32 sz) この関数は入力メッセージを暗号化し、結果を出力バッファーに格納します。電子コードブック（ECB）モードで DES 暗号化を使用します。
int	**wc_Des3_EcbEncrypt**(Des3 * des, byte * out, const byte * in, word32 sz) この関数は入力メッセージを暗号化し、結果を出力バッファーに格納します。電子コードブック（ECB）モードで DES3 暗号化を使用します。警告：ほぼすべてのユースケースで ECB モードは安全性が低いと考えられています。可能な限り ECB API を直接使用しないでください。
int	**wc_Des3_SetKey**(Des3 * des, const byte * key, const byte * iv, int dir) この関数は、引数として与えられた DES3 構造のキーと初期化ベクトル（IV）を設定します。また、これらがまだ初期化されていない場合は、暗号化と復号化に必要なバッファーのスペースを初期化して割り当てます。注：IV が指定されていない場合（i.e.iv == null）初期化ベクトルは、デフォルトの IV 0 になります。
int	**wc_Des3_SetIV**(Des3 * des, const byte * iv) この関数は、引数として与えられた DES3 構造の初期化ベクトル（IV）を設定します。NULL IV を渡したら、初期化ベクトルを 0 に設定します。
int	**wc_Des3_CbcEncrypt**(Des3 * des, byte * out, const byte * in, word32 sz) この関数は入力メッセージを暗号化し、結果を出力バッファーに格納します。暗号ブロックチェーン（CBC）モードでトリプル DES（3DES）暗号化を使用します。
int	**wc_Des3_CbcDecrypt**(Des3 * des, byte * out, const byte * in, word32 sz) この関数は入力暗号文を復号化し、結果の平文を出力バッファーに出力します。暗号ブロックチェーン（CBC）モードでトリプル DES（3DES）暗号化を使用します。
int	**wc_Des_CbcDecryptWithKey**(byte * out, const byte * in, word32 sz, const byte * key, const byte * iv) この関数は入力暗号文を復号化し、結果の平文を出力バッファーに出力します。暗号ブロックチェーンチェーン（CBC）モードで DES 暗号化を使用します。この関数は、wc_des_cbcdecrypt の代わりに、ユーザーが DES 構造体を直接インスタンス化せずにメッセージを復号化できるようにします。

	Name
int	**wc_Des_CbcEncryptWithKey**(byte * out, const byte * in, word32 sz, const byte * key, const byte * iv) この関数は入力平文を暗号化し、結果の暗号文を出力バッファーに出力します。暗号ブロックチェーンチェーン（CBC）モードで DES 暗号化を使用します。この関数は、WC_DES_CBCENCRYPT の代わりになり、ユーザーが DES 構造を直接インスタンス化せずにメッセージを暗号化できます。
int	**wc_Des3_CbcEncryptWithKey**(byte * out, const byte * in, word32 sz, const byte * key, const byte * iv) この関数は入力平文を暗号化し、結果の暗号文を出力バッファーに出力します。暗号ブロックチェーン（CBC）モードでトリプル DES（3DES）暗号化を使用します。この関数は、WC_DES3_CBCENCRYPT の代わりになり、ユーザーが DES3 構造を直接インスタンス化せずにメッセージを暗号化できます。
int	**wc_Des3_CbcDecryptWithKey**(byte * out, const byte * in, word32 sz, const byte * key, const byte * iv) この関数は入力暗号文を復号化し、結果の平文を出力バッファーに出力します。暗号ブロックチェーン（CBC）モードでトリプル DES（3DES）暗号化を使用します。この関数は、wc_des3_cbcdecrypt の代わりに、ユーザーが DES3 構造を直接インスタンス化せずにメッセージを復号化できるようにします。

B.12.2 Functions Documentation

B.12.2.1 function wc_Des_SetKey

```
int wc_Des_SetKey(
    Des * des,
    const byte * key,
    const byte * iv,
    int dir
)
```

この関数は、引数として与えられた DES 構造体のキーと初期化ベクトル（IV）を設定します。また、これらがまだ初期化されていない場合は、暗号化と復号化に必要なバッファーのスペースを初期化して割り当てます。注：IV が指定されていない場合（i.e.iv == null）初期化ベクトルは、デフォルトの IV 0 になります。

Parameters:

- **des** 初期化する DES 構造へのポインタ
- **key** DES 構造を初期化するための 8 バイトのキーを含むバッファへのポインタ
- **iv** DES 構造を初期化するための 8 バイト IV を含むバッファへのポインタ。これが提供されていない場合、IV はデフォルトで 0 になります *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
}
```

See:

- wc_Des_SetIV
- wc_Des3_SetKey

Return: 0 DES 構造体のキーと初期化ベクトルを正常に設定する

3

B.12.2.2 function wc_Des_SetIV

```
void wc_Des_SetIV(
    Des * des,
    const byte * iv
)
```

この関数は、引数として与えられた DES 構造体の初期化ベクトル（IV）を設定します。NULL IV を渡したら、初期化ベクトルを 0 に設定します。

Parameters:

- **des** IV を設定するための DES 構造へのポインタ *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);
}
```

See: wc_Des_SetKey

Return: none いいえ返します。

3

B.12.2.3 function wc_Des_CbcEncrypt

```
int wc_Des_CbcEncrypt(
    Des * des,
    byte * out,
    const byte * in,
    word32 sz
)
```

この関数は入力メッセージを暗号化し、結果を出力バッファーに格納します。暗号ブロックチェーンチェーン（CBC）モードで DES 暗号化を使用します。

Parameters:

- **des** 暗号化に使用する DES 構造へのポインタ
- **out** 暗号化された暗号文を保存するバッファへのポインタ
- **in** 暗号化するメッセージを含む入力バッファへのポインタ *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
}
```

See:

- wc_Des_SetKey
- wc_Des_CbcDecrypt

Return: 0 与えられた入力メッセージの暗号化に成功したときに返されます

3

B.12.2.4 function wc_Des_CbcDecrypt

```
int wc_Des_CbcDecrypt(
    Des * des,
    byte * out,
    const byte * in,
    word32 sz
)
```

この関数は入力暗号文を復号化し、結果の平文を出力バッファーに出力します。暗号ブロックチェーンチェーン（CBC）モードで DES 暗号化を使用します。

Parameters:

- **des** 復号化に使用する DES 構造へのポインタ
- **out** 復号化された平文を保存するバッファへのポインタ
- **in** 暗号化された暗号文を含む入力バッファへのポインタ *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
}
```

See:

- wc_Des_SetKey
- wc_Des_CbcEncrypt

Return: 0 与えられた暗号文を正常に復号化したときに返されました

3

B.12.2.5 function wc_Des_EcbEncrypt

```
int wc_Des_EcbEncrypt(
    Des * des,
    byte * out,
    const byte * in,
    word32 sz
)
```

この関数は入力メッセージを暗号化し、結果を出力バッファーに格納します。電子コードブック（ECB）モードで DES 暗号化を使用します。

Parameters:

- **des** 暗号化に使用する DES 構造へのポインタ
- **out** 暗号化されたメッセージを保存するバッファへのポインタ
- **in** 暗号化する平文を含む入力バッファへのポインタ *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
}
```

See: wc_Des_SetKe

Return: 0: 与えられた平文を正常に暗号化すると返されます。

3

B.12.2.6 function wc_Des3_EcbEncrypt

```
int wc_Des3_EcbEncrypt(
    Des3 * des,
    byte * out,
    const byte * in,
    word32 sz
)
```

この関数は入力メッセージを暗号化し、結果を出力バッファーに格納します。電子コードブック（ECB）モードで DES3 暗号化を使用します。警告：ほぼすべてのユースケースで ECB モードは安全性が低いと考えられています。可能な限り ECB API を直接使用しないでください。

Parameters:

- **des3** 暗号化に使用する DES3 構造へのポインタ
- **out** 暗号化されたメッセージを保存するバッファへのポインタ
- **in** 暗号化する平文を含む入力バッファへのポインタ *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
}
```

See: wc_Des3_SetKey

Return: 0 与えられた平文を正常に暗号化すると返されます

3

B.12.2.7 function wc_Des3_SetKey

```
int wc_Des3_SetKey(
    Des3 * des,
    const byte * key,
    const byte * iv,
    int dir
)
```

この関数は、引数として与えられた DES3 構造のキーと初期化ベクトル（IV）を設定します。また、これらがまだ初期化されていない場合は、暗号化と復号化に必要なバッファーのスペースを初期化して割り当てます。注：IV が指定されていない場合（i.e.iv == null）初期化ベクトルは、デフォルトの IV 0 になります。

Parameters:

- **des3** 初期化する DES3 構造へのポインタ
- **key** DES3 構造を初期化する 24 バイトのキーを含むバッファへのポインタ
- **iv** DES3 構造を初期化するための 8 バイト IV を含むバッファへのポインタ。これが提供されていない場合、IV はデフォルトで 0 になります *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
}
```

See:

- wc_Des3_SetIV
- wc_Des3_CbcEncrypt
- wc_Des3_CbcDecrypt

Return: 0 DES 構造体のキーと初期化ベクトルを正常に設定する

3

B.12.2.8 function wc_Des3_SetIV

```
int wc_Des3_SetIV(
    Des3 * des,
    const byte * iv
)
```

この関数は、引数として与えられた DES3 構造の初期化ベクトル（IV）を設定します。NULL IV を渡したら、初期化ベクトルを 0 に設定します。

Parameters:

- **des** IV を設定するための DES3 構造へのポインタ *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);
}
```

See: wc_Des3_SetKey

Return: none いいえ返します。

3

B.12.2.9 function wc_Des3_CbcEncrypt

```
int wc_Des3_CbcEncrypt(
    Des3 * des,
    byte * out,
    const byte * in,
    word32 sz
)
```

この関数は入力メッセージを暗号化し、結果を出力バッファーに格納します。暗号ブロックチェーン（CBC）モードでトリプル DES（3DES）暗号化を使用します。

Parameters:

- **des** 暗号化に使用する DES3 構造へのポインタ
- **out** 暗号化された暗号文を保存するバッファへのポインタ
- **in** 暗号化するメッセージを含む入力バッファへのポインタ *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
}
```

See:

- wc_Des3_SetKey
- wc_Des3_CbcDecrypt

Return: 0 与えられた入力メッセージの暗号化に成功したときに返されます

3

B.12.2.10 function wc_Des3_CbcDecrypt

```
int wc_Des3_CbcDecrypt(
    Des3 * des,
    byte * out,
    const byte * in,
    word32 sz
)
```

この関数は入力暗号文を復号化し、結果の平文を出力バッファーに出力します。暗号ブロックチェーン（CBC）モードでトリプル DES（3DES）暗号化を使用します。

Parameters:

- **des** 復号化に使用する DES3 構造へのポインタ
- **out** 復号化された平文を保存するバッファへのポインタ
- **in** 暗号化された暗号文を含む入力バッファへのポインタ *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
}
```

See:

- wc_Des3_SetKey
- wc_Des3_CbcEncrypt

Return: 0 与えられた暗号文を正常に復号化したときに返されました

3

B.12.2.11 function wc_Des_CbcDecryptWithKey

```
int wc_Des_CbcDecryptWithKey(
    byte * out,
    const byte * in,
    word32 sz,
    const byte * key,
    const byte * iv
)
```

この関数は入力暗号文を復号化し、結果の平文を出力バッファーに出力します。暗号ブロックチェーンチェーン（CBC）モードで DES 暗号化を使用します。この関数は、wc_des_cbcdecrypt の代わりに、ユーザーが DES 構造体を直接インスタンス化せずにメッセージを復号化できるようにします。

Parameters:

- **out** 復号化された平文を保存するバッファへのポインタ
- **in** 暗号化された暗号文を含む入力バッファへのポインタ
- **sz** 復号化する暗号文の長さ
- **key** 復号化に使用する 8 バイトのキーを含むバッファへのポインタ *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
}
```

See: wc_Des_CbcDecrypt

Return:

- 0 与えられた暗号文を正常に復号化したときに返されました
- MEMORY_E DES 構造体の割り当てスペースが割り当てられている場合に返された

3

B.12.2.12 function wc_Des_CbcEncryptWithKey

```
int wc_Des_CbcEncryptWithKey(
    byte * out,
    const byte * in,
    word32 sz,
    const byte * key,
    const byte * iv
)
```

この関数は入力平文を暗号化し、結果の暗号文を出力バッファーに出力します。暗号ブロックチェーンチェーン（CBC）モードで DES 暗号化を使用します。この関数は、WC_DES_CBCENCRYPT の代わりになり、ユーザーが DES 構造を直接インスタンス化せずにメッセージを暗号化できます。

Parameters:

- **out** 最終暗号化データ
- **in** 暗号化されるデータは、DES ブロックサイズに埋められなければなりません。
- **sz** 入力バッファのサイズ
- **key** 暗号化に使用するキーへのポインタ。*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
}
```

See:

- wc_Des_CbcDecryptWithKey
- wc_Des_CbcEncrypt

Return:

- 0 データの暗号化に成功した後に返されます。
- MEMORY_E DES 構造体にメモリを割り当てるエラーがある場合は返されます。
- <0 暗号化中に任意のエラーに戻ります。

3

B.12.2.13 function wc_Des3_CbcEncryptWithKey

```
int wc_Des3_CbcEncryptWithKey(
    byte * out,
    const byte * in,
    word32 sz,
    const byte * key,
    const byte * iv
)
```

この関数は入力平文を暗号化し、結果の暗号文を出力バッファーに出力します。暗号ブロックチェーン（CBC）モードでトリプル DES（3DES）暗号化を使用します。この関数は、WC_DES3_CBCENCRYPT の代わりになり、ユーザーが DES3 構造を直接インスタンス化せずにメッセージを暗号化できます。

Parameters:

- **out** 最終暗号化データ
- **in** 暗号化されるデータは、DES ブロックサイズに埋められなければなりません。

- **sz** 入力バッファのサイズ
- **key** 暗号化に使用するキーへのポインタ。*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
}
```

See:

- wc_Des3_CbcDecryptWithKey
- wc_Des_CbcEncryptWithKey
- wc_Des_CbcDecryptWithKey

Return:

- 0 データの暗号化に成功した後に返されます。
- MEMORY_E DES 構造体にメモリを割り当てるエラーがある場合は返されます。
- <0 暗号化中に任意のエラーに戻ります。

3

B.12.2.14 function wc_Des3_CbcDecryptWithKey

```
int wc_Des3_CbcDecryptWithKey(
    byte * out,
    const byte * in,
    word32 sz,
    const byte * key,
    const byte * iv
)
```

この関数は入力暗号文を復号化し、結果の平文を出力バッファーに出力します。暗号ブロックチェーン（CBC）モードでトリプル DES（3DES）暗号化を使用します。この関数は、wc_des3_cbcdecrypt の代わりに、ユーザーが DES3 構造を直接インスタンス化せずにメッセージを復号化できるようにします。

Parameters:

- **out** 復号化された平文を保存するバッファへのポインタ
- **in** 暗号化された暗号文を含む入力バッファへのポインタ
- **sz** 復号化する暗号文の長さ
- **key** 復号化に使用する 24 バイトのキーを含むバッファへのポインタ *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
}
```

See: wc_Des3_CbcDecrypt

Return:

- 0 与えられた暗号文を正常に復号化したときに返されました
- MEMORY_E DES 構造体の割り当てスペースが割り当てられている場合に返された

3

B.13 Algorithms - AES

B.13.1 Functions

	Name
int	**wc_AesSetKey**(Aes * aes, const byte * key, word32 len, const byte * iv, int dir) この関数は、鍵を設定して初期化ベクトルを設定することで Aes 構造体を初期化します。
int	**wc_AesSetIV**(Aes * aes, const byte * iv) この関数は、指定された Aes 構造体の初期化ベクトルを設定します。Aes 構造体は、この関数を呼び出す前に初期化されていることが必要です。
int	**wc_AesCbcEncrypt**(Aes * aes, byte * out, const byte * in, word32 sz) 入力バッファの平文メッセージを暗号化し、AES で Cipher Block Chaining を使用して出力バッファに出力します。この関数呼び出しには、メッセージの暗号化前に wc_AesSetKey を呼び出して AES オブジェクトが初期化されている必要があります。この関数は、入力メッセージが AES ブロック長であると仮定し、入力された長さがブロック長の倍数になることを想定しているため、ビルド構成で WOLFSSL_AES_CBC_LENGTH_CHECKS が定義されている場合は任意選択でチェックおよび適用されます。ブロック多入力を保証するために、PKCS#7 スタイルのパディングを事前に追加する必要があります。これは自動的にパディングを追加する OpenSSL AES-CBC メソッドとは異なります。WOLFSSL と対応する OpenSSL 関数を相互運用するには、OpenSSL コマンドライン関数で-nopad オプションを指定して、wolfSSL_AesCbcEncrypt メソッドのように動作し、暗号化中に追加のパディングを追加しません。

	Name
int	**wc_AesCbcDecrypt**(Aes * aes, byte * out, const byte * in, word32 sz) 入力バッファからの暗号メッセージを復号し、AES で Cipher Block Chaining を使用して出力バッファに出力します。この関数呼び出しには、メッセージの暗号化前に wc_AesSetKey を呼び出して AES オブジェクトが初期化されている必要があります。この関数は、元のメッセージが AES ブロック長で整列していたと仮定し、入力された長さがブロック長の倍数になると予想しています。これは OpenSSL AES-CBC メソッドとは異なります。これは、PKCS#7 パディングを自動的に追加するため、ブロックマルチ入力を必要としません。wolfSSL 機能と同等の OpenSSL 関数を相互運用するには、OpenSSL コマンドライン関数で-nopad オプションを指定し、wolfSSL_ AesCbcEncrypt メソッドのように動作し、復号中にエラーを発生させません。
int	**wc_AesCtrEncrypt**(Aes * aes, byte * out, const byte * in, word32 sz) 入力バッファからメッセージを暗号化/復号し、AES CTR モードを使用して出力バッファーに出力します。この関数は、wolfSSL_Aes_Counter がコンパイル時に有効になっている場合にのみ有効になります。この機能を呼び出す前に、Aes 構造体を wc_AesSetKey で初期化する必要があります。この関数は復号と暗号化の両方に使用されます。_ 注: 暗号化と復号のための同じ API を使用することについて。ユーザーは暗号化/復号のための Aes 構造体を区別する必要があります。
int	**wc_AesEncryptDirect**(Aes * aes, byte * out, const byte * in) この関数は、入力ブロック in で与えられた単一の平文データブロックを暗号化して単一の出力ブロック out に出力します。その際に、Aes 構造体で提供されたの鍵を使用します。鍵はこの機能を呼び出す前に wc_AesSetKey で初期化されている必要があります。wc_AesSetKey への入力 iv には NULL を指定して呼び出してください。これは、Configure Option WolfSSL_AES_DIRECT が有効になっている場合にのみ有効になります。__ warning：ほぼすべてのユースケースで ECB モードは安全性が低いと考えられています。可能な限り ECB API を直接使用しないでください。

	Name
int	**wc_AesDecryptDirect**(Aes * aes, byte * out, const byte * in) この関数は、入力ブロック in で与えられた単一の暗号データブロックを復号して単一の出力ブロック out に出力します。提供された Aes 構造体の鍵を使用します。Aes 構造体は、この機能を呼び出す前に wc_AesSetKey で初期化される必要があります。wc_AesSetKey は、iv が NULL で呼び出される必要があります。これは、Configure Option WOLFSSL_AES_DIRECT が有効になっている場合にのみ有効になります。__ warning：ほぼすべてのユースケースで ECB モードは安全性が低いと考えられています。可能な限り ECB API を直接使用しないでください。
int	**wc_AesSetKeyDirect**(Aes * aes, const byte * key, word32 len, const byte * iv, int dir) この関数は、CTR モードの AES 鍵を AES で設定するために使用されます。指定された鍵、iv（初期化ベクトル）、および暗号化 dir（方向）で AES オブジェクトを初期化します。構成オプション WOLFSSL_AES_DIRECT が有効になっている場合にのみ有効になります。wc_AesEncryptDirect と wc_AesDecryptDirect を呼び出す際の Aes 構造体の初期化にはこの関数を使う必要があります。現在 wc_AesSetKeyDirect は内部的に wc_AesSetKey を使用します。__ warning：ほぼすべてのユースケースで ECB モードは安全性が低いと考えられています。可能な限り ECB API を直接使用しないでください
int	**wc_AesGcmSetKey**(Aes * aes, const byte * key, word32 len) この機能は、AES GCM（Galois/Counter Mode）の鍵を設定するために使用されます。与えられた key で Aes 構造体を初期化します。コンパイル時に Configure オプション HAVE_AESGCM が有効になっている場合にのみ有効になります。
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) この関数は、バッファ in に格納されている平文メッセージを暗号化し結果を出力バッファ out に出力します。暗号化する呼び出しごとに新しい iv(初期化ベクトル) が必要です。また、入力認証ベクトル、authIn、authTag への入力認証ベクトルをエンコードします。

	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) この関数は、バッファ in で与えられた入力暗号テキストを復号し、結果を出力バッファ out に格納します。また、指定された認証タグ、authTag に対して、入力認証ベクトル、authIn をチェックします。
int	**wc_GmacSetKey**(Gmac * gmac, const byte * key, word32 len) この関数は、GAROIS メッセージ認証に使用される Gmac 構造体の鍵を初期化して設定します。
int	**wc_GmacUpdate**(Gmac * gmac, const byte * iv, word32 ivSz, const byte * authIn, word32 authInSz, byte * authTag, word32 authTagSz) この関数は authIn Input の GMAC ハッシュを生成し、結果を authTag バッファに格納します。wc_GmacUpdate を実行した後、生成された authTag を既知の認証タグに比較してメッセージの信頼性を検証する必要があります。
int	**wc_AesCcmSetKey**(Aes * aes, const byte * key, word32 keySz) この関数は、CCM を使用して AES オブジェクトの鍵を設定します（CBC-MAC のカウンタ）。Aes 構造体へのポインタを取り、引数で与えられた key で初期化します。
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) この関数は、CCM を使用して、入力メッセージ、IN、OUT、OUT、OUT を CCM（CBC-MAC のカウンタ）を暗号化します。その後、Authin Input から認証タグ、AuthtAg を計算して格納します。
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) この関数は、CCM を使用して、入力暗号テキストを、CCM（CBC-MAC のカウンタ）を使用して出力バッファーに復号します。その後、authIn 入力から authTag を計算します。認証タグが無効な場合は、出力バッファをゼロに設定し、AES_CCM_AUTH_E を返します。
int	**wc_AesXtsSetKey**(XtsAes * aes, const byte * key, word32 len, int dir, void * heap, int devId) この関数は、AES XTS モードを使用する暗号化または復号で使用する鍵の設定に使用します。完了したら、AES キーで wc_AesXtsFree を呼び出すことがユーザーになりました。

	Name
int	**wc_AesXtsEncryptSector**(XtsAes * aes, byte * out, const byte * in, word32 sz, word64 sector)wc_AesXtsEncrypt と同じ処理を行いますが、バイト配列の代わりに Tweak 値として word64 型を使用します。本関数で word64 をバイト配列に変換し、wc_AesXtsEncrypt を呼び出します。
int	**wc_AesXtsDecryptSector**(XtsAes * aes, byte * out, const byte * in, word32 sz, word64 sector)wc_AesXtsDecrypt と同じ処理を行いますが、バイト配列の代わりに Tweak 値として word64 タイプを使用します。本関数で word64 をバイト配列に変換するだけです。
int	**wc_AesXtsEncrypt**(XtsAes * aes, byte * out, const byte * in, word32 sz, const byte * i, word32 iSz)AES XTS モードで暗号化します。（XTS）XEX 暗号化と平文がブロック長の倍数でない場合の処理 (Ciphertext Stealing) を行います。
int	**wc_AesXtsDecrypt**(XtsAes * aes, byte * out, const byte * in, word32 sz, const byte * i, word32 iSz) 暗号化と同じプロセスですが、XtsAes 構造体は AES_Decryption タイプです。
int	**wc_AesXtsFree**(XtsAes * aes) この関数は XtsAes 構造体で使用されるすべてのリソースを解放します。
int	**wc_AesInit**(Aes * aes, void * heap, int devId)Aes 構造体を初期化します。ヒープヒントを設定し、ASYNC ハードウェアを使用する場合の ID も設定します。Aes 構造体の使用が終了した際に wc_AesFree を呼び出すのはユーザーに任されています。
int	**wc_AesFree**(Aes * aes)Aes 構造体に関連つけられたリソースを可能なら解放します。内部的にはノーオペレーションとなることもありますが、ベストプラクティスとしてどのケースでもこの関数を呼び出すことを推奨します。
int	**wc_AesCfbEncrypt**(Aes * aes, byte * out, const byte * in, word32 sz)AES CFB モードで暗号化を行います。
int	**wc_AesCfbDecrypt**(Aes * aes, byte * out, const byte * in, word32 sz)AES CFB モードで復号を行います。
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) この関数は、RFC 5297 に記載されているように SIV（合成初期化ベクトル）暗号化を実行します。

	Name
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) この機能は、RFC 5297 に記載されているように SIV（合成初期化ベクトル）復号を実行します
int	**wc_AesCbcDecryptWithKey**(byte * out, const byte * in, word32 inSz, const byte * key, word32 keySz, const byte * iv) 入力バッファーから暗号を復号化し、AES で Cipher Block Chaining を使用して出力バッファに出力バッファーに入れます。この関数は、AES 構造を初期化する必要はありません。代わりに、キーと IV（初期化ベクトル）を取り、これらを使用して AES オブジェクトを初期化してから暗号テキストを復号化します。

B.13.2 Functions Documentation

B.13.2.1 function wc_AesSetKey

```
int wc_AesSetKey(
    Aes * aes,
    const byte * key,
    word32 len,
    const byte * iv,
    int dir
)
```

この関数は、鍵を設定して初期化ベクトルを設定することで Aes 構造体を初期化します。

Parameters:

- **aes** 変更する Aes 構造体へのポインタ
- **key** 暗号化と復号のための 16,24、または 32 バイトの秘密鍵
- **len** 渡された鍵の長さ
- **iv** 鍵を初期化するために使用される初期化ベクトルへのポインタ *Example*

```
Aes enc;
int ret = 0;
byte key[] = { some 16, 24 or 32 byte key };
byte iv[]  = { some 16 byte iv };
if (ret = wc_AesSetKey(&enc, key, AES_BLOCK_SIZE, iv,
AES_ENCRYPTION) != 0) {
// failed to set aes key
}
```

See:

- wc_AesSetKeyDirect
- wc_AesSetIV

Return:

- 0 鍵と初期化ベクトルを正常に設定しました
- BAD_FUNC_ARG 鍵の長さが無効な場合に返されます。

B.13.2.2 function wc_AesSetIV

```
int wc_AesSetIV(
    Aes * aes,
    const byte * iv
)
```

この関数は、指定された Aes 構造体の初期化ベクトルを設定します。Aes 構造体は、この関数を呼び出す前に初期化されていることが必要です。

Parameters:

- **aes** 初期化ベクトルを設定する Aes 構造体へのポインタ *Example*

```
Aes enc;
// set enc key
byte iv[]  = { some 16 byte iv };
if (ret = wc_AesSetIV(&enc, iv) != 0) {
// failed to set aes iv
}
```

See:

- wc_AesSetKeyDirect
- wc_AesSetKey

Return:

- 0 初期化ベクトルを正常に設定します。
- BAD_FUNC_ARG Aes 構造体へのポインタが NULL の場合に返されます。

B.13.2.3 function wc_AesCbcEncrypt

```
int wc_AesCbcEncrypt(
    Aes * aes,
    byte * out,
    const byte * in,
    word32 sz
)
```

入力バッファの平文メッセージを暗号化し、AES で Cipher Block Chaining を使用して出力バッファに出力します。この関数呼び出しには、メッセージの暗号化前に wc_AesSetKey を呼び出して AES オブジェクトが初期化されている必要があります。この関数は、入力メッセージが AES ブロック長であると仮定し、入力された長さがブロック長の倍数になることを想定しているため、ビルド構成で WOLFSSL_AES_CBC_LENGTH_CHECKS が定義されている場合は任意選択でチェックおよび適用されます。ブロック多入力を保証するために、PKCS#7 スタイルのパディングを事前に追加する必要があります。これは自動的にパディングを追加する OpenSSL AES-CBC メソッドとは異なります。WOLFSSL と対応する OpenSSL 関数を相互運用するには、OpenSSL コマンドライン関数で-nopad オプションを指定して、wolfSSL_AesCbcEncrypt メソッドのように動作し、暗号化中に追加のパディングを追加しません。

Parameters:

- **aes** データの暗号化に使用される AES オブジェクトへのポインタ
- **out** 暗号化されたメッセージの暗号文を格納する出力バッファへのポインタ
- **in** 暗号化されるメッセージを含む入力バッファへのポインタ *Example*

```
Aes enc;
int ret = 0;
// initialize enc with 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
}
```

See:

- wc_AesSetKey
- wc_AesSetIV
- wc_AesCbcDecrypt

Return:

- 0 メッセージの暗号化に成功しました。
- BAD_ALIGN_E: ブロックアライメントエラー検出時に返される可能性があります
- BAD_LENGTH_E ライブラリーが WOLFSSL_AES_CBC_LENGTH_CHECKS で構築されている場合で、入力長が AES ブロック長の倍数でない場合に返されます。

B.13.2.4 function wc_AesCbcDecrypt

```
int wc_AesCbcDecrypt(
    Aes * aes,
    byte * out,
    const byte * in,
    word32 sz
)
```

入力バッファからの暗号メッセージを復号し、AES で Cipher Block Chaining を使用して出力バッファに出力します。この関数呼び出しには、メッセージの暗号化前に wc_AesSetKey を呼び出して AES オブジェクトが初期化されている必要があります。この関数は、元のメッセージが AES ブロック長で整列していたと仮定し、入力された長さがブロック長の倍数になると予想しています。これは OpenSSL AES-CBC メソッドとは異なります。これは、PKCS#7 パディングを自動的に追加するため、ブロックマルチ入力を必要としません。wolfSSL 機能と同等の OpenSSL 関数を相互運用するには、OpenSSL コマンドライン関数で-nopad オプションを指定し、wolfSSL_ AesCbcEncrypt メソッドのように動作し、復号中にエラーを発生させません。

Parameters:

- **aes** データを復号するために使用される AES オブジェクトへのポインタ。
- **out** 復号されたメッセージのプレーンテキストを保存する出力バッファへのポインタ。サイズは AES_BLOCK_SIZE の倍数でなければなりません。必要な場合はパディングは追加されます。
- **in** 復号する暗号テキストを含む入力バッファへのポインタ。サイズは AES_BLOCK_SIZE の倍数でなければなりません。パディングされている必要があります。
- **sz** 入力バッファのサイズ *Example*

```
Aes dec;
int ret = 0;
// initialize dec with 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
}
```

See:

- wc_AesSetKey
- wc_AesCbcEncrypt

Return:

- 0 メッセージを正常に復号しました
- BAD_ALIGN_E ブロックアライメントエラー検出時に返される可能性があります
- BAD_LENGTH_E ライブラリーが WOLFSSL_AES_CBC_LENGTH_CHECKS で構築されている場合で、入力長が AES ブロック長の倍数でない場合に返されます。

B.13.2.5 function wc_AesCtrEncrypt

```
int wc_AesCtrEncrypt(
    Aes * aes,
    byte * out,
    const byte * in,
    word32 sz
)
```

入力バッファからメッセージを暗号化/復号し、AES CTR モードを使用して出力バッファーに出力します。この関数は、wolfSSL_Aes_Counter がコンパイル時に有効になっている場合にのみ有効になります。この機能を呼び出す前に、Aes 構造体を wc_AesSetKey で初期化する必要があります。この関数は復号と暗号化の両方に使用されます。_ 注: 暗号化と復号のための同じ API を使用することについて。ユーザーは暗号化/復号のための Aes 構造体を区別する必要があります。

Parameters:

- **aes** データを復号するために使用される Aes 構造体へのポインタ
- **out** 暗号化されたメッセージの暗号化テキストを保存する出力バッファへのポインタ サイズは AES_BLOCK_SIZE の倍数でなければなりません。必要な場合はパディングは追加されます。
- **in** 暗号化されるプレーンテキストを含む入力バッファへのポインタ。サイズは AES_BLOCK_SIZE の倍数でなければなりません。パディングされている必要があります。
- **sz** 入力バッファのサイズ *Example*

```
Aes enc;
Aes dec;
// initialize enc and dec with 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
```

See: wc_AesSetKey

Return: int WolfSSL エラーまたは成功状況に対応する整数値

B.13.2.6 function wc_AesEncryptDirect

```
int wc_AesEncryptDirect(
    Aes * aes,
    byte * out,
    const byte * in
)
```

この関数は、入力ブロック in で与えられた単一の平文データブロックを暗号化して単一の出力ブロック out に出力します。その際に、Aes 構造体で提供されたの鍵を使用します。鍵はこの機能を呼び出す前に wc_AesSetKey で初期化されている必要があります。wc_AesSetKey への入力 iv には NULL を指定して呼び出してください。これは、Configure Option WolfSSL_AES_DIRECT が有効になっている場合にのみ有効になります。__ warning：ほぼすべてのユースケースで ECB モードは安全性が低いと考えられています。可能な限り ECB API を直接使用しないでください。

Parameters:

- **aes** データの暗号化に使用される Aes 構造体へのポインタ
- **out** 暗号化されたメッセージの暗号化テキストを保存する出力バッファへのポインタ *Example*

```
Aes enc;
// initialize enc with 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);
```

See:

- wc_AesDecryptDirect
- wc_AesSetKeyDirect

Return: int WolfSSL エラーまたは成功状況に対応する整数値

B.13.2.7 function wc_AesDecryptDirect

```
int wc_AesDecryptDirect(
    Aes * aes,
    byte * out,
    const byte * in
)
```

この関数は、入力ブロック in で与えられた単一の暗号データブロックを復号して単一の出力ブロック out に出力します。提供された Aes 構造体の鍵を使用します。Aes 構造体は、この機能を呼び出す前に wc_AesSetKey で初期化される必要があります。wc_AesSetKey は、iv が NULL で呼び出される必要があります。これは、Configure Option WOLFSSL_AES_DIRECT が有効になっている場合にのみ有効になります。__ warning：ほぼすべてのユースケースで ECB モードは安全性が低いと考えられています。可能な限り ECB API を直接使用しないでください。

Parameters:

- **aes** データの復号に使用される AES オブジェクトへのポインタ
- **out** 復号された平文テキストを格納する出力バッファへのポインタ *Example*

```
Aes dec;
// initialize enc with 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);
```

See:

- wc_AesEncryptDirect
- wc_AesSetKeyDirect

Return: int WolfSSL エラーまたは成功状況に対応する整数値

B.13.2.8 function wc_AesSetKeyDirect

```
int wc_AesSetKeyDirect(
    Aes * aes,
    const byte * key,
    word32 len,
    const byte * iv,
    int dir
)
```

この関数は、CTR モードの AES 鍵を AES で設定するために使用されます。指定された鍵、iv（初期化ベクトル）、および暗号化 dir（方向）で AES オブジェクトを初期化します。構成オプション WOLFSSL_AES_DIRECT が有効になっている場合にのみ有効になります。wc_AesEncryptDirect と wc_AesDecryptDirect を呼び出す際の Aes 構造体の初期化にはこの関数を使う必要があります。現在 wc_AesSetKeyDirect は内部的に wc_AesSetKey を使用します。__ warning：ほぼすべてのユースケースで ECB モードは安全性が低いと考えられています。可能な限り ECB API を直接使用しないでください

Parameters:

- **aes** データの暗号化に使用される AES オブジェクトへのポインタ
- **key** 暗号化と復号のための 16,24、または 32 バイトの秘密鍵
- **len** 渡された鍵の長さ
- **iv** 鍵を初期化するために使用される初期化ベクトル
- **dir** 暗号化の方向を指定します。wc_AesEncryptDirect に使用する際には AES_ENCRYPTION、wc_AesDecryptDirect には AES_DECRYPTION を指定します。(注意: wc_AesSetKeyDirect を Aes カウンターモードに使用する際には暗号化/復号によらず、AES_ENCRYPTION を指定してください。)

See:

- wc_AesEncryptDirect
- wc_AesDecryptDirect
- wc_AesSetKey

Return:

- 0 鍵の設定に成功しました。
- BAD_FUNC_ARG 与えられたキーが無効な長さの場合に返されます。

Example

```
Aes enc;
int ret = 0;
byte key[] = { some 16, 24, or 32 byte key };
byte iv[]  = { some 16 byte iv };
if (ret = wc_AesSetKeyDirect(&enc, key, sizeof(key), iv,
AES_ENCRYPTION) != 0) {
// failed to set aes key
}
```

B.13.2.9 function wc_AesGcmSetKey

```
int wc_AesGcmSetKey(
    Aes * aes,
    const byte * key,
    word32 len
)
```

この機能は、AES GCM（Galois/Counter Mode）の鍵を設定するために使用されます。与えられた key で Aes 構造体を初期化します。コンパイル時に Configure オプション HAVE_AESGCM が有効になっている場合にのみ有効になります。

Parameters:

- **aes** データの暗号化に使用される Aes 構造体へのポインタ
- **key** 暗号化と復号のための 16,24、または 32 バイトの秘密鍵 *Example*

```
Aes enc;
int ret = 0;
byte key[] = { some 16, 24,32 byte key };
if (ret = wc_AesGcmSetKey(&enc, key, sizeof(key)) != 0) {
// failed to set aes key
}
```

See:

- wc_AesGcmEncrypt
- wc_AesGcmDecrypt

Return:

- 0 鍵の設定に成功しました。
- BAD_FUNC_ARG 与えられた key が無効な長さの場合に返されます。

B.13.2.10 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
)
```

この関数は、バッファ in に格納されている平文メッセージを暗号化し結果を出力バッファ out に出力します。暗号化する呼び出しごとに新しい iv(初期化ベクトル) が必要です。また、入力認証ベクトル、authIn、authTag への入力認証ベクトルをエンコードします。

Parameters:

- **aes** データの暗号化に使用される AES オブジェクトへのポインタ
- **out** 暗号テキストを出力する先のバッファへのポインタ。バッファサイズは入力バッファ in のサイズ (sz) と同じでなければなりません。
- **in** 暗号化する平文メッセージを保持している入力バッファへのポインタ。サイズは AES_BLOCK_SIZE の倍数でなければなりません。パディングされている必要があります。
- **sz** 暗号化する入力メッセージの長さ
- **iv** 初期化ベクトルを含むバッファへのポインタ
- **ivSz** 初期化ベクトルの長さ
- **authTag** 認証タグを保存するバッファへのポインタ
- **authTagSz** 希望の認証タグの長さ
- **authIn** 入力認証ベクトルを含むバッファへのポインタ *Example*

```
Aes enc;
// initialize aes structure by calling 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));
```

See:

- wc_AesGcmSetKey
- wc_AesGcmDecrypt

Return: 0 入力メッセージの暗号化に成功しました

B.13.2.11 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
)
```

この関数は、バッファ in で与えられた入力暗号テキストを復号し、結果を出力バッファ out に格納します。また、指定された認証タグ、authTag に対して、入力認証ベクトル、authIn をチェックします。

Parameters:

- **aes** データの復号に使用される Aes 構造体へのポインタ
- **out** メッセージテキストを保存する出力バッファへのポインタ。サイズは入力バッファ in のサイズ (sz) と同じでなければならない。
- **in** 暗号テキストを保持する入力バッファへのポインタ。サイズは AES_BLOCK_SIZE の倍数でなければならない。
- **sz** 復号する暗号テキストの長さ
- **iv** 初期化ベクトルを含むバッファへのポインタ
- **ivSz** 初期化ベクトルの長さ
- **authTag** 認証タグを含むバッファへのポインタ
- **authTagSz** 希望の認証タグの長さ
- **authIn** 入力認証ベクトルを含むバッファへのポインタ *Example*

```
Aes enc; //can use the same struct as was passed to wc_AesGcmEncrypt
// initialize aes structure by calling 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));
```

See:

- wc_AesGcmSetKey
- wc_AesGcmEncrypt

Return:

- 0 入力メッセージの復号に成功しました
- AES_GCM_AUTH_E 認証タグが提供された認証コードベクトルと一致しない場合、authtag。

B.13.2.12 function wc_GmacSetKey

```
int wc_GmacSetKey(
    Gmac * gmac,
    const byte * key,
    word32 len
)
```

この関数は、GAROIS メッセージ認証に使用される Gmac 構造体の鍵を初期化して設定します。

Parameters:

- **gmac** 認証に使用される Gmac 構造体へのポインタ
- **key** 認証のための 16,24、または 32 バイトの秘密鍵 *Example*

```
Gmac gmac;
key[] = { some 16, 24, or 32 byte length key };
wc_GmacSetKey(&gmac, key, sizeof(key));
```

See: wc_GmacUpdate

Return:

- 0 鍵の設定に成功しました
- BAD_FUNC_ARG 引数 key の長さが無効な場合は返されます。

B.13.2.13 function wc_GmacUpdate

```
int wc_GmacUpdate(
    Gmac * gmac,
    const byte * iv,
    word32 ivSz,
    const byte * authIn,
    word32 authInSz,
    byte * authTag,
    word32 authTagSz
)
```

この関数は authIn Input の GMAC ハッシュを生成し、結果を authTag バッファに格納します。wc_GmacUpdate を実行した後、生成された authTag を既知の認証タグに比較してメッセージの信頼性を検証する必要があります。

Parameters:

- **gmac** 認証に使用される Gmac 構造体へのポインタ
- **iv** ハッシュに使用される初期化ベクトル

- **ivSz** 使用される初期化ベクトルのサイズ
- **authIn** 確認する認証ベクトルを含むバッファへのポインタ
- **authInSz** 認証ベクトルのサイズ
- **authTag** GMAC ハッシュを保存する出力バッファへのポインタ *Example*

```
Gmac gmac;
key[] = { some 16, 24, or 32 byte length key };
iv[] = { some 16 byte length iv };

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));
```

See: wc_GmacSetKey

Return: 0 GMAC ハッシュの計算に成功しました。

B.13.2.14 function wc_AesCcmSetKey

```
int wc_AesCcmSetKey(
    Aes * aes,
    const byte * key,
    word32 keySz
)
```

この関数は、CCM を使用して AES オブジェクトの鍵を設定します（CBC-MAC のカウンタ）。Aes 構造体へのポインタを取り、引数で与えられた key で初期化します。

Parameters:

- **aes** 引数 key を保管するための Aes 構造体
- **key** 暗号化と復号のための 16,24、または 32 バイトの秘密鍵 *Example*

```
Aes enc;
key[] = { some 16, 24, or 32 byte length key };

wc_AesCcmSetKey(&aes, key, sizeof(key));
```

See:

- wc_AesCcmEncrypt
- wc_AesCcmDecrypt

Return: none

B.13.2.15 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
)
```

この関数は、CCM を使用して、入力メッセージ、IN、OUT、OUT、OUT を CCM（CBC-MAC のカウンタ）を暗号化します。その後、Authin Input から認証タグ、AuthtAg を計算して格納します。

Parameters:

- **aes** データの暗号化に使用される Aes 構造体へのポインタ
- **out** 暗号テキストを保存する出力バッファへのポインタ
- **in** 暗号化するメッセージを保持している入力バッファへのポインタ
- **sz** 暗号化する入力メッセージの長さ
- **nonce** nonce を含むバッファへのポインタ（1 回だけ使用されている数）
- **nonceSz** ノンスの長さ
- **authTag** 認証タグを保存するバッファへのポインタ
- **authTagSz** 希望の認証タグの長さ
- **authIn** 入力認証ベクトルを含むバッファへのポインタ *Example*

```
Aes enc;
// initialize enc with 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));
```

See:

- wc_AesCcmSetKey
- wc_AesCcmDecrypt

Return: none

B.13.2.16 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
)
```

この関数は、CCM を使用して、入力暗号テキストを、CCM（CBC-MAC のカウンタ）を使用して出力バッファーに復号します。その後、authIn 入力から authTag を計算します。認証タグが無効な場合は、出力バッファをゼロに設定し、AES_CCM_AUTH_E を返します。

Parameters:

- **aes** データの復号に使用される Aes 構造体へのポインタ
- **out** 復号したテキストを出力する出力バッファへのポインタ
- **in** 復号するメッセージを保持している入力バッファへのポインタ
- **sz** 入力暗号テキストのサイズ
- **nonce** nonce を含むバッファへのポインタ（1 回だけ使用されている数）
- **nonceSz** ノンスの長さ
- **authTag** 認証タグを保存するバッファへのポインタ
- **authTagSz** 希望の認証タグの長さ
- **authIn** 入力認証ベクトルを含むバッファへのポインタ *Example*

```
Aes dec;
// initialize dec with 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
}
```

See:

- wc_AesCcmSetKey
- wc_AesCcmEncrypt

Return:

- 0 入力メッセージの復号に成功しました
- AES_CCM_AUTH_E 認証タグが提供された認証コードベクトルと一致しない場合

B.13.2.17 function wc_AesXtsSetKey

```
int wc_AesXtsSetKey(
    XtsAes * aes,
    const byte * key,
    word32 len,
    int dir,
    void * heap,
    int devId
)
```

この関数は、AES XTS モードを使用する暗号化または復号で使用する鍵の設定に使用します。完了したら、AES キーで wc_AesXtsFree を呼び出すことがユーザーになりました。

Parameters:

- **aes** 暗号化または復号処理に使用する XtsAes 構造体
- **key** 補正値（Tewak）を加味した AES 鍵を保持しているバッファ
- **len** 鍵バッファのサイズ。鍵サイズの 2 倍にする必要があります。（すなわち、16 バイトの鍵の場合は 32）
- **dir** 処理方向、AES_Encryption または AES_Decryption のいずれかを指定します。
- **heap** メモリに使用するヒープヒント。NULL を設定することもできます。*Example*

```
XtsAes aes;

if(wc_AesXtsSetKey(&aes, key, sizeof(key), AES_ENCRYPTION, NULL, 0) != 0)
{
    // Handle error
}
wc_AesXtsFree(&aes);
```

See:

- wc_AesXtsEncrypt
- wc_AesXtsDecrypt
- wc_AesXtsFree

Return: 0 成功

B.13.2.18 function wc_AesXtsEncryptSector

```
int wc_AesXtsEncryptSector(
    XtsAes * aes,
    byte * out,
    const byte * in,
    word32 sz,
    word64 sector
)
```

wc_AesXtsEncrypt と同じ処理を行いますが、バイト配列の代わりに Tweak 値として word64 型を使用します。本関数で word64 をバイト配列に変換し、wc_AesXtsEncrypt を呼び出します。

Parameters:

- **aes** ブロック暗号化/復号に使用する XtsAes 構造体
- **out** 暗号テキストを保持するための出力バッファ
- **in** 暗号化する入力プレーンテキストバッファ
- **sz** バッファ（in, out 両方）のサイズ *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);
```

See:

- wc_AesXtsEncrypt
- wc_AesXtsDecrypt
- wc_AesXtsSetKey
- wc_AesXtsFree

Return: 0 成功

B.13.2.19 function wc_AesXtsDecryptSector

```
int wc_AesXtsDecryptSector(
    XtsAes * aes,
    byte * out,
    const byte * in,
    word32 sz,
    word64 sector
)
```

wc_AesXtsDecrypt と同じ処理を行いますが、バイト配列の代わりに Tweak 値として word64 タイプを使用します。本関数で word64 をバイト配列に変換するだけです。

Parameters:

- **aes** ブロック暗号化/復号に使用する XtsAes 構造体
- **out** プレーンテキストを保持するための出力バッファ
- **in** 復号する暗号テキストバッファ
- **sz** バッファ（in, out 両方）のサイズ *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);
```

See:

- wc_AesXtsEncrypt
- wc_AesXtsDecrypt
- wc_AesXtsSetKey
- wc_AesXtsFree

Return: 0 成功

B.13.2.20 function wc_AesXtsEncrypt

```
int wc_AesXtsEncrypt(
    XtsAes * aes,
    byte * out,
    const byte * in,
    word32 sz,
    const byte * i,
    word32 iSz
)
```

AES XTS モードで暗号化します。(XTS) XEX 暗号化と平文がブロック長の倍数でない場合の処理 (Ciphertext Stealing) を行います。

Parameters:

- **aes** ブロック暗号化/復号に使用する XtsAes 構造体
- **out** 暗号テキストを保持するための出力バッファ

- **in** 暗号化する入力プレーンテキストを含むバッファ
- **sz** バッファ（in, out 両方）のサイズ
- **i** Tweak に使用する値 *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);
```

See:

- wc_AesXtsDecrypt
- wc_AesXtsSetKey
- wc_AesXtsFree

Return: 0 成功

B.13.2.21 function wc_AesXtsDecrypt

```
int wc_AesXtsDecrypt(
    XtsAes * aes,
    byte * out,
    const byte * in,
    word32 sz,
    const byte * i,
    word32 iSz
)
```

暗号化と同じプロセスですが、XtsAes 構造体は AES_Decryption タイプです。

Parameters:

- **aes** ブロック暗号化/復号に使用する XtsAes 構造体
- **out** プレーンテキストを保持するための出力バッファ
- **in** 復号する暗号テキストを含むバッファ
- **sz** バッファ（in, out 両方）のサイズ
- **i** Tweak に使用する値 *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);
```

See:

- wc_AesXtsEncrypt
- wc_AesXtsSetKey
- wc_AesXtsFree

Return: 0 成功

B.13.2.22 function wc_AesXtsFree

```
int wc_AesXtsFree(
    XtsAes * aes
)
```

この関数は XtsAes 構造体で使用されるすべてのリソースを解放します。

See:

- wc_AesXtsEncrypt
- wc_AesXtsDecrypt
- wc_AesXtsSetKey

Return: 0 成功 *Example*

```
XtsAes aes;

if(wc_AesXtsSetKey(&aes, key, sizeof(key), AES_ENCRYPTION, NULL, 0) != 0)
{
    // Handle error
}
wc_AesXtsFree(&aes);
```

B.13.2.23 function wc_AesInit

```
int wc_AesInit(
    Aes * aes,
    void * heap,
    int devId
)
```

Aes 構造体を初期化します。ヒープヒントを設定し、ASYNC ハードウェアを使用する場合の ID も設定します。Aes 構造体の使用が終了した際に wc_AesFree を呼び出すのはユーザーに任されています。

Parameters:

- **aes** 初期化対象の Aes 構造体
- **heap** 必要に応じて malloc / free に使用するヒープヒント *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);
```

See:

- wc_AesSetKey
- wc_AesSetIV

Return: 0 成功

B.13.2.24 function wc_AesFree

```
int wc_AesFree(
    Aes * aes
)
```

Aes 構造体に関連つけられたリソースを可能なら解放します。内部的にはノーオペレーションとなることもありますが、ベストプラクティスとしてどのケースでもこの関数を呼び出すことを推奨します。

Parameters:

- **aes** Free すべき Aes 構造体へのポインタ *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);
```

See: wc_AesInit

Return: 戻り値なし

B.13.2.25 function wc_AesCfbEncrypt

```
int wc_AesCfbEncrypt(
    Aes * aes,
    byte * out,
    const byte * in,
    word32 sz
)
```

AES CFB モードで暗号化を行います。

Parameters:

- **aes** ブロック暗号化/復号に使用する Aes 構造体
- **out** 暗号テキストを保持するための出力バッファは、少なくとも入力プレーンテキストバッファと同じサイズが必要です。
- **in** 暗号化する入力プレーンテキストバッファ *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
}
```

See:

- wc_AesCfbDecrypt
- wc_AesSetKey

Return: 0 成功時に返ります。失敗時には負値が返されます。

B.13.2.26 function wc_AesCfbDecrypt

```
int wc_AesCfbDecrypt(
    Aes * aes,
    byte * out,
    const byte * in,
    word32 sz
)
```

AES CFB モードで復号を行います。

Parameters:

- **aes** ブロック暗号化/復号に使用する Aes 構造体
- **out** 復号されたテキストを保持するための出力バッファは、少なくとも入力バッファと同じサイズが必要です。
- **in** 復号する暗号データを保持した入力バッファ *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
}
```

See:

- wc_AesCfbEncrypt
- wc_AesSetKey

Return: 0 成功時に返ります。失敗時には負値が返されます。

B.13.2.27 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
)
```

この関数は、RFC 5297 に記載されているように SIV（合成初期化ベクトル）暗号化を実行します。

Parameters:

- **key** 使用する鍵を含むバイトバッファ。
- **keySz** 鍵バッファの長さ（バイト単位）。
- **assoc** 追加の認証された関連データ（AD）。
- **assocSz** AD バッファのバイト数
- **nonce** ナンス（一度だけ使用される値）。AD と同じ方法でアルゴリズムによって使用されます。

- **nonceSz** バイト単位のナンスバッファの長さ。
- **in** 暗号化する平文のバッファ。
- **inSz** 平文バッファの長さ
- **siv** S2V による SIV 出力（RFC 5297 2.4 参照）。*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
}
```

See: wc_AesSivDecrypt

Return:

- 0 暗号化に成功した場合
- BAD_FUNC_ARG 鍵、SIV、または出力バッファが NULL の場合。鍵サイズが 32,48、または 64 バイトの場合にも返されます。
- Other その他の負のエラー値。AES または CMAC 操作が失敗した場合に返されます。

B.13.2.28 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
)
```

この機能は、RFC 5297 に記載されているように SIV（合成初期化ベクトル）復号を実行します

Parameters:

- **key** 使用する鍵を含むバイトバッファ。
- **keySz** 鍵バッファの長さ（バイト単位）。
- **assoc** 追加の認証された関連データ（AD）。
- **assocSz** AD バッファのバイト数
- **nonce** ナンス（一度だけ使用される値）。AD と同じ方法で、基礎となるアルゴリズムによって使用されます。
- **nonceSz** バイト単位のナンスバッファの長さ。
- **in** 復号する暗号文バッファ。
- **inSz** 暗号文バッファの長さ
- **siv** 暗号文に付随する SIV（RFC 5297 2.4 を参照）。*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
}
```

See: wc_AesSivEncrypt

Return:

- 0 復号に成功した場合
- BAD_FUNC_ARG 鍵、SIV、または出力バッファが NULL の場合。キーサイズが 32,48、または 64 バイトの場合にも返されます。
- AES_SIV_AUTH_E S2V によって派生した SIV が入力 SIV と一致しない場合（RFC 5297 2.7 を参照）。
- Other その他の負のエラー値。AES または CMAC 操作が失敗した場合に返されます。

B.13.2.29 function wc_AesCbcDecryptWithKey

```
int wc_AesCbcDecryptWithKey(
    byte * out,
    const byte * in,
    word32 inSz,
    const byte * key,
    word32 keySz,
    const byte * iv
)
```

入力バッファーから暗号を復号化し、AES で Cipher Block Chaining を使用して出力バッファに出力バッファーに入れます。この関数は、AES 構造を初期化する必要はありません。代わりに、キーと IV（初期化ベクトル）を取り、これらを使用して AES オブジェクトを初期化してから暗号テキストを復号化します。

Parameters:

- **out** 復号化されたメッセージのプレーンテキストを保存する出力バッファへのポインタ
- **in** 復号化される暗号テキストを含む入力バッファへのポインタ
- **inSz** 入力メッセージのサイズ
- **key** 復号化のための 16,24、または 32 バイトの秘密鍵 *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
}
```

See:

- wc_AesSetKey
- wc_AesSetIV
- wc_AesCbcEncrypt
- wc_AesCbcDecrypt

Return:

- 0 メッセージの復号化に成功しました
- BAD_ALIGN_E ブロック整列エラーに戻りました
- BAD_FUNC_ARG aesetiv の間にキーの長さが無効な場合、または AES オブジェクトが NULL の場合
- MEMORY_E wolfssl_small_stack が有効になっていて、xmalloc が AES オブジェクトのインスタンス化に失敗した場合に返されます。

B.14 Algorithms - ARC4

B.14.1 Functions

	Name
int	**wc_Arc4Process**(Arc4 * arc4, byte * out, const byte * in, word32 length) この関数は、バッファ内の入力メッセージを暗号化し、出力バッファーに暗号文を配置するか、またはバッファーから暗号文を復号化したり、ARC4 暗号化を使用して、出力バッファー OUT を出力したりします。この関数は暗号化と復号化の両方に使用されます。この方法が呼び出される可能性がある場合は、まず WC_ARC4SETKEY を使用して ARC4 構造を初期化する必要があります。
int	**wc_Arc4SetKey**(Arc4 * arc4, const byte * key, word32 length) この関数は ARC4 オブジェクトのキーを設定し、それを暗号として使用するために初期化します。WC_ARC4PROCESS を使用した暗号化に使用する前に呼び出される必要があります。

B.14.2 Functions Documentation

B.14.2.1 function wc_Arc4Process

```
int wc_Arc4Process(
    Arc4 * arc4,
    byte * out,
    const byte * in,
    word32 length
)
```

この関数は、バッファ内の入力メッセージを暗号化し、出力バッファーに暗号文を配置するか、またはバッファーから暗号文を復号化したり、ARC4 暗号化を使用して、出力バッファー OUT を出力したりします。この関数は暗号化と復号化の両方に使用されます。この方法が呼び出される可能性がある場合は、まず WC_ARC4SETKEY を使用して ARC4 構造を初期化する必要があります。

Parameters:

- **arc4** メッセージの処理に使用される ARC4 構造へのポインタ
- **out** 処理されたメッセージを保存する出力バッファへのポインタ
- **in** プロセスするメッセージを含む入力バッファへのポインタ *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));
```

See: wc_Arc4SetKey

Return: none

B.14.2.2 function wc_Arc4SetKey

```
int wc_Arc4SetKey(
    Arc4 * arc4,
    const byte * key,
    word32 length
)
```

この関数は ARC4 オブジェクトのキーを設定し、それを暗号として使用するために初期化します。WC_ARC4PROCESS を使用した暗号化に使用する前に呼び出される必要があります。

Parameters:

- **arc4** 暗号化に使用される ARC4 構造へのポインタ
- **key** ARC4 構造を初期化するためのキー *Example*

```
Arc4 enc;
byte key[] = { initialize with key to use for encryption };
wc_Arc4SetKey(&enc, key, sizeof(key));
```

See: wc_Arc4Process

Return: none

B.15 Algorithms - BLAKE2

B.15.1 Functions

	Name
int	**wc_InitBlake2b**(Blake2b * b2b, word32 digestSz) この関数は Blake2 Hash 関数で使用するための Blake2b 構造を初期化します。
int	**wc_Blake2bUpdate**(Blake2b * b2b, const byte * data, word32 sz) この関数は、与えられた入力データと Blake2B ハッシュを更新します。この関数は、wc_initblake2b の後に呼び出され、最後のハッシュ：wc_blake2bfinal の準備ができているまで繰り返します。

	Name
int	**wc_Blake2bFinal**(Blake2b * b2b, byte * final, word32 requestSz) この関数は、以前に供給された入力データの Blake2b ハッシュを計算します。出力ハッシュは長さ REQUESTSZ、あるいは要求された場合は B2B 構造の DigestSZ を使用します。この関数は、wc_initblake2b の後に呼び出され、wc_blake2bupdate は必要な各入力データに対して処理されています。

B.15.2 Functions Documentation

B.15.2.1 function wc_InitBlake2b

```
int wc_InitBlake2b(
    Blake2b * b2b,
    word32 digestSz
)
```

この関数は Blake2 Hash 関数で使用するための Blake2b 構造を初期化します。

Parameters:

- **b2b** 初期化するために Blake2b 構造へのポインタ *Example*

```
Blake2b b2b;
// initialize Blake2b structure with 64 byte digest
wc_InitBlake2b(&b2b, 64);
```

See: wc_Blake2bUpdate

Return: 0 Blake2B 構造の初期化に成功し、ダイジェストサイズを設定したときに返されます。

B.15.2.2 function wc_Blake2bUpdate

```
int wc_Blake2bUpdate(
    Blake2b * b2b,
    const byte * data,
    word32 sz
)
```

この関数は、与えられた入力データと Blake2B ハッシュを更新します。この関数は、wc_initblake2b の後に呼び出され、最後のハッシュ：wc_blake2bfinal の準備ができているまで繰り返します。

Parameters:

- **b2b** 更新する Blake2b 構造へのポインタ
- **data** 追加するデータを含むバッファへのポインタ *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
}
```

See:

- wc_InitBlake2b
- wc_Blake2bFinal

Return:

- 0 与えられたデータを使用して Blake2B 構造を正常に更新すると返されます。
- -1 入力データの圧縮中に障害が発生した場合

B.15.2.3 function wc_Blake2bFinal

```
int wc_Blake2bFinal(
    Blake2b * b2b,
    byte * final,
    word32 requestSz
)
```

この関数は、以前に供給された入力データの Blake2b ハッシュを計算します。出力ハッシュは長さ REQUESTSZ、あるいは要求された場合は B2B 構造の DigestSZ を使用します。この関数は、wc_initblake2b の後に呼び出され、wc_blake2bupdate は必要な各入力データに対して処理されています。

Parameters:

- **b2b** 更新する Blake2b 構造へのポインタ
- **final** Blake2B ハッシュを保存するバッファへのポインタ。長さ requestsz にする必要があります

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
}
```

See:

- wc_InitBlake2b
- wc_Blake2bUpdate

Return:

- 0 Blake2B Hash の計算に成功したときに返されました
- -1 blake2b ハッシュを解析している間に失敗がある場合

B.16 Algorithms - Camellia

B.16.1 Functions

	Name
int	**wc_CamelliaSetKey**(Camellia * cam, const byte * key, word32 len, const byte * iv) この関数は、Camellia オブジェクトのキーと初期化ベクトルを設定し、それを暗号として使用するために初期化します。
int	**wc_CamelliaSetIV**(Camellia * cam, const byte * iv) この関数は、Camellia オブジェクトの初期化ベクトルを設定します。
int	**wc_CamelliaEncryptDirect**(Camellia * cam, byte * out, const byte * in) この機能は、提供された Camellia オブジェクトを使用して 1 ブロック暗号化します。それはバッファーの最初の 16 バイトブロックを解析し、暗号化結果をバッファアウトに格納します。この機能を使用する前に、WC_CAMELLIASETKEY を使用して Camellia オブジェクトを初期化する必要があります。
int	**wc_CamelliaDecryptDirect**(Camellia * cam, byte * out, const byte * in) この機能は、提供された Camellia オブジェクトを使用して 1 ブロック復号化します。それはバッファ内の最初の 16 バイトブロックを解析し、それを復号化し、結果をバッファアウトに格納します。この機能を使用する前に、WC_CAMELLIASETKEY を使用して Camellia オブジェクトを初期化する必要があります。
int	**wc_CamelliaCbcEncrypt**(Camellia * cam, byte * out, const byte * in, word32 sz) この関数は、バッファーの平文を暗号化し、その出力をバッファ OUT に格納します。暗号ブロックチェーン（CBC）を使用して Camellia を使用してこの暗号化を実行します。
int	**wc_CamelliaCbcDecrypt**(Camellia * cam, byte * out, const byte * in, word32 sz) この関数は、バッファ内の暗号文を復号化し、その出力をバッファ OUT に格納します。暗号ブロックチェーン（CBC）を搭載した Camellia を使用してこの復号化を実行します。

B.16.2 Functions Documentation

B.16.2.1 function wc_CamelliaSetKey

```
int wc_CamelliaSetKey(
    Camellia * cam,
    const byte * key,
    word32 len,
    const byte * iv
)
```

この関数は、Camellia オブジェクトのキーと初期化ベクトルを設定し、それを暗号として使用するために初期化します。

Parameters:

- **cam** キーと IV を設定する椿構造へのポインタ
- **key** 暗号化と復号化に使用する 16,24、または 32 バイトのキーを含むバッファへのポインタ
- **len** 渡されたキーの長さ *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
}
```

See:

- wc_CamelliaEncryptDirect
- wc_CamelliaDecryptDirect
- wc_CamelliaCbcEncrypt
- wc_CamelliaCbcDecrypt

Return:

- 0 キーと初期化ベクトルを正常に設定すると返されます
- BAD_FUNC_ARG 入力引数の 1 つがエラー処理がある場合に返されます
- MEMORY_E xmalloc でメモリを割り当てるエラーがある場合

B.16.2.2 function wc_CamelliaSetIV

```
int wc_CamelliaSetIV(
    Camellia * cam,
    const byte * iv
)
```

この関数は、Camellia オブジェクトの初期化ベクトルを設定します。

Parameters:

- **cam** IV を設定する椿構造へのポインタ *Example*

```
Camellia cam;
byte iv[16];
// initialize iv
if( wc_CamelliaSetIV(&cam, iv) != 0) {
// error initializing camellia structure
}
```

See: wc_CamelliaSetKey

Return:

- 0 キーと初期化ベクトルを正常に設定すると返されます
- BAD_FUNC_ARG 入力引数の 1 つがエラー処理がある場合に返されます

B.16.2.3 function wc_CamelliaEncryptDirect

```
int wc_CamelliaEncryptDirect(
    Camellia * cam,
    byte * out,
    const byte * in
)
```

この機能は、提供された Camellia オブジェクトを使用して 1 ブロック暗号化します。それはバッファーの最初の 16 バイトブロックを解析し、暗号化結果をバッファアウトに格納します。この機能を使用する前に、WC_CAMELLIASETKEY を使用して Camellia オブジェクトを初期化する必要があります。

Parameters:

- **cam** 暗号化に使用する椿構造へのポインタ
- **out** 暗号化されたブロックを保存するバッファへのポインタ *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);
```

See: wc_CamelliaDecryptDirect

Return: none いいえ返します。

B.16.2.4 function wc_CamelliaDecryptDirect

```
int wc_CamelliaDecryptDirect(
    Camellia * cam,
    byte * out,
    const byte * in
)
```

この機能は、提供された Camellia オブジェクトを使用して 1 ブロック復号化します。それはバッファ内の最初の 16 バイトブロックを解析し、それを復号化し、結果をバッファアウトに格納します。この機能を使用する前に、WC_CAMELLIASETKEY を使用して Camellia オブジェクトを初期化する必要があります。

Parameters:

- **cam** 暗号化に使用する椿構造へのポインタ
- **out** 復号化された平文ブロックを保存するバッファへのポインタ *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);
```

See: wc_CamelliaEncryptDirect

Return: none いいえ返します。

B.16.2.5 function wc_CamelliaCbcEncrypt

```
int wc_CamelliaCbcEncrypt(
    Camellia * cam,
    byte * out,
    const byte * in,
    word32 sz
)
```

この関数は、バッファーの平文を暗号化し、その出力をバッファ OUT に格納します。暗号ブロックチェーン（CBC）を使用して Camellia を使用してこの暗号化を実行します。

Parameters:

- **cam** 暗号化に使用する椿構造へのポインタ
- **out** 暗号化された暗号文を保存するバッファへのポインタ
- **in** 暗号化する平文を含むバッファへのポインタ *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));
```

See: wc_CamelliaCbcDecrypt

Return: none いいえ返します。

B.16.2.6 function wc_CamelliaCbcDecrypt

```
int wc_CamelliaCbcDecrypt(
    Camellia * cam,
    byte * out,
    const byte * in,
    word32 sz
)
```

この関数は、バッファ内の暗号文を復号化し、その出力をバッファ OUT に格納します。暗号ブロックチェーン（CBC）を搭載した Camellia を使用してこの復号化を実行します。

Parameters:

- **cam** 暗号化に使用する椿構造へのポインタ
- **out** 復号化されたメッセージを保存するバッファへのポインタ
- **in** 暗号化された暗号文を含むバッファへのポインタ *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));
```

See: wc_CamelliaCbcEncrypt

Return: none いいえ返します。

B.17 Algorithms - ChaCha

B.17.1 Functions

	Name
int	**wc_Chacha_SetIV**(ChaCha * ctx, const byte * inIv, word32 counter) この関数は Chacha オブジェクトの初期化ベクトル（nonce）を設定し、暗号として使用するために初期化します。WC_CHACHA_SETKEY を使用して、キーが設定された後に呼び出されるべきです。暗号化の各ラウンドに差し違いを使用する必要があります。

	Name
int	**wc_Chacha_Process**(ChaCha * ctx, byte * cipher, const byte * plain, word32 msglen) この関数は、バッファ入力からテキストを処理し、暗号化または復号化し、結果をバッファ出力に格納します。
int	**wc_Chacha_SetKey**(ChaCha * ctx, const byte * key, word32 keySz) この関数は Chacha オブジェクトのキーを設定し、それを暗号として使用するために初期化します。NONCE を WC_CHACHA_SETIV で設定する前に、WC_CHACHA_PROCESS を使用した暗号化に使用する前に呼び出す必要があります。

B.17.2 Functions Documentation

B.17.2.1 function wc_Chacha_SetIV

```
int wc_Chacha_SetIV(
    ChaCha * ctx,
    const byte * inIv,
    word32 counter
)
```

この関数は Chacha オブジェクトの初期化ベクトル（nonce）を設定し、暗号として使用するために初期化します。WC_CHACHA_SETKEY を使用して、キーが設定された後に呼び出されるべきです。暗号化の各ラウンドに差し違いを使用する必要があります。

Parameters:

- **ctx** IV を設定する Chacha 構造へのポインタ
- **inIv** Chacha 構造を初期化するための 12 バイトの初期化ベクトルを含むバッファへのポインタ

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
}
```

See:

- wc_Chacha_SetKey
- wc_Chacha_Process

Return:

- 0 初期化ベクトルを正常に設定すると返されます
- BAD_FUNC_ARG CTX 入力引数の処理中にエラーが発生した場合

B.17.2.2 function wc_Chacha_Process

```
int wc_Chacha_Process(
    ChaCha * ctx,
    byte * cipher,
    const byte * plain,
```

```
    word32 msglen
)
```

この関数は、バッファ入力からテキストを処理し、暗号化または復号化し、結果をバッファ出力に格納します。

Parameters:

- **ctx** IV を設定する Chacha 構造へのポインタ
- **output** 出力暗号文または復号化された平文を保存するバッファへのポインタ
- **input** 暗号化する入力平文を含むバッファへのポインタまたは復号化する入力暗号文 *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
}
```

See:

- wc_Chacha_SetKey
- wc_Chacha_Process

Return:

- 0 入力の暗号化または復号化に成功したときに返されます
- BAD_FUNC_ARG CTX 入力引数の処理中にエラーが発生した場合

B.17.2.3 function wc_Chacha_SetKey

```
int wc_Chacha_SetKey(
    ChaCha * ctx,
    const byte * key,
    word32 keySz
)
```

この関数は Chacha オブジェクトのキーを設定し、それを暗号として使用するために初期化します。NONCE を WC_CHACHA_SETIV で設定する前に、WC_CHACHA_PROCESS を使用した暗号化に使用する前に呼び出す必要があります。

Parameters:

- **ctx** キーを設定する Chacha 構造へのポインタ
- **key** Chacha 構造を初期化するための 16 または 32 バイトのキーを含むバッファへのポインタ *Example*

```
ChaCha enc;
byte key[] = { // initialize key };

if( wc_Chacha_SetKey(&enc, key, sizeof(key)) != 0) {
    // error initializing ChaCha structure
}
```

See:

- wc_Chacha_SetIV
- wc_Chacha_Process

Return:

- 0 キーの設定に成功したときに返されます
- BAD_FUNC_ARG CTX 入力引数の処理中にエラーが発生した場合、またはキーが 16 または 32 バイトの長さがある場合

B.18 Algorithms - ChaCha20_Poly1305

More...

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]) この関数は、Chacha20 Stream 暗号を使用して、Output BufferText に入力メッセージ、InPleaintext を暗号化します。また、Poly-1305 認証（暗号テキスト）を実行し、生成した認証タグを出力バッファ OutauthTag に格納します。
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) この関数は、Chacha20 Stream 暗号を使用して、出力バッファ OutpleAntext に復号したデータを出力します。また、Poly-1305 認証を実行し、指定された inAuthTag を inAAD で生成された認証（任意の長さの追加認証データ）と比較します。注：生成された認証タグが提供された認証タグと一致しない場合、テキストは復号されません。

B.18.2 Detailed Description

B.18.3 Functions Documentation

B.18.3.1 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]
)
```

この関数は、Chacha20 Stream 暗号を使用して、Output BufferText に入力メッセージ、InPleaintext を暗号化します。また、Poly-1305 認証（暗号テキスト）を実行し、生成した認証タグを出力バッファ OutauthTag に格納します。

Parameters:

- **inKey** 暗号化に使用する 32 バイトの鍵を含むバッファへのポインタ
- **inIv** 暗号化に使用する 12 バイトの IV を含むバッファへのポインタ
- **inAAD** 任意の長さの追加認証データ（AAD）を含むバッファへのポインタ
- **inAADLen** 入力 AAD の長さ
- **inPlaintext** 暗号化する平文を含むバッファへのポインタ
- **inPlaintextLen** 暗号化するプレーンテキストの長さ
- **outCiphertext** 暗号文を保存するバッファーへのポインタ *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
}
```

See:

- wc_ChaCha20Poly1305_Decrypt
- wc_ChaCha_*
- wc_Poly1305*

Return:

- 0 メッセージの暗号化に成功したら返されます
- BAD_FUNC_ARG 暗号化プロセス中にエラーがある場合

B.18.3.2 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
)
```

この関数は、Chacha20 Stream 暗号を使用して、出力バッファ OutpleAntext に復号したデータを出力します。また、Poly-1305 認証を実行し、指定された inAuthTag を inAAD で生成された認証（任意の長さの追加認証データ）と比較します。注：生成された認証タグが提供された認証タグと一致しない場合、テキストは復号されません。

Parameters:

- **inKey** 復号に使用する 32 バイトの鍵を含むバッファへのポインタ
- **inIv** 復号に使用する 12 バイトの IV を含むバッファへのポインタ
- **inAAD** 任意の長さの追加認証データ（AAD）を含むバッファへのポインタ
- **inAADLen** 入力 AAD の長さ
- **inCiphertext** 復号する暗号文を含むバッファへのポインタ
- **outCiphertextLen** 復号する暗号文の長さ
- **inAuthTag** 認証のための 16 バイトのダイジェストを含むバッファへのポインタ *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
}
```

See:

- wc_ChaCha20Poly1305_Encrypt
- wc_ChaCha_*
- wc_Poly1305*

Return:

- 0 メッセージの復号に成功したときに返されました
- BAD_FUNC_ARG 関数引数のいずれかが予想されるものと一致しない場合に返されます
- MAC_CMP_FAILED_E 生成された認証タグが提供されている inAuthTag と一致しない場合に返されます。

B.19 Callbacks - CryptoCb

B.20 Algorithms - Curve25519

B.20.1 Functions

	Name
int	**wc_curve25519_make_key**(WC_RNG * rng, int keysize, curve25519_key * key) この関数は、与えられたサイズ（Keysize）の指定された乱数発生器 RNG を使用して Curve25519 キーを生成し、それを指定された Curve25519_Key 構造体に格納します。キー構造が WC_CURVE25519_INIT（）を介して初期化された後に呼び出されるべきです。
int	**wc_curve25519_shared_secret**(curve25519_key * private_key, curve25519_key * public_key, byte * out, word32 * outlen) この関数は、秘密の秘密鍵と受信した公開鍵を考えると、共有秘密鍵を計算します。生成された秘密鍵をバッファアウトに保存し、ounlent の秘密鍵の変数を割り当てます。ビッグエンディアンのみをサポートします。
int	**wc_curve25519_shared_secret_ex**(curve25519_key * private_key, curve25519_key * public_key, byte * out, word32 * outlen, int endian) この関数は、秘密の秘密鍵と受信した公開鍵を考えると、共有秘密鍵を計算します。生成された秘密鍵をバッファアウトに保存し、ounlent の秘密鍵の変数を割り当てます。ビッグ・リトルエンディアンの両方をサポートします。
int	**wc_curve25519_init**(curve25519_key * key) この関数は Curve25519 キーを初期化します。構造のキーを生成する前に呼び出されるべきです。
void	**wc_curve25519_free**(curve25519_key * key) この関数は Curve25519 オブジェクトを解放します。*Example*
int	**wc_curve25519_import_private**(const byte * priv, word32 privSz, curve25519_key * key) この関数は Curve25519 秘密鍵のみをインポートします。（ビッグエンディアン）。
int	**wc_curve25519_import_private_ex**(const byte * priv, word32 privSz, curve25519_key * key, int endian)CURVE25519 秘密鍵のインポートのみ。（大きなエンディアン）。
int	**wc_curve25519_import_private_raw**(const byte * priv, word32 privSz, const byte * pub, word32 pubSz, curve25519_key * key) この関数は、パブリック秘密鍵ペアを Curve25519_Key 構造体にインポートします。ビッグエンディアンのみ。
int	**wc_curve25519_import_private_raw_ex**(const byte * priv, word32 privSz, const byte * pub, word32 pubSz, curve25519_key * key, int endian) この関数は、パブリック秘密鍵ペアを Curve25519_Key 構造体にインポートします。ビッグ・リトルエンディアンの両方をサポートします。

	Name
int	**wc_curve25519_export_private_raw**(curve25519_key * key, byte * out, word32 * outLen) この関数は Curve25519_Key 構造体から秘密鍵をエクスポートし、それを指定されたアウトバッファに格納します。また、エクスポートされたキーのサイズになるように概要を設定します。ビッグエンディアンのみ。
int	**wc_curve25519_export_private_raw_ex**(curve25519_key * key, byte * out, word32 * outLen, int endian) この関数は Curve25519_Key 構造体から秘密鍵をエクスポートし、それを指定されたアウトバッファに格納します。また、エクスポートされたキーのサイズになるように概要を設定します。それがビッグ・リトルエンディアンかを指定できます。
int	**wc_curve25519_import_public**(const byte * in, word32 inLen, curve25519_key * key) この関数は、指定されたバッファから公開鍵をインポートし、それを Curve25519_Key 構造体に格納します。
int	**wc_curve25519_import_public_ex**(const byte * in, word32 inLen, curve25519_key * key, int endian) この関数は、指定されたバッファから公開鍵をインポートし、それを Curve25519_Key 構造体に格納します。
int	**wc_curve25519_check_public**(const byte * pub, word32 pubSz, int endian) この関数は、公開鍵バッファが指定されたエンディアンに対して有効な Curve2519 キー値を保持していることを確認します。
int	**wc_curve25519_export_public**(curve25519_key * key, byte * out, word32 * outLen) この関数は指定されたキー構造から公開鍵をエクスポートし、結果をアウトバッファに格納します。ビッグエンディアンのみ。
int	**wc_curve25519_export_public_ex**(curve25519_key * key, byte * out, word32 * outLen, int endian) この関数は指定されたキー構造から公開鍵をエクスポートし、結果をアウトバッファに格納します。ビッグ・リトルエンディアンの両方をサポートします。
int	**wc_curve25519_export_key_raw**(curve25519_key * key, byte * priv, word32 * privSz, byte * pub, word32 * pubSz)Export Curve25519 キーペア。ビッグエンディアンのみ。
int	**wc_curve25519_export_key_raw_ex**(curve25519_key * key, byte * priv, word32 * privSz, byte * pub, word32 * pubSz, int endian)Export Curve25519 キーペア。ビッグ・リトルエンディアン。
int	**wc_curve25519_size**(curve25519_key * key) この関数は与えられたキー構造のキーサイズを返します。

B.20.2 Functions Documentation

B.20.2.1 function wc_curve25519_make_key

```
int wc_curve25519_make_key(
    WC_RNG * rng,
    int keysize,
    curve25519_key * key
)
```

この関数は、与えられたサイズ（Keysize）の指定された乱数発生器 RNG を使用して Curve25519 キーを生成し、それを指定された Curve25519_Key 構造体に格納します。キー構造が WC_CURVE25519_INIT（）を介して初期化された後に呼び出されるべきです。

Parameters:

- **RNG** ECC キーの生成に使用される RNG オブジェクトへのポインタ。
- **キーサイズ生成キーのサイズ。Curve25519 の 32 バイトでなければなりません。** *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
}
```

See: wc_curve25519_init

Return:

- 0 キーの生成に成功し、それを指定された Curve25519_Key 構造体に格納します。
- ECC_BAD_ARG_E 入力キーサイズが Curve25519 キー（32 バイト）のキーシェイズに対応していない場合は返されます。
- RNG_FAILURE_E RNG の内部ステータスが DRBG_OK でない場合、または RNG を使用して次のランダムブロックを生成する場合に返されます。
- BAD_FUNC_ARG 渡された入力パラメータのいずれかが NULL の場合に返されます。

B.20.2.2 function wc_curve25519_shared_secret

```
int wc_curve25519_shared_secret(
    curve25519_key * private_key,
    curve25519_key * public_key,
    byte * out,
    word32 * outlen
)
```

この関数は、秘密の秘密鍵と受信した公開鍵を考えると、共有秘密鍵を計算します。生成された秘密鍵をバッファアウトに保存し、ounlent の秘密鍵の変数を割り当てます。ビッグエンディアンのみをサポートします。

Parameters:

- **Private_Key** Curve25519_Key 構造体の秘密鍵で初期化されました。
- **public_key 受信した公開鍵を含む Curve25519_Key 構造体へのポインタ。**
- **32 バイト計算された秘密鍵を格納するバッファへのポインタ。** *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
}
```

See:

- wc_curve25519_init
- wc_curve25519_make_key
- wc_curve25519_shared_secret_ex

Return:

- 0 共有秘密鍵を正常に計算したときに返されました。
- BAD_FUNC_ARG 渡された入力パラメータのいずれかが NULL の場合に返されます。
- ECC_BAD_ARG_E 公開鍵の最初のビットが設定されている場合は、実装の指紋を避けるために返されます。

B.20.2.3 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
)
```

この関数は、秘密の秘密鍵と受信した公開鍵を考えると、共有秘密鍵を計算します。生成された秘密鍵をバッファアウトに保存し、ounlent の秘密鍵の変数を割り当てます。ビッグ・リトルエンディアンの両方をサポートします。

Parameters:

- **Private_Key** Curve25519_Key 構造体の秘密鍵で初期化されました。
- **public_key 受信した公開鍵を含む Curve25519_Key 構造体へのポインタ。**
- **32 バイト計算された秘密鍵を格納するバッファへのポインタ。**
- **pinout]** 出力バッファに書き込まれた長さを記憶するポインタの概要。*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
}
```

See:

- wc_curve25519_init
- wc_curve25519_make_key
- wc_curve25519_shared_secret

Return:

- 0 共有秘密鍵を正常に計算したときに返されました。
- BAD_FUNC_ARG 渡された入力パラメータのいずれかが NULL の場合に返されます。
- ECC_BAD_ARG_E 公開鍵の最初のビットが設定されている場合は、実装の指紋を避けるために返されます。

B.20.2.4 function wc_curve25519_init

```
int wc_curve25519_init(
    curve25519_key * key
)
```

この関数は Curve25519 キーを初期化します。構造のキーを生成する前に呼び出されるべきです。

See: wc_curve25519_make_key

Return:

- 0 Curve25519_Key 構造体の初期化に成功しました。
- BAD_FUNC_ARG キーが NULL のときに返されます。*Example*

```
curve25519_key key;
wc_curve25519_init(&key); // initialize key
// make key and proceed to encryption
```

B.20.2.5 function wc_curve25519_free

```
void wc_curve25519_free(
    curve25519_key * key
)
```

この関数は Curve25519 オブジェクトを解放します。*Example*

See:

- wc_curve25519_init
- wc_curve25519_make_key

```
curve25519_key privKey;
// initialize key, use it to generate shared secret key
wc_curve25519_free(&privKey);
```

B.20.2.6 function wc_curve25519_import_private

```
int wc_curve25519_import_private(
    const byte * priv,
    word32 privSz,
    curve25519_key * key
)
```

この関数は Curve25519 秘密鍵のみをインポートします。(ビッグエンディアン)。

Parameters:

- **インポートする秘密鍵を含むバッファへのポイント。**

- **インポートする秘密鍵の Privsz 長。** *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
}
```

See:

- wc_curve25519_import_private_ex
- wc_curve25519_size

Return:

- 0 秘密鍵のインポートに成功しました。
- BAD_FUNC_ARG キーまたは PRIV が NULL の場合は返します。
- ECC_BAD_ARG_E PRIVSZ が curve25519_KEY_SIZE と等しくない場合は返します。

B.20.2.7 function wc_curve25519_import_private_ex

```
int wc_curve25519_import_private_ex(
    const byte * priv,
    word32 privSz,
    curve25519_key * key,
    int endian
)
```

CURVE25519 秘密鍵のインポートのみ。(大きなエンディアン)。

Parameters:

- **インポートする秘密鍵を含むバッファへのポイント。**
- **インポートする秘密鍵の Privsz 長。**
- **インポートされたキーを保存する構造へのキーポインタ。** *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
}
```

See:

- wc_curve25519_import_private
- wc_curve25519_size

Return:

- 0 秘密鍵のインポートに成功しました。

- BAD_FUNC_ARG キーまたは PRIV が NULL の場合は返します。
- ECC_BAD_ARG_E PRIVSZ が curve25519_KEY_SIZE と等しくない場合は返します。

B.20.2.8 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
)
```

この関数は、パブリック秘密鍵ペアを Curve25519_Key 構造体にインポートします。ビッグエンディアンのみ。

Parameters:

- **インポートする秘密鍵を含むバッファへのポイント。**
- **インポートする秘密鍵の Privsz 長。**
- **パブリックキーをインポートするバッファへの Pub。**
- **インポートする公開鍵の Pubsz 長さ。** *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
}
```

See:

- wc_curve25519_init
- wc_curve25519_make_key
- wc_curve25519_import_public
- wc_curve25519_export_private_raw

Return:

- 0 Curve25519_Key 構造体へのインポートに返されます
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL の場合に返します。
- ECC_BAD_ARG_E 入力キーのキーサイズが Public キーサイズまたは秘密鍵サイズと一致しない場合に返されます。

B.20.2.9 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
)
```

この関数は、パブリック秘密鍵ペアを Curve25519_Key 構造体にインポートします。ビッグ・リトルエンディアンの両方をサポートします。

Parameters:

- **インポートする秘密鍵を含むバッファへのポイント。**
- **インポートする秘密鍵の Privsz 長。**
- **パブリックキーをインポートするバッファへの Pub。**
- **インポートする公開鍵の Pubsz 長さ。**
- **インポートされたキーを保存する構造へのキーポインタ。** *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
}
```

See:

- wc_curve25519_init
- wc_curve25519_make_key
- wc_curve25519_import_public
- wc_curve25519_export_private_raw
- wc_curve25519_import_private_raw

Return:

- 0 Curve25519_Key 構造体へのインポートに返されます
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL の場合に返します。
- ECC_BAD_ARG_E 戻された IF または入力キーのキーサイズがパブリックキーサイズまたは秘密鍵サイズと一致しない場合

B.20.2.10 function wc_curve25519_export_private_raw

```
int wc_curve25519_export_private_raw(
    curve25519_key * key,
    byte * out,
    word32 * outLen
)
```

この関数は Curve25519_Key 構造体から秘密鍵をエクスポートし、それを指定されたアウトバッファに格納します。また、エクスポートされたキーのサイズになるように概要を設定します。ビッグエンディアンのみ。

Parameters:

- **キーをエクスポートする構造へのキーポインタ。**
- **エクスポートされたキーを保存するバッファへのポインタ。** *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
}
```

See:

- wc_curve25519_init
- wc_curve25519_make_key
- wc_curve25519_import_private_raw
- wc_curve25519_export_private_raw_ex

Return:

- 0 Curve25519_Key 構造体から秘密鍵を正常にエクスポートしました。
- BAD_FUNC_ARG 入力パラメータが NULL の場合に返されます。
- ECC_BAD_ARG_E WC_CURVE25519_SIZE () がキーと等しくない場合に返されます。

B.20.2.11 function wc_curve25519_export_private_raw_ex

```
int wc_curve25519_export_private_raw_ex(
    curve25519_key * key,
    byte * out,
    word32 * outLen,
    int endian
)
```

この関数は Curve25519_Key 構造体から秘密鍵をエクスポートし、それを指定されたアウトバッファに格納します。また、エクスポートされたキーのサイズになるように概要を設定します。それがビッグ・リトルエンディアンかを指定できます。

Parameters:

- **キーをエクスポートする構造へのキーポインタ。**
- **エクスポートされたキーを保存するバッファへのポインタ。**
- **IN に照会は、バイト数のサイズです。ON** OUT では、出力バッファに書き込まれたバイトを保存します。*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
}
```

See:

- wc_curve25519_init
- wc_curve25519_make_key
- wc_curve25519_import_private_raw
- wc_curve25519_export_private_raw
- wc_curve25519_size

Return:

- 0 Curve25519_Key 構造体から秘密鍵を正常にエクスポートしました。
- BAD_FUNC_ARG 入力パラメータが NULL の場合に返されます。
- ECC_BAD_ARG_E WC_CURVE25519_SIZE () がキーと等しくない場合に返されます。

B.20.2.12 function wc_curve25519_import_public

```
int wc_curve25519_import_public(
    const byte * in,
    word32 inLen,
    curve25519_key * key
)
```

この関数は、指定されたバッファから公開鍵をインポートし、それを Curve25519_Key 構造体に格納します。

Parameters:

- **インポートする公開鍵を含むバッファへのポインタ。**
- **インポートする公開鍵のインレル長。** *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
}
```

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 公開鍵を Curve25519_Key 構造体に正常にインポートしました。
- ECC_BAD_ARG_E InLen パラメータがキー構造のキーサイズと一致しない場合に返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL の場合に返されます。

B.20.2.13 function wc_curve25519_import_public_ex

```
int wc_curve25519_import_public_ex(
    const byte * in,
    word32 inLen,
    curve25519_key * key,
    int endian
)
```

この関数は、指定されたバッファから公開鍵をインポートし、それを Curve25519_Key 構造体に格納します。

Parameters:

- **インポートする公開鍵を含むバッファへのポインタ。**
- **インポートする公開鍵のインレル長。**
- **キーを保存するカーブ 25519 キー構造へのキーポインタ。** *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
}
```

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 公開鍵を Curve25519_Key 構造体に正常にインポートしました。
- ECC_BAD_ARG_E InLen パラメータがキー構造のキーサイズと一致しない場合に返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL の場合に返されます。

B.20.2.14 function wc_curve25519_check_public

```
int wc_curve25519_check_public(
    const byte * pub,
    word32 pubSz,
    int endian
)
```

この関数は、公開鍵バッファが指定されたエンディアンに対して有効な Curve2519 キー値を保持していることを確認します。

Parameters:

- **チェックするための公開鍵を含むバッファへの Pub ポインタ。**

- **チェックするための公開鍵の長さを掲載します。** *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
}
```

See:

- wc_curve25519_init
- wc_curve25519_import_public
- wc_curve25519_import_public_ex
- wc_curve25519_size

Return:

- 0 公開鍵の値が有効なときに返されます。
- ECC_BAD_ARG_E 公開鍵の値が無効な場合は返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL の場合に返されます。

B.20.2.15 function wc_curve25519_export_public

```
int wc_curve25519_export_public(
    curve25519_key * key,
    byte * out,
    word32 * outLen
)
```

この関数は指定されたキー構造から公開鍵をエクスポートし、結果をアウトバッファに格納します。ビッグエンディアンのみ。

Parameters:

- **キーをエクスポートする Curve25519_Key 構造体へのキーポインタ。**
- **公開鍵を保存するバッファへのポインタ。** *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
}
```

See:

- wc_curve25519_init
- wc_curve25519_export_private_raw
- wc_curve25519_import_public

Return:

- 0 Curve25519_Key 構造体から公開鍵を正常にエクスポートする上で返されます。

- ECC_BAD_ARG_E outlen が curve25519_pub_key_size より小さい場合に返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL の場合に返されます。

B.20.2.16 function wc_curve25519_export_public_ex

```
int wc_curve25519_export_public_ex(
    curve25519_key * key,
    byte * out,
    word32 * outLen,
    int endian
)
```

この関数は指定されたキー構造から公開鍵をエクスポートし、結果をアウトバッファに格納します。ビッグ・リトルエンディアンの両方をサポートします。

Parameters:

- **キーをエクスポートする Curve25519_Key 構造体へのキーポインタ。**
- **公開鍵を保存するバッファへのポインタ。**
- **IN に照会は、バイト数のサイズです。ON** OUT では、出力バッファに書き込まれたバイトを保存します。*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
}
```

See:

- wc_curve25519_init
- wc_curve25519_export_private_raw
- wc_curve25519_import_public

Return:

- 0 Curve25519_Key 構造体から公開鍵を正常にエクスポートする上で返されます。
- ECC_BAD_ARG_E outlen が curve25519_pub_key_size より小さい場合に返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL の場合に返されます。

B.20.2.17 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 キーペア。ビッグエンディアンのみ。

Parameters:

- **キーペアをエクスポートする CURUN448_KEY 構造体へのキーポインタ。**
- **秘密鍵を保存するバッファへの PRIV ポインタ。**
- **PRIVSZ** ON IN は、PRIV バッファのサイズをバイト単位で）です。ON OUT は、PRIV バッファに書き込まれたバイトを保存します。
- **パブリックキーを保存するバッファへの Pub。** *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
}
```

See:

- wc_curve25519_export_key_raw_ex
- wc_curve25519_export_private_raw

Return:

- 0 Curve25519_Key 構造体からキーペアのエクスポートに成功しました。
- BAD_FUNC_ARG 入力パラメータが NULL の場合に返されます。
- ECC_BAD_ARG_E PRIVSZ が CURUV25519_SEY_SIZE または PUBSZ よりも小さい場合は、PUBSZ が CURUG25519_PUB_KEY_SIZE よりも小さい場合に返されます。

B.20.2.18 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 キーペア。ビッグ・リトルエンディアン。

Parameters:

- **キーペアをエクスポートする CURUN448_KEY 構造体へのキーポインタ。**
- **秘密鍵を保存するバッファへの PRIV ポインタ。**
- **PRIVSZ** ON IN は、PRIV バッファのサイズをバイト単位で）です。ON OUT は、PRIV バッファに書き込まれたバイトを保存します。
- **パブリックキーを保存するバッファへの Pub。**
- **PUBSZ** ON IN は、パブバッファのサイズをバイト単位で）です。ON OUT では、PUB バッファに書き込まれたバイトを保存します。*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
}
```

See:

- wc_curve25519_export_key_raw
- wc_curve25519_export_private_raw_ex
- wc_curve25519_export_public_ex

Return:

- 0 Curve25519_Key 構造体からキーペアのエクスポートに成功しました。
- BAD_FUNC_ARG 入力パラメータが NULL の場合に返されます。
- ECC_BAD_ARG_E PRIVSZ が CURUV25519_SEY_SIZE または PUBSZ よりも小さい場合は、PUBSZ が CURUG25519_PUB_KEY_SIZE よりも小さい場合に返されます。

B.20.2.19 function wc_curve25519_size

```
int wc_curve25519_size(
    curve25519_key * key
)
```

この関数は与えられたキー構造のキーサイズを返します。

See:

- wc_curve25519_init
- wc_curve25519_make_key

Return:

- Success 有効な初期化された Curve25519_Key 構造体を考慮すると、キーのサイズを返します。
- 0 キーが NULL の場合は返されます *Example*

```
int keySz;

curve25519_key key;
// initialize and make key

keySz = wc_curve25519_size(&key);
```

B.21 Algorithms - Curve448

B.21.1 Functions

	Name
int	**wc_curve448_make_key**(WC_RNG * rng, int keysize, curve448_key * key) この関数は、与えられたサイズ（Keysize）のサイズの指定された乱数発生器 RNG を使用して Curve448 キーを生成し、それを指定された Curve448_Key 構造体に格納します。キー構造が WC_CURVE448_INIT（）を介して初期化された後に呼び出されるべきです。
int	**wc_curve448_shared_secret**(curve448_key * private_key, curve448_key * public_key, byte * out, word32 * outlen) この関数は、秘密の秘密鍵と受信した公開鍵を考えると、共有秘密鍵を計算します。生成された秘密鍵をバッファアウトに保存し、ounlent の秘密鍵の変数を割り当てます。ビッグエンディアンのみをサポートします。
int	**wc_curve448_shared_secret_ex**(curve448_key * private_key, curve448_key * public_key, byte * out, word32 * outlen, int endian) この関数は、秘密の秘密鍵と受信した公開鍵を考えると、共有秘密鍵を計算します。生成された秘密鍵をバッファアウトに保存し、ounlent の秘密鍵の変数を割り当てます。ビッグ・リトルエンディアンの両方をサポートします。
int	**wc_curve448_init**(curve448_key * key) この関数は Curve448 キーを初期化します。構造のキーを生成する前に呼び出されるべきです。
void	**wc_curve448_free**(curve448_key * key) この関数は Curve448 オブジェクトを解放します。 *Example*
int	**wc_curve448_import_private**(const byte * priv, word32 privSz, curve448_key * key) この関数は Curve448 秘密鍵のみをインポートします。（ビッグエンディアン）。
int	**wc_curve448_import_private_ex**(const byte * priv, word32 privSz, curve448_key * key, int endian)CURVE448 秘密鍵のインポートのみ。（ビッグエンディアン）。
int	**wc_curve448_import_private_raw**(const byte * priv, word32 privSz, const byte * pub, word32 pubSz, curve448_key * key) この関数は、public-秘密鍵のペアを Curve448_Key 構造体にインポートします。ビッグエンディアンのみ。
int	**wc_curve448_import_private_raw_ex**(const byte * priv, word32 privSz, const byte * pub, word32 pubSz, curve448_key * key, int endian) この関数は、public-秘密鍵のペアを Curve448_Key 構造体にインポートします。ビッグ・リトルエンディアンの両方をサポートします。

	Name
int	**wc_curve448_export_private_raw**(curve448_key * key, byte * out, word32 * outLen) この関数は Curve448_Key 構造体から秘密鍵をエクスポートし、それを指定されたバッファに格納します。また、エクスポートされたキーのサイズになるように概要を設定します。ビッグエンディアンのみ。
int	**wc_curve448_export_private_raw_ex**(curve448_key * key, byte * out, word32 * outLen, int endian) この関数は Curve448_Key 構造体から秘密鍵をエクスポートし、それを指定されたバッファに格納します。また、エクスポートされたキーのサイズになるように概要を設定します。それが大きいかリトルエンディアンかを指定できます。
int	**wc_curve448_import_public**(const byte * in, word32 inLen, curve448_key * key) この関数は、指定されたバッファから公開鍵をインポートし、それを Curve448_Key 構造体に格納します。
int	**wc_curve448_import_public_ex**(const byte * in, word32 inLen, curve448_key * key, int endian) この関数は、指定されたバッファから公開鍵をインポートし、それを Curve448_Key 構造体に格納します。
int	**wc_curve448_check_public**(const byte * pub, word32 pubSz, int endian) この関数は、公開鍵バッファがエンディアン順序付けを与えられた有効な Curve448 キー値を保持することを確認します。
int	**wc_curve448_export_public**(curve448_key * key, byte * out, word32 * outLen) この関数は指定されたキー構造から公開鍵をエクスポートし、結果をアウトバッファに格納します。ビッグエンディアンのみ。
int	**wc_curve448_export_public_ex**(curve448_key * key, byte * out, word32 * outLen, int endian) この関数は指定されたキー構造から公開鍵をエクスポートし、結果をアウトバッファに格納します。ビッグ・リトルエンディアンの両方をサポートします。
int	**wc_curve448_export_key_raw**(curve448_key * key, byte * priv, word32 * privSz, byte * pub, word32 * pubSz) この関数は指定されたキー構造からキーペアをエクスポートし、結果をアウトバッファに格納します。ビッグエンディアンのみ。
int	**wc_curve448_export_key_raw_ex**(curve448_key * key, byte * priv, word32 * privSz, byte * pub, word32 * pubSz, int endian)Curve448 キーペアをエクスポートします。ビッグ、またはリトルエンディアン。
int	**wc_curve448_size**(curve448_key * key) この関数は与えられたキー構造のキーサイズを返します。

B.21.2 Functions Documentation

B.21.2.1 function wc_curve448_make_key

```
int wc_curve448_make_key(
    WC_RNG * rng,
    int keysize,
    curve448_key * key
)
```

この関数は、与えられたサイズ（Keysize）のサイズの指定された乱数発生器 RNG を使用して Curve448 キーを生成し、それを指定された Curve448_Key 構造体に格納します。キー構造が WC_CURVE448_INIT（）を介して初期化された後に呼び出されるべきです。

Parameters:

- **RNG** ECC キーの生成に使用される RNG オブジェクトへのポインタ。
- **キーサイズ生成キーのサイズ。Curve448 の場合は 56 バイトでなければなりません。** *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
}
```

See: wc_curve448_init

Return:

- 0 キーの生成に成功し、それを指定された Curve448_Key 構造体に格納します。
- ECC_BAD_ARG_E 入力キーサイズが Curve448 キー（56 バイト）のキーシェイズに対応していない場合は返されます。
- RNG_FAILURE_E RNG の内部ステータスが DRBG_OK でない場合、または RNG を使用して次のランダムブロックを生成する場合に返されます。
- BAD_FUNC_ARG 渡された入力パラメータのいずれかが NULL の場合に返されます。

B.21.2.2 function wc_curve448_shared_secret

```
int wc_curve448_shared_secret(
    curve448_key * private_key,
    curve448_key * public_key,
    byte * out,
    word32 * outlen
)
```

この関数は、秘密の秘密鍵と受信した公開鍵を考えると、共有秘密鍵を計算します。生成された秘密鍵をバッファアウトに保存し、ounlent の秘密鍵の変数を割り当てます。ビッグエンディアンのみをサポートします。

Parameters:

- **Private_Key** Curve448_Key 構造体へのポインタユーザーの秘密鍵で初期化されました。
- **public_key 受信した公開鍵を含む Curve448_Key 構造体へのポインタ。**
- **56 バイトの計算された秘密鍵を保存するバッファへのポインタ。** *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
}
```

See:

- wc_curve448_init
- wc_curve448_make_key
- wc_curve448_shared_secret_ex

Return:

- 0 共有秘密鍵を正常に計算する上で返却されました
- BAD_FUNC_ARG 渡された入力パラメーターのいずれかが NULL の場合に返されます

B.21.2.3 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
)
```

この関数は、秘密の秘密鍵と受信した公開鍵を考えると、共有秘密鍵を計算します。生成された秘密鍵をバッファアウトに保存し、ounlent の秘密鍵の変数を割り当てます。ビッグ・リトルエンディアンの両方をサポートします。

Parameters:

- **Private_Key** Curve448_Key 構造体へのポインタユーザーの秘密鍵で初期化されました。
- **public_key 受信した公開鍵を含む Curve448_Key 構造体へのポインタ。**
- **56 バイトの計算された秘密鍵を保存するバッファへのポインタ。**
- **出力バッファに書き込まれた長さを記憶するポインタの概要。** *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
}
```

See:

- wc_curve448_init
- wc_curve448_make_key
- wc_curve448_shared_secret

Return:

- 0 共有秘密鍵を正常に計算したときに返されました。
- BAD_FUNC_ARG 渡された入力パラメータのいずれかが NULL の場合に返されます。

B.21.2.4 function wc_curve448_init

```
int wc_curve448_init(
    curve448_key * key
)
```

この関数は Curve448 キーを初期化します。構造のキーを生成する前に呼び出されるべきです。

See: wc_curve448_make_key

Return:

- 0 Curve448_Key 構造体の初期化に成功しました。
- BAD_FUNC_ARG キーが NULL のときに返されます。*Example*

```
curve448_key key;
wc_curve448_init(&key); // initialize key
// make key and proceed to encryption
```

B.21.2.5 function wc_curve448_free

```
void wc_curve448_free(
    curve448_key * key
)
```

この関数は Curve448 オブジェクトを解放します。*Example*

See:

- wc_curve448_init
- wc_curve448_make_key

```
curve448_key privKey;
// initialize key, use it to generate shared secret key
wc_curve448_free(&privKey);
```

B.21.2.6 function wc_curve448_import_private

```
int wc_curve448_import_private(
    const byte * priv,
    word32 privSz,
    curve448_key * key
)
```

この関数は Curve448 秘密鍵のみをインポートします。(ビッグエンディアン)。

Parameters:

- **インポートする秘密鍵を含むバッファへのポイント。**
- **インポートする秘密鍵の Privsz 長。** *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
}
```

See:

- wc_curve448_import_private_ex
- wc_curve448_size

Return:

- 0 秘密鍵のインポートに成功しました。
- BAD_FUNC_ARG キーまたは PRIV が NULL の場合は返します。
- ECC_BAD_ARG_E PRIVSZ が CURUG448_KEY_SIZE と等しくない場合は返します。

B.21.2.7 function wc_curve448_import_private_ex

```
int wc_curve448_import_private_ex(
    const byte * priv,
    word32 privSz,
    curve448_key * key,
    int endian
)
```

CURVE448 秘密鍵のインポートのみ。(ビッグエンディアン)。

Parameters:

- **インポートする秘密鍵を含むバッファへのポイント。**
- **インポートする秘密鍵の Privsz 長。**
- **インポートされたキーを保存する構造へのキーポインタ。** *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
}
```

See:

- wc_curve448_import_private
- wc_curve448_size

Return:

- 0 秘密鍵のインポートに成功しました。
- BAD_FUNC_ARG キーまたは PRIV が NULL の場合は返します。

- ECC_BAD_ARG_E PRIVSZ が CURUG448_KEY_SIZE と等しくない場合は返します。

B.21.2.8 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
)
```

この関数は、public-秘密鍵のペアを Curve448_Key 構造体にインポートします。ビッグエンディアンのみ。

Parameters:

- **インポートする秘密鍵を含むバッファへのポイント。**
- **インポートする秘密鍵の Privsz 長。**
- **パブリックキーをインポートするバッファへの Pub。**
- **インポートする公開鍵の Pubsz 長さ。** *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
}
```

See:

- wc_curve448_init
- wc_curve448_make_key
- wc_curve448_import_public
- wc_curve448_export_private_raw

Return:

- 0 Curve448_Key 構造体へのインポート時に返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL の場合に返します。
- ECC_BAD_ARG_E 入力キーのキーサイズが Public キーサイズまたは秘密鍵サイズと一致しない場合に返されます。

B.21.2.9 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
)
```

この関数は、public-秘密鍵のペアを Curve448_Key 構造体にインポートします。ビッグ・リトルエンディアンの両方をサポートします。

Parameters:

- **インポートする秘密鍵を含むバッファへのポイント。**
- **インポートする秘密鍵の Privsz 長。**
- **パブリックキーをインポートするバッファへの Pub。**
- **インポートする公開鍵の Pubsz 長さ。**
- **インポートされたキーを保存する構造へのキーポインタ。** *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
}
```

See:

- wc_curve448_init
- wc_curve448_make_key
- wc_curve448_import_public
- wc_curve448_export_private_raw
- wc_curve448_import_private_raw

Return:

- 0 Curve448_Key 構造体へのインポート時に返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL の場合に返します。
- ECC_BAD_ARG_E 入力キーのキーサイズが Public キーサイズまたは秘密鍵サイズと一致しない場合に返されます。

B.21.2.10 function wc_curve448_export_private_raw

```
int wc_curve448_export_private_raw(
    curve448_key * key,
    byte * out,
    word32 * outLen
)
```

この関数は Curve448_Key 構造体から秘密鍵をエクスポートし、それを指定されたバッファに格納します。また、エクスポートされたキーのサイズになるように概要を設定します。ビッグエンディアンのみ。

Parameters:

- **キーをエクスポートする構造へのキーポインタ。**

- **エクスポートされたキーを保存するバッファへのポインタ。** *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
}
```

See:

- wc_curve448_init
- wc_curve448_make_key
- wc_curve448_import_private_raw
- wc_curve448_export_private_raw_ex

Return:

- 0 Curve448_Key 構造体から秘密鍵を正常にエクスポートする上で返されました。
- BAD_FUNC_ARG 入力パラメータが NULL の場合に返されます。
- ECC_BAD_ARG_E WC_CURVE448_SIZE () がキーと等しくない場合に返されます。

B.21.2.11 function wc_curve448_export_private_raw_ex

```
int wc_curve448_export_private_raw_ex(
    curve448_key * key,
    byte * out,
    word32 * outLen,
    int endian
)
```

この関数は Curve448_Key 構造体から秘密鍵をエクスポートし、それを指定されたバッファに格納します。また、エクスポートされたキーのサイズになるように概要を設定します。それが大きいかリトルエンディアンかを指定できます。

Parameters:

- **キーをエクスポートする構造へのキーポインタ。**
- **エクスポートされたキーを保存するバッファへのポインタ。**
- **IN に照会は、バイト数のサイズです。ON** OUT では、出力バッファに書き込まれたバイトを保存します。*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
}
```

See:

- wc_curve448_init
- wc_curve448_make_key
- wc_curve448_import_private_raw
- wc_curve448_export_private_raw
- wc_curve448_size

Return:

- 0 Curve448_Key 構造体から秘密鍵を正常にエクスポートする上で返されました。
- BAD_FUNC_ARG 入力パラメータが NULL の場合に返されます。
- ECC_BAD_ARG_E WC_CURVE448_SIZE () がキーと等しくない場合に返されます。

B.21.2.12 function wc_curve448_import_public

```
int wc_curve448_import_public(
    const byte * in,
    word32 inLen,
    curve448_key * key
)
```

この関数は、指定されたバッファから公開鍵をインポートし、それを Curve448_Key 構造体に格納します。

Parameters:

- **インポートする公開鍵を含むバッファへのポインタ。**
- **インポートする公開鍵のインレル長。** *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
}
```

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 公開鍵を Curve448_Key 構造体に正常にインポートしました。
- ECC_BAD_ARG_E InLen パラメータがキー構造のキーサイズと一致しない場合に返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL の場合に返されます。

B.21.2.13 function wc_curve448_import_public_ex

```
int wc_curve448_import_public_ex(
```

```
    const byte * in,
    word32 inLen,
    curve448_key * key,
    int endian
)
```

この関数は、指定されたバッファから公開鍵をインポートし、それを Curve448_Key 構造体に格納します。

Parameters:

- **インポートする公開鍵を含むバッファへのポインタ。**
- **インポートする公開鍵のインレル長。**
- **キーを保存する Curve448_Key 構造体へのキーポインタ。** *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
}
```

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 公開鍵を Curve448_Key 構造体に正常にインポートしました。
- ECC_BAD_ARG_E InLen パラメータがキー構造のキーサイズと一致しない場合に返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL の場合に返されます。

B.21.2.14 function wc_curve448_check_public

```
int wc_curve448_check_public(
    const byte * pub,
    word32 pubSz,
    int endian
)
```

この関数は、公開鍵バッファがエンディアン順序付けを与えられた有効な Curve448 キー値を保持することを確認します。

Parameters:

- **チェックするための公開鍵を含むバッファへの Pub ポインタ。**
- **チェックするための公開鍵の長さを掲載します。** *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
}
```

See:

- wc_curve448_init
- wc_curve448_import_public
- wc_curve448_import_public_ex
- wc_curve448_size

Return:

- 0 公開鍵の値が有効なときに返されます。
- ECC_BAD_ARG_E 公開鍵の値が無効な場合は返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL の場合に返されます。

B.21.2.15 function wc_curve448_export_public

```
int wc_curve448_export_public(
    curve448_key * key,
    byte * out,
    word32 * outLen
)
```

この関数は指定されたキー構造から公開鍵をエクスポートし、結果をアウトバッファに格納します。ビッグエンディアンのみ。

Parameters:

- **キーをエクスポートする Curve448_Key 構造体へのキーポインタ。**
- **公開鍵を保存するバッファへのポインタ。** *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
}
```

See:

- wc_curve448_init
- wc_curve448_export_private_raw
- wc_curve448_import_public

Return:

- 0 Curve448_Key 構造体から公開鍵のエクスポートに成功しました。
- ECC_BAD_ARG_E outlen が curve448_pub_key_size より小さい場合に返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL の場合に返されます。

B.21.2.16 function wc_curve448_export_public_ex

```
int wc_curve448_export_public_ex(
    curve448_key * key,
    byte * out,
    word32 * outLen,
    int endian
)
```

この関数は指定されたキー構造から公開鍵をエクスポートし、結果をアウトバッファに格納します。ビッグ・リトルエンディアンの両方をサポートします。

Parameters:

- **キーをエクスポートする Curve448_Key 構造体へのキーポインタ。**
- **公開鍵を保存するバッファへのポインタ。**
- **IN に照会は、バイト数のサイズです。ON** OUT では、出力バッファに書き込まれたバイトを保存します。*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
}
```

See:

- wc_curve448_init
- wc_curve448_export_private_raw
- wc_curve448_import_public

Return:

- 0 Curve448_Key 構造体から公開鍵のエクスポートに成功しました。
- ECC_BAD_ARG_E outlen が curve448_pub_key_size より小さい場合に返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL の場合に返されます。

B.21.2.17 function wc_curve448_export_key_raw

```
int wc_curve448_export_key_raw(
    curve448_key * key,
    byte * priv,
    word32 * privSz,
    byte * pub,
    word32 * pubSz
)
```

この関数は指定されたキー構造からキーペアをエクスポートし、結果をアウトバッファに格納します。ビッグエンディアンのみ。

Parameters:

- **キーペアをエクスポートする CURUN448_KEY 構造体へのキーポインタ。**

- **秘密鍵を保存するバッファへの PRIV ポインタ。**
- **PRIVSZ** ON IN は、PRIV バッファのサイズをバイト単位で）です。ON OUT は、PRIV バッファに書き込まれたバイトを保存します。
- **パブリックキーを保存するバッファへの Pub。** *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
}
```

See:

- wc_curve448_export_key_raw_ex
- wc_curve448_export_private_raw

Return:

- 0 Curve448_Key 構造体からキーペアのエクスポートに成功しました。
- BAD_FUNC_ARG 入力パラメータが NULL の場合に返されます。
- ECC_BAD_ARG_E PRIVSZ が CURUV448_KEY_SIZE または PUBSZ よりも小さい場合は､Curge448_PUB_KEY_SIZE よりも小さい場合に返されます。

B.21.2.18 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
)
```

Curve448 キーペアをエクスポートします。ビッグ、またはリトルエンディアン。

Parameters:

- **キーペアをエクスポートする CURUN448_KEY 構造体へのキーポインタ。**
- **秘密鍵を保存するバッファへの PRIV ポインタ。**
- **PRIVSZ** ON IN は、PRIV バッファのサイズをバイト単位で）です。ON OUT は、PRIV バッファに書き込まれたバイトを保存します。
- **パブリックキーを保存するバッファへの Pub。**
- **PUBSZ** ON IN は、パブバッファのサイズをバイト単位で）です。ON OUT では、PUB バッファに書き込まれたバイトを保存します。*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
}
```

See:

- wc_curve448_export_key_raw
- wc_curve448_export_private_raw_ex
- wc_curve448_export_public_ex

Return:

- 0 成功
- BAD_FUNC_ARG 入力パラメータが NULL の場合に返されます。
- ECC_BAD_ARG_E PRIVSZ が CURUV448_KEY_SIZE または PUBSZ よりも小さい場合は、Curge448_PUB_KEY_SIZE よりも小さい場合に返されます。

この関数は指定されたキー構造からキーペアをエクスポートし、結果をアウトバッファに格納します。ビッグ、またはリトルエンディアン。

B.21.2.19 function wc_curve448_size

```
int wc_curve448_size(
    curve448_key * key
)
```

この関数は与えられたキー構造のキーサイズを返します。

See:

- wc_curve448_init
- wc_curve448_make_key

Return:

- Success 有効な初期化された Curve448_Key 構造体を考慮すると、キーのサイズを返します。
- 0 キーが NULL の場合は返されます。*Example*

```
int keySz;

curve448_key key;
// initialize and make key

keySz = wc_curve448_size(&key);
```

B.22 Algorithms - DSA

B.22.1 Functions

	Name
int	**wc_InitDsaKey**(DsaKey * key) この関数は、デジタル署名アルゴリズム（DSA）を介した認証に使用するために DSAKEY オブジェクトを初期化します。
void	**wc_FreeDsaKey**(DsaKey * key) この関数は、使用された後に dsakey オブジェクトを解放します。
int	**wc_DsaSign**(const byte * digest, byte * out, DsaKey * key, WC_RNG * rng) この機能は入力ダイジェストに署名し、結果を出力バッファーに格納します。
int	**wc_DsaVerify**(const byte * digest, const byte * sig, DsaKey * key, int * answer) この関数は、秘密鍵を考えると、ダイジェストの署名を検証します。回答パラメータでキーが正しく検証されているかどうか、正常な検証に対応する 1、および失敗した検証に対応する 0 が格納されます。
int	**wc_DsaPublicKeyDecode**(const byte * input, word32 * inOutIdx, DsaKey * key, word32 inSz) この機能は、DSA 公開鍵を含む DER フォーマットの証明書バッファを復号し、与えられた DSakey 構造体にキーを格納します。また、入力読み取りの長さに応じて INOUTIDX パラメータを設定します。
int	**wc_DsaPrivateKeyDecode**(const byte * input, word32 * inOutIdx, DsaKey * key, word32 inSz) この機能は、DSA 秘密鍵を含む DER フォーマットの証明書バッファをデコードし、指定された DSakey 構造体にキーを格納します。また、入力読み取りの長さに応じて INOUTIDX パラメータを設定します。
int	**wc_DsaKeyToDer**(DsaKey * key, byte * output, word32 inLen)DSAKEY キーを DER フォーマット、出力への書き込み（Inlen）、書き込まれたバイトを返します。
int	**wc_MakeDsaKey**(WC_RNG * rng, DsaKey * dsa)DSA キーを作成します。
int	**wc_MakeDsaParameters**(WC_RNG * rng, int modulus_size, DsaKey * dsa)FIPS 186-4 は、modulus_size 値の有効な値を定義します（1024,160）（2048,256）（3072,256）

B.22.2 Functions Documentation

B.22.2.1 function wc_InitDsaKey

```
int wc_InitDsaKey(
    DsaKey * key
)
```

この関数は、デジタル署名アルゴリズム（DSA）を介した認証に使用するために DSAKEY オブジェクトを初期化します。

See: wc_FreeDsaKey

Return:

- 0 成功に戻りました。
- BAD_FUNC_ARG NULL キーが渡された場合に返されます。*Example*

```
DsaKey key;
int ret;
ret = wc_InitDsaKey(&key); // initialize DSA key
```

B.22.2.2 function wc_FreeDsaKey

```
void wc_FreeDsaKey(
    DsaKey * key
)
```

この関数は、使用された後に dsakey オブジェクトを解放します。

See: wc_FreeDsaKey

Return: none いいえ返します。*Example*

```
DsaKey key;
// initialize key, use for authentication
...
wc_FreeDsaKey(&key); // free DSA key
```

B.22.2.3 function wc_DsaSign

```
int wc_DsaSign(
    const byte * digest,
    byte * out,
    DsaKey * key,
    WC_RNG * rng
)
```

この機能は入力ダイジェストに署名し、結果を出力バッファーに格納します。

Parameters:

- **digest** 署名するハッシュへのポインタ
- **out** 署名を保存するバッファへのポインタ
- **key** 署名を生成するための初期化された Dsakey 構造へのポインタ *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
}
```

See: wc_DsaVerify

Return:

- 0 入力ダイジェストに正常に署名したときに返されました

- MP_INIT_E DSA 署名の処理にエラーがある場合は返される可能性があります。
- MP_READ_E DSA 署名の処理にエラーがある場合は返される可能性があります。
- MP_CMP_E DSA 署名の処理にエラーがある場合は返される可能性があります。
- MP_INVMOD_E DSA 署名の処理にエラーがある場合は返される可能性があります。
- MP_EXPTMOD_E DSA 署名の処理にエラーがある場合は返される可能性があります。
- MP_MOD_E DSA 署名の処理にエラーがある場合は返される可能性があります。
- MP_MUL_E DSA 署名の処理にエラーがある場合は返される可能性があります。
- MP_ADD_E DSA 署名の処理にエラーがある場合は返される可能性があります。
- MP_MULMOD_E DSA 署名の処理にエラーがある場合は返される可能性があります。
- MP_TO_E DSA 署名の処理にエラーがある場合は返される可能性があります。
- MP_MEM DSA 署名の処理にエラーがある場合は返される可能性があります。

B.22.2.4 function wc_DsaVerify

```
int wc_DsaVerify(
    const byte * digest,
    const byte * sig,
    DsaKey * key,
    int * answer
)
```

この関数は、秘密鍵を考えると、ダイジェストの署名を検証します。回答パラメータでキーが正しく検証されているかどうか、正常な検証に対応する 1、および失敗した検証に対応する 0 が格納されます。

Parameters:

- **digest** 署名の主題を含むダイジェストへのポインタ
- **sig** 確認する署名を含むバッファへのポインタ
- **key** 署名を検証するための初期化された Dsakey 構造へのポインタ *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
}
```

See: wc_DsaSign

Return:

- 0 検証要求の処理に成功したときに返されます。注：これは、署名が検証されていることを意味するわけではなく、関数が成功したというだけです。
- MP_INIT_E DSA 署名の処理にエラーがある場合は返される可能性があります。
- MP_READ_E DSA 署名の処理にエラーがある場合は返される可能性があります。
- MP_CMP_E DSA 署名の処理にエラーがある場合は返される可能性があります。
- MP_INVMOD_E DSA 署名の処理にエラーがある場合は返される可能性があります。
- MP_EXPTMOD_E DSA 署名の処理にエラーがある場合は返される可能性があります。
- MP_MOD_E DSA 署名の処理にエラーがある場合は返される可能性があります。
- MP_MUL_E DSA 署名の処理にエラーがある場合は返される可能性があります。
- MP_ADD_E DSA 署名の処理にエラーがある場合は返される可能性があります。

- MP_MULMOD_E DSA 署名の処理にエラーがある場合は返される可能性があります。
- MP_TO_E DSA 署名の処理にエラーがある場合は返される可能性があります。
- MP_MEM DSA 署名の処理にエラーがある場合は返される可能性があります。

B.22.2.5 function wc_DsaPublicKeyDecode

```
int wc_DsaPublicKeyDecode(
    const byte * input,
    word32 * inOutIdx,
    DsaKey * key,
    word32 inSz
)
```

この機能は、DSA 公開鍵を含む DER フォーマットの証明書バッファを復号し、与えられた DSakey 構造体にキーを格納します。また、入力読み取りの長さに応じて INOUTIDX パラメータを設定します。

Parameters:

- **input** DER フォーマット DSA 公開鍵を含むバッファへのポインタ
- **inOutIdx** 証明書の最後のインデックスを保存する整数へのポインタ
- **key** 公開鍵を保存する Dsakey 構造へのポインタ *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
}
```

See:

- wc_InitDsaKey
- wc_DsaPrivateKeyDecode

Return:

- 0 dsakey オブジェクトの公開鍵を正常に設定する
- ASN_PARSE_E 証明書バッファを読みながらエンコーディングにエラーがある場合
- ASN_DH_KEY_E DSA パラメータの 1 つが誤ってフォーマットされている場合に返されます

B.22.2.6 function wc_DsaPrivateKeyDecode

```
int wc_DsaPrivateKeyDecode(
    const byte * input,
    word32 * inOutIdx,
    DsaKey * key,
    word32 inSz
)
```

この機能は、DSA 秘密鍵を含む DER フォーマットの証明書バッファをデコードし、指定された DSakey 構造体にキーを格納します。また、入力読み取りの長さに応じて INOUTIDX パラメータを設定します。

Parameters:

- **input** DER フォーマット DSA 秘密鍵を含むバッファへのポインタ
- **inOutIdx** 証明書の最後のインデックスを保存する整数へのポインタ
- **key** 秘密鍵を保存する DSakey 構造へのポインタ *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
}
```

See:

- wc_InitDsaKey
- wc_DsaPublicKeyDecode

Return:

- 0 dsakey オブジェクトの秘密鍵を正常に設定するに返されました
- ASN_PARSE_E 証明書バッファを読みながらエンコーディングにエラーがある場合
- ASN_DH_KEY_E DSA パラメータの 1 つが誤ってフォーマットされている場合に返されます

B.22.2.7 function wc_DsaKeyToDer

```
int wc_DsaKeyToDer(
    DsaKey * key,
    byte * output,
    word32 inLen
)
```

DSAKEY キーを DER フォーマット、出力への書き込み（Inlen）、書き込まれたバイトを返します。

Parameters:

- **key** 変換する dsakey 構造へのポインタ。
- **output** 変換キーの出力バッファへのポインタ。*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);
```

See:

- wc_InitDsaKey
- wc_FreeDsaKey
- wc_MakeDsaKey

Return:

- outLen 成功、書かれたバイト数
- BAD_FUNC_ARG キーまたは出力は NULL またはキー - > タイプが DSA_PRIVATE ではありません。
- MEMORY_E メモリの割り当て中にエラーが発生しました。

B.22.2.8 function wc_MakeDsaKey

```
int wc_MakeDsaKey(
    WC_RNG * rng,
    DsaKey * dsa
)
```

DSA キーを作成します。

Parameters:

- **rng** WC_RNG 構造体へのポインタ。*Example*

```
WC_RNG rng;
DsaKey dsa;
wc_InitRng(&rng);
wc_InitDsa(&dsa);
if(wc_MakeDsaKey(&rng, &dsa) != 0)
{
    // Error creating key
}
```

See:

- wc_InitDsaKey
- wc_FreeDsaKey
- wc_DsaSign

Return:

- MP_OKAY 成功
- BAD_FUNC_ARG RNG または DSA のどちらかが null です。
- MEMORY_E バッファにメモリを割り当てることができませんでした。
- MP_INIT_E MP_INT の初期化エラー

B.22.2.9 function wc_MakeDsaParameters

```
int wc_MakeDsaParameters(
    WC_RNG * rng,
    int modulus_size,
    DsaKey * dsa
)
```

FIPS 186-4 は、modulus_size 値の有効な値を定義します（1024,160）（2048,256）（3072,256）

Parameters:

- **rng** WolfCrypt RNG へのポインタ。
- **modulus_size** 1024,2048、または 3072 は有効な値です。*Example*

```
DsaKey key;
WC_RNG rng;
wc_InitDsaKey(&key);
wc_InitRng(&rng);
if(wc_MakeDsaParameters(&rng, 1024, &genKey) != 0)
{
    // Handle error
}
```

See:

- wc_MakeDsaKey

- wc_DsaKeyToDer
- wc_InitDsaKey

Return:

- 0 成功
- BAD_FUNC_ARG RNG または DSA は NULL または MODULUS_SIZE が無効です。
- MEMORY_E メモリを割り当てようとするエラーが発生しました。

B.23 Algorithms - Diffie-Hellman

B.23.1 Functions

	Name
int	**wc_InitDhKey**(DhKey * key) この関数は、Diffie-Hellman Exchange プロトコルを使用して安全な秘密鍵を交渉するのに使用するための Diffie-Hellman キーを初期化します。
void	**wc_FreeDhKey**(DhKey * key) この関数は、Diffie-Hellman Exchange プロトコルを使用して安全な秘密鍵をネゴシエートするために使用された後に Diffie-Hellman キーを解放します。
int	**wc_DhGenerateKeyPair**(DhKey * key, WC_RNG * rng, byte * priv, word32 * privSz, byte * pub, word32 * pubSz) この関数は diffie-hellman パブリックパラメータに基づいてパブリック/秘密鍵ペアを生成し、PRIVS の秘密鍵と Pub の公開鍵を格納します。初期化された Diffie-Hellman キーと初期化された RNG 構造を取ります。
int	**wc_DhAgree**(DhKey * key, byte * agree, word32 * agreeSz, const byte * priv, word32 privSz, const byte * otherPub, word32 pubSz) この関数は、ローカル秘密鍵と受信した公開鍵に基づいて合意された秘密鍵を生成します。交換の両側で完了した場合、この関数は対称通信のための秘密鍵の合意を生成します。共有秘密鍵の生成に成功すると、書かれた秘密鍵のサイズは仲間に保存されます。
int	**wc_DhKeyDecode**(const byte * input, word32 * inOutIdx, DhKey * key, word32) この機能は、DER フォーマットのキーを含む与えられた入力バッファから Diffie-Hellman キーをデコードします。結果を DHKEY 構造体に格納します。
int	**wc_DhSetKey**(DhKey * key, const byte * p, word32 pSz, const byte * g, word32 gSz) この関数は、入力秘密鍵パラメータを使用して DHKEY 構造体のキーを設定します。WC_DHKEYDECODE とは異なり、この関数は入力キーが DER フォーマットでフォーマットされ、代わりに PARSED 入力パラメータ P（Prime）と G（Base）を受け入れる必要はありません。

	Name
int	**wc_DhParamsLoad**(const byte * input, word32 inSz, byte * p, word32 * pInOutSz, byte * g, word32 * gInOutSz) この関数は、与えられた入力バッファから Diffie-Hellman パラメータ P（Prime）と G（ベース）をフォーマットされています。
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)

B.23.2 Functions Documentation

B.23.2.1 function wc_InitDhKey

```
int wc_InitDhKey(
    DhKey * key
)
```

この関数は、Diffie-Hellman Exchange プロトコルを使用して安全な秘密鍵を交渉するのに使用するための Diffie-Hellman キーを初期化します。

See:

- wc_FreeDhKey
- wc_DhGenerateKeyPair

Return: none いいえ返します。*Example*

```
DhKey key;
wc_InitDhKey(&key); // initialize DH key
```

B.23.2.2 function wc_FreeDhKey

```
void wc_FreeDhKey(
    DhKey * key
)
```

この関数は、Diffie-Hellman Exchange プロトコルを使用して安全な秘密鍵をネゴシエートするために使用された後に Diffie-Hellman キーを解放します。

See: wc_InitDhKey

Return: none いいえ返します。*Example*

```
DhKey key;
// initialize key, perform key exchange

wc_FreeDhKey(&key); // free DH key to avoid memory leaks
```

B.23.2.3 function wc_DhGenerateKeyPair

```
int wc_DhGenerateKeyPair(
    DhKey * key,
    WC_RNG * rng,
    byte * priv,
    word32 * privSz,
    byte * pub,
    word32 * pubSz
)
```

この関数は diffie-hellman パブリックパラメータに基づいてパブリック/秘密鍵ペアを生成し、PRIVS の秘密鍵と Pub の公開鍵を格納します。初期化された Diffie-Hellman キーと初期化された RNG 構造を取ります。

Parameters:

- **key** キーペアを生成する DHKEY 構造体へのポインタ
- **rng** キーを生成するための初期化された乱数発生器（RNG）へのポインタ
- **priv** 秘密鍵を保存するバッファへのポインタ
- **privSz** PRIV に書かれた秘密鍵のサイズを保存します
- **pub** 公開鍵を保存するバッファへのポインタ *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);
```

See:

- wc_InitDhKey
- wc_DhSetKey
- wc_DhKeyDecode

Return:

- BAD_FUNC_ARG この関数への入力の 1 つを解析するエラーがある場合に返されます
- RNG_FAILURE_E RNG を使用して乱数を生成するエラーが発生した場合
- MP_INIT_E 公開鍵の生成中に数学ライブラリにエラーがある場合は返却される可能性があります
- MP_READ_E 公開鍵の生成中に数学ライブラリにエラーがある場合は返却される可能性があります
- MP_EXPTMOD_E 公開鍵の生成中に数学ライブラリにエラーがある場合は返却される可能性があります
- MP_TO_E 公開鍵の生成中に数学ライブラリにエラーがある場合は返却される可能性があります

B.23.2.4 function wc_DhAgree

```
int wc_DhAgree(
    DhKey * key,
    byte * agree,
    word32 * agreeSz,
    const byte * priv,
    word32 privSz,
```

```
    const byte * otherPub,
    word32 pubSz
)
```

この関数は、ローカル秘密鍵と受信した公開鍵に基づいて合意された秘密鍵を生成します。交換の両側で完了した場合、この関数は対称通信のための秘密鍵の合意を生成します。共有秘密鍵の生成に成功すると、書かれた秘密鍵のサイズは仲間に保存されます。

Parameters:

- **key** 共有キーを計算するために使用する DHKEY 構造体へのポインタ
- **agree** 秘密キーを保存するバッファへのポインタ
- **agreeSz** 成功した後に秘密鍵のサイズを保持します
- **priv** ローカル秘密鍵を含むバッファへのポインタ
- **privSz** 地元の秘密鍵のサイズ
- **otherPub** 受信した公開鍵を含むバッファへのポインタ *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
}
```

See: wc_DhGenerateKeyPair

Return:

- 0 合意された秘密鍵の生成に成功しました
- MP_INIT_E 共有秘密鍵の生成中にエラーが発生した場合に返却される可能性があります
- MP_READ_E 共有秘密鍵の生成中にエラーが発生した場合に返却される可能性があります
- MP_EXPTMOD_E 共有秘密鍵の生成中にエラーが発生した場合に返却される可能性があります
- MP_TO_E 共有秘密鍵の生成中にエラーが発生した場合に返却される可能性があります

B.23.2.5 function wc_DhKeyDecode

```
int wc_DhKeyDecode(
    const byte * input,
    word32 * inOutIdx,
    DhKey * key,
    word32
)
```

この機能は、DER フォーマットのキーを含む与えられた入力バッファから Diffie-Hellman キーをデコードします。結果を DHKEY 構造体に格納します。

Parameters:

- **input** der フォーマットされた diffie-hellman キーを含むバッファへのポインタ
- **inOutIdx** キーをデコードしている間に解析されたインデックスを保存する整数へのポインタ
- **key** 入力キーで初期化するための DHKEY 構造体へのポインタ *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
}
```

See: wc_DhSetKey

Return:

- 0 入力キーの復号に成功したときに返されます
- ASN_PARSE_E 入力のシーケンスを解析したエラーがある場合に返されます
- ASN_DH_KEY_E 解析された入力から秘密鍵パラメータを読み取るエラーがある場合

B.23.2.6 function wc_DhSetKey

```
int wc_DhSetKey(
    DhKey * key,
    const byte * p,
    word32 pSz,
    const byte * g,
    word32 gSz
)
```

この関数は、入力秘密鍵パラメータを使用して DHKEY 構造体のキーを設定します。WC_DHKEYDECODE とは異なり、この関数は入力キーが DER フォーマットでフォーマットされ、代わりに PARSED 入力パラメータ P（Prime）と G（Base）を受け入れる必要はありません。

Parameters:

- **key** キーを設定する DHKEY 構造体へのポインタ
- **p** キーで使用するためのプライムを含むバッファへのポインタ
- **pSz** 入力プライムの長さ
- **g** キーで使用するためのベースを含むバッファへのポインタ *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
}
```

See: wc_DhKeyDecode

Return:

- 0 鍵の設定に成功しました
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL に評価された場合に返されます。
- MP_INIT_E ストレージのキーパラメータの初期化中にエラーがある場合に返されます。

- ASN_DH_KEY_E DH キーパラメータ P および G でエラーの読み取りがある場合は返されます

B.23.2.7 function wc_DhParamsLoad

```
int wc_DhParamsLoad(
    const byte * input,
    word32 inSz,
    byte * p,
    word32 * pInOutSz,
    byte * g,
    word32 * gInOutSz
)
```

この関数は、与えられた入力バッファから Diffie-Hellman パラメータ P（Prime）と G（ベース）をフォーマットされています。

Parameters:

- **input** 解析する DER フォーマットされた Difie-Hellman 証明書を含むバッファへのポインタ
- **inSz** 入力バッファのサイズ
- **p** 解析されたプライムを保存するバッファへのポインタ
- **pInOutSz** P バッファ内の使用可能なサイズを含む Word32 オブジェクトへのポインタ。関数呼び出しを完了した後にバッファに書き込まれたバイト数で上書きされます。
- **g** 解析されたベースを保存するバッファへのポインタ *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
}
```

See:

- wc_DhSetKey
- wc_DhKeyDecode

Return:

- 0 DH パラメータの抽出に成功しました
- ASN_PARSE_E DER フォーマットの DH 証明書の解析中にエラーが発生した場合に返されます。
- BUFFER_E 解析されたパラメータを格納するために P または G に不適切なスペースがある場合

B.23.2.8 function wc_Dh_ffdhe2048_Get

```
const DhParams * wc_Dh_ffdhe2048_Get(
    void
)
```

See:

- wc_Dh_ffdhe3072_Get
- wc_Dh_ffdhe4096_Get
- wc_Dh_ffdhe6144_Get
- wc_Dh_ffdhe8192_Get

B.23.2.9 function wc_Dh_ffdhe3072_Get

```
const DhParams * wc_Dh_ffdhe3072_Get(
    void
)
```

See:

- wc_Dh_ffdhe2048_Get
- wc_Dh_ffdhe4096_Get
- wc_Dh_ffdhe6144_Get
- wc_Dh_ffdhe8192_Get

B.23.2.10 function wc_Dh_ffdhe4096_Get

```
const DhParams * wc_Dh_ffdhe4096_Get(
    void
)
```

See:

- wc_Dh_ffdhe2048_Get
- wc_Dh_ffdhe3072_Get
- wc_Dh_ffdhe6144_Get
- wc_Dh_ffdhe8192_Get

B.23.2.11 function wc_Dh_ffdhe6144_Get

```
const DhParams * wc_Dh_ffdhe6144_Get(
    void
)
```

See:

- wc_Dh_ffdhe2048_Get
- wc_Dh_ffdhe3072_Get
- wc_Dh_ffdhe4096_Get
- wc_Dh_ffdhe8192_Get

B.23.2.12 function wc_Dh_ffdhe8192_Get

```
const DhParams * wc_Dh_ffdhe8192_Get(
    void
)
```

See:

- wc_Dh_ffdhe2048_Get
- wc_Dh_ffdhe3072_Get
- wc_Dh_ffdhe4096_Get
- wc_Dh_ffdhe6144_Get

B.23.2.13 function wc_DhCheckKeyPair

```
int wc_DhCheckKeyPair(
    DhKey * key,
    const byte * pub,
    word32 pubSz,
    const byte * priv,
    word32 privSz
)
```

B.23.2.14 function wc_DhCheckPrivKey

```
int wc_DhCheckPrivKey(
    DhKey * key,
    const byte * priv,
    word32 pubSz
)
```

B.24 Algorithms - ECC

B.24.1 Functions

	Name
int	**wc_ecc_make_key**(WC_RNG * rng, int keysize, ecc_key * key) この関数は新しい ECC_KEY を生成し、それをキーに格納します。
int	**wc_ecc_make_key_ex**(WC_RNG * rng, int keysize, ecc_key * key, int curve_id) この関数は新しい ECC_KEY を生成し、それをキーに格納します。
int	**wc_ecc_check_key**(ecc_key * key)ECC キーの有効性を有効にします。
void	**wc_ecc_key_free**(ecc_key * key) この関数は、使用された後に ECC_KEY キーを解放します。 *Example*
int	**wc_ecc_shared_secret**(ecc_key * private_key, ecc_key * public_key, byte * out, word32 * outlen) この関数は、ローカル秘密鍵と受信した公開鍵を使用して新しい秘密鍵を生成します。この共有秘密鍵をバッファアウトに格納し、出力バッファに書き込まれたバイト数を保持するためにoutlenten を更新します。
int	**wc_ecc_shared_secret_ex**(ecc_key * private_key, ecc_point * point, byte * out, word32 * outlen) 秘密鍵とパブリックポイントの間に ECC 共有秘密を作成します。
int	**wc_ecc_sign_hash**(const byte * in, word32 inlen, byte * out, word32 * outlen, WC_RNG * rng, ecc_key * key) この関数は、信頼性を保証するために ECC_KEY オブジェクトを使用してメッセージダイジェストに署名します。
int	**wc_ecc_sign_hash_ex**(const byte * in, word32 inlen, WC_RNG * rng, ecc_key * key, mp_int * r, mp_int * s) メッセージダイジェストに署名します。
int	**wc_ecc_verify_hash**(const byte * sig, word32 siglen, const byte * hash, word32 hashlen, int * stat, ecc_key * key) この関数は、真正性を確保するためにハッシュの ECC シグネチャを検証します。答えを介して、有効な署名に対応する 1、無効な署名に対応する 0 で答えを返します。

	Name
int	**wc_ecc_verify_hash_ex**(mp_int * r, mp_int * s, const byte * hash, word32 hashlen, int * stat, ecc_key * key)ECC 署名を確認してください。結果は stat に書き込まれます。1 が有効で、0 が無効です。注：有効なテストに戻り値を使用しないでください。stat のみを使用してください。
int	**wc_ecc_init**(ecc_key * key) この関数は、メッセージ検証または鍵交渉で将来の使用のために ECC_KEY オブジェクトを初期化します。
int	**wc_ecc_init_ex**(ecc_key * key, void * heap, int devId) この関数は、メッセージ検証または鍵交渉で将来の使用のために ECC_KEY オブジェクトを初期化します。
ecc_key *	**wc_ecc_key_new**(void * heap) この関数はユーザー定義ヒープを使用し、キー構造のスペースを割り当てます。
int	**wc_ecc_free**(ecc_key * key) この関数は、使用後に ECC_KEY オブジェクトを解放します。
void	**wc_ecc_fp_free**(void) この関数は固定小数点キャッシュを解放します。これは ECC で使用でき、計算時間を高速化します。この機能を使用するには、FP_ECC（固定小数点 ECC）を定義する必要があります。
int	**wc_ecc_is_valid_idx**(int n)ECC IDX が有効かどうかを確認します。
ecc_point *	**wc_ecc_new_point**(void) 新しい ECC ポイントを割り当てます。
void	**wc_ecc_del_point**(ecc_point * p) メモリから ECC ポイントを解放します。
int	**wc_ecc_copy_point**(ecc_point * p, ecc_point * r) あるポイントの値を別のポイントにコピーします。
int	**wc_ecc_cmp_point**(ecc_point * a, ecc_point * b) ポイントの値を別のものと比較してください。
int	**wc_ecc_point_is_at_infinity**(ecc_point * p) ポイントが無限大にあるかどうかを確認します。返品 1 が無限大である場合は 0、そうでない場合は 0、<0 エラー時の 0
int	**wc_ecc_mulmod**(mp_int * k, ecc_point * G, ecc_point * R, mp_int * a, mp_int * modulus, int map)ECC 固定点乗算を実行します。
int	**wc_ecc_export_x963**(ecc_key * key, byte * out, word32 * outLen) この関数は ECC キーを ECC_KEY 構造体からエクスポートし、結果を OUT に格納します。キーは ANSI X9.63 フォーマットに保存されます。outlen の出力バッファに書き込まれたバイトを格納します。

	Name
int	**wc_ecc_export_x963_ex**(ecc_key * key, byte * out, word32 * outLen, int compressed) この関数は ECC キーを ECC_KEY 構造体からエクスポートし、結果を OUT に格納します。キーは ANSI X9.63 フォーマットに保存されます。outlen の出力バッファに書き込まれたバイトを格納します。この関数は、圧縮されたパラメータを介して証明書を圧縮する追加のオプションを使用する。このパラメータが true の場合、キーは ANSI X9.63 圧縮形式で保存されます。
int	**wc_ecc_import_x963**(const byte * in, word32 inLen, ecc_key * key) この関数は、ANSI X9.63 形式で保存されているキーを含むバッファからパブリック ECC キーをインポートします。この関数は、圧縮キーが hand_comp_key オプションを介してコンパイル時に有効になっている限り、圧縮キーと非圧縮キーの両方を処理します。
int	**wc_ecc_import_private_key**(const byte * priv, word32 privSz, const byte * pub, word32 pubSz, ecc_key * key) この関数は、生の秘密鍵を含むバッファと、ANSI X9.63 フォーマットされた公開鍵を含む 2 番目のバッファーからパブリック/プライベート ECC キーのペアをインポートします。この関数は、圧縮キーが hand_comp_key オプションを介してコンパイル時に有効になっている限り、圧縮キーと非圧縮キーの両方を処理します。
int	**wc_ecc_rs_to_sig**(const char * r, const char * s, byte * out, word32 * outlen) この関数は、ECC シグネチャの R 部分と S 部分を DER 符号化 ECDSA シグネチャに変換します。この機能は、outlen では、出力バッファに書き込まれた長さも記憶されています。
int	**wc_ecc_import_raw**(ecc_key * key, const char * qx, const char * qy, const char * d, const char * curveName) この関数は、ECC 署名の RAW 成分を持つ ECC_KEY 構造体を埋めます。
int	**wc_ecc_export_private_only**(ecc_key * key, byte * out, word32 * outLen) この関数は、ECC_KEY 構造体から秘密鍵のみをエクスポートします。秘密鍵をバッファアウトに格納し、outlen にこのバッファに書き込まれたバイトを設定します。
int	**wc_ecc_export_point_der**(const int curve_idx, ecc_point * point, byte * out, word32 * outLen)DER へのエクスポートポイント。
int	**wc_ecc_import_point_der**(byte * in, word32 inLen, const int curve_idx, ecc_point * point)Der フォーマットからのインポートポイント。
int	**wc_ecc_size**(ecc_key * key) この関数は、ecc_key 構造体のキーサイズをオクテットで返します。

	Name
int	**wc_ecc_sig_size_calc**(int sz) この関数は、次のようにして指定された ECC シグネチャの最悪の場合のサイズを返します。（KEYSZ * 2）+ SIG_HEADER_SZ + ECC_MAX_PAD_SZ。実際のシグネチャサイズは、WC_ECC_SIGN_HASH で計算できます。
int	**wc_ecc_sig_size**(ecc_key * key) この関数は、次のようにして指定された ECC シグネチャの最悪の場合のサイズを返します。（KEYSZ * 2）+ SIG_HEADER_SZ + ECC_MAX_PAD_SZ。実際のシグネチャサイズは、WC_ECC_SIGN_HASH で計算できます。
ecEncCtx *	**wc_ecc_ctx_new**(int flags, WC_RNG * rng) この機能は、ECC との安全なメッセージ交換を可能にするために、新しい ECC コンテキストオブジェクトのスペースを割り当て、初期化します。
void	**wc_ecc_ctx_free**(ecEncCtx *) この関数は、メッセージの暗号化と復号化に使用される ECENCCTX オブジェクトを解放します。
int	**wc_ecc_ctx_reset**(ecEncCtx * ctx, WC_RNG * rng) この関数は ECENCCTX 構造をリセットして、新しいコンテキストオブジェクトを解放し、新しいコンテキストオブジェクトを割り当てます。
int	**wc_ecc_ctx_set_algo**(ecEncCtx * ctx, byte encAlgo, byte kdfAlgo, byte macAlgo) この関数は、wc_ecc_ctx_new の後にオプションで呼び出されることができます。暗号化、KDF、および MAC アルゴリズムを ECENCENCCTX オブジェクトに設定します。
const byte *	**wc_ecc_ctx_get_own_salt**(ecEncCtx *) この関数は ECENCENCCTX オブジェクトのソルトを返します。この関数は、ECENCCTX の状態が ECSRV_INIT または ECCLI_INIT の場合にのみ呼び出す必要があります。
int	**wc_ecc_ctx_set_peer_salt**(ecEncCtx * ctx, const byte * salt) この関数は、ECENCENCCTX オブジェクトのピアソルトを設定します。
int	**wc_ecc_ctx_set_info**(ecEncCtx * ctx, const byte * info, int sz) この関数は、wc_ecc_ctx_set_peer_salt の前後にオプションで呼び出されることができます。ECENCCTX オブジェクトのオプションの情報を設定します。

	Name
int	**wc_ecc_encrypt**(ecc_key * privKey, ecc_key * pubKey, const byte * msg, word32 msgSz, byte * out, word32 * outSz, ecEncCtx * ctx) この関数は指定された入力メッセージを MSG から OUT に暗号化します。この関数はパラメータとしてオプションの CTX オブジェクトを取ります。提供されている場合、ECENCCTX の Encalgo、Kdfalgo、および Macalgo に基づいて暗号化が進みます。CTX が指定されていない場合、処理はデフォルトのアルゴリズム、ECAES_128_CBC、ECHKDF_SHA256、ECHMAC_SHA256 で完了します。この機能は、メッセージが CTX で指定された暗号化タイプに従って埋め込まれている必要があります。
int	**wc_ecc_encrypt_ex**(ecc_key * privKey, ecc_key * pubKey, const byte * msg, word32 msgSz, byte * out, word32 * outSz, ecEncCtx * ctx, int compressed) この関数は指定された入力メッセージを MSG から OUT に暗号化します。この関数はパラメータとしてオプションの CTX オブジェクトを取ります。提供されている場合、ECENCCTX の Encalgo、Kdfalgo、および Macalgo に基づいて暗号化が進みます。CTX が指定されていない場合、処理はデフォルトのアルゴリズム、ECAES_128_CBC、ECHKDF_SHA256、ECHMAC_SHA256 で完了します。この機能は、メッセージが CTX で指定された暗号化タイプに従って埋め込まれている必要があります。
int	**wc_ecc_decrypt**(ecc_key * privKey, ecc_key * pubKey, const byte * msg, word32 msgSz, byte * out, word32 * outSz, ecEncCtx * ctx) この関数は MSG から OUT への暗号文を復号化します。この関数はパラメータとしてオプションの CTX オブジェクトを取ります。提供されている場合、ECENCCTX の Encalgo、Kdfalgo、および Macalgo に基づいて暗号化が進みます。CTX が指定されていない場合、処理はデフォルトのアルゴリズム、ECAES_128_CBC、ECHKDF_SHA256、ECHMAC_SHA256 で完了します。この機能は、メッセージが CTX で指定された暗号化タイプに従って埋め込まれている必要があります。
int	**wc_ecc_set_nonblock**(ecc_key * key, ecc_nb_ctx_t * ctx) 非ブロック操作のための ECC サポートを有効にします。次のビルドオプションを使用した単精度（SP）数学でサポートされています.WolfSSL_SP_SP_SMALL WOLFSSL_SP_NO_MALLOC WC_ECC_NONBLOCK

B.24.2 Functions Documentation

B.24.2.1 function wc_ecc_make_key

```
int wc_ecc_make_key(
    WC_RNG * rng,
    int keysize,
    ecc_key * key
)
```

この関数は新しい ECC_KEY を生成し、それをキーに格納します。

Parameters:

- **rng** キーを生成するための初期化された RNG オブジェクトへのポインタ
- **keysize** ECC_KEY の希望の長さ *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
```

See:

- wc_ecc_init
- wc_ecc_shared_secret

Return:

- 0 成功に戻りました。
- ECC_BAD_ARG_E RNG またはキーが NULL に評価された場合に返されます
- BAD_FUNC_ARG 指定されたキーサイズがサポートされているキーの正しい範囲にない場合に返されます。
- MEMORY_E ECC キーの計算中にメモリの割り当てエラーがある場合に返されます。
- MP_INIT_E ECC キーの計算中にエラーが発生した場合に返される可能性があります
- MP_READ_E ECC キーの計算中にエラーが発生した場合に返される可能性があります
- MP_CMP_E ECC キーの計算中にエラーが発生した場合に返される可能性があります
- MP_INVMOD_E ECC キーの計算中にエラーが発生した場合に返される可能性があります
- MP_EXPTMOD_E ECC キーの計算中にエラーが発生した場合に返される可能性があります
- MP_MOD_E ECC キーの計算中にエラーが発生した場合に返される可能性があります
- MP_MUL_E ECC キーの計算中にエラーが発生した場合に返される可能性があります
- MP_ADD_E ECC キーの計算中にエラーが発生した場合に返される可能性があります
- MP_MULMOD_E ECC キーの計算中にエラーが発生した場合に返される可能性があります
- MP_TO_E ECC キーの計算中にエラーが発生した場合に返される可能性があります
- MP_MEM ECC キーの計算中にエラーが発生した場合に返される可能性があります

B.24.2.2 function wc_ecc_make_key_ex

```
int wc_ecc_make_key_ex(
    WC_RNG * rng,
    int keysize,
    ecc_key * key,
    int curve_id
)
```

この関数は新しい ECC_KEY を生成し、それをキーに格納します。

Parameters:

- **key** 作成したキーを保存するためのポインタ。
- **keysize** CavenID に基づいて設定されたバイト単位で作成するキーのサイズ
- **rng** 鍵作成に使用される RNG *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
}
```

See:

- wc_ecc_make_key
- wc_ecc_get_curve_size_from_id

Return:

- 0 成功に戻りました。
- ECC_BAD_ARG_E RNG またはキーが NULL に評価された場合に返されます
- BAD_FUNC_ARG 指定されたキーサイズがサポートされているキーの正しい範囲にない場合に返されます。
- MEMORY_E ECC キーの計算中にメモリの割り当てエラーがある場合に返されます。
- MP_INIT_E ECC キーの計算中にエラーが発生した場合に返される可能性があります
- MP_READ_E ECC キーの計算中にエラーが発生した場合に返される可能性があります
- MP_CMP_E ECC キーの計算中にエラーが発生した場合に返される可能性があります
- MP_INVMOD_E ECC キーの計算中にエラーが発生した場合に返される可能性があります
- MP_EXPTMOD_E ECC キーの計算中にエラーが発生した場合に返される可能性があります
- MP_MOD_E ECC キーの計算中にエラーが発生した場合に返される可能性があります
- MP_MUL_E ECC キーの計算中にエラーが発生した場合に返される可能性があります
- MP_ADD_E ECC キーの計算中にエラーが発生した場合に返される可能性があります
- MP_MULMOD_E ECC キーの計算中にエラーが発生した場合に返される可能性があります
- MP_TO_E ECC キーの計算中にエラーが発生した場合に返される可能性があります
- MP_MEM ECC キーの計算中にエラーが発生した場合に返される可能性があります

B.24.2.3 function wc_ecc_check_key

```
int wc_ecc_check_key(
    ecc_key * key
)
```

ECC キーの有効性を有効にします。

See: wc_ecc_point_is_at_infinity

Return:

- MP_OKAY 成功、キーは大丈夫です。
- BAD_FUNC_ARG キーが NULL の場合は返します。
- ECC_INF_E WC_ECC_POINT_IS_AT_INFINITY が 1 を返す場合に返します。*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
}
```

B.24.2.4 function wc_ecc_key_free

```
void wc_ecc_key_free(
    ecc_key * key
)
```

この関数は、使用された後に ECC_KEY キーを解放します。*Example*

See:

- wc_ecc_key_new
- wc_ecc_init_ex

```
// initialize key and perform ECC operations
...
wc_ecc_key_free(&key);
```

B.24.2.5 function wc_ecc_shared_secret

```
int wc_ecc_shared_secret(
    ecc_key * private_key,
    ecc_key * public_key,
    byte * out,
    word32 * outlen
)
```

この関数は、ローカル秘密鍵と受信した公開鍵を使用して新しい秘密鍵を生成します。この共有秘密鍵をバッファアウトに格納し、出力バッファに書き込まれたバイト数を保持するために outlenten を更新します。

Parameters:

- **private_key** ローカル秘密鍵を含む ECC_KEY 構造体へのポインタ
- **public_key** 受信した公開鍵を含む ECC_Key 構造体へのポインタ
- **out** 生成された共有秘密鍵を保存する出力バッファへのポインタ *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
}
```

See:

- wc_ecc_init
- wc_ecc_make_key

Return:

- 0 共有秘密鍵の生成に成功したときに返されます
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL に評価された場合に返されます。
- ECC_BAD_ARG_E 引数として与えられた秘密鍵の種類が ecc_privatekey ではない場合、またはパブリックキータイプ（ECC-> DP によって与えられた）が同等でない場合に返されます。
- MEMORY_E 新しい ECC ポイントを生成するエラーが発生した場合
- BUFFER_E 生成された共有秘密鍵が提供されたバッファーに格納するのに長すぎる場合に返されます
- MP_INIT_E 共有キーの計算中にエラーが発生した場合は返される可能性があります
- MP_READ_E 共有キーの計算中にエラーが発生した場合は返される可能性があります
- MP_CMP_E 共有キーの計算中にエラーが発生した場合は返される可能性があります
- MP_INVMOD_E 共有キーの計算中にエラーが発生した場合は返される可能性があります
- MP_EXPTMOD_E 共有キーの計算中にエラーが発生した場合は返される可能性があります
- MP_MOD_E 共有キーの計算中にエラーが発生した場合は返される可能性があります
- MP_MUL_E 共有キーの計算中にエラーが発生した場合は返される可能性があります
- MP_ADD_E 共有キーの計算中にエラーが発生した場合は返される可能性があります
- MP_MULMOD_E 共有キーの計算中にエラーが発生した場合は返される可能性があります
- MP_TO_E 共有キーの計算中にエラーが発生した場合は返される可能性があります
- MP_MEM 共有キーの計算中にエラーが発生した場合は返される可能性があります

B.24.2.6 function wc_ecc_shared_secret_ex

```
int wc_ecc_shared_secret_ex(
    ecc_key * private_key,
    ecc_point * point,
    byte * out,
    word32 * outlen
)
```

秘密鍵とパブリックポイントの間に ECC 共有秘密を作成します。

Parameters:

- **private_key** プライベート ECC キー。
- **point** 使用するポイント（公開鍵）。
- **out** 共有秘密の出力先。ANSI X9.63 から EC-DH に準拠しています。*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
}
```

See: wc_ecc_verify_hash_ex

Return:

- MP_OKAY 成功を示します。
- BAD_FUNC_ARG 引数が NULL のときにエラーが返されます。
- ECC_BAD_ARG_E private_key-> type が ecc_privatekey または private_key-> idx が検証できない場合に返されました。
- BUFFER_E outlen が小さすぎるとエラーが発生します。
- MEMORY_E 新しいポイントを作成するためのエラー。
- MP_VAL 初期化失敗が発生したときに可能です。
- MP_MEM 初期化失敗が発生したときに可能です。

B.24.2.7 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
)
```

この関数は、信頼性を保証するために ECC_KEY オブジェクトを使用してメッセージダイジェストに署名します。

Parameters:

- **in** サインするメッセージハッシュを含むバッファへのポインタ
- **inlen** 署名するメッセージの長さ
- **out** 生成された署名を保存するためのバッファ
- **outlen** 出力バッファの最大長。メッセージ署名の生成に成功したときに書き込まれたバイトを保存します *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
}
```

See: wc_ecc_verify_hash

Return:

- 0 メッセージダイジェストの署名を正常に生成したときに返されました

- BAD_FUNC_ARG 入力パラメータのいずれかが NULL に評価された場合、または出力バッファが小さすぎて生成された署名を保存する場合は返されます。
- ECC_BAD_ARG_E 入力キーが秘密鍵ではない場合、または ECC OID が無効な場合
- RNG_FAILURE_E RNG が満足のいくキーを正常に生成できない場合に返されます。
- MP_INIT_E メッセージ署名の計算中にエラーが発生した場合に返される可能性があります。
- MP_READ_E メッセージ署名の計算中にエラーが発生した場合に返される可能性があります。
- MP_CMP_E メッセージ署名の計算中にエラーが発生した場合に返される可能性があります。
- MP_INVMOD_E メッセージ署名の計算中にエラーが発生した場合に返される可能性があります。
- MP_EXPTMOD_E メッセージ署名の計算中にエラーが発生した場合に返される可能性があります。
- MP_MOD_E メッセージ署名の計算中にエラーが発生した場合に返される可能性があります。
- MP_MUL_E メッセージ署名の計算中にエラーが発生した場合に返される可能性があります。
- MP_ADD_E メッセージ署名の計算中にエラーが発生した場合に返される可能性があります。
- MP_MULMOD_E メッセージ署名の計算中にエラーが発生した場合に返される可能性があります。
- MP_TO_E メッセージ署名の計算中にエラーが発生した場合に返される可能性があります。
- MP_MEM メッセージ署名の計算中にエラーが発生した場合に返される可能性があります。

B.24.2.8 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
)
```

メッセージダイジェストに署名します。

Parameters:

- **in** メッセージがサインにダイジェスト。
- **inlen** ダイジェストの長さ。
- **rng** WC_RNG 構造体へのポインタ。
- **key** プライベート ECC キー。
- **r** 署名の R コンポーネントの宛先。*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
}
```

See: wc_ecc_verify_hash_ex

Return:

- MP_OKAY メッセージダイジェストの署名を正常に生成したときに返されました
- ECC_BAD_ARG_E 入力キーが秘密鍵ではない場合、または ECC IDX が無効な場合、またはいずれかの入力パラメータが NULL に評価されている場合、または出力バッファが小さすぎて生成された署名を保存するには小さすぎる場合
- RNG_FAILURE_E RNG が満足のいくキーを正常に生成できない場合に返されます。
- MP_INIT_E メッセージ署名の計算中にエラーが発生した場合に返される可能性があります。
- MP_READ_E メッセージ署名の計算中にエラーが発生した場合に返される可能性があります。
- MP_CMP_E メッセージ署名の計算中にエラーが発生した場合に返される可能性があります。
- MP_INVMOD_E メッセージ署名の計算中にエラーが発生した場合に返される可能性があります。
- MP_EXPTMOD_E メッセージ署名の計算中にエラーが発生した場合に返される可能性があります。
- MP_MOD_E メッセージ署名の計算中にエラーが発生した場合に返される可能性があります。
- MP_MUL_E メッセージ署名の計算中にエラーが発生した場合に返される可能性があります。
- MP_ADD_E メッセージ署名の計算中にエラーが発生した場合に返される可能性があります。
- MP_MULMOD_E メッセージ署名の計算中にエラーが発生した場合に返される可能性があります。
- MP_TO_E メッセージ署名の計算中にエラーが発生した場合に返される可能性があります。
- MP_MEM メッセージ署名の計算中にエラーが発生した場合に返される可能性があります。

B.24.2.9 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
)
```

この関数は、真正性を確保するためにハッシュの ECC シグネチャを検証します。答えを介して、有効な署名に対応する 1、無効な署名に対応する 0 で答えを返します。

Parameters:

- **sig** 確認する署名を含むバッファへのポインタ
- **siglen** 検証する署名の長さ
- **hash** 確認されたメッセージのハッシュを含むバッファへのポインタ
- **hashlen** 認証されたメッセージのハッシュの長さ
- **stat** 検証の結果へのポインタ。1 メッセージが正常に認証されたことを示します *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
}
```

See:

- wc_ecc_sign_hash
- wc_ecc_verify_hash_ex

Return:

- 0 署名検証に正常に実行されたときに返されます。注：これは署名が検証されていることを意味するわけではありません。信頼性情報は代わりに STAT で格納されます
- BAD_FUNC_ARG 返された入力パラメータは NULL に評価されます
- MEMORY_E メモリの割り当て中にエラーが発生した場合に返されます
- MP_INIT_E メッセージ署名の計算中にエラーが発生した場合に返される可能性があります。
- MP_READ_E メッセージ署名の計算中にエラーが発生した場合に返される可能性があります。
- MP_CMP_E メッセージ署名の計算中にエラーが発生した場合に返される可能性があります。
- MP_INVMOD_E メッセージ署名の計算中にエラーが発生した場合に返される可能性があります。
- MP_EXPTMOD_E メッセージ署名の計算中にエラーが発生した場合に返される可能性があります。
- MP_MOD_E メッセージ署名の計算中にエラーが発生した場合に返される可能性があります。
- MP_MUL_E メッセージ署名の計算中にエラーが発生した場合に返される可能性があります。
- MP_ADD_E メッセージ署名の計算中にエラーが発生した場合に返される可能性があります。
- MP_MULMOD_E メッセージ署名の計算中にエラーが発生した場合に返される可能性があります。
- MP_TO_E メッセージ署名の計算中にエラーが発生した場合に返される可能性があります。
- MP_MEM メッセージ署名の計算中にエラーが発生した場合に返される可能性があります。

B.24.2.10 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
)
```

ECC 署名を確認してください。結果は stat に書き込まれます。1 が有効で、0 が無効です。注：有効なテストに戻り値を使用しないでください。stat のみを使用してください。

Parameters:

- **r** 検証する署名 R コンポーネント
- **s** 検証するシグネチャ S コンポーネント
- **hash** 署名されたハッシュ（メッセージダイジェスト）
- **hashlen** ハッシュの長さ（オクテット）
- **stat** 署名の結果、1 == 有効、0 == 無効 *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
}
```

See: wc_ecc_verify_hash

Return:

- MP_OKAY 成功した場合（署名が無効であっても）

- ECC_BAD_ARG_E 引数が NULL の場合、または key-idx が無効な場合は返します。
- MEMORY_E INT またはポイントの割り当て中にエラーが発生しました。

B.24.2.11 function wc_ecc_init

```
int wc_ecc_init(
    ecc_key * key
)
```

この関数は、メッセージ検証または鍵交渉で将来の使用のために ECC_KEY オブジェクトを初期化します。

See:

- wc_ecc_make_key
- wc_ecc_free

Return:

- 0 ECC_Key オブジェクトの初期化に成功したときに返されます
- MEMORY_E メモリの割り当て中にエラーが発生した場合に返されます *Example*

```
ecc_key key;
wc_ecc_init(&key);
```

B.24.2.12 function wc_ecc_init_ex

```
int wc_ecc_init_ex(
    ecc_key * key,
    void * heap,
    int devId
)
```

この関数は、メッセージ検証または鍵交渉で将来の使用のために ECC_KEY オブジェクトを初期化します。

Parameters:

- **key** 初期化する ECC_Key オブジェクトへのポインタ
- **devId** 非同期ハードウェアで使用する ID *Example*

```
ecc_key key;
wc_ecc_init_ex(&key, heap, devId);
```

See:

- wc_ecc_make_key
- wc_ecc_free
- wc_ecc_init

Return:

- 0 ECC_Key オブジェクトの初期化に成功したときに返されます
- MEMORY_E メモリの割り当て中にエラーが発生した場合に返されます

B.24.2.13 function wc_ecc_key_new

```
ecc_key * wc_ecc_key_new(
    void * heap
)
```

この関数はユーザー定義ヒープを使用し、キー構造のスペースを割り当てます。

See:

- wc_ecc_make_key
- wc_ecc_key_free
- wc_ecc_init

Return: 0 ECC_Key オブジェクトの初期化に成功したときに返されます *Example*

```
wc_ecc_key_new(&heap);
```

B.24.2.14 function wc_ecc_free

```
int wc_ecc_free(
    ecc_key * key
)
```

この関数は、使用後に ECC_KEY オブジェクトを解放します。

See: wc_ecc_init

Return: int integer が WolfSSL エラーまたは成功状況を示すことを返しました。*Example*

```
// initialize key and perform secure exchanges
...
wc_ecc_free(&key);
```

B.24.2.15 function wc_ecc_fp_free

```
void wc_ecc_fp_free(
    void
)
```

この関数は固定小数点キャッシュを解放します。これは ECC で使用でき、計算時間を高速化します。この機能を使用するには、FP_ECC（固定小数点 ECC）を定義する必要があります。

See: wc_ecc_free

Return: none いいえ返します。*Example*

```
ecc_key key;
// initialize key and perform secure exchanges
...

wc_ecc_fp_free();
```

B.24.2.16 function wc_ecc_is_valid_idx

```
int wc_ecc_is_valid_idx(
    int n
)
```

ECC IDX が有効かどうかを確認します。

See: none

Return:

- 1 有効な場合は返品してください。
- 0 無効な場合は返します。*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
}
```

B.24.2.17 function wc_ecc_new_point

```
ecc_point * wc_ecc_new_point(
    void
)
```

新しい ECC ポイントを割り当てます。

See:

- wc_ecc_del_point
- wc_ecc_cmp_point
- wc_ecc_copy_point

Return:

- p 新しく割り当てられたポイント。
- NULL エラー時に NULL を返します。*Example*

```
ecc_point* point;
point = wc_ecc_new_point();
if (point == NULL)
{
    // Handle point creation error
}
// Do stuff with point
```

B.24.2.18 function wc_ecc_del_point

```
void wc_ecc_del_point(
    ecc_point * p
)
```

メモリから ECC ポイントを解放します。

See:

- wc_ecc_new_point
- wc_ecc_cmp_point
- wc_ecc_copy_point

Return: none いいえ返します。*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);
```

B.24.2.19 function wc_ecc_copy_point

```
int wc_ecc_copy_point(
    ecc_point * p,
    ecc_point * r
)
```

あるポイントの値を別のポイントにコピーします。

Parameters:

- **p** コピーするポイント。*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
}
```

See:

- wc_ecc_new_point
- wc_ecc_cmp_point
- wc_ecc_del_point

Return:

- ECC_BAD_ARG_E P または R が NULL のときにスローされたエラー。
- MP_OKAY ポイントが正常にコピーされました
- ret 内部関数からのエラー。になることができます...

B.24.2.20 function wc_ecc_cmp_point

```
int wc_ecc_cmp_point(
    ecc_point * a,
    ecc_point * b
)
```

ポイントの値を別のものと比較してください。

Parameters:

- **a** 比較する最初のポイント。*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
}
```

See:

- wc_ecc_new_point
- wc_ecc_del_point
- wc_ecc_copy_point

Return:

- BAD_FUNC_ARG 1 つまたは両方の引数は null です。
- MP_EQ ポイントは同じです。
- ret mp_lt または mp_gt のどちらかで、ポイントが等しくないことを意味します。

B.24.2.21 function wc_ecc_point_is_at_infinity

```
int wc_ecc_point_is_at_infinity(
    ecc_point * p
)
```

ポイントが無限大にあるかどうかを確認します。返品 1 が無限大である場合は 0、そうでない場合は 0、<0 エラー時の 0

See:

- wc_ecc_new_point
- wc_ecc_del_point
- wc_ecc_cmp_point
- wc_ecc_copy_point

Return:

- 1 P は無限大です。
- 0 P は無限大ではありません。
- <0 エラー。*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
}
```

B.24.2.22 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
)
```

ECC 固定点乗算を実行します。

Parameters:

- **k** 計量。
- **G** 乗算する基点。
- **R** 商品の目的地
- **modulus** 曲線の弾性率 *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;
```

See: none

Return:

- MP_OKAY 成功した操作で返します。
- MP_INIT_E 複数の Precision Integer（MP_INT）ライブラリで使用するための整数を初期化するエラーがある場合に返されます。

B.24.2.23 function wc_ecc_export_x963

```
int wc_ecc_export_x963(
    ecc_key * key,
    byte * out,
    word32 * outLen
)
```

この関数は ECC キーを ECC_KEY 構造体からエクスポートし、結果を OUT に格納します。キーは ANSI X9.63 フォーマットに保存されます。outlen の出力バッファに書き込まれたバイトを格納します。

Parameters:

- **key** エクスポートする ECC_KEY オブジェクトへのポインタ
- **out** ANSI X9.63 フォーマット済みキーを保存するバッファへのポインタ *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
}
```

See:

- wc_ecc_export_x963_ex
- wc_ecc_import_x963

Return:

- 0 ECC_KEY のエクスポートに正常に返されました
- LENGTH_ONLY_E 出力バッファが NULL に評価されている場合は返されますが、他の 2 つの入力パラメータは有効です。関数がキーを保存するのに必要な長さを返すだけであることを示します
- ECC_BAD_ARG_E 入力パラメータのいずれかが NULL の場合、またはキーがサポートされていない場合（無効なインデックスがあります）
- BUFFER_E 出力バッファが小さすぎて ECC キーを保存する場合は返されます。出力バッファが小さすぎると、必要なサイズは outlen に返されます。
- MEMORY_E xmalloc でメモリを割り当てるエラーがある場合
- MP_INIT_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_READ_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_CMP_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_INVMOD_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_EXPTMOD_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_MOD_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_MUL_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_ADD_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_MULMOD_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_TO_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_MEM ECC_KEY の処理中にエラーが発生した場合に返される可能性があります

B.24.2.24 function wc_ecc_export_x963_ex

```
int wc_ecc_export_x963_ex(
    ecc_key * key,
    byte * out,
    word32 * outLen,
    int compressed
)
```

この関数は ECC キーを ECC_KEY 構造体からエクスポートし、結果を OUT に格納します。キーは ANSI X9.63 フォーマットに保存されます。outlen の出力バッファに書き込まれたバイトを格納します。この関数は、圧縮されたパラメータを介して証明書を圧縮する追加のオプションを使用する。このパラメータが true の場合、キーは ANSI X9.63 圧縮形式で保存されます。

Parameters:

- **key** エクスポートする ECC_KEY オブジェクトへのポインタ
- **out** ANSI X9.63 フォーマット済みキーを保存するバッファへのポインタ
- **outLen** 出力バッファのサイズ。キーの保存に成功した場合は、出力バッファに書き込まれたバイトを保持します。*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
}
```

See:

- wc_ecc_export_x963
- wc_ecc_import_x963

Return:

- 0 ECC_KEY のエクスポートに正常に返されました
- NOT_COMPILED_IN hand_comp_key がコンパイル時に有効になっていない場合は返されますが、キーは圧縮形式で要求されました
- LENGTH_ONLY_E 出力バッファが NULL に評価されている場合は返されますが、他の 2 つの入力パラメータは有効です。関数がキーを保存するのに必要な長さを返すだけであることを示します
- ECC_BAD_ARG_E 入力パラメータのいずれかが NULL の場合、またはキーがサポートされていない場合（無効なインデックスがあります）
- BUFFER_E 出力バッファが小さすぎて ECC キーを保存する場合は返されます。出力バッファが小さすぎると、必要なサイズは outlen に返されます。
- MEMORY_E xmalloc でメモリを割り当てるエラーがある場合
- MP_INIT_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_READ_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_CMP_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_INVMOD_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_EXPTMOD_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_MOD_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_MUL_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_ADD_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_MULMOD_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_TO_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_MEM ECC_KEY の処理中にエラーが発生した場合に返される可能性があります

B.24.2.25 function wc_ecc_import_x963

```
int wc_ecc_import_x963(
    const byte * in,
    word32 inLen,
    ecc_key * key
)
```

この関数は、ANSI X9.63 形式で保存されているキーを含むバッファからパブリック ECC キーをインポートします。この関数は、圧縮キーが hand_comp_key オプションを介してコンパイル時に有効になっている限り、圧縮キーと非圧縮キーの両方を処理します。

Parameters:

- **in** ANSI x9.63 フォーマットされた ECC キーを含むバッファへのポインタ
- **inLen** 入力バッファの長さ *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
}
```

See:

- wc_ecc_export_x963
- wc_ecc_import_private_key

Return:

- 0 ECC_KEY のインポートに成功しました
- NOT_COMPILED_IN hand_comp_key がコンパイル時に有効になっていない場合は返されますが、キーは圧縮形式で保存されます。
- ECC_BAD_ARG_E IN または KEY が NULL に評価された場合、または Inlen が偶数の場合（X9.63 規格によれば、キーは奇数でなければなりません）。
- MEMORY_E メモリの割り当て中にエラーが発生した場合に返されます
- ASN_PARSE_E ECC キーの解析中にエラーがある場合は返されます。ECC キーが有効な ANSI X9.63 フォーマットに格納されていないことを示すことがあります。
- IS_POINT_E エクスポートされた公開鍵が ECC 曲線上の点ではない場合に返されます
- MP_INIT_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_READ_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_CMP_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_INVMOD_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_EXPTMOD_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_MOD_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_MUL_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_ADD_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_MULMOD_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_TO_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_MEM ECC_KEY の処理中にエラーが発生した場合に返される可能性があります

B.24.2.26 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
)
```

この関数は、生の秘密鍵を含むバッファと、ANSI X9.63 フォーマットされた公開鍵を含む 2 番目のバッファーからパブリック/プライベート ECC キーのペアをインポートします。この関数は、圧縮キーが hand_comp_key オプションを介してコンパイル時に有効になっている限り、圧縮キーと非圧縮キーの両方を処理します。

Parameters:

- **priv** RAW 秘密鍵を含むバッファへのポインタ
- **privSz** 秘密鍵バッファのサイズ
- **pub** ANSI x9.63 フォーマットされた ECC 公開鍵を含むバッファへのポインタ
- **pubSz** 公開鍵入力バッファの長さ *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
}
```

See:

- wc_ecc_export_x963
- wc_ecc_import_private_key

Return:

- 0 habe_comp_key がコンパイル時に有効になっていない場合は、ecc_key not_compiled_in を正常にインポートしましたが、キーは圧縮形式で保存されます。
- ECC_BAD_ARG_E IN または KEY が NULL に評価された場合、または Inlen が偶数の場合（X9.63 規格によれば、キーは奇数でなければなりません）。
- MEMORY_E メモリの割り当て中にエラーが発生した場合に返されます
- ASN_PARSE_E ECC キーの解析中にエラーがある場合は返されます。ECC キーが有効な ANSI X9.63 フォーマットに格納されていないことを示すことがあります。
- IS_POINT_E エクスポートされた公開鍵が ECC 曲線上の点ではない場合に返されます
- MP_INIT_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_READ_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_CMP_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_INVMOD_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_EXPTMOD_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_MOD_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_MUL_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_ADD_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_MULMOD_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_TO_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_MEM ECC_KEY の処理中にエラーが発生した場合に返される可能性があります

B.24.2.27 function wc_ecc_rs_to_sig

```
int wc_ecc_rs_to_sig(
    const char * r,
    const char * s,
    byte * out,
    word32 * outlen
)
```

この関数は、ECC シグネチャの R 部分と S 部分を DER 符号化 ECDSA シグネチャに変換します。この機能は、outlen では、出力バッファに書き込まれた長さも記憶されています。

Parameters:

- **r** 署名の R 部分を文字列として含むバッファへのポインタ
- **s** シグネチャの S 部分を含むバッファへのポインタ文字列としてのポインタ
- **out** DER エンコードされた ECDSA シグネチャを保存するバッファへのポインタ *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
}
```

See:

- wc_ecc_sign_hash
- wc_ecc_sig_size

Return:

- 0 署名の変換に成功したことに戻りました
- ECC_BAD_ARG_E いずれかの入力パラメータが NULL に評価された場合、または入力バッファが DER エンコードされた ECDSA シグネチャを保持するのに十分な大きさでない場合に返されます。
- MP_INIT_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_READ_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_CMP_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_INVMOD_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_EXPTMOD_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_MOD_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_MUL_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_ADD_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_MULMOD_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_TO_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_MEM ECC_KEY の処理中にエラーが発生した場合に返される可能性があります

B.24.2.28 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
)
```

この関数は、ECC 署名の RAW 成分を持つ ECC_KEY 構造体を埋めます。

Parameters:

- **key** 塗りつぶすための ECC_KEY 構造体へのポインタ
- **qx** ASCII 六角文字列として基点の X コンポーネントを含むバッファへのポインタ
- **qy** ASCII 六角文字列として基点の Y 成分を含むバッファへのポインタ
- **d** ASCII hex 文字列として秘密鍵を含むバッファへのポインタ *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
}
```

See: wc_ecc_import_private_key

Return:

- 0 ECC_Key 構造体に正常にインポートされたときに返されます
- ECC_BAD_ARG_E いずれかの入力値が NULL に評価された場合に返されます。
- MEMORY_E ECC_Key のパラメータを格納するためのエラーの初期化スペースがある場合に返されます。
- ASN_PARSE_E 入力カーベナデが ECC_SETS で定義されていない場合
- MP_INIT_E 入力パラメータの処理中にエラーが発生した場合に返される可能性があります。
- MP_READ_E 入力パラメータの処理中にエラーが発生した場合に返される可能性があります。
- MP_CMP_E 入力パラメータの処理中にエラーが発生した場合に返される可能性があります。
- MP_INVMOD_E 入力パラメータの処理中にエラーが発生した場合に返される可能性があります。
- MP_EXPTMOD_E 入力パラメータの処理中にエラーが発生した場合に返される可能性があります。
- MP_MOD_E 入力パラメータの処理中にエラーが発生した場合に返される可能性があります。
- MP_MUL_E 入力パラメータの処理中にエラーが発生した場合に返される可能性があります。
- MP_ADD_E 入力パラメータの処理中にエラーが発生した場合に返される可能性があります。
- MP_MULMOD_E 入力パラメータの処理中にエラーが発生した場合に返される可能性があります。
- MP_TO_E 入力パラメータの処理中にエラーが発生した場合に返される可能性があります。
- MP_MEM 入力パラメータの処理中にエラーが発生した場合に返される可能性があります。

B.24.2.29 function wc_ecc_export_private_only

```
int wc_ecc_export_private_only(
    ecc_key * key,
    byte * out,
    word32 * outLen
)
```

この関数は、ECC_KEY 構造体から秘密鍵のみをエクスポートします。秘密鍵をバッファアウトに格納し、outlen にこのバッファに書き込まれたバイトを設定します。

Parameters:

- **key** 秘密鍵をエクスポートする ECC_Key 構造体へのポインタ
- **out** 秘密鍵を保存するバッファへのポインタ *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
}
```

See: wc_ecc_import_private_key

Return:

- 0 秘密鍵のエクスポートに成功したときに返されます
- ECC_BAD_ARG_E いずれかの入力値が NULL に評価された場合に返されます。
- MEMORY_E ECC_Key のパラメータを格納するためのエラーの初期化スペースがある場合に返されます。
- ASN_PARSE_E 入力カーベナデが ECC_SETS で定義されていない場合
- MP_INIT_E 入力パラメータの処理中にエラーが発生した場合に返される可能性があります。
- MP_READ_E 入力パラメータの処理中にエラーが発生した場合に返される可能性があります。
- MP_CMP_E 入力パラメータの処理中にエラーが発生した場合に返される可能性があります。
- MP_INVMOD_E 入力パラメータの処理中にエラーが発生した場合に返される可能性があります。
- MP_EXPTMOD_E 入力パラメータの処理中にエラーが発生した場合に返される可能性があります。
- MP_MOD_E 入力パラメータの処理中にエラーが発生した場合に返される可能性があります。
- MP_MUL_E 入力パラメータの処理中にエラーが発生した場合に返される可能性があります。
- MP_ADD_E 入力パラメータの処理中にエラーが発生した場合に返される可能性があります。
- MP_MULMOD_E 入力パラメータの処理中にエラーが発生した場合に返される可能性があります。
- MP_TO_E 入力パラメータの処理中にエラーが発生した場合に返される可能性があります。
- MP_MEM 入力パラメータの処理中にエラーが発生した場合に返される可能性があります。

B.24.2.30 function wc_ecc_export_point_der

```
int wc_ecc_export_point_der(
    const int curve_idx,
    ecc_point * point,
    byte * out,
    word32 * outLen
)
```

DER へのエクスポートポイント。

Parameters:

- **curve_idx** ECC_SETS から使用される曲線のインデックス。
- **point** Der へのエクスポートを指す。
- **out** 出力の目的地 *Example*

```
int curve_idx;
ecc_point* point;
byte out[];
word32 outLen;
wc_ecc_export_point_der(curve_idx, point, out, &outLen);
```

See: wc_ecc_import_point_der

Return:

- 0 成功に戻りました。
- ECC_BAD_ARG_E curve_idx が 0 未満または無効である場合は返します。いつ来るのか
- LENGTH_ONLY_E outlen は設定されていますが、他にはありません。
- BUFFER_E outlenn が 1 + 2 * 曲線サイズよりも小さい場合は返します。
- MEMORY_E メモリの割り当てに問題がある場合は返します。

B.24.2.31 function wc_ecc_import_point_der

```
int wc_ecc_import_point_der(
    byte * in,
    word32 inLen,
    const int curve_idx,
    ecc_point * point
```

)

Der フォーマットからのインポートポイント。

Parameters:

- **in** からのポイントをインポートするための Der Buffer。
- **inLen** DER バッファの長さ
- **curve_idx** 曲線のインデックス *Example*

```
byte in[];
word32 inLen;
int curve_idx;
ecc_point* point;
wc_ecc_import_point_der(in, inLen, curve_idx, point);
```

See: wc_ecc_export_point_der

Return:

- ECC_BAD_ARG_E 引数が null の場合、または Inlen が偶数の場合は返します。
- MEMORY_E エラー初期化がある場合に返します
- NOT_COMPILED_IN habe_comp_key が真実でない場合は返され、in は圧縮証明書です
- MP_OKAY 操作が成功しました。

B.24.2.32 function wc_ecc_size

```
int wc_ecc_size(
    ecc_key * key
)
```

この関数は、ecc_key 構造体のキーサイズをオクテットで返します。

See: wc_ecc_make_key

Return:

- Given 有効なキー、オクテットのキーサイズを返します
- 0 与えられたキーが NULL の場合に返されます *Example*

```
int keySz;
ecc_key key;
// initialize key, make key
keySz = wc_ecc_size(&key);
if ( keySz == 0) {
    // error determining key size
}
```

B.24.2.33 function wc_ecc_sig_size_calc

```
int wc_ecc_sig_size_calc(
    int sz
)
```

この関数は、次のようにして指定された ECC シグネチャの最悪の場合のサイズを返します。(KEYSZ * 2) + SIG_HEADER_SZ + ECC_MAX_PAD_SZ。実際のシグネチャサイズは、WC_ECC_SIGN_HASH で計算できます。

See:

- wc_ecc_sign_hash

- wc_ecc_sig_size

Return: returns 最大署名サイズ（オクテット）*Example*

```
int sigSz = wc_ecc_sig_size_calc(32);
if ( sigSz == 0) {
    // error determining sig size
}
```

B.24.2.34 function wc_ecc_sig_size

```
int wc_ecc_sig_size(
    ecc_key * key
)
```

この関数は、次のようにして指定された ECC シグネチャの最悪の場合のサイズを返します。（KEYSZ * 2）+ SIG_HEADER_SZ + ECC_MAX_PAD_SZ。実際のシグネチャサイズは、WC_ECC_SIGN_HASH で計算できます。

See:

- wc_ecc_sign_hash
- wc_ecc_sig_size_calc

Return:

- Success 有効なキーを考えると、最大署名サイズをオクテットで返します。
- 0 与えられたキーが NULL の場合に返されます *Example*

```
int sigSz;
ecc_key key;
// initialize key, make key

sigSz = wc_ecc_sig_size(&key);
if ( sigSz == 0) {
    // error determining sig size
}
```

B.24.2.35 function wc_ecc_ctx_new

```
ecEncCtx * wc_ecc_ctx_new(
    int flags,
    WC_RNG * rng
)
```

この機能は、ECC との安全なメッセージ交換を可能にするために、新しい ECC コンテキストオブジェクトのスペースを割り当て、初期化します。

Parameters:

- **flags** これがサーバーであるかクライアントのコンテキストオプションがあるかどうかを示します.req_resp_client、および req_resp_server *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
}
```

See:

- wc_ecc_encrypt
- wc_ecc_encrypt_ex
- wc_ecc_decrypt

Return:

- Success 新しい ECENCCTX オブジェクトの生成に成功した場合は、そのオブジェクトへのポインタを返します
- NULL 関数が新しい ECENCCTX オブジェクトを生成できない場合に返されます

B.24.2.36 function wc_ecc_ctx_free

```
void wc_ecc_ctx_free(
    ecEncCtx *
)
```

この関数は、メッセージの暗号化と復号化に使用される ECENCCTX オブジェクトを解放します。

See: wc_ecc_ctx_new

Return: none 戻り値。*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);
```

B.24.2.37 function wc_ecc_ctx_reset

```
int wc_ecc_ctx_reset(
    ecEncCtx * ctx,
    WC_RNG * rng
)
```

この関数は ECENCCTX 構造をリセットして、新しいコンテキストオブジェクトを解放し、新しいコンテキストオブジェクトを割り当てます。

Parameters:

- **ctx** リセットする ECENCCTX オブジェクトへのポインタ *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
```

See: wc_ecc_ctx_new

Return:

- 0 ecencctx 構造が正常にリセットされた場合に返されます
- BAD_FUNC_ARG RNG または CTX が NULL の場合に返されます
- RNG_FAILURE_E ECC オブジェクトに新しいソルトを生成するエラーがある場合

B.24.2.38 function wc_ecc_ctx_set_algo

```
int wc_ecc_ctx_set_algo(
    ecEncCtx * ctx,
    byte encAlgo,
    byte kdfAlgo,
    byte macAlgo
)
```

この関数は、wc_ecc_ctx_new の後にオプションで呼び出されることができます。暗号化、KDF、および MAC アルゴリズムを ECENCENCCTX オブジェクトに設定します。

Parameters:

- **ctx** 情報を設定する ECENCCTX へのポインタ
- **encAlgo** 使用する暗号化アルゴリズム
- **kdfAlgo** 使用する KDF アルゴリズム *Example*

```
ecEncCtx* ctx;
// initialize ctx
if(wc_ecc_ctx_set_algo(&ctx, ecAES_128_CTR, ecHKDF_SHA256, ecHMAC_SHA256)) {
    // error setting info
}
```

See: wc_ecc_ctx_new

Return:

- 0 ECENCCTX オブジェクトの情報を正常に設定すると返されます。
- BAD_FUNC_ARG 指定された ecencctx オブジェクトが NULL の場合に返されます。

B.24.2.39 function wc_ecc_ctx_get_own_salt

```
const byte * wc_ecc_ctx_get_own_salt(
    ecEncCtx *
)
```

この関数は ECENCENCCTX オブジェクトのソルトを返します。この関数は、ECENCCTX の状態が ECSRV_INIT または ECCLI_INIT の場合にのみ呼び出す必要があります。

See:

- wc_ecc_ctx_new
- wc_ecc_ctx_set_peer_salt

Return:

- 成功すると、ecEncCtx ソルトを返します
- NULL ecencctx オブジェクトが NULL の場合、または ECENCCTX の状態が ECSRV_INIT または ECCLI_INIT でない場合に返されます。後者の 2 つのケースでは、この機能はそれぞれ ECSRV_BAD_STATE または ECCLI_BAD_STATE に ECENCCTX の状態を設定します。*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
}
```

B.24.2.40 function wc_ecc_ctx_set_peer_salt

```
int wc_ecc_ctx_set_peer_salt(
    ecEncCtx * ctx,
    const byte * salt
)
```

この関数は、ECENCENCCTX オブジェクトのピアソルトを設定します。

Parameters:

- **ctx** ソルトを設定するための ecencctx へのポインタ *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);
```

See: wc_ecc_ctx_get_own_salt

Return:

- 0 ECENCCTX オブジェクトのピアソルトの設定に成功したときに返されます。
- BAD_FUNC_ARG 指定された ecencctx オブジェクトが null または無効なプロトコルがある場合、または指定されたソルトが NULL の場合
- BAD_ENC_STATE_E ecencctx の状態が ECSRV_SALT_GET または ECCLI_SALT_GET の場合に返されます。後者の 2 つのケースでは、この機能はそれぞれ ECSRV_BAD_STATE または ECCLI_BAD_STATE に ECENCCTX の状態を設定します。

B.24.2.41 function wc_ecc_ctx_set_info

```
int wc_ecc_ctx_set_info(
    ecEncCtx * ctx,
    const byte * info,
    int sz
)
```

この関数は、wc_ecc_ctx_set_peer_salt の前後にオプションで呼び出されることができます。ECENCCTX オブジェクトのオプションの情報を設定します。

Parameters:

- **ctx** 情報を設定する ECENCCTX へのポインタ
- **info** 設定する情報を含むバッファへのポインタ *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
}
```

See: wc_ecc_ctx_new

Return:

- 0 ECENCCTX オブジェクトの情報を正常に設定すると返されます。
- BAD_FUNC_ARG 与えられた ECENCCTX オブジェクトが NULL の場合、入力情報は NULL またはサイズが無効です。

B.24.2.42 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
)
```

この関数は指定された入力メッセージを MSG から OUT に暗号化します。この関数はパラメータとしてオプションの CTX オブジェクトを取ります。提供されている場合、ECENCCTX の Encalgo、Kdfalgo、および Macalgo に基づいて暗号化が進みます。CTX が指定されていない場合、処理はデフォルトのアルゴリズム、ECAES_128_CBC、ECHKDF_SHA256、ECHMAC_SHA256 で完了します。この機能は、メッセージが CTX で指定された暗号化タイプに従って埋め込まれている必要があります。

Parameters:

- **privKey** 暗号化に使用する秘密鍵を含む ECC_KEY オブジェクトへのポインタ
- **pubKey** コミュニケーションを希望するピアの公開鍵を含む ECC_Key オブジェクトへのポインタ
- **msg** 暗号化するメッセージを保持しているバッファへのポインタ
- **msgSz** 暗号化するバッファのサイズ
- **out** 暗号化された暗号文を保存するバッファへのポインタ
- **outSz** OUT バッファ内の使用可能なサイズを含む Word32 オブジェクトへのポインタ。メッセージの暗号化に成功したら、出力バッファに書き込まれたバイト数を保持します。*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
}
```

See:

- wc_ecc_encrypt_ex
- wc_ecc_decrypt

Return:

- 0 入力メッセージの暗号化に成功したら返されます
- BAD_FUNC_ARG PRIVKEY、PUBKEY、MSG、MSGSZ、OUT、OUTSZ が NULL の場合、または CTX オブジェクトがサポートされていない暗号化タイプを指定します。
- BAD_ENC_STATE_E 指定された CTX オブジェクトが暗号化に適していない状態にある場合に返されます。
- BUFFER_E 指定された出力バッファが小さすぎて暗号化された暗号文を保存する場合に返されます
- MEMORY_E 共有秘密鍵のメモリの割り当て中にエラーがある場合に返されます

B.24.2.43 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
)
```

この関数は指定された入力メッセージを MSG から OUT に暗号化します。この関数はパラメータとしてオプションの CTX オブジェクトを取ります。提供されている場合、ECENCCTX の Encalgo、Kdfalgo、および Macalgo に基づいて暗号化が進みます。CTX が指定されていない場合、処理はデフォルトのアルゴリズム、ECAES_128_CBC、ECHKDF_SHA256、ECHMAC_SHA256 で完了します。この機能は、メッセージが CTX で指定された暗号化タイプに従って埋め込まれている必要があります。

Parameters:

- **privKey** 暗号化に使用する秘密鍵を含む ECC_KEY オブジェクトへのポインタ
- **pubKey** コミュニケーションを希望するピアの公開鍵を含む ECC_Key オブジェクトへのポインタ
- **msg** 暗号化するメッセージを保持しているバッファへのポインタ
- **msgSz** 暗号化するバッファのサイズ
- **out** 暗号化された暗号文を保存するバッファへのポインタ
- **outSz** OUT バッファ内の使用可能なサイズを含む Word32 オブジェクトへのポインタ。メッセージの暗号化に成功したら、出力バッファに書き込まれたバイト数を保持します。
- **ctx** オプション：使用するさまざまな暗号化アルゴリズムを指定する ECENCCTX オブジェクトへのポインタ *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
}
```

See:

- wc_ecc_encrypt
- wc_ecc_decrypt

Return:

- 0 入力メッセージの暗号化に成功したら返されます
- BAD_FUNC_ARG PRIVKEY、PUBKEY、MSG、MSGSZ、OUT、OUTSZ が NULL の場合、または CTX オブジェクトがサポートされていない暗号化タイプを指定します。
- BAD_ENC_STATE_E 指定された CTX オブジェクトが暗号化に適していない状態にある場合に返されます。
- BUFFER_E 指定された出力バッファが小さすぎて暗号化された暗号文を保存する場合に返されます
- MEMORY_E 共有秘密鍵のメモリの割り当て中にエラーがある場合に返されます

B.24.2.44 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
)
```

この関数は MSG から OUT への暗号文を復号化します。この関数はパラメータとしてオプションの CTX オブジェクトを取ります。提供されている場合、ECENCCTX の Encalgo、Kdfalgo、および Macalgo に基づいて暗号化が進みます。CTX が指定されていない場合、処理はデフォルトのアルゴリズム、ECAES_128_CBC、ECHKDF_SHA256、ECHMAC_SHA256 で完了します。この機能は、メッセージが CTX で指定された暗号化タイプに従って埋め込まれている必要があります。

Parameters:

- **privKey** 復号化に使用する秘密鍵を含む ECC_Key オブジェクトへのポインタ
- **pubKey** コミュニケーションを希望するピアの公開鍵を含む ECC_Key オブジェクトへのポインタ
- **msg** 暗号文を復号化するためのバッファへのポインタ
- **msgSz** 復号化するバッファのサイズ
- **out** 復号化された平文を保存するバッファへのポインタ
- **outSz** OUT バッファ内の使用可能なサイズを含む Word32 オブジェクトへのポインタ。暗号文を正常に復号化すると、出力バッファに書き込まれたバイト数を保持します。*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
}
```

See:

- wc_ecc_encrypt
- wc_ecc_encrypt_ex

Return:

- 0 入力メッセージの復号化に成功したときに返されます
- BAD_FUNC_ARG PRIVKEY、PUBKEY、MSG、MSGSZ、OUT、OUTSZ が NULL の場合、または CTX オブジェクトがサポートされていない暗号化タイプを指定します。
- BAD_ENC_STATE_E 指定された CTX オブジェクトが復号化に適していない状態にある場合に返されます。
- BUFFER_E 供給された出力バッファが小さすぎて復号化された平文を保存する場合は返されます。
- MEMORY_E 共有秘密鍵のメモリの割り当て中にエラーがある場合に返されます

B.24.2.45 function wc_ecc_set_nonblock

```
int wc_ecc_set_nonblock(
    ecc_key * key,
    ecc_nb_ctx_t * ctx
)
```

非ブロック操作のための ECC サポートを有効にします。次のビルドオプションを使用した単精度（SP）数学でサポートされています.WolfSSL_SP_SP_SMALL WOLFSSL_SP_NO_MALLOC WC_ECC_NONBLOCK

Parameters:

- **key** ECC_KEY オブジェクトへのポインタ *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);
}
```

Return: 0 コールバックコンテキストを入力メッセージに正常に設定すると返されます。

B.25 Algorithms - ED25519

B.25.1 Functions

	Name
int	**wc_ed25519_make_public**(ed25519_key * key, unsigned char * pubKey, word32 pubKeySz) この関数は Ed25519 秘密鍵から Ed25519 公開鍵を生成します。公開鍵をバッファ pubkey に出力します。この関数の呼び出しに先立ち、ed25519_key 構造体には Ed25519 秘密鍵がインポートされている必要があります。
int	**wc_ed25519_make_key**(WC_RNG * rng, int keysize, ed25519_key * key) この関数は新しい ed25519_key 構造体を生成し、それを引数 key のバッファに格納します。
int	**wc_ed25519_sign_msg**(const byte * in, word32 inlen, byte * out, word32 * outlen, ed25519_key * key) この関数は、ed25519_key 構造体を使用してメッセージに署名します。
int	**wc_ed25519ctx_sign_msg**(const byte * in, word32 inlen, byte * out, word32 * outlen, ed25519_key * key, const byte * context, byte contextLen) この関数は、ed25519_key 構造体を使用してメッセージに署名します。コンテキストは署名されるデータの一部です。
int	**wc_ed25519ph_sign_hash**(const byte * hash, word32 hashLen, byte * out, word32 * outLen, ed25519_key * key, const byte * context, byte contextLen) この関数は、ed25519_key 構造体を使用してメッセージダイジェストに署名します。コンテキストは署名されるデータの一部として含まれています。署名計算の前にメッセージは事前にハッシュされています。メッセージダイジェストを作成するために使用されるハッシュアルゴリズムは Shake-256 でなければなりません。
int	**wc_ed25519ph_sign_msg**(const byte * in, word32 inlen, byte * out, word32 * outlen, ed25519_key * key, const byte * context, byte contextLen) この関数は、ed25519_key 構造体を使用して認証を保証するメッセージに署名します。コンテキストは署名されたデータの一部として含まれています。署名計算の前にメッセージは事前にハッシュされています。
int	**wc_ed25519_verify_msg**(const byte * sig, word32 siglen, const byte * msg, word32 msgLen, int * ret, ed25519_key * key) この関数はメッセージの Ed25519 署名を検証します。ret を介して答えを返し、有効な署名の場合は 1、無効な署名の場合には 0 を返します。

	Name
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) この関数はメッセージの Ed25519 署名を検証します。コンテキストは署名されたデータの一部として含まれています。答えは変数 ret を介して返され、署名が有効ならば 1、無効ならば 0 を返します。
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) この関数は、メッセージのダイジェストの Ed25519 署名を検証します。引数 hash は、署名計算前のプリハッシュメッセージです。メッセージダイジェストを作成するために使用されるハッシュアルゴリズムは SHA-512 でなければなりません。答えは変数 ret を介して返され、署名が有効ならば 1、無効ならば 0 を返します。
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) この関数は、メッセージのダイジェストの Ed25519 署名を検証します。引数 context は検証すべきデータの一部として含まれています。検証前にメッセージがプリハッシュされています。答えは変数 res を介して返され、署名が有効ならば 1、無効ならば 0 を返します。
int	**wc_ed25519_init**(ed25519_key * key) この関数は、後のメッセージ検証で使用のために ed25519_key 構造体を初期化します。
void	**wc_ed25519_free**(ed25519_key * key) この関数は、使用済みの ed25519_key 構造体を解放します。
int	**wc_ed25519_import_public**(const byte * in, word32 inLen, ed25519_key * key) この関数はバッファから ed25519 公開鍵を ed25519_key 構造体へインポートします。圧縮あるいは非圧縮の両方の形式の鍵を扱います。
int	**wc_ed25519_import_public_ex**(const byte * in, word32 inLen, ed25519_key * key, int trusted) この関数はバッファから ed25519 公開鍵を ed25519_key 構造体へインポートします。圧縮あるいは非圧縮の両方の形式の鍵を扱います。秘密鍵が既にインポートされている場合で、trusted 引数が 1 以外の場合は両鍵が対応しているかをチェックします。
int	**wc_ed25519_import_private_only**(const byte * priv, word32 privSz, ed25519_key * key) この関数は、ed25519 秘密鍵のみをバッファからインポートします。

	Name
int	**wc_ed25519_import_private_key**(const byte * priv, word32 privSz, const byte * pub, word32 pubSz, ed25519_key * key) この関数は、Ed25519 公開鍵/秘密鍵をそれぞれ含む一対のバッファから Ed25519 鍵ペアをインポートします。この関数は圧縮と非圧縮の両方の鍵を処理します。
int	**wc_ed25519_import_private_key_ex**(const byte * priv, word32 privSz, const byte * pub, word32 pubSz, ed25519_key * key, int trusted) この関数は一対のバッファから Ed25519 公開鍵/秘密鍵ペアをインポートします。この関数は圧縮キーと非圧縮キーの両方を処理します。公開鍵は trusted 引数により信頼されていないとされた場合には秘密鍵に対して検証されます。
int	**wc_ed25519_export_public**(ed25519_key * key, byte * out, word32 * outLen) この関数は、ed25519_key 構造体から公開鍵をエクスポートします。公開鍵をバッファ out に格納し、outLen にこのバッファに書き込まれたバイトを設定します。
int	**wc_ed25519_export_private_only**(ed25519_key * key, byte * out, word32 * outLen) この関数は、ed25519_key 構造体からの秘密鍵のみをエクスポートします。秘密鍵をバッファアウトに格納し、outlen にこのバッファに書き込まれたバイトを設定します。
int	**wc_ed25519_export_private**(ed25519_key * key, byte * out, word32 * outLen) この関数は、ed25519_key 構造体から鍵ペアをエクスポートします。鍵ペアをバッファ out に格納し、ounteren でこのバッファに書き込まれたバイトを設定します。
int	**wc_ed25519_export_key**(ed25519_key * key, byte * priv, word32 * privSz, byte * pub, word32 * pubSz) この関数は、ed25519_key 構造体から秘密鍵と公開鍵を別々にエクスポートします。秘密鍵をバッファ priv に格納し、priovSz にこのバッファに書き込んだバイト数を設定します。公開鍵をバッファ pub に格納し、pubSz にこのバッファに書き込んだバイト数を設定します。
int	**wc_ed25519_check_key**(ed25519_key * key) この関数は、ed25519_key 構造体の公開鍵をチェックします。
int	**wc_ed25519_size**(ed25519_key * key) この関数は、Ed25519 - 32 バイトのサイズを返します。
int	**wc_ed25519_priv_size**(ed25519_key * key) この関数は、秘密鍵サイズ（secret + public）をバイト単位で返します。
int	**wc_ed25519_pub_size**(ed25519_key * key) この関数は圧縮鍵サイズをバイト単位で返します（公開鍵）。

	Name
int	**wc_ed25519_sig_size**(ed25519_key * key) この関数は、ED25519 シグネチャのサイズ（バイト数 64）を返します。

B.25.2 Functions Documentation

B.25.2.1 function wc_ed25519_make_public

```
int wc_ed25519_make_public(
    ed25519_key * key,
    unsigned char * pubKey,
    word32 pubKeySz
)
```

この関数は Ed25519 秘密鍵から Ed25519 公開鍵を生成します。公開鍵をバッファ pubkey に出力します。この関数の呼び出しに先立ち、ed25519_key 構造体には Ed25519 秘密鍵がインポートされている必要があります。

Parameters:

- **key** Ed25519 秘密鍵がインポートされている ed25519_key 構造体へのポインタ。
- **pubKey** 公開鍵を出力するバッファへのポインタ。
- **pubKeySz** バッファのサイズ。常に ED25519_PUB_KEY_SIZE(32) でなければなりません。

See:

- wc_ed25519_init
- wc_ed25519_import_private_only
- wc_ed25519_make_key

Return:

- 0 公開鍵の作成に成功したときに返されます。
- BAD_FUNC_ARG 引数 key または pubKey が NULL の場合、または指定された鍵サイズが 32 バイトではない場合（ED25519 に 32 バイトのキーがあります）。
- ECC_PRIV_KEY_E ed25519_key 構造体に Ed25519 秘密鍵がインポートされていない場合に返されます。
- MEMORY_E 関数の実行中にメモリを割り当てエラーがある場合に返されます。

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
}
```

B.25.2.2 function wc_ed25519_make_key

```
int wc_ed25519_make_key(
    WC_RNG * rng,
    int keysize,
    ed25519_key * key
)
```

この関数は新しい ed25519_key 構造体を生成し、それを引数 key のバッファに格納します。

Parameters:

- **rng** RNG キーを生成する初期化された RNG オブジェクトへのポインタ。
- **keysize** key の長さ。ED25519 の場合は常に 32 になります。

See: wc_ed25519_init

Return:

- 0 ed25519_key 構造体を正常に生成すると返されます。
- BAD_FUNC_ARG RNG または KEY が NULL に評価された場合、または指定された keysize が 32 バイトではない場合 (Ed25519 鍵には常に 32 バイトを指定する必要があります)。
- MEMORY_E 関数の実行中にメモリ割り当てエラーが発生した場合に返されます。

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
}
```

B.25.2.3 function wc_ed25519_sign_msg

```
int wc_ed25519_sign_msg(
    const byte * in,
    word32 inlen,
    byte * out,
    word32 * outlen,
    ed25519_key * key
)
```

この関数は、ed25519_key 構造体を使用してメッセージに署名します。

Parameters:

- **in** 署名するメッセージを含むバッファへのポインタ。
- **inlen** 署名するメッセージのサイズ
- **out** 生成された署名を格納するためのバッファ。
- **outlen** 出力バッファの最大長。メッセージ署名の生成に成功したときに、書き込まれたバイト数を保持します。
- **key** 署名を生成するために使用する秘密鍵を保持している ed25519_key 構造体へのポインタ。

See:

- wc_ed25519ctx_sign_msg
- wc_ed25519ph_sign_hash
- wc_ed25519ph_sign_msg
- wc_ed25519_verify_msg

Return:

- 0 メッセージの署名を正常に生成すると返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL に評価された場合、または出力バッファが小さすぎて生成された署名を保存する場合は返されます。
- MEMORY_E 関数の実行中にメモリ割り当てエラーが発生した場合に返されます。

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
}
```

B.25.2.4 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
)
```

この関数は、ed25519_key 構造体を使用してメッセージに署名します。コンテキストは署名されるデータの一部です。

Parameters:

- **in** 署名するメッセージを含むバッファへのポインタ。
- **inlen** 署名するメッセージのサイズ
- **out** 生成された署名を格納するためのバッファ。
- **outlen** 出力バッファの最大長。メッセージ署名の生成に成功したときに、書き込まれたバイトを保存します。
- **key** 署名を生成するために使用する秘密鍵を保持している ed25519_key 構造体へのポインタ。
- **context** メッセージが署名されているコンテキストを含むバッファへのポインタ。
- **contextLen** コンテキストバッファのサイズ

See:

- wc_ed25519_sign_msg

- wc_ed25519ph_sign_hash
- wc_ed25519ph_sign_msg
- wc_ed25519_verify_msg

Return:

- 0 メッセージの署名を正常に生成すると返されます。
- BAD_FUNC_ARG 返された入力パラメータは NULL に評価されます。出力バッファが小さすぎて生成された署名を保存するには小さすぎます。
- MEMORY_E 関数の実行中にメモリ割り当てエラーが発生した場合に返されます。

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
}
```

B.25.2.5 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
)
```

この関数は、ed25519_key 構造体を使用してメッセージダイジェストに署名します。コンテキストは署名されるデータの一部として含まれています。署名計算の前にメッセージは事前にハッシュされています。メッセージダイジェストを作成するために使用されるハッシュアルゴリズムは Shake-256 でなければなりません。

Parameters:

- **hash** 署名するメッセージのハッシュを含むバッファへのポインタ。
- **hashLen** 署名するメッセージのハッシュのサイズ
- **out** 生成された署名を格納するためのバッファ。
- **outlen** 出力バッファの最大長。メッセージ署名の生成に成功したときに、書き込まれたバイトを保存します。
- **key** 署名を生成するのに使用する秘密鍵を含んだ ed25519_key 構造体へのポインタ。
- **context** メッセージが署名されているコンテキストを含むバッファへのポインタ。
- **contextLen** コンテキストバッファのサイズ

See:

- wc_ed25519_sign_msg
- wc_ed25519ctx_sign_msg
- wc_ed25519ph_sign_msg
- wc_ed25519_verify_msg

Return:

- 0 メッセージダイジェストの署名を正常に生成すると返されます。
- BAD_FUNC_ARG 返された入力パラメータは NULL に評価されます。出力バッファが小さすぎて生成された署名を保存するには小さすぎます。
- MEMORY_E 関数の実行中にメモリ割り当てエラーが発生した場合に返されます。

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
}
```

B.25.2.6 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
)
```

この関数は、ed25519_key 構造体を使用して認証を保証するメッセージに署名します。コンテキストは署名されたデータの一部として含まれています。署名計算の前にメッセージは事前にハッシュされています。

Parameters:

- **in** 署名するメッセージを含むバッファへのポインタ。
- **inlen** 署名するメッセージのインレル長。
- **out** 生成された署名を格納するためのバッファ。
- **outlen** 出力バッファの最大長。メッセージ署名の生成に成功したときに、書き込まれたバイトを保存します。
- **key** 署名を生成するプライベート ed25519_key 構造体へのポインタ。
- **context** メッセージが署名されているコンテキストを含むバッファへのポインタ。

- **contextLen** コンテキストバッファのサイズ

See:

- wc_ed25519_sign_msg
- wc_ed25519ctx_sign_msg
- wc_ed25519ph_sign_hash
- wc_ed25519_verify_msg

Return:

- 0 メッセージの署名を正常に生成すると返されます。
- BAD_FUNC_ARG 返された入力パラメータは NULL に評価されます。出力バッファが小さすぎて生成された署名を保存するには小さすぎます。
- MEMORY_E 関数の実行中にメモリを割り当てエラーが発生した場合に返されます。

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
}
```

B.25.2.7 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
)
```

この関数はメッセージの Ed25519 署名を検証します。ret を介して答えを返し、有効な署名の場合は 1、無効な署名の場合には 0 を返します。

Parameters:

- **sig** 検証するシグネチャを含むバッファへのポインタ。
- **siglen** 検証するシグネチャのサイズ
- **msg** メッセージを含むバッファへのポインタ
- **msgLen** 検証するメッセージのサイズ
- **ret** 検証の結果を格納する変数へのポインタ。1 はメッセージが正常に検証されたことを示します。
- **key** 署名を検証するための Ed25519 公開鍵へのポインタ。

See:

- wc_ed25519ctx_verify_msg
- wc_ed25519ph_verify_hash
- wc_ed25519ph_verify_msg
- wc_ed25519_sign_msg

Return:

- 0 署名検証と認証を正常に実行したときに返されます。
- BAD_FUNC_ARG いずれかの入力パラメータが NULL に評価された場合、または SIGLEN が署名の実際の長さと一致しない場合に返されます。
- SIG_VERIFY_E 検証が完了した場合は返されますが、生成された署名は提供された署名と一致しません。

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
}
```

B.25.2.8 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
)
```

この関数はメッセージの Ed25519 署名を検証します。コンテキストは署名されたデータの一部として含まれています。答えは変数 ret を介して返され、署名が有効ならば 1、無効ならば 0 を返します。

Parameters:

- **sig** 検証するシグネチャを含むバッファへのポインタ。
- **siglen** 検証するシグネチャのサイズ
- **msg** メッセージを含むバッファへのポインタ
- **msgLen** 検証するメッセージのサイズ
- **ret** 検証の結果を格納する変数へのポインタ。1 はメッセージが正常に検証されたことを示します。
- **key** 署名を検証するための Ed25519 公開鍵へのポインタ。
- **context** メッセージが署名されているコンテキストを含むバッファへのポインタ。
- **contextLen** コンテキストバッファのサイズ

See:

- wc_ed25519_verify_msg
- wc_ed25519ph_verify_hash
- wc_ed25519ph_verify_msg
- wc_ed25519_sign_msg

Return:

- 0 署名検証と認証を正常に実行したときに返されます。
- BAD_FUNC_ARG いずれかの入力パラメータが NULL に評価された場合、または SIGLEN が署名の実際の長さと一致しない場合に返されます。
- SIG_VERIFY_E 検証が完了した場合は返されますが、生成された署名は提供された署名と一致しません。

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
}
```

B.25.2.9 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
)
```

この関数は、メッセージのダイジェストの Ed25519 署名を検証します。引数 hash は、署名計算前のプリハッシュメッセージです。メッセージダイジェストを作成するために使用されるハッシュアルゴリズムは SHA-512 でなければなりません。答えは変数 ret を介して返され、署名が有効ならば 1、無効ならば 0 を返します。

Parameters:

- **sig** 検証するシグネチャを含むバッファへのポインタ。
- **siglen** 検証するシグネチャのサイズ
- **msg** メッセージを含むバッファへのポインタ
- **msgLen** 検証するメッセージのサイズ
- **ret** 検証の結果を格納する変数へのポインタ。1 はメッセージが正常に検証されたことを示します。
- **key** 署名を検証するための Ed25519 公開鍵へのポインタ。
- **context** メッセージが署名されたコンテキストを含むバッファへのポインタ。
- **contextLen** コンテキストのサイズ

See:

- wc_ed25519_verify_msg
- wc_ed25519ctx_verify_msg
- wc_ed25519ph_verify_msg
- wc_ed25519_sign_msg

Return:

- 0 署名検証と認証を正常に実行したときに返されます。
- BAD_FUNC_ARG いずれかの入力パラメータが NULL に評価された場合、または SIGLEN が署名の実際の長さと一致しない場合に返されます。
- SIG_VERIFY_E 検証が完了した場合は返されますが、生成された署名は提供された署名と一致しません。

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
}
```

B.25.2.10 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
)
```

この関数は、メッセージのダイジェストの Ed25519 署名を検証します。引数 context は検証すべきデータの一部として含まれています。検証前にメッセージがプリハッシュされています。答えは変数 res を介して返され、署名が有効ならば 1、無効ならば 0 を返します。

Parameters:

- **sig** 検証するシグネチャを含むバッファへのポインタ。
- **siglen** 検証するシグネチャのサイズ
- **msg** メッセージを含むバッファへのポインタ
- **msgLen** 検証するメッセージのサイズ
- **ret** 検証の結果を格納する変数へのポインタ。1 はメッセージが正常に検証されたことを示します。
- **key** 署名を検証するための Ed25519 公開鍵へのポインタ。
- **context** メッセージが署名されたコンテキストを含むバッファへのポインタ。

- **contextLen** コンテキストのサイズ

See:

- wc_ed25519_verify_msg
- wc_ed25519ph_verify_hash
- wc_ed25519ph_verify_msg
- wc_ed25519_sign_msg

Return:

- 0 署名検証と認証を正常に実行したときに返されます。
- BAD_FUNC_ARG いずれかの入力パラメータが NULL に評価された場合、または SIGLEN が署名の実際の長さと一致しない場合に返されます。
- SIG_VERIFY_E 検証が完了した場合は返されますが、生成された署名は提供された署名と一致しません。

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
}
```

B.25.2.11 function wc_ed25519_init

```
int wc_ed25519_init(
    ed25519_key * key
)
```

この関数は、後のメッセージ検証で使用のために ed25519_key 構造体を初期化します。

Parameters:

- **key** ed25519_key 構造体へのポインタ

See:

- wc_ed25519_make_key
- wc_ed25519_free

Return:

- 0 ed25519_key 構造体の初期化に成功したときに返されます。
- BAD_FUNC_ARG 引数 key が NULL の場合に返されます。

Example

```
ed25519_key key;
wc_ed25519_init(&key);
```

B.25.2.12 function wc_ed25519_free

```
void wc_ed25519_free(
    ed25519_key * key
)
```

この関数は、使用済みの ed25519_key 構造体を解放します。

Parameters:

- **key** ed25519_key 構造体へのポインタ

See: wc_ed25519_init

Example

```
ed25519_key key;
// initialize key and perform secure exchanges
...
wc_ed25519_free(&key);
```

B.25.2.13 function wc_ed25519_import_public

```
int wc_ed25519_import_public(
    const byte * in,
    word32 inLen,
    ed25519_key * key
)
```

この関数はバッファから ed25519 公開鍵を ed25519_key 構造体へインポートします。圧縮あるいは非圧縮の両方の形式の鍵を扱います。

Parameters:

- **in** 公開鍵を含んだバッファへのポインタ
- **inLen** 公開鍵を含んだバッファのサイズ
- **key** ed25519_key 構造体へのポインタ

See:

- wc_ed25519_import_public_ex
- wc_ed25519_import_private_key
- wc_ed25519_import_private_key_ex
- wc_ed25519_export_public

Return:

- 0 ed25519 公開鍵のインポートに成功した場合に返されます。
- BAD_FUNC_ARG in または key が null に評価された場合、または inlen が ED25519 鍵のサイズよりも小さい場合に返されます。

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
}
```

B.25.2.14 function wc_ed25519_import_public_ex

```
int wc_ed25519_import_public_ex(
    const byte * in,
    word32 inLen,
    ed25519_key * key,
    int trusted
)
```

この関数はバッファから ed25519 公開鍵を ed25519_key 構造体へインポートします。圧縮あるいは非圧縮の両方の形式の鍵を扱います。秘密鍵が既にインポートされている場合で、trusted 引数が 1 以外の場合は両鍵が対応しているかをチェックします。

Parameters:

- **in** 公開鍵を含んだバッファへのポインタ
- **inLen** 公開鍵を含んだバッファのサイズ
- **key** ed25519_key 構造体へのポインタ
- **trusted** 公開鍵が信頼おけるか否かを示すフラグ

See:

- wc_ed25519_import_public
- wc_ed25519_import_private_key
- wc_ed25519_import_private_key_ex
- wc_ed25519_export_public

Return:

- 0 ed25519 公開鍵のインポートに成功した場合に返されます。
- BAD_FUNC_ARG Returned 引数 in あるいは key が NULL の場合, あるいは引数 inLen が Ed25519 鍵のサイズより小さい場合に返されます。

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
}
```

B.25.2.15 function wc_ed25519_import_private_only

```
int wc_ed25519_import_private_only(
    const byte * priv,
    word32 privSz,
    ed25519_key * key
)
```

この関数は、ed25519 秘密鍵のみをバッファからインポートします。

Parameters:

- **priv** 秘密鍵を含むバッファへのポインタ。
- **privSz** 秘密鍵を含むバッファのサイズ

See:

- wc_ed25519_import_public
- wc_ed25519_import_private_key
- wc_ed25519_export_private_only

Return:

- 0 Ed25519 秘密鍵のインポートに成功した際に返されます。
- BAD_FUNC_ARG priv または key が NULL に評価された場合、または privSz が ED25519_KEY_SIZE と異なる場合に返されます。

Example

```
int ret;
byte priv[] = { initialize with 32 byte private key };

ed25519_key key;
wc_ed25519_init_key(&key);
ret = wc_ed25519_import_private_key(priv, sizeof(priv), &key);
if (ret != 0) {
    // error importing private key
}
```

B.25.2.16 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
)
```

この関数は、Ed25519 公開鍵/秘密鍵をそれぞれ含む一対のバッファから Ed25519 鍵ペアをインポートします。この関数は圧縮と非圧縮の両方の鍵を処理します。

Parameters:

- **priv** 秘密鍵を含むバッファへのポインタ。
- **privSz** 秘密鍵バッファのサイズ
- **pub** 公開鍵を含むバッファへのポインタ。
- **pubSz** 公開鍵バッファのサイズ

See:

- wc_ed25519_import_public
- wc_ed25519_import_private_only
- wc_ed25519_export_private

Return:

- 0 Ed25519_KEY のインポートに成功しました。
- BAD_FUNC_ARG priv または key が NULL に評価された場合、privSz が ED25519_KEY_SIZE と異なるあるいは ED25519_PRV_KEY_SIZE とも異なる場合、pubSz が ED25519_PUB_KEY_SIZE よりも小さい場合に返されます。

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
}
```

B.25.2.17 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
)
```

この関数は一対のバッファから Ed25519 公開鍵/秘密鍵ペアをインポートします。この関数は圧縮キーと非圧縮キーの両方を処理します。公開鍵は trusted 引数により信頼されていないとされた場合には秘密鍵に対して検証されます。

Parameters:

- **priv** 秘密鍵を保持するバッファへのポインタ
- **privSz** 秘密鍵バッファのサイズ
- **pub** 公開鍵を保持するバッファへのポインタ
- **pubSz** 公開鍵バッファのサイズ
- **key** インポートされた公開鍵/秘密鍵を保持する ed25519_key オブジェクトへのポインター
- **trusted** 公開鍵が信頼できるか否かを指定するフラグ

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 ed25519_key のインポートに成功しました。
- BAD_FUNC_ARG Returned if priv あるいは key が NULL に評価された場合、privSz が ED25519_KEY_SIZE とも ED25519_PRV_KEY_SIZE とも異なる場合、pubSz が ED25519_PUB_KEY_SIZE より小さい場合に返されます。

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
}
```

B.25.2.18 function wc_ed25519_export_public

```
int wc_ed25519_export_public(
    ed25519_key * key,
    byte * out,
    word32 * outLen
)
```

この関数は、ed25519_key 構造体から公開鍵をエクスポートします。公開鍵をバッファ out に格納し、outLen にこのバッファに書き込まれたバイトを設定します。

Parameters:

- **key** 公開鍵をエクスポートするための ed25519_key 構造体へのポインタ。
- **out** 公開鍵を保存するバッファへのポインタ。
- **outLen** 公開鍵を出力する先のバッファサイズを格納する word32 型変数へのポインタ。入力の際はバッファサイズを格納して渡し、出力の際はエクスポートした公開鍵のサイズを格納します。

See:

- wc_ed25519_import_public
- wc_ed25519_export_private_only

Return:

- 0 公開鍵のエクスポートに成功したら返されます。
- BAD_FUNC_ARG いずれかの入力値が NULL に評価された場合に返されます。
- BUFFER_E 提供されたバッファーが公開鍵を保存するのに十分な大きさでない場合に返されます。このエラーを返すと、outlen に必要なサイズを設定します。

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
}
```

B.25.2.19 function wc_ed25519_export_private_only

```
int wc_ed25519_export_private_only(
    ed25519_key * key,
    byte * out,
    word32 * outLen
)
```

この関数は、ed25519_key 構造体からの秘密鍵のみをエクスポートします。秘密鍵をバッファアウトに格納し、outlen にこのバッファに書き込まれたバイトを設定します。

Parameters:

- **key** 秘密鍵をエクスポートするための ed25519_key 構造体へのポインタ。
- **out** 秘密鍵を保存するバッファへのポインタ。
- **outLen** 秘密鍵を出力する先のバッファサイズを格納する word32 型変数へのポインタ。入力の際はバッファサイズを格納して渡し、出力の際はエクスポートした秘密鍵のサイズを格納します。

See:

- wc_ed25519_export_public
- wc_ed25519_import_private_key

Return:

- 0 秘密鍵のエクスポートに成功したら返されます。
- BAD_FUNC_ARG いずれかの入力値が NULL に評価された場合に返されます。
- BUFFER_E 提供されたバッファーが秘密鍵を保存するのに十分な大きさでない場合に返されます。

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
}
```

B.25.2.20 function wc_ed25519_export_private

```
int wc_ed25519_export_private(
    ed25519_key * key,
    byte * out,
    word32 * outLen
)
```

この関数は、ed25519_key 構造体から鍵ペアをエクスポートします。鍵ペアをバッファ out に格納し、ounteren でこのバッファに書き込まれたバイトを設定します。

Parameters:

- **鍵ペアをエクスポートするための ed25519_key 構造体へのポインタ。**
- **鍵ペアを保存するバッファへのポインタ。**
- **outLen** 鍵ペアを出力する先のバッファサイズを格納する word32 型変数へのポインタ。入力の際はバッファサイズを格納して渡し、出力の際はエクスポートした鍵ペアのサイズを格納します。

See:

- wc_ed25519_import_private_key
- wc_ed25519_export_private_only

Return:

- 0 鍵ペアのエクスポートに成功したら返されます。
- BAD_FUNC_ARG いずれかの入力値が NULL に評価された場合に返されます。
- BUFFER_E 提供されているバッファーが鍵ペアを保存するのに十分な大きさでない場合に返されます。

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
}
```

B.25.2.21 function wc_ed25519_export_key

```
int wc_ed25519_export_key(
    ed25519_key * key,
    byte * priv,
    word32 * privSz,
    byte * pub,
    word32 * pubSz
)
```

この関数は、ed25519_key 構造体から秘密鍵と公開鍵を別々にエクスポートします。秘密鍵をバッファ priv に格納し、priovSz にこのバッファに書き込んだバイト数を設定します。公開鍵をバッファ pub に格納し、pubSz にこのバッファに書き込んだバイト数を設定します。

Parameters:

- **key** 鍵ペアをエクスポートするための ed25519_key 構造体へのポインタ。
- **priv** 秘密鍵を出力するバッファへのポインタ。
- **privSz** 秘密鍵を出力する先のバッファのサイズを保持する word32 型変数へのポインタ。秘密鍵のエクスポート後には書き込まれたバイト数がセットされます。
- **pub** パブリックキーを出力するバッファへのポインタ
- **pubSz** 公開鍵を出力する先のバッファのサイズを保持する word32 型変数へのポインタ。公開鍵のエクスポート後には書き込まれたバイト数がセットされます。

See:

- wc_ed25519_export_private
- wc_ed25519_export_public

Return:

- 0 鍵ペアのエクスポートに成功したら返されます。
- BAD_FUNC_ARG いずれかの入力値が NULL に評価された場合に返されます。
- BUFFER_E 提供されているバッファが鍵ペアを保存するのに十分な大きさでない場合に返されます。

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
}
```

B.25.2.22 function wc_ed25519_check_key

```
int wc_ed25519_check_key(
    ed25519_key * key
)
```

この関数は、ed25519_key 構造体の公開鍵をチェックします。

Parameters:

- **key** 公開鍵と秘密鍵の両方を保持している ed25519_key 構造体へのポインタ

See: wc_ed25519_import_private_key

Return:

- 0 プライベートキーと公開鍵が一致した場合に返されます。
- BAD_FUNC_ARG 与えられた鍵が NULL の場合に返されます。
- PUBLIC_KEY_E 公開鍵が参照できないか無効の場合に返されます。

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(priv, sizeof(priv), pub, sizeof(pub), &key);
ret = wc_ed25519_check_key(&key);
if (ret != 0) {
    // error checking key
}
```

B.25.2.23 function wc_ed25519_size

```
int wc_ed25519_size(
    ed25519_key * key
)
```

この関数は、Ed25519 - 32 バイトのサイズを返します。

Parameters:

- **key** ed25519_key 構造体へのポインタ

See: wc_ed25519_make_key

Return:

- ED25519_KEY_SIZE 有効な秘密鍵のサイズ（32 バイト）。
- BAD_FUNC_ARG 与えられた引数 key が NULL の場合に返されます。

Example

```
int keySz;
ed25519_key key;
// initialize key, make key
keySz = wc_ed25519_size(&key);
if (keySz == 0) {
    // error determining key size
}
```

B.25.2.24 function wc_ed25519_priv_size

```
int wc_ed25519_priv_size(
    ed25519_key * key
)
```

この関数は、秘密鍵サイズ（secret + public）をバイト単位で返します。

Parameters:

- **key** ed25519_key 構造体へのポインタ

See: wc_ed25519_pub_size

Return:

- ED25519_PRV_KEY_SIZE 秘密鍵のサイズ（64 バイト）。
- BAD_FUNC_ARG key 引数が null の場合に返されます。

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);
```

B.25.2.25 function wc_ed25519_pub_size

```
int wc_ed25519_pub_size(
    ed25519_key * key
)
```

この関数は圧縮鍵サイズをバイト単位で返します（公開鍵）。

Parameters:

- **key** ed25519_key 構造体へのポインタ

See: wc_ed25519_priv_size

Return:

- ED25519_PUB_KEY_SIZE 圧縮公開鍵のサイズ（32 バイト）。
- BAD_FUNC_ARG key 引数が null の場合は返します。

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);
```

B.25.2.26 function wc_ed25519_sig_size

```
int wc_ed25519_sig_size(
    ed25519_key * key
)
```

この関数は、ED25519 シグネチャのサイズ（バイト数 64）を返します。

Parameters:

- **key** ed25519_key 構造体へのポインタ

See: wc_ed25519_sign_msg

Return:

- ED25519_SIG_SIZE ED25519 シグネチャ（64 バイト）のサイズ。
- BAD_FUNC_ARG key 引数が null の場合は返します。

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.26.1 Functions

	Name
int	**wc_ed448_make_public**(ed448_key * key, unsigned char * pubKey, word32 pubKeySz) この関数は、秘密鍵から ED448 公開鍵を生成します。公開鍵をバッファ Pubkey に格納し、Pubkeysz でこのバッファに書き込まれたバイトを設定します。
int	**wc_ed448_make_key**(WC_RNG * rng, int keysize, ed448_key * key) この関数は新しい ED448 キーを生成し、それをキーに格納します。
int	**wc_ed448_sign_msg**(const byte * in, word32 inlen, byte * out, word32 * outlen, ed448_key * key) この関数は、ED448_Key オブジェクトを使用したメッセージに正解を保証します。

	Name
int	**wc_ed448ph_sign_hash**(const byte * hash, word32 hashLen, byte * out, word32 * outLen, ed448_key * key, const byte * context, byte contextLen) この関数は、Ed448_Key オブジェクトを使用してメッセージダイジェストに署名して信頼性を保証します。コンテキストは署名されたデータの一部として含まれています。ハッシュは、署名計算前のプリハッシュメッセージです。メッセージダイジェストを作成するために使用されるハッシュアルゴリズムは Shake-256 でなければなりません。
int	**wc_ed448ph_sign_msg**(const byte * in, word32 inLen, byte * out, word32 * outLen, ed448_key * key, const byte * context, byte contextLen) この関数は、ED448_Key オブジェクトを使用したメッセージに正解を保証します。コンテキストは署名されたデータの一部として含まれています。署名計算の前にメッセージは事前にハッシュされています。
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) この関数は、メッセージの ED448 署名を確認して信頼性を確保します。文脈はデータ検証済みの一部として含まれています。答えは RES を介して返され、有効な署名に対応する 1、無効な署名に対応する 0 を返します。
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) この関数は、メッセージのダイジェストの ED448 シグネチャを検証して、信頼性を確保します。文脈はデータ検証済みの一部として含まれています。ハッシュは、署名計算前のプリハッシュメッセージです。メッセージダイジェストを作成するために使用されるハッシュアルゴリズムは Shake-256 でなければなりません。答えは RES を介して返され、有効な署名に対応する 1、無効な署名に対応する 0 を返します。
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) この関数は、メッセージの ED448 署名を確認して信頼性を確保します。文脈はデータ検証済みの一部として含まれています。検証前にメッセージがプリハッシュされています。答えは RES を介して返され、有効な署名に対応する 1、無効な署名に対応する 0 を返します。
int	**wc_ed448_init**(ed448_key * key) この関数は、メッセージ検証で将来の使用のために ED448_Key オブジェクトを初期化します。

	Name
void	**wc_ed448_free**(ed448_key * key) この関数は、それが使用された後に ED448 オブジェクトを解放します。*Example*
int	**wc_ed448_import_public**(const byte * in, word32 inLen, ed448_key * key) この関数は、公開鍵を含むバッファから Public ED448_Key ペアをインポートします。この関数は圧縮キーと非圧縮キーの両方を処理します。
int	**wc_ed448_import_private_only**(const byte * priv, word32 privSz, ed448_key * key) この関数は、ed448 秘密鍵をバッファからのみインポートします。
int	**wc_ed448_import_private_key**(const byte * priv, word32 privSz, const byte * pub, word32 pubSz, ed448_key * key) この関数は、一対のバッファからパブリック/プライベート ED448 キーペアをインポートします。この関数は圧縮キーと非圧縮キーの両方を処理します。
int	**wc_ed448_export_public**(ed448_key * key, byte * out, word32 * outLen) この関数は、ED448_Key 構造体から秘密鍵をエクスポートします。公開鍵をバッファアウトに格納し、ounteren でこのバッファに書き込まれたバイトを設定します。
int	**wc_ed448_export_private_only**(ed448_key * key, byte * out, word32 * outLen) この関数は、ED448_Key 構造体からの秘密鍵のみをエクスポートします。秘密鍵をバッファアウトに格納し、outlen にこのバッファに書き込まれたバイトを設定します。
int	**wc_ed448_export_private**(ed448_key * key, byte * out, word32 * outLen) この関数は、ED448_Key 構造体からキーペアをエクスポートします。キーペアをバッファ OUT に格納し、ounteren でこのバッファに書き込まれたバイトを設定します。
int	**wc_ed448_export_key**(ed448_key * key, byte * priv, word32 * privSz, byte * pub, word32 * pubSz) この関数は、ED448_Key 構造体とは別にプライベートキーと公開鍵をエクスポートします。秘密鍵をバッファー Priv に格納し、PRIVSZ でこのバッファに書き込まれたバイトを設定します。公開鍵をバッファ PUB に格納し、Pubsz でこのバッファに書き込まれたバイトを設定します。
int	**wc_ed448_check_key**(ed448_key * key) この関数は、ed448_key 構造体の公開鍵をチェックします。
int	**wc_ed448_size**(ed448_key * key) この関数は、ED448 秘密鍵のサイズ - 57 バイトを返します。
int	**wc_ed448_priv_size**(ed448_key * key) この関数は、秘密鍵サイズ（secret + public）をバイト単位で返します。

	Name
int	**wc_ed448_pub_size**(ed448_key * key) この関数は圧縮鍵サイズをバイト単位で返します（公開鍵）。
int	**wc_ed448_sig_size**(ed448_key * key) この関数は、ED448 シグネチャのサイズ（バイト数 114）を返します。

B.26.2 Functions Documentation

B.26.2.1 function wc_ed448_make_public

```
int wc_ed448_make_public(
    ed448_key * key,
    unsigned char * pubKey,
    word32 pubKeySz
)
```

この関数は、秘密鍵から ED448 公開鍵を生成します。公開鍵をバッファ Pubkey に格納し、Pubkeysz でこのバッファに書き込まれたバイトを設定します。

Parameters:

- **キーを生成する ED448_Key へのキーポインタ。**
- **公開鍵を保存するバッファへのポインタ。** *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
}
```

See:

- wc_ed448_init
- wc_ed448_import_private_only
- wc_ed448_make_key

Return:

- 0 公開鍵の作成に成功したときに返されます。
- BAD_FUNC_ARG IFI キーまたは PubKey が NULL に評価された場合、または指定されたキーサイズが 57 バイトではない場合（ED448 には 57 バイトのキーがあります）。
- MEMORY_E 関数の実行中にメモリを割り当てるエラーがある場合に返されます。

B.26.2.2 function wc_ed448_make_key

```
int wc_ed448_make_key(
    WC_RNG * rng,
    int keysize,
```

```
    ed448_key * key
)
```

この関数は新しい ED448 キーを生成し、それをキーに格納します。

Parameters:

- **RNG キーを生成する初期化された RNG オブジェクトへのポインタ。**
- **keysize** key の長さを生成します。ED448 の場合は常に 57 になります。*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
}
```

See: wc_ed448_init

Return:

- 0 ED448_Key を正常に作成したときに返されます。
- BAD_FUNC_ARG RNG または Key が NULL に評価された場合、または指定されたキーサイズが 57 バイトではない場合（ED448 には 57 バイトのキーがあります）。
- MEMORY_E 関数の実行中にメモリを割り当てるエラーがある場合に返されます。

B.26.2.3 function wc_ed448_sign_msg

```
int wc_ed448_sign_msg(
    const byte * in,
    word32 inlen,
    byte * out,
    word32 * outlen,
    ed448_key * key
)
```

この関数は、ED448_Key オブジェクトを使用したメッセージに正解を保証します。

Parameters:

- **署名するメッセージを含むバッファへのポインタ。**
- **署名するメッセージのインレル長。**
- **生成された署名を格納するためのバッファー。**
- **出力バッファの最大長の範囲内。メッセージ署名の生成に成功したときに、書き込まれたバイトを保存します。** *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
}
```

See:

- wc_ed448ph_sign_hash
- wc_ed448ph_sign_msg
- wc_ed448_verify_msg

Return:

- 0 メッセージの署名を正常に生成すると返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL に評価された場合、または出力バッファが小さすぎて生成された署名を保存する場合は返されます。
- MEMORY_E 関数の実行中にメモリを割り当てるエラーがある場合に返されます。

B.26.2.4 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
)
```

この関数は、Ed448_Key オブジェクトを使用してメッセージダイジェストに署名して信頼性を保証します。コンテキストは署名されたデータの一部として含まれています。ハッシュは、署名計算前のプリハッシュメッセージです。メッセージダイジェストを作成するために使用されるハッシュアルゴリズムは Shake-256 でなければなりません。

Parameters:

- **サインへのメッセージのハッシュを含むバッファへのハッシュポインタ。**
- **サインへのメッセージのハッシュのハッシュの長さ。**
- **生成された署名を格納するためのバッファー。**
- **出力バッファの最大長の範囲内。メッセージ署名の生成に成功したときに、書き込まれたバイトを保存します。**
- **署名を生成するためのプライベート ED448_Key へのキーポインタ。**
- **メッセージが署名されているコンテキストを含むバッファへのコンテキストポインタ。** *Example*

```
ed448_key key;
WC_RNG rng;
int ret, sigSz;

byte sig[114]; // will hold generated signature
sigSz = sizeof(sig);
byte hash[] = { initialize with SHAKE-256 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
}
```

See:

- wc_ed448_sign_msg
- wc_ed448ph_sign_msg
- wc_ed448ph_verify_hash

Return:

- 0 メッセージダイジェストの署名を正常に生成すると返されます。
- BAD_FUNC_ARG 返された入力パラメータは NULL に評価されます。出力バッファが小さすぎて生成された署名を保存するには小さすぎます。
- MEMORY_E 関数の実行中にメモリを割り当てるエラーがある場合に返されます。

B.26.2.5 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
)
```

この関数は、ED448_Key オブジェクトを使用したメッセージに正解を保証します。コンテキストは署名されたデータの一部として含まれています。署名計算の前にメッセージは事前にハッシュされています。

Parameters:

- **署名するメッセージを含むバッファへのポインタ。**
- **署名するメッセージのインレル長。**
- **生成された署名を格納するためのバッファー。**
- **出力バッファの最大長の範囲内。メッセージ署名の生成に成功したときに、書き込まれたバイトを保存します。**
- **署名を生成するためのプライベート ED448_Key へのキーポインタ。**
- **メッセージが署名されているコンテキストを含むバッファへのコンテキストポインタ。** *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
}
```

See:

- wc_ed448_sign_msg
- wc_ed448ph_sign_hash
- wc_ed448ph_verify_msg

Return:

- 0 メッセージの署名を正常に生成すると返されます。
- BAD_FUNC_ARG 返された入力パラメータは NULL に評価されます。出力バッファが小さすぎて生成された署名を保存するには小さすぎます。
- MEMORY_E 関数の実行中にメモリを割り当てるエラーがある場合に返されます。

B.26.2.6 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
)
```

この関数は、メッセージの ED448 署名を確認して信頼性を確保します。文脈はデータ検証済みの一部として含まれています。答えは RES を介して返され、有効な署名に対応する 1、無効な署名に対応する 0 を返します。

Parameters:

- **検証するシグネチャを含むバッファへの SIG ポインタ。**
- **検証するシグネチャのシグレンの長さ。**
- **メッセージを含むバッファへの MSG ポインタを確認する。**
- **検証するメッセージの MSGlen 長。**
- **署名を検証するためのパブリック ED448 キーへのキーポインタ。**
- **メッセージが署名されたコンテキストを含むバッファへのコンテキストポインタ。** *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
}
```

See:

- wc_ed448ph_verify_hash
- wc_ed448ph_verify_msg
- wc_ed448_sign_msg

Return:

- 0 署名検証と認証を正常に実行したときに返されます。
- BAD_FUNC_ARG いずれかの入力パラメータが NULL に評価された場合、または SIGLEN が署名の実際の長さと一致しない場合に返されます。
- SIG_VERIFY_E 検証が完了した場合は返されますが、生成された署名は提供された署名と一致しません。

B.26.2.7 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
)
```

この関数は、メッセージのダイジェストの ED448 シグネチャを検証して、信頼性を確保します。文脈はデータ検証済みの一部として含まれています。ハッシュは、署名計算前のプリハッシュメッセージです。メッセージダイジェストを作成するために使用されるハッシュアルゴリズムは Shake-256 でなければなりません。答えは RES を介して返され、有効な署名に対応する 1、無効な署名に対応する 0 を返します。

Parameters:

- **検証するシグネチャを含むバッファへの SIG ポインタ。**
- **検証するシグネチャのシグレンの長さ。**
- **検証するメッセージのハッシュを含むバッファへのハッシュポインタ。**
- **検証するハッシュのハッシュレン長。**
- **署名を検証するためのパブリック ED448 キーへのキーポインタ。**
- **メッセージが署名されたコンテキストを含むバッファへのコンテキストポインタ。** *Example*

```
ed448_key key;
int ret, verified = 0;

byte sig[] { initialize with received signature };
byte hash[] = { initialize with SHAKE-256 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
}
```

See:

- wc_ed448_verify_msg
- wc_ed448ph_verify_msg
- wc_ed448ph_sign_hash

Return:

- 0 署名検証と認証を正常に実行したときに返されます。
- BAD_FUNC_ARG いずれかの入力パラメータが NULL に評価された場合、または SIGLEN が署名の実際の長さと一致しない場合に返されます。
- SIG_VERIFY_E 検証が完了した場合は返されますが、生成された署名は提供された署名と一致しません。

B.26.2.8 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
)
```

この関数は、メッセージの ED448 署名を確認して信頼性を確保します。文脈はデータ検証済みの一部として含まれています。検証前にメッセージがプリハッシュされています。答えは RES を介して返され、有効な署名に対応する 1、無効な署名に対応する 0 を返します。

Parameters:

- **検証するシグネチャを含むバッファへの SIG ポインタ。**
- **検証するシグネチャのシグレンの長さ。**
- **メッセージを含むバッファへの MSG ポインタを確認する。**
- **検証するメッセージの MSGlen 長。**
- **署名を検証するためのパブリック ED448 キーへのキーポインタ。**
- **メッセージが署名されたコンテキストを含むバッファへのコンテキストポインタ。** *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
}
```

See:

- wc_ed448_verify_msg
- wc_ed448ph_verify_hash

- wc_ed448ph_sign_msg

Return:

- 0 署名検証と認証を正常に実行したときに返されます。
- BAD_FUNC_ARG いずれかの入力パラメータが NULL に評価された場合、または SIGLEN が署名の実際の長さと一致しない場合に返されます。
- SIG_VERIFY_E 検証が完了した場合は返されますが、生成された署名は提供された署名と一致しません。

B.26.2.9 function wc_ed448_init

```
int wc_ed448_init(
    ed448_key * key
)
```

この関数は、メッセージ検証で将来の使用のために ED448_Key オブジェクトを初期化します。

See:

- wc_ed448_make_key
- wc_ed448_free

Return:

- 0 ED448_Key オブジェクトの初期化に成功したら返されます。
- BAD_FUNC_ARG キーが NULL の場合は返されます。*Example*

```
ed448_key key;
wc_ed448_init(&key);
```

B.26.2.10 function wc_ed448_free

```
void wc_ed448_free(
    ed448_key * key
)
```

この関数は、それが使用された後に ED448 オブジェクトを解放します。*Example*

See: wc_ed448_init

```
ed448_key key;
// initialize key and perform secure exchanges
...
wc_ed448_free(&key);
```

B.26.2.11 function wc_ed448_import_public

```
int wc_ed448_import_public(
    const byte * in,
    word32 inLen,
    ed448_key * key
)
```

この関数は、公開鍵を含むバッファから Public ED448_Key ペアをインポートします。この関数は圧縮キーと非圧縮キーの両方を処理します。

Parameters:

- **公開鍵を含むバッファへのポインタ。**
- **公開鍵を含むバッファのインレル長。** *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
}
```

See:

- wc_ed448_import_private_key
- wc_ed448_export_public

Return:

- 0 ED448_Key のインポートに成功しました。
- BAD_FUNC_ARG IN または KEY が NULL に評価されている場合、または INLEN が ED448 キーのサイズより小さい場合に返されます。

B.26.2.12 function wc_ed448_import_private_only

```
int wc_ed448_import_private_only(
    const byte * priv,
    word32 privSz,
    ed448_key * key
)
```

この関数は、ed448 秘密鍵をバッファからのみインポートします。

Parameters:

- **秘密鍵を含むバッファへの PRIV ポインタ。**
- **秘密鍵の Privsz 長さ。** *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
}
```

See:

- wc_ed448_import_public
- wc_ed448_import_private_key
- wc_ed448_export_private_only

Return:

- 0 ED448 秘密鍵のインポートに成功しました。
- BAD_FUNC_ARG IN または KEY が NULL に評価された場合、または PRIVSZ が ED448_KEY_SIZE よりも小さい場合に返されます。

B.26.2.13 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
)
```

この関数は、一対のバッファからパブリック/プライベート ED448 キーペアをインポートします。この関数は圧縮キーと非圧縮キーの両方を処理します。

Parameters:

- **秘密鍵を含むバッファへの PRIV ポインタ。**
- **秘密鍵の Privsz 長さ。**
- **公開鍵を含むバッファへの Pub ポインタ。**
- **公開鍵の Pubsz の長さ。** *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
}
```

See:

- wc_ed448_import_public
- wc_ed448_import_private_only
- wc_ed448_export_private

Return:

- 0 ED448 キーのインポートに成功しました。
- BAD_FUNC_ARG IN または KEY が NULL に評価された場合、または PROVSZ が ED448_KEY_SIZE または PUBSZ のいずれかが eD448_PUB_KEY_SIZE よりも小さい場合に返されます。

B.26.2.14 function wc_ed448_export_public

```
int wc_ed448_export_public(
    ed448_key * key,
    byte * out,
    word32 * outLen
)
```

この関数は、ED448_Key 構造体から秘密鍵をエクスポートします。公開鍵をバッファアウトに格納し、ounteren でこのバッファに書き込まれたバイトを設定します。

Parameters:

- **公開鍵をエクスポートする ED448_Key 構造体へのキーポインタ。**
- **公開鍵を保存するバッファへのポインタ。** *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
}
```

See:

- wc_ed448_import_public
- wc_ed448_export_private_only

Return:

- 0 公開鍵のエクスポートに成功したら返されます。
- BAD_FUNC_ARG いずれかの入力値が NULL に評価された場合に返されます。
- BUFFER_E 提供されたバッファーが秘密鍵を保存するのに十分な大きさでない場合に返されます。このエラーを返すと、outlen に必要なサイズを設定します。

B.26.2.15 function wc_ed448_export_private_only

```
int wc_ed448_export_private_only(
    ed448_key * key,
    byte * out,
    word32 * outLen
)
```

この関数は、ED448_Key 構造体からの秘密鍵のみをエクスポートします。秘密鍵をバッファアウトに格納し、outlen にこのバッファに書き込まれたバイトを設定します。

Parameters:

- **秘密鍵をエクスポートする ED448_Key 構造体へのキーポインタ。**
- **秘密鍵を保存するバッファへのポインタ。** *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
}
```

See:

- wc_ed448_export_public
- wc_ed448_import_private_key

Return:

- 0 秘密鍵のエクスポートに成功したら返されます。
- ECC_BAD_ARG_E いずれかの入力値が NULL に評価された場合に返されます。

- BUFFER_E 提供されたバッファーが秘密鍵を保存するのに十分な大きさでない場合に返されます。

B.26.2.16 function wc_ed448_export_private

```
int wc_ed448_export_private(
    ed448_key * key,
    byte * out,
    word32 * outLen
)
```

この関数は、ED448_Key 構造体からキーペアをエクスポートします。キーペアをバッファ OUT に格納し、ounteren でこのバッファに書き込まれたバイトを設定します。

Parameters:

- **キーペアをエクスポートするための ED448_Key 構造体へのキーポインタ。**
- **キーペアを保存するバッファへのポインタ。** *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
}
```

See:

- wc_ed448_import_private
- wc_ed448_export_private_only

Return:

- 0 キーペアのエクスポートに成功したら返されます。
- ECC_BAD_ARG_E いずれかの入力値が NULL に評価された場合に返されます。
- BUFFER_E 提供されているバッファーがキーペアを保存するのに十分な大きさでない場合に返されます。

B.26.2.17 function wc_ed448_export_key

```
int wc_ed448_export_key(
    ed448_key * key,
    byte * priv,
    word32 * privSz,
    byte * pub,
    word32 * pubSz
)
```

この関数は、ED448_Key 構造体とは別にプライベートキーと公開鍵をエクスポートします。秘密鍵をバッファー Priv に格納し、PRIVSZ でこのバッファに書き込まれたバイトを設定します。公開鍵をバッファ PUB に格納し、Pubsz でこのバッファに書き込まれたバイトを設定します。

Parameters:

- **キーペアをエクスポートするための ED448_Key 構造体へのキーポインタ。**
- **秘密鍵を保存するバッファへの PRIV ポインタ。**
- **PRIVSZ** PIVINSZ ポインタサイズが表示されているサイズを持つ Word32 オブジェクトへのポインタ。秘密鍵のエクスポート後に書き込まれたバイト数を設定します。
- **パブリックキーを保存するバッファへの Pub。** *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
}
```

See:

- wc_ed448_export_private
- wc_ed448_export_public

Return:

- 0 キーペアのエクスポートに成功したら返されます。
- ECC_BAD_ARG_E いずれかの入力値が NULL に評価された場合に返されます。
- BUFFER_E 提供されているバッファーがキーペアを保存するのに十分な大きさでない場合に返されます。

B.26.2.18 function wc_ed448_check_key

```
int wc_ed448_check_key(
    ed448_key * key
)
```

この関数は、ed448_key 構造体の公開鍵をチェックします。

See: wc_ed448_import_private_key

Return:

- 0 プライベートキーと公開鍵が一致した場合に返されます。
- BAD_FUNC_ARGS 与えられたキーが NULL の場合に返されます。*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(priv, sizeof(priv), pub, sizeof(pub), &key);
ret = wc_ed448_check_key(&key);
if (ret != 0) {
    // error checking key
}
```

B.26.2.19 function wc_ed448_size

```
int wc_ed448_size(
    ed448_key * key
)
```

この関数は、ED448 秘密鍵のサイズ - 57 バイトを返します。

See: wc_ed448_make_key

Return:

- ED448_KEY_SIZE 有効な秘密鍵のサイズ（57 バイト）。
- BAD_FUNC_ARGS 与えられたキーが NULL の場合に返されます。*Example*

```
int keySz;
ed448_key key;
// initialize key, make key
keySz = wc_ed448_size(&key);
if (keySz == 0) {
    // error determining key size
}
```

B.26.2.20 function wc_ed448_priv_size

```
int wc_ed448_priv_size(
    ed448_key * key
)
```

この関数は、秘密鍵サイズ（secret + public）をバイト単位で返します。

See: wc_ed448_pub_size

Return:

- ED448_PRV_KEY_SIZE 秘密鍵のサイズ（114 バイト）。
- BAD_FUNC_ARG key 引数が null の場合は返します。*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);
```

B.26.2.21 function wc_ed448_pub_size

```
int wc_ed448_pub_size(
    ed448_key * key
)
```

この関数は圧縮鍵サイズをバイト単位で返します（公開鍵）。

See: wc_ed448_priv_size

Return:

- ED448_PUB_KEY_SIZE 圧縮公開鍵のサイズ（57 バイト）。
- BAD_FUNC_ARG key 引数が null の場合は返します。*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);
```

B.26.2.22 function wc_ed448_sig_size

```
int wc_ed448_sig_size(
    ed448_key * key
)
```

この関数は、ED448 シグネチャのサイズ（バイト数 114）を返します。

See: wc_ed448_sign_msg

Return:

- ED448_SIG_SIZE ED448 シグネチャ（114 バイト）のサイズ。
- BAD_FUNC_ARG key 引数が null の場合は返します。*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.27.1 Functions

	Name
int	**wolfSSL_CTX_psa_enable**(WOLFSSL_CTX * ctx) この関数は、与えられたコンテキストでの PSA サポートを可能にします。
int	**wolfSSL_set_psa_ctx**(WOLFSSL * ssl, struct psa_ssl_ctx * ctx) 与えられた SSL セッションの PSA コンテキストを設定する機能
void	**wolfSSL_free_psa_ctx**(struct psa_ssl_ctx * ctx) この関数は PSA コンテキストによって使用されるリソースを解放します
int	**wolfSSL_psa_set_private_key_id**(struct psa_ssl_ctx * ctx, psa_key_id_t id) この関数は、SSL セッションによって使用される秘密鍵を設定します

B.27.2 Functions Documentation

B.27.2.1 function wolfSSL_CTX_psa_enable

```
int wolfSSL_CTX_psa_enable(
    WOLFSSL_CTX * ctx
)
```

この関数は、与えられたコンテキストでの PSA サポートを可能にします。

Parameters:

- **ctx** PSA サポートを有効にする必要がある WOLFSSL_CTX オブジェクトへのポインタ

See: wolfSSL_set_psa_ctx

Return: WOLFSSL_SUCCESS 成功した *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");
```

B.27.2.2 function wolfSSL_set_psa_ctx

```
int wolfSSL_set_psa_ctx(
    WOLFSSL * ssl,
    struct psa_ssl_ctx * ctx
)
```

与えられた SSL セッションの PSA コンテキストを設定する機能

Parameters:

- **ssl** CTX が有効になる WolfSSL へのポインタ
- **ctx** Struct PSA_SSL_CTX へのポインタ（SSL セッションに固有である必要があります）

See:

- wolfSSL_psa_set_private_key_id
- wolfSSL_psa_free_psa_ctx

Return: WOLFSSL_SUCCESS 成功した *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);
```

B.27.2.3 function wolfSSL_free_psa_ctx

```
void wolfSSL_free_psa_ctx(
    struct psa_ssl_ctx * ctx
)
```

この関数は PSA コンテキストによって使用されるリソースを解放します

See: wolfSSL_set_psa_ctx

B.27.2.4 function wolfSSL_psa_set_private_key_id

```
int wolfSSL_psa_set_private_key_id(
    struct psa_ssl_ctx * ctx,
    psa_key_id_t id
)
```

この関数は、SSL セッションによって使用される秘密鍵を設定します

Parameters:

- **ctx** 構造体 PSA_SSL_CTX へのポインタ *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);
```

See: wolfSSL_set_psa_ctx

B.28 Algorithm - SipHash

B.28.1 Functions

	Name
int	**wc_InitSipHash**(SipHash * siphash, const unsigned char * key, unsigned char outSz) この関数は、Mac サイズのキーで Siphash を初期化します。
int	**wc_SipHashUpdate**(SipHash * siphash, const unsigned char * in, word32 inSz) 長さ LEN の提供されたバイト配列を絶えずハッシュするように呼び出すことができます。
int	**wc_SipHashFinal**(SipHash * siphash, unsigned char * out, unsigned char outSz) データの Macing を確定します。結果が出入りする。
int	**wc_SipHash**(const unsigned char * key, const unsigned char * in, word32 inSz, unsigned char * out, unsigned char outSz) この機能は Siphash を使用してデータを 1 ショットして、キーに基づいて MAC を計算します。

B.28.2 Functions Documentation

B.28.2.1 function wc_InitSipHash

```
int wc_InitSipHash(
    SipHash * siphash,
    const unsigned char * key,
    unsigned char outSz
)
```

この関数は、Mac サイズのキーで Siphash を初期化します。

Parameters:

- **siphash** Macing に使用するサイプハッシュ構造へのポインタ
- **key** 16 バイト配列へのポインタ *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");
}
```

See:

- wc_SipHash
- wc_SipHashUpdate
- wc_SipHashFinal

Return:

- 0 初期化に成功したときに返されます
- BAD_FUNC_ARG Siphash またはキーが NULL のときに返されます
- BAD_FUNC_ARG OUTSZ が 8 でも 16 でもない場合に返されます

B.28.2.2 function wc_SipHashUpdate

```
int wc_SipHashUpdate(
    SipHash * siphash,
    const unsigned char * in,
    word32 inSz
)
```

長さ LEN の提供されたバイト配列を絶えずハッシュするように呼び出すことができます。

Parameters:

- **siphash** Macing に使用するサイプハッシュ構造へのポインタ
- **in** マイートするデータ *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");
}
```

See:

- wc_SipHash
- wc_InitSipHash
- wc_SipHashFinal

Return:

- 0 Mac にデータを追加したら、返されます
- BAD_FUNC_ARG Siphash が null のとき返されました
- BAD_FUNC_ARG inne が null のとき返され、Insz はゼロではありません

B.28.2.3 function wc_SipHashFinal

```
int wc_SipHashFinal(
    SipHash * siphash,
    unsigned char * out,
    unsigned char outSz
)
```

データの Macing を確定します。結果が出入りする。

Parameters:

- **siphash** Macing に使用するサイプハッシュ構造へのポインタ
- **out** MAC 値を保持するためのバイト配列 *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");
}
```

See:

- wc_SipHash
- wc_InitSipHash
- wc_SipHashUpdate

Return:

- 0 ファイナライズに成功したときに返されます。
- BAD_FUNC_ARG Siphash の OUT が NULL のときに返されます

- BAD_FUNC_ARG OUTSZ が初期化された値と同じではない場合に返されます

B.28.2.4 function wc_SipHash

```
int wc_SipHash(
    const unsigned char * key,
    const unsigned char * in,
    word32 inSz,
    unsigned char * out,
    unsigned char outSz
)
```

この機能は Siphash を使用してデータを 1 ショットして、キーに基づいて MAC を計算します。

Parameters:

- **key** 16 バイト配列へのポインタ
- **in** マイートするデータ
- **inSz** マイクされるデータのサイズ
- **out** MAC 値を保持するためのバイト配列 *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");
}
```

See:

- wc_InitSipHash
- wc_SipHashUpdate
- wc_SipHashFinal

Return:

- 0 Macing に成功したときに返されました
- BAD_FUNC_ARG キーまたは OUT が NULL のときに返されます
- BAD_FUNC_ARG inne が null のとき返され、Insz はゼロではありません
- BAD_FUNC_ARG OUTSZ が 8 でも 16 でもない場合に返されます

C API ヘッダーファイル

C.1 dox_comments/header_files-ja/aes.h

C.1.1 Functions

	Name
int	**wc_AesSetKey**(Aes * aes, const byte * key, word32 len, const byte * iv, int dir) この関数は、鍵を設定して初期化ベクトルを設定することで Aes 構造体を初期化します。
int	**wc_AesSetIV**(Aes * aes, const byte * iv) この関数は、指定された Aes 構造体の初期化ベクトルを設定します。Aes 構造体は、この関数を呼び出す前に初期化されていることが必要です。
int	**wc_AesCbcEncrypt**(Aes * aes, byte * out, const byte * in, word32 sz) 入力バッファの平文メッセージを暗号化し、AES で Cipher Block Chaining を使用して出力バッファに出力します。この関数呼び出しには、メッセージの暗号化前に wc_AesSetKey を呼び出して AES オブジェクトが初期化されている必要があります。この関数は、入力メッセージが AES ブロック長であると仮定し、入力された長さがブロック長の倍数になることを想定しているため、ビルド構成で WOLFSSL_AES_CBC_LENGTH_CHECKS が定義されている場合は任意選択でチェックおよび適用されます。ブロック多入力を保証するために、PKCS#7 スタイルのパディングを事前に追加する必要があります。これは自動的にパディングを追加する OpenSSL AES-CBC メソッドとは異なります。WOLFSSL と対応する OpenSSL 関数を相互運用するには、OpenSSL コマンドライン関数で-nopad オプションを指定して、wolfSSL_AesCbcEncrypt メソッドのように動作し、暗号化中に追加のパディングを追加しません。

	Name
int	**wc_AesCbcDecrypt**(Aes * aes, byte * out, const byte * in, word32 sz) 入力バッファからの暗号メッセージを復号し、AES で Cipher Block Chaining を使用して出力バッファに出力します。この関数呼び出しには、メッセージの暗号化前に wc_AesSetKey を呼び出して AES オブジェクトが初期化されている必要があります。この関数は、元のメッセージが AES ブロック長で整列していたと仮定し、入力された長さがブロック長の倍数になると予想しています。これは OpenSSL AES-CBC メソッドとは異なります。これは、PKCS#7 パディングを自動的に追加するため、ブロックマルチ入力を必要としません。wolfSSL 機能と同等の OpenSSL 関数を相互運用するには、OpenSSL コマンドライン関数で-nopad オプションを指定し、wolfSSL_ AesCbcEncrypt メソッドのように動作し、復号中にエラーを発生させません。
int	**wc_AesCtrEncrypt**(Aes * aes, byte * out, const byte * in, word32 sz) 入力バッファからメッセージを暗号化/復号し、AES CTR モードを使用して出力バッファーに出力します。この関数は、wolfSSL_Aes_Counter がコンパイル時に有効になっている場合にのみ有効になります。この機能を呼び出す前に、Aes 構造体を wc_AesSetKey で初期化する必要があります。この関数は復号と暗号化の両方に使用されます。_ 注: 暗号化と復号のための同じ API を使用することについて。ユーザーは暗号化/復号のための Aes 構造体を区別する必要があります。
int	**wc_AesEncryptDirect**(Aes * aes, byte * out, const byte * in) この関数は、入力ブロック in で与えられた単一の平文データブロックを暗号化して単一の出力ブロック out に出力します。その際に、Aes 構造体で提供されたの鍵を使用します。鍵はこの機能を呼び出す前に wc_AesSetKey で初期化されている必要があります。wc_AesSetKey への入力 iv には NULL を指定して呼び出してください。これは、Configure Option WolfSSL_AES_DIRECT が有効になっている場合にのみ有効になります。__ warning：ほぼすべてのユースケースで ECB モードは安全性が低いと考えられています。可能な限り ECB API を直接使用しないでください。

	Name
int	**wc_AesDecryptDirect**(Aes * aes, byte * out, const byte * in) この関数は、入力ブロック in で与えられた単一の暗号データブロックを復号して単一の出力ブロック out に出力します。提供された Aes 構造体の鍵を使用します。Aes 構造体は、この機能を呼び出す前に wc_AesSetKey で初期化される必要があります。wc_AesSetKey は、iv が NULL で呼び出される必要があります。これは、Configure Option WOLFSSL_AES_DIRECT が有効になっている場合にのみ有効になります。__ warning：ほぼすべてのユースケースで ECB モードは安全性が低いと考えられています。可能な限り ECB API を直接使用しないでください。
int	**wc_AesSetKeyDirect**(Aes * aes, const byte * key, word32 len, const byte * iv, int dir) この関数は、CTR モードの AES 鍵を AES で設定するために使用されます。指定された鍵、iv（初期化ベクトル）、および暗号化 dir（方向）で AES オブジェクトを初期化します。構成オプション WOLFSSL_AES_DIRECT が有効になっている場合にのみ有効になります。wc_AesEncryptDirect と wc_AesDecryptDirect を呼び出す際の Aes 構造体の初期化にはこの関数を使う必要があります。現在 wc_AesSetKeyDirect は内部的に wc_AesSetKey を使用します。__ warning：ほぼすべてのユースケースで ECB モードは安全性が低いと考えられています。可能な限り ECB API を直接使用しないでください
int	**wc_AesGcmSetKey**(Aes * aes, const byte * key, word32 len) この機能は、AES GCM（Galois/Counter Mode）の鍵を設定するために使用されます。与えられた key で Aes 構造体を初期化します。コンパイル時に Configure オプション HAVE_AESGCM が有効になっている場合にのみ有効になります。
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) この関数は、バッファ in に格納されている平文メッセージを暗号化し結果を出力バッファ out に出力します。暗号化する呼び出しごとに新しい iv(初期化ベクトル) が必要です。また、入力認証ベクトル、authIn、authTag への入力認証ベクトルをエンコードします。

	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) この関数は、バッファ in で与えられた入力暗号テキストを復号し、結果を出力バッファ out に格納します。また、指定された認証タグ、authTag に対して、入力認証ベクトル、authIn をチェックします。
int	**wc_GmacSetKey**(Gmac * gmac, const byte * key, word32 len) この関数は、GAROIS メッセージ認証に使用される Gmac 構造体の鍵を初期化して設定します。
int	**wc_GmacUpdate**(Gmac * gmac, const byte * iv, word32 ivSz, const byte * authIn, word32 authInSz, byte * authTag, word32 authTagSz) この関数は authIn Input の GMAC ハッシュを生成し、結果を authTag バッファに格納します。wc_GmacUpdate を実行した後、生成された authTag を既知の認証タグに比較してメッセージの信頼性を検証する必要があります。
int	**wc_AesCcmSetKey**(Aes * aes, const byte * key, word32 keySz) この関数は、CCM を使用して AES オブジェクトの鍵を設定します（CBC-MAC のカウンタ）。Aes 構造体へのポインタを取り、引数で与えられた key で初期化します。
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) この関数は、CCM を使用して、入力メッセージ、IN、OUT、OUT、OUT を CCM（CBC-MAC のカウンタ）を暗号化します。その後、Authin Input から認証タグ、AuthtAg を計算して格納します。
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) この関数は、CCM を使用して、入力暗号テキストを、CCM（CBC-MAC のカウンタ）を使用して出力バッファーに復号します。その後、authIn 入力から authTag を計算します。認証タグが無効な場合は、出力バッファをゼロに設定し、AES_CCM_AUTH_E を返します。
int	**wc_AesXtsSetKey**(XtsAes * aes, const byte * key, word32 len, int dir, void * heap, int devId) この関数は、AES XTS モードを使用する暗号化または復号で使用する鍵の設定に使用します。完了したら、AES キーで wc_AesXtsFree を呼び出すことがユーザーになりました。

	Name
int	**wc_AesXtsEncryptSector**(XtsAes * aes, byte * out, const byte * in, word32 sz, word64 sector)wc_AesXtsEncrypt と同じ処理を行いますが、バイト配列の代わりに Tweak 値として word64 型を使用します。本関数で word64 をバイト配列に変換し、wc_AesXtsEncrypt を呼び出します。
int	**wc_AesXtsDecryptSector**(XtsAes * aes, byte * out, const byte * in, word32 sz, word64 sector)wc_AesXtsDecrypt と同じ処理を行いますが、バイト配列の代わりに Tweak 値として word64 タイプを使用します。本関数で word64 をバイト配列に変換するだけです。
int	**wc_AesXtsEncrypt**(XtsAes * aes, byte * out, const byte * in, word32 sz, const byte * i, word32 iSz)AES XTS モードで暗号化します。（XTS）XEX 暗号化と平文がブロック長の倍数でない場合の処理 (Ciphertext Stealing) を行います。
int	**wc_AesXtsDecrypt**(XtsAes * aes, byte * out, const byte * in, word32 sz, const byte * i, word32 iSz) 暗号化と同じプロセスですが、XtsAes 構造体は AES_Decryption タイプです。
int	**wc_AesXtsFree**(XtsAes * aes) この関数は XtsAes 構造体で使用されるすべてのリソースを解放します。
int	**wc_AesInit**(Aes * aes, void * heap, int devId)Aes 構造体を初期化します。ヒープヒントを設定し、ASYNC ハードウェアを使用する場合の ID も設定します。Aes 構造体の使用が終了した際に wc_AesFree を呼び出すのはユーザーに任されています。
int	**wc_AesFree**(Aes * aes)Aes 構造体に関連つけられたリソースを可能なら解放します。内部的にはノーオペレーションとなることもありますが、ベストプラクティスとしてどのケースでもこの関数を呼び出すことを推奨します。
int	**wc_AesCfbEncrypt**(Aes * aes, byte * out, const byte * in, word32 sz)AES CFB モードで暗号化を行います。
int	**wc_AesCfbDecrypt**(Aes * aes, byte * out, const byte * in, word32 sz)AES CFB モードで復号を行います。
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) この関数は、RFC 5297 に記載されているように SIV（合成初期化ベクトル）暗号化を実行します。

	Name
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) この機能は、RFC 5297 に記載されているように SIV（合成初期化ベクトル）復号を実行します

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
)
```

この関数は、鍵を設定して初期化ベクトルを設定することで Aes 構造体を初期化します。

Parameters:

- **aes** 変更する Aes 構造体へのポインタ
- **key** 暗号化と復号のための 16,24、または 32 バイトの秘密鍵
- **len** 渡された鍵の長さ
- **iv** 鍵を初期化するために使用される初期化ベクトルへのポインタ *Example*

```
Aes enc;
int ret = 0;
byte key[] = { some 16, 24 or 32 byte key };
byte iv[]  = { some 16 byte iv };
if (ret = wc_AesSetKey(&enc, key, AES_BLOCK_SIZE, iv,
AES_ENCRYPTION) != 0) {
// failed to set aes key
}
```

See:

- wc_AesSetKeyDirect
- wc_AesSetIV

Return:

- 0 鍵と初期化ベクトルを正常に設定しました
- BAD_FUNC_ARG 鍵の長さが無効な場合に返されます。

C.1.2.2 function wc_AesSetIV

```
int wc_AesSetIV(
    Aes * aes,
    const byte * iv
)
```

この関数は、指定された Aes 構造体の初期化ベクトルを設定します。Aes 構造体は、この関数を呼び出す前に初期化されていることが必要です。

Parameters:

- **aes** 初期化ベクトルを設定する Aes 構造体へのポインタ *Example*

```
Aes enc;
// set enc key
byte iv[]  = { some 16 byte iv };
if (ret = wc_AesSetIV(&enc, iv) != 0) {
// failed to set aes iv
}
```

See:

- wc_AesSetKeyDirect
- wc_AesSetKey

Return:

- 0 初期化ベクトルを正常に設定します。
- BAD_FUNC_ARG Aes 構造体へのポインタが NULL の場合に返されます。

C.1.2.3 function wc_AesCbcEncrypt

```
int wc_AesCbcEncrypt(
    Aes * aes,
    byte * out,
    const byte * in,
    word32 sz
)
```

入力バッファの平文メッセージを暗号化し、AES で Cipher Block Chaining を使用して出力バッファに出力します。この関数呼び出しには、メッセージの暗号化前に wc_AesSetKey を呼び出して AES オブジェクトが初期化されている必要があります。この関数は、入力メッセージが AES ブロック長であると仮定し、入力された長さがブロック長の倍数になることを想定しているため、ビルド構成で WOLFSSL_AES_CBC_LENGTH_CHECKS が定義されている場合は任意選択でチェックおよび適用されます。ブロック多入力を保証するために、PKCS#7 スタイルのパディングを事前に追加する必要があります。これは自動的にパディングを追加する OpenSSL AES-CBC メソッドとは異なります。WOLFSSL と対応する OpenSSL 関数を相互運用するには、OpenSSL コマンドライン関数で-nopad オプションを指定して、wolfSSL_AesCbcEncrypt メソッドのように動作し、暗号化中に追加のパディングを追加しません。

Parameters:

- **aes** データの暗号化に使用される AES オブジェクトへのポインタ
- **out** 暗号化されたメッセージの暗号文を格納する出力バッファへのポインタ
- **in** 暗号化されるメッセージを含む入力バッファへのポインタ *Example*

```
Aes enc;
int ret = 0;
// initialize enc with 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
}
```

See:

- wc_AesSetKey
- wc_AesSetIV
- wc_AesCbcDecrypt

Return:

- 0 メッセージの暗号化に成功しました。
- BAD_ALIGN_E: ブロックアライメントエラー検出時に返される可能性があります
- BAD_LENGTH_E ライブラリーが WOLFSSL_AES_CBC_LENGTH_CHECKS で構築されている場合で、入力長が AES ブロック長の倍数でない場合に返されます。

C.1.2.4 function wc_AesCbcDecrypt

```
int wc_AesCbcDecrypt(
    Aes * aes,
    byte * out,
    const byte * in,
    word32 sz
)
```

入力バッファからの暗号メッセージを復号し、AES で Cipher Block Chaining を使用して出力バッファに出力します。この関数呼び出しには、メッセージの暗号化前に wc_AesSetKey を呼び出して AES オブジェクトが初期化されている必要があります。この関数は、元のメッセージが AES ブロック長で整列していたと仮定し、入力された長さがブロック長の倍数になると予想しています。これは OpenSSL AES-CBC メソッドとは異なります。これは、PKCS#7 パディングを自動的に追加するため、ブロックマルチ入力を必要としません。wolfSSL 機能と同等の OpenSSL 関数を相互運用するには、OpenSSL コマンドライン関数で-nopad オプションを指定し、wolfSSL_ AesCbcEncrypt メソッドのように動作し、復号中にエラーを発生させません。

Parameters:

- **aes** データを復号するために使用される AES オブジェクトへのポインタ。
- **out** 復号されたメッセージのプレーンテキストを保存する出力バッファへのポインタ。サイズは AES_BLOCK_SIZE の倍数でなければなりません。必要な場合はパディングは追加されます。
- **in** 復号する暗号テキストを含む入力バッファへのポインタ。サイズは AES_BLOCK_SIZE の倍数でなければなりません。パディングされている必要があります。
- **sz** 入力バッファのサイズ *Example*

```
Aes dec;
int ret = 0;
// initialize dec with 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
}
```

See:

- wc_AesSetKey
- wc_AesCbcEncrypt

Return:

- 0 メッセージを正常に復号しました
- BAD_ALIGN_E ブロックアライメントエラー検出時に返される可能性があります
- BAD_LENGTH_E ライブラリーが WOLFSSL_AES_CBC_LENGTH_CHECKS で構築されている場合で、入力長が AES ブロック長の倍数でない場合に返されます。

C.1.2.5 function wc_AesCtrEncrypt

```
int wc_AesCtrEncrypt(
    Aes * aes,
```

```
    byte * out,
    const byte * in,
    word32 sz
)
```

入力バッファからメッセージを暗号化/復号し、AES CTR モードを使用して出力バッファーに出力します。この関数は、wolfSSL_Aes_Counter がコンパイル時に有効になっている場合にのみ有効になります。この機能を呼び出す前に、Aes 構造体を wc_AesSetKey で初期化する必要があります。この関数は復号と暗号化の両方に使用されます。_ 注: 暗号化と復号のための同じ API を使用することについて。ユーザーは暗号化/復号のための Aes 構造体を区別する必要があります。

Parameters:

- **aes** データを復号するために使用される Aes 構造体へのポインタ
- **out** 暗号化されたメッセージの暗号化テキストを保存する出力バッファへのポインタ サイズは AES_BLOCK_SIZE の倍数でなければなりません。必要な場合はパディングは追加されます。
- **in** 暗号化されるプレーンテキストを含む入力バッファへのポインタ。サイズは AES_BLOCK_SIZE の倍数でなければなりません。パディングされている必要があります。
- **sz** 入力バッファのサイズ *Example*

```
Aes enc;
Aes dec;
// initialize enc and dec with 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
```

See: wc_AesSetKey

Return: int WolfSSL エラーまたは成功状況に対応する整数値

C.1.2.6 function wc_AesEncryptDirect

```
int wc_AesEncryptDirect(
    Aes * aes,
    byte * out,
    const byte * in
)
```

この関数は、入力ブロック in で与えられた単一の平文データブロックを暗号化して単一の出力ブロック out に出力します。その際に、Aes 構造体で提供されたの鍵を使用します。鍵はこの機能を呼び出す前に wc_AesSetKey で初期化されている必要があります。wc_AesSetKey への入力 iv には NULL を指定して呼び出してください。これは、Configure Option WolfSSL_AES_DIRECT が有効になっている場合にのみ有効になります。__ warning：ほぼすべてのユースケースで ECB モードは安全性が低いと考えられています。可能な限り ECB API を直接使用しないでください。

Parameters:

- **aes** データの暗号化に使用される Aes 構造体へのポインタ

- **out** 暗号化されたメッセージの暗号化テキストを保存する出力バッファへのポインタ *Example*

```
Aes enc;
// initialize enc with 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);
```

See:

- wc_AesDecryptDirect
- wc_AesSetKeyDirect

Return: int WolfSSL エラーまたは成功状況に対応する整数値

C.1.2.7 function wc_AesDecryptDirect

```
int wc_AesDecryptDirect(
    Aes * aes,
    byte * out,
    const byte * in
)
```

この関数は、入力ブロック in で与えられた単一の暗号データブロックを復号して単一の出力ブロック out に出力します。提供された Aes 構造体の鍵を使用します。Aes 構造体は、この機能を呼び出す前に wc_AesSetKey で初期化される必要があります。wc_AesSetKey は、iv が NULL で呼び出される必要があります。これは、Configure Option WOLFSSL_AES_DIRECT が有効になっている場合にのみ有効になります。__ warning：ほぼすべてのユースケースで ECB モードは安全性が低いと考えられています。可能な限り ECB API を直接使用しないでください。

Parameters:

- **aes** データの復号に使用される AES オブジェクトへのポインタ
- **out** 復号された平文テキストを格納する出力バッファへのポインタ *Example*

```
Aes dec;
// initialize enc with 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);
```

See:

- wc_AesEncryptDirect
- wc_AesSetKeyDirect

Return: int WolfSSL エラーまたは成功状況に対応する整数値

C.1.2.8 function wc_AesSetKeyDirect

```
int wc_AesSetKeyDirect(
    Aes * aes,
    const byte * key,
    word32 len,
    const byte * iv,
    int dir
)
```

この関数は、CTR モードの AES 鍵を AES で設定するために使用されます。指定された鍵、iv（初期化ベクトル）、および暗号化 dir（方向）で AES オブジェクトを初期化します。構成オプション WOLFSSL_AES_DIRECT が有効になっている場合にのみ有効になります。wc_AesEncryptDirect と wc_AesDecryptDirect を呼び出す際の Aes 構造体の初期化にはこの関数を使う必要があります。現在 wc_AesSetKeyDirect は内部的に wc_AesSetKey を使用します。__ warning：ほぼすべてのユースケースで ECB モードは安全性が低いと考えられています。可能な限り ECB API を直接使用しないでください

Parameters:

- **aes** データの暗号化に使用される AES オブジェクトへのポインタ
- **key** 暗号化と復号のための 16,24、または 32 バイトの秘密鍵
- **len** 渡された鍵の長さ
- **iv** 鍵を初期化するために使用される初期化ベクトル
- **dir** 暗号化の方向を指定します。wc_AesEncryptDirect に使用する際には AES_ENCRYPTION、wc_AesDecryptDirect には AES_DECRYPTION を指定します。(注意: wc_AesSetKeyDirect を Aes カウンターモードに使用する際には暗号化/復号によらず、AES_ENCRYPTION を指定してください。)

See:

- wc_AesEncryptDirect
- wc_AesDecryptDirect
- wc_AesSetKey

Return:

- 0 鍵の設定に成功しました。
- BAD_FUNC_ARG 与えられたキーが無効な長さの場合に返されます。

Example

```
Aes enc;
int ret = 0;
byte key[] = { some 16, 24, or 32 byte key };
byte iv[]  = { some 16 byte iv };
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
)
```

この機能は、AES GCM（Galois/Counter Mode）の鍵を設定するために使用されます。与えられた key で Aes 構造体を初期化します。コンパイル時に Configure オプション HAVE_AESGCM が有効になっている場合にのみ有効になります。

Parameters:

- **aes** データの暗号化に使用される Aes 構造体へのポインタ
- **key** 暗号化と復号のための 16,24、または 32 バイトの秘密鍵 *Example*

```
Aes enc;
int ret = 0;
byte key[] = { some 16, 24,32 byte key };
if (ret = wc_AesGcmSetKey(&enc, key, sizeof(key)) != 0) {
```

```
// failed to set aes key
}
```

See:

- wc_AesGcmEncrypt
- wc_AesGcmDecrypt

Return:

- 0 鍵の設定に成功しました。
- BAD_FUNC_ARG 与えられた key が無効な長さの場合に返されます。

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
)
```

この関数は、バッファ in に格納されている平文メッセージを暗号化し結果を出力バッファ out に出力します。暗号化する呼び出しごとに新しい iv(初期化ベクトル) が必要です。また、入力認証ベクトル、authIn、authTag への入力認証ベクトルをエンコードします。

Parameters:

- **aes** データの暗号化に使用される AES オブジェクトへのポインタ
- **out** 暗号テキストを出力する先のバッファへのポインタ。バッファサイズは入力バッファ in のサイズ (sz) と同じでなければなりません。
- **in** 暗号化する平文メッセージを保持している入力バッファへのポインタ。サイズは AES_BLOCK_SIZE の倍数でなければなりません。パディングされている必要があります。
- **sz** 暗号化する入力メッセージの長さ
- **iv** 初期化ベクトルを含むバッファへのポインタ
- **ivSz** 初期化ベクトルの長さ
- **authTag** 認証タグを保存するバッファへのポインタ
- **authTagSz** 希望の認証タグの長さ
- **authIn** 入力認証ベクトルを含むバッファへのポインタ *Example*

```
Aes enc;
// initialize aes structure by calling 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));
```

See:

- wc_AesGcmSetKey
- wc_AesGcmDecrypt

Return: 0 入力メッセージの暗号化に成功しました

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
)
```

この関数は、バッファ in で与えられた入力暗号テキストを復号し、結果を出力バッファ out に格納します。また、指定された認証タグ、authTag に対して、入力認証ベクトル、authIn をチェックします。

Parameters:

- **aes** データの復号に使用される Aes 構造体へのポインタ
- **out** メッセージテキストを保存する出力バッファへのポインタ。サイズは入力バッファ in のサイズ (sz) と同じでなければならない。
- **in** 暗号テキストを保持する入力バッファへのポインタ。サイズは AES_BLOCK_SIZE の倍数でなければならない。
- **sz** 復号する暗号テキストの長さ
- **iv** 初期化ベクトルを含むバッファへのポインタ
- **ivSz** 初期化ベクトルの長さ
- **authTag** 認証タグを含むバッファへのポインタ
- **authTagSz** 希望の認証タグの長さ
- **authIn** 入力認証ベクトルを含むバッファへのポインタ *Example*

```
Aes enc; //can use the same struct as was passed to wc_AesGcmEncrypt
// initialize aes structure by calling 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));
```

See:

- wc_AesGcmSetKey

- wc_AesGcmEncrypt

Return:

- 0 入力メッセージの復号に成功しました
- AES_GCM_AUTH_E 認証タグが提供された認証コードベクトルと一致しない場合、authtag。

C.1.2.12 function wc_GmacSetKey

```
int wc_GmacSetKey(
    Gmac * gmac,
    const byte * key,
    word32 len
)
```

この関数は、GAROIS メッセージ認証に使用される Gmac 構造体の鍵を初期化して設定します。

Parameters:

- **gmac** 認証に使用される Gmac 構造体へのポインタ
- **key** 認証のための 16,24、または 32 バイトの秘密鍵 *Example*

```
Gmac gmac;
key[] = { some 16, 24, or 32 byte length key };
wc_GmacSetKey(&gmac, key, sizeof(key));
```

See: wc_GmacUpdate

Return:

- 0 鍵の設定に成功しました
- BAD_FUNC_ARG 引数 key の長さが無効な場合は返されます。

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
)
```

この関数は authIn Input の GMAC ハッシュを生成し、結果を authTag バッファに格納します。wc_GmacUpdate を実行した後、生成された authTag を既知の認証タグに比較してメッセージの信頼性を検証する必要があります。

Parameters:

- **gmac** 認証に使用される Gmac 構造体へのポインタ
- **iv** ハッシュに使用される初期化ベクトル
- **ivSz** 使用される初期化ベクトルのサイズ
- **authIn** 確認する認証ベクトルを含むバッファへのポインタ
- **authInSz** 認証ベクトルのサイズ
- **authTag** GMAC ハッシュを保存する出力バッファへのポインタ *Example*

```
Gmac gmac;
key[] = { some 16, 24, or 32 byte length key };
iv[] = { some 16 byte length iv };
```

```
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));
```

See: wc_GmacSetKey

Return: 0 GMAC ハッシュの計算に成功しました。

C.1.2.14 function wc_AesCcmSetKey

```
int wc_AesCcmSetKey(
    Aes * aes,
    const byte * key,
    word32 keySz
)
```

この関数は、CCM を使用して AES オブジェクトの鍵を設定します（CBC-MAC のカウンタ）。Aes 構造体へのポインタを取り、引数で与えられた key で初期化します。

Parameters:

- **aes** 引数 key を保管するための Aes 構造体
- **key** 暗号化と復号のための 16,24、または 32 バイトの秘密鍵 *Example*

```
Aes enc;
key[] = { some 16, 24, or 32 byte length key };

wc_AesCcmSetKey(&aes, key, sizeof(key));
```

See:

- wc_AesCcmEncrypt
- wc_AesCcmDecrypt

Return: none

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
)
```

この関数は、CCM を使用して、入力メッセージ、IN、OUT、OUT、OUT を CCM（CBC-MAC のカウンタ）を暗号化します。その後、Authin Input から認証タグ、AuthtAg を計算して格納します。

Parameters:

- **aes** データの暗号化に使用される Aes 構造体へのポインタ
- **out** 暗号テキストを保存する出力バッファへのポインタ
- **in** 暗号化するメッセージを保持している入力バッファへのポインタ
- **sz** 暗号化する入力メッセージの長さ
- **nonce** nonce を含むバッファへのポインタ（1 回だけ使用されている数）
- **nonceSz** ノンスの長さ
- **authTag** 認証タグを保存するバッファへのポインタ
- **authTagSz** 希望の認証タグの長さ
- **authIn** 入力認証ベクトルを含むバッファへのポインタ *Example*

```
Aes enc;
// initialize enc with 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));
```

See:

- wc_AesCcmSetKey
- wc_AesCcmDecrypt

Return: none

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
)
```

この関数は、CCM を使用して、入力暗号テキストを、CCM（CBC-MAC のカウンタ）を使用して出力バッファーに復号します。その後、authIn 入力から authTag を計算します。認証タグが無効な場合は、出力バッファをゼロに設定し、AES_CCM_AUTH_E を返します。

Parameters:

- **aes** データの復号に使用される Aes 構造体へのポインタ
- **out** 復号したテキストを出力する出力バッファへのポインタ
- **in** 復号するメッセージを保持している入力バッファへのポインタ
- **sz** 入力暗号テキストのサイズ
- **nonce** nonce を含むバッファへのポインタ（1 回だけ使用されている数）
- **nonceSz** ノンスの長さ
- **authTag** 認証タグを保存するバッファへのポインタ

- **authTagSz** 希望の認証タグの長さ
- **authIn** 入力認証ベクトルを含むバッファへのポインタ *Example*

```
Aes dec;
// initialize dec with 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
}
```

See:

- wc_AesCcmSetKey
- wc_AesCcmEncrypt

Return:

- 0 入力メッセージの復号に成功しました
- AES_CCM_AUTH_E 認証タグが提供された認証コードベクトルと一致しない場合

C.1.2.17 function wc_AesXtsSetKey

```
int wc_AesXtsSetKey(
    XtsAes * aes,
    const byte * key,
    word32 len,
    int dir,
    void * heap,
    int devId
)
```

この関数は、AES XTS モードを使用する暗号化または復号で使用する鍵の設定に使用します。完了したら、AES キーで wc_AesXtsFree を呼び出すことがユーザーになりました。

Parameters:

- **aes** 暗号化または復号処理に使用する XtsAes 構造体
- **key** 補正値（Tewak）を加味した AES 鍵を保持しているバッファ
- **len** 鍵バッファのサイズ。鍵サイズの 2 倍にする必要があります。（すなわち、16 バイトの鍵の場合は 32）
- **dir** 処理方向、AES_Encryption または AES_Decryption のいずれかを指定します。
- **heap** メモリに使用するヒープヒント。NULL を設定することもできます。*Example*

```
XtsAes aes;

if(wc_AesXtsSetKey(&aes, key, sizeof(key), AES_ENCRYPTION, NULL, 0) != 0)
{
    // Handle error
}
wc_AesXtsFree(&aes);
```

See:

- wc_AesXtsEncrypt
- wc_AesXtsDecrypt
- wc_AesXtsFree

Return: 0 成功

C.1.2.18 function wc_AesXtsEncryptSector

```
int wc_AesXtsEncryptSector(
    XtsAes * aes,
    byte * out,
    const byte * in,
    word32 sz,
    word64 sector
)
```

wc_AesXtsEncrypt と同じ処理を行いますが、バイト配列の代わりに Tweak 値として word64 型を使用します。本関数で word64 をバイト配列に変換し、wc_AesXtsEncrypt を呼び出します。

Parameters:

- **aes** ブロック暗号化/復号に使用する XtsAes 構造体
- **out** 暗号テキストを保持するための出力バッファ
- **in** 暗号化する入力プレーンテキストバッファ
- **sz** バッファ（in, out 両方）のサイズ *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);
```

See:

- wc_AesXtsEncrypt
- wc_AesXtsDecrypt
- wc_AesXtsSetKey
- wc_AesXtsFree

Return: 0 成功

C.1.2.19 function wc_AesXtsDecryptSector

```
int wc_AesXtsDecryptSector(
    XtsAes * aes,
    byte * out,
    const byte * in,
    word32 sz,
    word64 sector
)
```

wc_AesXtsDecrypt と同じ処理を行いますが、バイト配列の代わりに Tweak 値として word64 タイプを使用します。本関数で word64 をバイト配列に変換するだけです。

Parameters:

- **aes** ブロック暗号化/復号に使用する XtsAes 構造体
- **out** プレーンテキストを保持するための出力バッファ
- **in** 復号する暗号テキストバッファ
- **sz** バッファ（in, out 両方）のサイズ *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);
```

See:

- wc_AesXtsEncrypt
- wc_AesXtsDecrypt
- wc_AesXtsSetKey
- wc_AesXtsFree

Return: 0 成功

C.1.2.20 function wc_AesXtsEncrypt

```
int wc_AesXtsEncrypt(
    XtsAes * aes,
    byte * out,
    const byte * in,
    word32 sz,
    const byte * i,
    word32 iSz
)
```

AES XTS モードで暗号化します。(XTS) XEX 暗号化と平文がブロック長の倍数でない場合の処理 (Ciphertext Stealing) を行います。

Parameters:

- **aes** ブロック暗号化/復号に使用する XtsAes 構造体
- **out** 暗号テキストを保持するための出力バッファ
- **in** 暗号化する入力プレーンテキストを含むバッファ
- **sz** バッファ（in, out 両方）のサイズ
- **i** Tweak に使用する値 *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);
```

See:

- wc_AesXtsDecrypt
- wc_AesXtsSetKey
- wc_AesXtsFree

Return: 0 成功

C.1.2.21 function wc_AesXtsDecrypt

```
int wc_AesXtsDecrypt(
    XtsAes * aes,
    byte * out,
    const byte * in,
    word32 sz,
    const byte * i,
    word32 iSz
)
```

暗号化と同じプロセスですが、XtsAes 構造体は AES_Decryption タイプです。

Parameters:

- **aes** ブロック暗号化/復号に使用する XtsAes 構造体
- **out** プレーンテキストを保持するための出力バッファ
- **in** 復号する暗号テキストを含むバッファ
- **sz** バッファ（in, out 両方）のサイズ
- **i** Tweak に使用する値 *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);
```

See:

- wc_AesXtsEncrypt
- wc_AesXtsSetKey
- wc_AesXtsFree

Return: 0 成功

C.1.2.22 function wc_AesXtsFree

```
int wc_AesXtsFree(
    XtsAes * aes
)
```

この関数は XtsAes 構造体で使用されるすべてのリソースを解放します。

See:

- wc_AesXtsEncrypt
- wc_AesXtsDecrypt
- wc_AesXtsSetKey

Return: 0 成功 *Example*

```
XtsAes aes;

if(wc_AesXtsSetKey(&aes, key, sizeof(key), AES_ENCRYPTION, NULL, 0) != 0)
{
    // Handle error
}
wc_AesXtsFree(&aes);
```

C.1.2.23 function wc_AesInit

```
int wc_AesInit(
    Aes * aes,
    void * heap,
    int devId
)
```

Aes 構造体を初期化します。ヒープヒントを設定し、ASYNC ハードウェアを使用する場合の ID も設定します。Aes 構造体の使用が終了した際に wc_AesFree を呼び出すのはユーザーに任されています。

Parameters:

- **aes** 初期化対象の Aes 構造体
- **heap** 必要に応じて malloc / free に使用するヒープヒント *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);
```

See:

- wc_AesSetKey
- wc_AesSetIV

Return: 0 成功

C.1.2.24 function wc_AesFree

```
int wc_AesFree(
    Aes * aes
)
```

Aes 構造体に関連つけられたリソースを可能なら解放します。内部的にはノーオペレーションとなることもありますが、ベストプラクティスとしてどのケースでもこの関数を呼び出すことを推奨します。

Parameters:

- **aes** Free すべき Aes 構造体へのポインタ *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);
```

See: wc_AesInit

Return: 戻り値なし

C.1.2.25 function wc_AesCfbEncrypt

```
int wc_AesCfbEncrypt(
    Aes * aes,
    byte * out,
    const byte * in,
    word32 sz
)
```

AES CFB モードで暗号化を行います。

Parameters:

- **aes** ブロック暗号化/復号に使用する Aes 構造体
- **out** 暗号テキストを保持するための出力バッファは、少なくとも入力プレーンテキストバッファと同じサイズが必要です。
- **in** 暗号化する入力プレーンテキストバッファ *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
}
```

See:

- wc_AesCfbDecrypt
- wc_AesSetKey

Return: 0 成功時に返ります。失敗時には負値が返されます。

C.1.2.26 function wc_AesCfbDecrypt

```
int wc_AesCfbDecrypt(
    Aes * aes,
    byte * out,
```

```
    const byte * in,
    word32 sz
)
```

AES CFB モードで復号を行います。

Parameters:

- **aes** ブロック暗号化/復号に使用する Aes 構造体
- **out** 復号されたテキストを保持するための出力バッファは、少なくとも入力バッファと同じサイズが必要です。
- **in** 復号する暗号データを保持した入力バッファ *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
}
```

See:

- wc_AesCfbEncrypt
- wc_AesSetKey

Return: 0 成功時に返ります。失敗時には負値が返されます。

C.1.2.27 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
)
```

この関数は、RFC 5297 に記載されているように SIV（合成初期化ベクトル）暗号化を実行します。

Parameters:

- **key** 使用する鍵を含むバイトバッファ。
- **keySz** 鍵バッファの長さ（バイト単位）。
- **assoc** 追加の認証された関連データ（AD）。
- **assocSz** AD バッファのバイト数
- **nonce** ナンス（一度だけ使用される値）。AD と同じ方法でアルゴリズムによって使用されます。
- **nonceSz** バイト単位のナンスバッファの長さ。
- **in** 暗号化する平文のバッファ。
- **inSz** 平文バッファの長さ
- **siv** S2V による SIV 出力（RFC 5297 2.4 参照）。*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
}
```

See: wc_AesSivDecrypt

Return:

- 0 暗号化に成功した場合
- BAD_FUNC_ARG 鍵、SIV、または出力バッファが NULL の場合。鍵サイズが 32,48、または 64 バイトの場合にも返されます。
- Other その他の負のエラー値。AES または CMAC 操作が失敗した場合に返されます。

C.1.2.28 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
)
```

この機能は、RFC 5297 に記載されているように SIV（合成初期化ベクトル）復号を実行します

Parameters:

- **key** 使用する鍵を含むバイトバッファ。
- **keySz** 鍵バッファの長さ（バイト単位）。
- **assoc** 追加の認証された関連データ（AD）。
- **assocSz** AD バッファのバイト数
- **nonce** ナンス（一度だけ使用される値）。AD と同じ方法で、基礎となるアルゴリズムによって使用されます。
- **nonceSz** バイト単位のナンスバッファの長さ。
- **in** 復号する暗号文バッファ。
- **inSz** 暗号文バッファの長さ
- **siv** 暗号文に付随する SIV（RFC 5297 2.4 を参照）。*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
}
```

See: wc_AesSivEncrypt

Return:

- 0 復号に成功した場合
- BAD_FUNC_ARG 鍵、SIV、または出力バッファが NULL の場合。キーサイズが 32,48、または 64 バイトの場合にも返されます。
- AES_SIV_AUTH_E S2V によって派生した SIV が入力 SIV と一致しない場合（RFC 5297 2.7 を参照）。
- Other その他の負のエラー値。AES または CMAC 操作が失敗した場合に返されます。

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_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);
```

C.2 dox_comments/header_files-ja/arc4.h

C.2.1 Functions

	Name
int	**wc_Arc4Process**(Arc4 * arc4, byte * out, const byte * in, word32 length) この関数は、バッファ内の入力メッセージを暗号化し、出力バッファーに暗号文を配置するか、またはバッファーから暗号文を復号化したり、ARC4 暗号化を使用して、出力バッファー OUT を出力したりします。この関数は暗号化と復号化の両方に使用されます。この方法が呼び出される可能性がある場合は、まず WC_ARC4SETKEY を使用して ARC4 構造を初期化する必要があります。
int	**wc_Arc4SetKey**(Arc4 * arc4, const byte * key, word32 length) この関数は ARC4 オブジェクトのキーを設定し、それを暗号として使用するために初期化します。WC_ARC4PROCESS を使用した暗号化に使用する前に呼び出される必要があります。

C.2.2 Functions Documentation

C.2.2.1 function wc_Arc4Process

```
int wc_Arc4Process(
    Arc4 * arc4,
    byte * out,
    const byte * in,
    word32 length
)
```

この関数は、バッファ内の入力メッセージを暗号化し、出力バッファーに暗号文を配置するか、またはバッファーから暗号文を復号化したり、ARC4 暗号化を使用して、出力バッファー OUT を出力したりします。この関数は暗号化と復号化の両方に使用されます。この方法が呼び出される可能性がある場合は、まず WC_ARC4SETKEY を使用して ARC4 構造を初期化する必要があります。

Parameters:

- **arc4** メッセージの処理に使用される ARC4 構造へのポインタ
- **out** 処理されたメッセージを保存する出力バッファへのポインタ
- **in** プロセスするメッセージを含む入力バッファへのポインタ *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));
```

See: wc_Arc4SetKey

Return: none

C.2.2.2 function wc_Arc4SetKey

```
int wc_Arc4SetKey(
    Arc4 * arc4,
    const byte * key,
    word32 length
)
```

この関数は ARC4 オブジェクトのキーを設定し、それを暗号として使用するために初期化します。WC_ARC4PROCESS を使用した暗号化に使用する前に呼び出される必要があります。

Parameters:

- **arc4** 暗号化に使用される ARC4 構造へのポインタ
- **key** ARC4 構造を初期化するためのキー *Example*

```
Arc4 enc;
byte key[] = { initialize with key to use for encryption };
wc_Arc4SetKey(&enc, key, sizeof(key));
```

See: wc_Arc4Process

Return: none

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-ja/asn.h

C.4 dox_comments/header_files-ja/asn_public.h

C.4.1 Functions

	Name
int	**wc_InitCert**(Cert *) この関数は Cert 構造体をデフォルトの値で初期化します。デフォルトのオプション：version = 3（0x2）、sigtype = sha_with_rsa、issuer = 空白、dayValid = 500、selfsigned = 1（true）発行者としての件名 = 空白
Cert *	**wc_CertNew**(void * heap) この関数は証明書操作の為に新たな Cert 構造体を割り当てます。割り当てた Cert 構造体はこの関数内で初期化されるので、wc_InitCert() を呼び出す必要はありません。アプリケーションがこの Cert 構造体の使用を終了する際には wc_CertFree() を呼び出す必要があります。
void	**wc_CertFree**(Cert * cert) この関数は wc_CertNew() で確保された Cert 構造体を解放します。

	Name
int	**wc_MakeCert**(Cert * cert, byte * derBuffer, word32 derSz, RsaKey * rsaKey, ecc_key * eccKey, WC_RNG * rng)CA 署名付き証明書を作成するために使用されます。サブジェクト情報を入力した後に呼び出す必要があります。この関数は、証明書入力から X.509v3 RSA または ECC 証明書を作成し derBuffer に書き込みます。証明書を生成するための RsaKey または EccKey のいずれかを引数として取ります。この関数が呼び出される前に、証明書を wc_InitCert で初期化する必要があります。
int	**wc_MakeCertReq**(Cert * cert, byte * derBuffer, word32 derSz, RsaKey * rsaKey, ecc_key * eccKey) この関数は、入力された Cert 構造体を使用して証明書署名要求を作成し derBuffer に書き込みます。証明書要求の生成には RsaKey または EccKey のいずれかの鍵を受け取り使用します。この関数の後に、署名するために wc_SignCert() を呼び出す必要があります。この関数の使用例については、wolfCrypt テストアプリケーション (./wolfcrypt/test/test.c) を参照してください。
int	**wc_SignCert**(int requestSz, int sigType, byte * derBuffer, word32 derSz, RsaKey * rsaKey, ecc_key * eccKey, WC_RNG * rng) この関数はバッファーの内容に署名し、署名をバッファの最後に追加します。署名の種類を取ります。CA 署名付き証明書を作成する場合は、wc_MakeCert() または wc_MakeCertReq() の後に呼び出す必要があります。
int	**wc_MakeSelfCert**(Cert * cert, byte * derBuffer, word32 derSz, RsaKey * key, WC_RNG * rng) この関数は、以前の 2 つの関数、wc_MakeCert、および自己署名のための wc_SignCert の組み合わせです（前の関数は CA 要求に使用される場合があります）。証明書を作成してから、それに署名し、自己署名証明書を生成します。
int	**wc_SetIssuer**(Cert * cert, const char * issuerFile) この関数は PEM 形式の issureFile で与えられた発行者を証明書の発行者として設定します。また、その際に、証明書の自己署名プロパティを false に変更します。発行者は証明書の発行者として設定される前に検証されます。この関数は証明書への署名に先立ち呼び出される必要があります。
int	**wc_SetSubject**(Cert * cert, const char * subjectFile) この関数は PEM 形式の subjectFile で与えられた主体者を証明書の主体者として設定します。この関数は証明書への署名に先立ち呼び出される必要があります。

	Name
int	**wc_SetSubjectRaw**(Cert * cert, const byte * der, int derSz) この関数は DER 形式でバッファに格納されている Raw-Subject 情報を証明書の Raw-Subject 情報として設定します。この関数は証明書への署名に先立ち呼び出される必要があります。
int	**wc_GetSubjectRaw**(byte ** subjectRaw, Cert * cert) この関数は Cert 構造体から Raw-Subject 情報を取り出します。
int	**wc_SetAltNames**(Cert * cert, const char * file) この関数は引数で与えられた PEM 形式の証明書の主体者の別名を Cert 構造体に設定します。複数のドメインで同一の証明書を使用する際には主体者の別名を付与する機能は有用です。この関数は証明書への署名に先立ち呼び出される必要があります。
int	**wc_SetIssuerBuffer**(Cert * cert, const byte * der, int derSz) この関数は DER 形式でバッファに格納されている発行者を証明書の発行者として設定します。加えて、証明書の事故署名プロパティを false に設定します。この関数は証明書への署名に先立ち呼び出される必要があります。
int	**wc_SetIssuerRaw**(Cert * cert, const byte * der, int derSz) この関数は DER 形式でバッファに格納されている Raw-Issuer 情報を証明書の Raw-Issuer 情報として設定します。この関数は証明書への署名に先立ち呼び出される必要があります。
int	**wc_SetSubjectBuffer**(Cert * cert, const byte * der, int derSz) この関数は DER 形式でバッファに格納されている主体者を証明書の主体者として設定します。この関数は証明書への署名に先立ち呼び出される必要があります。
int	**wc_SetAltNamesBuffer**(Cert * cert, const byte * der, int derSz) この関数は DER 形式でバッファに格納されている「別名情報」を証明書の「別名情報」として設定します。この機能は複数ドメインを一つの証明書を使ってセキュアにする際に有用です。この関数は証明書への署名に先立ち呼び出される必要があります。
int	**wc_SetDatesBuffer**(Cert * cert, const byte * der, int derSz) この関数は DER 形式でバッファに格納されている「有効期間」情報を証明書の「有効期間」情報として設定します。この関数は証明書への署名に先立ち呼び出される必要があります。
int	**wc_SetAuthKeyIdFromPublicKey**(Cert * cert, RsaKey * rsakey, ecc_key * eckey) この関数は指定された RSA あるいは ECC 公開鍵の一方から得た AKID（認証者鍵 ID）を証明書の AKID として設定します。

	Name
int	**wc_SetAuthKeyIdFromCert**(Cert * cert, const byte * der, int derSz) この関数は DER 形式でバッファに格納された証明書から得た AKID(認証者鍵 ID) を証明書の AKID として設定します。
int	**wc_SetAuthKeyId**(Cert * cert, const char * file) この関数は PEM 形式の証明書から得た AKID(認証者鍵 ID) を証明書の AKID として設定します。
int	**wc_SetSubjectKeyIdFromPublicKey**(Cert * cert, RsaKey * rsakey, ecc_key * eckey) この関数は指定された RSA あるいは ECC 公開鍵の一方から得た SKID（主体者鍵 ID）を証明書の SKID として設定します。
int	**wc_SetSubjectKeyId**(Cert * cert, const char * file) この関数は PEM 形式の証明書から得た SKID(主体者鍵 ID) を証明書の SKID として設定します。引数は両方が与えられることが必要です。
int	**wc_SetKeyUsage**(Cert * cert, const char * value) この関数は鍵の用途を設定します。設定値の指定はコンマ区切りトークンを使用できます。受け付けられるトークンは：digitalSignature, nonRepudiation, contentCommitment, keyCertSign, cRLSign, dataEncipherment, keyAgreement, keyEncipherment, encipherOnly, decipherOnly です。指定例：“digitalSignature,nonRepudiation”。nonRepudiation と contentCommitment は同じ用途を意味します。
int	**wc_PemPubKeyToDer**(const char * fileName, unsigned char * derBuf, int derSz)PEM 形式の鍵ファイルをロードし DER 形式に変換してバッファに出力します。
int	**wc_PubKeyPemToDer**(const unsigned char * pem, int pemSz, unsigned char * buff, int buffSz)PEM 形式の鍵データを DER 形式に変換してバッファに出力し、出力バイト数あるいは負のエラー値を返します。
int	**wc_PemCertToDer**(const char * fileName, unsigned char * derBuf, int derSz) この関数は PEM 形式の証明書を DER 形式に変換し、与えられたバッファに出力します。
int	**wc_DerToPem**(const byte * der, word32 derSz, byte * output, word32 outputSz, int type) この関数はバッファで与えられた DER 形式の証明書を PEM 形式に変換し、与えられた出力用バッファに出力します。この関数は入力バッファと出力バッファを共用することはできません。両バッファは必ず別のものを用意してください。

	Name
int	**wc_DerToPemEx**(const byte * der, word32 derSz, byte * output, word32 outputSz, byte * cipherIno, int type) この関数は DER 形式証明書を入力バッファから読み出し、PEM 形式に変換して出力バッファに出力します。この関数は入力バッファと出力バッファを共用することはできません。両バッファは必ず別のものを用意してください。追加の暗号情報を指定することができます。
int	**wc_KeyPemToDer**(const unsigned char * pem, int pemSz, unsigned char * buff, int buffSz, const char * pass)PEM 形式の鍵を DER 形式に変換します。
int	**wc_CertPemToDer**(const unsigned char * pem, int pemSz, unsigned char * buff, int buffSz, int type) この関数は PEM 形式の証明書を DER 形式に変換します。内部では OpenSSL 互換 API の PemToDer を呼び出します。
int	**wc_GetPubKeyDerFromCertは DER/ASN.1 エンコードされた証明書を受け付けます。PEM 形式の鍵を DER 形式で取得する場合には、wc_InitDecodedCert() より先に wc_CertPemToDer() を呼び出してください。
int	**wc_EccPrivateKeyDecode**(const byte * input, word32 * inOutIdx, ecc_key * key, word32 inSz) この関数は ECC 秘密鍵を入力バッファから読み込み、解析の後 ecc_key 構造体を作成してそこに鍵を格納します。
int	**wc_EccKeyToDer**(ecc_key * key, byte * output, word32 inLen) この関数は ECC 秘密鍵を DER 形式でバッファに出力します。
int	**wc_EccPublicKeyDecode**(const byte * input, word32 * inOutIdx, ecc_key * key, word32 inSz) この関数は入力バッファの ECC 公開鍵を ASN シーケンスをデコードして取り出します。
int	**wc_EccPublicKeyToDer**(ecc_key * key, byte * output, word32 inLen, int with_AlgCurve) この関数は ECC 公開鍵を DER 形式に変換します。処理したバッファのサイズを返します。変換して得られる DER 形式の ECC 公開鍵は出力バッファに格納されます。AlgCurve フラグの指定により、アルゴリズムと曲線情報をヘッダーに含めることができます。
int	**wc_EccPublicKeyToDer_ex**(ecc_key * key, byte * output, word32 inLen, int with_AlgCurve, int comp) この関数は ECC 公開鍵を DER 形式に変換します。処理したバッファサイズを返します。変換された DER 形式の ECC 公開鍵は出力バッファに格納されます。AlgCurve フラグの指定により、アルゴリズムと曲線情報をヘッダーに含めることができます。comp パラメータは公開鍵を圧縮して出力するか否かを指定します。

	Name
word32	**wc_EncodeSignature**(byte * out, const byte * digest, word32 digSz, int hashOID) この関数はデジタル署名をエンコードして出力バッファに出力し、生成された署名のサイズを返します。
int	**wc_GetCTC_HashOID**(int type) この関数はハッシュタイプに対応したハッシュ OID を返します。例えば、ハッシュタイプが “WC_SHA512” の場合、この関数は “SHA512h” を対応するハッシュ OID として返します。
void	**wc_SetCert_Free**(Cert * cert) この関数はキャッシュされていた Cert 構造体で使用されたメモリとリソースをクリーンアップします。WOLFSSL_CERT_GEN_CACHE が定義されている場合には DecodedCert 構造体が Cert 構造体内部にキャッシュされ、後続する set 系関数の呼び出しの都度 DecodedCert 構造体がパースされることを防ぎます。
int	**wc_GetPkcs8TraditionalOffset**(byte * input, word32 * inOutIdx, word32 sz) この関数は PKCS#8 の暗号化されていないバッファ内部の従来の秘密鍵の開始位置を検出して返します。
int	**wc_CreatePKCS8Key**(byte * out, word32 * outSz, byte * key, word32 keySz, int algoID, const byte * curveOID, word32 oidSz) この関数は DER 形式の秘密鍵を入力とし、RKCS#8 形式に変換します。また、PKCS#12 のシュロ — ディットキーバッグの作成にも使用できます。RFC5208 を参照のこと。
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) この関数は暗号化されていない PKCS#8 の DER 形式の鍵 (例えば wc_CreatePKCS8Key で生成された鍵) を受け取り、PKCS#8 暗号化形式に変換します。結果として得られた暗号化鍵は wc_DecryptPKCS8Key を使って復号できます。RFC5208 を参照してください。
int	**wc_DecryptPKCS8Key**(byte * input, word32 sz, const char * password, int passwordSz) この関数は暗号化された PKCS#8 の DER 形式の鍵を受け取り、復号して PKCS#8 DER 形式に変換します。wc_EncryptPKCS8Key によって行われた暗号化を元に戻します。RFC5208 を参照してください。入力データは復号データによって上書きされます。

	Name
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) この関数は従来の DER 形式の鍵を PKCS#8 フォーマットに変換し、暗号化を行います。この処理には wc_CreatePKCS8Key と wc_EncryptPKCS8Key を使用します。
void	**wc_InitDecodedCert**(struct DecodedCert * cert, const byte * source, word32 inSz, void * heap) この関数は cert 引数で与えられた DecodedCert 構造体を初期化します。DER 形式の証明書を含んでいる source 引数の指すポインタから証明書サイズ inSz の長さを内部に保存します。この関数の後に呼び出される wc_ParseCert によって証明書が解析されます。
int	**wc_ParseCert**(DecodedCert * cert, int type, int verify, void * cm) この関数は DecodedCert 構造体に保存されている DER 形式の証明書を解析し、その構造体に各種フィールドを設定します。DecodedCert 構造体は wc_InitDecodedCert を呼び出して初期化しておく必要があります。この関数はオプションで CertificateManager 構造体へのポインタを受け取り、CA が証明書マネジャーで検索できた場合には、その CA に関する情報も DecodedCert 構造体に追加設定します。
void	**wc_FreeDecodedCert**(struct DecodedCert * cert) この関数は wc_InitDecodedCert で初期化済みの DecodedCert 構造体を解放します。
int	**wc_SetTimeCb**(wc_time_cb f) この関数はタイムコールバック関数を登録します。wolfSSL が現在時刻を必要としたタイミングでこのコールバックを呼び出します。このタイムコールバック関数のプロトタイプ（シグネチャ）は C 標準ライブラリの “time” 関数と同一です。
time_t	**wc_Time**(time_t * t) この関数は現在時刻を取得します。デフォルトで XTIME マクロ関数を使います。このマクロ関数はプラットフォーム依存です。ユーザーはこのマクロの代わりに wc_SetTimeCb でタイムコールバック関数を使うように設定することができます
int	**wc_SetCustomExtension**(Cert * cert, int critical, const char * oid, const byte * der, word32 derSz) この関数は X.509 証明書にカスタム拡張を追加します。注: この関数に渡すポインタ引数が保持する内容は証明書が生成されるまで変更されてはいけません。この関数ではポインタが指す先の内容は別のバッファには複製しません。

	Name
int	**wc_SetUnknownExtCallback**(DecodedCert * cert, wc_UnknownExtCallback cb) この関数は wolfSSL が証明書の解析中に未知の X.509 拡張に遭遇した際に呼び出すコールバック関数を登録します。コールバック関数のプロトタイプは使用例を参照してください。
int	**wc_CheckCertSigPubKey**(const byte * cert, word32 certSz, void * heap, const byte * pubKey, word32 pubKeySz, int pubKeyOID) この関数は DER 形式の X.509 証明書の署名を与えられた公開鍵を使って検証します。公開鍵は DER 形式で全公開鍵情報を含んだものが求められます。
int	**wc_Asn1PrintOptions_Init**(Asn1PrintOptions * opts) この関数は Asn1PrintOptions 構造体を初期化します。
int	**wc_Asn1PrintOptions_Set**(Asn1PrintOptions * opts, enum Asn1PrintOpt opt, word32 val) この関数は Asn1PrintOptions 構造体にプリント情報を設定します。
int	**wc_Asn1_Init**(Asn1 * asn1) この関数は Asn1 構造体を初期化します。
int	**wc_Asn1_SetFile**(Asn1 * asn1, XFILE file) この関数は出力先として使用するファイルを Asn1 構造体にセットします。
int	**wc_Asn1_PrintAll**(Asn1 * asn1, Asn1PrintOptions * opts, unsigned char * data, word32 len)ASN.1 アイテムをプリントします。

C.4.2 Functions Documentation

C.4.2.1 function wc_InitCert

```
int wc_InitCert(
    Cert *
)
```

この関数は Cert 構造体をデフォルトの値で初期化します。デフォルトのオプション：version = 3（0x2）、sigtype = sha_with_rsa、issuer = 空白、dayValid = 500、selfsigned = 1（true）発行者としての件名 = 空白

See:

- wc_MakeCert
- wc_MakeCertReq

Return: 成功した場合 0 を返します。

Example

```
Cert myCert;
wc_InitCert(&myCert);
```

C.4.2.2 function wc_CertNew

```
Cert * wc_CertNew(
    void * heap
```

)

この関数は証明書操作の為に新たな Cert 構造体を割り当てます。割り当てた Cert 構造体はこの関数内で初期化されるので、wc_InitCert() を呼び出す必要はありません。アプリケーションがこの Cert 構造体の使用を終了する際には wc_CertFree() を呼び出す必要があります。

Parameters:

- **メモリの動的確保で使用されるヒープへのポインタ。NULL の指定も可。** *Example*

```
Cert*   myCert;

myCert = wc_CertNew(NULL);
if (myCert == NULL) {
    // Cert creation failure
}
```

See:

- wc_InitCert
- wc_MakeCert
- wc_CertFree

Return:

- 処理が成功した際には新に割り当てられた Cert 構造体へのポインタを返します。
- メモリ確保に失敗した場合には NULL を返します。

C.4.2.3 function wc_CertFree

```
void wc_CertFree(
    Cert * cert
)
```

この関数は wc_CertNew() で確保された Cert 構造体を解放します。

Parameters:

- **解放すべき Cert 構造体へのポインタ** *Example*

```
Cert*   myCert;

myCert = wc_CertNew(NULL);

// Perform cert operations.

wc_CertFree(myCert);
```

See:

- wc_InitCert
- wc_MakeCert
- wc_CertNew

Return: 無し

C.4.2.4 function wc_MakeCert

```
int wc_MakeCert(
    Cert * cert,
    byte * derBuffer,
    word32 derSz,
```

```
    RsaKey * rsaKey,
    ecc_key * eccKey,
    WC_RNG * rng
)
```

CA 署名付き証明書を作成するために使用されます。サブジェクト情報を入力した後に呼び出す必要があります。この関数は、証明書入力から X.509v3 RSA または ECC 証明書を作成し derBuffer に書き込みます。証明書を生成するための RsaKey または EccKey のいずれかを引数として取ります。この関数が呼び出される前に、証明書を wc_InitCert で初期化する必要があります。

Parameters:

- **cert** 初期化された Cert 構造体へのポインタ
- **derBuffer** 生成された証明書を保持するバッファへのポインタ
- **derSz** 証明書を保存するバッファのサイズ
- **rsaKey** 証明書の生成に使用される RSA 鍵を含む RsaKey 構造体へのポインタ
- **eccKey** 証明書の生成に使用される ECC 鍵を含む EccKey 構造体へのポインタ

See:

- wc_InitCert
- wc_MakeCertReq

Return:

- 指定された入力証明書から X509 証明書が正常に生成された場合、生成された証明書のサイズを返します。
- MEMORY_E xmalloc でのメモリ割り当てでエラーが発生した場合に返ります。
- BUFFER_E 提供された derBuffer が生成された証明書を保存するには小さすぎる場合に返されます
- Others 証明書の生成が成功しなかった場合、追加のエラーメッセージが返される可能性があります。

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
)
```

この関数は、入力された Cert 構造体を使用して証明書署名要求を作成し derBuffer に書き込みます。証明書要求の生成には RsaKey または EccKey のいずれかの鍵を受け取り使用します。この関数の後に、署名するために wc_SignCert() を呼び出す必要があります。この関数の使用例については、wolfCrypt テストアプリケーション (./wolfcrypt/test/test.c) を参照してください。

Parameters:

- **cert** 初期化された Cert 構造体へのポインタ
- **derBuffer** 生成された証明書署名要求を保持するバッファへのポインタ
- **derSz** 証明書署名要求を保存するバッファのサイズ
- **rsaKey** 証明書署名要求を生成するために使用される RSA 鍵を含む RsaKey 構造体へのポインタ
- **eccKey** 証明書署名要求を生成するために使用される RECC 鍵を含む EccKey 構造体へのポインタ

See:

- wc_InitCert
- wc_MakeCert

Return:

- 証明書署名要求が正常に生成されると、生成された証明書署名要求のサイズを返します。
- MEMORY_E xmalloc でのメモリ割り当てでエラーが発生した場合
- BUFFER_E 提供された derBuffer が生成された証明書を保存するには小さすぎる場合
- Other 証明書署名要求の生成が成功しなかった場合、追加のエラーメッセージが返される可能性があります。

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
)
```

この関数はバッファーの内容に署名し、署名をバッファの最後に追加します。署名の種類を取ります。CA 署名付き証明書を作成する場合は、wc_MakeCert() または wc_MakeCertReq() の後に呼び出す必要があります。

Parameters:

- **requestSz** 署名対象の証明書本文のサイズ
- **sigType** 作成する署名の種類。有効なオプションは次のとおりです:CTC_MD5WRSA、CTC_SHAWRSA、CTC_SHAWECDSA、CTC_SHA256WECDSA、ANDCTC_SHA256WRSA
- **derBuffer** 署名対象の証明書を含むバッファへのポインタ。関数の処理成功時には署名が付加された証明書を保持します。
- **derSz** 新たに署名された証明書を保存するバッファの（合計）サイズ
- **rsaKey** 証明書に署名するために使用される RSA 鍵を含む RsaKey 構造体へのポインタ
- **eccKey** 証明書に署名するために使用される ECC 鍵を含む EccKey 構造体へのポインタ
- **rng** 署名に使用する乱数生成器 (WC_RNG 構造体) へのポインタ

See:

- wc_InitCert
- wc_MakeCert

Return:

- 証明書への署名に成功した場合は、証明書の新しいサイズ (署名を含む) を返します。
- MEMORY_E xmalloc でのメモリを割り当てでエラーがある場合
- BUFFER_E 提供された証明書を保存するには提供されたバッファが小さすぎる場合に返されます。
- Other 証明書の生成が成功しなかった場合、追加のエラーメッセージが返される可能性があります。

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
)
```

この関数は、以前の 2 つの関数、wc_MakeCert、および自己署名のための wc_SignCert の組み合わせです (前の関数は CA 要求に使用される場合があります)。証明書を作成してから、それに署名し、自己署名証明書を生成します。

Parameters:

- **cert** 署名する対象の Cert 構造体へのポインタ
- **derBuffer** 署名付き証明書を保持するためのバッファへのポインタ
- **derSz** 署名付き証明書を保存するバッファのサイズ
- **key** 証明書に署名するために使用される RSA 鍵を含む RsaKey 構造体へのポインタ
- **rng** 署名に使用する乱数生成器 (WC_RNG 構造体) へのポインタ

See:

- wc_InitCert
- wc_MakeCert
- wc_SignCert

Return:

- 証明書への署名が成功した場合は、証明書の新しいサイズを返します。
- MEMORY_E xmalloc でのメモリを割り当てでエラーがある場合
- BUFFER_E 提供された証明書を保存するには提供されたバッファが小さすぎる場合に返されます。
- Other 証明書の生成が成功しなかった場合、追加のエラーメッセージが返される可能性があります。

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
)
```

この関数は PEM 形式の issureFile で与えられた発行者を証明書の発行者として設定します。また、その際に、証明書の自己署名プロパティを false に変更します。発行者は証明書の発行者として設定される前に検証されます。この関数は証明書への署名に先立ち呼び出される必要があります。

Parameters:

- **cert** 発行者を設定する対象の Cert 構造体へのポインタ
- **issuerFile** PEM 形式の証明書ファイルへのファイルパス

See:

- wc_InitCert
- wc_SetSubject
- wc_SetIssuerBuffer

Return:

- 0 証明書の発行者の設定に成功した場合に返されます。
- MEMORY_E XMALLOC でメモリの確保に失敗した際に返されます。
- ASN_PARSE_E 証明書のヘッダーファイルの解析に失敗した際に返されます。
- ASN_OBJECT_ID_E 証明書の暗号タイプの解析でエラーが発生した際に返されます。
- ASN_EXPECT_0_E 証明書の暗号化仕様にフォーマットエラーが検出された際に返されます。
- ASN_BEFORE_DATE_E 証明書の使用開始日より前であった場合に返されます。
- ASN_AFTER_DATE_E 証明書の有効期限日より後であった場合に返されます。
- ASN_BITSTR_E 証明書のビットストリング要素の解析でエラーが発生した際に返されます。
- ECC_CURVE_OID_E 証明書の ECC 鍵の解析でエラーが発生した際に返されます。
- ASN_UNKNOWN_OID_E 証明書が未知のオブジェクト ID を使用していた際に返されます。
- ASN_VERSION_E ALLOW_V1_EXTENSIONS マクロが定義されていないのに証明書が V1 あるいは V2 形式であった場合に返されます。
- BAD_FUNC_ARG 証明書の拡張情報の解析でエラーが発生した際に返されます。
- ASN_CRIT_EXT_E 証明書の解析中に未知のクリティカル拡張に遭遇した際に返されます。
- ASN_SIG_OID_E 署名暗号化タイプが引数で渡された証明書のタイプと異なる場合に返されます。
- ASN_SIG_CONFIRM_E 証明書の署名の検証に失敗した際に返されます。
- ASN_NAME_INVALID_E 証明書の名前が CA の名前に関数制限によって許されていない場合に返されます。
- ASN_NO_SIGNER_E CA 証明書の発行者を検証することができない場合に返されます。

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
)
```

この関数は PEM 形式の subjectFile で与えられた主体者を証明書の主体者として設定します。この関数は証明書への署名に先立ち呼び出される必要があります。

Parameters:

- **主体者を設定する対象の Cert 構造体へのポインタ**
- **subjectFile** PEM 形式の証明書ファイルへのファイルパス

See:

- wc_InitCert
- wc_SetIssuer

Return:

- 0 証明書の主体者の設定に成功した場合に返されます。
- MEMORY_E XMALLOC でメモリの確保に失敗した際に返されます。
- ASN_PARSE_E 証明書のヘッダーファイルの解析に失敗した際に返されます。
- ASN_OBJECT_ID_E 証明書の暗号タイプの解析でエラーが発生した際に返されます。
- ASN_EXPECT_0_E 証明書の暗号化仕様にフォーマットエラーが検出された際に返されます。
- ASN_BEFORE_DATE_E 証明書の使用開始日より前であった場合に返されます。
- ASN_AFTER_DATE_E 証明書の有効期限日より後であった場合に返されます。
- ASN_BITSTR_E 証明書のビットストリング要素の解析でエラーが発生した際に返されます。
- ECC_CURVE_OID_E 証明書の ECC 鍵の解析でエラーが発生した際に返されます。
- ASN_UNKNOWN_OID_E 証明書が未知のオブジェクト ID を使用していた際に返されます。
- ASN_VERSION_E ALLOW_V1_EXTENSIONS マクロが定義されていないのに証明書が V1 あるいは V2 形式であった場合に返されます。
- BAD_FUNC_ARG 証明書の拡張情報の解析でエラーが発生した際に返されます。
- ASN_CRIT_EXT_E 証明書の解析中に未知のクリティカル拡張に遭遇した際に返されます。
- ASN_SIG_OID_E 署名暗号化タイプが引数で渡された証明書のタイプと異なる場合に返されます。
- ASN_SIG_CONFIRM_E 証明書の署名の検証に失敗した際に返されます。
- ASN_NAME_INVALID_E 証明書の名前が CA の名前に関数制限によって許されていない場合に返されます。
- ASN_NO_SIGNER_E CA 証明書の主体者を検証することができない場合に返されます。

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
)
```

この関数は DER 形式でバッファに格納されている Raw-Subject 情報を証明書の Raw-Subject 情報として設定します。この関数は証明書への署名に先立ち呼び出される必要があります。

Parameters:

- **cert** Raw-Subject 情報を設定する対象の Cert 構造体へのポインタ
- **der** DER 形式の証明書を格納しているバッファへのポインタ。この証明書の Raw-Subject 情報が取り出されて cert に設定されます。
- **derSz** DER 形式の証明書を格納しているバッファのサイズ

See:

- wc_InitCert
- wc_SetSubject

Return:

- 0 証明書の Raw-Subject 情報の設定に成功した場合に返されます。
- MEMORY_E XMALLOC でメモリの確保に失敗した際に返されます。
- ASN_PARSE_E 証明書のヘッダーファイルの解析に失敗した際に返されます。
- ASN_OBJECT_ID_E 証明書の暗号タイプの解析でエラーが発生した際に返されます。
- ASN_EXPECT_0_E 証明書の暗号化仕様にフォーマットエラーが検出された際に返されます。
- ASN_BEFORE_DATE_E 証明書の使用開始日より前であった場合に返されます。
- ASN_AFTER_DATE_E 証明書の有効期限日より後であった場合に返されます。
- ASN_BITSTR_E 証明書のビットストリング要素の解析でエラーが発生した際に返されます。
- ECC_CURVE_OID_E 証明書の ECC 鍵の解析でエラーが発生した際に返されます。
- ASN_UNKNOWN_OID_E 証明書が未知のオブジェクト ID を使用していた際に返されます。
- ASN_VERSION_E ALLOW_V1_EXTENSIONS マクロが定義されていないのに証明書が V1 あるいは V2 形式であった場合に返されます。
- BAD_FUNC_ARG 証明書の拡張情報の解析でエラーが発生した際に返されます。
- ASN_CRIT_EXT_E 証明書の解析中に未知のクリティカル拡張に遭遇した際に返されます。
- ASN_SIG_OID_E 署名暗号化タイプが引数で渡された証明書のタイプと異なる場合に返されます。
- ASN_SIG_CONFIRM_E 証明書の署名の検証に失敗した際に返されます。
- ASN_NAME_INVALID_E 証明書の名前が CA の名前に関数制限によって許されていない場合に返されます。
- ASN_NO_SIGNER_E CA 証明書の主体者を検証することができない場合に返されます。
- ASN_NO_SIGNER_E CA 証明書の主体者を検証することができない場合に返されます。

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
)
```

この関数は Cert 構造体から Raw-Subject 情報を取り出します。

Parameters:

- **subjectRaw** 処理が成功した際に返される Raw-Subject 情報を格納するバッファへのポインタのポインタ
- **cert** Raw-Subject 情報を保持する Cert 構造体へのポインタ

See:

- wc_InitCert
- wc_SetSubjectRaw

Return:

- 0 証明書の Raw-Subject 情報の取得に成功した場合に返されます。
- BAD_FUNC_ARG 証明書の拡張情報の解析でエラーが発生した際に返されます。

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
)
```

この関数は引数で与えられた PEM 形式の証明書の主体者の別名を Cert 構造体に設定します。複数のドメインで同一の証明書を使用する際には主体者の別名を付与する機能は有用です。この関数は証明書への署名に先立ち呼び出される必要があります。

Parameters:

- **cert** 主体者の別名を設定する対象の Cert 構造体へのポインタ
- **file** PEM 形式の証明書のファイルパス

See:

- wc_InitCert
- wc_SetIssuer

Return:

- 0 証明書の主体者の設定に成功した場合に返されます。
- MEMORY_E XMALLOC でメモリの確保に失敗した際に返されます。
- ASN_PARSE_E 証明書のヘッダーファイルの解析に失敗した際に返されます。
- ASN_OBJECT_ID_E 証明書の暗号タイプの解析でエラーが発生した際に返されます。
- ASN_EXPECT_0_E 証明書の暗号化仕様にフォーマットエラーが検出された際に返されます。
- ASN_BEFORE_DATE_E 証明書の使用開始日より前であった場合に返されます。
- ASN_AFTER_DATE_E 証明書の有効期限日より後であった場合に返されます。
- ASN_BITSTR_E 証明書のビットストリング要素の解析でエラーが発生した際に返されます。
- ECC_CURVE_OID_E 証明書の ECC 鍵の解析でエラーが発生した際に返されます。
- ASN_UNKNOWN_OID_E 証明書が未知のオブジェクト ID を使用していた際に返されます。

- ASN_VERSION_E ALLOW_V1_EXTENSIONS マクロが定義されていないのに証明書が V1 あるいは V2 形式であった場合に返されます。
- BAD_FUNC_ARG 証明書の拡張情報の解析でエラーが発生した際に返されます。
- ASN_CRIT_EXT_E 証明書の解析中に未知のクリティカル拡張に遭遇した際に返されます。
- ASN_SIG_OID_E 署名暗号化タイプが引数で渡された証明書のタイプと異なる場合に返されます。
- ASN_SIG_CONFIRM_E 証明書の署名の検証に失敗した際に返されます。
- ASN_NAME_INVALID_E 証明書の名前が CA の名前に関数制限によって許されていない場合に返されます。
- ASN_NO_SIGNER_E CA 証明書の主体者を検証することができない場合に返されます。
- ASN_NO_SIGNER_E CA 証明書の主体者を検証することができない場合に返されます。

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
)
```

この関数は DER 形式でバッファに格納されている発行者を証明書の発行者として設定します。加えて、証明書の事故署名プロパティを false に設定します。この関数は証明書への署名に先立ち呼び出される必要があります。

Parameters:

- **cert** 発行者を設定する対象の Cert 構造体へのポインタ
- **der** DER 形式の証明書を格納しているバッファへのポインタ。この証明書の発行者情報が取り出されて cert に設定されます。
- **derSz** DER 形式の証明書を格納しているバッファのサイズ

See:

- wc_InitCert
- wc_SetIssuer

Return:

- 0 証明書の発行者の設定に成功した場合に返されます。
- MEMORY_E XMALLOC でメモリの確保に失敗した際に返されます。
- ASN_PARSE_E 証明書のヘッダーファイルの解析に失敗した際に返されます。
- ASN_OBJECT_ID_E 証明書の暗号タイプの解析でエラーが発生した際に返されます。
- ASN_EXPECT_0_E 証明書の暗号化仕様にフォーマットエラーが検出された際に返されます。
- ASN_BEFORE_DATE_E 証明書の使用開始日より前であった場合に返されます。
- ASN_AFTER_DATE_E 証明書の有効期限日より後であった場合に返されます。
- ASN_BITSTR_E 証明書のビットストリング要素の解析でエラーが発生した際に返されます。
- ECC_CURVE_OID_E 証明書の ECC 鍵の解析でエラーが発生した際に返されます。
- ASN_UNKNOWN_OID_E 証明書が未知のオブジェクト ID を使用していた際に返されます。
- ASN_VERSION_E ALLOW_V1_EXTENSIONS マクロが定義されていないのに証明書が V1 あるいは V2 形式であった場合に返されます。
- BAD_FUNC_ARG 証明書の拡張情報の解析でエラーが発生した際に返されます。
- ASN_CRIT_EXT_E 証明書の解析中に未知のクリティカル拡張に遭遇した際に返されます。

- ASN_SIG_OID_E 署名暗号化タイプが引数で渡された証明書のタイプと異なる場合に返されます。
- ASN_SIG_CONFIRM_E 証明書の署名の検証に失敗した際に返されます。
- ASN_NAME_INVALID_E 証明書の名前が CA の名前に関数制限によって許されていない場合に返されます。
- ASN_NO_SIGNER_E CA 証明書の主体者を検証することができない場合に返されます。
- ASN_NO_SIGNER_E CA 証明書の主体者を検証することができない場合に返されます。

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
)
```

この関数は DER 形式でバッファに格納されている Raw-Issuer 情報を証明書の Raw-Issuer 情報として設定します。この関数は証明書への署名に先立ち呼び出される必要があります。

Parameters:

- **cert** Raw-Issuer 情報を設定する対象の Cert 構造体へのポインタ
- **der** DER 形式の証明書を格納しているバッファへのポインタ。この証明書の Raw-Issuer 情報が取り出されて cert に設定されます。
- **derSz** DER 形式の証明書を格納しているバッファのサイズ

See:

- wc_InitCert
- wc_SetIssuer

Return:

- 0 証明書の Raw-Issuer 情報の設定に成功した場合に返されます。
- MEMORY_E XMALLOC でメモリの確保に失敗した際に返されます。
- ASN_PARSE_E 証明書のヘッダーファイルの解析に失敗した際に返されます。
- ASN_OBJECT_ID_E 証明書の暗号タイプの解析でエラーが発生した際に返されます。
- ASN_EXPECT_0_E 証明書の暗号化仕様にフォーマットエラーが検出された際に返されます。
- ASN_BEFORE_DATE_E 証明書の使用開始日より前であった場合に返されます。
- ASN_AFTER_DATE_E 証明書の有効期限日より後であった場合に返されます。
- ASN_BITSTR_E 証明書のビットストリング要素の解析でエラーが発生した際に返されます。
- ECC_CURVE_OID_E 証明書の ECC 鍵の解析でエラーが発生した際に返されます。
- ASN_UNKNOWN_OID_E 証明書が未知のオブジェクト ID を使用していた際に返されます。
- ASN_VERSION_E ALLOW_V1_EXTENSIONS マクロが定義されていないのに証明書が V1 あるいは V2 形式であった場合に返されます。
- BAD_FUNC_ARG 証明書の拡張情報の解析でエラーが発生した際に返されます。
- ASN_CRIT_EXT_E 証明書の解析中に未知のクリティカル拡張に遭遇した際に返されます。
- ASN_SIG_OID_E 署名暗号化タイプが引数で渡された証明書のタイプと異なる場合に返されます。
- ASN_SIG_CONFIRM_E 証明書の署名の検証に失敗した際に返されます。

- ASN_NAME_INVALID_E 証明書の名前が CA の名前に関数制限によって許されていない場合に返されます。
- ASN_NO_SIGNER_E CA 証明書の主体者を検証することができない場合に返されます。
- ASN_NO_SIGNER_E CA 証明書の主体者を検証することができない場合に返されます。

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
)
```

この関数は DER 形式でバッファに格納されている主体者を証明書の主体者として設定します。この関数は証明書への署名に先立ち呼び出される必要があります。

Parameters:

- **cert** 主体者を設定する対象の Cert 構造体へのポインタ
- **der** DER 形式の証明書を格納しているバッファへのポインタ。この証明書の主体者が取り出されて cert に設定されます。
- **derSz** DER 形式の証明書を格納しているバッファのサイズ

See:

- wc_InitCert
- wc_SetSubject

Return:

- 0 証明書の主体者の設定に成功した場合に返されます。
- MEMORY_E XMALLOC でメモリの確保に失敗した際に返されます。
- ASN_PARSE_E 証明書のヘッダーファイルの解析に失敗した際に返されます。
- ASN_OBJECT_ID_E 証明書の暗号タイプの解析でエラーが発生した際に返されます。
- ASN_EXPECT_0_E 証明書の暗号化仕様にフォーマットエラーが検出された際に返されます。
- ASN_BEFORE_DATE_E 証明書の使用開始日より前であった場合に返されます。
- ASN_AFTER_DATE_E 証明書の有効期限日より後であった場合に返されます。
- ASN_BITSTR_E 証明書のビットストリング要素の解析でエラーが発生した際に返されます。
- ECC_CURVE_OID_E 証明書の ECC 鍵の解析でエラーが発生した際に返されます。
- ASN_UNKNOWN_OID_E 証明書が未知のオブジェクト ID を使用していた際に返されます。
- ASN_VERSION_E ALLOW_V1_EXTENSIONS マクロが定義されていないのに証明書が V1 あるいは V2 形式であった場合に返されます。
- BAD_FUNC_ARG 証明書の拡張情報の解析でエラーが発生した際に返されます。
- ASN_CRIT_EXT_E 証明書の解析中に未知のクリティカル拡張に遭遇した際に返されます。
- ASN_SIG_OID_E 署名暗号化タイプが引数で渡された証明書のタイプと異なる場合に返されます。
- ASN_SIG_CONFIRM_E 証明書の署名の検証に失敗した際に返されます。
- ASN_NAME_INVALID_E 証明書の名前が CA の名前に関数制限によって許されていない場合に返されます。

- ASN_NO_SIGNER_E CA 証明書の主体者を検証することができない場合に返されます。
- ASN_NO_SIGNER_E CA 証明書の主体者を検証することができない場合に返されます。

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
)
```

この関数は DER 形式でバッファに格納されている「別名情報」を証明書の「別名情報」として設定します。この機能は複数ドメインを一つの証明書を使ってセキュアにする際に有用です。この関数は証明書への署名に先立ち呼び出される必要があります。

Parameters:

- **cert** 別名情報を設定する対象の Cert 構造体へのポインタ
- **der** DER 形式の証明書を格納しているバッファへのポインタ。この証明書の別名情報が取り出されて cert に設定されます。
- **derSz** DER 形式の証明書を格納しているバッファのサイズ

See:

- wc_InitCert
- wc_SetAltNames

Return:

- 0 証明書の別名情報の設定に成功した場合に返されます。
- MEMORY_E XMALLOC でメモリの確保に失敗した際に返されます。
- ASN_PARSE_E 証明書のヘッダーファイルの解析に失敗した際に返されます。
- ASN_OBJECT_ID_E 証明書の暗号タイプの解析でエラーが発生した際に返されます。
- ASN_EXPECT_0_E 証明書の暗号化仕様にフォーマットエラーが検出された際に返されます。
- ASN_BEFORE_DATE_E 証明書の使用開始日より前であった場合に返されます。
- ASN_AFTER_DATE_E 証明書の有効期限日より後であった場合に返されます。
- ASN_BITSTR_E 証明書のビットストリング要素の解析でエラーが発生した際に返されます。
- ECC_CURVE_OID_E 証明書の ECC 鍵の解析でエラーが発生した際に返されます。
- ASN_UNKNOWN_OID_E 証明書が未知のオブジェクト ID を使用していた際に返されます。
- ASN_VERSION_E ALLOW_V1_EXTENSIONS マクロが定義されていないのに証明書が V1 あるいは V2 形式であった場合に返されます。
- BAD_FUNC_ARG 証明書の拡張情報の解析でエラーが発生した際に返されます。
- ASN_CRIT_EXT_E 証明書の解析中に未知のクリティカル拡張に遭遇した際に返されます。
- ASN_SIG_OID_E 署名暗号化タイプが引数で渡された証明書のタイプと異なる場合に返されます。
- ASN_SIG_CONFIRM_E 証明書の署名の検証に失敗した際に返されます。
- ASN_NAME_INVALID_E 証明書の名前が CA の名前に関数制限によって許されていない場合に返されます。
- ASN_NO_SIGNER_E CA 証明書の主体者を検証することができない場合に返されます。

- ASN_NO_SIGNER_E CA 証明書の主体者を検証することができない場合に返されます。

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
)
```

この関数は DER 形式でバッファに格納されている「有効期間」情報を証明書の「有効期間」情報として設定します。この関数は証明書への署名に先立ち呼び出される必要があります。

Parameters:

- **cert** 有効期間情報を設定する対象の Cert 構造体へのポインタ
- **der** DER 形式の証明書を格納しているバッファへのポインタ。この証明書の有効期間情報が取り出されて cert に設定されます。
- **derSz** DER 形式の証明書を格納しているバッファのサイズ

See: wc_InitCert

Return:

- 0 証明書の有効期間情報の設定に成功した場合に返されます。
- MEMORY_E XMALLOC でメモリの確保に失敗した際に返されます。
- ASN_PARSE_E 証明書のヘッダーファイルの解析に失敗した際に返されます。
- ASN_OBJECT_ID_E 証明書の暗号タイプの解析でエラーが発生した際に返されます。
- ASN_EXPECT_0_E 証明書の暗号化仕様にフォーマットエラーが検出された際に返されます。
- ASN_BEFORE_DATE_E 証明書の使用開始日より前であった場合に返されます。
- ASN_AFTER_DATE_E 証明書の有効期限日より後であった場合に返されます。
- ASN_BITSTR_E 証明書のビットストリング要素の解析でエラーが発生した際に返されます。
- ECC_CURVE_OID_E 証明書の ECC 鍵の解析でエラーが発生した際に返されます。
- ASN_UNKNOWN_OID_E 証明書が未知のオブジェクト ID を使用していた際に返されます。
- ASN_VERSION_E ALLOW_V1_EXTENSIONS マクロが定義されていないのに証明書が V1 あるいは V2 形式であった場合に返されます。
- BAD_FUNC_ARG 証明書の拡張情報の解析でエラーが発生した際に返されます。
- ASN_CRIT_EXT_E 証明書の解析中に未知のクリティカル拡張に遭遇した際に返されます。
- ASN_SIG_OID_E 署名暗号化タイプが引数で渡された証明書のタイプと異なる場合に返されます。
- ASN_SIG_CONFIRM_E 証明書の署名の検証に失敗した際に返されます。
- ASN_NAME_INVALID_E 証明書の名前が CA の名前に関数制限によって許されていない場合に返されます。
- ASN_NO_SIGNER_E CA 証明書の主体者を検証することができない場合に返されます。
- ASN_NO_SIGNER_E CA 証明書の主体者を検証することができない場合に返されます。

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
)
```

この関数は指定された RSA あるいは ECC 公開鍵の一方から得た AKID（認証者鍵 ID）を証明書の AKID として設定します。

Parameters:

- **cert** AKID を設定する対象の Cert 構造体へのポインタ
- **rsakey** RsaKey 構造体へのポインタ
- **eckey** ecc_key 構造体へのポインタ

See:

- wc_SetSubjectKeyId
- wc_SetAuthKeyId
- wc_SetAuthKeyIdFromCert

Return:

- 0 証明書の AKID の設定に成功した場合に返されます。
- BAD_FUNC_ARG Cert 構造体へのポインタ (cert) が NULL か RsaKey 構造体へのポインタ (rsakey) と ecc_key 構造体へのポインタ (eckey) の両方が NULL である場合に返されます。
- MEMORY_E メモリの確保に失敗した際に返されます。
- PUBLIC_KEY_E 公開鍵の取得に失敗した際に返されます。

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
)
```

この関数は DER 形式でバッファに格納された証明書から得た AKID(認証者鍵 ID) を証明書の AKID として設定します。

Parameters:

- **cert** AKID を設定する対象の Cert 構造体へのポインタ。
- **der** DER 形式の証明書を格納しているバッファへのポインタ。
- **derSz** DER 形式の証明書を格納しているバッファのサイズ。

See:

- wc_SetAuthKeyIdFromPublicKey
- wc_SetAuthKeyId

Return:

- 0 証明書の AKID の設定に成功した場合に返されます。
- BAD_FUNC_ARG 引数のいずれかが NULL, あるいは derSz が 0 より小さい場合に返されます。
- MEMORY_E メモリの確保に失敗した際に返されます。
- ASN_NO_SKID 認証者鍵 ID が見つからない場合に返されます。

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
)
```

この関数は PEM 形式の証明書から得た AKID(認証者鍵 ID) を証明書の AKID として設定します。

Parameters:

- **cert** AKID を設定する対象の Cert 構造体へのポインタ。
- **file** PEM 形式の証明書ファイルへのファイルパス

See:

- wc_SetAuthKeyIdFromPublicKey
- wc_SetAuthKeyIdFromCert

Return:

- 0 証明書の AKID の設定に成功した場合に返されます。
- BAD_FUNC_ARG 引数のいずれかが NULL の場合に返されます。
- MEMORY_E メモリの確保に失敗した際に返されます。

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
)
```

この関数は指定された RSA あるいは ECC 公開鍵の一方から得た SKID（主体者鍵 ID）を証明書の SKID として設定します。

Parameters:

- **cert** SKID を設定する対象の Cert 構造体へのポインタ
- **rsakey** RsaKey 構造体へのポインタ
- **eckey** ecc_key 構造体へのポインタ

See: wc_SetSubjectKeyId

Return:

- 0 証明書の SKID の設定に成功した場合に返されます。
- BAD_FUNC_ARG Cert 構造体へのポインタ (cert) が NULL か RsaKey 構造体へのポインタ (rsakey) と ecc_key 構造体へのポインタ (eckey) の両方が NULL である場合に返されます。
- MEMORY_E メモリの確保に失敗した際に返されます。
- PUBLIC_KEY_E 公開鍵の取得に失敗した際に返されます。

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
)
```

この関数は PEM 形式の証明書から得た SKID(主体者鍵 ID) を証明書の SKID として設定します。引数は両方が与えられることが必要です。

Parameters:

- **cert** SKID を設定する対象の Cert 構造体へのポインタ。
- **file** PEM 形式の証明書ファイルへのファイルパス

See: wc_SetSubjectKeyIdFromPublicKey

Return:

- 0 証明書の SKID の設定に成功した場合に返されます。
- BAD_FUNC_ARG 引数のいずれかが NULL の場合に返されます。
- MEMORY_E メモリの確保に失敗した際に返されます。
- PUBLIC_KEY_E 公開鍵のデコードに失敗した際に返されます。

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
)
```

この関数は鍵の用途を設定します。設定値の指定はコンマ区切りトークンを使用できます。受け付けられるトークンは：digitalSignature, nonRepudiation, contentCommitment, keyCertSign, cRLSign, dataEncipherment, keyAgreement, keyEncipherment, encipherOnly, decipherOnly です。指定例：“digitalSignature,nonRepudiation”。nonRepudiation と contentCommitment は同じ用途を意味します。

Parameters:

- **cert** 鍵の用途を設定する対象の初期化済み Cert 構造体へのポインタ。
- **value** 鍵の用途を意味するコンマ区切りトークン文字列へのポインタ

See:

- wc_InitCert
- wc_MakeRsaKey

Return:

- 0 証明書の用途の設定に成功した場合に返されます。
- BAD_FUNC_ARG 引数のいずれかが NULL の場合に返されます。
- MEMORY_E メモリの確保に失敗した際に返されます。
- KEYUSAGE_E 未知のトークンが検出された際に返されます。

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
)
```

PEM 形式の鍵ファイルをロードし DER 形式に変換してバッファに出力します。

Parameters:

- **fileName** PEM 形式のファイルパス
- **derBuf** DER 形式鍵を出力する先のバッファ
- **derSz** 出力先バッファのサイズ

See: wc_PubKeyPemToDer

Return:

- 0 処理成功時に返されます。
- <0 エラー発生時に返されます。
- SSL_BAD_FILE ファイルのオープンに問題が生じた際に返されます。
- MEMORY_E メモリの確保に失敗した際に返されます。
- BUFFER_E 与えられた出力バッファ derBuf が結果を保持するのに十分な大きさがない場合に返されます。

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
)
```

PEM 形式の鍵データを DER 形式に変換してバッファに出力し、出力バイト数あるいは負のエラー値を返します。

Parameters:

- **pem** PEM 形式の鍵を含んだバッファへのポインタ
- **pemSz** PEM 形式の鍵を含んだバッファのサイズ
- **buff** 出力先バッファへのポインタ
- **buffSz** 出力先バッファのサイズ

See: wc_PemPubKeyToDer

Return:

- 0 処理成功時には出力したバイト数が返されます。
- BAD_FUNC_ARG 引数の pem, buff, あるいは buffSz のいずれかば NULL の場合に返されます。
- <0 エラーが発生した際に返されます。

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
)
```

この関数は PEM 形式の証明書を DER 形式に変換し、与えられたバッファに出力します。

Parameters:

- **fileName** PEM 形式のファイルパス
- **derBuf** DER 形式証明書を出力する先のバッファへのポインタ
- **derSz** DER 形式証明書を出力する先のバッファのサイズ

See: none

Return:

- 処理成功時には出力したバイト数が返されます。
- BUFFER_E 与えられた出力バッファ derBuf が結果を保持するのに十分な大きさがない場合に返されます。
- MEMORY_E メモリの確保に失敗した際に返されます。

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
)
```

この関数はバッファで与えられた DER 形式の証明書を PEM 形式に変換し、与えられた出力用バッファに出力します。この関数は入力バッファと出力バッファを共用することはできません。両バッファは必ず別のものを用意してください。

Parameters:

- **der** DER 形式証明書データを保持するバッファへのポインタ
- **derSz** DER 形式証明書データのサイズ
- **output** PEM 形式証明書データを出力する先のバッファへのポインタ
- **outSz** PEM 形式証明書データを出力する先のバッファのサイズ
- **type** 変換する証明書のタイプ。次のタイプが指定可: CERT_TYPE, PRIVATEKEY_TYPE, ECC_PRIVATEKEY_TYPE, and CERTREQ_TYPE.

See: wc_PemCertToDer

Return:

- 処理成功時には変換後の PEM 形式データのサイズを返します。
- BAD_FUNC_ARG DER 形式証明書データの解析中にエラーが発生した際、あるいは PEM 形式に変換の際にエラーが発生した際に返されます。
- MEMORY_E メモリの確保に失敗した際に返されます。
- ASN_INPUT_E Base64 エンコーディングエラーが検出された際に返されます。
- BUFFER_E 与えられた出力バッファが結果を保持するのに十分な大きさがない場合に返されます。

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
)
```

この関数は DER 形式証明書を入力バッファから読み出し、PEM 形式に変換して出力バッファに出力します。この関数は入力バッファと出力バッファを共用することはできません。両バッファは必ず別のものを用意してください。追加の暗号情報を指定することができます。

Parameters:

- **der** DER 形式証明書データを保持するバッファへのポインタ
- **derSz** DER 形式証明書データのサイズ
- **output** PEM 形式証明書データを出力する先のバッファへのポインタ
- **outSz** PEM 形式証明書データを出力する先のバッファのサイズ
- **cipher_inf** 追加の暗号情報
- **type** 生成する証明書タイプ。指定可能なタイプ: CERT_TYPE, PRIVATEKEY_TYPE, ECC_PRIVATEKEY_TYPE と CERTREQ_TYPE

See: wc_PemCertToDer

Return:

- 処理成功時には変換後の PEM 形式データのサイズを返します。

- BAD_FUNC_ARG Returned DER 形式証明書データの解析中にエラーが発生した際、あるいは PEM 形式に変換の際にエラーが発生した際に返されます。
- MEMORY_E メモリの確保に失敗した際に返されます。
- ASN_INPUT_E Base64 エンコーディングエラーが検出された際に返されます。
- BUFFER_E 与えられた出力バッファが結果を保持するのに十分な大きさがない場合に返されます。

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
)
```

PEM 形式の鍵を DER 形式に変換します。

Parameters:

- **pem** PEM 形式の証明書データへのポインタ
- **pemSz** PEM 形式の証明書データのサイズ
- **buff** DerBuffer 構造体の buffer メンバーのコピーへのポインタ
- **buffSz** DerBuffer 構造体の buffer メンバーへ確保されたバッファのサイズ
- **pass** パスワード

See: wc_PemToDer

Return:

- 変換に成功した際には出力バッファに書き込んだデータサイズを返します。
- エラー発生時には負の整数値を返します。

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
)
```

この関数は PEM 形式の証明書を DER 形式に変換します。内部では OpenSSL 互換 API の PemToDer を呼び出します。

Parameters:

- **pem** PEM 形式の証明書を含むバッファへのポインタ
- **pemSz** PEM 形式の証明書を含むバッファのサイズ
- **buff** DER 形式に変換した証明書データの出力先バッファへのポインタ
- **buffSz** 出力先バッファのサイズ
- **type** 証明書のタイプ。asn_public.h で定義の enum CertType の値。

See: wc_PemToDer

Return: バッファに出力したサイズを返します。

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
)
```

この関数は公開鍵を DER 形式で DecodedCert 構造体から取り出します。wc_InitDecodedCert()は DER/ASN.1 エンコードされた証明書を受け付けます。PEM 形式の鍵を DER 形式で取得する場合には、wc_InitDecodedCert() より先に wc_CertPemToDer() を呼び出してください。

Parameters:

- **cert** X.509 証明書を保持した DecodedCert 構造体へのポインタ
- **derKey** DER 形式の公開鍵を出力する先のバッファへのポインタ
- **derKeySz** [IN/OUT] 入力時には derKey で与えられるバッファのサイズ, 出力時には公開鍵のサイズを保持します。もし、derKey が NULL で渡された場合には, derKeySz には必要なバッファサイズが格納され、LENGTH_ONLY_E が戻り値として返されます。

See: wc_GetPubKeyDerFromCert

Return:

- 成功時に 0 を返します。エラー発生時には負の整数を返します。
- LENGTH_ONLY_E derKey が NULL の際に返されます。

C.4.2.32 function wc_EccPrivateKeyDecode

```
int wc_EccPrivateKeyDecode(
    const byte * input,
    word32 * inOutIdx,
    ecc_key * key,
    word32 inSz
)
```

この関数は ECC 秘密鍵を入力バッファから読み込み、解析の後 ecc_key 構造体を作成してそこに鍵を格納します。

Parameters:

- **input** 入力となる秘密鍵データを含んでいるバッファへのポインタ
- **inOutIdx** word32 型変数で内容として入力バッファの処理開始位置を先頭からのインデクス値として保持している。
- **key** デコードされた秘密鍵が格納される初期化済みの ecc_key 構造体へのポインタ
- **inSz** 秘密鍵を含んでいる入力バッファのサイズ

See: wc_RSA_PrivateKeyDecode

Return:

- 0 秘密鍵のデコードと結果の ecc_key 構造体への格納成功時に返されます。
- ASN_PARSE_E 入力バッファの解析あるいは結果の格納時にエラーが発生した場合に返されます。
- MEMORY_E メモリの確保に失敗した際に返されます。
- BUFFER_E 入力された証明書が最大証明書サイズより大きかった場合に返されます。
- ASN_OBJECT_ID_E 証明書が無効なオブジェクト ID を含んでいる場合に返されます。
- ECC_CURVE_OID_E 与えられた秘密鍵の ECC 曲線がサポートされていない場合に返されます。
- ECC_BAD_ARG_E ECC 秘密鍵のフォーマットにエラーがある場合に返されます。
- NOT_COMPILED_IN 秘密鍵が圧縮されていて圧縮鍵が提供されていない場合に返されます。
- MP_MEM 秘密鍵の解析で使用される数学ライブラリがエラーを検出した場合に返されます。
- MP_VAL 秘密鍵の解析で使用される数学ライブラリがエラーを検出した場合に返されます。
- MP_RANGE 秘密鍵の解析で使用される数学ライブラリがエラーを検出した場合に返されます。

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
)
```

この関数は ECC 秘密鍵を DER 形式でバッファに出力します。

Parameters:

- **key** 入力となる ECC 秘密鍵データを含んでいるバッファへのポインタ
- **output** DER 形式の ECC 秘密鍵を出力する先のバッファへのポインタ
- **inLen** DER 形式の ECC 秘密鍵を出力する先のバッファのサイズ

See: wc_RsaKeyToDer

Return:

- ECC 秘密鍵を DER 形式での出力に成功した場合にはバッファへ出力したサイズを返します。
- BAD_FUNC_ARG 出力バッファ output が NULL あるいは inLen がゼロの場合に返します。
- MEMORY_E メモリの確保に失敗した際に返されます。
- BUFFER_E 出力バッファが必要量より小さい
- ASN_UNKNOWN_OID_E ECC 秘密鍵が未知のタイプの場合に返します。
- MP_MEM 秘密鍵の解析で使用される数学ライブラリがエラーを検出した場合に返されます。
- MP_VAL 秘密鍵の解析で使用される数学ライブラリがエラーを検出した場合に返されます。
- MP_RANGE 秘密鍵の解析で使用される数学ライブラリがエラーを検出した場合に返されます。

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
)
```

この関数は入力バッファの ECC 公開鍵を ASN シーケンスをデコードして取り出します。

Parameters:

- **input** DER 形式の公開鍵を含んだバッファへのポインタ
- **inOutIdx** バッファの読み出し位置インデクス値を保持している変数へのポインタ (入力時)。出力時にはこの変数に解析済みのバッファのインデクス値が格納されます。
- **key** ecc_key 構造体へのポインタ
- **inSz** 入力バッファのサイズ

See: wc_ecc_import_x963

Return:

- 0 処理成功時に返します。
- BAD_FUNC_ARG Returns いずれかの引数が NULL の場合に返します。
- ASN_PARSE_E 解析中にエラーが発生した場合に返します。
- ASN_ECC_KEY_E 鍵のインポートでエラーが発生した場合に返します。発生理由については wc_ecc_import_x963() を参照のこと。

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
)
```

この関数は ECC 公開鍵を DER 形式に変換します。処理したバッファのサイズを返します。変換して得られる DER 形式の ECC 公開鍵は出力バッファに格納されます。AlgCurve フラグの指定により、アルゴリズムと曲線情報をヘッダーに含めることができます。

Parameters:

- **key** ecc_key 構造体へのポインタ
- **output** 出力バッファへのポインタ
- **inLen** 出力バッファのサイズ
- **with_AlgCurve** アルゴリズムと曲線情報をヘッダーに含める際には１を指定

See:

- wc_EccKeyToDer
- wc_EccPrivateKeyDecode

Return:

- 成功時には処理したバッファのサイズを返します。
- BAD_FUNC_ARG 出力バッファ output あるいは ecc_key 構造体 key が NULL の場合に返します。
- LENGTH_ONLY_E ECC 公開鍵のサイズ取得に失敗した場合に返します。
- BUFFER_E 出力バッファが必要量より小さい場合に返します。

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
)
```

この関数は ECC 公開鍵を DER 形式に変換します。処理したバッファサイズを返します。変換された DER 形式の ECC 公開鍵は出力バッファに格納されます。AlgCurve フラグの指定により、アルゴリズムと曲線情報をヘッダーに含めることができます。comp パラメータは公開鍵を圧縮して出力するか否かを指定します。

Parameters:

- **key** ecc_key 構造体へのポインタ
- **output** 出力バッファへのポインタ
- **inLen** 出力バッファのサイズ
- **with_AlgCurve** アルゴリズムと曲線情報をヘッダーに含める際には 1 を指定
- **comp** 非ゼロ値の指定時には ECC 公開鍵は圧縮形式で出力されます。ゼロが指定された場合には非圧縮で出力されます。

See:

- wc_EccKeyToDer
- wc_EccPublicKeyDecode

Return:

- 0 成功時には処理したバッファのサイズを返します。
- BAD_FUNC_ARG 出力バッファ output あるいは ecc_key 構造体 key が NULL の場合に返します。
- LENGTH_ONLY_E ECC 公開鍵のサイズ取得に失敗した場合に返します。
- BUFFER_E 出力バッファが必要量より小さい場合に返します。

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_EncodeSignature

```
word32 wc_EncodeSignature(
    byte * out,
    const byte * digest,
    word32 digSz,
    int hashOID
)
```

この関数はデジタル署名をエンコードして出力バッファに出力し、生成された署名のサイズを返します。

Parameters:

- **out** エンコードした署名データを出力する先のバッファへのポインタ
- **digest** 署名データのエンコードに使用するダイジェストへのポインタ
- **digSz** ダイジェストを含んでいるバッファのサイズ
- **hashOID** ハッシュタイプを示すオブジェクト ID。有効な値は: SHAh, SHA256h, SHA384h, SHA512h, MD2h, MD5h, DESb, DES3b, CTC_MD5wRSA, CTC_SHAwRSA, CTC_SHA256wRSA, CTC_SHA384wRSA, CTC_SHA512wRSA, CTC_SHAwECDSA, CTC_SHA256wECDSA, CTC_SHA384wECDSA, と CTC_SHA512wECDSA。

See: none

Return: 成功時には署名を出力バッファに出力し、出力したサイズを返します。

```
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.4.2.38 function wc_GetCTC_HashOID

```
int wc_GetCTC_HashOID(
    int type
)
```

この関数はハッシュタイプに対応したハッシュ OID を返します。例えば、ハッシュタイプが “WC_SHA512” の場合、この関数は “SHA512h” を対応するハッシュ OID として返します。

Parameters:

- **type** ハッシュタイプ。指定可能なタイプ: WC_MD5, WC_SHA, WC_SHA256, WC_SHA384, WC_SHA512, WC_SHA3_224, WC_SHA3_256, WC_SHA3_384, WC_SHA3_512

See: none

Return:

- 成功時には指定されたハッシュタイプと対応するハッシュ OID を返します。
- 0 認識できないハッシュタイプが引数として指定された場合に返します。

Example

```
int hashOID;

hashOID = wc_GetCTC_HashOID(WC_SHA512);
if (hashOID == 0) {
```

```
    // WOLFSSL_SHA512 not defined
}
```

C.4.2.39 function wc_SetCert_Free

```
void wc_SetCert_Free(
    Cert * cert
)
```

この関数はキャッシュされていた Cert 構造体で使用されたメモリとリソースをクリーンアップします。WOLFSSL_CERT_GEN_CACHE が定義されている場合には DecodedCert 構造体が Cert 構造体内部にキャッシュされ、後続する set 系関数の呼び出しの都度 DecodedCert 構造体がパースされることを防ぎます。

Parameters:

- **cert** 未初期化の Cert 構造体へのポインタ

See:

- wc_SetAuthKeyIdFromCert
- wc_SetIssuerBuffer
- wc_SetSubjectBuffer
- wc_SetSubjectRaw
- wc_SetIssuerRaw
- wc_SetAltNamesBuffer
- wc_SetDatesBuffer

Return:

- 0 成功時に返されます。
- BAD_FUNC_ARG 引数として無効な値が渡された場合に返されます。

Example

```
Cert cert; // Initialized certificate structure

wc_SetCert_Free(&cert);
```

C.4.2.40 function wc_GetPkcs8TraditionalOffset

```
int wc_GetPkcs8TraditionalOffset(
    byte * input,
    word32 * inOutIdx,
    word32 sz
)
```

この関数は PKCS#8 の暗号化されていないバッファ内部の従来の秘密鍵の開始位置を検出して返します。

Parameters:

- **input** PKCS#8 の暗号化されていない秘密鍵を保持するバッファへのポインタ
- **inOutIdx** バッファのインデクス位置を保持する変数へのポインタ。入力時にはこの変数の内容はバッファ内部の PKCS#8 の開始位置を示します。出力時には、秘密鍵の先頭位置を保持します。
- **sz** 入力バッファのサイズ

See:

- wc_CreatePKCS8Key
- wc_EncryptPKCS8Key
- wc_DecryptPKCS8Key
- wc_CreateEncryptedPKCS8Key

Return:

- 成功時には従来の秘密鍵の長さを返します。
- エラー時には負の整数値を返します。

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.4.2.41 function wc_CreatePKCS8Key

```
int wc_CreatePKCS8Key(
    byte * out,
    word32 * outSz,
    byte * key,
    word32 keySz,
    int algoID,
    const byte * curveOID,
    word32 oidSz
)
```

この関数は DER 形式の秘密鍵を入力とし、RKCS#8 形式に変換します。また、PKCS#12 のシュロ — ディットキーバッグの作成にも使用できます。RFC5208 を参照のこと。

Parameters:

- **out** 結果の出力先バッファへのポインタ。NULL の場合には必要な出力先バッファのサイズが outSz に格納されます。
- **outSz** 出力先バッファのサイズ
- **key** 従来の DER 形式の秘密鍵を含むバッファへのポインタ
- **keySz** 秘密鍵を含むバッファのサイズ
- **algoID** アルゴリズム ID (RSAk 等の)
- **curveOID** ECC 曲線 OID。RSA 鍵を使用する場合には NULL にすること。
- **oidSz** ECC 曲線 OID のサイズ。curveOID が NULL の場合には 0 にすること。

See:

- wc_GetPkcs8TraditionalOffset
- wc_EncryptPKCS8Key
- wc_DecryptPKCS8Key
- wc_CreateEncryptedPKCS8Key

Return:

- 成功時には出力された PKCS#8 鍵のサイズを返します。
- LENGTH_ONLY_E 出力先バッファ out が NULL として渡された場合にはこのエラーコードが返され、outSz に必要な出力バッファのサイズが格納されます。
- エラー時には負の整数値が返されます。

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.4.2.42 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
)
```

この関数は暗号化されていない PKCS#8 の DER 形式の鍵 (例えば wc_CreatePKCS8Key で生成された鍵) を受け取り、PKCS#8 暗号化形式に変換します。結果として得られた暗号化鍵は wc_DecryptPKCS8Key を使って復号できます。RFC5208 を参照してください。

Parameters:

- **key** 従来の DER 形式の鍵を含んだバッファへのポインタ
- **keySz** 鍵を含んだバッファのサイズ
- **out** 出力結果を格納する先のバッファへのポインタ。NULL の場合には必要な出力先バッファのサイズが outSz に格納されます。
- **outSz** 出力先バッファのサイズ
- **password** パスワードベース暗号化アルゴリズムに使用されるパスワード
- **passwordSz** パスワードのサイズ (NULL 終端文字は含まない)
- **vPKCS** 使用する PKCS のバージョン番号。1 は PKCS12 か PKCS5。
- **pbeOid** パスワードベース暗号化スキームの OID(PBES2 あるいは RFC2898 A.3 にある OID の一つ)
- **encAlgId** 暗号化アルゴリズム ID(例えば AES256CBCb)。
- **salt** ソルト。NULL の場合はランダムに選定したソルトが使用されます。
- **saltSz** ソルトサイズ。salt に NULL を渡した場合には 0 を指定できます。
- **itt** 鍵導出のための繰り返し回数
- **rng** 初期化済みの WC_RNG 構造体へのポインタ
- **heap** 動的メモリ確保のためのヒープ。NULL 指定も可。

See:

- wc_GetPkcs8TraditionalOffset
- wc_CreatePKCS8Key
- wc_DecryptPKCS8Key
- wc_CreateEncryptedPKCS8Key

Return:

- 成功時には出力先バッファに出力された暗号化鍵のサイズを返します。
- LENGTH_ONLY_E 出力先バッファ out が NULL として渡された場合にはこのエラーコードが返され、outSz に必要な出力バッファのサイズが格納されます。
- エラー時には負の整数値が返されます。

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.4.2.43 function wc_DecryptPKCS8Key

```
int wc_DecryptPKCS8Key(
    byte * input,
    word32 sz,
    const char * password,
    int passwordSz
)
```

この関数は暗号化された PKCS#8 の DER 形式の鍵を受け取り、復号して PKCS#8 DER 形式に変換します。wc_EncryptPKCS8Key によって行われた暗号化を元に戻します。RFC5208 を参照してください。入力データは復号データによって上書きされます。

Parameters:

- **input** 入力時には暗号化された PKCS#8 鍵データを含みます。出力時には復号された PKCS#8 鍵データを含みます。
- **sz** 入力バッファのサイズ
- **password** 鍵を暗号化する際のパスワード
- **passwordSz** パスワードのサイズ (NULL 終端文字は含まない)

See:

- wc_GetPkcs8TraditionalOffset
- wc_CreatePKCS8Key
- wc_EncryptPKCS8Key
- wc_CreateEncryptedPKCS8Key

Return:

- 成功時には復号データの長さを返します。
- エラー発生時には負の整数値を返します。

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.4.2.44 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
)
```

この関数は従来の DER 形式の鍵を PKCS#8 フォーマットに変換し、暗号化を行います。この処理には wc_CreatePKCS8Key と wc_EncryptPKCS8Key を使用します。

Parameters:

- **key** 従来の DER 形式の鍵を含んだバッファへのポインタ
- **keySz** 鍵を含んだバッファのサイズ
- **out** 結果を出力する先のバッファへのポインタ。NULL が指定された場合には、必要なバッファサイズが outSz に格納されます。
- **outSz** 結果を出力する先のバッファのサイズ
- **password** パスワードベース暗号アルゴリズムに使用されるパスワード
- **passwordSz** パスワードのサイズ (NULL 終端文字は含まない)
- **vPKCS** 使用する PKCS のバージョン番号。1 は PKCS12 か PKCS5。
- **pbeOid** パスワードベース暗号化スキームの OID(PBES2 あるいは RFC2898 A.3 にある OID の一つ)
- **encAlgId** 暗号化アルゴリズム ID(例えば AES256CBCb)。
- **salt** ソルト。NULL の場合はランダムに選定したソルトが使用されます。
- **saltSz** ソルトサイズ。salt に NULL を渡した場合には 0 を指定できます。
- **itt** 鍵導出のための繰り返し回数
- **rng** 初期化済みの WC_RNG 構造体へのポインタ
- **heap** 動的メモリ確保のためのヒープ。NULL 指定も可。

See:

- wc_GetPkcs8TraditionalOffset
- wc_CreatePKCS8Key

- wc_EncryptPKCS8Key
- wc_DecryptPKCS8Key

Return:

- 成功時には出力した暗号化鍵のサイズを返します。
- LENGTH_ONLY_E もし出力用バッファ out に NULL が渡された場合に返されます。その際には outSz 変数に必要な出力用バッファサイズを格納します。
- エラー発生時には負の整数値を返します。

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.4.2.45 function wc_InitDecodedCert

```
void wc_InitDecodedCert(
    struct DecodedCert * cert,
    const byte * source,
    word32 inSz,
    void * heap
)
```

この関数は cert 引数で与えられた DecodedCert 構造体を初期化します。DER 形式の証明書を含んでいる source 引数の指すポインタから証明書サイズ inSz の長さを内部に保存します。この関数の後に呼び出される wc_ParseCert によって証明書が解析されます。

Parameters:

- **cert** DecodedCert 構造体へのポインタ
- **source** DER 形式の証明書データへのポインタ
- **inSz** 証明書データのサイズ（バイト数）
- **heap** 動的メモリ確保のためのヒープ。NULL 指定も可。

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.4.2.46 function wc_ParseCert

```
int wc_ParseCert(
    DecodedCert * cert,
    int type,
    int verify,
    void * cm
)
```

この関数は DecodedCert 構造体に保存されている DER 形式の証明書を解析し、その構造体に各種フィールドを設定します。DecodedCert 構造体は wc_InitDecodedCert を呼び出して初期化しておく必要があります。この関数はオプションで CertificateManager 構造体へのポインタを受け取り、CA が証明書マネジャーで検索できた場合には、その CA に関する情報も DecodedCert 構造体に追加設定します。

Parameters:

- **cert** 初期化済みの DecodedCert 構造体へのポインタ。
- **type** 証明書タイプ。タイプの設定値については asn_public.h の CertType enum 定義を参照してください。
- **verify** 呼び出し側が証明書の検証を求めていることを指示すフラグです。
- **cm** CertificateManager 構造体へのポインタ。オプションで指定可。NULL でも可。

See:

- wc_InitDecodedCert
- wc_FreeDecodedCert

Return:

- 0 成功時に返します。
- エラー発生時には負の整数値を返します。

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.4.2.47 function wc_FreeDecodedCert

```
void wc_FreeDecodedCert(
    struct DecodedCert * cert
)
```

この関数は wc_InitDecodedCert で初期化済みの DecodedCert 構造体を解放します。

Parameters:

- **cert** 初期化済みの DecodedCert 構造体へのポインタ。

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.4.2.48 function wc_SetTimeCb

```
int wc_SetTimeCb(
    wc_time_cb f
)
```

この関数はタイムコールバック関数を登録します。wolfSSL が現在時刻を必要としたタイミングでこのコールバックを呼び出します。このタイムコールバック関数のプロトタイプ（シグネチャ）は C 標準ライブラリの “time” 関数と同一です。

Parameters:

- **f** タイムコールバック関数ポインタ

See: wc_Time

Return: 0 成功時に返します。

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.4.2.49 function wc_Time

```
time_t wc_Time(
    time_t * t
)
```

この関数は現在時刻を取得します。デフォルトで XTIME マクロ関数を使います。このマクロ関数はプラットフォーム依存です。ユーザーはこのマクロの代わりに wc_SetTimeCb でタイムコールバック関数を使うように設定することができます

Parameters:

- **t** 現在時刻を返却するオプションの time_t 型変数。

See: wc_SetTimeCb

Return: 成功時には現在時刻を返します。

Example

```
time_t currentTime = 0;
currentTime = wc_Time(NULL);
wc_Time(&currentTime);
```

C.4.2.50 function wc_SetCustomExtension

```
int wc_SetCustomExtension(
    Cert * cert,
    int critical,
    const char * oid,
    const byte * der,
    word32 derSz
)
```

この関数は X.509 証明書にカスタム拡張を追加します。注: この関数に渡すポインタ引数が保持する内容は証明書が生成されるまで変更されてはいけません。この関数ではポインタが指す先の内容は別のバッファには複製しません。

Parameters:

- **cert** 初期化済みの DecodedCert 構造体へのポインタ。
- **critical** 0 が指定された場合には追加する拡張はクリティカルとはマークされません。0 以外が指定された場合にはクリティカルとマークされます。
- **oid** ドット区切りの oid 文字列。例えば、“1.2.840.10045.3.1.7”
- **der** 拡張情報の DER エンコードされた内容を含むバッファへのポインタ。
- **derSz** DER エンコードされた内容を含むバッファのサイズ

See:

- wc_InitCert
- wc_SetUnknownExtCallback

Return:

- 0 成功時に返します。
- エラー発生時には負の整数値を返します。

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.4.2.51 function wc_SetUnknownExtCallback

```
int wc_SetUnknownExtCallback(
    DecodedCert * cert,
    wc_UnknownExtCallback cb
)
```

この関数は wolfSSL が証明書の解析中に未知の X.509 拡張に遭遇した際に呼び出すコールバック関数を登録します。コールバック関数のプロトタイプは使用例を参照してください。

Parameters:

- **cert** コールバック関数を登録する対象の DecodedCert 構造体へのポインタ。
- **cb** 登録されるコールバック関数ポインタ

See:

- ParseCert
- wc_SetCustomExtension

Return:

- 0 成功時に返します。
- エラー発生時には負の整数値を返します。

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: OID を構成するドット区切りの数を格納した配列
// oidSz: oid 内の値の数
// crit: 拡張がクリティカルとマークされているか
// der: DER エンコードされている拡張の内容
// derSz: 拡張の内容のサイズ
int myCustomExtCallback(const word16* oid, word32 oidSz, int crit,
                        const unsigned char* der, word32 derSz) {

    // 拡張を解析するロジックはここに記述します

    // NOTE: コールバック関数から 0 を返すと wolfSSL に対してこの拡張を受け入れ可能と
    // 表明することになります。この拡張を処理できると判断できない場合にはエラーを
    // 返してください。クリティカルとマークされている未知の拡張に遭遇した際の標準的
    // な振る舞いは ASN_CRIT_EXT_E を返すことです。
    // 簡潔にするためにこの例ではすべての拡張情報を受け入れ可としていますが、実際には実情に沿
    ↪ うようにロジックを追加してください。
```

```
    return 0;
}
```

C.4.2.52 function wc_CheckCertSigPubKey

```
int wc_CheckCertSigPubKey(
    const byte * cert,
    word32 certSz,
    void * heap,
    const byte * pubKey,
    word32 pubKeySz,
    int pubKeyOID
)
```

この関数は DER 形式の X.509 証明書の署名を与えられた公開鍵を使って検証します。公開鍵は DER 形式で全公開鍵情報を含んだものが求められます。

Parameters:

- **cert** DER 形式の X.509 証明書を含んだバッファへのポインタ
- **certSz** 証明書を含んだバッファのサイズ
- **heap** 動的メモリ確保のためのヒープ。NULL 指定も可。
- **pubKey** DER 形式の公開鍵を含んだバッファへのポインタ
- **pubKeySz** 公開鍵を含んだバッファのサイズ
- **pubKeyOID** 公開鍵のアルゴリズムを特定する OID(すなわち: ECDSAk, DSAk や RSAk)

Return:

- 0 成功時に返します。
- エラー発生時には負の整数値を返します。

C.4.2.53 function wc_Asn1PrintOptions_Init

```
int wc_Asn1PrintOptions_Init(
    Asn1PrintOptions * opts
)
```

この関数は Asn1PrintOptions 構造体を初期化します。

Parameters:

- **opts** プリントのための Asn1PrintOptions 構造体へのポインタ

See:

- wc_Asn1PrintOptions_Set
- wc_Asn1_PrintAll

Return:

- 0 成功時に返します。
- BAD_FUNC_ARG asn1 が NULL の場合に返されます。

Example

```
Asn1PrintOptions opt;

// Initialize ASN.1 print options before use.
wc_Asn1PrintOptions_Init(&opt);
```

C.4.2.54 function wc_Asn1PrintOptions_Set

```
int wc_Asn1PrintOptions_Set(
    Asn1PrintOptions * opts,
    enum Asn1PrintOpt opt,
    word32 val
)
```

この関数は Asn1PrintOptions 構造体にプリント情報を設定します。

Parameters:

- **opts** Asn1PrintOptions 構造体へのポインタ
- **opt** 設定する情報へのポインタ
- **val** 設定値

See:

- wc_Asn1PrintOptions_Init
- wc_Asn1_PrintAll

Return:

- 0 成功時に返します。
- BAD_FUNC_ARG asn1 が NULL の場合に返されます。
- BAD_FUNC_ARG val が範囲外の場合に返されます。

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.4.2.55 function wc_Asn1_Init

```
int wc_Asn1_Init(
    Asn1 * asn1
)
```

この関数は Asn1 構造体を初期化します。

Parameters:

- **asn1** Asn1 構造体へのポインタ

See:

- wc_Asn1_SetFile
- wc_Asn1_PrintAll

Return:

- 0 成功時に返します。
- BAD_FUNC_ARG asn1 が NULL の場合に返されます。

Example

```
Asn1 asn1;

// Initialize ASN.1 parse object before use.
wc_Asn1_Init(&asn1);
```

C.4.2.56 function wc_Asn1_SetFile

```
int wc_Asn1_SetFile(
    Asn1 * asn1,
    XFILE file
)
```

この関数は出力先として使用するファイルを Asn1 構造体にセットします。

Parameters:

- **asn1** Asn1 構造体へのポインタ
- **file** プリント先のファイル

See:

- wc_Asn1_Init
- wc_Asn1_PrintAll

Return:

- 0 成功時に返します。
- BAD_FUNC_ARG asn1 が NULL の場合に返されます。
- BAD_FUNC_ARG file が XBADFILE の場合に返されます。.

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.4.2.57 function wc_Asn1_PrintAll

```
int wc_Asn1_PrintAll(
    Asn1 * asn1,
    Asn1PrintOptions * opts,
    unsigned char * data,
    word32 len
)
```

ASN.1 アイテムをプリントします。

Parameters:

- **asn1** Asn1 構造体へのポインタ
- **opts** Asn1PrintOptions 構造体へのポインタ
- **data** BER/DER 形式のプリント対象データへのポインタ
- **len** プリント対象データのサイズ（バイト数）

See:

- wc_Asn1_Init
- wc_Asn1_SetFile

Return:

- 0 成功時に返します。
- BAD_FUNC_ARG asn1 か opts が NULL の場合に返されます。
- ASN_LEN_E ASN.1 アイテムが長すぎる場合に返されます。
- ASN_DEPTH_E 終了オフセットが無効の場合に返されます。

- ASN_PARSE_E 全の ASN.1 アイテムの解析が完了できなかった場合に返されます。

```
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);

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-ja/blake2.h

C.5.1 Functions

	Name
int	**wc_InitBlake2b**(Blake2b * b2b, word32 digestSz) この関数は Blake2 Hash 関数で使用するための Blake2b 構造を初期化します。
int	**wc_Blake2bUpdate**(Blake2b * b2b, const byte * data, word32 sz) この関数は、与えられた入力データと Blake2B ハッシュを更新します。この関数は、wc_initblake2b の後に呼び出され、最後のハッシュ：wc_blake2bfinal の準備ができているまで繰り返します。
int	**wc_Blake2bFinal**(Blake2b * b2b, byte * final, word32 requestSz) この関数は、以前に供給された入力データの Blake2b ハッシュを計算します。出力ハッシュは長さ REQUESTSZ、あるいは要求された場合は B2B 構造の DigestSZ を使用します。この関数は、wc_initblake2b の後に呼び出され、wc_blake2bupdate は必要な各入力データに対して処理されています。

C.5.2 Functions Documentation

C.5.2.1 function wc_InitBlake2b

```
int wc_InitBlake2b(
    Blake2b * b2b,
    word32 digestSz
)
```

この関数は Blake2 Hash 関数で使用するための Blake2b 構造を初期化します。

Parameters:

- **b2b** 初期化するために Blake2b 構造へのポインタ *Example*

```
Blake2b b2b;
// initialize Blake2b structure with 64 byte digest
wc_InitBlake2b(&b2b, 64);
```

See: wc_Blake2bUpdate

Return: 0 Blake2B 構造の初期化に成功し、ダイジェストサイズを設定したときに返されます。

C.5.2.2 function wc_Blake2bUpdate

```
int wc_Blake2bUpdate(
    Blake2b * b2b,
    const byte * data,
    word32 sz
)
```

この関数は、与えられた入力データと Blake2B ハッシュを更新します。この関数は、wc_initblake2b の後に呼び出され、最後のハッシュ：wc_blake2bfinal の準備ができているまで繰り返します。

Parameters:

- **b2b** 更新する Blake2b 構造へのポインタ
- **data** 追加するデータを含むバッファへのポインタ *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
}
```

See:

- wc_InitBlake2b
- wc_Blake2bFinal

Return:

- 0 与えられたデータを使用して Blake2B 構造を正常に更新すると返されます。
- -1 入力データの圧縮中に障害が発生した場合

C.5.2.3 function wc_Blake2bFinal

```
int wc_Blake2bFinal(
    Blake2b * b2b,
    byte * final,
    word32 requestSz
)
```

この関数は、以前に供給された入力データの Blake2b ハッシュを計算します。出力ハッシュは長さ REQUESTSZ、あるいは要求された場合は B2B 構造の DigestSZ を使用します。この関数は、wc_initblake2b の後に呼び出され、wc_blake2bupdate は必要な各入力データに対して処理されています。

Parameters:

- **b2b** 更新する Blake2b 構造へのポインタ
- **final** Blake2B ハッシュを保存するバッファへのポインタ。長さ requestsz にする必要があります *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
}
```

See:

- wc_InitBlake2b
- wc_Blake2bUpdate

Return:

- 0 Blake2B Hash の計算に成功したときに返されました
- -1 blake2b ハッシュを解析している間に失敗がある場合

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-ja/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) この関数は、次の数学「R = (A ^ P) %M」を実行します。

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
)
```

この関数は、次の数学「R = (A ^ P) %M」を実行します。

Parameters:

- **r** 結果を保持するための構造。

- **a** 電力で上げられる値。
- **p** によって上げる力。
- **m** 使用率 *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
```

See:

- wolfSSL_BN_new
- wolfSSL_BN_free

Return:

- SSL_SUCCESS 数学操作をうまく実行します。
- SSL_FAILURE エラーケースに遭遇した場合

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-ja/camellia.h

C.7.1 Functions

	Name
int	**wc_CamelliaSetKey**(Camellia * cam, const byte * key, word32 len, const byte * iv) この関数は、Camellia オブジェクトのキーと初期化ベクトルを設定し、それを暗号として使用するために初期化します。
int	**wc_CamelliaSetIV**(Camellia * cam, const byte * iv) この関数は、Camellia オブジェクトの初期化ベクトルを設定します。
int	**wc_CamelliaEncryptDirect**(Camellia * cam, byte * out, const byte * in) この機能は、提供された Camellia オブジェクトを使用して 1 ブロック暗号化します。それはバッファーの最初の 16 バイトブロックを解析し、暗号化結果をバッファアウトに格納します。この機能を使用する前に、WC_CAMELLIASETKEY を使用して Camellia オブジェクトを初期化する必要があります。

	Name
int	**wc_CamelliaDecryptDirect**(Camellia * cam, byte * out, const byte * in) この機能は、提供された Camellia オブジェクトを使用して 1 ブロック復号化します。それはバッファ内の最初の 16 バイトブロックを解析し、それを復号化し、結果をバッファアウトに格納します。この機能を使用する前に、WC_CAMELLIASETKEY を使用して Camellia オブジェクトを初期化する必要があります。
int	**wc_CamelliaCbcEncrypt**(Camellia * cam, byte * out, const byte * in, word32 sz) この関数は、バッファーの平文を暗号化し、その出力をバッファ OUT に格納します。暗号ブロックチェーン（CBC）を使用して Camellia を使用してこの暗号化を実行します。
int	**wc_CamelliaCbcDecrypt**(Camellia * cam, byte * out, const byte * in, word32 sz) この関数は、バッファ内の暗号文を復号化し、その出力をバッファ OUT に格納します。暗号ブロックチェーン（CBC）を搭載した Camellia を使用してこの復号化を実行します。

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
)
```

この関数は、Camellia オブジェクトのキーと初期化ベクトルを設定し、それを暗号として使用するために初期化します。

Parameters:

- **cam** キーと IV を設定する椿構造へのポインタ
- **key** 暗号化と復号化に使用する 16,24、または 32 バイトのキーを含むバッファへのポインタ
- **len** 渡されたキーの長さ *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
}
```

See:

- wc_CamelliaEncryptDirect
- wc_CamelliaDecryptDirect

- wc_CamelliaCbcEncrypt
- wc_CamelliaCbcDecrypt

Return:

- 0 キーと初期化ベクトルを正常に設定すると返されます
- BAD_FUNC_ARG 入力引数の 1 つがエラー処理がある場合に返されます
- MEMORY_E xmalloc でメモリを割り当てるエラーがある場合

C.7.2.2 function wc_CamelliaSetIV

```
int wc_CamelliaSetIV(
    Camellia * cam,
    const byte * iv
)
```

この関数は、Camellia オブジェクトの初期化ベクトルを設定します。

Parameters:

- **cam** IV を設定する椿構造へのポインタ *Example*

```
Camellia cam;
byte iv[16];
// initialize iv
if( wc_CamelliaSetIV(&cam, iv) != 0) {
// error initializing camellia structure
}
```

See: wc_CamelliaSetKey

Return:

- 0 キーと初期化ベクトルを正常に設定すると返されます
- BAD_FUNC_ARG 入力引数の 1 つがエラー処理がある場合に返されます

C.7.2.3 function wc_CamelliaEncryptDirect

```
int wc_CamelliaEncryptDirect(
    Camellia * cam,
    byte * out,
    const byte * in
)
```

この機能は、提供された Camellia オブジェクトを使用して 1 ブロック暗号化します。それはバッファーの最初の 16 バイトブロックを解析し、暗号化結果をバッファアウトに格納します。この機能を使用する前に、WC_CAMELLIASETKEY を使用して Camellia オブジェクトを初期化する必要があります。

Parameters:

- **cam** 暗号化に使用する椿構造へのポインタ
- **out** 暗号化されたブロックを保存するバッファへのポインタ *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);
```

See: wc_CamelliaDecryptDirect

Return: none いいえ返します。

C.7.2.4 function wc_CamelliaDecryptDirect

```
int wc_CamelliaDecryptDirect(
    Camellia * cam,
    byte * out,
    const byte * in
)
```

この機能は、提供された Camellia オブジェクトを使用して 1 ブロック復号化します。それはバッファ内の最初の 16 バイトブロックを解析し、それを復号化し、結果をバッファアウトに格納します。この機能を使用する前に、WC_CAMELLIASETKEY を使用して Camellia オブジェクトを初期化する必要があります。

Parameters:

- **cam** 暗号化に使用する椿構造へのポインタ
- **out** 復号化された平文ブロックを保存するバッファへのポインタ *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);
```

See: wc_CamelliaEncryptDirect

Return: none いいえ返します。

C.7.2.5 function wc_CamelliaCbcEncrypt

```
int wc_CamelliaCbcEncrypt(
    Camellia * cam,
    byte * out,
    const byte * in,
    word32 sz
)
```

この関数は、バッファーの平文を暗号化し、その出力をバッファ OUT に格納します。暗号ブロックチェーン（CBC）を使用して Camellia を使用してこの暗号化を実行します。

Parameters:

- **cam** 暗号化に使用する椿構造へのポインタ
- **out** 暗号化された暗号文を保存するバッファへのポインタ
- **in** 暗号化する平文を含むバッファへのポインタ *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));
```

See: wc_CamelliaCbcDecrypt

Return: none いいえ返します。

C.7.2.6 function wc_CamelliaCbcDecrypt

```
int wc_CamelliaCbcDecrypt(
    Camellia * cam,
    byte * out,
    const byte * in,
    word32 sz
)
```

この関数は、バッファ内の暗号文を復号化し、その出力をバッファ OUT に格納します。暗号ブロックチェーン（CBC）を搭載した Camellia を使用してこの復号化を実行します。

Parameters:

- **cam** 暗号化に使用する椿構造へのポインタ
- **out** 復号化されたメッセージを保存するバッファへのポインタ
- **in** 暗号化された暗号文を含むバッファへのポインタ *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));
```

See: wc_CamelliaCbcEncrypt

Return: none いいえ返します。

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-ja/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 outAuth-Tag[CHACHA20_POLY1305_AEAD_AUTHTAG_SIZE]) この関数は、Chacha20 Stream 暗号を使用して、Output BufferText に入力メッセージ、InPleaintext を暗号化します。また、Poly-1305 認証（暗号テキスト）を実行し、生成した認証タグを出力バッファ OutauthTag に格納します。
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) この関数は、Chacha20 Stream 暗号を使用して、出力バッファ OutpleAntext に復号したデータを出力します。また、Poly-1305 認証を実行し、指定された inAuthTag を inAAD で生成された認証（任意の長さの追加認証データ）と比較します。注：生成された認証タグが提供された認証タグと一致しない場合、テキストは復号されません。

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]
)
```

この関数は、Chacha20 Stream 暗号を使用して、Output BufferText に入力メッセージ、InPleaintext を暗号化します。また、Poly-1305 認証（暗号テキスト）を実行し、生成した認証タグを出力バッファ OutauthTag に格納します。

Parameters:

- **inKey** 暗号化に使用する 32 バイトの鍵を含むバッファへのポインタ
- **inIv** 暗号化に使用する 12 バイトの IV を含むバッファへのポインタ

- **inAAD** 任意の長さの追加認証データ（AAD）を含むバッファへのポインタ
- **inAADLen** 入力 AAD の長さ
- **inPlaintext** 暗号化する平文を含むバッファへのポインタ
- **inPlaintextLen** 暗号化するプレーンテキストの長さ
- **outCiphertext** 暗号文を保存するバッファーへのポインタ *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
}
```

See:

- wc_ChaCha20Poly1305_Decrypt
- wc_ChaCha_*
- wc_Poly1305*

Return:

- 0 メッセージの暗号化に成功したら返されます
- BAD_FUNC_ARG 暗号化プロセス中にエラーがある場合

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
)
```

この関数は、Chacha20 Stream 暗号を使用して、出力バッファ OutpleAntext に復号したデータを出力します。また、Poly-1305 認証を実行し、指定された inAuthTag を inAAD で生成された認証（任意の長さの追加認証データ）と比較します。注：生成された認証タグが提供された認証タグと一致しない場合、テキストは復号されません。

Parameters:

- **inKey** 復号に使用する 32 バイトの鍵を含むバッファへのポインタ
- **inIv** 復号に使用する 12 バイトの IV を含むバッファへのポインタ
- **inAAD** 任意の長さの追加認証データ（AAD）を含むバッファへのポインタ
- **inAADLen** 入力 AAD の長さ
- **inCiphertext** 復号する暗号文を含むバッファへのポインタ
- **outCiphertextLen** 復号する暗号文の長さ

- **inAuthTag** 認証のための 16 バイトのダイジェストを含むバッファへのポインタ *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
}
```

See:

- wc_ChaCha20Poly1305_Encrypt
- wc_ChaCha_*
- wc_Poly1305*

Return:

- 0 メッセージの復号に成功したときに返されました
- BAD_FUNC_ARG 関数引数のいずれかが予想されるものと一致しない場合に返されます
- MAC_CMP_FAILED_E 生成された認証タグが提供されている inAuthTag と一致しない場合に返されます。

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-ja/chacha.h

C.9.1 Functions

	Name
int	**wc_Chacha_SetIV**(ChaCha * ctx, const byte * inIv, word32 counter) この関数は Chacha オブジェクトの初期化ベクトル（nonce）を設定し、暗号として使用するために初期化します。WC_CHACHA_SETKEY を使用して、キーが設定された後に呼び出されるべきです。暗号化の各ラウンドに差し違いを使用する必要があります。
int	**wc_Chacha_Process**(ChaCha * ctx, byte * cipher, const byte * plain, word32 msglen) この関数は、バッファ入力からテキストを処理し、暗号化または復号化し、結果をバッファ出力に格納します。
int	**wc_Chacha_SetKey**(ChaCha * ctx, const byte * key, word32 keySz) この関数は Chacha オブジェクトのキーを設定し、それを暗号として使用するために初期化します。NONCE を WC_CHACHA_SETIV で設定する前に、WC_CHACHA_PROCESS を使用した暗号化に使用する前に呼び出す必要があります。

C.9.2 Functions Documentation

C.9.2.1 function wc_Chacha_SetIV

```
int wc_Chacha_SetIV(
    ChaCha * ctx,
    const byte * inIv,
    word32 counter
)
```

この関数は Chacha オブジェクトの初期化ベクトル（nonce）を設定し、暗号として使用するために初期化します。WC_CHACHA_SETKEY を使用して、キーが設定された後に呼び出されるべきです。暗号化の各ラウンドに差し違いを使用する必要があります。

Parameters:

- **ctx** IV を設定する Chacha 構造へのポインタ
- **inIv** Chacha 構造を初期化するための 12 バイトの初期化ベクトルを含むバッファへのポインタ

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
}
```

See:

- wc_Chacha_SetKey
- wc_Chacha_Process

Return:

- 0 初期化ベクトルを正常に設定すると返されます

- BAD_FUNC_ARG CTX 入力引数の処理中にエラーが発生した場合

C.9.2.2 function wc_Chacha_Process

```
int wc_Chacha_Process(
    ChaCha * ctx,
    byte * cipher,
    const byte * plain,
    word32 msglen
)
```

この関数は、バッファ入力からテキストを処理し、暗号化または復号化し、結果をバッファ出力に格納します。

Parameters:

- **ctx** IV を設定する Chacha 構造へのポインタ
- **output** 出力暗号文または復号化された平文を保存するバッファへのポインタ
- **input** 暗号化する入力平文を含むバッファへのポインタまたは復号化する入力暗号文 *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
}
```

See:

- wc_Chacha_SetKey
- wc_Chacha_Process

Return:

- 0 入力の暗号化または復号化に成功したときに返されます
- BAD_FUNC_ARG CTX 入力引数の処理中にエラーが発生した場合

C.9.2.3 function wc_Chacha_SetKey

```
int wc_Chacha_SetKey(
    ChaCha * ctx,
    const byte * key,
    word32 keySz
)
```

この関数は Chacha オブジェクトのキーを設定し、それを暗号として使用するために初期化します。NONCE を WC_CHACHA_SETIV で設定する前に、WC_CHACHA_PROCESS を使用した暗号化に使用する前に呼び出す必要があります。

Parameters:

- **ctx** キーを設定する Chacha 構造へのポインタ
- **key** Chacha 構造を初期化するための 16 または 32 バイトのキーを含むバッファへのポインタ *Example*

```
ChaCha enc;
byte key[] = { // initialize key };
```

```
if( wc_Chacha_SetKey(&enc, key, sizeof(key)) != 0) {
    // error initializing ChaCha structure
}
```

See:

- wc_Chacha_SetIV
- wc_Chacha_Process

Return:

- 0 キーの設定に成功したときに返されます
- BAD_FUNC_ARG CTX 入力引数の処理中にエラーが発生した場合、またはキーが 16 または 32 バイトの長さがある場合

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-ja/cmac.h

C.10.1 Functions

	Name
int	**wc_InitCmac**(Cmac * cmac, const byte *key*、*word32 keySz*、*int type*、*void* unused)Cmac 構造体をデフォルト値で初期化します
int	**wc_InitCmac_ex**(Cmac * cmac, const byte * key, word32 keySz, int type, void *unused*、*void* heap, int devId)Cmac 構造体をデフォルト値で初期化します
int	**wc_CmacUpdate**(Cmac * cmac, const byte * in, word32 inSz) 暗号ベースのメッセージ認証コード入力データを追加
int	**wc_CmacFinalNoFree**(Cmac * cmac, byte * out, word32 * outSz) 暗号ベースのメッセージ認証コードの最終結果を生成します。ただし、使用したコンテキストのクリーンアップは行いません。
int	**wc_CmacFinal**(Cmac * cmac, byte * out, word32 * outSz) 暗号ベースのメッセージ認証コードを使用して最終結果を生成します。加えて、内部で wc_CmacFree を呼び出してコンテキスとをクリーンアップします。
int	**wc_CmacFree**(Cmac * cmac)CMAC 処理中に Cmac 構造体内に確保されたオブジェクトを開放します。

	Name
int	**wc_AesCmacGenerate**(byte * out, word32 * outSz, const byte *in*、*word32 inSz*、*const byte* key, word32 keySz)CMAC を生成するためのシングルショット関数
int	**wc_AesCmacVerify**(const byte * check, word32 checkSz, const byte *in*、*word32 inSz*、*const byte* key, word32 keySz)CMAC を検証するためのシングルショット関数
int	**wc_CMAC_Grow**(Cmac * cmac, const byte * in, int inSz)WOLFSSL_HASH_KEEP マクロ定義時のみ使用可能。ハードウェアがシングルショットを必要とし、更新をメモリにキャッシュする必要がある場合に使用します。

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
)
```

Cmac 構造体をデフォルト値で初期化します

Parameters:

- **cmac** Cmac 構造体へのポインタ
- **key** 鍵データへのポインタ
- **keySz** 鍵データのサイズ (16、24、または 32)
- **type** 常に WC_CMAC_AES（=1）
- **unused** 使用されていません。互換性に関する将来の潜在的な使用のために存在します

See:

- wc_InitCmac_ex
- wc_CmacUpdate
- wc_CmacFinal
- wc_CmacFinalNoFree
- wc_CmacFree

Return: 成功したら 0 を返します

例

```
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
)
```

Cmac 構造体をデフォルト値で初期化します

Parameters:

- **cmac** Cmac 構造体へのポインタ
- **key** 鍵データへのポインタ
- **keySz** 鍵データのサイズ (16、24、または 32)
- **type** 常に WC_CMAC_AES（=1）
- **unused** 使用されていません。互換性に関する将来の潜在的な使用のために存在します
- **heap** 動的割り当てに使用されるヒープヒントへのポインタ。通常、スタティックメモリオプションで使用されます。NULL にすることができます。
- **devId** 非同期ハードウェアで使用する ID。非同期ハードウェアを使用していない場合は、INVALID_DEVID に設定します。

See:

- wc_InitCmac_ex
- wc_CmacUpdate
- wc_CmacFinal
- wc_CmacFinalNoFree
- wc_CmacFree

Return: 成功したら 0 を返します

例

```
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
)
```

暗号ベースのメッセージ認証コード入力データを追加

Parameters:

- **cmac** Cmac 構造体へのポインタ
- **in** 処理する入力データへのポインタ
- **inSz** 入力データのサイズ

See:

- wc_InitCmac
- wc_CmacFinal
- wc_CmacFinalNoFree
- wc_CmacFree

Return: 成功したら 0 を返します

例

```
ret = wc_CmacUpdate(cmac、in、inSz);
```

C.10.2.4 function wc_CmacFinalNoFree

```
int wc_CmacFinalNoFree(
    Cmac * cmac,
    byte * out,
    word32 * outSz
)
```

暗号ベースのメッセージ認証コードの最終結果を生成します。ただし、使用したコンテキストのクリーンアップは行いません。

Parameters:

- **cmac** Cmac 構造体へのポインタ
- **out** 結果を格納するバッファへのポインタ
- **outSz** 結果出力先バッファのサイズ

See:

- wc_InitCmac
- wc_CmacFinal
- wc_CmacFinalNoFree
- wc_CmacFree

Return: 成功したら 0 を返します

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
)
```

暗号ベースのメッセージ認証コードを使用して最終結果を生成します。加えて、内部で wc_CmacFree を呼び出してコンテキスとをクリーンアップします。

Parameters:

- **cmac** Cmac 構造体へのポインタ
- **out** 結果を格納するバッファへのポインタ
- **outSz** 結果出力先バッファのサイズ

See:

- wc_InitCmac
- wc_CmacFinal
- wc_CmacFinalNoFree
- wc_CmacFree

Return: 成功したら 0 を返します

例

```
ret = wc_CmacFinal(cmac, out, &outSz);
```

C.10.2.6 function wc_CmacFree

```
int wc_CmacFree(
    Cmac * cmac
)
```

CMAC 処理中に Cmac 構造体内に確保されたオブジェクトを開放します。

Parameters:

- **cmac** Cmac 構造体へのポインタ

See:

- wc_InitCmac
- wc_CmacFinalNoFree
- wc_CmacFinal
- wc_CmacFree

Return: 成功したら 0 を返します

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
)
```

CMAC を生成するためのシングルショット関数

Parameters:

- **out** 結果の出力先バッファへのポインタ
- **outSz** 出力のポインタサイズ (in/out)
- **in** 処理する入力データのポインタ
- **inSz** 入力データのサイズ
- **key** 鍵データへのポインタ
- **keySz** 鍵データのサイズ (16、24、または 32)

See: wc_AesCmacVerify

Return: 成功したら 0 を返します

例

```
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
)
```

CMAC を検証するためのシングルショット関数

Parameters:

- **check** 検証対象となる CMAC 処理結果データへのポインタ
- **checkSz** CMAC 処理結果データのサイズ
- **in** 処理する入力データのポインタ
- **inSz** 入力データのサイズ
- **key** 鍵データへのポインタ
- **keySz** 鍵データのサイズ (16、24、または 32)

See: wc_AesCmacGenerate

Return: 成功したら 0 を返します

例

```
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
)
```

WOLFSSL_HASH_KEEP マクロ定義時のみ使用可能。ハードウェアがシングルショットを必要とし、更新をメモリにキャッシュする必要がある場合に使用します。

Parameters:

- **cmac** Cmac 構造体へのポインタ
- **in** 処理する入力データへのポインタ
- **inSz** 入力データのサイズ

Return: 成功したら 0 を返します

例

```
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-ja/coding.h

C.11.1 Functions

	Name
int	**Base64_Decode**(const byte * in, word32 inLen, byte * out, word32 * outLen) この機能は、与えられた BASS64 符号化入力、IN、および出力バッファを出力バッファ OUT に格納します。また、変数 outlen 内の出力バッファに書き込まれたサイズも設定します。
int	**Base64_Encode**(const byte * in, word32 inLen, byte * out, word32 * outLen) この機能は与えられた入力を符号化し、符号化結果を出力バッファ OUT に格納します。エスケープ%0A 行末の代わりに、従来の' N' 行の終わりを持つデータを書き込みます。正常に完了すると、この機能はまた、出力バッファに書き込まれたバイト数に統一されます。
int	**Base64_EncodeEsc**(const byte * in, word32 inLen, byte * out, word32 * outLen) この機能は与えられた入力を符号化し、符号化結果を出力バッファ OUT に格納します。それは' n " 行の終わりではなく、%0a エスケープ行の終わりを持つデータを書き込みます。正常に完了すると、この機能はまた、出力バッファに書き込まれたバイト数に統一されます。

	Name
int	**Base64_Encode_NoNl**(const byte * in, word32 inLen, byte * out, word32 * outLen) この機能は与えられた入力を符号化し、符号化結果を出力バッファ OUT に格納します。それは新しい行なしでデータを書き込みます。正常に完了すると、この関数はまた、出力バッファに書き込まれたバイト数に統一されたものを設定します
int	**Base16_Decode**(const byte * in, word32 inLen, byte * out, word32 * outLen) この機能は、与えられた BASE16 符号化入力、IN、および出力バッファへの結果を記憶する。また、変数 outlen 内の出力バッファに書き込まれたサイズも設定します。
int	**Base16_Encode**(const byte * in, word32 inLen, byte * out, word32 * outLen)BASE16 出力へのエンコード入力。

C.11.2 Functions Documentation

C.11.2.1 function Base64_Decode

```
int Base64_Decode(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 * outLen
)
```

この機能は、与えられた BASS64 符号化入力、IN、および出力バッファを出力バッファ OUT に格納します。また、変数 outlen 内の出力バッファに書き込まれたサイズも設定します。

Parameters:

- **in** デコードする入力バッファへのポインタ
- **inLen** デコードする入力バッファの長さ
- **out** デコードされたメッセージを保存する出力バッファへのポインタ *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
}
```

See:

- Base64_Encode
- Base16_Decode

Return:

- 0 Base64 エンコード入力の復号化に成功したときに返されます
- BAD_FUNC_ARG 復号化された入力を保存するには、出力バッファが小さすぎる場合は返されます。

- ASN_INPUT_E 入力バッファ内の文字が BASE64 範囲（[A-ZA-Z0-9 + / =]）の外側にある場合、または BASE64 エンコード入力に無効な行が終了した場合

C.11.2.2 function Base64_Encode

```
int Base64_Encode(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 * outLen
)
```

この機能は与えられた入力を符号化し、符号化結果を出力バッファ OUT に格納します。エスケープ%0A 行末の代わりに、従来の' N' 行の終わりを持つデータを書き込みます。正常に完了すると、この機能はまた、出力バッファに書き込まれたバイト数に統一されます。

Parameters:

- **in** エンコードする入力バッファへのポインタ
- **inLen** エンコードする入力バッファの長さ
- **out** エンコードされたメッセージを保存する出力バッファへのポインタ *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
}
```

See:

- Base64_EncodeEsc
- Base64_Decode

Return:

- 0 Base64 エンコード入力の復号化に成功したときに返されます
- BAD_FUNC_ARG 出力バッファが小さすぎてエンコードされた入力を保存する場合は返されます。
- BUFFER_E 出力バッファがエンコード中に部屋の外に実行された場合に返されます。

C.11.2.3 function Base64_EncodeEsc

```
int Base64_EncodeEsc(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 * outLen
)
```

この機能は与えられた入力を符号化し、符号化結果を出力バッファ OUT に格納します。それは' n " 行の終わりではなく、%0a エスケープ行の終わりを持つデータを書き込みます。正常に完了すると、この機能はまた、出力バッファに書き込まれたバイト数に統一されます。

Parameters:

- **in** エンコードする入力バッファへのポインタ
- **inLen** エンコードする入力バッファの長さ
- **out** エンコードされたメッセージを保存する出力バッファへのポインタ *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
}
```

See:

- Base64_Encode
- Base64_Decode

Return:

- 0 Base64 エンコード入力の復号化に成功したときに返されます
- BAD_FUNC_ARG 出力バッファが小さすぎてエンコードされた入力を保存する場合は返されます。
- BUFFER_E 出力バッファがエンコード中に部屋の外に実行された場合に返されます。
- ASN_INPUT_E 入力メッセージのデコードの処理中にエラーが発生した場合

C.11.2.4 function Base64_Encode_NoNl

```
int Base64_Encode_NoNl(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 * outLen
)
```

この機能は与えられた入力を符号化し、符号化結果を出力バッファ OUT に格納します。それは新しい行なしでデータを書き込みます。正常に完了すると、この関数はまた、出力バッファに書き込まれたバイト数に統一されたものを設定します

Parameters:

- **in** エンコードする入力バッファへのポインタ
- **inLen** エンコードする入力バッファの長さ
- **out** エンコードされたメッセージを保存する出力バッファへのポインタ *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
}
```

See:

- Base64_Encode
- Base64_Decode

Return:

- 0 Base64 エンコード入力の復号化に成功したときに返されます
- BAD_FUNC_ARG 出力バッファが小さすぎてエンコードされた入力を保存する場合は返されます。
- BUFFER_E 出力バッファがエンコード中に部屋の外に実行された場合に返されます。
- ASN_INPUT_E 入力メッセージのデコードの処理中にエラーが発生した場合

C.11.2.5 function Base16_Decode

```
int Base16_Decode(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 * outLen
)
```

この機能は、与えられた BASE16 符号化入力、IN、および出力バッファへの結果を記憶する。また、変数 outlen 内の出力バッファに書き込まれたサイズも設定します。

Parameters:

- **in** デコードする入力バッファへのポインタ
- **inLen** デコードする入力バッファの長さ
- **out** デコードされたメッセージを保存する出力バッファへのポインタ *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
}
```

See:

- Base64_Encode
- Base64_Decode
- Base16_Encode

Return:

- 0 Base16 エンコード入力の復号にうまく復号化したときに返されます
- BAD_FUNC_ARG 出力バッファが復号化された入力を保存するにも小さすぎる場合、または入力長が 2 つの倍数でない場合に返されます。
- ASN_INPUT_E 入力バッファ内の文字が BASE16 の範囲外にある場合は返されます（[0-9a-f]）

C.11.2.6 function Base16_Encode

```
int Base16_Encode(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 * outLen
)
```

BASE16 出力へのエンコード入力。

Parameters:

- **in** エンコードされる入力バッファへのポインタ。
- **inLen** 入力バッファの長さ
- **out** 出力バッファへのポインタ。*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
}
```

See:

- Base64_Encode
- Base64_Decode
- Base16_Decode

Return:

- 0 成功
- BAD_FUNC_ARG IN、OUT、または outlen が NULL の場合、または outlen が Inlen Plus 1 を超えている場合は返します。

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-ja/compress.h

C.12.1 Functions

	Name
int	**wc_Compress**(byte * out, word32 outSz, const byte * in, word32 inSz, word32 flags) この関数は、ハフマン符号化を用いて与えられた入力データを圧縮し、出力を OUT に格納する。出力バッファは、圧縮が可能でないことが存在するため、出力バッファが入力バッファよりも大きいはずです。これはまだルックアップテーブルを必要とします。出力バッファに対して SRCSZ + 0.1%+ 12 を割り当てることをお勧めします。
int	**wc_DeCompress**(byte * out, word32 outSz, const byte * in, word32 inSz) この関数は、ハフマン符号化を用いて所定の圧縮データを解凍し、出力を OUT に格納する。

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
)
```

この関数は、ハフマン符号化を用いて与えられた入力データを圧縮し、出力を OUT に格納する。出力バッファは、圧縮が可能でないことが存在するため、出力バッファが入力バッファよりも大きいはずです。これはまだルックアップテーブルを必要とします。出力バッファに対して SRCSZ + 0.1%+ 12 を割り当てることをお勧めします。

Parameters:

- **out** 圧縮データを格納する出力バッファへのポインタ
- **outSz** 出力バッファで保存されているサイズ
- **in** 圧縮するメッセージを含むバッファへのポインタ
- **inSz** 圧縮する入力メッセージのサイズ *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
}
```

See: wc_DeCompress

Return:

- On 入力データの圧縮に成功し、出力バッファに格納されているバイト数を返します。
- COMPRESS_INIT_E 圧縮のためにストリームの初期化中にエラーがある場合
- COMPRESS_E 圧縮中にエラーが発生した場合は返されます

C.12.2.2 function wc_DeCompress

```
int wc_DeCompress(
    byte * out,
    word32 outSz,
    const byte * in,
    word32 inSz
)
```

この関数は、ハフマン符号化を用いて所定の圧縮データを解凍し、出力を OUT に格納する。

Parameters:

- **out** 解凍されたデータを格納する出力バッファへのポインタ
- **outSz** 出力バッファで保存されているサイズ
- **in** 解凍するメッセージを含むバッファへのポインタ *Example*

```
byte compressed[] = { // initialize compressed message };
byte decompressed[MAX_MESSAGE_SIZE];
```

```
if( wc_DeCompress(decompressed, sizeof(decompressed),
compressed, sizeof(compressed)) != 0 ) {
    // error decompressing data
}
```

See: wc_Compress

Return:

- Success 入力データの解凍に成功した場合は、出力バッファに格納されているバイト数を返します。
- COMPRESS_INIT_E: 圧縮のためにストリームの初期化中にエラーがある場合
- COMPRESS_E: 圧縮中にエラーが発生した場合は返されます

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-ja/cryptocb.h

C.13.1 Functions

	Name
int	**wc_CryptoCb_RegisterDevice**(int devId, CryptoDevCallbackFunc cb, void * ctx) この関数は、Crypto Operations をキーストア、Secure Element、HSM、PKCS11 または TPM などの外部ハードウェアにオフロードするための固有のデバイス識別子（DEVID）とコールバック関数を登録します。Crypto コールバックの STSAFE の場合は、wolfcrypt / src / port / st / stsafe.c と wolfssl_stsafe_cryptodevcb 関数を参照してください。TPM ベースの Crypto コールバックの例では、wolftpm src / tpm2_wrap.c の wolftpm2_cryptodevcb 関数を参照してください。
void	**wc_CryptoCb_UnRegisterDevice**(int devId) この関数は、固有のデバイス識別子（devid）コールバック関数を除外します。

C.13.2 Functions Documentation

C.13.2.1 function wc_CryptoCb_RegisterDevice

```
int wc_CryptoCb_RegisterDevice(
    int devId,
    CryptoDevCallbackFunc cb,
    void * ctx
)
```

この関数は、Crypto Operations をキーストア、Secure Element、HSM、PKCS11 または TPM などの外部ハードウェアにオフロードするための固有のデバイス識別子（DEVID）とコールバック関数を登録します。Crypto コールバックの STSAFE の場合は、wolfcrypt / src / port / st / stsafe.c と wolfssl_stsafe_cryptodevcb 関数を参照してください。TPM ベースの Crypto コールバックの例では、wolftpm src / tpm2_wrap.c の wolftpm2_cryptodevcb 関数を参照してください。

Parameters:

- **devId** -2（invalid_devid）ではなく、一意の値ではありません。*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);
```

See:

- wc_CryptoCb_UnRegisterDevice
- wolfSSL_SetDevId
- wolfSSL_CTX_SetDevId

Return:

- CRYPTOCB_UNAVAILABLE ソフトウェア暗号を使用するためにフォールバックする
- 0 成功のために
- negative 失敗の値

C.13.2.2 function wc_CryptoCb_UnRegisterDevice

```
void wc_CryptoCb_UnRegisterDevice(
    int devId
)
```

この関数は、固有のデバイス識別子（devid）コールバック関数を除外します。

See:

- wc_CryptoCb_RegisterDevice
- wolfSSL_SetDevId
- wolfSSL_CTX_SetDevId

Return: none いいえ返します。*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-ja/curve25519.h

C.14.1 Functions

	Name
int	**wc_curve25519_make_key**(WC_RNG * rng, int keysize, curve25519_key * key) この関数は、与えられたサイズ（Keysize）の指定された乱数発生器 RNG を使用して Curve25519 キーを生成し、それを指定された Curve25519_Key 構造体に格納します。キー構造が WC_CURVE25519_INIT（）を介して初期化された後に呼び出されるべきです。
int	**wc_curve25519_shared_secret**(curve25519_key * private_key, curve25519_key * public_key, byte * out, word32 * outlen) この関数は、秘密の秘密鍵と受信した公開鍵を考えると、共有秘密鍵を計算します。生成された秘密鍵をバッファアウトに保存し、ounlent の秘密鍵の変数を割り当てます。ビッグエンディアンのみをサポートします。
int	**wc_curve25519_shared_secret_ex**(curve25519_key * private_key, curve25519_key * public_key, byte * out, word32 * outlen, int endian) この関数は、秘密の秘密鍵と受信した公開鍵を考えると、共有秘密鍵を計算します。生成された秘密鍵をバッファアウトに保存し、ounlent の秘密鍵の変数を割り当てます。ビッグ・リトルエンディアンの両方をサポートします。
int	**wc_curve25519_init**(curve25519_key * key) この関数は Curve25519 キーを初期化します。構造のキーを生成する前に呼び出されるべきです。
void	**wc_curve25519_free**(curve25519_key * key) この関数は Curve25519 オブジェクトを解放します。*Example*
int	**wc_curve25519_import_private**(const byte * priv, word32 privSz, curve25519_key * key) この関数は Curve25519 秘密鍵のみをインポートします。（ビッグエンディアン）。
int	**wc_curve25519_import_private_ex**(const byte * priv, word32 privSz, curve25519_key * key, int endian)CURVE25519 秘密鍵のインポートのみ。（大きなエンディアン）。
int	**wc_curve25519_import_private_raw**(const byte * priv, word32 privSz, const byte * pub, word32 pubSz, curve25519_key * key) この関数は、パブリック秘密鍵ペアを Curve25519_Key 構造体にインポートします。ビッグエンディアンのみ。
int	**wc_curve25519_import_private_raw_ex**(const byte * priv, word32 privSz, const byte * pub, word32 pubSz, curve25519_key * key, int endian) この関数は、パブリック秘密鍵ペアを Curve25519_Key 構造体にインポートします。ビッグ・リトルエンディアンの両方をサポートします。

	Name
int	**wc_curve25519_export_private_raw**(curve25519_key * key, byte * out, word32 * outLen) この関数は Curve25519_Key 構造体から秘密鍵をエクスポートし、それを指定されたアウトバッファに格納します。また、エクスポートされたキーのサイズになるように概要を設定します。ビッグエンディアンのみ。
int	**wc_curve25519_export_private_raw_ex**(curve25519_key * key, byte * out, word32 * outLen, int endian) この関数は Curve25519_Key 構造体から秘密鍵をエクスポートし、それを指定されたアウトバッファに格納します。また、エクスポートされたキーのサイズになるように概要を設定します。それがビッグ・リトルエンディアンかを指定できます。
int	**wc_curve25519_import_public**(const byte * in, word32 inLen, curve25519_key * key) この関数は、指定されたバッファから公開鍵をインポートし、それを Curve25519_Key 構造体に格納します。
int	**wc_curve25519_import_public_ex**(const byte * in, word32 inLen, curve25519_key * key, int endian) この関数は、指定されたバッファから公開鍵をインポートし、それを Curve25519_Key 構造体に格納します。
int	**wc_curve25519_check_public**(const byte * pub, word32 pubSz, int endian) この関数は、公開鍵バッファが指定されたエンディアンに対して有効な Curve2519 キー値を保持していることを確認します。
int	**wc_curve25519_export_public**(curve25519_key * key, byte * out, word32 * outLen) この関数は指定されたキー構造から公開鍵をエクスポートし、結果をアウトバッファに格納します。ビッグエンディアンのみ。
int	**wc_curve25519_export_public_ex**(curve25519_key * key, byte * out, word32 * outLen, int endian) この関数は指定されたキー構造から公開鍵をエクスポートし、結果をアウトバッファに格納します。ビッグ・リトルエンディアンの両方をサポートします。
int	**wc_curve25519_export_key_raw**(curve25519_key * key, byte * priv, word32 * privSz, byte * pub, word32 * pubSz)Export Curve25519 キーペア。ビッグエンディアンのみ。
int	**wc_curve25519_export_key_raw_ex**(curve25519_key * key, byte * priv, word32 * privSz, byte * pub, word32 * pubSz, int endian)Export Curve25519 キーペア。ビッグ・リトルエンディアン。
int	**wc_curve25519_size**(curve25519_key * key) この関数は与えられたキー構造のキーサイズを返します。

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
)
```

この関数は、与えられたサイズ（Keysize）の指定された乱数発生器 RNG を使用して Curve25519 キーを生成し、それを指定された Curve25519_Key 構造体に格納します。キー構造が WC_CURVE25519_INIT（）を介して初期化された後に呼び出されるべきです。

Parameters:

- **RNG** ECC キーの生成に使用される RNG オブジェクトへのポインタ。
- **キーサイズ生成キーのサイズ。Curve25519 の 32 バイトでなければなりません。** *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
}
```

See: wc_curve25519_init

Return:

- 0 キーの生成に成功し、それを指定された Curve25519_Key 構造体に格納します。
- ECC_BAD_ARG_E 入力キーサイズが Curve25519 キー（32 バイト）のキーシェイズに対応していない場合は返されます。
- RNG_FAILURE_E RNG の内部ステータスが DRBG_OK でない場合、または RNG を使用して次のランダムブロックを生成する場合に返されます。
- BAD_FUNC_ARG 渡された入力パラメータのいずれかが NULL の場合に返されます。

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
)
```

この関数は、秘密の秘密鍵と受信した公開鍵を考えると、共有秘密鍵を計算します。生成された秘密鍵をバッファアウトに保存し、ounlent の秘密鍵の変数を割り当てます。ビッグエンディアンのみをサポートします。

Parameters:

- **Private_Key** Curve25519_Key 構造体の秘密鍵で初期化されました。
- **public_key 受信した公開鍵を含む Curve25519_Key 構造体へのポインタ。**
- **32 バイト計算された秘密鍵を格納するバッファへのポインタ。** *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
}
```

See:

- wc_curve25519_init
- wc_curve25519_make_key
- wc_curve25519_shared_secret_ex

Return:

- 0 共有秘密鍵を正常に計算したときに返されました。
- BAD_FUNC_ARG 渡された入力パラメータのいずれかが NULL の場合に返されます。
- ECC_BAD_ARG_E 公開鍵の最初のビットが設定されている場合は、実装の指紋を避けるために返されます。

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
)
```

この関数は、秘密の秘密鍵と受信した公開鍵を考えると、共有秘密鍵を計算します。生成された秘密鍵をバッファアウトに保存し、ounlent の秘密鍵の変数を割り当てます。ビッグ・リトルエンディアンの両方をサポートします。

Parameters:

- **Private_Key** Curve25519_Key 構造体の秘密鍵で初期化されました。
- **public_key 受信した公開鍵を含む Curve25519_Key 構造体へのポインタ。**
- **32 バイト計算された秘密鍵を格納するバッファへのポインタ。**
- **pinout]** 出力バッファに書き込まれた長さを記憶するポインタの概要。*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
}
```

See:

- wc_curve25519_init
- wc_curve25519_make_key
- wc_curve25519_shared_secret

Return:

- 0 共有秘密鍵を正常に計算したときに返されました。
- BAD_FUNC_ARG 渡された入力パラメータのいずれかが NULL の場合に返されます。
- ECC_BAD_ARG_E 公開鍵の最初のビットが設定されている場合は、実装の指紋を避けるために返されます。

C.14.2.4 function wc_curve25519_init

```
int wc_curve25519_init(
    curve25519_key * key
)
```

この関数は Curve25519 キーを初期化します。構造のキーを生成する前に呼び出されるべきです。

See: wc_curve25519_make_key

Return:

- 0 Curve25519_Key 構造体の初期化に成功しました。
- BAD_FUNC_ARG キーが NULL のときに返されます。*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
)
```

この関数は Curve25519 オブジェクトを解放します。*Example*

See:

- wc_curve25519_init
- wc_curve25519_make_key

```
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
)
```

この関数は Curve25519 秘密鍵のみをインポートします。(ビッグエンディアン)。

Parameters:

- **インポートする秘密鍵を含むバッファへのポイント。**
- **インポートする秘密鍵の Privsz 長。** *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
}
```

See:

- wc_curve25519_import_private_ex
- wc_curve25519_size

Return:

- 0 秘密鍵のインポートに成功しました。
- BAD_FUNC_ARG キーまたは PRIV が NULL の場合は返します。
- ECC_BAD_ARG_E PRIVSZ が curve25519_KEY_SIZE と等しくない場合は返します。

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 秘密鍵のインポートのみ。(大きなエンディアン)。

Parameters:

- **インポートする秘密鍵を含むバッファへのポイント。**
- **インポートする秘密鍵の Privsz 長。**
- **インポートされたキーを保存する構造へのキーポインタ。** *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
}
```

See:

- wc_curve25519_import_private

- wc_curve25519_size

Return:

- 0 秘密鍵のインポートに成功しました。
- BAD_FUNC_ARG キーまたは PRIV が NULL の場合は返します。
- ECC_BAD_ARG_E PRIVSZ が curve25519_KEY_SIZE と等しくない場合は返します。

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
)
```

この関数は、パブリック秘密鍵ペアを Curve25519_Key 構造体にインポートします。ビッグエンディアンのみ。

Parameters:

- **インポートする秘密鍵を含むバッファへのポイント。**
- **インポートする秘密鍵の Privsz 長。**
- **パブリックキーをインポートするバッファへの Pub。**
- **インポートする公開鍵の Pubsz 長さ。** *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
}
```

See:

- wc_curve25519_init
- wc_curve25519_make_key
- wc_curve25519_import_public
- wc_curve25519_export_private_raw

Return:

- 0 Curve25519_Key 構造体へのインポートに返されます
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL の場合に返します。
- ECC_BAD_ARG_E 入力キーのキーサイズが Public キーサイズまたは秘密鍵サイズと一致しない場合に返されます。

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
)
```

この関数は、パブリック秘密鍵ペアを Curve25519_Key 構造体にインポートします。ビッグ・リトルエンディアンの両方をサポートします。

Parameters:

- **インポートする秘密鍵を含むバッファへのポイント。**
- **インポートする秘密鍵の Privsz 長。**
- **パブリックキーをインポートするバッファへの Pub。**
- **インポートする公開鍵の Pubsz 長さ。**
- **インポートされたキーを保存する構造へのキーポインタ。** *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
}
```

See:

- wc_curve25519_init
- wc_curve25519_make_key
- wc_curve25519_import_public
- wc_curve25519_export_private_raw
- wc_curve25519_import_private_raw

Return:

- 0 Curve25519_Key 構造体へのインポートに返されます
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL の場合に返します。
- ECC_BAD_ARG_E 戻された IF または入力キーのキーサイズがパブリックキーサイズまたは秘密鍵サイズと一致しない場合

C.14.2.10 function wc_curve25519_export_private_raw

```
int wc_curve25519_export_private_raw(
    curve25519_key * key,
    byte * out,
    word32 * outLen
)
```

この関数は Curve25519_Key 構造体から秘密鍵をエクスポートし、それを指定されたアウトバッファに格納します。また、エクスポートされたキーのサイズになるように概要を設定します。ビッグエンディアンのみ。

Parameters:

- **キーをエクスポートする構造へのキーポインタ。**
- **エクスポートされたキーを保存するバッファへのポインタ。** *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
}
```

See:

- wc_curve25519_init
- wc_curve25519_make_key
- wc_curve25519_import_private_raw
- wc_curve25519_export_private_raw_ex

Return:

- 0 Curve25519_Key 構造体から秘密鍵を正常にエクスポートしました。
- BAD_FUNC_ARG 入力パラメータが NULL の場合に返されます。
- ECC_BAD_ARG_E WC_CURVE25519_SIZE () がキーと等しくない場合に返されます。

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
)
```

この関数は Curve25519_Key 構造体から秘密鍵をエクスポートし、それを指定されたアウトバッファに格納します。また、エクスポートされたキーのサイズになるように概要を設定します。それがビッグ・リトルエンディアンかを指定できます。

Parameters:

- **キーをエクスポートする構造へのキーポインタ。**
- **エクスポートされたキーを保存するバッファへのポインタ。**
- **IN に照会は、バイト数のサイズです。ON** OUT では、出力バッファに書き込まれたバイトを保存します。*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
}
```

See:

- wc_curve25519_init
- wc_curve25519_make_key
- wc_curve25519_import_private_raw
- wc_curve25519_export_private_raw
- wc_curve25519_size

Return:

- 0 Curve25519_Key 構造体から秘密鍵を正常にエクスポートしました。
- BAD_FUNC_ARG 入力パラメータが NULL の場合に返されます。
- ECC_BAD_ARG_E WC_CURVE25519_SIZE () がキーと等しくない場合に返されます。

C.14.2.12 function wc_curve25519_import_public

```
int wc_curve25519_import_public(
    const byte * in,
    word32 inLen,
    curve25519_key * key
)
```

この関数は、指定されたバッファから公開鍵をインポートし、それを Curve25519_Key 構造体に格納します。

Parameters:

- **インポートする公開鍵を含むバッファへのポインタ。**
- **インポートする公開鍵のインレル長。** *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
}
```

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 公開鍵を Curve25519_Key 構造体に正常にインポートしました。
- ECC_BAD_ARG_E InLen パラメータがキー構造のキーサイズと一致しない場合に返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL の場合に返されます。

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
)
```

この関数は、指定されたバッファから公開鍵をインポートし、それを Curve25519_Key 構造体に格納します。

Parameters:

- **インポートする公開鍵を含むバッファへのポインタ。**
- **インポートする公開鍵のインレル長。**
- **キーを保存するカーブ 25519 キー構造へのキーポインタ。** *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
}
```

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 公開鍵を Curve25519_Key 構造体に正常にインポートしました。
- ECC_BAD_ARG_E InLen パラメータがキー構造のキーサイズと一致しない場合に返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL の場合に返されます。

C.14.2.14 function wc_curve25519_check_public

```
int wc_curve25519_check_public(
    const byte * pub,
    word32 pubSz,
    int endian
)
```

この関数は、公開鍵バッファが指定されたエンディアンに対して有効な Curve2519 キー値を保持していることを確認します。

Parameters:

- **チェックするための公開鍵を含むバッファへの Pub ポインタ。**
- **チェックするための公開鍵の長さを掲載します。** *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
}
```

See:

- wc_curve25519_init
- wc_curve25519_import_public
- wc_curve25519_import_public_ex
- wc_curve25519_size

Return:

- 0 公開鍵の値が有効なときに返されます。
- ECC_BAD_ARG_E 公開鍵の値が無効な場合は返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL の場合に返されます。

C.14.2.15 function wc_curve25519_export_public

```
int wc_curve25519_export_public(
    curve25519_key * key,
    byte * out,
    word32 * outLen
)
```

この関数は指定されたキー構造から公開鍵をエクスポートし、結果をアウトバッファに格納します。ビッグエンディアンのみ。

Parameters:

- **キーをエクスポートする Curve25519_Key 構造体へのキーポインタ。**
- **公開鍵を保存するバッファへのポインタ。** *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
}
```

See:

- wc_curve25519_init

- wc_curve25519_export_private_raw
- wc_curve25519_import_public

Return:

- 0 Curve25519_Key 構造体から公開鍵を正常にエクスポートする上で返されます。
- ECC_BAD_ARG_E outlen が curve25519_pub_key_size より小さい場合に返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL の場合に返されます。

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
)
```

この関数は指定されたキー構造から公開鍵をエクスポートし、結果をアウトバッファに格納します。ビッグ・リトルエンディアンの両方をサポートします。

Parameters:

- **キーをエクスポートする Curve25519_Key 構造体へのキーポインタ。**
- **公開鍵を保存するバッファへのポインタ。**
- **IN に照会は、バイト数のサイズです。ON** OUT では、出力バッファに書き込まれたバイトを保存します。*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
}
```

See:

- wc_curve25519_init
- wc_curve25519_export_private_raw
- wc_curve25519_import_public

Return:

- 0 Curve25519_Key 構造体から公開鍵を正常にエクスポートする上で返されます。
- ECC_BAD_ARG_E outlen が curve25519_pub_key_size より小さい場合に返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL の場合に返されます。

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 キーペア。ビッグエンディアンのみ。

Parameters:

- **キーペアをエクスポートする CURUN448_KEY 構造体へのキーポインタ。**
- **秘密鍵を保存するバッファへの PRIV ポインタ。**
- **PRIVSZ** ON IN は、PRIV バッファのサイズをバイト単位で）です。ON OUT は、PRIV バッファに書き込まれたバイトを保存します。
- **パブリックキーを保存するバッファへの Pub。** *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
}
```

See:

- wc_curve25519_export_key_raw_ex
- wc_curve25519_export_private_raw

Return:

- 0 Curve25519_Key 構造体からキーペアのエクスポートに成功しました。
- BAD_FUNC_ARG 入力パラメータが NULL の場合に返されます。
- ECC_BAD_ARG_E PRIVSZ が CURUV25519_SEY_SIZE または PUBSZ よりも小さい場合は、PUBSZ が CURUG25519_PUB_KEY_SIZE よりも小さい場合に返されます。

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 キーペア。ビッグ・リトルエンディアン。

Parameters:

- **キーペアをエクスポートする CURUN448_KEY 構造体へのキーポインタ。**
- **秘密鍵を保存するバッファへの PRIV ポインタ。**
- **PRIVSZ** ON IN は、PRIV バッファのサイズをバイト単位で）です。ON OUT は、PRIV バッファに書き込まれたバイトを保存します。

- **パブリックキーを保存するバッファへの Pub。**
- **PUBSZ** ON IN は、パブバッファのサイズをバイト単位で）です。ON OUT では、PUB バッファに書き込まれたバイトを保存します。*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
}
```

See:

- wc_curve25519_export_key_raw
- wc_curve25519_export_private_raw_ex
- wc_curve25519_export_public_ex

Return:

- 0 Curve25519_Key 構造体からキーペアのエクスポートに成功しました。
- BAD_FUNC_ARG 入力パラメータが NULL の場合に返されます。
- ECC_BAD_ARG_E PRIVSZ が CURUV25519_SEY_SIZE または PUBSZ よりも小さい場合は、PUBSZ が CURUG25519_PUB_KEY_SIZE よりも小さい場合に返されます。

C.14.2.19 function wc_curve25519_size

```
int wc_curve25519_size(
    curve25519_key * key
)
```

この関数は与えられたキー構造のキーサイズを返します。

See:

- wc_curve25519_init
- wc_curve25519_make_key

Return:

- Success 有効な初期化された Curve25519_Key 構造体を考慮すると、キーのサイズを返します。
- 0 キーが NULL の場合は返されます *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-ja/curve448.h

C.15.1 Functions

	Name
int	**wc_curve448_make_key**(WC_RNG * rng, int keysize, curve448_key * key) この関数は、与えられたサイズ（Keysize）のサイズの指定された乱数発生器 RNG を使用して Curve448 キーを生成し、それを指定された Curve448_Key 構造体に格納します。キー構造が WC_CURVE448_INIT () を介して初期化された後に呼び出されるべきです。
int	**wc_curve448_shared_secret**(curve448_key * private_key, curve448_key * public_key, byte * out, word32 * outlen) この関数は、秘密の秘密鍵と受信した公開鍵を考えると、共有秘密鍵を計算します。生成された秘密鍵をバッファアウトに保存し、ounlent の秘密鍵の変数を割り当てます。ビッグエンディアンのみをサポートします。
int	**wc_curve448_shared_secret_ex**(curve448_key * private_key, curve448_key * public_key, byte * out, word32 * outlen, int endian) この関数は、秘密の秘密鍵と受信した公開鍵を考えると、共有秘密鍵を計算します。生成された秘密鍵をバッファアウトに保存し、ounlent の秘密鍵の変数を割り当てます。ビッグ・リトルエンディアンの両方をサポートします。
int	**wc_curve448_init**(curve448_key * key) この関数は Curve448 キーを初期化します。構造のキーを生成する前に呼び出されるべきです。
void	**wc_curve448_free**(curve448_key * key) この関数は Curve448 オブジェクトを解放します。*Example*
int	**wc_curve448_import_private**(const byte * priv, word32 privSz, curve448_key * key) この関数は Curve448 秘密鍵のみをインポートします。（ビッグエンディアン）。
int	**wc_curve448_import_private_ex**(const byte * priv, word32 privSz, curve448_key * key, int endian)CURVE448 秘密鍵のインポートのみ。（ビッグエンディアン）。
int	**wc_curve448_import_private_raw**(const byte * priv, word32 privSz, const byte * pub, word32 pubSz, curve448_key * key) この関数は、public-秘密鍵のペアを Curve448_Key 構造体にインポートします。ビッグエンディアンのみ。

	Name
int	**wc_curve448_import_private_raw_ex**(const byte * priv, word32 privSz, const byte * pub, word32 pubSz, curve448_key * key, int endian) この関数は、public-秘密鍵のペアを Curve448_Key 構造体にインポートします。ビッグ・リトルエンディアンの両方をサポートします。
int	**wc_curve448_export_private_raw**(curve448_key * key, byte * out, word32 * outLen) この関数は Curve448_Key 構造体から秘密鍵をエクスポートし、それを指定されたバッファに格納します。また、エクスポートされたキーのサイズになるように概要を設定します。ビッグエンディアンのみ。
int	**wc_curve448_export_private_raw_ex**(curve448_key * key, byte * out, word32 * outLen, int endian) この関数は Curve448_Key 構造体から秘密鍵をエクスポートし、それを指定されたバッファに格納します。また、エクスポートされたキーのサイズになるように概要を設定します。それが大きいかリトルエンディアンかを指定できます。
int	**wc_curve448_import_public**(const byte * in, word32 inLen, curve448_key * key) この関数は、指定されたバッファから公開鍵をインポートし、それを Curve448_Key 構造体に格納します。
int	**wc_curve448_import_public_ex**(const byte * in, word32 inLen, curve448_key * key, int endian) この関数は、指定されたバッファから公開鍵をインポートし、それを Curve448_Key 構造体に格納します。
int	**wc_curve448_check_public**(const byte * pub, word32 pubSz, int endian) この関数は、公開鍵バッファがエンディアン順序付けを与えられた有効な Curve448 キー値を保持することを確認します。
int	**wc_curve448_export_public**(curve448_key * key, byte * out, word32 * outLen) この関数は指定されたキー構造から公開鍵をエクスポートし、結果をアウトバッファに格納します。ビッグエンディアンのみ。
int	**wc_curve448_export_public_ex**(curve448_key * key, byte * out, word32 * outLen, int endian) この関数は指定されたキー構造から公開鍵をエクスポートし、結果をアウトバッファに格納します。ビッグ・リトルエンディアンの両方をサポートします。
int	**wc_curve448_export_key_raw**(curve448_key * key, byte * priv, word32 * privSz, byte * pub, word32 * pubSz) この関数は指定されたキー構造からキーペアをエクスポートし、結果をアウトバッファに格納します。ビッグエンディアンのみ。

	Name
int	**wc_curve448_export_key_raw_ex**(curve448_key * key, byte * priv, word32 * privSz, byte * pub, word32 * pubSz, int endian)Curve448 キーペアをエクスポートします。ビッグ、またはリトルエンディアン。
int	**wc_curve448_size**(curve448_key * key) この関数は与えられたキー構造のキーサイズを返します。

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
)
```

この関数は、与えられたサイズ（Keysize）のサイズの指定された乱数発生器 RNG を使用して Curve448 キーを生成し、それを指定された Curve448_Key 構造体に格納します。キー構造が WC_CURVE448_INIT（）を介して初期化された後に呼び出されるべきです。

Parameters:

- **RNG** ECC キーの生成に使用される RNG オブジェクトへのポインタ。
- **キーサイズ生成キーのサイズ。Curve448 の場合は 56 バイトでなければなりません。** *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
}
```

See: wc_curve448_init

Return:

- 0 キーの生成に成功し、それを指定された Curve448_Key 構造体に格納します。
- ECC_BAD_ARG_E 入力キーサイズが Curve448 キー（56 バイト）のキーシェイズに対応していない場合は返されます。
- RNG_FAILURE_E RNG の内部ステータスが DRBG_OK でない場合、または RNG を使用して次のランダムブロックを生成する場合に返されます。
- BAD_FUNC_ARG 渡された入力パラメータのいずれかが NULL の場合に返されます。

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
)
```

この関数は、秘密の秘密鍵と受信した公開鍵を考えると、共有秘密鍵を計算します。生成された秘密鍵をバッファアウトに保存し、ounlent の秘密鍵の変数を割り当てます。ビッグエンディアンのみをサポートします。

Parameters:

- **Private_Key** Curve448_Key 構造体へのポインタユーザーの秘密鍵で初期化されました。
- **public_key 受信した公開鍵を含む Curve448_Key 構造体へのポインタ。**
- **56 バイトの計算された秘密鍵を保存するバッファへのポインタ。** *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
}
```

See:

- wc_curve448_init
- wc_curve448_make_key
- wc_curve448_shared_secret_ex

Return:

- 0 共有秘密鍵を正常に計算する上で返却されました
- BAD_FUNC_ARG 渡された入力パラメーターのいずれかが NULL の場合に返されます

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
)
```

この関数は、秘密の秘密鍵と受信した公開鍵を考えると、共有秘密鍵を計算します。生成された秘密鍵をバッファアウトに保存し、ounlent の秘密鍵の変数を割り当てます。ビッグ・リトルエンディアンの両方をサポートします。

Parameters:

- **Private_Key** Curve448_Key 構造体へのポインタユーザーの秘密鍵で初期化されました。
- **public_key 受信した公開鍵を含む Curve448_Key 構造体へのポインタ。**
- **56 バイトの計算された秘密鍵を保存するバッファへのポインタ。**
- **出力バッファに書き込まれた長さを記憶するポインタの概要。** *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
}
```

See:

- wc_curve448_init
- wc_curve448_make_key
- wc_curve448_shared_secret

Return:

- 0 共有秘密鍵を正常に計算したときに返されました。
- BAD_FUNC_ARG 渡された入力パラメータのいずれかが NULL の場合に返されます。

C.15.2.4 function wc_curve448_init

```
int wc_curve448_init(
    curve448_key * key
)
```

この関数は Curve448 キーを初期化します。構造のキーを生成する前に呼び出されるべきです。

See: wc_curve448_make_key

Return:

- 0 Curve448_Key 構造体の初期化に成功しました。
- BAD_FUNC_ARG キーが NULL のときに返されます。*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
)
```

この関数は Curve448 オブジェクトを解放します。*Example*

See:

- wc_curve448_init
- wc_curve448_make_key

```
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
)
```

この関数は Curve448 秘密鍵のみをインポートします。(ビッグエンディアン)。

Parameters:

- **インポートする秘密鍵を含むバッファへのポイント。**
- **インポートする秘密鍵の Privsz 長。** *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
}
```

See:

- wc_curve448_import_private_ex
- wc_curve448_size

Return:

- 0 秘密鍵のインポートに成功しました。
- BAD_FUNC_ARG キーまたは PRIV が NULL の場合は返します。
- ECC_BAD_ARG_E PRIVSZ が CURUG448_KEY_SIZE と等しくない場合は返します。

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 秘密鍵のインポートのみ。(ビッグエンディアン)。

Parameters:

- **インポートする秘密鍵を含むバッファへのポイント。**
- **インポートする秘密鍵の Privsz 長。**
- **インポートされたキーを保存する構造へのキーポインタ。** *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
}
```

See:

- wc_curve448_import_private
- wc_curve448_size

Return:

- 0 秘密鍵のインポートに成功しました。
- BAD_FUNC_ARG キーまたは PRIV が NULL の場合は返します。
- ECC_BAD_ARG_E PRIVSZ が CURUG448_KEY_SIZE と等しくない場合は返します。

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
)
```

この関数は、public-秘密鍵のペアを Curve448_Key 構造体にインポートします。ビッグエンディアンのみ。

Parameters:

- **インポートする秘密鍵を含むバッファへのポイント。**
- **インポートする秘密鍵の Privsz 長。**
- **パブリックキーをインポートするバッファへの Pub。**
- **インポートする公開鍵の Pubsz 長さ。** *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
}
```

See:

- wc_curve448_init
- wc_curve448_make_key
- wc_curve448_import_public
- wc_curve448_export_private_raw

Return:

- 0 Curve448_Key 構造体へのインポート時に返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL の場合に返します。
- ECC_BAD_ARG_E 入力キーのキーサイズが Public キーサイズまたは秘密鍵サイズと一致しない場合に返されます。

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
)
```

この関数は、public-秘密鍵のペアを Curve448_Key 構造体にインポートします。ビッグ・リトルエンディアンの両方をサポートします。

Parameters:

- **インポートする秘密鍵を含むバッファへのポイント。**
- **インポートする秘密鍵の Privsz 長。**
- **パブリックキーをインポートするバッファへの Pub。**
- **インポートする公開鍵の Pubsz 長さ。**
- **インポートされたキーを保存する構造へのキーポインタ。** *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
}
```

See:

- wc_curve448_init
- wc_curve448_make_key
- wc_curve448_import_public
- wc_curve448_export_private_raw
- wc_curve448_import_private_raw

Return:

- 0 Curve448_Key 構造体へのインポート時に返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL の場合に返します。
- ECC_BAD_ARG_E 入力キーのキーサイズが Public キーサイズまたは秘密鍵サイズと一致しない場合に返されます。

C.15.2.10 function wc_curve448_export_private_raw

```
int wc_curve448_export_private_raw(
    curve448_key * key,
    byte * out,
    word32 * outLen
)
```

この関数は Curve448_Key 構造体から秘密鍵をエクスポートし、それを指定されたバッファに格納します。また、エクスポートされたキーのサイズになるように概要を設定します。ビッグエンディアンのみ。

Parameters:

- **キーをエクスポートする構造へのキーポインタ。**
- **エクスポートされたキーを保存するバッファへのポインタ。** *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
}
```

See:

- wc_curve448_init
- wc_curve448_make_key
- wc_curve448_import_private_raw
- wc_curve448_export_private_raw_ex

Return:

- 0 Curve448_Key 構造体から秘密鍵を正常にエクスポートする上で返されました。
- BAD_FUNC_ARG 入力パラメータが NULL の場合に返されます。
- ECC_BAD_ARG_E WC_CURVE448_SIZE () がキーと等しくない場合に返されます。

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
)
```

この関数は Curve448_Key 構造体から秘密鍵をエクスポートし、それを指定されたバッファに格納します。また、エクスポートされたキーのサイズになるように概要を設定します。それが大きいかリトルエンディアンかを指定できます。

Parameters:

- **キーをエクスポートする構造へのキーポインタ。**
- **エクスポートされたキーを保存するバッファへのポインタ。**
- **IN に照会は、バイト数のサイズです。ON** OUT では、出力バッファに書き込まれたバイトを保存します。*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
}
```

See:

- wc_curve448_init
- wc_curve448_make_key
- wc_curve448_import_private_raw
- wc_curve448_export_private_raw
- wc_curve448_size

Return:

- 0 Curve448_Key 構造体から秘密鍵を正常にエクスポートする上で返されました。
- BAD_FUNC_ARG 入力パラメータが NULL の場合に返されます。
- ECC_BAD_ARG_E WC_CURVE448_SIZE () がキーと等しくない場合に返されます。

C.15.2.12 function wc_curve448_import_public

```
int wc_curve448_import_public(
    const byte * in,
    word32 inLen,
    curve448_key * key
)
```

この関数は、指定されたバッファから公開鍵をインポートし、それを Curve448_Key 構造体に格納します。

Parameters:

- **インポートする公開鍵を含むバッファへのポインタ。**
- **インポートする公開鍵のインレル長。** *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
}
```

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 公開鍵を Curve448_Key 構造体に正常にインポートしました。
- ECC_BAD_ARG_E InLen パラメータがキー構造のキーサイズと一致しない場合に返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL の場合に返されます。

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
)
```

この関数は、指定されたバッファから公開鍵をインポートし、それを Curve448_Key 構造体に格納します。

Parameters:

- **インポートする公開鍵を含むバッファへのポインタ。**
- **インポートする公開鍵のインレル長。**
- **キーを保存する Curve448_Key 構造体へのキーポインタ。** *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
}
```

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 公開鍵を Curve448_Key 構造体に正常にインポートしました。
- ECC_BAD_ARG_E InLen パラメータがキー構造のキーサイズと一致しない場合に返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL の場合に返されます。

C.15.2.14 function wc_curve448_check_public

```
int wc_curve448_check_public(
    const byte * pub,
```

```
    word32 pubSz,
    int endian
)
```

この関数は、公開鍵バッファがエンディアン順序付けを与えられた有効な Curve448 キー値を保持することを確認します。

Parameters:

- **チェックするための公開鍵を含むバッファへの Pub ポインタ。**
- **チェックするための公開鍵の長さを掲載します。** *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
}
```

See:

- wc_curve448_init
- wc_curve448_import_public
- wc_curve448_import_public_ex
- wc_curve448_size

Return:

- 0 公開鍵の値が有効なときに返されます。
- ECC_BAD_ARG_E 公開鍵の値が無効な場合は返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL の場合に返されます。

C.15.2.15 function wc_curve448_export_public

```
int wc_curve448_export_public(
    curve448_key * key,
    byte * out,
    word32 * outLen
)
```

この関数は指定されたキー構造から公開鍵をエクスポートし、結果をアウトバッファに格納します。ビッグエンディアンのみ。

Parameters:

- **キーをエクスポートする Curve448_Key 構造体へのキーポインタ。**
- **公開鍵を保存するバッファへのポインタ。** *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
}
```

See:

- wc_curve448_init
- wc_curve448_export_private_raw
- wc_curve448_import_public

Return:

- 0 Curve448_Key 構造体から公開鍵のエクスポートに成功しました。
- ECC_BAD_ARG_E outlen が curve448_pub_key_size より小さい場合に返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL の場合に返されます。

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
)
```

この関数は指定されたキー構造から公開鍵をエクスポートし、結果をアウトバッファに格納します。ビッグ・リトルエンディアンの両方をサポートします。

Parameters:

- **キーをエクスポートする Curve448_Key 構造体へのキーポインタ。**
- **公開鍵を保存するバッファへのポインタ。**
- **IN に照会は、バイト数のサイズです。ON** OUT では、出力バッファに書き込まれたバイトを保存します。*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
}
```

See:

- wc_curve448_init
- wc_curve448_export_private_raw
- wc_curve448_import_public

Return:

- 0 Curve448_Key 構造体から公開鍵のエクスポートに成功しました。
- ECC_BAD_ARG_E outlen が curve448_pub_key_size より小さい場合に返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL の場合に返されます。

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
)
```

この関数は指定されたキー構造からキーペアをエクスポートし、結果をアウトバッファに格納します。ビッグエンディアンのみ。

Parameters:

- **キーペアをエクスポートする CURUN448_KEY 構造体へのキーポインタ。**
- **秘密鍵を保存するバッファへの PRIV ポインタ。**
- **PRIVSZ** ON IN は、PRIV バッファのサイズをバイト単位で）です。ON OUT は、PRIV バッファに書き込まれたバイトを保存します。
- **パブリックキーを保存するバッファへの Pub。** *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
}
```

See:

- wc_curve448_export_key_raw_ex
- wc_curve448_export_private_raw

Return:

- 0 Curve448_Key 構造体からキーペアのエクスポートに成功しました。
- BAD_FUNC_ARG 入力パラメータが NULL の場合に返されます。
- ECC_BAD_ARG_E PRIVSZ が CURUV448_KEY_SIZE または PUBSZ よりも小さい場合は､Curge448_PUB_KEY_SIZE よりも小さい場合に返されます。

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
)
```

Curve448 キーペアをエクスポートします。ビッグ、またはリトルエンディアン。

Parameters:

- **キーペアをエクスポートする CURUN448_KEY 構造体へのキーポインタ。**
- **秘密鍵を保存するバッファへの PRIV ポインタ。**
- **PRIVSZ** ON IN は、PRIV バッファのサイズをバイト単位で）です。ON OUT は、PRIV バッファに書き込まれたバイトを保存します。
- **パブリックキーを保存するバッファへの Pub。**
- **PUBSZ** ON IN は、パブバッファのサイズをバイト単位で）です。ON OUT では、PUB バッファに書き込まれたバイトを保存します。*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
}
```

See:

- wc_curve448_export_key_raw
- wc_curve448_export_private_raw_ex
- wc_curve448_export_public_ex

Return:

- 0 成功
- BAD_FUNC_ARG 入力パラメータが NULL の場合に返されます。
- ECC_BAD_ARG_E PRIVSZ が CURUV448_KEY_SIZE または PUBSZ よりも小さい場合は、Curge448_PUB_KEY_SIZE よりも小さい場合に返されます。

この関数は指定されたキー構造からキーペアをエクスポートし、結果をアウトバッファに格納します。ビッグ、またはリトルエンディアン。

C.15.2.19 function wc_curve448_size

```
int wc_curve448_size(
    curve448_key * key
)
```

この関数は与えられたキー構造のキーサイズを返します。

See:

- wc_curve448_init
- wc_curve448_make_key

Return:

- Success 有効な初期化された Curve448_Key 構造体を考慮すると、キーのサイズを返します。
- 0 キーが NULL の場合は返されます。*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-ja/des3.h

C.16.1 Functions

	Name
int	**wc_Des_SetKey**(Des * des, const byte * key, const byte * iv, int dir) この関数は、引数として与えられた DES 構造体のキーと初期化ベクトル（IV）を設定します。また、これらがまだ初期化されていない場合は、暗号化と復号化に必要なバッファーのスペースを初期化して割り当てます。注：IV が指定されていない場合（i.e.iv == null）初期化ベクトルは、デフォルトの IV 0 になります。
void	**wc_Des_SetIV**(Des * des, const byte * iv) この関数は、引数として与えられた DES 構造体の初期化ベクトル（IV）を設定します。NULL IV を渡したら、初期化ベクトルを 0 に設定します。
int	**wc_Des_CbcEncrypt**(Des * des, byte * out, const byte * in, word32 sz) この関数は入力メッセージを暗号化し、結果を出力バッファーに格納します。暗号ブロックチェーンチェーン（CBC）モードで DES 暗号化を使用します。
int	**wc_Des_CbcDecrypt**(Des * des, byte * out, const byte * in, word32 sz) この関数は入力暗号文を復号化し、結果の平文を出力バッファーに出力します。暗号ブロックチェーンチェーン（CBC）モードで DES 暗号化を使用します。
int	**wc_Des_EcbEncrypt**(Des * des, byte * out, const byte * in, word32 sz) この関数は入力メッセージを暗号化し、結果を出力バッファーに格納します。電子コードブック（ECB）モードで DES 暗号化を使用します。
int	**wc_Des3_EcbEncrypt**(Des3 * des, byte * out, const byte * in, word32 sz) この関数は入力メッセージを暗号化し、結果を出力バッファーに格納します。電子コードブック（ECB）モードで DES3 暗号化を使用します。警告：ほぼすべてのユースケースで ECB モードは安全性が低いと考えられています。可能な限り ECB API を直接使用しないでください。

	Name
int	**wc_Des3_SetKey**(Des3 * des, const byte * key, const byte * iv, int dir) この関数は、引数として与えられた DES3 構造のキーと初期化ベクトル（IV）を設定します。また、これらがまだ初期化されていない場合は、暗号化と復号化に必要なバッファーのスペースを初期化して割り当てます。注：IV が指定されていない場合（i.e.iv == null）初期化ベクトルは、デフォルトの IV 0 になります。
int	**wc_Des3_SetIV**(Des3 * des, const byte * iv) この関数は、引数として与えられた DES3 構造の初期化ベクトル（IV）を設定します。NULL IV を渡したら、初期化ベクトルを 0 に設定します。
int	**wc_Des3_CbcEncrypt**(Des3 * des, byte * out, const byte * in, word32 sz) この関数は入力メッセージを暗号化し、結果を出力バッファーに格納します。暗号ブロックチェーン（CBC）モードでトリプル DES（3DES）暗号化を使用します。
int	**wc_Des3_CbcDecrypt**(Des3 * des, byte * out, const byte * in, word32 sz) この関数は入力暗号文を復号化し、結果の平文を出力バッファーに出力します。暗号ブロックチェーン（CBC）モードでトリプル DES（3DES）暗号化を使用します。

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
)
```

この関数は、引数として与えられた DES 構造体のキーと初期化ベクトル（IV）を設定します。また、これらがまだ初期化されていない場合は、暗号化と復号化に必要なバッファーのスペースを初期化して割り当てます。注：IV が指定されていない場合（i.e.iv == null）初期化ベクトルは、デフォルトの IV 0 になります。

Parameters:

- **des** 初期化する DES 構造へのポインタ
- **key** DES 構造を初期化するための 8 バイトのキーを含むバッファへのポインタ
- **iv** DES 構造を初期化するための 8 バイト IV を含むバッファへのポインタ。これが提供されていない場合、IV はデフォルトで 0 になります *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
}
```

See:

- wc_Des_SetIV
- wc_Des3_SetKey

Return: 0 DES 構造体のキーと初期化ベクトルを正常に設定する

3

C.16.2.2 function wc_Des_SetIV

```
void wc_Des_SetIV(
    Des * des,
    const byte * iv
)
```

この関数は、引数として与えられた DES 構造体の初期化ベクトル（IV）を設定します。NULL IV を渡したら、初期化ベクトルを 0 に設定します。

Parameters:

- **des** IV を設定するための DES 構造へのポインタ *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);
}
```

See: wc_Des_SetKey

Return: none いいえ返します。

3

C.16.2.3 function wc_Des_CbcEncrypt

```
int wc_Des_CbcEncrypt(
    Des * des,
    byte * out,
    const byte * in,
    word32 sz
)
```

この関数は入力メッセージを暗号化し、結果を出力バッファーに格納します。暗号ブロックチェーンチェーン（CBC）モードで DES 暗号化を使用します。

Parameters:

- **des** 暗号化に使用する DES 構造へのポインタ
- **out** 暗号化された暗号文を保存するバッファへのポインタ
- **in** 暗号化するメッセージを含む入力バッファへのポインタ *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
}
```

See:

- wc_Des_SetKey
- wc_Des_CbcDecrypt

Return: 0 与えられた入力メッセージの暗号化に成功したときに返されます

3

C.16.2.4 function wc_Des_CbcDecrypt

```
int wc_Des_CbcDecrypt(
    Des * des,
    byte * out,
    const byte * in,
    word32 sz
)
```

この関数は入力暗号文を復号化し、結果の平文を出力バッファーに出力します。暗号ブロックチェーンチェーン（CBC）モードで DES 暗号化を使用します。

Parameters:

- **des** 復号化に使用する DES 構造へのポインタ
- **out** 復号化された平文を保存するバッファへのポインタ
- **in** 暗号化された暗号文を含む入力バッファへのポインタ *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
}
```

See:

- wc_Des_SetKey
- wc_Des_CbcEncrypt

Return: 0 与えられた暗号文を正常に復号化したときに返されました

3

C.16.2.5 function wc_Des_EcbEncrypt

```
int wc_Des_EcbEncrypt(
    Des * des,
    byte * out,
    const byte * in,
    word32 sz
)
```

この関数は入力メッセージを暗号化し、結果を出力バッファーに格納します。電子コードブック（ECB）モードで DES 暗号化を使用します。

Parameters:

- **des** 暗号化に使用する DES 構造へのポインタ
- **out** 暗号化されたメッセージを保存するバッファへのポインタ
- **in** 暗号化する平文を含む入力バッファへのポインタ *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
}
```

See: wc_Des_SetKe

Return: 0: 与えられた平文を正常に暗号化すると返されます。

3

C.16.2.6 function wc_Des3_EcbEncrypt

```
int wc_Des3_EcbEncrypt(
    Des3 * des,
    byte * out,
    const byte * in,
    word32 sz
)
```

この関数は入力メッセージを暗号化し、結果を出力バッファーに格納します。電子コードブック（ECB）モードで DES3 暗号化を使用します。警告：ほぼすべてのユースケースで ECB モードは安全性が低いと考えられています。可能な限り ECB API を直接使用しないでください。

Parameters:

- **des3** 暗号化に使用する DES3 構造へのポインタ
- **out** 暗号化されたメッセージを保存するバッファへのポインタ
- **in** 暗号化する平文を含む入力バッファへのポインタ *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
}
```

See: wc_Des3_SetKey

Return: 0 与えられた平文を正常に暗号化すると返されます

3

C.16.2.7 function wc_Des3_SetKey

```
int wc_Des3_SetKey(
    Des3 * des,
    const byte * key,
    const byte * iv,
    int dir
)
```

この関数は、引数として与えられた DES3 構造のキーと初期化ベクトル（IV）を設定します。また、これらがまだ初期化されていない場合は、暗号化と復号化に必要なバッファーのスペースを初期化して割り当てます。注：IV が指定されていない場合（i.e.iv == null）初期化ベクトルは、デフォルトの IV 0 になります。

Parameters:

- **des3** 初期化する DES3 構造へのポインタ
- **key** DES3 構造を初期化する 24 バイトのキーを含むバッファへのポインタ
- **iv** DES3 構造を初期化するための 8 バイト IV を含むバッファへのポインタ。これが提供されていない場合、IV はデフォルトで 0 になります *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
}
```

See:

- wc_Des3_SetIV
- wc_Des3_CbcEncrypt
- wc_Des3_CbcDecrypt

Return: 0 DES 構造体のキーと初期化ベクトルを正常に設定する

3

C.16.2.8 function wc_Des3_SetIV

```
int wc_Des3_SetIV(
    Des3 * des,
    const byte * iv
)
```

この関数は、引数として与えられた DES3 構造の初期化ベクトル（IV）を設定します。NULL IV を渡したら、初期化ベクトルを 0 に設定します。

Parameters:

- **des** IV を設定するための DES3 構造へのポインタ *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);
}
```

See: wc_Des3_SetKey

Return: none いいえ返します。

3

C.16.2.9 function wc_Des3_CbcEncrypt

```
int wc_Des3_CbcEncrypt(
    Des3 * des,
    byte * out,
    const byte * in,
    word32 sz
)
```

この関数は入力メッセージを暗号化し、結果を出力バッファーに格納します。暗号ブロックチェーン（CBC）モードでトリプル DES（3DES）暗号化を使用します。

Parameters:

- **des** 暗号化に使用する DES3 構造へのポインタ
- **out** 暗号化された暗号文を保存するバッファへのポインタ
- **in** 暗号化するメッセージを含む入力バッファへのポインタ *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
}
```

See:

- wc_Des3_SetKey
- wc_Des3_CbcDecrypt

Return: 0 与えられた入力メッセージの暗号化に成功したときに返されます

3

C.16.2.10 function wc_Des3_CbcDecrypt

```
int wc_Des3_CbcDecrypt(
    Des3 * des,
    byte * out,
    const byte * in,
    word32 sz
)
```

この関数は入力暗号文を復号化し、結果の平文を出力バッファーに出力します。暗号ブロックチェーン（CBC）モードでトリプル DES（3DES）暗号化を使用します。

Parameters:

- **des** 復号化に使用する DES3 構造へのポインタ
- **out** 復号化された平文を保存するバッファへのポインタ
- **in** 暗号化された暗号文を含む入力バッファへのポインタ *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
}
```

See:

- wc_Des3_SetKey
- wc_Des3_CbcEncrypt

Return: 0 与えられた暗号文を正常に復号化したときに返されました

3

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-ja/dh.h

C.17.1 Functions

	Name
int	**wc_InitDhKey**(DhKey * key) この関数は、Diffie-Hellman Exchange プロトコルを使用して安全な秘密鍵を交渉するのに使用するための Diffie-Hellman キーを初期化します。
void	**wc_FreeDhKey**(DhKey * key) この関数は、Diffie-Hellman Exchange プロトコルを使用して安全な秘密鍵をネゴシエートするために使用された後に Diffie-Hellman キーを解放します。
int	**wc_DhGenerateKeyPair**(DhKey * key, WC_RNG * rng, byte * priv, word32 * privSz, byte * pub, word32 * pubSz) この関数は diffie-hellman パブリックパラメータに基づいてパブリック/秘密鍵ペアを生成し、PRIVS の秘密鍵と Pub の公開鍵を格納します。初期化された Diffie-Hellman キーと初期化された RNG 構造を取ります。
int	**wc_DhAgree**(DhKey * key, byte * agree, word32 * agreeSz, const byte * priv, word32 privSz, const byte * otherPub, word32 pubSz) この関数は、ローカル秘密鍵と受信した公開鍵に基づいて合意された秘密鍵を生成します。交換の両側で完了した場合、この関数は対称通信のための秘密鍵の合意を生成します。共有秘密鍵の生成に成功すると、書かれた秘密鍵のサイズは仲間に保存されます。
int	**wc_DhKeyDecode**(const byte * input, word32 * inOutIdx, DhKey * key, word32) この機能は、DER フォーマットのキーを含む与えられた入力バッファから Diffie-Hellman キーをデコードします。結果を DHKEY 構造体に格納します。
int	**wc_DhSetKey**(DhKey * key, const byte * p, word32 pSz, const byte * g, word32 gSz) この関数は、入力秘密鍵パラメータを使用して DHKEY 構造体のキーを設定します。WC_DHKEYDECODE とは異なり、この関数は入力キーが DER フォーマットでフォーマットされ、代わりに PARSED 入力パラメータ P（Prime）と G（Base）を受け入れる必要はありません。
int	**wc_DhParamsLoad**(const byte * input, word32 inSz, byte * p, word32 * pInOutSz, byte * g, word32 * gInOutSz) この関数は、与えられた入力バッファから Diffie-Hellman パラメータ P（Prime）と G（ベース）をフォーマットされています。
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)

	Name
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)

C.17.2 Functions Documentation

C.17.2.1 function wc_InitDhKey

```
int wc_InitDhKey(
    DhKey * key
)
```

この関数は、Diffie-Hellman Exchange プロトコルを使用して安全な秘密鍵を交渉するのに使用するための Diffie-Hellman キーを初期化します。

See:

- wc_FreeDhKey
- wc_DhGenerateKeyPair

Return: none いいえ返します。*Example*

```
DhKey key;
wc_InitDhKey(&key); // initialize DH key
```

C.17.2.2 function wc_FreeDhKey

```
void wc_FreeDhKey(
    DhKey * key
)
```

この関数は、Diffie-Hellman Exchange プロトコルを使用して安全な秘密鍵をネゴシエートするために使用された後に Diffie-Hellman キーを解放します。

See: wc_InitDhKey

Return: none いいえ返します。*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
)
```

この関数は diffie-hellman パブリックパラメータに基づいてパブリック/秘密鍵ペアを生成し、PRIVS の秘密鍵と Pub の公開鍵を格納します。初期化された Diffie-Hellman キーと初期化された RNG 構造を取ります。

Parameters:

- **key** キーペアを生成する DHKEY 構造体へのポインタ
- **rng** キーを生成するための初期化された乱数発生器（RNG）へのポインタ
- **priv** 秘密鍵を保存するバッファへのポインタ
- **privSz** PRIV に書かれた秘密鍵のサイズを保存します
- **pub** 公開鍵を保存するバッファへのポインタ *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);
```

See:

- wc_InitDhKey
- wc_DhSetKey
- wc_DhKeyDecode

Return:

- BAD_FUNC_ARG この関数への入力の 1 つを解析するエラーがある場合に返されます
- RNG_FAILURE_E RNG を使用して乱数を生成するエラーが発生した場合
- MP_INIT_E 公開鍵の生成中に数学ライブラリにエラーがある場合は返却される可能性があります
- MP_READ_E 公開鍵の生成中に数学ライブラリにエラーがある場合は返却される可能性があります
- MP_EXPTMOD_E 公開鍵の生成中に数学ライブラリにエラーがある場合は返却される可能性があります
- MP_TO_E 公開鍵の生成中に数学ライブラリにエラーがある場合は返却される可能性があります

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
)
```

この関数は、ローカル秘密鍵と受信した公開鍵に基づいて合意された秘密鍵を生成します。交換の両側で完了した場合、この関数は対称通信のための秘密鍵の合意を生成します。共有秘密鍵の生成に成功すると、書かれた秘密鍵のサイズは仲間に保存されます。

Parameters:

- **key** 共有キーを計算するために使用する DHKEY 構造体へのポインタ
- **agree** 秘密キーを保存するバッファへのポインタ
- **agreeSz** 成功した後に秘密鍵のサイズを保持します
- **priv** ローカル秘密鍵を含むバッファへのポインタ
- **privSz** 地元の秘密鍵のサイズ
- **otherPub** 受信した公開鍵を含むバッファへのポインタ *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
}
```

See: wc_DhGenerateKeyPair

Return:

- 0 合意された秘密鍵の生成に成功しました
- MP_INIT_E 共有秘密鍵の生成中にエラーが発生した場合に返却される可能性があります
- MP_READ_E 共有秘密鍵の生成中にエラーが発生した場合に返却される可能性があります
- MP_EXPTMOD_E 共有秘密鍵の生成中にエラーが発生した場合に返却される可能性があります
- MP_TO_E 共有秘密鍵の生成中にエラーが発生した場合に返却される可能性があります

C.17.2.5 function wc_DhKeyDecode

```
int wc_DhKeyDecode(
    const byte * input,
    word32 * inOutIdx,
    DhKey * key,
    word32
)
```

この機能は、DER フォーマットのキーを含む与えられた入力バッファから Diffie-Hellman キーをデコードします。結果を DHKEY 構造体に格納します。

Parameters:

- **input** der フォーマットされた diffie-hellman キーを含むバッファへのポインタ
- **inOutIdx** キーをデコードしている間に解析されたインデックスを保存する整数へのポインタ
- **key** 入力キーで初期化するための DHKEY 構造体へのポインタ *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
}
```

See: wc_DhSetKey

Return:

- 0 入力キーの復号に成功したときに返されます
- ASN_PARSE_E 入力のシーケンスを解析したエラーがある場合に返されます
- ASN_DH_KEY_E 解析された入力から秘密鍵パラメータを読み取るエラーがある場合

C.17.2.6 function wc_DhSetKey

```
int wc_DhSetKey(
    DhKey * key,
    const byte * p,
    word32 pSz,
    const byte * g,
    word32 gSz
)
```

この関数は、入力秘密鍵パラメータを使用して DHKEY 構造体のキーを設定します。WC_DHKEYDECODE とは異なり、この関数は入力キーが DER フォーマットでフォーマットされ、代わりに PARSED 入力パラメータ P（Prime）と G（Base）を受け入れる必要はありません。

Parameters:

- **key** キーを設定する DHKEY 構造体へのポインタ
- **p** キーで使用するためのプライムを含むバッファへのポインタ
- **pSz** 入力プライムの長さ
- **g** キーで使用するためのベースを含むバッファへのポインタ *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
}
```

See: wc_DhKeyDecode

Return:

- 0 鍵の設定に成功しました
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL に評価された場合に返されます。
- MP_INIT_E ストレージのキーパラメータの初期化中にエラーがある場合に返されます。
- ASN_DH_KEY_E DH キーパラメータ P および G でエラーの読み取りがある場合は返されます

C.17.2.7 function wc_DhParamsLoad

```
int wc_DhParamsLoad(
    const byte * input,
    word32 inSz,
    byte * p,
    word32 * pInOutSz,
    byte * g,
    word32 * gInOutSz
)
```

この関数は、与えられた入力バッファから Diffie-Hellman パラメータ P（Prime）と G（ベース）をフォーマットされています。

Parameters:

- **input** 解析する DER フォーマットされた Difie-Hellman 証明書を含むバッファへのポインタ
- **inSz** 入力バッファのサイズ
- **p** 解析されたプライムを保存するバッファへのポインタ
- **pInOutSz** P バッファ内の使用可能なサイズを含む Word32 オブジェクトへのポインタ。関数呼び出しを完了した後にバッファに書き込まれたバイト数で上書きされます。
- **g** 解析されたベースを保存するバッファへのポインタ *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
}
```

See:

- wc_DhSetKey
- wc_DhKeyDecode

Return:

- 0 DH パラメータの抽出に成功しました
- ASN_PARSE_E DER フォーマットの DH 証明書の解析中にエラーが発生した場合に返されます。
- BUFFER_E 解析されたパラメータを格納するために P または G に不適切なスペースがある場合

C.17.2.8 function wc_Dh_ffdhe2048_Get

```
const DhParams * wc_Dh_ffdhe2048_Get(
    void
)
```

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
)
```

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
)
```

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
)
```

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
)
```

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
)
```

C.17.2.14 function wc_DhCheckPrivKey

```
int wc_DhCheckPrivKey(
    DhKey * key,
    const byte * priv,
    word32 pubSz
)
```

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-ja/doxygen_groups.h

C.19 dox_comments/header_files-ja/doxygen_pages.h

C.20 dox_comments/header_files-ja/dsa.h

C.20.1 Functions

	Name
int	**wc_InitDsaKey**(DsaKey * key) この関数は、デジタル署名アルゴリズム（DSA）を介した認証に使用するために DSAKEY オブジェクトを初期化します。
void	**wc_FreeDsaKey**(DsaKey * key) この関数は、使用された後に dsakey オブジェクトを解放します。
int	**wc_DsaSign**(const byte * digest, byte * out, DsaKey * key, WC_RNG * rng) この機能は入力ダイジェストに署名し、結果を出力バッファーに格納します。
int	**wc_DsaVerify**(const byte * digest, const byte * sig, DsaKey * key, int * answer) この関数は、秘密鍵を考えると、ダイジェストの署名を検証します。回答パラメータでキーが正しく検証されているかどうか、正常な検証に対応する 1、および失敗した検証に対応する 0 が格納されます。
int	**wc_DsaPublicKeyDecode**(const byte * input, word32 * inOutIdx, DsaKey * key, word32 inSz) この機能は、DSA 公開鍵を含む DER フォーマットの証明書バッファを復号し、与えられた DSakey 構造体にキーを格納します。また、入力読み取りの長さに応じて INOUTIDX パラメータを設定します。
int	**wc_DsaPrivateKeyDecode**(const byte * input, word32 * inOutIdx, DsaKey * key, word32 inSz) この機能は、DSA 秘密鍵を含む DER フォーマットの証明書バッファをデコードし、指定された DSakey 構造体にキーを格納します。また、入力読み取りの長さに応じて INOUTIDX パラメータを設定します。
int	**wc_DsaKeyToDer**(DsaKey * key, byte * output, word32 inLen)DSAKEY キーを DER フォーマット、出力への書き込み（Inlen）、書き込まれたバイトを返します。
int	**wc_MakeDsaKey**(WC_RNG * rng, DsaKey * dsa)DSA キーを作成します。
int	**wc_MakeDsaParameters**(WC_RNG * rng, int modulus_size, DsaKey * dsa)FIPS 186-4 は、modulus_size 値の有効な値を定義します（1024,160）（2048,256）（3072,256）

C.20.2 Functions Documentation

C.20.2.1 function wc_InitDsaKey

```
int wc_InitDsaKey(
    DsaKey * key
)
```

この関数は、デジタル署名アルゴリズム（DSA）を介した認証に使用するために DSAKEY オブジェクトを初期化します。

See: wc_FreeDsaKey

Return:

- 0 成功に戻りました。
- BAD_FUNC_ARG NULL キーが渡された場合に返されます。*Example*

```
DsaKey key;
int ret;
ret = wc_InitDsaKey(&key); // initialize DSA key
```

C.20.2.2 function wc_FreeDsaKey

```
void wc_FreeDsaKey(
    DsaKey * key
)
```

この関数は、使用された後に dsakey オブジェクトを解放します。

See: wc_FreeDsaKey

Return: none いいえ返します。*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
)
```

この機能は入力ダイジェストに署名し、結果を出力バッファーに格納します。

Parameters:

- **digest** 署名するハッシュへのポインタ
- **out** 署名を保存するバッファへのポインタ
- **key** 署名を生成するための初期化された Dsakey 構造へのポインタ *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
}
```

See: wc_DsaVerify

Return:

- 0 入力ダイジェストに正常に署名したときに返されました
- MP_INIT_E DSA 署名の処理にエラーがある場合は返される可能性があります。
- MP_READ_E DSA 署名の処理にエラーがある場合は返される可能性があります。
- MP_CMP_E DSA 署名の処理にエラーがある場合は返される可能性があります。
- MP_INVMOD_E DSA 署名の処理にエラーがある場合は返される可能性があります。
- MP_EXPTMOD_E DSA 署名の処理にエラーがある場合は返される可能性があります。
- MP_MOD_E DSA 署名の処理にエラーがある場合は返される可能性があります。
- MP_MUL_E DSA 署名の処理にエラーがある場合は返される可能性があります。
- MP_ADD_E DSA 署名の処理にエラーがある場合は返される可能性があります。
- MP_MULMOD_E DSA 署名の処理にエラーがある場合は返される可能性があります。
- MP_TO_E DSA 署名の処理にエラーがある場合は返される可能性があります。
- MP_MEM DSA 署名の処理にエラーがある場合は返される可能性があります。

C.20.2.4 function wc_DsaVerify

```
int wc_DsaVerify(
    const byte * digest,
    const byte * sig,
    DsaKey * key,
    int * answer
)
```

この関数は、秘密鍵を考えると、ダイジェストの署名を検証します。回答パラメータでキーが正しく検証されているかどうか、正常な検証に対応する 1、および失敗した検証に対応する 0 が格納されます。

Parameters:

- **digest** 署名の主題を含むダイジェストへのポインタ
- **sig** 確認する署名を含むバッファへのポインタ
- **key** 署名を検証するための初期化された Dsakey 構造へのポインタ *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
}
```

See: wc_DsaSign

Return:

- 0 検証要求の処理に成功したときに返されます。注：これは、署名が検証されていることを意味するわけではなく、関数が成功したというだけです。

- MP_INIT_E DSA 署名の処理にエラーがある場合は返される可能性があります。
- MP_READ_E DSA 署名の処理にエラーがある場合は返される可能性があります。
- MP_CMP_E DSA 署名の処理にエラーがある場合は返される可能性があります。
- MP_INVMOD_E DSA 署名の処理にエラーがある場合は返される可能性があります。
- MP_EXPTMOD_E DSA 署名の処理にエラーがある場合は返される可能性があります。
- MP_MOD_E DSA 署名の処理にエラーがある場合は返される可能性があります。
- MP_MUL_E DSA 署名の処理にエラーがある場合は返される可能性があります。
- MP_ADD_E DSA 署名の処理にエラーがある場合は返される可能性があります。
- MP_MULMOD_E DSA 署名の処理にエラーがある場合は返される可能性があります。
- MP_TO_E DSA 署名の処理にエラーがある場合は返される可能性があります。
- MP_MEM DSA 署名の処理にエラーがある場合は返される可能性があります。

C.20.2.5 function wc_DsaPublicKeyDecode

```
int wc_DsaPublicKeyDecode(
    const byte * input,
    word32 * inOutIdx,
    DsaKey * key,
    word32 inSz
)
```

この機能は、DSA 公開鍵を含む DER フォーマットの証明書バッファを復号し、与えられた DSakey 構造体にキーを格納します。また、入力読み取りの長さに応じて INOUTIDX パラメータを設定します。

Parameters:

- **input** DER フォーマット DSA 公開鍵を含むバッファへのポインタ
- **inOutIdx** 証明書の最後のインデックスを保存する整数へのポインタ
- **key** 公開鍵を保存する Dsakey 構造へのポインタ *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
}
```

See:

- wc_InitDsaKey
- wc_DsaPrivateKeyDecode

Return:

- 0 dsakey オブジェクトの公開鍵を正常に設定する
- ASN_PARSE_E 証明書バッファを読みながらエンコーディングにエラーがある場合
- ASN_DH_KEY_E DSA パラメータの 1 つが誤ってフォーマットされている場合に返されます

C.20.2.6 function wc_DsaPrivateKeyDecode

```
int wc_DsaPrivateKeyDecode(
    const byte * input,
    word32 * inOutIdx,
    DsaKey * key,
```

```
    word32 inSz
)
```

この機能は、DSA 秘密鍵を含む DER フォーマットの証明書バッファをデコードし、指定された DSakey 構造体にキーを格納します。また、入力読み取りの長さに応じて INOUTIDX パラメータを設定します。

Parameters:

- **input** DER フォーマット DSA 秘密鍵を含むバッファへのポインタ
- **inOutIdx** 証明書の最後のインデックスを保存する整数へのポインタ
- **key** 秘密鍵を保存する DSakey 構造へのポインタ *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
}
```

See:

- wc_InitDsaKey
- wc_DsaPublicKeyDecode

Return:

- 0 dsakey オブジェクトの秘密鍵を正常に設定するに返されました
- ASN_PARSE_E 証明書バッファを読みながらエンコーディングにエラーがある場合
- ASN_DH_KEY_E DSA パラメータの 1 つが誤ってフォーマットされている場合に返されます

C.20.2.7 function wc_DsaKeyToDer

```
int wc_DsaKeyToDer(
    DsaKey * key,
    byte * output,
    word32 inLen
)
```

DSAKEY キーを DER フォーマット、出力への書き込み（Inlen）、書き込まれたバイトを返します。

Parameters:

- **key** 変換する dsakey 構造へのポインタ。
- **output** 変換キーの出力バッファへのポインタ。*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);
```

See:

- wc_InitDsaKey
- wc_FreeDsaKey
- wc_MakeDsaKey

Return:

- outLen 成功、書かれたバイト数
- BAD_FUNC_ARG キーまたは出力は NULL またはキー - > タイプが DSA_PRIVATE ではありません。
- MEMORY_E メモリの割り当て中にエラーが発生しました。

C.20.2.8 function wc_MakeDsaKey

```
int wc_MakeDsaKey(
    WC_RNG * rng,
    DsaKey * dsa
)
```

DSA キーを作成します。

Parameters:

- **rng** WC_RNG 構造体へのポインタ。*Example*

```
WC_RNG rng;
DsaKey dsa;
wc_InitRng(&rng);
wc_InitDsa(&dsa);
if(wc_MakeDsaKey(&rng, &dsa) != 0)
{
    // Error creating key
}
```

See:

- wc_InitDsaKey
- wc_FreeDsaKey
- wc_DsaSign

Return:

- MP_OKAY 成功
- BAD_FUNC_ARG RNG または DSA のどちらかが null です。
- MEMORY_E バッファにメモリを割り当てることができませんでした。
- MP_INIT_E MP_INT の初期化エラー

C.20.2.9 function wc_MakeDsaParameters

```
int wc_MakeDsaParameters(
    WC_RNG * rng,
    int modulus_size,
    DsaKey * dsa
)
```

FIPS 186-4 は、modulus_size 値の有効な値を定義します（1024,160）（2048,256）（3072,256）

Parameters:

- **rng** WolfCrypt RNG へのポインタ。
- **modulus_size** 1024,2048、または 3072 は有効な値です。*Example*

```
DsaKey key;
WC_RNG rng;
wc_InitDsaKey(&key);
wc_InitRng(&rng);
if(wc_MakeDsaParameters(&rng, 1024, &genKey) != 0)
{
    // Handle error
}
```

See:

- wc_MakeDsaKey
- wc_DsaKeyToDer
- wc_InitDsaKey

Return:

- 0 成功
- BAD_FUNC_ARG RNG または DSA は NULL または MODULUS_SIZE が無効です。
- MEMORY_E メモリを割り当てようとするエラーが発生しました。

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-ja/ecc.h

C.21.1 Functions

	Name
int	**wc_ecc_make_key**(WC_RNG * rng, int keysize, ecc_key * key) この関数は新しい ECC_KEY を生成し、それをキーに格納します。

	Name
int	**wc_ecc_make_key_ex**(WC_RNG * rng, int keysize, ecc_key * key, int curve_id) この関数は新しい ECC_KEY を生成し、それをキーに格納します。
int	**wc_ecc_check_key**(ecc_key * key)ECC キーの有効性を有効にします。
void	**wc_ecc_key_free**(ecc_key * key) この関数は、使用された後に ECC_KEY キーを解放します。 *Example*
int	**wc_ecc_shared_secret**(ecc_key * private_key, ecc_key * public_key, byte * out, word32 * outlen) この関数は、ローカル秘密鍵と受信した公開鍵を使用して新しい秘密鍵を生成します。この共有秘密鍵をバッファアウトに格納し、出力バッファに書き込まれたバイト数を保持するために outlenten を更新します。
int	**wc_ecc_shared_secret_ex**(ecc_key * private_key, ecc_point * point, byte * out, word32 * outlen) 秘密鍵とパブリックポイントの間に ECC 共有秘密を作成します。
int	**wc_ecc_sign_hash**(const byte * in, word32 inlen, byte * out, word32 * outlen, WC_RNG * rng, ecc_key * key) この関数は、信頼性を保証するために ECC_KEY オブジェクトを使用してメッセージダイジェストに署名します。
int	**wc_ecc_sign_hash_ex**(const byte * in, word32 inlen, WC_RNG * rng, ecc_key * key, mp_int * r, mp_int * s) メッセージダイジェストに署名します。
int	**wc_ecc_verify_hash**(const byte * sig, word32 siglen, const byte * hash, word32 hashlen, int * stat, ecc_key * key) この関数は、真正性を確保するためにハッシュの ECC シグネチャを検証します。答えを介して、有効な署名に対応する 1、無効な署名に対応する 0 で答えを返します。
int	**wc_ecc_verify_hash_ex**(mp_int * r, mp_int * s, const byte * hash, word32 hashlen, int * stat, ecc_key * key)ECC 署名を確認してください。結果は stat に書き込まれます。1 が有効で、0 が無効です。注：有効なテストに戻り値を使用しないでください。stat のみを使用してください。
int	**wc_ecc_init**(ecc_key * key) この関数は、メッセージ検証または鍵交渉で将来の使用のために ECC_KEY オブジェクトを初期化します。
int	**wc_ecc_init_ex**(ecc_key * key, void * heap, int devId) この関数は、メッセージ検証または鍵交渉で将来の使用のために ECC_KEY オブジェクトを初期化します。
ecc_key *	**wc_ecc_key_new**(void * heap) この関数はユーザー定義ヒープを使用し、キー構造のスペースを割り当てます。

	Name
int	**wc_ecc_free**(ecc_key * key) この関数は、使用後に ECC_KEY オブジェクトを解放します。
void	**wc_ecc_fp_free**(void) この関数は固定小数点キャッシュを解放します。これは ECC で使用でき、計算時間を高速化します。この機能を使用するには、FP_ECC（固定小数点 ECC）を定義する必要があります。
int	**wc_ecc_is_valid_idx**(int n)ECC IDX が有効かどうかを確認します。
ecc_point *	**wc_ecc_new_point**(void) 新しい ECC ポイントを割り当てます。
void	**wc_ecc_del_point**(ecc_point * p) メモリから ECC ポイントを解放します。
int	**wc_ecc_copy_point**(ecc_point * p, ecc_point * r) あるポイントの値を別のポイントにコピーします。
int	**wc_ecc_cmp_point**(ecc_point * a, ecc_point * b) ポイントの値を別のものと比較してください。
int	**wc_ecc_point_is_at_infinity**(ecc_point * p) ポイントが無限大にあるかどうかを確認します。返品 1 が無限大である場合は 0、そうでない場合は 0、<0 エラー時の 0
int	**wc_ecc_mulmod**(mp_int * k, ecc_point * G, ecc_point * R, mp_int * a, mp_int * modulus, int map)ECC 固定点乗算を実行します。
int	**wc_ecc_export_x963**(ecc_key * key, byte * out, word32 * outLen) この関数は ECC キーを ECC_KEY 構造体からエクスポートし、結果を OUT に格納します。キーは ANSI X9.63 フォーマットに保存されます。outlen の出力バッファに書き込まれたバイトを格納します。
int	**wc_ecc_export_x963_ex**(ecc_key * key, byte * out, word32 * outLen, int compressed) この関数は ECC キーを ECC_KEY 構造体からエクスポートし、結果を OUT に格納します。キーは ANSI X9.63 フォーマットに保存されます。outlen の出力バッファに書き込まれたバイトを格納します。この関数は、圧縮されたパラメータを介して証明書を圧縮する追加のオプションを使用する。このパラメータが true の場合、キーは ANSI X9.63 圧縮形式で保存されます。
int	**wc_ecc_import_x963**(const byte * in, word32 inLen, ecc_key * key) この関数は、ANSI X9.63 形式で保存されているキーを含むバッファからパブリック ECC キーをインポートします。この関数は、圧縮キーが hand_comp_key オプションを介してコンパイル時に有効になっている限り、圧縮キーと非圧縮キーの両方を処理します。

	Name
int	**wc_ecc_import_private_key**(const byte * priv, word32 privSz, const byte * pub, word32 pubSz, ecc_key * key) この関数は、生の秘密鍵を含むバッファと、ANSI X9.63 フォーマットされた公開鍵を含む 2 番目のバッファーからパブリック/プライベート ECC キーのペアをインポートします。この関数は、圧縮キーが hand_comp_key オプションを介してコンパイル時に有効になっている限り、圧縮キーと非圧縮キーの両方を処理します。
int	**wc_ecc_rs_to_sig**(const char * r, const char * s, byte * out, word32 * outlen) この関数は、ECC シグネチャの R 部分と S 部分を DER 符号化 ECDSA シグネチャに変換します。この機能は、outlen では、出力バッファに書き込まれた長さも記憶されています。
int	**wc_ecc_import_raw**(ecc_key * key, const char * qx, const char * qy, const char * d, const char * curveName) この関数は、ECC 署名の RAW 成分を持つ ECC_KEY 構造体を埋めます。
int	**wc_ecc_export_private_only**(ecc_key * key, byte * out, word32 * outLen) この関数は、ECC_KEY 構造体から秘密鍵のみをエクスポートします。秘密鍵をバッファアウトに格納し、outlen にこのバッファに書き込まれたバイトを設定します。
int	**wc_ecc_export_point_der**(const int curve_idx, ecc_point * point, byte * out, word32 * outLen)DER へのエクスポートポイント。
int	**wc_ecc_import_point_der**(byte * in, word32 inLen, const int curve_idx, ecc_point * point)Der フォーマットからのインポートポイント。
int	**wc_ecc_size**(ecc_key * key) この関数は、ecc_key 構造体のキーサイズをオクテットで返します。
int	**wc_ecc_sig_size_calc**(int sz) この関数は、次のようにして指定された ECC シグネチャの最悪の場合のサイズを返します。（KEYSZ * 2）+ SIG_HEADER_SZ + ECC_MAX_PAD_SZ。実際のシグネチャサイズは、WC_ECC_SIGN_HASH で計算できます。
int	**wc_ecc_sig_size**(ecc_key * key) この関数は、次のようにして指定された ECC シグネチャの最悪の場合のサイズを返します。（KEYSZ * 2）+ SIG_HEADER_SZ + ECC_MAX_PAD_SZ。実際のシグネチャサイズは、WC_ECC_SIGN_HASH で計算できます。
ecEncCtx *	**wc_ecc_ctx_new**(int flags, WC_RNG * rng) この機能は、ECC との安全なメッセージ交換を可能にするために、新しい ECC コンテキストオブジェクトのスペースを割り当て、初期化します。
void	**wc_ecc_ctx_free**(ecEncCtx *) この関数は、メッセージの暗号化と復号化に使用される ECENCCTX オブジェクトを解放します。

	Name
int	**wc_ecc_ctx_reset**(ecEncCtx * ctx, WC_RNG * rng) この関数は ECENCCTX 構造をリセットして、新しいコンテキストオブジェクトを解放し、新しいコンテキストオブジェクトを割り当てます。
int	**wc_ecc_ctx_set_algo**(ecEncCtx * ctx, byte encAlgo, byte kdfAlgo, byte macAlgo) この関数は、wc_ecc_ctx_new の後にオプションで呼び出されることができます。暗号化、KDF、および MAC アルゴリズムを ECENCENCCTX オブジェクトに設定します。
const byte *	**wc_ecc_ctx_get_own_salt**(ecEncCtx *) この関数は ECENCENCCTX オブジェクトのソルトを返します。この関数は、ECENCCTX の状態が ECSRV_INIT または ECCLI_INIT の場合にのみ呼び出す必要があります。
int	**wc_ecc_ctx_set_peer_salt**(ecEncCtx * ctx, const byte * salt) この関数は、ECENCENCCTX オブジェクトのピアソルトを設定します。
int	**wc_ecc_ctx_set_info**(ecEncCtx * ctx, const byte * info, int sz) この関数は、wc_ecc_ctx_set_peer_salt の前後にオプションで呼び出されることができます。ECENCCTX オブジェクトのオプションの情報を設定します。
int	**wc_ecc_encrypt**(ecc_key * privKey, ecc_key * pubKey, const byte * msg, word32 msgSz, byte * out, word32 * outSz, ecEncCtx * ctx) この関数は指定された入力メッセージを MSG から OUT に暗号化します。この関数はパラメータとしてオプションの CTX オブジェクトを取ります。提供されている場合、ECENCCTX の Encalgo、Kdfalgo、および Macalgo に基づいて暗号化が進みます。CTX が指定されていない場合、処理はデフォルトのアルゴリズム、ECAES_128_CBC、ECHKDF_SHA256、ECHMAC_SHA256 で完了します。この機能は、メッセージが CTX で指定された暗号化タイプに従って埋め込まれている必要があります。
int	**wc_ecc_encrypt_ex**(ecc_key * privKey, ecc_key * pubKey, const byte * msg, word32 msgSz, byte * out, word32 * outSz, ecEncCtx * ctx, int compressed) この関数は指定された入力メッセージを MSG から OUT に暗号化します。この関数はパラメータとしてオプションの CTX オブジェクトを取ります。提供されている場合、ECENCCTX の Encalgo、Kdfalgo、および Macalgo に基づいて暗号化が進みます。CTX が指定されていない場合、処理はデフォルトのアルゴリズム、ECAES_128_CBC、ECHKDF_SHA256、ECHMAC_SHA256 で完了します。この機能は、メッセージが CTX で指定された暗号化タイプに従って埋め込まれている必要があります。

	Name
int	**wc_ecc_decrypt**(ecc_key * privKey, ecc_key * pubKey, const byte * msg, word32 msgSz, byte * out, word32 * outSz, ecEncCtx * ctx) この関数は MSG から OUT への暗号文を復号化します。この関数はパラメータとしてオプションの CTX オブジェクトを取ります。提供されている場合、ECENCCTX の Encalgo、Kdfalgo、および Macalgo に基づいて暗号化が進みます。CTX が指定されていない場合、処理はデフォルトのアルゴリズム、ECAES_128_CBC、ECHKDF_SHA256、ECHMAC_SHA256 で完了します。この機能は、メッセージが CTX で指定された暗号化タイプに従って埋め込まれている必要があります。
int	**wc_ecc_set_nonblock**(ecc_key * key, ecc_nb_ctx_t * ctx) 非ブロック操作のための ECC サポートを有効にします。次のビルドオプションを使用した単精度（SP）数学でサポートされています.WolfSSL_SP_SP_SMALL WOLFSSL_SP_NO_MALLOC WC_ECC_NONBLOCK

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
)
```

この関数は新しい ECC_KEY を生成し、それをキーに格納します。

Parameters:

- **rng** キーを生成するための初期化された RNG オブジェクトへのポインタ
- **keysize** ECC_KEY の希望の長さ *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
```

See:

- wc_ecc_init
- wc_ecc_shared_secret

Return:

- 0 成功に戻りました。
- ECC_BAD_ARG_E RNG またはキーが NULL に評価された場合に返されます
- BAD_FUNC_ARG 指定されたキーサイズがサポートされているキーの正しい範囲にない場合に返されます。
- MEMORY_E ECC キーの計算中にメモリの割り当てエラーがある場合に返されます。

- MP_INIT_E ECC キーの計算中にエラーが発生した場合に返される可能性があります
- MP_READ_E ECC キーの計算中にエラーが発生した場合に返される可能性があります
- MP_CMP_E ECC キーの計算中にエラーが発生した場合に返される可能性があります
- MP_INVMOD_E ECC キーの計算中にエラーが発生した場合に返される可能性があります
- MP_EXPTMOD_E ECC キーの計算中にエラーが発生した場合に返される可能性があります
- MP_MOD_E ECC キーの計算中にエラーが発生した場合に返される可能性があります
- MP_MUL_E ECC キーの計算中にエラーが発生した場合に返される可能性があります
- MP_ADD_E ECC キーの計算中にエラーが発生した場合に返される可能性があります
- MP_MULMOD_E ECC キーの計算中にエラーが発生した場合に返される可能性があります
- MP_TO_E ECC キーの計算中にエラーが発生した場合に返される可能性があります
- MP_MEM ECC キーの計算中にエラーが発生した場合に返される可能性があります

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
)
```

この関数は新しい ECC_KEY を生成し、それをキーに格納します。

Parameters:

- **key** 作成したキーを保存するためのポインタ。
- **keysize** CavenID に基づいて設定されたバイト単位で作成するキーのサイズ
- **rng** 鍵作成に使用される RNG *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
}
```

See:

- wc_ecc_make_key
- wc_ecc_get_curve_size_from_id

Return:

- 0 成功に戻りました。
- ECC_BAD_ARG_E RNG またはキーが NULL に評価された場合に返されます
- BAD_FUNC_ARG 指定されたキーサイズがサポートされているキーの正しい範囲にない場合に返されます。
- MEMORY_E ECC キーの計算中にメモリの割り当てエラーがある場合に返されます。
- MP_INIT_E ECC キーの計算中にエラーが発生した場合に返される可能性があります
- MP_READ_E ECC キーの計算中にエラーが発生した場合に返される可能性があります
- MP_CMP_E ECC キーの計算中にエラーが発生した場合に返される可能性があります
- MP_INVMOD_E ECC キーの計算中にエラーが発生した場合に返される可能性があります
- MP_EXPTMOD_E ECC キーの計算中にエラーが発生した場合に返される可能性があります

- MP_MOD_E ECC キーの計算中にエラーが発生した場合に返される可能性があります
- MP_MUL_E ECC キーの計算中にエラーが発生した場合に返される可能性があります
- MP_ADD_E ECC キーの計算中にエラーが発生した場合に返される可能性があります
- MP_MULMOD_E ECC キーの計算中にエラーが発生した場合に返される可能性があります
- MP_TO_E ECC キーの計算中にエラーが発生した場合に返される可能性があります
- MP_MEM ECC キーの計算中にエラーが発生した場合に返される可能性があります

C.21.2.3 function wc_ecc_check_key

```
int wc_ecc_check_key(
    ecc_key * key
)
```

ECC キーの有効性を有効にします。

See: wc_ecc_point_is_at_infinity

Return:

- MP_OKAY 成功、キーは大丈夫です。
- BAD_FUNC_ARG キーが NULL の場合は返します。
- ECC_INF_E WC_ECC_POINT_IS_AT_INFINITY が 1 を返す場合に返します。*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
)
```

この関数は、使用された後に ECC_KEY キーを解放します。*Example*

See:

- wc_ecc_key_new
- wc_ecc_init_ex

```
// 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
)
```

この関数は、ローカル秘密鍵と受信した公開鍵を使用して新しい秘密鍵を生成します。この共有秘密鍵をバッファアウトに格納し、出力バッファに書き込まれたバイト数を保持するために outlenten を更新します。

Parameters:

- **private_key** ローカル秘密鍵を含む ECC_KEY 構造体へのポインタ
- **public_key** 受信した公開鍵を含む ECC_Key 構造体へのポインタ
- **out** 生成された共有秘密鍵を保存する出力バッファへのポインタ *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
}
```

See:

- wc_ecc_init
- wc_ecc_make_key

Return:

- 0 共有秘密鍵の生成に成功したときに返されます
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL に評価された場合に返されます。
- ECC_BAD_ARG_E 引数として与えられた秘密鍵の種類が ecc_privatekey ではない場合、またはパブリックキータイプ（ECC-> DP によって与えられた）が同等でない場合に返されます。
- MEMORY_E 新しい ECC ポイントを生成するエラーが発生した場合
- BUFFER_E 生成された共有秘密鍵が提供されたバッファーに格納するのに長すぎる場合に返されます
- MP_INIT_E 共有キーの計算中にエラーが発生した場合は返される可能性があります
- MP_READ_E 共有キーの計算中にエラーが発生した場合は返される可能性があります
- MP_CMP_E 共有キーの計算中にエラーが発生した場合は返される可能性があります
- MP_INVMOD_E 共有キーの計算中にエラーが発生した場合は返される可能性があります
- MP_EXPTMOD_E 共有キーの計算中にエラーが発生した場合は返される可能性があります
- MP_MOD_E 共有キーの計算中にエラーが発生した場合は返される可能性があります
- MP_MUL_E 共有キーの計算中にエラーが発生した場合は返される可能性があります
- MP_ADD_E 共有キーの計算中にエラーが発生した場合は返される可能性があります
- MP_MULMOD_E 共有キーの計算中にエラーが発生した場合は返される可能性があります
- MP_TO_E 共有キーの計算中にエラーが発生した場合は返される可能性があります
- MP_MEM 共有キーの計算中にエラーが発生した場合は返される可能性があります

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
)
```

秘密鍵とパブリックポイントの間に ECC 共有秘密を作成します。

Parameters:

- **private_key** プライベート ECC キー。
- **point** 使用するポイント（公開鍵）。
- **out** 共有秘密の出力先。ANSI X9.63 から EC-DH に準拠しています。*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
}
```

See: wc_ecc_verify_hash_ex

Return:

- MP_OKAY 成功を示します。
- BAD_FUNC_ARG 引数が NULL のときにエラーが返されます。
- ECC_BAD_ARG_E private_key-> type が ecc_privatekey または private_key-> idx が検証できない場合に返されました。
- BUFFER_E outlen が小さすぎるとエラーが発生します。
- MEMORY_E 新しいポイントを作成するためのエラー。
- MP_VAL 初期化失敗が発生したときに可能です。
- MP_MEM 初期化失敗が発生したときに可能です。

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
)
```

この関数は、信頼性を保証するために ECC_KEY オブジェクトを使用してメッセージダイジェストに署名します。

Parameters:

- **in** サインするメッセージハッシュを含むバッファへのポインタ
- **inlen** 署名するメッセージの長さ
- **out** 生成された署名を保存するためのバッファ
- **outlen** 出力バッファの最大長。メッセージ署名の生成に成功したときに書き込まれたバイトを保存します *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
}
```

See: wc_ecc_verify_hash

Return:

- 0 メッセージダイジェストの署名を正常に生成したときに返されました
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL に評価された場合、または出力バッファが小さすぎて生成された署名を保存する場合は返されます。
- ECC_BAD_ARG_E 入力キーが秘密鍵ではない場合、または ECC OID が無効な場合
- RNG_FAILURE_E RNG が満足のいくキーを正常に生成できない場合に返されます。
- MP_INIT_E メッセージ署名の計算中にエラーが発生した場合に返される可能性があります。
- MP_READ_E メッセージ署名の計算中にエラーが発生した場合に返される可能性があります。
- MP_CMP_E メッセージ署名の計算中にエラーが発生した場合に返される可能性があります。
- MP_INVMOD_E メッセージ署名の計算中にエラーが発生した場合に返される可能性があります。
- MP_EXPTMOD_E メッセージ署名の計算中にエラーが発生した場合に返される可能性があります。
- MP_MOD_E メッセージ署名の計算中にエラーが発生した場合に返される可能性があります。
- MP_MUL_E メッセージ署名の計算中にエラーが発生した場合に返される可能性があります。
- MP_ADD_E メッセージ署名の計算中にエラーが発生した場合に返される可能性があります。
- MP_MULMOD_E メッセージ署名の計算中にエラーが発生した場合に返される可能性があります。
- MP_TO_E メッセージ署名の計算中にエラーが発生した場合に返される可能性があります。
- MP_MEM メッセージ署名の計算中にエラーが発生した場合に返される可能性があります。

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
)
```

メッセージダイジェストに署名します。

Parameters:

- **in** メッセージがサインにダイジェスト。
- **inlen** ダイジェストの長さ。
- **rng** WC_RNG 構造体へのポインタ。
- **key** プライベート ECC キー。
- **r** 署名の R コンポーネントの宛先。*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
}
```

See: wc_ecc_verify_hash_ex

Return:

- MP_OKAY メッセージダイジェストの署名を正常に生成したときに返されました
- ECC_BAD_ARG_E 入力キーが秘密鍵ではない場合、または ECC IDX が無効な場合、またはいずれかの入力パラメータが NULL に評価されている場合、または出力バッファが小さすぎて生成された署名を保存するには小さすぎる場合
- RNG_FAILURE_E RNG が満足のいくキーを正常に生成できない場合に返されます。
- MP_INIT_E メッセージ署名の計算中にエラーが発生した場合に返される可能性があります。
- MP_READ_E メッセージ署名の計算中にエラーが発生した場合に返される可能性があります。
- MP_CMP_E メッセージ署名の計算中にエラーが発生した場合に返される可能性があります。
- MP_INVMOD_E メッセージ署名の計算中にエラーが発生した場合に返される可能性があります。
- MP_EXPTMOD_E メッセージ署名の計算中にエラーが発生した場合に返される可能性があります。
- MP_MOD_E メッセージ署名の計算中にエラーが発生した場合に返される可能性があります。
- MP_MUL_E メッセージ署名の計算中にエラーが発生した場合に返される可能性があります。
- MP_ADD_E メッセージ署名の計算中にエラーが発生した場合に返される可能性があります。
- MP_MULMOD_E メッセージ署名の計算中にエラーが発生した場合に返される可能性があります。
- MP_TO_E メッセージ署名の計算中にエラーが発生した場合に返される可能性があります。
- MP_MEM メッセージ署名の計算中にエラーが発生した場合に返される可能性があります。

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
```

)

この関数は、真正性を確保するためにハッシュの ECC シグネチャを検証します。答えを介して、有効な署名に対応する 1、無効な署名に対応する 0 で答えを返します。

Parameters:

- **sig** 確認する署名を含むバッファへのポインタ
- **siglen** 検証する署名の長さ
- **hash** 確認されたメッセージのハッシュを含むバッファへのポインタ
- **hashlen** 認証されたメッセージのハッシュの長さ
- **stat** 検証の結果へのポインタ。1 メッセージが正常に認証されたことを示します *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
}
```

See:

- wc_ecc_sign_hash
- wc_ecc_verify_hash_ex

Return:

- 0 署名検証に正常に実行されたときに返されます。注：これは署名が検証されていることを意味するわけではありません。信頼性情報は代わりに STAT で格納されます
- BAD_FUNC_ARG 返された入力パラメータは NULL に評価されます
- MEMORY_E メモリの割り当て中にエラーが発生した場合に返されます
- MP_INIT_E メッセージ署名の計算中にエラーが発生した場合に返される可能性があります。
- MP_READ_E メッセージ署名の計算中にエラーが発生した場合に返される可能性があります。
- MP_CMP_E メッセージ署名の計算中にエラーが発生した場合に返される可能性があります。
- MP_INVMOD_E メッセージ署名の計算中にエラーが発生した場合に返される可能性があります。
- MP_EXPTMOD_E メッセージ署名の計算中にエラーが発生した場合に返される可能性があります。
- MP_MOD_E メッセージ署名の計算中にエラーが発生した場合に返される可能性があります。
- MP_MUL_E メッセージ署名の計算中にエラーが発生した場合に返される可能性があります。
- MP_ADD_E メッセージ署名の計算中にエラーが発生した場合に返される可能性があります。
- MP_MULMOD_E メッセージ署名の計算中にエラーが発生した場合に返される可能性があります。
- MP_TO_E メッセージ署名の計算中にエラーが発生した場合に返される可能性があります。
- MP_MEM メッセージ署名の計算中にエラーが発生した場合に返される可能性があります。

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
)
```

ECC 署名を確認してください。結果は stat に書き込まれます。1 が有効で、0 が無効です。注：有効なテストに戻り値を使用しないでください。stat のみを使用してください。

Parameters:

- **r** 検証する署名 R コンポーネント
- **s** 検証するシグネチャ S コンポーネント
- **hash** 署名されたハッシュ（メッセージダイジェスト）
- **hashlen** ハッシュの長さ（オクテット）
- **stat** 署名の結果、1 == 有効、0 == 無効 *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
}
```

See: wc_ecc_verify_hash

Return:

- MP_OKAY 成功した場合（署名が無効であっても）
- ECC_BAD_ARG_E 引数が NULL の場合、または key-idx が無効な場合は返します。
- MEMORY_E INT またはポイントの割り当て中にエラーが発生しました。

C.21.2.11 function wc_ecc_init

```
int wc_ecc_init(
    ecc_key * key
)
```

この関数は、メッセージ検証または鍵交渉で将来の使用のために ECC_KEY オブジェクトを初期化します。

See:

- wc_ecc_make_key
- wc_ecc_free

Return:

- 0 ECC_Key オブジェクトの初期化に成功したときに返されます
- MEMORY_E メモリの割り当て中にエラーが発生した場合に返されます *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
)
```

この関数は、メッセージ検証または鍵交渉で将来の使用のために ECC_KEY オブジェクトを初期化します。

Parameters:

- **key** 初期化する ECC_Key オブジェクトへのポインタ
- **devId** 非同期ハードウェアで使用する ID *Example*

```
ecc_key key;
wc_ecc_init_ex(&key, heap, devId);
```

See:

- wc_ecc_make_key
- wc_ecc_free
- wc_ecc_init

Return:

- 0 ECC_Key オブジェクトの初期化に成功したときに返されます
- MEMORY_E メモリの割り当て中にエラーが発生した場合に返されます

C.21.2.13 function wc_ecc_key_new

```
ecc_key * wc_ecc_key_new(
    void * heap
)
```

この関数はユーザー定義ヒープを使用し、キー構造のスペースを割り当てます。

See:

- wc_ecc_make_key
- wc_ecc_key_free
- wc_ecc_init

Return: 0 ECC_Key オブジェクトの初期化に成功したときに返されます *Example*

```
wc_ecc_key_new(&heap);
```

C.21.2.14 function wc_ecc_free

```
int wc_ecc_free(
    ecc_key * key
)
```

この関数は、使用後に ECC_KEY オブジェクトを解放します。

See: wc_ecc_init

Return: int integer が WolfSSL エラーまたは成功状況を示すことを返しました。*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
)
```

この関数は固定小数点キャッシュを解放します。これは ECC で使用でき、計算時間を高速化します。この機能を使用するには、FP_ECC（固定小数点 ECC）を定義する必要があります。

See: wc_ecc_free

Return: none いいえ返します。*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
)
```

ECC IDX が有効かどうかを確認します。

See: none

Return:

- 1 有効な場合は返品してください。
- 0 無効な場合は返します。*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
)
```

新しい ECC ポイントを割り当てます。

See:

- wc_ecc_del_point
- wc_ecc_cmp_point
- wc_ecc_copy_point

Return:

- p 新しく割り当てられたポイント。

- NULL エラー時に NULL を返します。*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
)
```

メモリから ECC ポイントを解放します。

See:

- wc_ecc_new_point
- wc_ecc_cmp_point
- wc_ecc_copy_point

Return: none いいえ返します。*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
)
```

あるポイントの値を別のポイントにコピーします。

Parameters:

- **p** コピーするポイント。*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
}
```

See:

- wc_ecc_new_point
- wc_ecc_cmp_point
- wc_ecc_del_point

Return:

- ECC_BAD_ARG_E P または R が NULL のときにスローされたエラー。
- MP_OKAY ポイントが正常にコピーされました
- ret 内部関数からのエラー。になることができます...

C.21.2.20 function wc_ecc_cmp_point

```
int wc_ecc_cmp_point(
    ecc_point * a,
    ecc_point * b
)
```

ポイントの値を別のものと比較してください。

Parameters:

- **a** 比較する最初のポイント。*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
}
```

See:

- wc_ecc_new_point
- wc_ecc_del_point
- wc_ecc_copy_point

Return:

- BAD_FUNC_ARG 1 つまたは両方の引数は null です。
- MP_EQ ポイントは同じです。
- ret mp_lt または mp_gt のどちらかで、ポイントが等しくないことを意味します。

C.21.2.21 function wc_ecc_point_is_at_infinity

```
int wc_ecc_point_is_at_infinity(
    ecc_point * p
)
```

ポイントが無限大にあるかどうかを確認します。返品 1 が無限大である場合は 0、そうでない場合は 0、<0 エラー時の 0

See:

- wc_ecc_new_point
- wc_ecc_del_point
- wc_ecc_cmp_point
- wc_ecc_copy_point

Return:

- 1 P は無限大です。
- 0 P は無限大ではありません。
- <0 エラー。*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
)
```

ECC 固定点乗算を実行します。

Parameters:

- **k** 計量。
- **G** 乗算する基点。
- **R** 商品の目的地
- **modulus** 曲線の弾性率 *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;
```

See: none

Return:

- MP_OKAY 成功した操作で返します。
- MP_INIT_E 複数の Precision Integer（MP_INT）ライブラリで使用するための整数を初期化するエラーがある場合に返されます。

C.21.2.23 function wc_ecc_export_x963

```
int wc_ecc_export_x963(
    ecc_key * key,
    byte * out,
    word32 * outLen
)
```

この関数は ECC キーを ECC_KEY 構造体からエクスポートし、結果を OUT に格納します。キーは ANSI X9.63 フォーマットに保存されます。outlen の出力バッファに書き込まれたバイトを格納します。

Parameters:

- **key** エクスポートする ECC_KEY オブジェクトへのポインタ
- **out** ANSI X9.63 フォーマット済みキーを保存するバッファへのポインタ *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
}
```

See:

- wc_ecc_export_x963_ex
- wc_ecc_import_x963

Return:

- 0 ECC_KEY のエクスポートに正常に返されました
- LENGTH_ONLY_E 出力バッファが NULL に評価されている場合は返されますが、他の 2 つの入力パラメータは有効です。関数がキーを保存するのに必要な長さを返すだけであることを示します
- ECC_BAD_ARG_E 入力パラメータのいずれかが NULL の場合、またはキーがサポートされていない場合（無効なインデックスがあります）
- BUFFER_E 出力バッファが小さすぎて ECC キーを保存する場合は返されます。出力バッファが小さすぎると、必要なサイズは outlen に返されます。
- MEMORY_E xmalloc でメモリを割り当てるエラーがある場合

- MP_INIT_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_READ_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_CMP_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_INVMOD_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_EXPTMOD_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_MOD_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_MUL_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_ADD_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_MULMOD_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_TO_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_MEM ECC_KEY の処理中にエラーが発生した場合に返される可能性があります

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
)
```

この関数は ECC キーを ECC_KEY 構造体からエクスポートし、結果を OUT に格納します。キーは ANSI X9.63 フォーマットに保存されます。outlen の出力バッファに書き込まれたバイトを格納します。この関数は、圧縮されたパラメータを介して証明書を圧縮する追加のオプションを使用する。このパラメータが true の場合、キーは ANSI X9.63 圧縮形式で保存されます。

Parameters:

- **key** エクスポートする ECC_KEY オブジェクトへのポインタ
- **out** ANSI X9.63 フォーマット済みキーを保存するバッファへのポインタ
- **outLen** 出力バッファのサイズ。キーの保存に成功した場合は、出力バッファに書き込まれたバイトを保持します。*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
}
```

See:

- wc_ecc_export_x963
- wc_ecc_import_x963

Return:

- 0 ECC_KEY のエクスポートに正常に返されました
- NOT_COMPILED_IN hand_comp_key がコンパイル時に有効になっていない場合は返されますが、キーは圧縮形式で要求されました
- LENGTH_ONLY_E 出力バッファが NULL に評価されている場合は返されますが、他の 2 つの入力パラメータは有効です。関数がキーを保存するのに必要な長さを返すだけであることを示します
- ECC_BAD_ARG_E 入力パラメータのいずれかが NULL の場合、またはキーがサポートされていない場合（無効なインデックスがあります）

- BUFFER_E 出力バッファが小さすぎて ECC キーを保存する場合は返されます。出力バッファが小さすぎると、必要なサイズは outlen に返されます。
- MEMORY_E xmalloc でメモリを割り当てるエラーがある場合
- MP_INIT_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_READ_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_CMP_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_INVMOD_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_EXPTMOD_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_MOD_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_MUL_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_ADD_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_MULMOD_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_TO_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_MEM ECC_KEY の処理中にエラーが発生した場合に返される可能性があります

C.21.2.25 function wc_ecc_import_x963

```
int wc_ecc_import_x963(
    const byte * in,
    word32 inLen,
    ecc_key * key
)
```

この関数は、ANSI X9.63 形式で保存されているキーを含むバッファからパブリック ECC キーをインポートします。この関数は、圧縮キーが hand_comp_key オプションを介してコンパイル時に有効になっている限り、圧縮キーと非圧縮キーの両方を処理します。

Parameters:

- **in** ANSI x9.63 フォーマットされた ECC キーを含むバッファへのポインタ
- **inLen** 入力バッファの長さ *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
}
```

See:

- wc_ecc_export_x963
- wc_ecc_import_private_key

Return:

- 0 ECC_KEY のインポートに成功しました
- NOT_COMPILED_IN hand_comp_key がコンパイル時に有効になっていない場合は返されますが、キーは圧縮形式で保存されます。
- ECC_BAD_ARG_E IN または KEY が NULL に評価された場合、または Inlen が偶数の場合（X9.63 規格によれば、キーは奇数でなければなりません）。
- MEMORY_E メモリの割り当て中にエラーが発生した場合に返されます
- ASN_PARSE_E ECC キーの解析中にエラーがある場合は返されます。ECC キーが有効な ANSI X9.63 フォーマットに格納されていないことを示すことがあります。

- IS_POINT_E エクスポートされた公開鍵が ECC 曲線上の点ではない場合に返されます
- MP_INIT_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_READ_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_CMP_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_INVMOD_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_EXPTMOD_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_MOD_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_MUL_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_ADD_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_MULMOD_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_TO_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_MEM ECC_KEY の処理中にエラーが発生した場合に返される可能性があります

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
)
```

この関数は、生の秘密鍵を含むバッファと、ANSI X9.63 フォーマットされた公開鍵を含む 2 番目のバッファーからパブリック/プライベート ECC キーのペアをインポートします。この関数は、圧縮キーが hand_comp_key オプションを介してコンパイル時に有効になっている限り、圧縮キーと非圧縮キーの両方を処理します。

Parameters:

- **priv** RAW 秘密鍵を含むバッファへのポインタ
- **privSz** 秘密鍵バッファのサイズ
- **pub** ANSI x9.63 フォーマットされた ECC 公開鍵を含むバッファへのポインタ
- **pubSz** 公開鍵入力バッファの長さ *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
}
```

See:

- wc_ecc_export_x963
- wc_ecc_import_private_key

Return:

- 0 habe_comp_key がコンパイル時に有効になっていない場合は、ecc_key not_compiled_in を正常にインポートしましたが、キーは圧縮形式で保存されます。
- ECC_BAD_ARG_E IN または KEY が NULL に評価された場合、または Inlen が偶数の場合（X9.63 規格によれば、キーは奇数でなければなりません）。

- MEMORY_E メモリの割り当て中にエラーが発生した場合に返されます
- ASN_PARSE_E ECC キーの解析中にエラーがある場合は返されます。ECC キーが有効な ANSI X9.63 フォーマットに格納されていないことを示すことがあります。
- IS_POINT_E エクスポートされた公開鍵が ECC 曲線上の点ではない場合に返されます
- MP_INIT_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_READ_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_CMP_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_INVMOD_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_EXPTMOD_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_MOD_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_MUL_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_ADD_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_MULMOD_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_TO_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_MEM ECC_KEY の処理中にエラーが発生した場合に返される可能性があります

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
)
```

この関数は、ECC シグネチャの R 部分と S 部分を DER 符号化 ECDSA シグネチャに変換します。この機能は、outlen では、出力バッファに書き込まれた長さも記憶されています。

Parameters:

- **r** 署名の R 部分を文字列として含むバッファへのポインタ
- **s** シグネチャの S 部分を含むバッファへのポインタ文字列としてのポインタ
- **out** DER エンコードされた ECDSA シグネチャを保存するバッファへのポインタ *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
}
```

See:

- wc_ecc_sign_hash
- wc_ecc_sig_size

Return:

- 0 署名の変換に成功したことに戻りました
- ECC_BAD_ARG_E いずれかの入力パラメータが NULL に評価された場合、または入力バッファが DER エンコードされた ECDSA シグネチャを保持するのに十分な大きさでない場合に返されます。

- MP_INIT_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_READ_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_CMP_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_INVMOD_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_EXPTMOD_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_MOD_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_MUL_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_ADD_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_MULMOD_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_TO_E ECC_KEY の処理中にエラーが発生した場合に返される可能性があります
- MP_MEM ECC_KEY の処理中にエラーが発生した場合に返される可能性があります

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
)
```

この関数は、ECC 署名の RAW 成分を持つ ECC_KEY 構造体を埋めます。

Parameters:

- **key** 塗りつぶすための ECC_KEY 構造体へのポインタ
- **qx** ASCII 六角文字列として基点の X コンポーネントを含むバッファへのポインタ
- **qy** ASCII 六角文字列として基点の Y 成分を含むバッファへのポインタ
- **d** ASCII hex 文字列として秘密鍵を含むバッファへのポインタ *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
}
```

See: wc_ecc_import_private_key

Return:

- 0 ECC_Key 構造体に正常にインポートされたときに返されます
- ECC_BAD_ARG_E いずれかの入力値が NULL に評価された場合に返されます。
- MEMORY_E ECC_Key のパラメータを格納するためのエラーの初期化スペースがある場合に返されます。
- ASN_PARSE_E 入力カーベナデが ECC_SETS で定義されていない場合
- MP_INIT_E 入力パラメータの処理中にエラーが発生した場合に返される可能性があります。
- MP_READ_E 入力パラメータの処理中にエラーが発生した場合に返される可能性があります。
- MP_CMP_E 入力パラメータの処理中にエラーが発生した場合に返される可能性があります。
- MP_INVMOD_E 入力パラメータの処理中にエラーが発生した場合に返される可能性があります。
- MP_EXPTMOD_E 入力パラメータの処理中にエラーが発生した場合に返される可能性があります。
- MP_MOD_E 入力パラメータの処理中にエラーが発生した場合に返される可能性があります。

- MP_MUL_E 入力パラメータの処理中にエラーが発生した場合に返される可能性があります。
- MP_ADD_E 入力パラメータの処理中にエラーが発生した場合に返される可能性があります。
- MP_MULMOD_E 入力パラメータの処理中にエラーが発生した場合に返される可能性があります。
- MP_TO_E 入力パラメータの処理中にエラーが発生した場合に返される可能性があります。
- MP_MEM 入力パラメータの処理中にエラーが発生した場合に返される可能性があります。

C.21.2.29 function wc_ecc_export_private_only

```
int wc_ecc_export_private_only(
    ecc_key * key,
    byte * out,
    word32 * outLen
)
```

この関数は、ECC_KEY 構造体から秘密鍵のみをエクスポートします。秘密鍵をバッファアウトに格納し、outlen にこのバッファに書き込まれたバイトを設定します。

Parameters:

- **key** 秘密鍵をエクスポートする ECC_Key 構造体へのポインタ
- **out** 秘密鍵を保存するバッファへのポインタ *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
}
```

See: wc_ecc_import_private_key

Return:

- 0 秘密鍵のエクスポートに成功したときに返されます
- ECC_BAD_ARG_E いずれかの入力値が NULL に評価された場合に返されます。
- MEMORY_E ECC_Key のパラメータを格納するためのエラーの初期化スペースがある場合に返されます。
- ASN_PARSE_E 入力カーベナデが ECC_SETS で定義されていない場合
- MP_INIT_E 入力パラメータの処理中にエラーが発生した場合に返される可能性があります。
- MP_READ_E 入力パラメータの処理中にエラーが発生した場合に返される可能性があります。
- MP_CMP_E 入力パラメータの処理中にエラーが発生した場合に返される可能性があります。
- MP_INVMOD_E 入力パラメータの処理中にエラーが発生した場合に返される可能性があります。
- MP_EXPTMOD_E 入力パラメータの処理中にエラーが発生した場合に返される可能性があります。
- MP_MOD_E 入力パラメータの処理中にエラーが発生した場合に返される可能性があります。
- MP_MUL_E 入力パラメータの処理中にエラーが発生した場合に返される可能性があります。
- MP_ADD_E 入力パラメータの処理中にエラーが発生した場合に返される可能性があります。
- MP_MULMOD_E 入力パラメータの処理中にエラーが発生した場合に返される可能性があります。
- MP_TO_E 入力パラメータの処理中にエラーが発生した場合に返される可能性があります。
- MP_MEM 入力パラメータの処理中にエラーが発生した場合に返される可能性があります。

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
)
```

DER へのエクスポートポイント。

Parameters:

- **curve_idx** ECC_SETS から使用される曲線のインデックス。
- **point** Der へのエクスポートを指す。
- **out** 出力の目的地 *Example*

```
int curve_idx;
ecc_point* point;
byte out[];
word32 outLen;
wc_ecc_export_point_der(curve_idx, point, out, &outLen);
```

See: wc_ecc_import_point_der

Return:

- 0 成功に戻りました。
- ECC_BAD_ARG_E curve_idx が 0 未満または無効である場合は返します。いつ来るのか
- LENGTH_ONLY_E outlen は設定されていますが、他にはありません。
- BUFFER_E outlenn が 1 + 2 * 曲線サイズよりも小さい場合は返します。
- MEMORY_E メモリの割り当てに問題がある場合は返します。

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
)
```

Der フォーマットからのインポートポイント。

Parameters:

- **in** からのポイントをインポートするための Der Buffer。
- **inLen** DER バッファの長さ
- **curve_idx** 曲線のインデックス *Example*

```
byte in[];
word32 inLen;
int curve_idx;
ecc_point* point;
wc_ecc_import_point_der(in, inLen, curve_idx, point);
```

See: wc_ecc_export_point_der

Return:

- ECC_BAD_ARG_E 引数が null の場合、または Inlen が偶数の場合は返します。
- MEMORY_E エラー初期化がある場合に返します
- NOT_COMPILED_IN habe_comp_key が真実でない場合は返され、in は圧縮証明書です
- MP_OKAY 操作が成功しました。

C.21.2.32 function wc_ecc_size

```
int wc_ecc_size(
    ecc_key * key
)
```

この関数は、ecc_key 構造体のキーサイズをオクテットで返します。

See: wc_ecc_make_key

Return:

- Given 有効なキー、オクテットのキーサイズを返します
- 0 与えられたキーが NULL の場合に返されます *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
)
```

この関数は、次のようにして指定された ECC シグネチャの最悪の場合のサイズを返します。（KEYSZ * 2）+ SIG_HEADER_SZ + ECC_MAX_PAD_SZ。実際のシグネチャサイズは、WC_ECC_SIGN_HASH で計算できます。

See:

- wc_ecc_sign_hash
- wc_ecc_sig_size

Return: returns 最大署名サイズ（オクテット）*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
)
```

この関数は、次のようにして指定された ECC シグネチャの最悪の場合のサイズを返します。（KEYSZ * 2）+ SIG_HEADER_SZ + ECC_MAX_PAD_SZ。実際のシグネチャサイズは、WC_ECC_SIGN_HASH で計算できます。

See:

- wc_ecc_sign_hash
- wc_ecc_sig_size_calc

Return:

- Success 有効なキーを考えると、最大署名サイズをオクテットで返します。
- 0 与えられたキーが NULL の場合に返されます *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
)
```

この機能は、ECC との安全なメッセージ交換を可能にするために、新しい ECC コンテキストオブジェクトのスペースを割り当て、初期化します。

Parameters:

- **flags** これがサーバーであるかクライアントのコンテキストオプションがあるかどうかを示します.req_resp_client、および req_resp_server *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
}
```

See:

- wc_ecc_encrypt
- wc_ecc_encrypt_ex
- wc_ecc_decrypt

Return:

- Success 新しい ECENCCTX オブジェクトの生成に成功した場合は、そのオブジェクトへのポインタを返します
- NULL 関数が新しい ECENCCTX オブジェクトを生成できない場合に返されます

C.21.2.36 function wc_ecc_ctx_free

```
void wc_ecc_ctx_free(
    ecEncCtx *
)
```

この関数は、メッセージの暗号化と復号化に使用される ECENCCTX オブジェクトを解放します。

See: wc_ecc_ctx_new

Return: none 戻り値。*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
)
```

この関数は ECENCCTX 構造をリセットして、新しいコンテキストオブジェクトを解放し、新しいコンテキストオブジェクトを割り当てます。

Parameters:

- **ctx** リセットする ECENCCTX オブジェクトへのポインタ *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
```

See: wc_ecc_ctx_new

Return:

- 0 ecencctx 構造が正常にリセットされた場合に返されます
- BAD_FUNC_ARG RNG または CTX が NULL の場合に返されます
- RNG_FAILURE_E ECC オブジェクトに新しいソルトを生成するエラーがある場合

C.21.2.38 function wc_ecc_ctx_set_algo

```
int wc_ecc_ctx_set_algo(
    ecEncCtx * ctx,
    byte encAlgo,
    byte kdfAlgo,
    byte macAlgo
)
```

この関数は、wc_ecc_ctx_new の後にオプションで呼び出されることができます。暗号化、KDF、および MAC アルゴリズムを ECENCENCCTX オブジェクトに設定します。

Parameters:

- **ctx** 情報を設定する ECENCCTX へのポインタ
- **encAlgo** 使用する暗号化アルゴリズム
- **kdfAlgo** 使用する KDF アルゴリズム *Example*

```
ecEncCtx* ctx;
// initialize ctx
if(wc_ecc_ctx_set_algo(&ctx, ecAES_128_CTR, ecHKDF_SHA256, ecHMAC_SHA256)) {
```

```
    // error setting info
}
```

See: wc_ecc_ctx_new

Return:

- 0 ECENCCTX オブジェクトの情報を正常に設定すると返されます。
- BAD_FUNC_ARG 指定された ecencctx オブジェクトが NULL の場合に返されます。

C.21.2.39 function wc_ecc_ctx_get_own_salt

```
const byte * wc_ecc_ctx_get_own_salt(
    ecEncCtx *
)
```

この関数は ECENCENCCTX オブジェクトのソルトを返します。この関数は、ECENCCTX の状態が ECSRV_INIT または ECCLI_INIT の場合にのみ呼び出す必要があります。

See:

- wc_ecc_ctx_new
- wc_ecc_ctx_set_peer_salt

Return:

- 成功すると、ecEncCtx ソルトを返します
- NULL ecencctx オブジェクトが NULL の場合、または ECENCCTX の状態が ECSRV_INIT または ECCLI_INIT でない場合に返されます。後者の 2 つのケースでは、この機能はそれぞれ ECSRV_BAD_STATE または ECCLI_BAD_STATE に ECENCCTX の状態を設定します。*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
)
```

この関数は、ECENCENCCTX オブジェクトのピアソルトを設定します。

Parameters:

- **ctx** ソルトを設定するための ecencctx へのポインタ *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);
```

See: wc_ecc_ctx_get_own_salt

Return:

- 0 ECENCCTX オブジェクトのピアソルトの設定に成功したときに返されます。
- BAD_FUNC_ARG 指定された ecencctx オブジェクトが null または無効なプロトコルがある場合、または指定されたソルトが NULL の場合
- BAD_ENC_STATE_E ecencctx の状態が ECSRV_SALT_GET または ECCLI_SALT_GET の場合に返されます。後者の 2 つのケースでは、この機能はそれぞれ ECSRV_BAD_STATE または ECCLI_BAD_STATE に ECENCCTX の状態を設定します。

C.21.2.41 function wc_ecc_ctx_set_info

```
int wc_ecc_ctx_set_info(
    ecEncCtx * ctx,
    const byte * info,
    int sz
)
```

この関数は、wc_ecc_ctx_set_peer_salt の前後にオプションで呼び出されることができます。ECENCCTX オブジェクトのオプションの情報を設定します。

Parameters:

- **ctx** 情報を設定する ECENCCTX へのポインタ
- **info** 設定する情報を含むバッファへのポインタ *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
}
```

See: wc_ecc_ctx_new

Return:

- 0 ECENCCTX オブジェクトの情報を正常に設定すると返されます。
- BAD_FUNC_ARG 与えられた ECENCCTX オブジェクトが NULL の場合、入力情報は NULL またはサイズが無効です。

C.21.2.42 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
)
```

この関数は指定された入力メッセージを MSG から OUT に暗号化します。この関数はパラメータとしてオプションの CTX オブジェクトを取ります。提供されている場合、ECENCCTX の Encalgo、Kdfalgo、および Macalgo に基づいて暗号化が進みます。CTX が指定されていない場合、処理はデフォルトのアルゴリズム、ECAES_128_CBC、ECHKDF_SHA256、ECHMAC_SHA256 で完了します。この機能は、メッセージが CTX で指定された暗号化タイプに従って埋め込まれている必要があります。

Parameters:

- **privKey** 暗号化に使用する秘密鍵を含む ECC_KEY オブジェクトへのポインタ
- **pubKey** コミュニケーションを希望するピアの公開鍵を含む ECC_Key オブジェクトへのポインタ
- **msg** 暗号化するメッセージを保持しているバッファへのポインタ
- **msgSz** 暗号化するバッファのサイズ
- **out** 暗号化された暗号文を保存するバッファへのポインタ
- **outSz** OUT バッファ内の使用可能なサイズを含む Word32 オブジェクトへのポインタ。メッセージの暗号化に成功したら、出力バッファに書き込まれたバイト数を保持します。*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
}
```

See:

- wc_ecc_encrypt_ex
- wc_ecc_decrypt

Return:

- 0 入力メッセージの暗号化に成功したら返されます
- BAD_FUNC_ARG PRIVKEY、PUBKEY、MSG、MSGSZ、OUT、OUTSZ が NULL の場合、または CTX オブジェクトがサポートされていない暗号化タイプを指定します。
- BAD_ENC_STATE_E 指定された CTX オブジェクトが暗号化に適していない状態にある場合に返されます。
- BUFFER_E 指定された出力バッファが小さすぎて暗号化された暗号文を保存する場合に返されます
- MEMORY_E 共有秘密鍵のメモリの割り当て中にエラーがある場合に返されます

C.21.2.43 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
)
```

この関数は指定された入力メッセージを MSG から OUT に暗号化します。この関数はパラメータとしてオプションの CTX オブジェクトを取ります。提供されている場合、ECENCCTX の Encalgo、Kdfalgo、および Macalgo に基づいて暗号化が進みます。CTX が指定されていない場合、処理はデフォルトのアルゴリズム、ECAES_128_CBC、ECHKDF_SHA256、ECHMAC_SHA256 で完了します。この機能は、メッセージが CTX で指定された暗号化タイプに従って埋め込まれている必要があります。

Parameters:

- **privKey** 暗号化に使用する秘密鍵を含む ECC_KEY オブジェクトへのポインタ
- **pubKey** コミュニケーションを希望するピアの公開鍵を含む ECC_Key オブジェクトへのポインタ
- **msg** 暗号化するメッセージを保持しているバッファへのポインタ
- **msgSz** 暗号化するバッファのサイズ
- **out** 暗号化された暗号文を保存するバッファへのポインタ
- **outSz** OUT バッファ内の使用可能なサイズを含む Word32 オブジェクトへのポインタ。メッセージの暗号化に成功したら、出力バッファに書き込まれたバイト数を保持します。
- **ctx** オプション：使用するさまざまな暗号化アルゴリズムを指定する ECENCCTX オブジェクトへのポインタ *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
}
```

See:

- wc_ecc_encrypt
- wc_ecc_decrypt

Return:

- 0 入力メッセージの暗号化に成功したら返されます
- BAD_FUNC_ARG PRIVKEY、PUBKEY、MSG、MSGSZ、OUT、OUTSZ が NULL の場合、または CTX オブジェクトがサポートされていない暗号化タイプを指定します。
- BAD_ENC_STATE_E 指定された CTX オブジェクトが暗号化に適していない状態にある場合に返されます。
- BUFFER_E 指定された出力バッファが小さすぎて暗号化された暗号文を保存する場合に返されます
- MEMORY_E 共有秘密鍵のメモリの割り当て中にエラーがある場合に返されます

C.21.2.44 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
)
```

この関数は MSG から OUT への暗号文を復号化します。この関数はパラメータとしてオプションの CTX オブジェクトを取ります。提供されている場合、ECENCCTX の Encalgo、Kdfalgo、および Macalgo に基づいて暗号化が進みます。CTX が指定されていない場合、処理はデフォルトのアルゴリズム、ECAES_128_CBC、ECHKDF_SHA256、ECHMAC_SHA256 で完了します。この機能は、メッセージが CTX で指定された暗号化タイプに従って埋め込まれている必要があります。

Parameters:

- **privKey** 復号化に使用する秘密鍵を含む ECC_Key オブジェクトへのポインタ
- **pubKey** コミュニケーションを希望するピアの公開鍵を含む ECC_Key オブジェクトへのポインタ
- **msg** 暗号文を復号化するためのバッファへのポインタ
- **msgSz** 復号化するバッファのサイズ
- **out** 復号化された平文を保存するバッファへのポインタ
- **outSz** OUT バッファ内の使用可能なサイズを含む Word32 オブジェクトへのポインタ。暗号文を正常に復号化すると、出力バッファに書き込まれたバイト数を保持します。*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
}
```

See:

- wc_ecc_encrypt
- wc_ecc_encrypt_ex

Return:

- 0 入力メッセージの復号化に成功したときに返されます
- BAD_FUNC_ARG PRIVKEY、PUBKEY、MSG、MSGSZ、OUT、OUTSZ が NULL の場合、または CTX オブジェクトがサポートされていない暗号化タイプを指定します。
- BAD_ENC_STATE_E 指定された CTX オブジェクトが復号化に適していない状態にある場合に返されます。
- BUFFER_E 供給された出力バッファが小さすぎて復号化された平文を保存する場合は返されます。
- MEMORY_E 共有秘密鍵のメモリの割り当て中にエラーがある場合に返されます

C.21.2.45 function wc_ecc_set_nonblock

```
int wc_ecc_set_nonblock(
    ecc_key * key,
    ecc_nb_ctx_t * ctx
)
```

非ブロック操作のための ECC サポートを有効にします。次のビルドオプションを使用した単精度（SP）数学でサポートされています.WolfSSL_SP_SP_SMALL WOLFSSL_SP_NO_MALLOC WC_ECC_NONBLOCK

Parameters:

- **key** ECC_KEY オブジェクトへのポインタ *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);
}
```

Return: 0 コールバックコンテキストを入力メッセージに正常に設定すると返されます。

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_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);
```

C.22 dox_comments/header_files-ja/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)

	Name
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)

	Name
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-ja/ed25519.h

C.23.1 Functions

	Name
int	**wc_ed25519_make_public**(ed25519_key * key, unsigned char * pubKey, word32 pubKeySz) この関数は Ed25519 秘密鍵から Ed25519 公開鍵を生成します。公開鍵をバッファ pubkey に出力します。この関数の呼び出しに先立ち、ed25519_key 構造体には Ed25519 秘密鍵がインポートされている必要があります。
int	**wc_ed25519_make_key**(WC_RNG * rng, int keysize, ed25519_key * key) この関数は新しい ed25519_key 構造体を生成し、それを引数 key のバッファに格納します。
int	**wc_ed25519_sign_msg**(const byte * in, word32 inlen, byte * out, word32 * outlen, ed25519_key * key) この関数は、ed25519_key 構造体を使用してメッセージに署名します。
int	**wc_ed25519ctx_sign_msg**(const byte * in, word32 inlen, byte * out, word32 * outlen, ed25519_key * key, const byte * context, byte contextLen) この関数は、ed25519_key 構造体を使用してメッセージに署名します。コンテキストは署名されるデータの一部です。

	Name
int	**wc_ed25519ph_sign_hash**(const byte * hash, word32 hashLen, byte * out, word32 * outLen, ed25519_key * key, const byte * context, byte contextLen) この関数は、ed25519_key 構造体を使用してメッセージダイジェストに署名します。コンテキストは署名されるデータの一部として含まれています。署名計算の前にメッセージは事前にハッシュされています。メッセージダイジェストを作成するために使用されるハッシュアルゴリズムは Shake-256 でなければなりません。
int	**wc_ed25519ph_sign_msg**(const byte * in, word32 inlen, byte * out, word32 * outlen, ed25519_key * key, const byte * context, byte contextLen) この関数は、ed25519_key 構造体を使用して認証を保証するメッセージに署名します。コンテキストは署名されたデータの一部として含まれています。署名計算の前にメッセージは事前にハッシュされています。
int	**wc_ed25519_verify_msg**(const byte * sig, word32 siglen, const byte * msg, word32 msgLen, int * ret, ed25519_key * key) この関数はメッセージの Ed25519 署名を検証します。ret を介して答えを返し、有効な署名の場合は 1、無効な署名の場合には 0 を返します。
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) この関数はメッセージの Ed25519 署名を検証します。コンテキストは署名されたデータの一部として含まれています。答えは変数 ret を介して返され、署名が有効ならば 1、無効ならば 0 を返します。
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) この関数は、メッセージのダイジェストの Ed25519 署名を検証します。引数 hash は、署名計算前のプリハッシュメッセージです。メッセージダイジェストを作成するために使用されるハッシュアルゴリズムは SHA-512 でなければなりません。答えは変数 ret を介して返され、署名が有効ならば 1、無効ならば 0 を返します。

	Name
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) この関数は、メッセージのダイジェストの Ed25519 署名を検証します。引数 context は検証すべきデータの一部として含まれています。検証前にメッセージがプリハッシュされています。答えは変数 res を介して返され、署名が有効ならば 1、無効ならば 0 を返します。
int	**wc_ed25519_init**(ed25519_key * key) この関数は、後のメッセージ検証で使用のために ed25519_key 構造体を初期化します。
void	**wc_ed25519_free**(ed25519_key * key) この関数は、使用済みの ed25519_key 構造体を解放します。
int	**wc_ed25519_import_public**(const byte * in, word32 inLen, ed25519_key * key) この関数はバッファから ed25519 公開鍵を ed25519_key 構造体へインポートします。圧縮あるいは非圧縮の両方の形式の鍵を扱います。
int	**wc_ed25519_import_public_ex**(const byte * in, word32 inLen, ed25519_key * key, int trusted) この関数はバッファから ed25519 公開鍵を ed25519_key 構造体へインポートします。圧縮あるいは非圧縮の両方の形式の鍵を扱います。秘密鍵が既にインポートされている場合で、trusted 引数が 1 以外の場合は両鍵が対応しているかをチェックします。
int	**wc_ed25519_import_private_only**(const byte * priv, word32 privSz, ed25519_key * key) この関数は、ed25519 秘密鍵のみをバッファからインポートします。
int	**wc_ed25519_import_private_key**(const byte * priv, word32 privSz, const byte * pub, word32 pubSz, ed25519_key * key) この関数は、Ed25519 公開鍵/秘密鍵をそれぞれ含む一対のバッファから Ed25519 鍵ペアをインポートします。この関数は圧縮と非圧縮の両方の鍵を処理します。
int	**wc_ed25519_import_private_key_ex**(const byte * priv, word32 privSz, const byte * pub, word32 pubSz, ed25519_key * key, int trusted) この関数は一対のバッファから Ed25519 公開鍵/秘密鍵ペアをインポートします。この関数は圧縮キーと非圧縮キーの両方を処理します。公開鍵は trusted 引数により信頼されていないとされた場合には秘密鍵に対して検証されます。

	Name
int	**wc_ed25519_export_public**(ed25519_key * key, byte * out, word32 * outLen) この関数は、ed25519_key 構造体から公開鍵をエクスポートします。公開鍵をバッファ out に格納し、outLen にこのバッファに書き込まれたバイトを設定します。
int	**wc_ed25519_export_private_only**(ed25519_key * key, byte * out, word32 * outLen) この関数は、ed25519_key 構造体からの秘密鍵のみをエクスポートします。秘密鍵をバッファアウトに格納し、outlen にこのバッファに書き込まれたバイトを設定します。
int	**wc_ed25519_export_private**(ed25519_key * key, byte * out, word32 * outLen) この関数は、ed25519_key 構造体から鍵ペアをエクスポートします。鍵ペアをバッファ out に格納し、ounteren でこのバッファに書き込まれたバイトを設定します。
int	**wc_ed25519_export_key**(ed25519_key * key, byte * priv, word32 * privSz, byte * pub, word32 * pubSz) この関数は、ed25519_key 構造体から秘密鍵と公開鍵を別々にエクスポートします。秘密鍵をバッファ priv に格納し、priovSz にこのバッファに書き込んだバイト数を設定します。公開鍵をバッファ pub に格納し、pubSz にこのバッファに書き込んだバイト数を設定します。
int	**wc_ed25519_check_key**(ed25519_key * key) この関数は、ed25519_key 構造体の公開鍵をチェックします。
int	**wc_ed25519_size**(ed25519_key * key) この関数は、Ed25519 - 32 バイトのサイズを返します。
int	**wc_ed25519_priv_size**(ed25519_key * key) この関数は、秘密鍵サイズ（secret + public）をバイト単位で返します。
int	**wc_ed25519_pub_size**(ed25519_key * key) この関数は圧縮鍵サイズをバイト単位で返します（公開鍵）。
int	**wc_ed25519_sig_size**(ed25519_key * key) この関数は、ED25519 シグネチャのサイズ（バイト数 64）を返します。

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
)
```

この関数は Ed25519 秘密鍵から Ed25519 公開鍵を生成します。公開鍵をバッファ pubkey に出力します。この関数の呼び出しに先立ち、ed25519_key 構造体には Ed25519 秘密鍵がインポートされている必要があ

ります。

Parameters:

- **key** Ed25519 秘密鍵がインポートされている ed25519_key 構造体へのポインタ。
- **pubKey** 公開鍵を出力するバッファへのポインタ。
- **pubKeySz** バッファのサイズ。常に ED25519_PUB_KEY_SIZE(32) でなければなりません。

See:

- wc_ed25519_init
- wc_ed25519_import_private_only
- wc_ed25519_make_key

Return:

- 0 公開鍵の作成に成功したときに返されます。
- BAD_FUNC_ARG 引数 key または pubKey が NULL の場合、または指定された鍵サイズが 32 バイトではない場合（ED25519 に 32 バイトのキーがあります）。
- ECC_PRIV_KEY_E ed25519_key 構造体に Ed25519 秘密鍵がインポートされていない場合に返されます。
- MEMORY_E 関数の実行中にメモリを割り当てエラーがある場合に返されます。

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
)
```

この関数は新しい ed25519_key 構造体を生成し、それを引数 key のバッファに格納します。

Parameters:

- **rng** RNG キーを生成する初期化された RNG オブジェクトへのポインタ。
- **keysize** key の長さ。ED25519 の場合は常に 32 になります。

See: wc_ed25519_init

Return:

- 0 ed25519_key 構造体を正常に生成すると返されます。
- BAD_FUNC_ARG RNG または KEY が NULL に評価された場合、または指定された keysize が 32 バイトではない場合 (Ed25519 鍵には常に 32 バイトを指定する必要があります)。

- MEMORY_E 関数の実行中にメモリ割り当てエラーが発生した場合に返されます。

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
)
```

この関数は、ed25519_key 構造体を使用してメッセージに署名します。

Parameters:

- **in** 署名するメッセージを含むバッファへのポインタ。
- **inlen** 署名するメッセージのサイズ
- **out** 生成された署名を格納するためのバッファ。
- **outlen** 出力バッファの最大長。メッセージ署名の生成に成功したときに、書き込まれたバイト数を保持します。
- **key** 署名を生成するために使用する秘密鍵を保持している ed25519_key 構造体へのポインタ。

See:

- wc_ed25519ctx_sign_msg
- wc_ed25519ph_sign_hash
- wc_ed25519ph_sign_msg
- wc_ed25519_verify_msg

Return:

- 0 メッセージの署名を正常に生成すると返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL に評価された場合、または出力バッファが小さすぎて生成された署名を保存する場合は返されます。
- MEMORY_E 関数の実行中にメモリ割り当てエラーが発生した場合に返されます。

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
)
```

この関数は、ed25519_key 構造体を使用してメッセージに署名します。コンテキストは署名されるデータの一部です。

Parameters:

- **in** 署名するメッセージを含むバッファへのポインタ。
- **inlen** 署名するメッセージのサイズ
- **out** 生成された署名を格納するためのバッファ。
- **outlen** 出力バッファの最大長。メッセージ署名の生成に成功したときに、書き込まれたバイトを保存します。
- **key** 署名を生成するために使用する秘密鍵を保持している ed25519_key 構造体へのポインタ。
- **context** メッセージが署名されているコンテキストを含むバッファへのポインタ。
- **contextLen** コンテキストバッファのサイズ

See:

- wc_ed25519_sign_msg
- wc_ed25519ph_sign_hash
- wc_ed25519ph_sign_msg
- wc_ed25519_verify_msg

Return:

- 0 メッセージの署名を正常に生成すると返されます。
- BAD_FUNC_ARG 返された入力パラメータは NULL に評価されます。出力バッファが小さすぎて生成された署名を保存するには小さすぎます。
- MEMORY_E 関数の実行中にメモリ割り当てエラーが発生した場合に返されます。

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
)
```

この関数は、ed25519_key 構造体を使用してメッセージダイジェストに署名します。コンテキストは署名されるデータの一部として含まれています。署名計算の前にメッセージは事前にハッシュされています。メッセージダイジェストを作成するために使用されるハッシュアルゴリズムは Shake-256 でなければなりません。

Parameters:

- **hash** 署名するメッセージのハッシュを含むバッファへのポインタ。
- **hashLen** 署名するメッセージのハッシュのサイズ
- **out** 生成された署名を格納するためのバッファ。
- **outlen** 出力バッファの最大長。メッセージ署名の生成に成功したときに、書き込まれたバイトを保存します。
- **key** 署名を生成するのに使用する秘密鍵を含んだ ed25519_key 構造体へのポインタ。
- **context** メッセージが署名されているコンテキストを含むバッファへのポインタ。
- **contextLen** コンテキストバッファのサイズ

See:

- wc_ed25519_sign_msg
- wc_ed25519ctx_sign_msg
- wc_ed25519ph_sign_msg
- wc_ed25519_verify_msg

Return:

- 0 メッセージダイジェストの署名を正常に生成すると返されます。
- BAD_FUNC_ARG 返された入力パラメータは NULL に評価されます。出力バッファが小さすぎて生成された署名を保存するには小さすぎます。
- MEMORY_E 関数の実行中にメモリ割り当てエラーが発生した場合に返されます。

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
)
```

この関数は、ed25519_key 構造体を使用して認証を保証するメッセージに署名します。コンテキストは署名されたデータの一部として含まれています。署名計算の前にメッセージは事前にハッシュされています。

Parameters:

- **in** 署名するメッセージを含むバッファへのポインタ。
- **inlen** 署名するメッセージのインレル長。
- **out** 生成された署名を格納するためのバッファ。
- **outlen** 出力バッファの最大長。メッセージ署名の生成に成功したときに、書き込まれたバイトを保存します。
- **key** 署名を生成するプライベート ed25519_key 構造体へのポインタ。
- **context** メッセージが署名されているコンテキストを含むバッファへのポインタ。
- **contextLen** コンテキストバッファのサイズ

See:

- wc_ed25519_sign_msg
- wc_ed25519ctx_sign_msg
- wc_ed25519ph_sign_hash
- wc_ed25519_verify_msg

Return:

- 0 メッセージの署名を正常に生成すると返されます。
- BAD_FUNC_ARG 返された入力パラメータは NULL に評価されます。出力バッファが小さすぎて生成された署名を保存するには小さすぎます。
- MEMORY_E 関数の実行中にメモリを割り当てエラーが発生した場合に返されます。

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
)
```

この関数はメッセージの Ed25519 署名を検証します。ret を介して答えを返し、有効な署名の場合は 1、無効な署名の場合には 0 を返します。

Parameters:

- **sig** 検証するシグネチャを含むバッファへのポインタ。
- **siglen** 検証するシグネチャのサイズ
- **msg** メッセージを含むバッファへのポインタ
- **msgLen** 検証するメッセージのサイズ
- **ret** 検証の結果を格納する変数へのポインタ。1 はメッセージが正常に検証されたことを示します。
- **key** 署名を検証するための Ed25519 公開鍵へのポインタ。

See:

- wc_ed25519ctx_verify_msg
- wc_ed25519ph_verify_hash
- wc_ed25519ph_verify_msg
- wc_ed25519_sign_msg

Return:

- 0 署名検証と認証を正常に実行したときに返されます。
- BAD_FUNC_ARG いずれかの入力パラメータが NULL に評価された場合、または SIGLEN が署名の実際の長さと一致しない場合に返されます。
- SIG_VERIFY_E 検証が完了した場合は返されますが、生成された署名は提供された署名と一致しません。

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
)
```

この関数はメッセージの Ed25519 署名を検証します。コンテキストは署名されたデータの一部として含まれています。答えは変数 ret を介して返され、署名が有効ならば 1、無効ならば 0 を返します。

Parameters:

- **sig** 検証するシグネチャを含むバッファへのポインタ。
- **siglen** 検証するシグネチャのサイズ
- **msg** メッセージを含むバッファへのポインタ
- **msgLen** 検証するメッセージのサイズ
- **ret** 検証の結果を格納する変数へのポインタ。1 はメッセージが正常に検証されたことを示します。
- **key** 署名を検証するための Ed25519 公開鍵へのポインタ。
- **context** メッセージが署名されているコンテキストを含むバッファへのポインタ。
- **contextLen** コンテキストバッファのサイズ

See:

- wc_ed25519_verify_msg
- wc_ed25519ph_verify_hash
- wc_ed25519ph_verify_msg
- wc_ed25519_sign_msg

Return:

- 0 署名検証と認証を正常に実行したときに返されます。
- BAD_FUNC_ARG いずれかの入力パラメータが NULL に評価された場合、または SIGLEN が署名の実際の長さと一致しない場合に返されます。
- SIG_VERIFY_E 検証が完了した場合は返されますが、生成された署名は提供された署名と一致しません。

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
)
```

この関数は、メッセージのダイジェストの Ed25519 署名を検証します。引数 hash は、署名計算前のプリハッシュメッセージです。メッセージダイジェストを作成するために使用されるハッシュアルゴリズムは SHA-512 でなければなりません。答えは変数 ret を介して返され、署名が有効ならば 1、無効ならば 0 を返します。

Parameters:

- **sig** 検証するシグネチャを含むバッファへのポインタ。
- **siglen** 検証するシグネチャのサイズ
- **msg** メッセージを含むバッファへのポインタ
- **msgLen** 検証するメッセージのサイズ
- **ret** 検証の結果を格納する変数へのポインタ。1 はメッセージが正常に検証されたことを示します。
- **key** 署名を検証するための Ed25519 公開鍵へのポインタ。
- **context** メッセージが署名されたコンテキストを含むバッファへのポインタ。
- **contextLen** コンテキストのサイズ

See:

- wc_ed25519_verify_msg
- wc_ed25519ctx_verify_msg
- wc_ed25519ph_verify_msg
- wc_ed25519_sign_msg

Return:

- 0 署名検証と認証を正常に実行したときに返されます。
- BAD_FUNC_ARG いずれかの入力パラメータが NULL に評価された場合、または SIGLEN が署名の実際の長さと一致しない場合に返されます。
- SIG_VERIFY_E 検証が完了した場合は返されますが、生成された署名は提供された署名と一致しません。

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
)
```

この関数は、メッセージのダイジェストの Ed25519 署名を検証します。引数 context は検証すべきデータの一部として含まれています。検証前にメッセージがプリハッシュされています。答えは変数 res を介して返され、署名が有効ならば 1、無効ならば 0 を返します。

Parameters:

- **sig** 検証するシグネチャを含むバッファへのポインタ。
- **siglen** 検証するシグネチャのサイズ
- **msg** メッセージを含むバッファへのポインタ
- **msgLen** 検証するメッセージのサイズ
- **ret** 検証の結果を格納する変数へのポインタ。1 はメッセージが正常に検証されたことを示します。
- **key** 署名を検証するための Ed25519 公開鍵へのポインタ。
- **context** メッセージが署名されたコンテキストを含むバッファへのポインタ。
- **contextLen** コンテキストのサイズ

See:

- wc_ed25519_verify_msg
- wc_ed25519ph_verify_hash
- wc_ed25519ph_verify_msg
- wc_ed25519_sign_msg

Return:

- 0 署名検証と認証を正常に実行したときに返されます。
- BAD_FUNC_ARG いずれかの入力パラメータが NULL に評価された場合、または SIGLEN が署名の実際の長さと一致しない場合に返されます。
- SIG_VERIFY_E 検証が完了した場合は返されますが、生成された署名は提供された署名と一致しません。

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
)
```

この関数は、後のメッセージ検証で使用のために ed25519_key 構造体を初期化します。

Parameters:

- **key** ed25519_key 構造体へのポインタ

See:

- wc_ed25519_make_key
- wc_ed25519_free

Return:

- 0 ed25519_key 構造体の初期化に成功したときに返されます。
- BAD_FUNC_ARG 引数 key が NULL の場合に返されます。

Example

```
ed25519_key key;
wc_ed25519_init(&key);
```

C.23.2.12 function wc_ed25519_free

```
void wc_ed25519_free(
    ed25519_key * key
)
```

この関数は、使用済みの ed25519_key 構造体を解放します。

Parameters:

- **key** ed25519_key 構造体へのポインタ

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
)
```

この関数はバッファから ed25519 公開鍵を ed25519_key 構造体へインポートします。圧縮あるいは非圧縮の両方の形式の鍵を扱います。

Parameters:

- **in** 公開鍵を含んだバッファへのポインタ
- **inLen** 公開鍵を含んだバッファのサイズ
- **key** ed25519_key 構造体へのポインタ

See:

- wc_ed25519_import_public_ex
- wc_ed25519_import_private_key
- wc_ed25519_import_private_key_ex
- wc_ed25519_export_public

Return:

- 0 ed25519 公開鍵のインポートに成功した場合に返されます。
- BAD_FUNC_ARG in または key が null に評価された場合、または inlen が ED25519 鍵のサイズよりも小さい場合に返されます。

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
)
```

この関数はバッファから ed25519 公開鍵を ed25519_key 構造体へインポートします。圧縮あるいは非圧縮の両方の形式の鍵を扱います。秘密鍵が既にインポートされている場合で、trusted 引数が 1 以外の場合は両鍵が対応しているかをチェックします。

Parameters:

- **in** 公開鍵を含んだバッファへのポインタ
- **inLen** 公開鍵を含んだバッファのサイズ
- **key** ed25519_key 構造体へのポインタ
- **trusted** 公開鍵が信頼おけるか否かを示すフラグ

See:

- wc_ed25519_import_public
- wc_ed25519_import_private_key
- wc_ed25519_import_private_key_ex
- wc_ed25519_export_public

Return:

- 0 ed25519 公開鍵のインポートに成功した場合に返されます。
- BAD_FUNC_ARG Returned 引数 in あるいは key が NULL の場合, あるいは引数 inLen が Ed25519 鍵のサイズより小さい場合に返されます。

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
)
```

この関数は、ed25519 秘密鍵のみをバッファからインポートします。

Parameters:

- **priv** 秘密鍵を含むバッファへのポインタ。
- **privSz** 秘密鍵を含むバッファのサイズ

See:

- wc_ed25519_import_public
- wc_ed25519_import_private_key
- wc_ed25519_export_private_only

Return:

- 0 Ed25519 秘密鍵のインポートに成功した際に返されます。
- BAD_FUNC_ARG priv または key が NULL に評価された場合、または privSz が ED25519_KEY_SIZE と異なる場合に返されます。

Example

```
int ret;
byte priv[] = { initialize with 32 byte private key };

ed25519_key key;
wc_ed25519_init_key(&key);
ret = wc_ed25519_import_private_key(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
)
```

この関数は、Ed25519 公開鍵/秘密鍵をそれぞれ含む一対のバッファから Ed25519 鍵ペアをインポートします。この関数は圧縮と非圧縮の両方の鍵を処理します。

Parameters:

- **priv** 秘密鍵を含むバッファへのポインタ。
- **privSz** 秘密鍵バッファのサイズ
- **pub** 公開鍵を含むバッファへのポインタ。
- **pubSz** 公開鍵バッファのサイズ

See:

- wc_ed25519_import_public
- wc_ed25519_import_private_only
- wc_ed25519_export_private

Return:

- 0 Ed25519_KEY のインポートに成功しました。
- BAD_FUNC_ARG priv または key が NULL に評価された場合、privSz が ED25519_KEY_SIZE と異なるあるいは ED25519_PRV_KEY_SIZE とも異なる場合、pubSz が ED25519_PUB_KEY_SIZE よりも小さい場合に返されます。

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
)
```

この関数は一対のバッファから Ed25519 公開鍵/秘密鍵ペアをインポートします。この関数は圧縮キーと非圧縮キーの両方を処理します。公開鍵は trusted 引数により信頼されていないとされた場合には秘密鍵に対して検証されます。

Parameters:

- **priv** 秘密鍵を保持するバッファへのポインタ
- **privSz** 秘密鍵バッファのサイズ
- **pub** 公開鍵を保持するバッファへのポインタ
- **pubSz** 公開鍵バッファのサイズ
- **key** インポートされた公開鍵/秘密鍵を保持する ed25519_key オブジェクトへのポインター
- **trusted** 公開鍵が信頼できるか否かを指定するフラグ

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 ed25519_key のインポートに成功しました。
- BAD_FUNC_ARG Returned if priv あるいは key が NULL に評価された場合、privSz が ED25519_KEY_SIZE とも ED25519_PRV_KEY_SIZE とも異なる場合、pubSz が ED25519_PUB_KEY_SIZE より小さい場合に返されます。

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
)
```

この関数は、ed25519_key 構造体から公開鍵をエクスポートします。公開鍵をバッファ out に格納し、outLen にこのバッファに書き込まれたバイトを設定します。

Parameters:

- **key** 公開鍵をエクスポートするための ed25519_key 構造体へのポインタ。
- **out** 公開鍵を保存するバッファへのポインタ。

- **outLen** 公開鍵を出力する先のバッファサイズを格納する word32 型変数へのポインタ。入力の際はバッファサイズを格納して渡し、出力の際はエクスポートした公開鍵のサイズを格納します。

See:

- wc_ed25519_import_public
- wc_ed25519_export_private_only

Return:

- 0 公開鍵のエクスポートに成功したら返されます。
- BAD_FUNC_ARG いずれかの入力値が NULL に評価された場合に返されます。
- BUFFER_E 提供されたバッファーが公開鍵を保存するのに十分な大きさでない場合に返されます。このエラーを返すと、outlen に必要なサイズを設定します。

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
)
```

この関数は、ed25519_key 構造体からの秘密鍵のみをエクスポートします。秘密鍵をバッファアウトに格納し、outlen にこのバッファに書き込まれたバイトを設定します。

Parameters:

- **key** 秘密鍵をエクスポートするための ed25519_key 構造体へのポインタ。
- **out** 秘密鍵を保存するバッファへのポインタ。
- **outLen** 秘密鍵を出力する先のバッファサイズを格納する word32 型変数へのポインタ。入力の際はバッファサイズを格納して渡し、出力の際はエクスポートした秘密鍵のサイズを格納します。

See:

- wc_ed25519_export_public
- wc_ed25519_import_private_key

Return:

- 0 秘密鍵のエクスポートに成功したら返されます。
- BAD_FUNC_ARG いずれかの入力値が NULL に評価された場合に返されます。
- BUFFER_E 提供されたバッファーが秘密鍵を保存するのに十分な大きさでない場合に返されます。

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
)
```

この関数は、ed25519_key 構造体から鍵ペアをエクスポートします。鍵ペアをバッファ out に格納し、ounteren でこのバッファに書き込まれたバイトを設定します。

Parameters:

- **鍵ペアをエクスポートするための ed25519_key 構造体へのポインタ。**
- **鍵ペアを保存するバッファへのポインタ。**
- **outLen** 鍵ペアを出力する先のバッファサイズを格納する word32 型変数へのポインタ。入力の際はバッファサイズを格納して渡し、出力の際はエクスポートした鍵ペアのサイズを格納します。

See:

- wc_ed25519_import_private_key
- wc_ed25519_export_private_only

Return:

- 0 鍵ペアのエクスポートに成功したら返されます。
- BAD_FUNC_ARG いずれかの入力値が NULL に評価された場合に返されます。
- BUFFER_E 提供されているバッファーが鍵ペアを保存するのに十分な大きさでない場合に返されます。

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
)
```

この関数は、ed25519_key 構造体から秘密鍵と公開鍵を別々にエクスポートします。秘密鍵をバッファ priv に格納し、priovSz にこのバッファに書き込んだバイト数を設定します。公開鍵をバッファ pub に格納し、pubSz にこのバッファに書き込んだバイト数を設定します。

Parameters:

- **key** 鍵ペアをエクスポートするための ed25519_key 構造体へのポインタ。
- **priv** 秘密鍵を出力するバッファへのポインタ。
- **privSz** 秘密鍵を出力する先のバッファのサイズを保持する word32 型変数へのポインタ。秘密鍵のエクスポート後には書き込まれたバイト数がセットされます。
- **pub** パブリックキーを出力するバッファへのポインタ
- **pubSz** 公開鍵を出力する先のバッファのサイズを保持する word32 型変数へのポインタ。公開鍵のエクスポート後には書き込まれたバイト数がセットされます。

See:

- wc_ed25519_export_private
- wc_ed25519_export_public

Return:

- 0 鍵ペアのエクスポートに成功したら返されます。
- BAD_FUNC_ARG いずれかの入力値が NULL に評価された場合に返されます。
- BUFFER_E 提供されているバッファが鍵ペアを保存するのに十分な大きさでない場合に返されます。

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
)
```

この関数は、ed25519_key 構造体の公開鍵をチェックします。

Parameters:

- **key** 公開鍵と秘密鍵の両方を保持している ed25519_key 構造体へのポインタ

See: wc_ed25519_import_private_key

Return:

- 0 プライベートキーと公開鍵が一致した場合に返されます。
- BAD_FUNC_ARG 与えられた鍵が NULL の場合に返されます。
- PUBLIC_KEY_E 公開鍵が参照できないか無効の場合に返されます。

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(priv, sizeof(priv), pub, sizeof(pub), &key);
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
)
```

この関数は、Ed25519 - 32 バイトのサイズを返します。

Parameters:

- **key** ed25519_key 構造体へのポインタ

See: wc_ed25519_make_key

Return:

- ED25519_KEY_SIZE 有効な秘密鍵のサイズ（32 バイト）。
- BAD_FUNC_ARG 与えられた引数 key が NULL の場合に返されます。

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
)
```

この関数は、秘密鍵サイズ（secret + public）をバイト単位で返します。

Parameters:

- **key** ed25519_key 構造体へのポインタ

See: wc_ed25519_pub_size

Return:

- ED25519_PRV_KEY_SIZE 秘密鍵のサイズ（64 バイト）。
- BAD_FUNC_ARG key 引数が null の場合に返されます。

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
)
```

この関数は圧縮鍵サイズをバイト単位で返します（公開鍵）。

Parameters:

- **key** ed25519_key 構造体へのポインタ

See: wc_ed25519_priv_size

Return:

- ED25519_PUB_KEY_SIZE 圧縮公開鍵のサイズ（32 バイト）。
- BAD_FUNC_ARG key 引数が null の場合は返します。

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
)
```

この関数は、ED25519 シグネチャのサイズ（バイト数 64）を返します。

Parameters:

- **key** ed25519_key 構造体へのポインタ

See: wc_ed25519_sign_msg

Return:

- ED25519_SIG_SIZE ED25519 シグネチャ（64 バイト）のサイズ。
- BAD_FUNC_ARG key 引数が null の場合は返します。

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-ja/ed448.h

C.24.1 Functions

	Name
int	**wc_ed448_make_public**(ed448_key * key, unsigned char * pubKey, word32 pubKeySz) この関数は、秘密鍵から ED448 公開鍵を生成します。公開鍵をバッファ Pubkey に格納し、Pubkeysz でこのバッファに書き込まれたバイトを設定します。
int	**wc_ed448_make_key**(WC_RNG * rng, int keysize, ed448_key * key) この関数は新しい ED448 キーを生成し、それをキーに格納します。

	Name
int	**wc_ed448_sign_msg**(const byte * in, word32 inlen, byte * out, word32 * outlen, ed448_key * key) この関数は、ED448_Key オブジェクトを使用したメッセージに正解を保証します。
int	**wc_ed448ph_sign_hash**(const byte * hash, word32 hashLen, byte * out, word32 * outLen, ed448_key * key, const byte * context, byte contextLen) この関数は、Ed448_Key オブジェクトを使用してメッセージダイジェストに署名して信頼性を保証します。コンテキストは署名されたデータの一部として含まれています。ハッシュは、署名計算前のプリハッシュメッセージです。メッセージダイジェストを作成するために使用されるハッシュアルゴリズムは Shake-256 でなければなりません。
int	**wc_ed448ph_sign_msg**(const byte * in, word32 inLen, byte * out, word32 * outLen, ed448_key * key, const byte * context, byte contextLen) この関数は、ED448_Key オブジェクトを使用したメッセージに正解を保証します。コンテキストは署名されたデータの一部として含まれています。署名計算の前にメッセージは事前にハッシュされています。
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) この関数は、メッセージの ED448 署名を確認して信頼性を確保します。文脈はデータ検証済みの一部として含まれています。答えは RES を介して返され、有効な署名に対応する 1、無効な署名に対応する 0 を返します。
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) この関数は、メッセージのダイジェストの ED448 シグネチャを検証して、信頼性を確保します。文脈はデータ検証済みの一部として含まれています。ハッシュは、署名計算前のプリハッシュメッセージです。メッセージダイジェストを作成するために使用されるハッシュアルゴリズムは Shake-256 でなければなりません。答えは RES を介して返され、有効な署名に対応する 1、無効な署名に対応する 0 を返します。
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) この関数は、メッセージの ED448 署名を確認して信頼性を確保します。文脈はデータ検証済みの一部として含まれています。検証前にメッセージがプリハッシュされています。答えは RES を介して返され、有効な署名に対応する 1、無効な署名に対応する 0 を返します。

	Name
int	**wc_ed448_init**(ed448_key * key) この関数は、メッセージ検証で将来の使用のために ED448_Key オブジェクトを初期化します。
void	**wc_ed448_free**(ed448_key * key) この関数は、それが使用された後に ED448 オブジェクトを解放します。*Example*
int	**wc_ed448_import_public**(const byte * in, word32 inLen, ed448_key * key) この関数は、公開鍵を含むバッファから Public ED448_Key ペアをインポートします。この関数は圧縮キーと非圧縮キーの両方を処理します。
int	**wc_ed448_import_private_only**(const byte * priv, word32 privSz, ed448_key * key) この関数は、ed448 秘密鍵をバッファからのみインポートします。
int	**wc_ed448_import_private_key**(const byte * priv, word32 privSz, const byte * pub, word32 pubSz, ed448_key * key) この関数は、一対のバッファからパブリック/プライベート ED448 キーペアをインポートします。この関数は圧縮キーと非圧縮キーの両方を処理します。
int	**wc_ed448_export_public**(ed448_key * key, byte * out, word32 * outLen) この関数は、ED448_Key 構造体から秘密鍵をエクスポートします。公開鍵をバッファアウトに格納し、ounteren でこのバッファに書き込まれたバイトを設定します。
int	**wc_ed448_export_private_only**(ed448_key * key, byte * out, word32 * outLen) この関数は、ED448_Key 構造体からの秘密鍵のみをエクスポートします。秘密鍵をバッファアウトに格納し、outlen にこのバッファに書き込まれたバイトを設定します。
int	**wc_ed448_export_private**(ed448_key * key, byte * out, word32 * outLen) この関数は、ED448_Key 構造体からキーペアをエクスポートします。キーペアをバッファ OUT に格納し、ounteren でこのバッファに書き込まれたバイトを設定します。
int	**wc_ed448_export_key**(ed448_key * key, byte * priv, word32 * privSz, byte * pub, word32 * pubSz) この関数は、ED448_Key 構造体とは別にプライベートキーと公開鍵をエクスポートします。秘密鍵をバッファー Priv に格納し、PRIVSZ でこのバッファに書き込まれたバイトを設定します。公開鍵をバッファ PUB に格納し、Pubsz でこのバッファに書き込まれたバイトを設定します。
int	**wc_ed448_check_key**(ed448_key * key) この関数は、ed448_key 構造体の公開鍵をチェックします。
int	**wc_ed448_size**(ed448_key * key) この関数は、ED448 秘密鍵のサイズ - 57 バイトを返します。

	Name
int	**wc_ed448_priv_size**(ed448_key * key) この関数は、秘密鍵サイズ（secret + public）をバイト単位で返します。
int	**wc_ed448_pub_size**(ed448_key * key) この関数は圧縮鍵サイズをバイト単位で返します（公開鍵）。
int	**wc_ed448_sig_size**(ed448_key * key) この関数は、ED448 シグネチャのサイズ（バイト数 114）を返します。

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
)
```

この関数は、秘密鍵から ED448 公開鍵を生成します。公開鍵をバッファ Pubkey に格納し、Pubkeysz でこのバッファに書き込まれたバイトを設定します。

Parameters:

- **キーを生成する ED448_Key へのキーポインタ。**
- **公開鍵を保存するバッファへのポインタ。** *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
}
```

See:

- wc_ed448_init
- wc_ed448_import_private_only
- wc_ed448_make_key

Return:

- 0 公開鍵の作成に成功したときに返されます。
- BAD_FUNC_ARG IFI キーまたは PubKey が NULL に評価された場合、または指定されたキーサイズが 57 バイトではない場合（ED448 には 57 バイトのキーがあります）。
- MEMORY_E 関数の実行中にメモリを割り当てるエラーがある場合に返されます。

C.24.2.2 function wc_ed448_make_key

```
int wc_ed448_make_key(
    WC_RNG * rng,
    int keysize,
    ed448_key * key
)
```

この関数は新しい ED448 キーを生成し、それをキーに格納します。

Parameters:

- **RNG キーを生成する初期化された RNG オブジェクトへのポインタ。**
- **keysize** key の長さを生成します。ED448 の場合は常に 57 になります。*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
}
```

See: wc_ed448_init

Return:

- 0 ED448_Key を正常に作成したときに返されます。
- BAD_FUNC_ARG RNG または Key が NULL に評価された場合、または指定されたキーサイズが 57 バイトではない場合（ED448 には 57 バイトのキーがあります）。
- MEMORY_E 関数の実行中にメモリを割り当てるエラーがある場合に返されます。

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
)
```

この関数は、ED448_Key オブジェクトを使用したメッセージに正解を保証します。

Parameters:

- **署名するメッセージを含むバッファへのポインタ。**
- **署名するメッセージのインレル長。**
- **生成された署名を格納するためのバッファー。**
- **出力バッファの最大長の範囲内。メッセージ署名の生成に成功したときに、書き込まれたバイトを保存します。** *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
}
```

See:

- wc_ed448ph_sign_hash
- wc_ed448ph_sign_msg
- wc_ed448_verify_msg

Return:

- 0 メッセージの署名を正常に生成すると返されます。
- BAD_FUNC_ARG 入力パラメータのいずれかが NULL に評価された場合、または出力バッファが小さすぎて生成された署名を保存する場合は返されます。
- MEMORY_E 関数の実行中にメモリを割り当てるエラーがある場合に返されます。

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
)
```

この関数は、Ed448_Key オブジェクトを使用してメッセージダイジェストに署名して信頼性を保証します。コンテキストは署名されたデータの一部として含まれています。ハッシュは、署名計算前のプリハッシュメッセージです。メッセージダイジェストを作成するために使用されるハッシュアルゴリズムは Shake-256 でなければなりません。

Parameters:

- **サインへのメッセージのハッシュを含むバッファへのハッシュポインタ。**
- **サインへのメッセージのハッシュのハッシュの長さ。**
- **生成された署名を格納するためのバッファー。**
- **出力バッファの最大長の範囲内。メッセージ署名の生成に成功したときに、書き込まれたバイトを保存します。**
- **署名を生成するためのプライベート ED448_Key へのキーポインタ。**
- **メッセージが署名されているコンテキストを含むバッファへのコンテキストポインタ。** *Example*

```
ed448_key key;
WC_RNG rng;
int ret, sigSz;

byte sig[114]; // will hold generated signature
sigSz = sizeof(sig);
```

```
byte hash[] = { initialize with SHAKE-256 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
}
```

See:

- wc_ed448_sign_msg
- wc_ed448ph_sign_msg
- wc_ed448ph_verify_hash

Return:

- 0 メッセージダイジェストの署名を正常に生成すると返されます。
- BAD_FUNC_ARG 返された入力パラメータは NULL に評価されます。出力バッファが小さすぎて生成された署名を保存するには小さすぎます。
- MEMORY_E 関数の実行中にメモリを割り当てるエラーがある場合に返されます。

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
)
```

この関数は、ED448_Key オブジェクトを使用したメッセージに正解を保証します。コンテキストは署名されたデータの一部として含まれています。署名計算の前にメッセージは事前にハッシュされています。

Parameters:

- **署名するメッセージを含むバッファへのポインタ。**
- **署名するメッセージのインレル長。**
- **生成された署名を格納するためのバッファー。**
- **出力バッファの最大長の範囲内。メッセージ署名の生成に成功したときに、書き込まれたバイトを保存します。**
- **署名を生成するためのプライベート ED448_Key へのキーポインタ。**
- **メッセージが署名されているコンテキストを含むバッファへのコンテキストポインタ。** *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
}
```

See:

- wc_ed448_sign_msg
- wc_ed448ph_sign_hash
- wc_ed448ph_verify_msg

Return:

- 0 メッセージの署名を正常に生成すると返されます。
- BAD_FUNC_ARG 返された入力パラメータは NULL に評価されます。出力バッファが小さすぎて生成された署名を保存するには小さすぎます。
- MEMORY_E 関数の実行中にメモリを割り当てるエラーがある場合に返されます。

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
)
```

この関数は、メッセージの ED448 署名を確認して信頼性を確保します。文脈はデータ検証済みの一部として含まれています。答えは RES を介して返され、有効な署名に対応する 1、無効な署名に対応する 0 を返します。

Parameters:

- **検証するシグネチャを含むバッファへの SIG ポインタ。**
- **検証するシグネチャのシグレンの長さ。**
- **メッセージを含むバッファへの MSG ポインタを確認する。**
- **検証するメッセージの MSGlen 長。**
- **署名を検証するためのパブリック ED448 キーへのキーポインタ。**
- **メッセージが署名されたコンテキストを含むバッファへのコンテキストポインタ。** *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
}
```

See:

- wc_ed448ph_verify_hash
- wc_ed448ph_verify_msg
- wc_ed448_sign_msg

Return:

- 0 署名検証と認証を正常に実行したときに返されます。
- BAD_FUNC_ARG いずれかの入力パラメータが NULL に評価された場合、または SIGLEN が署名の実際の長さと一致しない場合に返されます。
- SIG_VERIFY_E 検証が完了した場合は返されますが、生成された署名は提供された署名と一致しません。

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
)
```

この関数は、メッセージのダイジェストの ED448 シグネチャを検証して、信頼性を確保します。文脈はデータ検証済みの一部として含まれています。ハッシュは、署名計算前のプリハッシュメッセージです。メッセージダイジェストを作成するために使用されるハッシュアルゴリズムは Shake-256 でなければなりません。答えは RES を介して返され、有効な署名に対応する 1、無効な署名に対応する 0 を返します。

Parameters:

- **検証するシグネチャを含むバッファへの SIG ポインタ。**
- **検証するシグネチャのシグレンの長さ。**
- **検証するメッセージのハッシュを含むバッファへのハッシュポインタ。**
- **検証するハッシュのハッシュレン長。**
- **署名を検証するためのパブリック ED448 キーへのキーポインタ。**
- **メッセージが署名されたコンテキストを含むバッファへのコンテキストポインタ。** *Example*

```
ed448_key key;
int ret, verified = 0;

byte sig[] { initialize with received signature };
byte hash[] = { initialize with SHAKE-256 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
}
```

See:

- wc_ed448_verify_msg
- wc_ed448ph_verify_msg
- wc_ed448ph_sign_hash

Return:

- 0 署名検証と認証を正常に実行したときに返されます。
- BAD_FUNC_ARG いずれかの入力パラメータが NULL に評価された場合、または SIGLEN が署名の実際の長さと一致しない場合に返されます。
- SIG_VERIFY_E 検証が完了した場合は返されますが、生成された署名は提供された署名と一致しません。

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
)
```

この関数は、メッセージの ED448 署名を確認して信頼性を確保します。文脈はデータ検証済みの一部として含まれています。検証前にメッセージがプリハッシュされています。答えは RES を介して返され、有効な署名に対応する 1、無効な署名に対応する 0 を返します。

Parameters:

- **検証するシグネチャを含むバッファへの SIG ポインタ。**
- **検証するシグネチャのシグレンの長さ。**
- **メッセージを含むバッファへの MSG ポインタを確認する。**
- **検証するメッセージの MSGlen 長。**
- **署名を検証するためのパブリック ED448 キーへのキーポインタ。**
- **メッセージが署名されたコンテキストを含むバッファへのコンテキストポインタ。** *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
}
```

See:

- wc_ed448_verify_msg
- wc_ed448ph_verify_hash
- wc_ed448ph_sign_msg

Return:

- 0 署名検証と認証を正常に実行したときに返されます。
- BAD_FUNC_ARG いずれかの入力パラメータが NULL に評価された場合、または SIGLEN が署名の実際の長さと一致しない場合に返されます。
- SIG_VERIFY_E 検証が完了した場合は返されますが、生成された署名は提供された署名と一致しません。

C.24.2.9 function wc_ed448_init

```
int wc_ed448_init(
    ed448_key * key
)
```

この関数は、メッセージ検証で将来の使用のために ED448_Key オブジェクトを初期化します。

See:

- wc_ed448_make_key
- wc_ed448_free

Return:

- 0 ED448_Key オブジェクトの初期化に成功したら返されます。
- BAD_FUNC_ARG キーが NULL の場合は返されます。*Example*

```
ed448_key key;
wc_ed448_init(&key);
```

C.24.2.10 function wc_ed448_free

```
void wc_ed448_free(
    ed448_key * key
)
```

この関数は、それが使用された後に ED448 オブジェクトを解放します。*Example*

See: wc_ed448_init

```
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
)
```

この関数は、公開鍵を含むバッファから Public ED448_Key ペアをインポートします。この関数は圧縮キーと非圧縮キーの両方を処理します。

Parameters:

- **公開鍵を含むバッファへのポインタ。**
- **公開鍵を含むバッファのインレル長。** *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
}
```

See:

- wc_ed448_import_private_key
- wc_ed448_export_public

Return:

- 0 ED448_Key のインポートに成功しました。
- BAD_FUNC_ARG IN または KEY が NULL に評価されている場合、または INLEN が ED448 キーのサイズより小さい場合に返されます。

C.24.2.12 function wc_ed448_import_private_only

```
int wc_ed448_import_private_only(
    const byte * priv,
    word32 privSz,
    ed448_key * key
)
```

この関数は、ed448 秘密鍵をバッファからのみインポートします。

Parameters:

- **秘密鍵を含むバッファへの PRIV ポインタ。**
- **秘密鍵の Privsz 長さ。** *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
}
```

See:

- wc_ed448_import_public
- wc_ed448_import_private_key
- wc_ed448_export_private_only

Return:

- 0 ED448 秘密鍵のインポートに成功しました。

- BAD_FUNC_ARG IN または KEY が NULL に評価された場合、または PRIVSZ が ED448_KEY_SIZE よりも小さい場合に返されます。

C.24.2.13 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
)
```

この関数は、一対のバッファからパブリック/プライベート ED448 キーペアをインポートします。この関数は圧縮キーと非圧縮キーの両方を処理します。

Parameters:

- **秘密鍵を含むバッファへの PRIV ポインタ。**
- **秘密鍵の Privsz 長さ。**
- **公開鍵を含むバッファへの Pub ポインタ。**
- **公開鍵の Pubsz の長さ。** *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
}
```

See:

- wc_ed448_import_public
- wc_ed448_import_private_only
- wc_ed448_export_private

Return:

- 0 ED448 キーのインポートに成功しました。
- BAD_FUNC_ARG IN または KEY が NULL に評価された場合、または PROVSZ が ED448_KEY_SIZE または PUBSZ のいずれかが eD448_PUB_KEY_SIZE よりも小さい場合に返されます。

C.24.2.14 function wc_ed448_export_public

```
int wc_ed448_export_public(
    ed448_key * key,
    byte * out,
    word32 * outLen
)
```

この関数は、ED448_Key 構造体から秘密鍵をエクスポートします。公開鍵をバッファアウトに格納し、ounteren でこのバッファに書き込まれたバイトを設定します。

Parameters:

- **公開鍵をエクスポートする ED448_Key 構造体へのキーポインタ。**
- **公開鍵を保存するバッファへのポインタ。** *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
}
```

See:

- wc_ed448_import_public
- wc_ed448_export_private_only

Return:

- 0 公開鍵のエクスポートに成功したら返されます。
- BAD_FUNC_ARG いずれかの入力値が NULL に評価された場合に返されます。
- BUFFER_E 提供されたバッファーが秘密鍵を保存するのに十分な大きさでない場合に返されます。このエラーを返すと、outlen に必要なサイズを設定します。

C.24.2.15 function wc_ed448_export_private_only

```
int wc_ed448_export_private_only(
    ed448_key * key,
    byte * out,
    word32 * outLen
)
```

この関数は、ED448_Key 構造体からの秘密鍵のみをエクスポートします。秘密鍵をバッファアウトに格納し、outlen にこのバッファに書き込まれたバイトを設定します。

Parameters:

- **秘密鍵をエクスポートする ED448_Key 構造体へのキーポインタ。**
- **秘密鍵を保存するバッファへのポインタ。** *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
}
```

See:

- wc_ed448_export_public
- wc_ed448_import_private_key

Return:

- 0 秘密鍵のエクスポートに成功したら返されます。
- ECC_BAD_ARG_E いずれかの入力値が NULL に評価された場合に返されます。
- BUFFER_E 提供されたバッファーが秘密鍵を保存するのに十分な大きさでない場合に返されます。

C.24.2.16 function wc_ed448_export_private

```
int wc_ed448_export_private(
    ed448_key * key,
    byte * out,
    word32 * outLen
)
```

この関数は、ED448_Key 構造体からキーペアをエクスポートします。キーペアをバッファ OUT に格納し、ounteren でこのバッファに書き込まれたバイトを設定します。

Parameters:

- **キーペアをエクスポートするための ED448_Key 構造体へのキーポインタ。**
- **キーペアを保存するバッファへのポインタ。** *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
}
```

See:

- wc_ed448_import_private
- wc_ed448_export_private_only

Return:

- 0 キーペアのエクスポートに成功したら返されます。
- ECC_BAD_ARG_E いずれかの入力値が NULL に評価された場合に返されます。
- BUFFER_E 提供されているバッファーがキーペアを保存するのに十分な大きさでない場合に返されます。

C.24.2.17 function wc_ed448_export_key

```
int wc_ed448_export_key(
    ed448_key * key,
    byte * priv,
    word32 * privSz,
    byte * pub,
    word32 * pubSz
)
```

この関数は、ED448_Key 構造体とは別にプライベートキーと公開鍵をエクスポートします。秘密鍵をバッファー Priv に格納し、PRIVSZ でこのバッファに書き込まれたバイトを設定します。公開鍵をバッファ PUB に格納し、Pubsz でこのバッファに書き込まれたバイトを設定します。

Parameters:

- **キーペアをエクスポートするための ED448_Key 構造体へのキーポインタ。**
- **秘密鍵を保存するバッファへの PRIV ポインタ。**
- **PRIVSZ** PIVINSZ ポインタサイズが表示されているサイズを持つ Word32 オブジェクトへのポインタ。秘密鍵のエクスポート後に書き込まれたバイト数を設定します。
- **パブリックキーを保存するバッファへの Pub。** *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
}
```

See:

- wc_ed448_export_private
- wc_ed448_export_public

Return:

- 0 キーペアのエクスポートに成功したら返されます。
- ECC_BAD_ARG_E いずれかの入力値が NULL に評価された場合に返されます。
- BUFFER_E 提供されているバッファーがキーペアを保存するのに十分な大きさでない場合に返されます。

C.24.2.18 function wc_ed448_check_key

```
int wc_ed448_check_key(
    ed448_key * key
)
```

この関数は、ed448_key 構造体の公開鍵をチェックします。

See: wc_ed448_import_private_key

Return:

- 0 プライベートキーと公開鍵が一致した場合に返されます。
- BAD_FUNC_ARGS 与えられたキーが NULL の場合に返されます。*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(priv, sizeof(priv), pub, sizeof(pub), &key);
```

```
ret = wc_ed448_check_key(&key);
if (ret != 0) {
    // error checking key
}
```

C.24.2.19 function wc_ed448_size

```
int wc_ed448_size(
    ed448_key * key
)
```

この関数は、ED448 秘密鍵のサイズ - 57 バイトを返します。

See: wc_ed448_make_key

Return:

- ED448_KEY_SIZE 有効な秘密鍵のサイズ（57 バイト）。
- BAD_FUNC_ARGS 与えられたキーが NULL の場合に返されます。*Example*

```
int keySz;
ed448_key key;
// initialize key, make key
keySz = wc_ed448_size(&key);
if (keySz == 0) {
    // error determining key size
}
```

C.24.2.20 function wc_ed448_priv_size

```
int wc_ed448_priv_size(
    ed448_key * key
)
```

この関数は、秘密鍵サイズ（secret + public）をバイト単位で返します。

See: wc_ed448_pub_size

Return:

- ED448_PRV_KEY_SIZE 秘密鍵のサイズ（114 バイト）。
- BAD_FUNC_ARG key 引数が null の場合は返します。*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.24.2.21 function wc_ed448_pub_size

```
int wc_ed448_pub_size(
    ed448_key * key
)
```

この関数は圧縮鍵サイズをバイト単位で返します（公開鍵）。

See: wc_ed448_priv_size

Return:

- ED448_PUB_KEY_SIZE 圧縮公開鍵のサイズ（57 バイト）。
- BAD_FUNC_ARG key 引数が null の場合は返します。*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.24.2.22 function wc_ed448_sig_size

```
int wc_ed448_sig_size(
    ed448_key * key
)
```

この関数は、ED448 シグネチャのサイズ（バイト数 114）を返します。

See: wc_ed448_sign_msg

Return:

- ED448_SIG_SIZE ED448 シグネチャ（114 バイト）のサイズ。
- BAD_FUNC_ARG key 引数が null の場合は返します。*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_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_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-ja/error-crypt.h

C.25.1 Functions

	Name
void	**wc_ErrorString**(int err, char * buff) この関数は、特定のバッファ内の特定のエラーコードのエラー文字列を格納します。
const char *	**wc_GetErrorString**(int error) この関数は、特定のエラーコードのエラー文字列を返します。

C.25.2 Functions Documentation

C.25.2.1 function wc_ErrorString

```
void wc_ErrorString(
    int err,
    char * buff
)
```

この関数は、特定のバッファ内の特定のエラーコードのエラー文字列を格納します。

Parameters:

- **error** 文字列を取得するためのエラーコード *Example*

```
char errorMsg[WOLFSSL_MAX_ERROR_SZ];
int err = wc_some_function();

if( err != 0) { // error occurred
    wc_ErrorString(err, errorMsg);
}
```

See: wc_GetErrorString

Return: none いいえ返します。

C.25.2.2 function wc_GetErrorString

```
const char * wc_GetErrorString(
    int error
)
```

この関数は、特定のエラーコードのエラー文字列を返します。

See: wc_ErrorString

Return: string エラーコードのエラー文字列を文字列リテラルとして返します。*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-ja/evp.h

C.26.1 Functions

	Name
const WOLFSSL_EVP_CIPHER *	**wolfSSL_EVP_des_ede3_ecb**(void) それぞれの wolfssl_evp_cipher ポインタのゲッター関数。最初にプログラム内で wolfssl_evp_init () を 1 回呼び出す必要があります。wolfssl_des_ecb マクロは、wolfssl_evp_des_ede3_ecb () に対して定義する必要があります。
const WOLFSSL_EVP_CIPHER *	**wolfSSL_EVP_des_cbc**(void) それぞれの wolfssl_evp_cipher ポインタのゲッター関数。最初にプログラム内で wolfssl_evp_init () を 1 回呼び出す必要があります。wolfssl_des_ecb マクロは、wolfssl_evp_des_ecb () に対して定義する必要があります。
int	**wolfSSL_EVP_DigestInit_ex**(WOLFSSL_EVP_MD_CTX * ctx, const WOLFSSL_EVP_MD * type, WOLFSSL_ENGINE * impl)wolfssl_evp_md_ctx を初期化する機能。この関数は wolfssl_engine が wolfssl_engine を使用しないため、wolfssl_evp_digestinit () のラッパーです。
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)wolfssl_evp_cipher_ctx を初期化する機能。この関数は wolfssl_engine が wolfssl_engine を使用しないため、wolfssl_ciphinit () のラッパーです。
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)wolfssl_evp_cipher_ctx を初期化する機能。WolfSSL は WOLFSSL_ENGINE を使用しないため、この関数は wolfssl_evp_ciphinit () のラッパーです。暗号化フラグを暗号化するように設定します。
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)wolfssl_evp_cipher_ctx を初期化する機能。WolfSSL は WOLFSSL_ENGINE を使用しないため、この関数は wolfssl_evp_ciphinit () のラッパーです。暗号化フラグを復号化するように設定します。

	Name
int	**wolfSSL_EVP_CipherUpdate**(WOLFSSL_EVP_CIPHER_CTX * ctx, unsigned char * out, int * outl, const unsigned char * in, int inl) データを暗号化/復号化する機能。バッファ内では暗号化または復号化され、OUT バッファが結果を保持します。OUTOR は暗号化/復号化された情報の長さになります。
int	**wolfSSL_EVP_CipherFinal**(WOLFSSL_EVP_CIPHER_CTX * ctx, unsigned char * out, int * outl) この関数は、パディングを追加する最終暗号化操作を実行します。wolfssl_evp_ciph_no_padding フラグが wolfssl_evp_cipher_ctx 構造に設定されている場合、1 が返され、暗号化/復号化は行われません。PADDING FLAG が SETI パディングを追加して暗号化すると、暗号化に CTX が設定されていると、復号化されたときにパディング値がチェックされます。
int	**wolfSSL_EVP_CIPHER_CTX_set_key_length**(WOLFSSL_EVP_CIPHE * ctx, int keylen)WolfSSL EVP_CIPHER_CTX 構造キー長の設定機能
int	**wolfSSL_EVP_CIPHER_CTX_block_size**(const WOLFSSL_EVP_CIPHER_CTX * ctx) これは CTX ブロックサイズの Getter 関数です。
int	**wolfSSL_EVP_CIPHER_block_size**(const WOLFSSL_EVP_CIPHER * cipher) これは暗号のブロックサイズのゲッター関数です。
void	**wolfSSL_EVP_CIPHER_CTX_set_flags**(WOLFSSL_EVP_CIPHER_CTX * ctx, int flags)WolfSSL evp_cipher_ctx 構造の設定機能
void	**wolfSSL_EVP_CIPHER_CTX_clear_flags**(WOLFSSL_EVP_CIPHER_C * ctx, int flags)WolfSSL evp_cipher_ctx 構造のクリア機能
int	**wolfSSL_EVP_CIPHER_CTX_set_padding**(WOLFSSL_EVP_CIPHER_ * c, int pad)wolfssl_evp_cipher_ctx 構造のためのセッター機能パディングを使用する。
unsigned long	**wolfSSL_EVP_CIPHER_CTX_flags**(const WOLFSSL_EVP_CIPHER_CTX * ctx)wolfssl_evp_cipher_ctx 構造のゲッター関数廃止予定の 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
)
```

それぞれの wolfssl_evp_cipher ポインタのゲッター関数。最初にプログラム内で wolfssl_evp_init () を 1 回呼び出す必要があります。wolfssl_des_ecb マクロは、wolfssl_evp_des_ede3_ecb () に対して定義する必要があります。

See: wolfSSL_EVP_CIPHER_CTX_init

Return: pointer DES EDE3 操作のための wolfssl_evp_cipher ポインタを返します。*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
)
```

それぞれの wolfssl_evp_cipher ポインタのゲッター関数。最初にプログラム内で wolfssl_evp_init () を 1 回呼び出す必要があります。wolfssl_des_ecb マクロは、wolfssl_evp_des_ecb () に対して定義する必要があります。

See: wolfSSL_EVP_CIPHER_CTX_init

Return: pointer DES 操作のための wolfssl_evp_cipher ポインタを返します。*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
)
```

wolfssl_evp_md_ctx を初期化する機能。この関数は wolfssl_engine が wolfssl_engine を使用しないため、wolfssl_evp_digestinit () のラッパーです。

Parameters:

- **ctx** 初期化する構造
- **type** SHA などのハッシュの種類。*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
```

See:

- wolfSSL_EVP_MD_CTX_new
- wolfCrypt_Init
- wolfSSL_EVP_MD_CTX_free

Return:

- SSL_SUCCESS 正常に設定されている場合。
- SSL_FAILURE 成功しなかった場合

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
)
```

wolfssl_evp_cipher_ctx を初期化する機能。この関数は wolfssl_engine が wolfssl_engine を使用しないため、wolfssl_ciphinit（）のラッパーです。

Parameters:

- **ctx** 初期化する構造
- **type** AES などの暗号化/復号化の種類。
- **impl** 使用するエンジン。wolfssl の n / a は、null になることができます。
- **key** 設定するキー
- **iv** アルゴリズムで必要な場合は IV。*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
```

See:

- wolfSSL_EVP_CIPHER_CTX_new
- wolfCrypt_Init
- wolfSSL_EVP_CIPHER_CTX_free

Return:

- SSL_SUCCESS 正常に設定されている場合。
- SSL_FAILURE 成功しなかった場合

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
)
```

wolfssl_evp_cipher_ctx を初期化する機能。WolfSSL は WOLFSSL_ENGINE を使用しないため、この関数は wolfssl_evp_ciphinit () のラッパーです。暗号化フラグを暗号化するように設定します。

Parameters:

- **ctx** 初期化する構造
- **type** AES などの暗号化の種類。
- **impl** 使用するエンジン。wolfssl の n / a は、null になることができます。
- **key** 使用する鍵 *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
```

See:

- wolfSSL_EVP_CIPHER_CTX_new
- wolfCrypt_Init
- wolfSSL_EVP_CIPHER_CTX_free

Return:

- SSL_SUCCESS 正常に設定されている場合。
- SSL_FAILURE 成功しなかった場合

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
)
```

wolfssl_evp_cipher_ctx を初期化する機能。WolfSSL は WOLFSSL_ENGINE を使用しないため、この関数は wolfssl_evp_ciphinit () のラッパーです。暗号化フラグを復号化するように設定します。

Parameters:

- **ctx** 初期化する構造
- **type** AES などの暗号化/復号化の種類。
- **impl** 使用するエンジン。wolfssl の n / a は、null になることができます。
- **key** 設定するキー
- **iv** アルゴリズムで必要な場合は IV。*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
```

See:

- wolfSSL_EVP_CIPHER_CTX_new
- wolfCrypt_Init
- wolfSSL_EVP_CIPHER_CTX_free

Return:

- SSL_SUCCESS 正常に設定されている場合。
- SSL_FAILURE 成功しなかった場合

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
)
```

データを暗号化/復号化する機能。バッファ内では暗号化または復号化され、OUT バッファが結果を保持します。OUTOR は暗号化/復号化された情報の長さになります。

Parameters:

- **ctx** から暗号化の種類を取得するための構造。
- **out** 出力を保持するためのバッファ。
- **outl** 出力のサイズになるように調整しました。
- **in** 操作を実行するためのバッファー。*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
```

See:

- wolfSSL_EVP_CIPHER_CTX_new

- wolfCrypt_Init
- wolfSSL_EVP_CIPHER_CTX_free

Return:

- SSL_SUCCESS 成功した場合
- SSL_FAILURE 成功しなかった場合

C.26.2.8 function wolfSSL_EVP_CipherFinal

```
int wolfSSL_EVP_CipherFinal(
    WOLFSSL_EVP_CIPHER_CTX * ctx,
    unsigned char * out,
    int * outl
)
```

この関数は、パディングを追加する最終暗号化操作を実行します。wolfssl_evp_ciph_no_padding フラグが wolfssl_evp_cipher_ctx 構造に設定されている場合、1 が返され、暗号化/復号化は行われません。PADDING FLAG が SETI パディングを追加して暗号化すると、暗号化に CTX が設定されていると、復号化されたときにパディング値がチェックされます。

Parameters:

- **ctx** 復号化/暗号化する構造。
- **out** 最後の復号化/暗号化のためのバッファ。*Example*

```
WOLFSSL_EVP_CIPHER_CTX* ctx;
int out1;
unsigned char out[64];
// create ctx
wolfSSL_EVP_CipherFinal(ctx, out, &out1);
```

See: wolfSSL_EVP_CIPHER_CTX_new

Return:

- 1 成功に戻りました。
- 0 失敗に遭遇した場合

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
)
```

WolfSSL EVP_CIPHER_CTX 構造キー長の設定機能

Parameters:

- **ctx** キーの長さを設定する構造 *Example*

```
WOLFSSL_EVP_CIPHER_CTX* ctx;
int keylen;
// create ctx
wolfSSL_EVP_CIPHER_CTX_set_key_length(ctx, keylen);
```

See: wolfSSL_EVP_CIPHER_flags

Return:

- SSL_SUCCESS 正常に設定されている場合。

- SSL_FAILURE キーの長さを設定できなかった場合。

C.26.2.10 function wolfSSL_EVP_CIPHER_CTX_block_size

```
int wolfSSL_EVP_CIPHER_CTX_block_size(
    const WOLFSSL_EVP_CIPHER_CTX * ctx
)
```

これは CTX ブロックサイズの Getter 関数です。

See: wolfSSL_EVP_CIPHER_block_size

Return: size ctx-> block_size を返します。*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
)
```

これは暗号のブロックサイズのゲッター関数です。

See: wolfSSL_EVP_aes_256_ctr

Return: size ブロックサイズを返します。*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
)
```

WolfSSL evp_cipher_ctx 構造の設定機能

Parameters:

- **ctx** フラグを設定する構造 *Example*

```
WOLFSSL_EVP_CIPHER_CTX* ctx;
int flag;
// create ctx
wolfSSL_EVP_CIPHER_CTX_set_flags(ctx, flag);
```

See:

- wolfSSL_EVP_CIPHER_flags
- wolfSSL_EVP_CIPHER_CTX_flags

Return: none いいえ返します。

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
)
```

WolfSSL evp_cipher_ctx 構造のクリア機能

Parameters:

- **ctx** フラグをクリアするための構造 *Example*

```
WOLFSSL_EVP_CIPHER_CTX* ctx;
int flag;
// create ctx
wolfSSL_EVP_CIPHER_CTX_clear_flags(ctx, flag);
```

See:

- wolfSSL_EVP_CIPHER_flags
- wolfSSL_EVP_CIPHER_CTX_flags

Return: none いいえ返します。

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
)
```

wolfssl_evp_cipher_ctx 構造のためのセッター機能パディングを使用する。

Parameters:

- **ctx** パディングフラグを設定する構造 *Example*

```
WOLFSSL_EVP_CIPHER_CTX* ctx;
// create ctx
wolfSSL_EVP_CIPHER_CTX_set_padding(ctx, 1);
```

See: wolfSSL_EVP_CIPHER_CTX_new

Return:

- SSL_SUCCESS 正常に設定されている場合。
- BAD_FUNC_ARG NULL 引数が渡された場合。

C.26.2.15 function wolfSSL_EVP_CIPHER_CTX_flags

```
unsigned long wolfSSL_EVP_CIPHER_CTX_flags(
    const WOLFSSL_EVP_CIPHER_CTX * ctx
)
```

wolfssl_evp_cipher_ctx 構造のゲッター関数廃止予定の V1.1.0

See:

- wolfSSL_EVP_CIPHER_CTX_new
- wolfSSL_EVP_CIPHER_flags

Return: unsigned フラグ/モードの長い。*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-ja/hash.h

C.27.1 Functions

	Name
int	**wc_HashGetOID**(enum wc_HashType hash_type) この関数は提供された wc_hashtype の OID を返します。
int	**wc_HashGetDigestSize**(enum wc_HashType hash_type) この関数は、hash_type のダイジェスト（出力）のサイズを返します。返品サイズは、WC_HASH に提供される出力バッファが十分に大きいことを確認するために使用されます。
int	**wc_Hash**(enum wc_HashType hash_type, const byte * data, word32 data_len, byte * hash, word32 hash_len) この関数は、提供されたデータバッファ上にハッシュを実行し、提供されたハッシュバッファにそれを返します。
int	**wc_Md5Hash**(const byte * data, word32 len, byte * hash) 利便性機能は、すべてのハッシュを処理し、その結果をハッシュに入れます。
int	**wc_ShaHash**(const byte * data, word32 len, byte * hash) 利便性機能は、すべてのハッシュを処理し、その結果をハッシュに入れます。
int	**wc_Sha256Hash**(const byte * data, word32 len, byte * hash) 利便性機能は、すべてのハッシュを処理し、その結果をハッシュに入れます。
int	**wc_Sha224Hash**(const byte * data, word32 len, byte * hash) 利便性機能は、すべてのハッシュを処理し、その結果をハッシュに入れます。
int	**wc_Sha512Hash**(const byte * data, word32 len, byte * hash) 利便性機能は、すべてのハッシュを処理し、その結果をハッシュに入れます。
int	**wc_Sha384Hash**(const byte * data, word32 len, byte * hash) 利便性機能は、すべてのハッシュを処理し、その結果をハッシュに入れます。

C.27.2 Functions Documentation

C.27.2.1 function wc_HashGetOID

```
int wc_HashGetOID(
    enum wc_HashType hash_type
)
```

この関数は提供された wc_hashtype の OID を返します。

See:

- wc_HashGetDigestSize
- wc_Hash

Return:

- OID 戻り値 0 を超えてください
- HASH_TYPE_E ハッシュ型はサポートされていません。
- BAD_FUNC_ARG 提供された引数の 1 つが正しくありません。*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
)
```

この関数は、hash_type のダイジェスト（出力）のサイズを返します。返品サイズは、WC_HASH に提供される出力バッファが十分に大きいことを確認するために使用されます。

See: wc_Hash

Return:

- Success 正の戻り値は、ハッシュのダイジェストサイズを示します。
- Error hash_type がサポートされていない場合は hash_type_e を返します。
- Failure 無効な hash_type が使用された場合、bad_func_arg を返します。*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
)
```

この関数は、提供されたデータバッファ上にハッシュを実行し、提供されたハッシュバッファにそれを返します。

Parameters:

- **hash_type** “wc_hash_type_sha256” などの “enum wc_hashtype” からのハッシュ型。
- **data** ハッシュへのデータを含むバッファへのポインタ。
- **data_len** データバッファの長さ。
- **hash** 最後のハッシュを出力するために使用されるバッファへのポインタ。*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
    }
}
```

See: wc_HashGetDigestSize

Return: 0 そうでなければ、それ以外の誤り（bad_func_arg や buffer_e など）。

C.27.2.4 function wc_Md5Hash

```
int wc_Md5Hash(
    const byte * data,
    word32 len,
    byte * hash
)
```

利便性機能は、すべてのハッシュを処理し、その結果をハッシュに入れます。

Parameters:

- **data** ハッシュへのデータ
- **len** データの長さ *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.
}
```

See:

- wc_Md5Hash
- wc_Md5Final
- wc_InitMd5

Return:

- 0 データを正常にハッシュしたときに返されます。
- Memory_E メモリエラー、メモリを割り当てることができません。これは、小さなスタックオプションが有効になっているだけです。

C.27.2.5 function wc_ShaHash

```
int wc_ShaHash(
    const byte * data,
    word32 len,
    byte * hash
)
```

利便性機能は、すべてのハッシュを処理し、その結果をハッシュに入れます。

Parameters:

- **data** ハッシュへのデータ

- **len** データの長さ *Example*

none

See:

- wc_ShaHash
- wc_ShaFinal
- wc_InitSha

Return:

- 0 うまく返されました...。
- Memory_E メモリエラー、メモリを割り当てることができません。これは、小さなスタックオプションが有効になっているだけです。

C.27.2.6 function wc_Sha256Hash

```
int wc_Sha256Hash(
    const byte * data,
    word32 len,
    byte * hash
)
```

利便性機能は、すべてのハッシュを処理し、その結果をハッシュに入れます。

Parameters:

- **data** ハッシュへのデータ
- **len** データの長さ *Example*

none

See:

- wc_Sha256Hash
- wc_Sha256Final
- wc_InitSha256

Return:

- 0 うまく返されました...
- Memory_E メモリエラー、メモリを割り当てることができません。これは、小さなスタックオプションが有効になっているだけです。

C.27.2.7 function wc_Sha224Hash

```
int wc_Sha224Hash(
    const byte * data,
    word32 len,
    byte * hash
)
```

利便性機能は、すべてのハッシュを処理し、その結果をハッシュに入れます。

Parameters:

- **data** ハッシュへのデータ
- **len** データの長さ *Example*

none

See:

- wc_InitSha224
- wc_Sha224Update
- wc_Sha224Final

Return:

- 0 成功
- <0 エラー

C.27.2.8 function wc_Sha512Hash

```
int wc_Sha512Hash(
    const byte * data,
    word32 len,
    byte * hash
)
```

利便性機能は、すべてのハッシュを処理し、その結果をハッシュに入れます。

Parameters:

- **data** ハッシュへのデータ
- **len** データの長さ *Example*

none

See:

- wc_Sha512Hash
- wc_Sha512Final
- wc_InitSha512

Return:

- 0 入力されたデータを正常にハッシュしたときに返されます
- Memory_E メモリエラー、メモリを割り当てることができません。これは、小さなスタックオプションが有効になっているだけです。

C.27.2.9 function wc_Sha384Hash

```
int wc_Sha384Hash(
    const byte * data,
    word32 len,
    byte * hash
)
```

利便性機能は、すべてのハッシュを処理し、その結果をハッシュに入れます。

Parameters:

- **data** ハッシュへのデータ
- **len** データの長さ *Example*

none

See:

- wc_Sha384Hash
- wc_Sha384Final
- wc_InitSha384

Return:

- 0 データを正常にハッシュしたときに返されます
- Memory_E メモリエラー、メモリを割り当てることができません。これは、小さなスタックオプションが有効になっているだけです。

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_Sha256Hash(const byte* data, word32 len, byte* hash);

int wc_Sha224Hash(const byte* data, word32 len, byte* hash);

int wc_Sha512Hash(const byte* data, word32 len, byte* hash);

int wc_Sha384Hash(const byte* data, word32 len, byte* hash);
```

C.28 dox_comments/header_files-ja/hmac.h

C.28.1 Functions

	Name
int	**wc_HmacSetKey**(Hmac * hmac, int type, const byte * key, word32 keySz) この関数は HMAC オブジェクトを初期化し、その暗号化タイプ、キー、および HMAC の長さを設定します。
int	**wc_HmacUpdate**(Hmac * hmac, const byte * in, word32 sz) この関数は、HMAC を使用して認証するメッセージを更新します。HMAC オブジェクトが WC_HMACSETKEY で初期化された後に呼び出されるべきです。この関数は、ハッシュへのメッセージを更新するために複数回呼び出されることがあります。必要に応じて wc_hmacupdate を呼び出した後、最終認証済みメッセージタグを取得するために wc_hmacfinal を呼び出す必要があります。
int	**wc_HmacFinal**(Hmac * hmac, byte * out) この関数は、HMAC オブジェクトのメッセージの最終ハッシュを計算します。
int	**wolfSSL_GetHmacMaxSize**(void) この関数は、構成された暗号スイートに基づいて使用可能な最大の HMAC ダイジェストサイズを返します。

	Name
int	**wc_HKDF**(int type, const byte * inKey, word32 inKeySz, const byte * salt, word32 saltSz, const byte * info, word32 infoSz, byte * out, word32 outSz) この関数は、HMAC キー導出機能（HKDF）へのアクセスを提供します。HMAC を利用して、任意の SALT とオプションの情報を派生したキーに変換します。0 または NULL が指定されている場合、ハッシュ型はデフォルトで MD5 になります。

C.28.2 Functions Documentation

C.28.2.1 function wc_HmacSetKey

```
int wc_HmacSetKey(
    Hmac * hmac,
    int type,
    const byte * key,
    word32 keySz
)
```

この関数は HMAC オブジェクトを初期化し、その暗号化タイプ、キー、および HMAC の長さを設定します。

Parameters:

- **hmac** 初期化する HMAC オブジェクトへのポインタ
- **type** HMAC オブジェクトを使用する暗号化方式を指定します。有効なオプションは次のとおりです.MD5、SHA、SHA256、SHA384、SHA3-224、SHA3-256、SHA3-384、SHA3-512
- **key** HMAC オブジェクトを初期化するキーを含むバッファへのポインタ *Example*

```
Hmac hmac;
byte key[] = { // initialize with key to use for encryption };
if (wc_HmacSetKey(&hmac, MD5, key, sizeof(key)) != 0) {
    // error initializing Hmac object
}
```

See:

- wc_HmacUpdate
- wc_HmacFinal

Return:

- 0 HMAC オブジェクトの初期化に成功しました
- BAD_FUNC_ARG 入力タイプが無効な場合は返されます。有効なオプションは次のとおりです.MD5、SHA、SHA256、SHA384、SHA3-224、SHA3-256、SHA3-384、SHA3-512
- MEMORY_E ハッシュに使用する構造体の割り当てメモリの割り当てエラーがある場合
- HMAC_MIN_KEYLEN_E FIPS 実装を使用するときに、指定されたキーが FIPS 規格の最小許容 (14 バイト) よりも短い

C.28.2.2 function wc_HmacUpdate

```
int wc_HmacUpdate(
    Hmac * hmac,
    const byte * in,
```

```
    word32 sz
)
```

この関数は、HMAC を使用して認証するメッセージを更新します。HMAC オブジェクトが WC_HMACSETKEY で初期化された後に呼び出されるべきです。この関数は、ハッシュへのメッセージを更新するために複数回呼び出されることがあります。必要に応じて wc_hmacupdate を呼び出した後、最終認証済みメッセージタグを取得するために wc_hmacfinal を呼び出す必要があります。

Parameters:

- **hmac** メッセージを更新する HMAC オブジェクトへのポインタ
- **msg** 追加するメッセージを含むバッファへのポインタ *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
}
```

See:

- wc_HmacSetKey
- wc_HmacFinal

Return:

- 0 認証するメッセージの更新に成功しました
- MEMORY_E ハッシュアルゴリズムで使用するためのメモリ割り当てエラーがある場合

C.28.2.3 function wc_HmacFinal

```
int wc_HmacFinal(
    Hmac * hmac,
    byte * out
)
```

この関数は、HMAC オブジェクトのメッセージの最終ハッシュを計算します。

Parameters:

- **hmac** 最終ハッシュを計算する HMAC オブジェクトへのポインタ *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
}
```

See:

- wc_HmacSetKey
- wc_HmacUpdate

Return:

- 0 最後のハッシュの計算に成功した
- MEMORY_E ハッシュアルゴリズムで使用するためにメモリを割り当てるエラーがある場合

C.28.2.4 function wolfSSL_GetHmacMaxSize

```
int wolfSSL_GetHmacMaxSize(
    void
)
```

この関数は、構成された暗号スイートに基づいて使用可能な最大の HMAC ダイジェストサイズを返します。

See: none

Return: Success 設定された暗号スイートに基づいて使用可能な最大の HMAC ダイジェストサイズを返します *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
)
```

この関数は、HMAC キー導出機能（HKDF）へのアクセスを提供します。HMAC を利用して、任意の SALT とオプションの情報を派生したキーに変換します。0 または NULL が指定されている場合、ハッシュ型はデフォルトで MD5 になります。

Parameters:

- **type** HKDF に使用するハッシュタイプ。有効な型は次のとおりです.MD5、SHA、SHA256、SHA384、SHA3-224、SHA3-256、SHA3-384、SHA3-512
- **inKey** KDF に使用するキーを含むバッファへのポインタ
- **inKeySz** 入力キーの長さ
- **salt** 任意のソルトを含むバッファへのポインタ。ソルトを使用しない場合は代わりに NULL を使用してください
- **saltSz** ソルトの長さ。ソルトを使用しない場合は 0 を使用してください
- **info** オプションの追加情報を含むバッファへのポインタ。追加情報を追加していない場合は NULL を使用してください
- **infoSz** 追加情報の長さ追加情報を使用しない場合は 0 を使用してください
- **out** 派生キーを保存するバッファへのポインタ *Example*

```
byte key[] = { // initialize with key };
byte salt[] = { // initialize with salt };
byte derivedKey[MAX_DIGEST_SIZE];

int ret = wc_HKDF(SHA512, key, sizeof(key), salt, sizeof(salt),
NULL, 0, derivedKey, sizeof(derivedKey));
if ( ret != 0 ) {
```

```
    // error generating derived key
}
```

See: wc_HmacSetKey

Return:

- 0 与えられた入力でキーの生成に成功したら返されます
- BAD_FUNC_ARG 無効なハッシュ型が引数として指定されている場合に返されます。有効な型は次のとおりです.MD5、SHA、SHA256、SHA384、SHA3-224、SHA3-256、SHA3-384、SHA3-512
- MEMORY_E メモリの割り当て中にエラーが発生した場合に返されます
- HMAC_MIN_KEYLEN_E FIPS 実装を使用するときに返されることがあり、指定されたキー長は最小許容 FIPS 規格よりも短いです。

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);
```

C.29 dox_comments/header_files-ja/iotsafe.h

C.29.1 Functions

	Name
int	**wolfSSL_CTX_iotsafe_enable**(WOLFSSL_CTX * ctx) この関数は与えられたコンテキストでの IoT セーフサポートを有効にします。
int	**wolfSSL_iotsafe_on**(WOLFSSL * ssl, byte privkey_id, byte ecdh_keypair_slot, byte peer_pubkey_slot, byte peer_cert_slot) この関数は、IOT-SAFE TLS コールバックを特定の SSL セッションに接続します。
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) この関数は、IOT-SAFE TLS コールバックを特定の SSL セッションに接続します。これは、IOT セーフスロットの ID を参照で渡すことができ、ID フィールドの長さをパラメータ “id_size” で指定できます。

	Name
void	**wolfIoTSafe_SetCSIM_read_cb**(wolfSSL_IOTSafe_CSIM_read_cb rf)AT + CSIM コマンドのリードコールバックを関連付けます。この入力関数は通常、モデムと通信する UART チャネルの読み取りイベントに関連付けられています。読み取りコールバックが関連付けられているのは、同時に IoT-Safe サポートを使用するすべてのコンテキストのグローバルと変更です。*Example*
void	**wolfIoTSafe_SetCSIM_write_cb**(wolfSSL_IOTSafe_CSIM_write_cb wf)AT + CSIM コマンドの書き込みコールバックを関連付けます。この出力関数は通常、モデムと通信する UART チャネル上のライトイベントに関連付けられています。Write Callback が関連付けられているのは、同時に IoT-Safe サポートを使用するすべてのコンテキストのグローバルと変更です。*Example*
int	**wolfIoTSafe_GetRandom**(unsigned char * out, word32 sz)IOT セーフ機能 getrandom を使用して、指定されたサイズのランダムなバッファを生成します。この関数は、WolfCrypt RNG オブジェクトによって自動的に使用されます。
int	**wolfIoTSafe_GetCert**(uint8_t id, unsigned char * output, unsigned long sz)IOT-Safe アプレット上のファイルに保存されている証明書をインポートし、ローカルにメモリに保存します。1 バイトのファイル ID フィールドで動作します。
int	**wolfIoTSafe_GetCert_ex**(uint8_t * id, uint16_t id_sz, unsigned char * output, unsigned long sz)IOT-Safe アプレット上のファイルに保存されている証明書をインポートし、ローカルにメモリに保存します。 ref wolfiotsafe_getcert “wolfiotsafe_getcert” と同等です。ただし、2 バイト以上のファイル ID で呼び出すことができます。
int	**wc_iotsafe_ecc_import_public**(ecc_key * key, byte key_id)IOT セーフアプレットに格納されている ECC 256 ビットの公開鍵を ECC_Key オブジェクトにインポートします。
int	**wc_iotsafe_ecc_export_public**(ecc_key * key, byte key_id)ECC_KEY オブジェクトから IOT-SAFE アプレットへの書き込み可能なパブリックキースロットに ECC 256 ビット公開鍵をエクスポートします。
int	**wc_iotsafe_ecc_import_public_ex**(ecc_key * key, byte * key_id, word16 id_size)ECC_KEY オブジェクトから IOT-SAFE アプレットへの書き込み可能なパブリックキースロットに ECC 256 ビット公開鍵をエクスポートします。 ref WC_IOTSAFE_ECC_IMPORT_PUBLIC「WC_IOTSAFE_ECC_IMPORT_PUBLIC」と同等のものは、2 バイト以上のキー ID で呼び出すことができる点を除きます。

	Name
int	**wc_iotsafe_ecc_export_private**(ecc_key * key, byte key_id)ECC 256 ビットキーを ECC_KEY オブジェクトから IOT セーフアプレットに書き込み可能なプライベートキースロットにエクスポートします。
int	**wc_iotsafe_ecc_export_private_ex**(ecc_key * key, byte * key_id, word16 id_size)ECC 256 ビットキーを ECC_KEY オブジェクトから IOT セーフアプレットに書き込み可能なプライベートキースロットにエクスポートします。 ref WC_IOTSAFE_ECC_EXPORT_PRIVATE「WC_IOTSAFE_ECC_EXPORT_PRIVATE」を除き、2 バイト以上のキー ID を呼び出すことができる点を除き、
int	**wc_iotsafe_ecc_sign_hash**(byte * in, word32 inlen, byte * out, word32 * outlen, byte key_id) 事前計算された 256 ビットハッシュに署名して、IOT-SAFE アプレットに、以前に保存されたプライベートキー、またはプリプロビジョニングされています。
int	**wc_iotsafe_ecc_sign_hash_ex**(byte * in, word32 inlen, byte * out, word32 * outlen, byte * key_id, word16 id_size) 事前計算された 256 ビットハッシュに署名して、IOT-SAFE アプレットに、以前に保存されたプライベートキー、またはプリプロビジョニングされています。 ref wc_iotsafe_ecc_sign_hash “wc_iotsafe_ecc_sign_hash” と同等です。ただし、2 バイト以上のキー ID で呼び出すことができます。
int	**wc_iotsafe_ecc_verify_hash**(byte * sig, word32 siglen, byte * hash, word32 hashlen, int * res, byte key_id) 予め計算された 256 ビットハッシュに対する ECC シグネチャを、IOT-SAFE アプレット内のプリプロビジョニング、またはプロビジョニングされたプリプロビジョニングを使用します。結果は RES に書き込まれます。1 が有効で、0 が無効です。注：有効なテストに戻り値を使用しないでください。Res のみを使用してください。

	Name
int	**wc_iotsafe_ecc_verify_hash_ex**(byte * sig, word32 siglen, byte * hash, word32 hashlen, int * res, byte * key_id, word16 id_size) 予め計算された 256 ビットハッシュに対する ECC シグネチャを、IOT-SAFE アプレット内のプリプロビジョニング、またはプロビジョニングされたプリプロビジョニングを使用します。結果は RES に書き込まれます。1 が有効で、0 が無効です。注：有効なテストに戻り値を使用しないでください。Res のみを使用してください。 ref WC_IOTSAFE_ECC_VERIFY_HASH “WC_IOTSAFE_ECC_VERIFY_HASH” を除き、2 バイト以上のキー ID で呼び出すことができる点を除きます。
int	**wc_iotsafe_ecc_gen_k**(byte key_id)ECC 256 ビットのキーペアを生成し、それを（書き込み可能な）スロットに IOT セーフなアプレットに保存します。

C.29.2 Functions Documentation

C.29.2.1 function wolfSSL_CTX_iotsafe_enable

```
int wolfSSL_CTX_iotsafe_enable(
    WOLFSSL_CTX * ctx
)
```

この関数は与えられたコンテキストでの IoT セーフサポートを有効にします。

Parameters:

- **ctx** IOT セーフサポートを有効にする必要がある WOLFSSL_CTX オブジェクトへのポインタ

See:

- wolfSSL_iotsafe_on
- wolfIoTSafe_SetCSIM_read_cb
- wolfIoTSafe_SetCSIM_write_cb

Return: 0 成功した *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
)
```

この関数は、IOT-SAFE TLS コールバックを特定の SSL セッションに接続します。

Parameters:

- **ssl** コールバックが有効になる WolfSSL オブジェクトへのポインタ
- **privkey_id** ホストの秘密鍵を含む IOT セーフなアプレットスロットの ID
- **ecdh_keypair_slot** ECDH 鍵ペアを保存するための IoT 安全アプレットスロットの ID
- **peer_pubkey_slot** ECDH 用の他のエンドポイントの公開鍵を保存するための IOT-SAFE アプレットスロットの ID
- **peer_cert_slot** 検証のための他のエンドポイントの公開鍵を保存するための IOT セーフなアプレットスロットの ID

See:

- wolfSSL_iotsafe_on_ex
- wolfSSL_CTX_iotsafe_enable

Return:

- 0 成功すると
- NOT_COMPILED_IN habe_pk_callbacks が無効になっている場合 *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);
}
```

スロットの ID が 1 バイトの長さの場合、SSL セッションを IoT-Safe アプレットに接続するように呼び出す必要があります。IOT セーフスロットの ID が 2 バイト以上の場合、 REF WOLFSSL_IOTSAFE_ON_EX「WOLFSSL_IOTSAFE_ON_EX ()」を使用する必要があります。

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
)
```

この関数は、IOT-SAFE TLS コールバックを特定の SSL セッションに接続します。これは、IOT セーフスロットの ID を参照で渡すことができ、ID フィールドの長さをパラメータ “id_size” で指定できます。

Parameters:

- **ssl** コールバックが有効になる WolfSSL オブジェクトへのポインタ

- **privkey_id** ホストの秘密鍵を含む IoT セーフアプレットスロットの ID へのポインタ
- **ecdh_keypair_slot** ECDH 鍵ペアを保存する IOT-Safe アプレットスロットの ID へのポインタ
- **peer_pubkey_slot** ECDH 用の他のエンドポイントの公開鍵を保存する IOT セーフアプレットスロットの ID へのポインタ
- **peer_cert_slot** 検証のために他のエンドポイントの公開鍵を保存するための IOT-SAFE アプレットスロットの ID へのポインタ
- **id_size** 各スロット ID のサイズ

See:

- wolfSSL_iotsafe_on
- wolfSSL_CTX_iotsafe_enable

Return:

- 0 成功すると
- NOT_COMPILED_IN habe_pk_callbacks が無効になっている場合 *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
)
```

AT + CSIM コマンドのリードコールバックを関連付けます。この入力関数は通常、モデムと通信する UART チャネルの読み取りイベントに関連付けられています。読み取りコールバックが関連付けられているのは、同時に IoT-Safe サポートを使用するすべてのコンテキストのグローバルと変更です。*Example*

See: wolfIoTSafe_SetCSIM_write_cb

```
// 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
)
```

AT + CSIM コマンドの書き込みコールバックを関連付けます。この出力関数は通常、モデムと通信する UART チャネル上のライトイベントに関連付けられています。Write Callback が関連付けられているのは、同時に IoT-Safe サポートを使用するすべてのコンテキストのグローバルと変更です。*Example*

See: wolfIoTSafe_SetCSIM_read_cb

```
// 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
)
```

IOT セーフ機能 getrandom を使用して、指定されたサイズのランダムなバッファを生成します。この関数は、WolfCrypt RNG オブジェクトによって自動的に使用されます。

Parameters:

- **out** ランダムなバイトシーケンスが格納されているバッファ。
- **sz** 生成するランダムシーケンスのサイズ（バイト単位）

C.29.2.7 function wolfIoTSafe_GetCert

```
int wolfIoTSafe_GetCert(
    uint8_t id,
    unsigned char * output,
    unsigned long sz
)
```

IOT-Safe アプレット上のファイルに保存されている証明書をインポートし、ローカルにメモリに保存します。1 バイトのファイル ID フィールドで動作します。

Parameters:

- **id** 証明書が保存されている IOT セーフ・アプレットのファイル ID
- **output** 証明書がインポートされるバッファー
- **sz** バッファ出力で使用可能な最大サイズ

Return: the 輸入された証明書の長さ *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
)
```

IOT-Safe アプレット上のファイルに保存されている証明書をインポートし、ローカルにメモリに保存します。 ref wolfiotsafe_getcert “wolfiotsafe_getcert” と同等です。ただし、2 バイト以上のファイル ID で呼び出すことができます。

Parameters:

- **id** 証明書が保存されている IOT-SAFE アプレットのファイル ID へのポインタ
- **id_sz** ファイル ID のサイズ：バイト数
- **output** 証明書がインポートされるバッファー
- **sz** バッファ出力で使用可能な最大サイズ

Return: the 輸入された証明書の長さ *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
)
```

IOT セーフアプレットに格納されている ECC 256 ビットの公開鍵を ECC_Key オブジェクトにインポートします。

Parameters:

- **key** IOT-SAFE アプレットからインポートされたキーを含む ECC_KEY オブジェクト
- **id** 公開鍵が保存されている IOT セーフアプレットのキー ID

See:

- wc_iotsafe_ecc_export_public
- wc_iotsafe_ecc_export_private

Return: 0 成功すると

C.29.2.10 function wc_iotsafe_ecc_export_public

```
int wc_iotsafe_ecc_export_public(
    ecc_key * key,
    byte key_id
)
```

ECC_KEY オブジェクトから IOT-SAFE アプレットへの書き込み可能なパブリックキースロットに ECC 256 ビット公開鍵をエクスポートします。

Parameters:

- **key** エクスポートする鍵を含む ecc_key オブジェクト
- **id** 公開鍵が保存されている IOT セーフアプレットのキー ID

See:

- wc_iotsafe_ecc_import_public_ex
- wc_iotsafe_ecc_export_private

Return: 0 成功すると

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
)
```

ECC_KEY オブジェクトから IOT-SAFE アプレットへの書き込み可能なパブリックキースロットに ECC 256 ビット公開鍵をエクスポートします。 ref WC_IOTSAFE_ECC_IMPORT_PUBLIC「WC_IOTSAFE_ECC_IMPORT_PUBLIC」と同等のものは、2 バイト以上のキー ID で呼び出すことができる点を除きます。

Parameters:

- **key** エクスポートする鍵を含む ecc_key オブジェクト
- **id** 公開鍵が保存される IOT セーフアプレットのキー ID へのポインタ
- **id_size** キー ID サイズ

See:

- wc_iotsafe_ecc_import_public
- wc_iotsafe_ecc_export_private

Return: 0 成功すると

C.29.2.12 function wc_iotsafe_ecc_export_private

```
int wc_iotsafe_ecc_export_private(
    ecc_key * key,
    byte key_id
)
```

ECC 256 ビットキーを ECC_KEY オブジェクトから IOT セーフアプレットに書き込み可能なプライベートキースロットにエクスポートします。

Parameters:

- **key** エクスポートする鍵を含む ecc_key オブジェクト
- **id** 秘密鍵が保存される IOT セーフアプレットのキー ID

See:

- wc_iotsafe_ecc_export_private_ex
- wc_iotsafe_ecc_import_public
- wc_iotsafe_ecc_export_public

Return: 0 成功すると

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
)
```

ECC 256 ビットキーを ECC_KEY オブジェクトから IOT セーフアプレットに書き込み可能なプライベートキースロットにエクスポートします。 ref WC_IOTSAFE_ECC_EXPORT_PRIVATE「WC_IOTSAFE_ECC_EXPORT_PRIVATE」を除き、2 バイト以上のキー ID を呼び出すことができる点を除き、

Parameters:

- **key** エクスポートする鍵を含む ecc_key オブジェクト
- **id** 秘密鍵が保存される IOT セーフアプレットのキー ID へのポインタ
- **id_size** キー ID サイズ

See:

- wc_iotsafe_ecc_export_private
- wc_iotsafe_ecc_import_public
- wc_iotsafe_ecc_export_public

Return: 0 成功すると

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
)
```

事前計算された 256 ビットハッシュに署名して、IOT-SAFE アプレットに、以前に保存されたプライベートキー、またはプリプロビジョニングされています。

Parameters:

- **in** サインするメッセージハッシュを含むバッファへのポインタ
- **inlen** 署名するメッセージの長さ
- **out** 生成された署名を保存するためのバッファ
- **outlen** 出力バッファの最大長。バイトを保存します
- **id** メッセージ署名の生成に成功したときに書き込まれたペイロードに署名するための秘密鍵を含むスロットの IOT セーフアプレットのキー ID

See:

- wc_iotsafe_ecc_sign_hash_ex
- wc_iotsafe_ecc_verify_hash
- wc_iotsafe_ecc_gen_k

Return: 0 成功すると

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
)
```

事前計算された 256 ビットハッシュに署名して、IOT-SAFE アプレットに、以前に保存されたプライベートキー、またはプリプロビジョニングされています。ref wc_iotsafe_ecc_sign_hash “wc_iotsafe_ecc_sign_hash” と同等です。ただし、2 バイト以上のキー ID で呼び出すことができます。

Parameters:

- **in** サインするメッセージハッシュを含むバッファへのポインタ
- **inlen** 署名するメッセージの長さ
- **out** 生成された署名を保存するためのバッファ
- **outlen** 出力バッファの最大長。バイトを保存します
- **id** 秘密鍵を含むスロットの IOT-SAFE アプレットのキー ID へのポインタメッセージ署名の生成に成功したときに書き込まれるペイロードに署名する
- **id_size** キー ID サイズ

See:

- wc_iotsafe_ecc_sign_hash
- wc_iotsafe_ecc_verify_hash
- wc_iotsafe_ecc_gen_k

Return: 0 成功すると

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
)
```

予め計算された 256 ビットハッシュに対する ECC シグネチャを、IOT-SAFE アプレット内のプリプロビジョニング、またはプロビジョニングされたプリプロビジョニングを使用します。結果は RES に書き込まれます。1 が有効で、0 が無効です。注：有効なテストに戻り値を使用しないでください。Res のみを使用してください。

Parameters:

- **sig** 検証する署名を含むバッファ
- **hash** 署名されたハッシュ（メッセージダイジェスト）
- **hashlen** ハッシュの長さ（オクテット）
- **res** 署名の結果、1 == 有効、0 == 無効

See:

- wc_iotsafe_ecc_verify_hash_ex
- wc_iotsafe_ecc_sign_hash
- wc_iotsafe_ecc_gen_k

Return:

- 0 成功すると（署名が無効であっても）
- < 故障の場合は 0

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
)
```

予め計算された 256 ビットハッシュに対する ECC シグネチャを、IOT-SAFE アプレット内のプリプロビジョニング、またはプロビジョニングされたプリプロビジョニングを使用します。結果は RES に書き込まれます。1 が有効で、0 が無効です。注：有効なテストに戻り値を使用しないでください。Res のみを使用してください。ref WC_IOTSAFE_ECC_VERIFY_HASH “WC_IOTSAFE_ECC_VERIFY_HASH” を除き、2 バイト以上のキー ID で呼び出すことができる点を除きます。

Parameters:

- **sig** 検証する署名を含むバッファ
- **hash** 署名されたハッシュ（メッセージダイジェスト）
- **hashlen** ハッシュの長さ（オクテット）

- **res** 署名の結果、1 == 有効、0 == 無効
- **key_id** パブリック ECC キーが IOT セーフアプレットに保存されているスロットの ID

See:

- wc_iotsafe_ecc_verify_hash
- wc_iotsafe_ecc_sign_hash
- wc_iotsafe_ecc_gen_k

Return:

- 0 成功すると（署名が無効であっても）
- < 故障の場合は 0

C.29.2.18 function wc_iotsafe_ecc_gen_k

```
int wc_iotsafe_ecc_gen_k(
    byte key_id
)
```

ECC 256 ビットのキーペアを生成し、それを（書き込み可能な）スロットに IOT セーフなアプレットに保存します。

Parameters:

- **key_id** ECC キーペアが IOT セーフアプレットに格納されているスロットの ID。
- **key_id** ECC キーペアが IOT セーフアプレットに格納されているスロットの ID。
- **id_size** キー ID サイズ

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 成功すると
- 0 成功すると

ECC 256 ビットのキーペアを生成し、それを（書き込み可能な）スロットに IOT セーフなアプレットに保存します。 ref wc_iotsafe_ecc_gen_k “wc_iotsafe_ecc_gen_k” と同等です。ただし、2 バイト以上のキー ID で呼び出すことができます。

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-ja/logging.h

C.30.1 Functions

	Name
int	**wolfSSL_SetLoggingCb**(wolfSSL_Logging_cb log_function) この関数は、WolfSSL ログメッセージを処理するために使用されるロギングコールバックを登録します。デフォルトでは、システムが IT fprintf () を STDERR にサポートしている場合は、この関数を使用することによって、ユーザーによって何でも実行できます。
int	**wolfSSL_Debugging_ON**(void) ビルド時にロギングが有効になっている場合、この関数は実行時にロギングをオンにします。ビルド時にログ記録を有効にするには–enable-debug または debug_wolfssl を定義します。
void	**wolfSSL_Debugging_OFF**(void) この関数はランタイムロギングメッセージをオフにします。彼らがすでに消えている場合は、行動はとられません。

C.30.2 Functions Documentation

C.30.2.1 function wolfSSL_SetLoggingCb

```
int wolfSSL_SetLoggingCb(
    wolfSSL_Logging_cb log_function
)
```

この関数は、WolfSSL ログメッセージを処理するために使用されるロギングコールバックを登録します。デフォルトでは、システムが IT fprintf () を STDERR にサポートしている場合は、この関数を使用することによって、ユーザーによって何でも実行できます。

See:

- wolfSSL_Debugging_ON
- wolfSSL_Debugging_OFF

Return:

- Success 成功した場合、この関数は 0 を返します。
- BAD_FUNC_ARG 関数ポインタが提供されていない場合に返されるエラーです。*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
)
```

ビルド時にロギングが有効になっている場合、この関数は実行時にロギングをオンにします。ビルド時にログ記録を有効にするには–enable-debug または debug_wolfssl を定義します。

See:

- wolfSSL_Debugging_OFF
- wolfSSL_SetLoggingCb

Return:

- 0 成功すると。
- NOT_COMPILED_IN このビルドに対してロギングが有効になっていない場合は返されるエラーです。
  Example

```
wolfSSL_Debugging_ON();
```

C.30.2.3 function wolfSSL_Debugging_OFF

```
void wolfSSL_Debugging_OFF(
    void
)
```

この関数はランタイムロギングメッセージをオフにします。彼らがすでに消えている場合は、行動はとられません。

See:

- wolfSSL_Debugging_ON
- wolfSSL_SetLoggingCb

Return: none いいえ返します。*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-ja/md2.h

C.31.1 Functions

	Name
void	**wc_InitMd2**(Md2 *) この関数は MD2 を初期化します。これは WC_MD2HASH によって自動的に呼び出されます。

	Name
void	**wc_Md2Update**(Md2 * md2, const byte * data, word32 len) 長さ LEN の提供されたバイト配列を絶えずハッシュするように呼び出すことができます。
void	**wc_Md2Final**(Md2 * md2, byte * hash) データのハッシュを確定します。結果はハッシュに入れられます。
int	**wc_Md2Hash**(const byte * data, word32 len, byte * hash) 利便性機能は、すべてのハッシュを処理し、その結果をハッシュに入れます。

C.31.2 Functions Documentation

C.31.2.1 function wc_InitMd2

```
void wc_InitMd2(
    Md2 *
)
```

この関数は MD2 を初期化します。これは WC_MD2HASH によって自動的に呼び出されます。

See:

- wc_Md2Hash
- wc_Md2Update
- wc_Md2Final

Return: 0 初期化に成功したときに返されます *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
)
```

長さ LEN の提供されたバイト配列を絶えずハッシュするように呼び出すことができます。

Parameters:

- **md2** 暗号化に使用する MD2 構造へのポインタ
- **data** ハッシュするデータ *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);
}
```

See:

- wc_Md2Hash
- wc_Md2Final
- wc_InitMd2

Return: 0 データをダイジェストに正常に追加すると返されます。

C.31.2.3 function wc_Md2Final

```
void wc_Md2Final(
    Md2 * md2,
    byte * hash
)
```

データのハッシュを確定します。結果はハッシュに入れられます。

Parameters:

- **md2** 暗号化に使用する MD2 構造へのポインタ *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);
}
```

See:

- wc_Md2Hash
- wc_Md2Final
- wc_InitMd2

Return: 0 ファイナライズに成功したときに返されます。

C.31.2.4 function wc_Md2Hash

```
int wc_Md2Hash(
    const byte * data,
    word32 len,
    byte * hash
)
```

利便性機能は、すべてのハッシュを処理し、その結果をハッシュに入れます。

Parameters:

- **data** ハッシュへのデータ
- **len** データの長さ *Example*

none

See:

- wc_Md2Hash
- wc_Md2Final
- wc_InitMd2

Return:

- 0 データを正常にハッシュしたときに返されます。
- Memory_E メモリエラー、メモリを割り当てることができません。これは、小さなスタックオプションが有効になっているだけです。

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-ja/md4.h

C.32.1 Functions

	Name
void	**wc_InitMd4**(Md4 *) この関数は MD4 を初期化します。これは WC_MD4HASH によって自動的に呼び出されます。
void	**wc_Md4Update**(Md4 * md4, const byte * data, word32 len) 長さ LEN の提供されたバイト配列を絶えずハッシュするように呼び出すことができます。
void	**wc_Md4Final**(Md4 * md4, byte * hash) データのハッシュを確定します。結果はハッシュに入れられます。

C.32.2 Functions Documentation

C.32.2.1 function wc_InitMd4

```
void wc_InitMd4(
    Md4 *
)
```

この関数は MD4 を初期化します。これは WC_MD4HASH によって自動的に呼び出されます。

See:

- wc_Md4Hash

- wc_Md4Update
- wc_Md4Final

Return: 0 初期化に成功したときに返されます *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
)
```

長さ LEN の提供されたバイト配列を絶えずハッシュするように呼び出すことができます。

Parameters:

- **md4** 暗号化に使用する MD4 構造へのポインタ
- **data** ハッシュするデータ *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);
}
```

See:

- wc_Md4Hash
- wc_Md4Final
- wc_InitMd4

Return: 0 データをダイジェストに正常に追加すると返されます。

C.32.2.3 function wc_Md4Final

```
void wc_Md4Final(
    Md4 * md4,
    byte * hash
)
```

データのハッシュを確定します。結果はハッシュに入れられます。

Parameters:

- **md4** 暗号化に使用する MD4 構造へのポインタ *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);
}
```

See:

- wc_Md4Hash
- wc_Md4Final
- wc_InitMd4

Return: 0 ファイナライズに成功したときに返されます。

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-ja/md5.h

C.33.1 Functions

	Name
int	**wc_InitMd5**(wc_Md5 *) この関数は MD5 を初期化します。これは WC_MD5HASH によって自動的に呼び出されます。
int	**wc_Md5Update**(wc_Md5 * md5, const byte * data, word32 len) 長さ LEN の提供されたバイト配列を絶えずハッシュするように呼び出すことができます。
int	**wc_Md5Final**(wc_Md5 * md5, byte * hash) データのハッシュを確定します。結果はハッシュに入れられます。MD5 構造体がリセットされます。注：この関数は、habe_intel_qa が定義されている場合に intelqasymmd5 () を呼び出す結果も返します。
void	**wc_Md5Free**(wc_Md5 *)MD5 構造をリセットします。注：これは、wolfssl_ti_hash が定義されている場合にのみサポートされています。
int	**wc_Md5GetHash**(wc_Md5 * md5, byte * hash) ハッシュデータを取得します。結果はハッシュに入れられます。MD5 構造はリセットされません。

C.33.2 Functions Documentation

C.33.2.1 function wc_InitMd5

```
int wc_InitMd5(
    wc_Md5 *
)
```

この関数は MD5 を初期化します。これは WC_MD5HASH によって自動的に呼び出されます。

See:

- wc_Md5Hash
- wc_Md5Update
- wc_Md5Final

Return:

- 0 初期化に成功したときに返されます。
- BAD_FUNC_ARG MD5 構造が NULL 値として渡された場合に返されます。*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
)
```

長さ LEN の提供されたバイト配列を絶えずハッシュするように呼び出すことができます。

Parameters:

- **md5** 暗号化に使用する MD5 構造へのポインタ
- **data** ハッシュするデータ *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.
   }
}
```

See:

- wc_Md5Hash
- wc_Md5Final
- wc_InitMd5

Return:

- 0 データをダイジェストに正常に追加すると返されます。
- BAD_FUNC_ARG MD5 構造が NULL の場合、またはデータが NULL で、LEN がゼロより大きい場合に返されます。DATA パラメーターが NULL で LEN がゼロの場合、関数はエラーを返してはいけません。

C.33.2.3 function wc_Md5Final

```
int wc_Md5Final(
    wc_Md5 * md5,
    byte * hash
)
```

データのハッシュを確定します。結果はハッシュに入れられます。MD5 構造体がリセットされます。注：この関数は、habe_intel_qa が定義されている場合に intelqasymmd5 () を呼び出す結果も返します。

Parameters:

- **md5** 暗号化に使用する MD5 構造へのポインタ *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.
   }
}
```

See:

- wc_Md5Hash
- wc_InitMd5
- wc_Md5GetHash

Return:

- 0 ファイナライズに成功したときに返されます。
- BAD_FUNC_ARG MD5 構造またはハッシュポインタが NULL で渡された場合に返されます。

C.33.2.4 function wc_Md5Free

```
void wc_Md5Free(
    wc_Md5 *
)
```

MD5 構造をリセットします。注：これは、wolfssl_ti_hash が定義されている場合にのみサポートされています。

See:

- wc_InitMd5
- wc_Md5Update
- wc_Md5Final

Return: none いいえ返します。*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
)
```

ハッシュデータを取得します。結果はハッシュに入れられます。MD5 構造はリセットされません。

Parameters:

- **md5** 暗号化に使用する MD5 構造へのポインタ。*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);
}
```

See:

- wc_Md5Hash

- wc_Md5Final
- wc_InitMd5

Return: none いいえリターン

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-ja/memory.h

C.34.1 Functions

	Name
void *	**wolfSSL_Malloc**(size_t size, void * heap, int type) この関数は malloc（）と似ていますが、WolfSSL が使用するように構成されているメモリ割り当て関数を呼び出します。デフォルトでは、WolfSSL は malloc（）を使用します。これは、WolfSSL メモリ抽象化レイヤを使用して変更できます - wolfssl_setAllocator（）を参照してください。注 WOLFSSL_MALLOC は、WOLFSSL によって直接呼び出されませんが、代わりに Macro XMalloc によって呼び出されます。デフォルトのビルドの場合、size 引数のみが存在します。wolfssl_static_memory ビルドを使用する場合は、ヒープとタイプ引数が含まれます。
void	**wolfSSL_Free**(void * ptr, void * heap, int type) この関数は free（）と似ていますが、WolfSSL が使用するように構成されているメモリフリー機能を呼び出します。デフォルトでは、WolfSSL は free（）を使用します。これは、WolfSSL メモリ抽象化レイヤを使用して変更できます - wolfssl_setAllocator（）を参照してください。注 WOLFSSL_FREE は WOLFSSL によって直接呼び出されませんが、代わりにマクロ XFree によって呼び出されます。デフォルトのビルドの場合、PTR 引数のみが存在します。wolfssl_static_memory ビルドを使用する場合は、ヒープとタイプ引数が含まれます。

	Name
void *	**wolfSSL_Realloc**(void * ptr, size_t size, void * heap, int type) この関数は REALLOC () と似ていますが、WolfSSL が使用するように構成されているメモリ再割り当て機能を呼び出します。デフォルトでは、WolfSSL は RealLoc () を使用します。これは、WolfSSL メモリ抽象化レイヤを使用して変更できます - wolfssl_setAllocator () を参照してください。注 WOLFSSL_REALLOC は WOLFSSL によって直接呼び出されませんが、代わりにマクロ Xrealloc によって呼び出されます。デフォルトのビルドの場合、size 引数のみが存在します。wolfssl_static_memory ビルドを使用する場合は、ヒープとタイプ引数が含まれます。
int	**wolfSSL_SetAllocators**(wolfSSL_Malloc_cb , wolfSSL_Free_cb , wolfSSL_Realloc_cb) この機能は、WolfSSL が使用する割り当て関数を登録します。デフォルトでは、システムがそれをサポートしている場合、Malloc / Free と RealLoc が使用されます。この機能を使用すると、実行時にユーザーは独自のメモリハンドラをインストールできます。
int	**wolfSSL_StaticBufferSz**(byte * buffer, word32 sz, int flag) この機能は、静的メモリ機能が使用されている場合（–enable-staticMemory）の場合に使用できます。メモリの「バケット」に最適なバッファサイズを示します。これにより、パーティション化された後に追加の未使用のメモリが終了しないように、バッファサイズを計算する方法が可能になります。返された値は、正の場合、使用するコンピュータのバッファサイズです。
int	**wolfSSL_MemoryPaddingSz**(void) この機能は、静的メモリ機能が使用されている場合（–enable-staticMemory）の場合に使用できます。メモリの各パーティションに必要なパディングのサイズを示します。このパディングサイズは、メモリアライメントのために追加のメモリ管理構造を含む必要があるサイズになります。

C.34.2 Functions Documentation

C.34.2.1 function wolfSSL_Malloc

```
void * wolfSSL_Malloc(
    size_t size,
    void * heap,
    int type
)
```

この関数は malloc () と似ていますが、WolfSSL が使用するように構成されているメモリ割り当て関数を呼び出します。デフォルトでは、WolfSSL は malloc () を使用します。これは、WolfSSL メモリ抽象化レイヤを使用して変更できます - wolfssl_setAllocator () を参照してください。注 WOLFSSL_MALLOC は、WOLFSSL によって直接呼び出されませんが、代わりに Macro XMalloc によって呼び出されます。デフォル

トのビルドの場合、size 引数のみが存在します。wolfssl_static_memory ビルドを使用する場合は、ヒープとタイプ引数が含まれます。

Parameters:

- **size** 割り当てるメモリのサイズ（バイト）
- **heap** メモリに使用するヒントヒント。null になることができます *Example*

```
int* tenInts = (int*)wolfSSL_Malloc(sizeof(int)*10);
```

See:

- wolfSSL_Free
- wolfSSL_Realloc
- wolfSSL_SetAllocators
- XMALLOC
- XFREE
- XREALLOC

Return:

- pointer 成功した場合、この関数は割り当てられたメモリへのポインタを返します。
- error エラーがある場合は、NULL が返されます。

C.34.2.2 function wolfSSL_Free

```
void wolfSSL_Free(
    void * ptr,
    void * heap,
    int type
)
```

この関数は free（）と似ていますが、WolfSSL が使用するように構成されているメモリフリー機能を呼び出します。デフォルトでは、WolfSSL は free（）を使用します。これは、WolfSSL メモリ抽象化レイヤを使用して変更できます - wolfssl_setAllocator（）を参照してください。注 WOLFSSL_FREE は WOLFSSL によって直接呼び出されませんが、代わりにマクロ XFree によって呼び出されます。デフォルトのビルドの場合、PTR 引数のみが存在します。wolfssl_static_memory ビルドを使用する場合は、ヒープとタイプ引数が含まれます。

Parameters:

- **ptr** 解放されるメモリへのポインタ。
- **heap** メモリに使用するヒントヒント。null になることができます *Example*

```
int* tenInts = (int*)wolfSSL_Malloc(sizeof(int)*10);
// process data as desired
...
if(tenInts) {
    wolfSSL_Free(tenInts);
}
```

See:

- wolfSSL_Alloc
- wolfSSL_Realloc
- wolfSSL_SetAllocators
- XMALLOC
- XFREE
- XREALLOC

Return: none いいえ返します。

C.34.2.3 function wolfSSL_Realloc

```
void * wolfSSL_Realloc(
    void * ptr,
    size_t size,
    void * heap,
    int type
)
```

この関数は REALLOC () と似ていますが、WolfSSL が使用するように構成されているメモリ再割り当て機能を呼び出します。デフォルトでは、WolfSSL は RealLoc () を使用します。これは、WolfSSL メモリ抽象化レイヤを使用して変更できます - wolfssl_setAllocator () を参照してください。注 WOLFSSL_REALLOC は WOLFSSL によって直接呼び出されませんが、代わりにマクロ Xrealloc によって呼び出されます。デフォルトのビルドの場合、size 引数のみが存在します。wolfssl_static_memory ビルドを使用する場合は、ヒープとタイプ引数が含まれます。

Parameters:

- **ptr** 再割り当てされているメモリへのポインタ。
- **size** 割り当てるバイト数。
- **heap** メモリに使用するヒントヒント。null になることができます *Example*

```
int* tenInts = (int*)wolfSSL_Malloc(sizeof(int)*10);
int* twentyInts = (int*)wolfSSL_Realloc(tenInts, sizeof(int)*20);
```

See:

- wolfSSL_Free
- wolfSSL_Malloc
- wolfSSL_SetAllocators
- XMALLOC
- XFREE
- XREALLOC

Return:

- pointer 成功した場合、この関数はマイポイントを再割り当てするためのポインタを返します。これは PTR と同じポインタ、または新しいポインタの場所であり得る。
- Null エラーがある場合は、NULL が返されます。

C.34.2.4 function wolfSSL_SetAllocators

```
int wolfSSL_SetAllocators(
    wolfSSL_Malloc_cb ,
    wolfSSL_Free_cb ,
    wolfSSL_Realloc_cb
)
```

この機能は、WolfSSL が使用する割り当て関数を登録します。デフォルトでは、システムがそれをサポートしている場合、Malloc / Free と RealLoc が使用されます。この機能を使用すると、実行時にユーザーは独自のメモリハンドラをインストールできます。

Parameters:

- **malloc_function** 使用する WolfSSL のメモリ割り当て機能関数署名は、上記の wolfssl_malloc_cb プロトタイプと一致する必要があります。
- **free_function** 使用する WolfSSL のメモリフリー機能関数シグネチャは、上記の wolfssl_free_cb プロトタイプと一致する必要があります。*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
}
```

See: none

Return:

- Success 成功した場合、この関数は 0 を返します。
- BAD_FUNC_ARG 関数ポインタが提供されていない場合に返されるエラーです。

C.34.2.5 function wolfSSL_StaticBufferSz

```
int wolfSSL_StaticBufferSz(
    byte * buffer,
    word32 sz,
    int flag
)
```

この機能は、静的メモリ機能が使用されている場合（–enable-staticMemory）の場合に使用できます。メモリの「バケット」に最適なバッファサイズを示します。これにより、パーティション化された後に追加の未使用のメモリが終了しないように、バッファサイズを計算する方法が可能になります。返された値は、正の場合、使用するコンピュータのバッファサイズです。

Parameters:

- **buffer** バッファへのポインタ
- **size** バッファのサイズ *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);
...
```

See:

- wolfSSL_Malloc
- wolfSSL_Free

Return:

- Success バッファサイズ計算を正常に完了すると、正の値が返されます。この返された値は最適なバッファサイズです。
- Failure すべての負の値はエラーの場合と見なされます。

C.34.2.6 function wolfSSL_MemoryPaddingSz

```
int wolfSSL_MemoryPaddingSz(
    void
)
```

この機能は、静的メモリ機能が使用されている場合（–enable-staticMemory）の場合に使用できます。メモリの各パーティションに必要なパディングのサイズを示します。このパディングサイズは、メモリアライメントのために追加のメモリ管理構造を含む必要があるサイズになります。

See:

- wolfSSL_Malloc
- wolfSSL_Free

Return:

- On 正常なメモリパディング計算戻り値は正の値になります
- All 負の値はエラーケースと見なされます。*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.34.3 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);
```

C.35 dox_comments/header_files-ja/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) この関数は、PEM 形式の wolfssl_bio 構造体にキーを書き込みます。

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
)
```

この関数は、PEM 形式の wolfssl_bio 構造体にキーを書き込みます。

Parameters:

- **bio** wolfssl_bio 構造体から PEM バッファを取得します。
- **key** PEM 形式に変換するためのキー。
- **cipher** EVP 暗号構造
- **passwd** パスワード。
- **len** パスワードの長さ
- **cb** パスワードコールバック *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
```

See: wolfSSL_PEM_read_bio_X509_AUX

Return:

- SSL_SUCCESS 成功すると。
- SSL_FAILURE 失敗すると。

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-ja/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)
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-ja/pkcs7.h

C.37.1 Functions

	Name
int	**wc_PKCS7_InitWithCert**(PKCS7 * pkcs7, byte * cert, word32 certSz) この関数は、DER フォーマットの証明書を使用して PKCS7 構造を初期化します。空の PKCS7 構造を初期化するには、NULL CERT と CERTSZ の場合は 0 を渡すことができます。
void	**wc_PKCS7_Free**(PKCS7 * pkcs7) この関数は、PKCS7 の初期化装置によって割り当てられたメモリを解放します。
int	**wc_PKCS7_EncodeData**(PKCS7 * pkcs7, byte * output, word32 outputSz) この関数は PKCS7 データコンテンツタイプを構築し、PKCS7 構造をパーセル可能な PKCS7 データパケットを含むバッファにエンコードします。

	Name
int	**wc_PKCS7_EncodeSignedData**(PKCS7 * pkcs7, byte * output, word32 outputSz) この関数は PKCS7 署名付きデータコンテンツタイプを構築し、PKCS7 構造を PARSable PKCS7 署名付きデータパケットを含むバッファにエンコードします。
int	**wc_PKCS7_EncodeSignedData_ex**(PKCS7 * pkcs7, const byte * hashBuf, word32 hashSz, byte * outputHead, word32 * outputHeadSz, byte * outputFoot, word32 * outputFootSz) この関数は、PKCS7 の署名付きデータコンテンツタイプを構築し、PKCS7 構造をエンコードし、Parsable PKCS7 署名付きデータパケットを含むヘッダーおよびフッターバッファにエンコードします。これにはコンテンツは含まれません。ハッシュを計算してデータに提供する必要があります
int	**wc_PKCS7_VerifySignedData**(PKCS7 * pkcs7, byte * pkiMsg, word32 pkiMsgSz) この関数は、送信された PKCS7 の署名付きデータメッセージを取り、証明書リストと証明書失効リストを抽出してから署名を検証します。与えられた PKCS7 構造に抽出されたコンテンツを格納します。
int	**wc_PKCS7_VerifySignedData_ex**(PKCS7 * pkcs7, const byte * hashBuf, word32 hashSz, byte * pkiMsgHead, word32 pkiMsgHeadSz, byte * pkiMsgFoot, word32 pkiMsgFootSz) この機能は、送信された PKCS7 署名付きデータメッセージを HASH /ヘッダー/フッターとして取り出してから、証明書リストと証明書失効リストを抽出してから、署名を検証します。与えられた PKCS7 構造に抽出されたコンテンツを格納します。
int	**wc_PKCS7_EncodeEnvelopedData**(PKCS7 * pkcs7, byte * output, word32 outputSz) この関数は、PKCS7 構造を編集し、PKCS7 構造を符号化し、Parsable PKCS7 エンベロープデータパケットを含むバッファに編集します。
int	**wc_PKCS7_DecodeEnvelopedData**(PKCS7 * pkcs7, byte * pkiMsg, word32 pkiMsgSz, byte * output, word32 outputSz) この関数は PKCS7 エンベロープデータコンテンツタイプをアントラップして復号化し、メッセージを出力にデコードします。渡された PKCS7 オブジェクトの秘密鍵を使用してメッセージを復号化します。

C.37.2 Functions Documentation

C.37.2.1 function wc_PKCS7_InitWithCert

```
int wc_PKCS7_InitWithCert(
    PKCS7 * pkcs7,
    byte * cert,
    word32 certSz
)
```

この関数は、DER フォーマットの証明書を使用して PKCS7 構造を初期化します。空の PKCS7 構造を初期化するには、NULL CERT と CERTSZ の場合は 0 を渡すことができます。

Parameters:

- **pkcs7** デコードされた証明書を保存する PKCS7 構造へのポインタ
- **cert** PKCS7 構造を初期化するための DER フォーマットの ASN.1 証明書を含むバッファへのポインタ *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
}
```

See: wc_PKCS7_Free

Return:

- 0 PKCS7 構造の初期化に成功しました
- MEMORY_E xmalloc でメモリを割り当てるエラーがある場合
- ASN_PARSE_E 証明書ヘッダーの解析中にエラーがある場合
- ASN_OBJECT_ID_E 証明書から暗号化タイプの解析中にエラーがある場合に返されます
- ASN_EXPECT_0_E CERT ファイルの暗号化仕様にフォーマットエラーがある場合
- ASN_BEFORE_DATE_E 日付が証明書開始日以前の場合返却
- ASN_AFTER_DATE_E 日付が証明書の有効期限の後にある場合に返されます
- ASN_BITSTR_E 証明書からビット文字列を解析したエラーがある場合に返されます。
- ECC_CURVE_OID_E 証明書から ECC キーの解析中にエラーがある場合
- ASN_UNKNOWN_OID_E 証明書が不明なキーオブジェクト ID を使用している場合に返されます
- ASN_VERSION_E allow_v1_extensions オプションが定義されておらず、証明書が V1 または V2 の証明書の場合に返されます。
- BAD_FUNC_ARG 証明書拡張機能の処理中にエラーがある場合
- ASN_CRIT_EXT_E 証明書の処理中になじみのない重要な拡張機能が発生した場合に返されます。
- ASN_SIG_OID_E 署名暗号化タイプが提供されたファイル内の証明書の暗号化タイプと同じでない場合に返されます。
- ASN_SIG_CONFIRM_E 認証署名が失敗したことを確認した場合に返されます
- ASN_NAME_INVALID_E 証明書の名前が CA 名制約によって許可されていない場合に返されます。
- ASN_NO_SIGNER_E 証明書の真正性を確認するための CA 署名者がない場合に返されました

C.37.2.2 function wc_PKCS7_Free

```
void wc_PKCS7_Free(
    PKCS7 * pkcs7
)
```

この関数は、PKCS7 の初期化装置によって割り当てられたメモリを解放します。

See: wc_PKCS7_InitWithCert

Return: none いいえ返します。*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
)
```

この関数は PKCS7 データコンテンツタイプを構築し、PKCS7 構造をパーセル可能な PKCS7 データパケットを含むバッファにエンコードします。

Parameters:

- **pkcs7** 符号化する PKCS7 構造へのポインタ
- **output** エンコードされた証明書を保存するバッファへのポインタ *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
}
```

See: wc_PKCS7_InitWithCert

Return:

- Success PKCS7 データをバッファに正常にエンコードすると、PKCS7 構造内の索引を返します。このインデックスは、出力バッファに書き込まれたバイトにも対応しています。
- BUFFER_E 指定されたバッファがエンコードされた証明書を保持するのに十分な大きさでない場合に返されます

C.37.2.4 function wc_PKCS7_EncodeSignedData

```
int wc_PKCS7_EncodeSignedData(
    PKCS7 * pkcs7,
    byte * output,
    word32 outputSz
)
```

この関数は PKCS7 署名付きデータコンテンツタイプを構築し、PKCS7 構造を PARSable PKCS7 署名付きデータパケットを含むバッファにエンコードします。

Parameters:

- **pkcs7** 符号化する PKCS7 構造へのポインタ
- **output** エンコードされた証明書を保存するバッファへのポインタ *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);
```

See:

- wc_PKCS7_InitWithCert
- wc_PKCS7_VerifySignedData

Return:

- Success PKCS7 データをバッファに正常にエンコードすると、PKCS7 構造内の索引を返します。このインデックスは、出力バッファに書き込まれたバイトにも対応しています。
- BAD_FUNC_ARG PKCS7 構造が署名付きデータパケットを生成するための 1 つ以上の要求要素が欠落している場合に返されます。
- MEMORY_E メモリの割り当て中にエラーが発生した場合に返されます
- PUBLIC_KEY_E 公開鍵の解析中にエラーがある場合
- RSA_BUFFER_E バッファエラーが発生した場合は、小さすぎたり入力が大きすぎたりし過ぎました
- BUFFER_E 指定されたバッファがエンコードされた証明書を保持するのに十分な大きさでない場合に返されます
- MP_INIT_E 署名を生成するエラーがある場合は返却される可能性があります
- MP_READ_E 署名を生成するエラーがある場合は返却される可能性があります
- MP_CMP_E 署名を生成するエラーがある場合は返却される可能性があります
- MP_INVMOD_E 署名を生成するエラーがある場合は返却される可能性があります
- MP_EXPTMOD_E 署名を生成するエラーがある場合は返却される可能性があります
- MP_MOD_E 署名を生成するエラーがある場合は返却される可能性があります
- MP_MUL_E 署名を生成するエラーがある場合は返却される可能性があります
- MP_ADD_E 署名を生成するエラーがある場合は返却される可能性があります
- MP_MULMOD_E 署名を生成するエラーがある場合は返却される可能性があります
- MP_TO_E 署名を生成するエラーがある場合は返却される可能性があります
- MP_MEM 署名を生成するエラーがある場合は返却される可能性があります

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
)
```

この関数は、PKCS7 の署名付きデータコンテンツタイプを構築し、PKCS7 構造をエンコードし、Parsable PKCS7 署名付きデータパケットを含むヘッダーおよびフッターバッファにエンコードします。これにはコンテンツは含まれません。ハッシュを計算してデータに提供する必要があります

Parameters:

- **pkcs7** 符号化する PKCS7 構造へのポインタ
- **hashBuf** コンテンツデータの計算ハッシュへのポインタ
- **hashSz** ダイジェストのサイズ
- **outputHead** エンコードされた証明書ヘッダーを保存するバッファへのポインタ
- **outputHeadSz** 出力ヘッダーバッファのサイズが入力され、実際のサイズを返します。
- **outputFoot** エンコードされた証明書フッターを保存するバッファへのポインタ *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);
```

See:

- wc_PKCS7_InitWithCert
- wc_PKCS7_VerifySignedData_ex

Return:

- 0=Success
- BAD_FUNC_ARG PKCS7 構造が署名付きデータパケットを生成するための 1 つ以上の要求要素が欠落している場合に返されます。
- MEMORY_E メモリの割り当て中にエラーが発生した場合に返されます
- PUBLIC_KEY_E 公開鍵の解析中にエラーがある場合
- RSA_BUFFER_E バッファエラーが発生した場合は、小さすぎたり入力が大きすぎたりし過ぎました
- BUFFER_E 指定されたバッファがエンコードされた証明書を保持するのに十分な大きさでない場合に返されます
- MP_INIT_E 署名を生成するエラーがある場合は返却される可能性があります
- MP_READ_E 署名を生成するエラーがある場合は返却される可能性があります
- MP_CMP_E 署名を生成するエラーがある場合は返却される可能性があります
- MP_INVMOD_E 署名を生成するエラーがある場合は返却される可能性があります
- MP_EXPTMOD_E 署名を生成するエラーがある場合は返却される可能性があります
- MP_MOD_E 署名を生成するエラーがある場合は返却される可能性があります
- MP_MUL_E 署名を生成するエラーがある場合は返却される可能性があります
- MP_ADD_E 署名を生成するエラーがある場合は返却される可能性があります
- MP_MULMOD_E 署名を生成するエラーがある場合は返却される可能性があります
- MP_TO_E 署名を生成するエラーがある場合は返却される可能性があります
- MP_MEM 署名を生成するエラーがある場合は返却される可能性があります

C.37.2.6 function wc_PKCS7_VerifySignedData

```
int wc_PKCS7_VerifySignedData(
    PKCS7 * pkcs7,
    byte * pkiMsg,
    word32 pkiMsgSz
)
```

この関数は、送信された PKCS7 の署名付きデータメッセージを取り、証明書リストと証明書失効リストを抽出してから署名を検証します。与えられた PKCS7 構造に抽出されたコンテンツを格納します。

Parameters:

- **pkcs7** 解析された証明書を保存する PKCS7 構造へのポインタ
- **pkiMsg** 署名されたメッセージを含むバッファへのポインタを検証および復号化する *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);
```

See:

- wc_PKCS7_InitWithCert
- wc_PKCS7_EncodeSignedData

Return:

- 0 メッセージから情報を抽出することに成功しました
- BAD_FUNC_ARG 入力パラメータの 1 つが無効な場合は返されます
- ASN_PARSE_E 与えられた PKIMSG から解析中のエラーがある場合に返されます
- PKCS7_OID_E 与えられた PKIMSG が署名付きデータ型ではない場合に返されます
- ASN_VERSION_E PKCS7 署名者情報がバージョン 1 ではない場合に返されます
- MEMORY_E メモリの割り当て中にエラーが発生した場合に返されます
- PUBLIC_KEY_E 公開鍵の解析中にエラーがある場合
- RSA_BUFFER_E バッファエラーが発生した場合は、小さすぎたり入力が大きすぎたりし過ぎません
- BUFFER_E 指定されたバッファがエンコードされた証明書を保持するのに十分な大きさでない場合に返されます
- MP_INIT_E 署名を生成するエラーがある場合は返却される可能性があります
- MP_READ_E 署名を生成するエラーがある場合は返却される可能性があります
- MP_CMP_E 署名を生成するエラーがある場合は返却される可能性があります
- MP_INVMOD_E 署名を生成するエラーがある場合は返却される可能性があります
- MP_EXPTMOD_E 署名を生成するエラーがある場合は返却される可能性があります
- MP_MOD_E 署名を生成するエラーがある場合は返却される可能性があります
- MP_MUL_E 署名を生成するエラーがある場合は返却される可能性があります
- MP_ADD_E 署名を生成するエラーがある場合は返却される可能性があります
- MP_MULMOD_E 署名を生成するエラーがある場合は返却される可能性があります
- MP_TO_E 署名を生成するエラーがある場合は返却される可能性があります
- MP_MEM 署名を生成するエラーがある場合は返却される可能性があります

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
)
```

この機能は、送信された PKCS7 署名付きデータメッセージを HASH /ヘッダー/フッターとして取り出してから、証明書リストと証明書失効リストを抽出してから、署名を検証します。与えられた PKCS7 構造に抽出されたコンテンツを格納します。

Parameters:

- **pkcs7** 解析された証明書を保存する PKCS7 構造へのポインタ
- **hashBuf** コンテンツデータの計算ハッシュへのポインタ
- **hashSz** ダイジェストのサイズ

- **pkiMsgHead** 署名されたメッセージヘッダーを含むバッファへのポインタを検証およびデコードする
- **pkiMsgHeadSz** 署名付きメッセージヘッダーのサイズ
- **pkiMsgFoot** 署名されたメッセージフッターを含むバッファへのポインタを検証してデコードする

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);
```

See:

- wc_PKCS7_InitWithCert
- wc_PKCS7_EncodeSignedData_ex

Return:

- 0 メッセージから情報を抽出することに成功しました
- BAD_FUNC_ARG 入力パラメータの 1 つが無効な場合は返されます
- ASN_PARSE_E 与えられた PKIMSG から解析中のエラーがある場合に返されます
- PKCS7_OID_E 与えられた PKIMSG が署名付きデータ型ではない場合に返されます
- ASN_VERSION_E PKCS7 署名者情報がバージョン 1 ではない場合に返されます
- MEMORY_E メモリの割り当て中にエラーが発生した場合に返されます
- PUBLIC_KEY_E 公開鍵の解析中にエラーがある場合
- RSA_BUFFER_E バッファエラーが発生した場合は、小さすぎたり入力が大きすぎたりし過ぎません

- BUFFER_E 指定されたバッファがエンコードされた証明書を保持するのに十分な大きさでない場合に返されます
- MP_INIT_E 署名を生成するエラーがある場合は返却される可能性があります
- MP_READ_E 署名を生成するエラーがある場合は返却される可能性があります
- MP_CMP_E 署名を生成するエラーがある場合は返却される可能性があります
- MP_INVMOD_E 署名を生成するエラーがある場合は返却される可能性があります
- MP_EXPTMOD_E 署名を生成するエラーがある場合は返却される可能性があります
- MP_MOD_E 署名を生成するエラーがある場合は返却される可能性があります
- MP_MUL_E 署名を生成するエラーがある場合は返却される可能性があります
- MP_ADD_E 署名を生成するエラーがある場合は返却される可能性があります
- MP_MULMOD_E 署名を生成するエラーがある場合は返却される可能性があります
- MP_TO_E 署名を生成するエラーがある場合は返却される可能性があります
- MP_MEM 署名を生成するエラーがある場合は返却される可能性があります

C.37.2.8 function wc_PKCS7_EncodeEnvelopedData

```
int wc_PKCS7_EncodeEnvelopedData(
    PKCS7 * pkcs7,
    byte * output,
    word32 outputSz
)
```

この関数は、PKCS7 構造を編集し、PKCS7 構造を符号化し、Parsable PKCS7 エンベロープデータパケットを含むバッファに編集します。

Parameters:

- **pkcs7** 符号化する PKCS7 構造へのポインタ
- **output** エンコードされた証明書を保存するバッファへのポインタ *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
}
```

See:

- wc_PKCS7_InitWithCert
- wc_PKCS7_DecodeEnvelopedData

Return:

- Success エンベロープデータ形式でメッセージを正常にエンコードする上で返信され、出力バッファに書き込まれたサイズを返します。

- BAD_FUNC_ARG: 入力パラメータの 1 つが無効な場合、または PKCS7 構造が必要な要素を欠落している場合
- ALGO_ID_E pkcs7 構造がサポートされていないアルゴリズムタイプを使用している場合に返されます。現在、DESB と DES3B のみがサポートされています
- BUFFER_E 与えられた出力バッファが小さすぎて出力データを保存する場合に返されます
- MEMORY_E メモリの割り当て中にエラーが発生した場合に返されます
- RNG_FAILURE_E 暗号化の乱数発生器の初期化中にエラーがある場合
- DRBG_FAILED 暗号化に使用される乱数発生器を使用して数字を生成するエラーが発生した場合

C.37.2.9 function wc_PKCS7_DecodeEnvelopedData

```
int wc_PKCS7_DecodeEnvelopedData(
    PKCS7 * pkcs7,
    byte * pkiMsg,
    word32 pkiMsgSz,
    byte * output,
    word32 outputSz
)
```

この関数は PKCS7 エンベロープデータコンテンツタイプをアントラップして復号化し、メッセージを出力にデコードします。渡された PKCS7 オブジェクトの秘密鍵を使用してメッセージを復号化します。

Parameters:

- **pkcs7** エンベロープデータパッケージをデコードする秘密鍵を含む PKCS7 構造へのポインタ
- **pkiMsg** エンベロープデータパッケージを含むバッファへのポインタ
- **pkiMsgSz** 包み込まれたデータパッケージのサイズ
- **output** デコードされたメッセージを保存するバッファへのポインタ *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
}
```

See:

- wc_PKCS7_InitWithCert
- wc_PKCS7_EncodeEnvelopedData

Return:

- On メッセージから情報を抽出するには、出力に書き込まれたバイト数を返します。
- BAD_FUNC_ARG 入力パラメータの 1 つが無効な場合は返されます
- ASN_PARSE_E 与えられた PKIMSG から解析中のエラーがある場合に返されます
- PKCS7_OID_E 与えられた PKIMSG がエンベロープデータ型ではない場合に返されます
- ASN_VERSION_E PKCS7 署名者情報がバージョン 0 ではない場合に返されます
- MEMORY_E メモリの割り当て中にエラーが発生した場合に返されます

- ALGO_ID_E pkcs7 構造がサポートされていないアルゴリズムタイプを使用している場合に返されます。現在、Signature Generation for Signature Generation の RSAK を使用して、DESB と DES3B のみが暗号化でサポートされています。
- PKCS7_RECIP_E 提供された受信者と一致するエンベロープデータに受信者が見つからない場合
- RSA_BUFFER_E バッファエラーが原因で RSA シグネチャ検証中にエラーがある場合は、小さすぎたり入力が大きすぎたりすると元に戻されます。
- MP_INIT_E 署名検証中にエラーがある場合は返却される可能性があります
- MP_READ_E 署名検証中にエラーがある場合は返却される可能性があります
- MP_CMP_E 署名検証中にエラーがある場合は返却される可能性があります
- MP_INVMOD_E 署名検証中にエラーがある場合は返却される可能性があります
- MP_EXPTMOD_E 署名検証中にエラーがある場合は返却される可能性があります
- MP_MOD_E 署名検証中にエラーがある場合は返却される可能性があります
- MP_MUL_E 署名検証中にエラーがある場合は返却される可能性があります
- MP_ADD_E 署名検証中にエラーがある場合は返却される可能性があります
- MP_MULMOD_E 署名検証中にエラーがある場合は返却される可能性があります
- MP_TO_E 署名検証中にエラーがある場合は返却される可能性があります
- MP_MEM 署名検証中にエラーがある場合は返却される可能性があります

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-ja/poly1305.h

C.38.1 Functions

	Name
int	**wc_Poly1305SetKey**(Poly1305 * poly1305, const byte * key, word32 kySz) この関数は、Poly1305 コンテキスト構造のキーを設定し、ハッシュに初期化します。注：セキュリティを確保するために、WC_POLY1305FINAL でメッセージハッシュを生成した後に新しいキーを設定する必要があります。
int	**wc_Poly1305Update**(Poly1305 * poly1305, const byte * m, word32 bytes) この関数は、Poly1305 構造を持つハッシュにメッセージを更新します。
int	**wc_Poly1305Final**(Poly1305 * poly1305, byte * tag) この関数は入力メッセージのハッシュを計算し、結果を MAC に格納します。この後、キーをリセットする必要があります。
int	**wc_Poly1305_MAC**(Poly1305 * ctx, byte * additional, word32 addSz, byte * input, word32 sz, byte * tag, word32 tagSz) 鍵がロードされ、最近の TLS AEAD パディング方式を使用して MAC（タグ）を作成する初期化された Poly1305 構造体を取ります。

C.38.2 Functions Documentation

C.38.2.1 function wc_Poly1305SetKey

```
int wc_Poly1305SetKey(
    Poly1305 * poly1305,
    const byte * key,
    word32 kySz
)
```

この関数は、Poly1305 コンテキスト構造のキーを設定し、ハッシュに初期化します。注：セキュリティを確保するために、WC_POLY1305FINAL でメッセージハッシュを生成した後に新しいキーを設定する必要があります。

Parameters:

- **ctx** 初期化するための Poly1305 構造へのポインタ
- **key** ハッシュに使用する鍵を含むバッファへのポインタ *Example*

```
Poly1305 enc;
byte key[] = { initialize with 32 byte key to use for hashing };
wc_Poly1305SetKey(&enc, key, sizeof(key));
```

See:

- wc_Poly1305Update
- wc_Poly1305Final

Return:

- 0 キーを正常に設定し、Poly1305 構造の初期化
- BAD_FUNC_ARG 与えられたキーが 32 バイトの長さでない場合、または Poly1305 コンテキストが NULL の場合

C.38.2.2 function wc_Poly1305Update

```
int wc_Poly1305Update(
    Poly1305 * poly1305,
    const byte * m,
    word32 bytes
)
```

この関数は、Poly1305 構造を持つハッシュにメッセージを更新します。

Parameters:

- **ctx** HASH にメッセージを更新するための Poly1305 構造へのポインタ
- **m** ハッシュに追加する必要があるメッセージを含むバッファへのポインタ *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
}
```

See:

- wc_Poly1305SetKey
- wc_Poly1305Final

Return:

- 0 ハッシュへのメッセージの更新に成功しました
- BAD_FUNC_ARG Poly1305 構造が NULL の場合に返されます

C.38.2.3 function wc_Poly1305Final

```
int wc_Poly1305Final(
    Poly1305 * poly1305,
    byte * tag
)
```

この関数は入力メッセージのハッシュを計算し、結果を MAC に格納します。この後、キーをリセットする必要があります。

Parameters:

- **ctx** MAC を生成するための Poly1305 構造へのポインタ *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
}
```

See:

- wc_Poly1305SetKey
- wc_Poly1305Update

Return:

- 0 最後の Mac の計算に成功した
- BAD_FUNC_ARG Poly1305 構造が NULL の場合に返されます

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
)
```

鍵がロードされ、最近の TLS AEAD パディング方式を使用して MAC（タグ）を作成する初期化された Poly1305 構造体を取ります。

Parameters:

- **ctx** 初期化された Poly1305 構造物
- **additional** 使用する追加データ
- **addSz** 追加バッファのサイズ
- **input** からタグを作成するための入力バッファ
- **sz** 入力バッファのサイズ
- **tag** 作成したタグを保持するためのバッファー *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
}
```

See:

- wc_Poly1305SetKey
- wc_Poly1305Update
- wcPoly1305Final

Return:

- 0 成功
- BAD_FUNC_ARG CTX、INPUT、または TAG が NULL の場合、または追加が NULL で、ADDSZ が 0 より大きい場合、または TAGSZ が WC_POLY1305_MAC_SZ より小さい場合に返されます。

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-ja/psa.h

C.39.1 Functions

	Name
int	**wolfSSL_CTX_psa_enable**(WOLFSSL_CTX * ctx) この関数は、与えられたコンテキストでの PSA サポートを可能にします。
int	**wolfSSL_set_psa_ctx**(WOLFSSL * ssl, struct psa_ssl_ctx * ctx) 与えられた SSL セッションの PSA コンテキストを設定する機能
void	**wolfSSL_free_psa_ctx**(struct psa_ssl_ctx * ctx) この関数は PSA コンテキストによって使用されるリソースを解放します
int	**wolfSSL_psa_set_private_key_id**(struct psa_ssl_ctx * ctx, psa_key_id_t id) この関数は、SSL セッションによって使用される秘密鍵を設定します

C.39.2 Functions Documentation

C.39.2.1 function wolfSSL_CTX_psa_enable

```
int wolfSSL_CTX_psa_enable(
    WOLFSSL_CTX * ctx
)
```

この関数は、与えられたコンテキストでの PSA サポートを可能にします。

Parameters:

- **ctx** PSA サポートを有効にする必要がある WOLFSSL_CTX オブジェクトへのポインタ

See: wolfSSL_set_psa_ctx

Return: WOLFSSL_SUCCESS 成功した *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
)
```

与えられた SSL セッションの PSA コンテキストを設定する機能

Parameters:

- **ssl** CTX が有効になる WolfSSL へのポインタ
- **ctx** Struct PSA_SSL_CTX へのポインタ（SSL セッションに固有である必要があります）

See:

- wolfSSL_psa_set_private_key_id
- wolfSSL_psa_free_psa_ctx

Return: WOLFSSL_SUCCESS 成功した *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
)
```

この関数は PSA コンテキストによって使用されるリソースを解放します

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
)
```

この関数は、SSL セッションによって使用される秘密鍵を設定します

Parameters:

- **ctx** 構造体 PSA_SSL_CTX へのポインタ *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);
```

See: wolfSSL_set_psa_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-ja/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) この機能はパスワードベースの鍵導出機能 1（PBKDF1）を実装し、入力パスワードを連結ソルトと共により安全な鍵に変換し、出力に記憶する。これにより、HASH 関数として SHA と MD5 を選択できます。
int	**wc_PBKDF2**(byte * output, const byte * passwd, int pLen, const byte * salt, int sLen, int iterations, int kLen, int typeH) この機能はパスワードベースのキー導出機能 2（PBKDF2）を実装し、入力パスワードを連結されたソルトとともにより安全なキーに変換し、出力に記憶されています。これにより、MD5、SHA、SHA256、SHA384、SHA512、および BLAKE2B など、サポートされている HMAC ハッシュ関数のいずれかを選択できます。

	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) この関数は、RFC 7292 付録 B に記載されているパスワードベースのキー導出機能（PBKDF）を実装しています。この関数は、入力パスワードを連結ソルトでより安全なキーに変換します。それは、MD5、SHA、SHA256、SHA384、SHA512、およびBLAKE2B を含む、ユーザーはサポートされているHMAC ハッシュ関数のいずれかを選択できます。

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
)
```

この機能はパスワードベースの鍵導出機能 1（PBKDF1）を実装し、入力パスワードを連結ソルトと共により安全な鍵に変換し、出力に記憶する。これにより、HASH 関数として SHA と MD5 を選択できます。

Parameters:

- **output** 生成されたキーを保存するバッファへのポインタ。少なくとも klen long になるべきです
- **passwd** キーの派生に使用するパスワードを含むバッファへのポインタ
- **pLen** キーの派生に使用するパスワードの長さ
- **salt** 鍵由来に使用するソルトを含むバッファへのポインター
- **sLen** ソルトの長さ
- **iterations** ハッシュを処理するための回数
- **kLen** 派生キーの希望の長さ。選択したハッシュのダイジェストサイズより長くしてはいけません
  Example

```
int ret;
byte key[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), MD5);
if ( ret != 0 ) {
    // error deriving key from password
}
```

See:

- wc_PBKDF2
- wc_PKCS12_PBKDF

Return:

- 0 入力パスワードからキーの派生に正常に戻された
- BAD_FUNC_ARG 与えられた無効なハッシュタイプがある場合（有効なタイプは：MD5 と SHA）、反復は 1 未満、または要求されたキーの長さ（Klen）は提供されたハッシュのハッシュ長よりも大きいです。
- MEMORY_E SHA または MD5 オブジェクトにメモリを割り当てるエラーがある場合は返されます。

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
)
```

この機能はパスワードベースのキー導出機能 2（PBKDF2）を実装し、入力パスワードを連結されたソルトとともにより安全なキーに変換し、出力に記憶されています。これにより、MD5、SHA、SHA256、SHA384、SHA512、および BLAKE2B など、サポートされている HMAC ハッシュ関数のいずれかを選択できます。

Parameters:

- **output** 生成されたキーを保存するバッファへのポインタ。klen long にするべきです
- **passwd** キーの派生に使用するパスワードを含むバッファへのポインタ
- **pLen** キーの派生に使用するパスワードの長さ
- **salt** 鍵由来に使用するソルトを含むバッファへのポインター
- **sLen** ソルトの長さ
- **iterations** ハッシュを処理するための回数
- **kLen** 派生鍵の望ましい長さ *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),
SHA512);
if ( ret != 0 ) {
    // error deriving key from password
}
```

See:

- wc_PBKDF1
- wc_PKCS12_PBKDF

Return:

- 0 入力パスワードからキーの派生に正常に戻された
- BAD_FUNC_ARG 無効なハッシュタイプがある場合、または反復が 1 未満の場合は返されます。
- MEMORY_E HMAC オブジェクトに割り振りメモリがある場合

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
)
```

この関数は、RFC 7292 付録 B に記載されているパスワードベースのキー導出機能（PBKDF）を実装しています。この関数は、入力パスワードを連結ソルトでより安全なキーに変換します。それは、MD5、SHA、SHA256、SHA384、SHA512、および BLAKE2B を含む、ユーザーはサポートされている HMAC ハッシュ関数のいずれかを選択できます。

Parameters:

- **output** 生成されたキーを保存するバッファへのポインタ。klen long にするべきです
- **passwd** キーの派生に使用するパスワードを含むバッファへのポインタ
- **pLen** キーの派生に使用するパスワードの長さ
- **salt** 鍵由来に使用するソルトを含むバッファへのポインター
- **sLen** ソルトの長さ
- **iterations** ハッシュを処理するための回数
- **kLen** 派生鍵の望ましい長さ
- **hashType** 使用するハッシュアルゴリズム有効な選択肢は次のとおりです.MD5、SHA、SHA256、SHA384、SHA512、および BLAKE2B *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), SHA512, 1);
if ( ret != 0 ) {
    // error deriving key from password
}
```

See:

- wc_PBKDF1
- wc_PBKDF2

Return:

- 0 入力パスワードからキーの派生に正常に戻された
- BAD_FUNC_ARG 返された無効なハッシュタイプが与えられた場合、繰り返しは 1 未満、または要求されたキーの長さ（klen）が提供されたハッシュのハッシュ長よりも大きいです。
- MEMORY_E 割り当てメモリがある場合は返されます
- MP_INIT_E キー生成中にエラーがある場合は返却される可能性があります
- MP_READ_E キー生成中にエラーがある場合は返却される可能性があります
- MP_CMP_E キー生成中にエラーがある場合は返却される可能性があります
- MP_INVMOD_E キー生成中にエラーがある場合は返却される可能性があります
- MP_EXPTMOD_E キー生成中にエラーがある場合は返却される可能性があります
- MP_MOD_E キー生成中にエラーがある場合は返却される可能性があります
- MP_MUL_E キー生成中にエラーがある場合は返却される可能性があります

- MP_ADD_E キー生成中にエラーがある場合は返却される可能性があります
- MP_MULMOD_E キー生成中にエラーがある場合は返却される可能性があります
- MP_TO_E キー生成中にエラーがある場合は返却される可能性があります
- MP_MEM キー生成中にエラーがある場合は返却される可能性があります

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-ja/quic.h

C.41.1 Functions

	Name
int	**wolfSSL_CTX_set_quic_method**(WOLFSSL_CTX * ctx, const WOLFSSL_QUIC_METHOD * quic_method)WOLFSSL_CTX および派生したすべての WOLFSSL インスタンスに対して QUIC プロトコルを有効にします 必要な 4 つのコールバックを提供します。CTX は TLSv1.3 である必要があります。
int	**wolfSSL_set_quic_method**(WOLFSSL * ssl, const WOLFSSL_QUIC_METHOD * quic_method) を提供して、WOLFSSL インスタンスの QUIC プロトコルを有効にします。4 つのコールバックが必要です。WOLFSSL は TLSv1.3 である必要があります。
int	**wolfSSL_is_quic**(WOLFSSL * ssl)QUIC が WOLFSSL インスタンスでアクティブ化されているかどうかを確認します。
WOLFSSL_ENCRYPTION_LEVEL	**wolfSSL_quic_read_level**(const WOLFSSL * ssl) 現在使用中の読み取りの暗号化レベルを決定します。場合にのみ意味があります。WOLFSSL インスタンスは QUIC を使用しています。
WOLFSSL_ENCRYPTION_LEVEL	**wolfSSL_quic_write_level**(const WOLFSSL * ssl) 現在使用中の書き込みの暗号化レベルを決定します。場合にのみ意味があります。WOLFSSL インスタンスは QUIC を使用しています。

	Name
void	**wolfSSL_set_quic_use_legacy_codepoint**(WOLFSSL * ssl, int use_legacy) どの QUIC バージョンを使用するかを設定します。これを呼ばずに、WOLFSSL は両方 (draft_27 と v1) をサーバーに提供します。受け入れる クライアントからの両方と、最新のものをネゴシエートします。
void	**wolfSSL_set_quic_transport_version**(WOLFSSL * ssl, int バージョン) どの QUIC バージョンを使用するかを設定します。
int	**wolfSSL_get_quic_transport_version**(const WOLFSSL * ssl) 構成された QUIC バージョンを取得します。
int	**wolfSSL_set_quic_transport_params**(WOLFSSL * ssl, const uint8_t * params, size_t params_len) 使用する QUIC トランスポート パラメータを設定します。
int	**wolfSSL_get_peer_quic_transport_version**(const WOLFSSL * ssl) ネゴシエートされた QUIC トランスポート バージョンを取得します。これは与えるだけです それぞれの TLS 拡張機能が有効になった後に呼び出されると、意味のある結果が得られます。ピアから見られました。
void	**wolfSSL_get_peer_quic_transport_params**(const WOLFSSL * ssl, const uint8_t ** out_params, size_t * out_params_len) ネゴシエートされた QUIC トランスポート パラメータを取得します。これは与えるだけです それぞれの TLS 拡張機能が有効になった後に呼び出されると、意味のある結果が得られます。ピアから見られました。
void	**wolfSSL_set_quic_early_data_enabled**(WOLFSSL * ssl, int enabled) 初期データを有効にするかどうかを構成します。サーバーがシグナルを送ることを目的としています これをクライアントに。
size_t	**wolfSSL_quic_max_handshake_flight_len**(const WOLFSSL * ssl, WOLFSSL_ENCRYPTION_LEVEL レベル)「飛行中」のデータ量についてアドバイスを得る。未承認 指定された暗号化レベルで。これは WOLFSSL インスタンスのデータ量です バッファする準備ができています。
int	**wolfSSL_provide_quic_data**(WOLFSSL * ssl, WOLFSSL_ENCRYPTION_LEVEL レベル, const uint8_t * data, size_t len) 復号化された CRYPTO データを、さらに処理するために WOLFSSL インスタンスに渡します。通話間の暗号化レベルは、すべて増加することが許可されており、また、暗号化の変更前にデータレコードが完全であることを確認しました レベルが受け入れられます。
WOLFSSL_API int	**wolfSSL_process_quic_post_handshake**(WOLFSSL * ssl) ハンドシェイク後に提供されたすべての CRYPTO レコードを処理します 完了しました。それより前に呼び出すと失敗します。

	Name
int	**wolfSSL_quic_read_write**(WOLFSSL * ssl) ハンドシェイク中またはハンドシェイク後に提供されたすべての CRYPTO レコードを処理します。ハンドシェイクがまだ完了していない場合は進行し、そうでない場合は次のように機能します wolfSSL_process_quic_post_handshake()。
const WOLFSSL_EVP_CIPHER *	**wolfSSL_quic_get_aad**(WOLFSSL * ssl)TLS ハンドシェイクでネゴシエートされた AEAD 暗号を取得します。
int	**wolfSSL_quic_aead_is_gcm**(const WOLFSSL_EVP_CIPHER * aead_cipher)AEAD 暗号が GCM かどうかを確認します。
int	**wolfSSL_quic_aead_is_ccm**(const WOLFSSL_EVP_CIPHER * aead_cipher)AEAD 暗号が CCM かどうかを確認します。
int	**wolfSSL_quic_aead_is_chacha20**(const WOLFSSL_EVP_CIPHER * aead_cipher)AEAD 暗号が CHACHA20 かどうかを確認します。
WOLFSSL_API size_t	**wolfSSL_quic_get_aead_tag_len**(const WOLFSSL_EVP_CIPHER * aead_cipher)AEAD 暗号のタグの長さを決定します。
WOLFSSL_API const WOLFSSL_EVP_MD *	**wolfSSL_quic_get_md**(WOLFSSL * ssl)TLS ハンドシェイクでネゴシエートされたメッセージ ダイジェストを決定します。
const WOLFSSL_EVP_CIPHER *	**wolfSSL_quic_get_hp**(WOLFSSL * ssl)TLS ハンドシェイクでネゴシエートされたヘッダー保護暗号を決定します。
WOLFSSL_EVP_CIPHER_CTX *	**wolfSSL_quic_crypt_new**(const WOLFSSL_EVP_CIPHER * cipher, const uint8_t * key, const uint8_t * iv, int encrypt) 暗号化/復号化のための暗号コンテキストを作成します。
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) 指定されたコンテキストでプレーン テキストを暗号化します。
int	**wolfSSL_quic_aad_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) 指定されたコンテキストで暗号文を復号化します。
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) 擬似乱数キーを抽出します。
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) 疑似ランダム キーを新しいキーに展開します。

	Name
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) 疑似乱数キーを展開して抽出します。

C.41.2 Attributes

	Name
int()(*WOLFSSL* ssl, WOLFSSL_ENCRYPTION_LEVEL レベル, const uint8_t *read_secret, const uint8_t* write_secret, size_t secret_len)	**set_encryption_secrets** ハンドシェイク中にシークレットが生成されたときに呼び出されるコールバック。QUIC プロトコル ハンドラはパケットの暗号化/復号化を実行するため、レベル Early_data/handshake/application のネゴシエートされたシークレットが必要です。
int()(*WOLFSSL* ssl, WOLFSSL_ENCRYPTION_LEVEL レベル, const uint8_t *data, size_t len)	**add_handshake_data** ハンドシェイク CRYPTO データをピアに転送するために呼び出されるコールバック。この方法で転送されるデータは暗号化されません。QUIC の仕事です これを行うためのプロトコル実装。どのシークレットを使用するか指定された暗号化レベルによって決まります。
int()(*WOLFSSL* ssl)	**flush_flight** 送信するデータのアドバイザリ フラッシュのために呼び出されるコールバック。
int()(*WOLFSSL* ssl, WOLFSSL_ENCRYPTION_LEVEL レベル, uint8_t アラート)	**send_alert** 処理中に SSL アラートが発生したときに呼び出されるコールバック。

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
)
```

WOLFSSL_CTX および派生したすべての WOLFSSL インスタンスに対して QUIC プロトコルを有効にします必要な 4 つのコールバックを提供します。CTX は TLSv1.3 である必要があります。

Parameters:

- **ctx** - wolfSSL_CTX_new() を使用して作成された WOLFSSL_CTX 構造体へのポインター。
- **quic_method** - コールバック構造

See:

- wolfSSL_is_quic
- wolfSSL_set_quic_method

Return: WOLFSSL_SUCCESS 成功した場合。

渡された quic_method には、SSL インスタンスよりも長い寿命が必要です。コピーされません。すべてのコールバックを提供する必要があります。

C.41.3.2 function wolfSSL_set_quic_method

```
int wolfSSL_set_quic_method(
    WOLFSSL * ssl,
    const WOLFSSL_QUIC_METHOD * quic_method
)
```

を提供して、WOLFSSL インスタンスの QUIC プロトコルを有効にします。4 つのコールバックが必要です。WOLFSSL は TLSv1.3 である必要があります。

Parameters:

- **ssl** - wolfSSL_new() を使用して作成された WOLFSSL 構造体へのポインタ。
- **quic_method** - コールバック構造

See:

- wolfSSL_is_quic
- wolfSSL_CTX_set_quic_method

Return: WOLFSSL_SUCCESS 成功した場合。

渡された quic_method には、SSL インスタンスよりも長い寿命が必要です。コピーされません。すべてのコールバックを提供する必要があります。

C.41.3.3 function wolfSSL_is_quic

```
int wolfSSL_is_quic(
    WOLFSSL * ssl
)
```

QUIC が WOLFSSL インスタンスでアクティブ化されているかどうかを確認します。

Parameters:

- **ssl** - wolfSSL_new() を使用して作成された WOLFSSL 構造体へのポインタ。

See:

- wolfSSL_CTX_quic_method
- wolfSSL_CTX_set_quic_method

Return: WOLFSSL が QUIC を使用している場合は 1 を返します。

C.41.3.4 function wolfSSL_quic_read_level

```
WOLFSSL_ENCRYPTION_LEVEL wolfSSL_quic_read_level(
    const WOLFSSL * ssl
)
```

現在使用中の読み取りの暗号化レベルを決定します。場合にのみ意味があります。WOLFSSL インスタンスは QUIC を使用しています。

Parameters:

- **ssl** - wolfSSL_new() を使用して作成された WOLFSSL 構造体へのポインタ。

See: wolfSSL_quic_write_level

Return: 暗号化レベル。

有効レベルは、データを返すときは常にパラメーターであることに注意してください。前方へ。ピアからのデータは、これを介して報告される以外のレベルで到着する可能性があります 関数。

C.41.3.5 function wolfSSL_quic_write_level

```
WOLFSSL_ENCRYPTION_LEVEL wolfSSL_quic_write_level(
    const WOLFSSL * ssl
)
```

現在使用中の書き込みの暗号化レベルを決定します。場合にのみ意味があります。WOLFSSL インスタンスは QUIC を使用しています。

Parameters:

- **ssl** - wolfSSL_new() を使用して作成された WOLFSSL 構造体へのポインタ。

See: wolfSSL_quic_read_level

Return: 暗号化レベル。

有効レベルは、データを返すときは常にパラメーターであることに注意してください。前方へ。ピアからのデータは、これを介して報告される以外のレベルで到着する可能性があります 関数。

C.41.3.6 function wolfSSL_set_quic_use_legacy_codepoint

```
void wolfSSL_set_quic_use_legacy_codepoint(
    WOLFSSL * ssl,
    int use_legacy
)
```

どの QUIC バージョンを使用するかを設定します。これを呼ばずに、WOLFSSL は両方 (draft-27 と v1) をサーバーに提供します。受け入れる クライアントからの両方と、最新のものをネゴシエートします。

Parameters:

- **ssl** - wolfSSL_new() を使用して作成された WOLFSSL 構造体へのポインタ。
- **use_legacy** - ドラフト 27 を使用する場合は true、QUICv1 のみを使用する場合は 0。

See: wolfSSL_set_quic_transport_version

Return: WOLFSSL_SUCCESS 成功した場合。

C.41.3.7 function wolfSSL_set_quic_transport_version

```
void wolfSSL_set_quic_transport_version(
    WOLFSSL * ssl,
    int バージョン
)
```

どの QUIC バージョンを使用するかを設定します。

Parameters:

- **ssl** - wolfSSL_new() を使用して作成された WOLFSSL 構造体へのポインタ。
- **version** - QUIC バージョン用に定義された TLS 拡張。

See: wolfSSL_set_quic_use_legacy_codepoint

Return: WOLFSSL_SUCCESS 成功した場合。

C.41.3.8 function wolfSSL_get_quic_transport_version

```
int wolfSSL_get_quic_transport_version(
    const WOLFSSL * ssl
)
```

構成された QUIC バージョンを取得します。

Parameters:

- **ssl** - wolfSSL_new() を使用して作成された WOLFSSL 構造体へのポインタ。

See:

- wolfSSL_set_quic_use_legacy_codepoint
- wolfSSL_set_quic_transport_version

Return: 構成されたバージョンの TLS 拡張。

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
)
```

使用する QUIC トランスポート パラメータを設定します。

Parameters:

- **ssl** - wolfSSL_new() を使用して作成された WOLFSSL 構造体へのポインタ。
- **params** - 使用するパラメータ バイト·param params_len - パラメータの長さ

See:

- wolfSSL_set_quic_use_legacy_codepoint
- wolfSSL_set_quic_transport_version

Return: WOLFSSL_SUCCESS 成功した場合。

C.41.3.10 function wolfSSL_get_peer_quic_transport_version

```
int wolfSSL_get_peer_quic_transport_version(
    const WOLFSSL * ssl
)
```

ネゴシエートされた QUIC トランスポート バージョンを取得します。これは与えるだけです それぞれの TLS 拡張機能が有効になった後に呼び出されると、意味のある結果が得られます。ピアから見られました。

Parameters:

- **ssl** - wolfSSL_new() を使用して作成された WOLFSSL 構造体へのポインタ。

See:

- wolfSSL_set_quic_use_legacy_codepoint
- wolfSSL_set_quic_transport_version

Return: ネゴシエートされたバージョンまたは -1 を返します。

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
)
```

ネゴシエートされた QUIC トランスポート パラメータを取得します。これは与えるだけです それぞれの TLS 拡張機能が有効になった後に呼び出されると、意味のある結果が得られます。ピアから見られました。

Parameters:

- **ssl** - wolfSSL_new() を使用して作成された WOLFSSL 構造体へのポインタ。
- **out_params** - ピアに送信されるパラメーター。利用できない場合は NULL に設定されます。
- **out_params_len** - ピアに送信されるパラメータの長さ。利用できない場合は 0 に設定

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
)
```

初期データを有効にするかどうかを構成します。サーバーがシグナルを送ることを目的としています これをクライアントに。

Parameters:

- **ssl** - wolfSSL_new() を使用して作成された WOLFSSL 構造体へのポインタ。
- **enabled** - != 初期データが有効な場合は 0

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 レベル
)
```

「飛行中」のデータ量についてアドバイスを得る。未承認 指定された暗号化レベルで。これは WOLFSSL インスタンスのデータ量です バッファする準備ができています。

Parameters:

- **ssl** - wolfSSL_new() を使用して作成された WOLFSSL 構造体へのポインタ。
- **level** - 問い合わせる暗号化レベル

Return: 飛行中の推奨最大データを返す

C.41.3.14 function wolfSSL_provide_quic_data

```
int wolfSSL_provide_quic_data(
    WOLFSSL * ssl,
    WOLFSSL_ENCRYPTION_LEVEL レベル,
    const uint8_t * data,
    size_t len
)
```

復号化された CRYPTO データを、さらに処理するために WOLFSSL インスタンスに渡します。通話間の暗号化レベルは、すべて増加することが許可されており、また、暗号化の変更前にデータレコードが完全であることを確認しました レベルが受け入れられます。

Parameters:

- **ssl** - wolfSSL_new() を使用して作成された WOLFSSL 構造体へのポインタ。
- **level** - データが暗号化されたレベル
- **data** - データ自体

- **len** - データの長さ

See: wolfSSL_process_quic_post_handshake

Return: WOLFSSL_SUCCESS 成功した場合。

C.41.3.15 function wolfSSL_process_quic_post_handshake

```
WOLFSSL_API int wolfSSL_process_quic_post_handshake(
    WOLFSSL * ssl
)
```

ハンドシェイク後に提供されたすべての CRYPTO レコードを処理します 完了しました。それより前に呼び出すと失敗します。

Parameters:

- **ssl** - wolfSSL_new() を使用して作成された WOLFSSL 構造体へのポインタ。

See:

- wolfSSL_provide_quic_data
- wolfSSL_quic_read_write
- wolfSSL_accept
- wolfSSL_connect

Return: WOLFSSL_SUCCESS 成功した場合。

C.41.3.16 function wolfSSL_quic_read_write

```
int wolfSSL_quic_read_write(
    WOLFSSL * ssl
)
```

ハンドシェイク中またはハンドシェイク後に提供されたすべての CRYPTO レコードを処理します。ハンドシェイクがまだ完了していない場合は進行し、そうでない場合は次のように機能します wolfSSL_process_quic_post_handshake()。

Parameters:

- **ssl** - wolfSSL_new() を使用して作成された WOLFSSL 構造体へのポインタ。

See:

- wolfSSL_provide_quic_data
- wolfSSL_quic_read_write
- wolfSSL_accept
- wolfSSL_connect

Return: WOLFSSL_SUCCESS 成功した場合。

C.41.3.17 function wolfSSL_quic_get_aad

```
const WOLFSSL_EVP_CIPHER * wolfSSL_quic_get_aad(
    WOLFSSL * ssl
)
```

TLS ハンドシェイクでネゴシエートされた AEAD 暗号を取得します。

Parameters:

- **ssl** - wolfSSL_new() を使用して作成された WOLFSSL 構造体へのポインタ。

See:

- wolfSSL_quic_aad_is_gcm
- wolfSSL_quic_aad_is_ccm
- wolfSSL_quic_aad_is_chacha20
- wolfSSL_quic_get_aad_tag_len
- wolfSSL_quic_get_md
- wolfSSL_quic_get_hp
- wolfSSL_quic_crypt_new
- wolfSSL_quic_aad_encrypt
- wolfSSL_quic_aad_decrypt

Return: ネゴシエートされた暗号、または決定されない場合は NULL を返します。

C.41.3.18 function wolfSSL_quic_aead_is_gcm

```
int wolfSSL_quic_aead_is_gcm(
    const WOLFSSL_EVP_CIPHER * aead_cipher
)
```

AEAD 暗号が GCM かどうかを確認します。

Parameters:

- **cipher** - 暗号

See:

- wolfSSL_quic_get_aad
- wolfSSL_quic_aad_is_ccm
- wolfSSL_quic_aad_is_chacha20
- wolfSSL_quic_get_aad_tag_len
- wolfSSL_quic_get_md
- wolfSSL_quic_get_hp
- wolfSSL_quic_crypt_new
- wolfSSL_quic_aad_encrypt
- wolfSSL_quic_aad_decrypt

Return: != 0 (AEAD 暗号が GCM の場合)。

C.41.3.19 function wolfSSL_quic_aead_is_ccm

```
int wolfSSL_quic_aead_is_ccm(
    const WOLFSSL_EVP_CIPHER * aead_cipher
)
```

AEAD 暗号が CCM かどうかを確認します。

Parameters:

- **cipher** - 暗号

See:

- wolfSSL_quic_get_aad
- wolfSSL_quic_aad_is_gcm
- wolfSSL_quic_aad_is_chacha20
- wolfSSL_quic_get_aad_tag_len
- wolfSSL_quic_get_md
- wolfSSL_quic_get_hp
- wolfSSL_quic_crypt_new

- wolfSSL_quic_aad_encrypt
- wolfSSL_quic_aad_decrypt

Return: != 0 AEAD 暗号が CCM の場合。

C.41.3.20 function wolfSSL_quic_aead_is_chacha20

```
int wolfSSL_quic_aead_is_chacha20(
    const WOLFSSL_EVP_CIPHER * aead_cipher
)
```

AEAD 暗号が CHACHA20 かどうかを確認します。

Parameters:

- **cipher** - 暗号

See:

- wolfSSL_quic_get_aad
- wolfSSL_quic_aad_is_ccm
- wolfSSL_quic_aad_is_gcm
- wolfSSL_quic_get_aad_tag_len
- wolfSSL_quic_get_md
- wolfSSL_quic_get_hp
- wolfSSL_quic_crypt_new
- wolfSSL_quic_aad_encrypt
- wolfSSL_quic_aad_decrypt

Return: != 0 は、AEAD 暗号が CHACHA20 の場合です。

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
)
```

AEAD 暗号のタグの長さを決定します。

Parameters:

- **cipher** - 暗号

See: wolfSSL_quic_get_aad

Return: AEAD 暗号のタグ長。

C.41.3.22 function wolfSSL_quic_get_md

```
WOLFSSL_API const WOLFSSL_EVP_MD * wolfSSL_quic_get_md(
    WOLFSSL * ssl
)
```

TLS ハンドシェイクでネゴシエートされたメッセージ ダイジェストを決定します。

Parameters:

- **ssl** - wolfSSL_new() を使用して作成された WOLFSSL 構造体へのポインタ。

See:

- wolfSSL_quic_get_aad
- wolfSSL_quic_get_hp

Return: TLS ハンドシェイクでネゴシエートされたメッセージ ダイジェストを返す

C.41.3.23 function wolfSSL_quic_get_hp

```
const WOLFSSL_EVP_CIPHER * wolfSSL_quic_get_hp(
    WOLFSSL * ssl
)
```

TLS ハンドシェイクでネゴシエートされたヘッダー保護暗号を決定します。

Parameters:

- **ssl** - wolfSSL_new() を使用して作成された WOLFSSL 構造体へのポインタ。

See:

- wolfSSL_quic_get_aad
- wolfSSL_quic_get_md

Return: TLS ハンドシェイクでネゴシエートされたヘッダー保護暗号を返します

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
)
```

暗号化/復号化のための暗号コンテキストを作成します。

Parameters:

- **cipher** - コンテキストで使用する暗号。
- **key** - コンテキストで使用するキー。
- **iv** - コンテキストで使用する iv。
- **encrypt** - 暗号化の場合は!= 0、それ以外の場合は復号化

See:

- wolfSSL_quic_get_aad
- wolfSSL_quic_get_hp
- wolfSSL_quic_aad_encrypt
- wolfSSL_quic_aad_decrypt

Return: エラーの場合は、作成されたコンテキストまたは NULL を返します。

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
)
```

指定されたコンテキストでプレーン テキストを暗号化します。

Parameters:

- **dest** - 暗号化されたデータの書き込み先
- **aead_ctx** - 使用する暗号コンテキスト
- **plain** - 暗号化するプレーン データ
- **plainlen** - プレーン データの長さ
- **iv** - 使用する iv
- **aad** - 使用する追加
- **aadlen** - aad の長さ

See:

- wolfSSL_quic_get_aad
- wolfSSL_quic_get_hp
- wolfSSL_quic_crypt_new
- wolfSSL_quic_aad_decrypt

Return: WOLFSSL_SUCCESS 成功した場合。

C.41.3.26 function wolfSSL_quic_aad_decrypt

```
int wolfSSL_quic_aad_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
)
```

指定されたコンテキストで暗号文を復号化します。

Parameters:

- **dest** - プレーンテキストの書き込み先
- **ctx** - 使用する暗号コンテキスト
- **enc** - 復号化する暗号化データ
- **envlen** - 暗号化されたデータの長さ
- **iv** - 使用する iv
- **aad** - 使用する追加
- **aadlen** - aad の長さ

See:

- wolfSSL_quic_get_aad
- wolfSSL_quic_get_hp
- wolfSSL_quic_crypt_new
- wolfSSL_quic_aad_encrypt

Return: WOLFSSL_SUCCESS 成功した場合。

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
)
```

擬似乱数キーを抽出します。

Parameters:

- **dest** - キーの書き込み先
- **md** - 使用するメッセージ ダイジェスト
- **secret** - 使用するシークレット
- **secretlen** - シークレットの長さ
- **salt** - 使用するソルト
- **saltlen** - ソルトの長さ

See:

- wolfSSL_quic_hkdf_expand
- wolfSSL_quic_hkdf

Return: WOLFSSL_SUCCESS 成功した場合。

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
)
```

疑似ランダム キーを新しいキーに展開します。

Parameters:

- **dest** - キーの書き込み先
- **destlen** - 展開するキーの長さ
- **md** - 使用するメッセージ ダイジェスト
- **secret** - 使用するシークレット
- **secretlen** - シークレットの長さ
- **info** - 使用する情報
- **infolen** - 情報の長さ

See:

- wolfSSL_quic_hkdf_extract
- wolfSSL_quic_hkdf

Return: WOLFSSL_SUCCESS 成功した場合。

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
)
```

疑似乱数キーを展開して抽出します。

Parameters:

- **dest** - キーの書き込み先
- **destlen** - キーの長さ
- **md** - 使用するメッセージ ダイジェスト
- **secret** - 使用するシークレット
- **secretlen** - シークレットの長さ
- **salt** - 使用するソルト
- **saltlen** - ソルトの長さ
- **info** - 使用する情報
- **infolen** - 情報の長さ

See:

- wolfSSL_quic_hkdf_extract
- wolfSSL_quic_hkdf_expand

Return: WOLFSSL_SUCCESS 成功した場合。

C.41.4 Attributes Documentation

C.41.4.1 variable set_encryption_secrets

```
int(*)(WOLFSSL *ssl, WOLFSSL_ENCRYPTION_LEVEL レベル, const uint8_t
 ↪  *read_secret, const uint8_t *write_secret, size_t secret_len)
 ↪  set_encryption_secrets;
```

ハンドシェイク中にシークレットが生成されたときに呼び出されるコールバック。QUIC プロトコル ハンドラはパケットの暗号化/復号化を実行するため、レベル Early_data/handshake/application のネゴシエートされたシークレットが必要です。

Parameters:

- **ssl** - wolfSSL_new() を使用して作成された WOLFSSL 構造体へのポインタ。
- **level** - シークレットの暗号化レベル
- **read_secret** - 特定のレベルで復号化に使用されるシークレット。NULL の場合があります。
- **write_secret** - 特定のレベルで暗号化に使用されるシークレット。NULL の場合があります。
- **secret_len** - シークレットの長さ

See: wolfSSL_set_quic_method

Return: 成功すると 1 を返し、失敗すると 0 を返します。

コールバックは、ハンドシェイク中に数回呼び出されます。両方のどちらか または、読み取りシークレットまたは書き込みシークレットのみが提供される場合があります。これは、指定された暗号化レベルはすでに有効になっています。

C.41.4.2 variable add_handshake_data

```
int(*)(WOLFSSL *ssl, WOLFSSL_ENCRYPTION_LEVEL レベル, const uint8_t *data,
 ↪  size_t len) add_handshake_data;
```

ハンドシェイク CRYPTO データをピアに転送するために呼び出されるコールバック。この方法で転送されるデータは暗号化されません。QUIC の仕事です これを行うためのプロトコル実装。どのシークレットを使用するか 指定された暗号化レベルによって決まります。

Parameters:

- **ssl** - wolfSSL_new() を使用して作成された WOLFSSL 構造体へのポインタ。
- **level** - データの暗号化に使用する暗号化レベル
- **data** - データ自体
- **len** - データの長さ

See: wolfSSL_set_quic_method

Return: 成功すると 1 を返し、失敗すると 0 を返します。

このコールバックは、ハンドシェイク中またはポスト ハンドシェイク中に数回呼び出される場合があります。処理。データは完全な CRYPTO レコードをカバーする場合がありますが、部分的であること。ただし、コールバックは以前にすべてのレコード データを受信しています。別の暗号化レベルを使用しています。

C.41.4.3 variable flush_flight

```
int(*)(WOLFSSL *ssl) flush_flight;
```

送信するデータのアドバイザリ フラッシュのために呼び出されるコールバック。

Parameters:

- **ssl** - wolfSSL_new() を使用して作成された WOLFSSL 構造体へのポインタ。

See: wolfSSL_set_quic_method

Return: 成功すると 1 を返し、失敗すると 0 を返します。

C.41.4.4 variable send_alert

```
int(*)(WOLFSSL *ssl, WOLFSSL_ENCRYPTION_LEVEL レベル, uint8_t アラート)
 ↪  send_alert;
```

処理中に SSL アラートが発生したときに呼び出されるコールバック。

Parameters:

- **ssl** - wolfSSL_new() を使用して作成された WOLFSSL 構造体へのポインタ。
- **level** - アラートが発生したときに有効だった暗号化レベル
- **alert** - エラー

See: wolfSSL_set_quic_method

Return: 成功すると 1 を返し、失敗すると 0 を返します。

C.41.5 Source code

```
int (*set_encryption_secrets)(WOLFSSL *ssl, WOLFSSL_ENCRYPTION_LEVEL レベル,
                              const uint8_t *read_secret,
                              const uint8_t *write_secret, size_t secret_len);

int (*add_handshake_data)(WOLFSSL *ssl, WOLFSSL_ENCRYPTION_LEVEL レベル,
                          const uint8_t *data, size_t len);

int (*flush_flight)(WOLFSSL *ssl);
```

```
int (*send_alert)(WOLFSSL *ssl, WOLFSSL_ENCRYPTION_LEVEL レベル, uint8_t アラー
 ↪  ト);

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 バージョン);

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 レベル);


int wolfSSL_provide_quic_data(WOLFSSL *ssl, WOLFSSL_ENCRYPTION_LEVEL レベル,
 ↪  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_aad(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_aad_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-ja/random.h

C.42.1 Functions

	Name
int	**wc_InitNetRandom**(const char * configFile, wnr_hmac_key hmac_cb, int timeout)Init Global WhiteWood Netrandom のコンテキスト
int	**wc_FreeNetRandom**(void) 無料の Global WhiteWood Netrandom コンテキスト。
int	**wc_InitRng**(WC_RNG *)RNG のシード（OS から）とキー暗号を取得します。割り当てられた RNG-> DRBG（決定論的ランダムビットジェネレータ）が割り当てられます（WC_FREERNG で割り当てられている必要があります）。これはブロッキング操作です。

	Name
int	**wc_RNG_GenerateBlock**(WC_RNG * rng, byte * b, word32 sz) 疑似ランダムデータの SZ バイトを出力にコピーします。必要に応じて RNG（ブロッキング）します。
WC_RNG *	**wc_rng_new**(byte * nonce, word32 nonceSz, void * heap) 新しい WC_RNG 構造を作成します。
int	**wc_FreeRng**(WC_RNG *)RNG が DRGB を安全に解放するために必要なときに呼び出されるべきです。ゼロと Xfrees RNG-DRBG。
WC_RNG *	**wc_rng_free**(WC_RNG * rng)RNG を安全に自由に解放するために RNG が不要になったときに呼び出されるべきです。*Example*
int	**wc_RNG_HealthTest**(int reseed, const byte * entropyA, word32 entropyASz, const byte * entropyB, word32 entropyBSz, byte * output, word32 outputSz)DRBG の機能を作成しテストします。

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 のコンテキスト

Parameters:

- **configFile** 設定ファイルへのパス
- **hmac_cb** HMAC コールバックを作成するにはオプションです。*Example*

```
char* config = "path/to/config/example.conf";
int time = // Some sufficient timeout value;

if (wc_InitNetRandom(config, NULL, time) != 0)
{
    // Some error occurred
}
```

See: wc_FreeNetRandom

Return:

- 0 成功
- BAD_FUNC_ARG configfile が null またはタイムアウトのどちらかが否定的です。

- RNG_FAILURE_E RNG の初期化に失敗しました。

C.42.3.2 function wc_FreeNetRandom

```
int wc_FreeNetRandom(
    void
)
```

無料の Global WhiteWood Netrandom コンテキスト。

See: wc_InitNetRandom

Return:

- 0 成功
- BAD_MUTEX_E Wnr_Mutex でミューテックスをロックするエラー *Example*

```
int ret = wc_FreeNetRandom();
if(ret != 0)
{
    // Handle the error
}
```

C.42.3.3 function wc_InitRng

```
int wc_InitRng(
    WC_RNG *
)
```

RNG のシード（OS から）とキー暗号を取得します。割り当てられた RNG-> DRBG（決定論的ランダムビットジェネレータ）が割り当てられます（WC_FREERNG で割り当てられている必要があります）。これはブロッキング操作です。

See:

- wc_InitRngCavium
- wc_RNG_GenerateBlock
- wc_RNG_GenerateByte
- wc_FreeRng
- wc_RNG_HealthTest

Return:

- 0 成功しています。
- MEMORY_E XMalloc に失敗しました
- WINCRYPT_E WC_GENERATSEED：コンテキストの取得に失敗しました
- CRYPTGEN_E WC_GENERATSEED：ランダムになりました
- BAD_FUNC_ARG WC_RNG_GenerateBlock 入力は NULL または SZ が MAX_REQUEST_LEN を超えています
- DRBG_CONT_FIPS_E wc_rng_generateblock：hash_gen は drbg_cont_failure を返しました
- RNG_FAILURE_E wc_rng_generateBlock：デフォルトエラーです。RNG のステータスはもともと OK ではなく、drbg_failed に設定されています *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
)
```

疑似ランダムデータの SZ バイトを出力にコピーします。必要に応じて RNG（ブロッキング）します。

Parameters:

- **rng** 乱数発生器は WC_INITRNG で初期化された
- **output** ブロックがコピーされるバッファ *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!
}
```

See:

- wc_InitRngCavium, wc_InitRng
- wc_RNG_GenerateByte
- wc_FreeRng
- wc_RNG_HealthTest

Return:

- 0 成功した
- BAD_FUNC_ARG 入力は NULL または SZ が MAX_REQUEST_LEN を超えています
- DRBG_CONT_FIPS_E hash_gen は drbg_cont_failure を返しました
- RNG_FAILURE_E デフォルトのエラー RNG のステータスはもともと OK ではなく、drbg_failed に設定されています

C.42.3.5 function wc_rng_new

```
WC_RNG * wc_rng_new(
    byte * nonce,
    word32 nonceSz,
```

```
    void * heap
)
```

新しい WC_RNG 構造を作成します。

Parameters:

- **heap** ヒープ識別子へのポインタ
- **nonce** nonce を含むバッファへのポインタ *Example*

```
RNG  rng;
byte nonce[] = { initialize nonce };
word32 nonceSz = sizeof(nonce);

wc_rng_new(&nonce, nonceSz, &heap);
```

- **rng** 乱数発生器は WC_INITRNG で初期化された *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!
}
```

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 成功の構造
- NULL 誤りに
- 0 成功した
- BAD_FUNC_ARG 入力は NULL または SZ が MAX_REQUEST_LEN を超えています
- DRBG_CONT_FIPS_E hash_gen は drbg_cont_failure を返しました
- RNG_FAILURE_E デフォルトのエラー RNG のステータスはもともと OK ではなく、drbg_failed に設定されています

wc_rng_generateBlock を呼び出して、疑似ランダムデータのバイトを b にコピーします。必要に応じて RNG が再販されます。

C.42.3.6 function wc_FreeRng

```
int wc_FreeRng(
    WC_RNG *
)
```

RNG が DRGB を安全に解放するために必要なときに呼び出されるべきです。ゼロと Xfrees RNG-DRBG。

See:

- wc_InitRngCavium
- wc_InitRng
- wc_RNG_GenerateBlock
- wc_RNG_GenerateByte,
- wc_RNG_HealthTest

Return:

- 0 成功した
- BAD_FUNC_ARG RNG または RNG-> DRGB NULL
- RNG_FAILURE_E DRBG の割り当て解除に失敗しました *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
)
```

RNG を安全に自由に解放するために RNG が不要になったときに呼び出されるべきです。*Example*

See:

- wc_InitRng
- wc_rng_new
- wc_FreeRng
- wc_RNG_HealthTest

```
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
)
```

DRBG の機能を作成しテストします。

Parameters:

- **int** RESEED：設定されている場合は、Reseed 機能をテストします
- **entropyA** DRGB をインスタンス化するエントロピー
- **entropyASz** バイト数のエントロピヤのサイズ
- **entropyB** Reseed Set を設定した場合、DRBG は Entropyb でリサイードされます
- **entropyBSz** バイト単位の Entropyb のサイズ
- **output** SEADRANDOM が設定されている場合は、Entropyb に播種されたランダムなデータに初期化され、それ以外の場合は Entropya *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
```

See:

- wc_InitRngCavium
- wc_InitRng
- wc_RNG_GenerateBlock
- wc_RNG_GenerateByte
- wc_FreeRng

Return:

- 0 成功した
- BAD_FUNC_ARG ELTOPYA と出力は NULL にしないでください。Reseed Set Entropyb が NULL でなければならない場合
- -1 テスト失敗

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-ja/ripemd.h

C.43.1 Functions

	Name
int	**wc_InitRipeMd**(RipeMd *) この関数は、RIPemd のダイジェスト、バッファ、LOLEN ,HILEN を初期化することによって RIPemd 構造を初期化します。
int	**wc_RipeMdUpdate**(RipeMd * ripemd, const byte * data, word32 len) この関数はデータ入力の RIPemd ダイジェストを生成し、結果を RIPemd-> Digest バッファに格納します。WC_RIPEMDUPDATE を実行した後、生成された RIPemd-> Digest を既知の認証タグに比較してメッセージの信頼性を比較する必要があります。

	Name
int	**wc_RipeMdFinal**(RipeMd * ripemd, byte * hash) この関数は計算されたダイジェストをハッシュにコピーします。無傷のブロックがある場合、この方法ではブロックを 0S でパッケージし、ハッシュにコピーする前にそのブロックのラウンドをダイジェストに含めます。RIPEMD の状態がリセットされます。

C.43.2 Functions Documentation

C.43.2.1 function wc_InitRipeMd

```
int wc_InitRipeMd(
    RipeMd *
)
```

この関数は、RIPemd のダイジェスト、バッファ、LOLEN ,HILEN を初期化することによって RIPemd 構造を初期化します。

See:

- wc_RipeMdUpdate
- wc_RipeMdFinal

Return:

- 0 機能の実行に成功したことに戻ります。RIPEMD 構造が初期化されます。
- BAD_FUNC_ARG RIPEMD 構造が NULL の場合に返されます。*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
)
```

この関数はデータ入力の RIPemd ダイジェストを生成し、結果を RIPemd-> Digest バッファに格納します。WC_RIPEMDUPDATE を実行した後、生成された RIPemd-> Digest を既知の認証タグに比較してメッセージの信頼性を比較する必要があります。

Parameters:

- **ripemd** WC_INTRIPEMD で初期化される RIPEMD 構造へのポインタ
- **data** ハッシュするデータ *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 …
```

See:

- wc_InitRipeMd
- wc_RipeMdFinal

Return:

- 0 機能の実行に成功したことに戻ります。
- BAD_FUNC_ARG RIPEMD 構造が NULL の場合、またはデータが NULL で、LEN がゼロでない場合に返されます。データが NULL であり、LEN が 0 の場合、この関数は実行されるはずです。

C.43.2.3 function wc_RipeMdFinal

```
int wc_RipeMdFinal(
    RipeMd * ripemd,
    byte * hash
)
```

この関数は計算されたダイジェストをハッシュにコピーします。無傷のブロックがある場合、この方法ではブロックを 0S でパッケージし、ハッシュにコピーする前にそのブロックのラウンドをダイジェストに含めます。RIPEMD の状態がリセットされます。

Parameters:

- **ripemd** WC_INITRIPEMD で初期化する RIPEMD 構造へのポインタ、および WC_RIPEMDUPDATE からハッシュを含む。状態はリセットされます *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.
}...
```

See: none

Return:

- 0 機能の実行に成功したことに戻ります。RIPEMD 構造の状態がリセットされました。
- BAD_FUNC_ARG RIPEMD 構造体またはハッシュパラメータが NULL の場合に返されます。

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-ja/rsa.h

C.44.1 Functions

	Name
int	**wc_InitRsaKey**(RsaKey * key, void * heap) この関数は提供された RsaKey 構造体を初期化します。また、ユーザー定義メモリオーバーライドで使用するためのヒープ識別子も取ります（XMALLOC、XFREE、XREALLOC を参照）。wc_rsa_blinding が有効な場合、キーは WC_RSASETRNG によって RNG に関連付けられなければなりません。
int	**wc_InitRsaKey_Id**(RsaKey * key, unsigned char * id, int len, void * heap, int devId) この関数は提供された RsaKey 構造体を初期化します。ID と LEN は、DEVID がデバイスを識別している間にデバイス上のキーを識別するために使用されます。また、ユーザー定義メモリオーバーライドで使用するためのヒープ識別子も取ります（XMALLOC、XFREE、XREALLOC を参照）。wc_rsa_blinding が有効な場合、キーは WC_RSASETRNG によって RNG に関連付けられなければなりません。
int	**wc_RsaSetRNG**(RsaKey * key, WC_RNG * rng) この関数は RNG をキーに関連付けます。WC_RSA_BLINDING が有効になっている場合は必要です。
int	**wc_FreeRsaKey**(RsaKey * key) この関数は、MP_Clear を使用して提供された RsaKey 構造体を解放します。
int	**wc_RsaPublicEncrypt**(const byte * in, word32 inLen, byte * out, word32 outLen, RsaKey * key, WC_RNG * rng) この関数はメッセージを IN から暗号化し、その結果を格納します。初期化された公開鍵と乱数発生器が必要です。副作用として、この関数は ounlen の中で書き込まれたバイトを返します。
int	**wc_RsaPrivateDecryptInline**(byte * in, word32 inLen, byte ** out, RsaKey * key) この関数は復号化のために WC_RSAPrivateCrypt 関数によって利用されます。
int	**wc_RsaPrivateDecrypt**(const byte * in, word32 inLen, byte * out, word32 outLen, RsaKey * key) この関数は秘密の RSA 復号化を提供します。
int	**wc_RsaSSL_Sign**(const byte * in, word32 inLen, byte * out, word32 outLen, RsaKey * key, WC_RNG * rng) 提供された配列に秘密鍵と署名します。

	Name
int	**wc_RsaSSL_VerifyInline**(byte * in, word32 inLen, byte ** out, RsaKey * key) メッセージが RSA キーによって署名されたことを確認するために使用されます。出力は入力と同じバイト配列を使用します。
int	**wc_RsaSSL_Verify**(const byte * in, word32 inLen, byte * out, word32 outLen, RsaKey * key) メッセージがキーによって署名されたことを確認するために使用されます。
int	**wc_RsaPSS_Sign**(const byte * in, word32 inLen, byte * out, word32 outLen, enum wc_HashType hash, int mgf, RsaKey * key, WC_RNG * rng) 提供された配列に秘密鍵と署名します。
int	**wc_RsaPSS_Verify**(byte * in, word32 inLen, byte * out, word32 outLen, enum wc_HashType hash, int mgf, RsaKey * key) 入力署名を復号して、メッセージが鍵によって署名されたことを確認します。WC_RSA_BLINDING が有効な場合、鍵は wc_RsaSetRNG によって RNG に関連付けられなければなりません。
int	**wc_RsaPSS_VerifyInline**(byte * in, word32 inLen, byte ** out, enum wc_HashType hash, int mgf, RsaKey * key) 入力署名を復号化して、メッセージが RSA キーによって署名されたことを確認します。出力は入力と同じバイト配列を使用します。WC_RSA_BLINDING が有効な場合、キーは WC_RSASETRNG によって RNG に関連付けられなければなりません。
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)RSA-PSS で署名されたメッセージを確認してください。ソルトの長さはハッシュ長に等しい。WC_RSA_BLINDING が有効な場合、キーは WC_RSASETRNG によって RNG に関連付けられなければなりません。
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)RSA-PSS で署名されたメッセージを確認してください。WC_RSA_BLINDING が有効な場合、キーは WC_RSASETRNG によって RNG に関連付けられなければなりません。

	Name
int	**wc_RsaPSS_VerifyCheckInline**(byte * in, word32 inLen, byte ** out, const byte * digest, word32 digentLen, enum wc_HashType hash, int mgf, RsaKey * key)RSA-PSS で署名されたメッセージを確認してください。入力バッファは出力バッファに再利用されます。ソルトの長さはハッシュ長に等しい。WC_RSA_BLINDING が有効な場合、キーは WC_RSASETRNG によって RNG に関連付けられなければなりません。
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)RSA-PSS で署名されたメッセージを確認してください。入力バッファは出力バッファに再利用されます。WC_RSA_BLINDING が有効な場合、キーは WC_RSASETRNG によって RNG に関連付けられなければなりません。
int	**wc_RsaPSS_CheckPadding**(const byte * in, word32 inLen, byte * sig, word32 sigSz, enum wc_HashType hashType)PSS データを確認して、署名が一致するようにします。ソルトの長さはハッシュ長に等しい。WC_RSA_BLINDING が有効な場合、キーは WC_RSASETRNG によって RNG に関連付けられなければなりません。
int	**wc_RsaPSS_CheckPadding_ex**(const byte * in, word32 inLen, byte * sig, word32 sigSz, enum wc_HashType hashType, int saltLen, int bits)PSS データを確認して、署名が一致するようにします。ソルトの長さはハッシュ長に等しい。
int	**wc_RsaEncryptSize**(RsaKey * key) 提供されたキー構造の暗号化サイズを返します。
int	**wc_RsaPrivateKeyDecode**(const byte * input, word32 * inOutIdx, RsaKey * key, word32 inSz) この関数は Der フォーマットされた RSA 秘密鍵を解析し、秘密鍵を抽出し、それを与えられた Resakey 構造に格納します。IDX に解析された距離も設定します。
int	**wc_RsaPublicKeyDecode**(const byte * input, word32 * inOutIdx, RsaKey * key, word32 inSz) この関数は Der フォーマットの RSA 公開鍵を解析し、公開鍵を抽出し、それを指定された RsaKey 構造体に格納します。IDX に解析された距離も設定します。
int	**wc_RsaPublicKeyDecodeRaw**(const byte * n, word32 nSz, const byte * e, word32 eSz, RsaKey * key) この関数は、公開弾性率（n）と指数（e）を撮影して、RSA 公開鍵の生の要素を復号します。これらの生の要素を提供された RsaKey 構造体に格納し、暗号化/復号化プロセスで使用することができます。

	Name
int	**wc_RsaKeyToDer**(RsaKey * key, byte * output, word32 inLen) この機能は RSAKEY キーを DER フォーマットに変換します。結果は出力に書き込まれ、書き込まれたバイト数を返します。
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) この機能は、どのパディングを使用するかを選択しながら RSA 暗号化を実行します。
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) この関数は RSA を使用してメッセージを復号化し、どのパディングタイプのオプションを指定します。
int	**wc_RsaPrivateDecryptInline_ex**(byte * in, word32 inLen, byte ** out, RsaKey * key, int type, enum wc_HashType hash, int mgf, byte * label, word32 labelSz) この関数は RSA を使用してメッセージをインラインで復号化し、どのパディングタイプのオプションを示します。IN バッファには、呼び出された後に復号化されたメッセージが含まれ、アウトバイトポインタはプレーンテキストがある「IN」バッファ内の場所を指します。
int	**wc_RsaFlattenPublicKey**(RsaKey * key, byte * e, word32 * eSz, byte * n, word32 * nSz)RSA アルゴリズムに使用される RsaKey 構造体の個々の要素（E、N）をバッファに取り出します。
int	**wc_RsaKeyToPublicDer**(RsaKey * key, byte * output, word32 inLen)RSA 公開鍵を DER フォーマットに変換します。出力に書き込み、書き込まれたバイト数を返します。
int	**wc_RsaKeyToPublicDer_ex**(RsaKey * key, byte * output, word32 inLen, int with_header)RSA 公開鍵を DER フォーマットに変換します。出力に書き込み、書き込まれたバイト数を返します。with_header が 0 の場合（seq + n + e）だけが ASN.1 Der フォーマットで返され、ヘッダーを除外します。

	Name
int	**wc_MakeRsaKey**(RsaKey * key, int size, long e, WC_RNG * rng) この関数は、長さサイズ（ビット単位）の RSA 秘密鍵を生成し、指数（e）を指定します。次に、このキーを提供された RsaKey 構造体に格納するため、暗号化/復号化に使用できます。E に使用するセキュア番号は 65537 です。サイズは、RSA_MIN_SIZE よりも大きく、RSA_MAX_SIZE よりも大きくなる必要があります。この機能が利用可能であるため、コンパイル時にオプション wolfssl_key_gen を有効にする必要があります。これは、- を使用してください./configure を使用する場合は、-enable-keygen で実現できます。
int	**wc_RsaSetNonBlock**(RsaKey * key, RsaNb * nb) この関数は、ブロックされていない RSA コンテキストを設定します。RSANB コンテキストが設定されている場合、RSA 関数を多くの小さな操作に分割する高速数学ベースの非ブロッキング EXPTMOD が可能になります。wc_rsa_nonblock が定義されているときに有効になっています。
int	**wc_RsaSetNonBlockTime**(RsaKey * key, word32 maxBlockUs, word32 cpuMHz) この関数は最大ブロック時間の最大ブロック時間をマイクロ秒単位で設定します。それは、メガヘルツの CPU 速度と共に事前計算されたテーブル（TFM.cexptModnbinst を参照）を使用して、提供された最大ブロック時間内に次の動作を完了できるかどうかを判断します。wc_rsa_nonblock_time が定義されているときに有効になります。

C.44.2 Functions Documentation

C.44.2.1 function wc_InitRsaKey

```
int wc_InitRsaKey(
    RsaKey * key,
    void * heap
)
```

この関数は提供された RsaKey 構造体を初期化します。また、ユーザー定義メモリオーバーライドで使用するためのヒープ識別子も取ります（XMALLOC、XFREE、XREALLOC を参照）。wc_rsa_blinding が有効な場合、キーは WC_RSASETRNG によって RNG に関連付けられなければなりません。

Parameters:

- **key** 初期化する RSAKEY 構造へのポインタ *Example*

```
RsaKey enc;
int ret;
ret = wc_InitRsaKey(&enc, NULL); // not using heap hint. No custom memory
if ( ret != 0 ) {
    // error initializing RSA key
}
```

See:

- wc_FreeRsaKey
- wc_RsaSetRNG

Return:

- 0 暗号化と復号化で使用するための RSA 構造の初期化に成功したときに返されます。
- BAD_FUNC_ARGS RSA キーポインタが NULL に評価された場合に返されます

C.44.2.2 function wc_InitRsaKey_Id

```
int wc_InitRsaKey_Id(
    RsaKey * key,
    unsigned char * id,
    int len,
    void * heap,
    int devId
)
```

この関数は提供された RsaKey 構造体を初期化します。ID と LEN は、DEVID がデバイスを識別している間にデバイス上のキーを識別するために使用されます。また、ユーザー定義メモリオーバーライドで使用するためのヒープ識別子も取ります（XMALLOC、XFREE、XREALLOC を参照）。wc_rsa_blinding が有効な場合、キーは WC_RSASETRNG によって RNG に関連付けられなければなりません。

Parameters:

- **key** 初期化する RsaKey 構造体へのポインタ
- **id** デバイス上のキーの識別子
- **len** バイト数の識別子の長さ
- **heap** メモリオーバーライドで使用するためのヒープ識別子へのポインタ。メモリ割り当てのカスタム処理を可能にします。このヒープは、この RSA オブジェクトで使用するためにメモリを割り当てるときに使用されるデフォルトになります。*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
}
```

See:

- wc_InitRsaKey
- wc_FreeRsaKey
- wc_RsaSetRNG

Return:

- 0 暗号化と復号化で使用するための RSA 構造の初期化に成功したときに返されます。
- BAD_FUNC_ARGS RSA キーポインタが NULL に評価された場合に返されます
- BUFFER_E LEN が RSA_MAX_ID_LEN よりも小さい場合、または大きい場合は返されます。

C.44.2.3 function wc_RsaSetRNG

```
int wc_RsaSetRNG(
    RsaKey * key,
    WC_RNG * rng
)
```

この関数は RNG をキーに関連付けます。WC_RSA_BLINDING が有効になっている場合は必要です。

Parameters:

- **key** 関連付けられる RsaKey 構造体へのポインタ *Example*

```
ret = wc_InitRsaKey(&key, NULL);
if (ret == 0) {
    ret = wc_InitRng(&rng);
} else return -1;
if (ret == 0) {
    ret = wc_RsaSetRNG(&key, &rng);
```

See:

- wc_InitRsaKey
- wc_RsaSetRNG

Return:

- 0 成功に戻った
- BAD_FUNC_ARGS RSA キーの場合、RNG ポインタが NULL に評価された場合

C.44.2.4 function wc_FreeRsaKey

```
int wc_FreeRsaKey(
    RsaKey * key
)
```

この関数は、MP_Clear を使用して提供された RsaKey 構造体を解放します。

See: wc_InitRsaKey

Return: 0 キーの解放に成功したら返品されます *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_RsaPublicEncrypt

```
int wc_RsaPublicEncrypt(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 outLen,
    RsaKey * key,
    WC_RNG * rng
)
```

この関数はメッセージを IN から暗号化し、その結果を格納します。初期化された公開鍵と乱数発生器が必要です。副作用として、この関数は ounlen の中で書き込まれたバイトを返します。

Parameters:

- **in** 暗号化する入力メッセージを含むバッファへのポインタ
- **inLen** 暗号化するメッセージの長さ
- **out** 出力暗号文を保存するバッファへのポインタ
- **outLen** 出力バッファの長さ
- **key** 暗号化に使用する公開鍵を含む RsaKey 構造体へのポインタ *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
}
```

See: wc_RsaPrivateDecrypt

Return:

- Success 入力メッセージの暗号化に成功したら、成功の場合は 0 を返し、障害の場合はゼロ未満です。また、outlen の値を格納することによって、OUT に書き込まれた数のバイト数を返します。
- BAD_FUNC_ARG 入力パラメータのいずれかが無効な場合に返されます
- RSA_BUFFER_E CipherText を保存するには、出力バッファが小さすぎる場合は返されます。
- RNG_FAILURE_E 提供された RNG 構造体を使用してランダムブロックを生成するエラーがある場合
- MP_INIT_E メッセージの暗号化中に使用されている数学ライブラリにエラーがある場合に返される可能性があります。
- MP_READ_E メッセージの暗号化中に使用されている数学ライブラリにエラーがある場合に返される可能性があります。
- MP_CMP_E メッセージの暗号化中に使用されている数学ライブラリにエラーがある場合に返される可能性があります。
- MP_INVMOD_E メッセージの暗号化中に使用されている数学ライブラリにエラーがある場合に返される可能性があります。
- MP_EXPTMOD_E メッセージの暗号化中に使用されている数学ライブラリにエラーがある場合に返される可能性があります。
- MP_MOD_E メッセージの暗号化中に使用されている数学ライブラリにエラーがある場合に返される可能性があります。
- MP_MUL_E メッセージの暗号化中に使用されている数学ライブラリにエラーがある場合に返される可能性があります。
- MP_ADD_E メッセージの暗号化中に使用されている数学ライブラリにエラーがある場合に返される可能性があります。
- MP_MULMOD_E メッセージの暗号化中に使用されている数学ライブラリにエラーがある場合に返される可能性があります。
- MP_TO_E メッセージの暗号化中に使用されている数学ライブラリにエラーがある場合に返される可能性があります。
- MP_MEM メッセージの暗号化中に使用されている数学ライブラリにエラーがある場合に返される可能性があります。
- MP_ZERO_E メッセージの暗号化中に使用されている数学ライブラリにエラーがある場合に返される可能性があります。

C.44.2.6 function wc_RsaPrivateDecryptInline

```
int wc_RsaPrivateDecryptInline(
    byte * in,
    word32 inLen,
    byte ** out,
    RsaKey * key
)
```

この関数は復号化のために WC_RSAPrivateCrypt 関数によって利用されます。

Parameters:

- **in** 復号化されるバイト配列。
- **inLen** の長さ
- **out** 格納する復号化データのバイト配列。*Example*

none

See: wc_RsaPrivateDecrypt

Return:

- Success 復号化データの長さ
- RSA_PAD_E RSAUNPAD エラー、フォーマットの悪いフォーマット

C.44.2.7 function wc_RsaPrivateDecrypt

```
int wc_RsaPrivateDecrypt(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 outLen,
    RsaKey * key
)
```

この関数は秘密の RSA 復号化を提供します。

Parameters:

- **in** 復号化されるバイト配列。
- **inLen** の長さ
- **out** 格納する復号化データのバイト配列。
- **outLen** の長さ *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;
}
```

See:

- RsaUnPad
- wc_RsaFunction
- wc_RsaPrivateDecryptInline

Return:

- Success 復号化データの長さ
- MEMORY_E -125、メモリエラーが発生しました
- BAD_FUNC_ARG -173、関数の不良引数が提供されています

C.44.2.8 function wc_RsaSSL_Sign

```
int wc_RsaSSL_Sign(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 outLen,
    RsaKey * key,
    WC_RNG * rng
)
```

提供された配列に秘密鍵と署名します。

Parameters:

- **in** 暗号化されるバイト配列。
- **inLen** の長さ
- **out** 格納する暗号化データのバイト配列。
- **outLen** の長さ
- **key** 暗号化に使用する鍵。*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;
}
```

See: wc_RsaPad

Return: RSA_BUFFER_E: -131、RSA バッファエラー、出力が小さすぎたり入力が大きすぎたりする

C.44.2.9 function wc_RsaSSL_VerifyInline

```
int wc_RsaSSL_VerifyInline(
    byte * in,
    word32 inLen,
    byte ** out,
    RsaKey * key
)
```

メッセージが RSA キーによって署名されたことを確認するために使用されます。出力は入力と同じバイト配列を使用します。

Parameters:

- **in** 復号化されるバイト配列。
- **inLen** バッファ入力の長さ。
- **out** 復号化された情報のポインタへのポインタ。*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
}
```

See:

- wc_RsaSSL_Verify
- wc_RsaSSL_Sign

Return:

- 0 テキストの長さ
- <0 エラーが発生しました。

C.44.2.10 function wc_RsaSSL_Verify

```
int wc_RsaSSL_Verify(
    const byte * in,
    word32 inLen,
    byte * out,
    word32 outLen,
    RsaKey * key
)
```

メッセージがキーによって署名されたことを確認するために使用されます。

Parameters:

- **in** 復号化されるバイト配列。
- **inLen** の長さ
- **out** 格納する復号化データのバイト配列。
- **outLen** の長さ *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;
}
```

See: wc_RsaSSL_Sign

Return:

- Success エラーのないテキストの長さ。
- MEMORY_E メモリ例外

C.44.2.11 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
)
```

提供された配列に秘密鍵と署名します。

Parameters:

- **in** 暗号化されるバイト配列。
- **inLen** の長さ
- **out** 格納する暗号化データのバイト配列。
- **outLen** の長さ
- **hash** メッセージに入るハッシュ型
- **mgf** マスク生成機能識別子 *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);
```

See:

- wc_RsaPSS_Verify
- wc_RsaSetRNG

Return: RSA_BUFFER_E: -131、RSA バッファエラー、出力が小さすぎたり入力が大きすぎたりする

C.44.2.12 function wc_RsaPSS_Verify

```
int wc_RsaPSS_Verify(
    byte * in,
    word32 inLen,
    byte * out,
    word32 outLen,
    enum wc_HashType hash,
    int mgf,
    RsaKey * key
)
```

入力署名を復号して、メッセージが鍵によって署名されたことを確認します。WC_RSA_BLINDING が有効な場合、鍵は wc_RsaSetRNG によって RNG に関連付けられなければなりません。

Parameters:

- **in** 復号される署名データが格納されているバッファ
- **inLen** 署名データの長さ
- **out** 復号データの出力先バッファ
- **outLen** 出力先バッファサイズ
- **hash** メッセージに入るハッシュ型
- **mgf** マスク生成機能識別子 *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);
```

See:

- wc_RsaPSS_Sign
- wc_RsaPSS_VerifyInline
- wc_RsaPSS_CheckPadding
- wc_RsaSetRNG

Return:

- Success エラーのない場合はテキストの長さを返します
- MEMORY_E メモリ例外

- MP_EXPTMOD_E - fastmath を使用する様に構成されている場合に FP_MAX_BITS が鍵サイズの少なくとも 2 倍に設定されていない (例えば 4096-bit 長の鍵を使用する場合には FP_MAX_BITS は 8192 以上に設定すること)。

C.44.2.13 function wc_RsaPSS_VerifyInline

```
int wc_RsaPSS_VerifyInline(
    byte * in,
    word32 inLen,
    byte ** out,
    enum wc_HashType hash,
    int mgf,
    RsaKey * key
)
```

入力署名を復号化して、メッセージが RSA キーによって署名されたことを確認します。出力は入力と同じバイト配列を使用します。WC_RSA_BLINDING が有効な場合、キーは WC_RSASETRNG によって RNG に関連付けられなければなりません。

Parameters:

- **in** 復号化されるバイト配列。
- **inLen** バッファ入力の長さ。
- **out** PSS データを含むアドレスへのポインタ。
- **hash** メッセージに入るハッシュ型
- **mgf** マスク生成機能識別子 *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);
```

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 テキストの長さ
- <0 エラーが発生しました。

C.44.2.14 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
)
```

RSA-PSS で署名されたメッセージを確認してください。ソルトの長さはハッシュ長に等しい。WC_RSA_BLINDING が有効な場合、キーは WC_RSASETRNG によって RNG に関連付けられなければなりません。

Parameters:

- **in** 復号化されるバイト配列。
- **inLen** の長さ
- **out** PSS データを含むアドレスへのポインタ。
- **outLen** の長さ
- **digest** 検証中のデータのハッシュ。
- **digestLen** ハッシュの長さ
- **hash** ハッシュアルゴリズム
- **mgf** マスク生成機能 *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);
```

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 PSS データの長さが成功し、負に障害が発生します。
- MEMORY_E メモリ例外

C.44.2.15 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
)
```

RSA-PSS で署名されたメッセージを確認してください。WC_RSA_BLINDING が有効な場合、キーは WC_RSASETRNG によって RNG に関連付けられなければなりません。

Parameters:

- **in** 復号化されるバイト配列。
- **inLen** の長さ
- **out** PSS データを含むアドレスへのポインタ。
- **outLen** の長さ
- **digest** 検証中のデータのハッシュ。
- **digestLen** ハッシュの長さ
- **hash** ハッシュアルゴリズム
- **mgf** マスク生成機能

- **saltLen** 使用されるソルトの長さ。RSA_PSSS_SALT_LEN_DEFAULT（-1）ソルトの長さはハッシュ長と同じです。RSA_PSS_SALT_LEN_DISCOVER は、ソルトの長さがデータから決定されます。*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);
```

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 PSS データの長さが成功し、負に障害が発生します。
- MEMORY_E メモリ例外

C.44.2.16 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
)
```

RSA-PSS で署名されたメッセージを確認してください。入力バッファは出力バッファに再利用されます。ソルトの長さはハッシュ長に等しい。WC_RSA_BLINDING が有効な場合、キーは WC_RSASETRNG によって RNG に関連付けられなければなりません。

Parameters:

- **in** 復号化されるバイト配列。
- **inLen** の長さ
- **out** 格納する復号化データのバイト配列。
- **digest** 検証中のデータのハッシュ。
- **digestLen** ハッシュの長さ
- **hash** メッセージに入るハッシュ型
- **mgf** マスク生成機能識別子 *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);
```

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 PSS データの長さが成功し、負に障害が発生します。

C.44.2.17 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
)
```

RSA-PSS で署名されたメッセージを確認してください。入力バッファは出力バッファに再利用されます。WC_RSA_BLINDING が有効な場合、キーは WC_RSASETRNG によって RNG に関連付けられなければなりません。

Parameters:

- **in** 復号化されるバイト配列。
- **inLen** の長さ
- **out** 格納する復号化データのバイト配列。
- **digest** 検証中のデータのハッシュ。
- **digestLen** ハッシュの長さ
- **hash** メッセージに入るハッシュ型
- **mgf** マスク生成機能識別子
- **saltLen** 使用されるソルトの長さ。RSA_PSSS_SALT_LEN_DEFAULT（-1）ソルトの長さはハッシュ長と同じです。RSA_PSS_SALT_LEN_DISCOVER は、ソルトの長さがデータから決定されます。*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);
```

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 PSS データの長さが成功し、負に障害が発生します。

C.44.2.18 function wc_RsaPSS_CheckPadding

```
int wc_RsaPSS_CheckPadding(
    const byte * in,
    word32 inLen,
    byte * sig,
    word32 sigSz,
    enum wc_HashType hashType
)
```

PSS データを確認して、署名が一致するようにします。ソルトの長さはハッシュ長に等しい。WC_RSA_BLINDING が有効な場合、キーは WC_RSASETRNG によって RNG に関連付けられなければなりません。

Parameters:

- **in** 検証中のデータのハッシュ。
- **inSz** ハッシュの長さ
- **sig** PSS データを保持するバッファ。
- **sigSz** PSS データのサイズ。*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);
```

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 PSS データが無効な場合、NULL が IN または SIG または INSZ に渡されると、BAD_FUNC_ARG はハッシュアルゴリズムの長さと同じではありません。
- MEMORY_E メモリ例外

C.44.2.19 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
)
```

PSS データを確認して、署名が一致するようにします。ソルトの長さはハッシュ長に等しい。

Parameters:

- **in** 検証中のデータのハッシュ。
- **inSz** ハッシュの長さ
- **sig** PSS データを保持するバッファ。

- **sigSz** PSS データのサイズ。
- **hashType** ハッシュアルゴリズム
- **saltLen** 使用されるソルトの長さ。RSA_PSSS_SALT_LEN_DEFAULT（-1）ソルトの長さはハッシュ長と同じです。RSA_PSS_SALT_LEN_DISCOVER は、ソルトの長さがデータから決定されます。*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);
```

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 PSS データが無効な場合、NULL が IN または SIG または INSZ に渡されると、BAD_FUNC_ARG はハッシュアルゴリズムの長さと同じではありません。
- MEMORY_E メモリ例外

C.44.2.20 function wc_RsaEncryptSize

```
int wc_RsaEncryptSize(
    RsaKey * key
)
```

提供されたキー構造の暗号化サイズを返します。

See:

- wc_InitRsaKey
- wc_InitRsaKey_ex
- wc_MakeRsaKey

Return: Success 提供されたキー構造の暗号化サイズ。*Example*

```
int sz = wc_RsaEncryptSize(&key);
```

C.44.2.21 function wc_RsaPrivateKeyDecode

```
int wc_RsaPrivateKeyDecode(
    const byte * input,
    word32 * inOutIdx,
    RsaKey * key,
    word32 inSz
)
```

この関数は Der フォーマットされた RSA 秘密鍵を解析し、秘密鍵を抽出し、それを与えられた Resakey 構造に格納します。IDX に解析された距離も設定します。

Parameters:

- **input** デコードする DER フォーマット秘密鍵を含むバッファへのポインタ
- **inOutIdx** キーが始まるバッファ内のインデックスへのポインタ（通常は 0）。この関数の副作用として、InoutIDX は入力バッファを介して解析された距離を記憶します
- **key** デコードされた秘密鍵を保存する RSAKEY 構造へのポインタ *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
}
```

See:

- wc_RsaPublicKeyDecode
- wc_MakeRsaKey

Return:

- 0 DER エンコード入力から秘密鍵の解析に成功したときに返されます
- ASN_PARSE_E 入力バッファから秘密鍵を解析するエラーがある場合に返されます。これは、入力秘密鍵が ASN.1 規格に従って正しくフォーマットされていない場合に発生する可能性があります。
- ASN_RSA_KEY_E RSA キー入力の秘密鍵要素を読み取るエラーがある場合

C.44.2.22 function wc_RsaPublicKeyDecode

```
int wc_RsaPublicKeyDecode(
    const byte * input,
    word32 * inOutIdx,
    RsaKey * key,
```

```
    word32 inSz
)
```

この関数は Der フォーマットの RSA 公開鍵を解析し、公開鍵を抽出し、それを指定された RsaKey 構造体に格納します。IDX に解析された距離も設定します。

Parameters:

- **input** 復号する入力 DER エンコード RSA 公開鍵を含むバッファへのポインタ
- **inOutIdx** キーが始まるバッファ内のインデックスへのポインタ（通常は 0）。この関数の副作用として、InoutIDX は入力バッファを介して解析された距離を記憶します
- **key** デコードされた公開鍵を保存する RsaKey 構造体へのポインタ *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
}
```

See: wc_RsaPublicKeyDecodeRaw

Return:

- 0 DER エンコード入力から公開鍵の解析に成功したときに返された
- ASN_PARSE_E 入力バッファから公開鍵を解析したエラーがある場合に返されます。これは、入力公開鍵が ASN.1 規格に従って正しくフォーマットされていない場合に発生する可能性があります。
- ASN_OBJECT_ID_E ASN.1 オブジェクト ID が RSA 公開鍵のそれと一致しない場合に返されます。
- ASN_EXPECT_0_E 入力キーが ASN.1 規格に従って正しくフォーマットされていない場合
- ASN_BITSTR_E 入力キーが ASN.1 規格に従って正しくフォーマットされていない場合
- ASN_RSA_KEY_E RSA キー入力の公開鍵要素を読み取るエラーがある場合

C.44.2.23 function wc_RsaPublicKeyDecodeRaw

```
int wc_RsaPublicKeyDecodeRaw(
    const byte * n,
    word32 nSz,
    const byte * e,
    word32 eSz,
    RsaKey * key
)
```

この関数は、公開弾性率（n）と指数（e）を撮影して、RSA 公開鍵の生の要素を復号します。これらの生の要素を提供された RsaKey 構造体に格納し、暗号化/復号化プロセスで使用することができます。

Parameters:

- **n** Public RSA キーの RAW モジュラスパラメータを含むバッファへのポインタ
- **nSz** N を含むバッファのサイズ
- **e** Public RSA キーの RAW 指数パラメータを含むバッファへのポインタ
- **eSz** E を含むバッファのサイズ *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
}
```

See: wc_RsaPublicKeyDecode

Return:

- 0 公開鍵の生の要素を RsaKey 構造体に復号したときに返された
- BAD_FUNC_ARG いずれかの入力引数が NULL に評価された場合に返されます。
- MP_INIT_E 複数の精密整数（MP_INT）ライブラリで使用するための整数の初期化中にエラーがある場合
- ASN_GETINT_E 提供された RSA キー要素、n または e のいずれかを読むエラーがある場合に返されます

C.44.2.24 function wc_RsaKeyToDer

```
int wc_RsaKeyToDer(
    RsaKey * key,
    byte * output,
    word32 inLen
)
```

この機能は RSAKEY キーを DER フォーマットに変換します。結果は出力に書き込まれ、書き込まれたバイト数を返します。

Parameters:

- **key** 初期化された RsaKey 構造体
- **output** 出力バッファへのポインタ。*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
}
```

See:

- wc_RsaKeyToPublicDer
- wc_InitRsaKey
- wc_MakeRsaKey
- wc_InitRng

Return:

- 0 成功、書かれたバイト数。

- BAD_FUNC_ARG キーまたは出力が NULL の場合、またはキー - > タイプが RSA_PRIVATE でない場合、または INLEN が出力バッファに十分な大きさでない場合は返されます。
- MEMORY_E メモリの割り当て中にエラーが発生した場合に返されます。

C.44.2.25 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
)
```

この機能は、どのパディングを使用するかを選択しながら RSA 暗号化を実行します。

Parameters:

- **in** 暗号化のためのバッファへのポインタ
- **inLen** 暗号化するバッファの長さ
- **out** 暗号化された MSG が作成されました
- **outLen** 暗号化された MSG を保持するために利用可能なバッファの長さ
- **key** 初期化ずみ RsaKey 構造体
- **rng** 初期化された WC_RNG 構造体
- **type** 使用するパディングの種類（WC_RSA_OAEP_PAD または WC_RSA_PKCSV15_PAD）
- **hash** 使用するハッシュの種類（選択は hash.h にあります）
- **mgf** 使用するマスク生成機能の種類
- **label** 暗号化されたメッセージに関連付けるオプションのラベル *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
}
```

See:

- wc_RsaPublicEncrypt
- wc_RsaPrivateDecrypt_ex

Return:

- size 正常に暗号化されていると、暗号化されたバッファのサイズが返されます
- RSA_BUFFER_E RSA バッファエラー、出力が小さすぎたり入力が大きすぎたりする

C.44.2.26 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
)
```

この関数は RSA を使用してメッセージを復号化し、どのパディングタイプのオプションを指定します。

Parameters:

- **in** 復号化のためのバッファへのポインタ
- **inLen** 復号化するバッファの長さ
- **out** 復号化された MSG が作成されました
- **outLen** 復号化された MSG を保持するために利用可能なバッファの長さ
- **key** 初期化ずみ RsaKey 構造体
- **type** 使用するパディングの種類（WC_RSA_OAEP_PAD または WC_RSA_PKCSV15_PAD）
- **hash** 使用するハッシュの種類（選択は hash.h にあります）
- **mgf** 使用するマスク生成機能の種類
- **label** 暗号化されたメッセージに関連付けるオプションのラベル *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
}
```

See: none

Return:

- size 復号化が成功すると、復号化されたメッセージのサイズが返されます。
- MEMORY_E 必要な配列を Malloc に Malloc にするのに十分なメモリがない場合は返されます。
- BAD_FUNC_ARG 関数に渡された引数が渡された場合に返されます。

C.44.2.27 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
)
```

この関数は RSA を使用してメッセージをインラインで復号化し、どのパディングタイプのオプションを示します。IN バッファには、呼び出された後に復号化されたメッセージが含まれ、アウトバイトポインタはプレーンテキストがある「IN」バッファ内の場所を指します。

Parameters:

- **in** 復号化のためのバッファへのポインタ
- **inLen** 復号化するバッファの長さ
- **out** “in” バッファの復号化されたメッセージの位置へのポインタ
- **key** 初期化ずみ RsaKey 構造体
- **type** 使用するパディングの種類（WC_RSA_OAEP_PAD または WC_RSA_PKCSV15_PAD）
- **hash** 使用するハッシュの種類（選択は hash.h にあります）
- **mgf** 使用するマスク生成機能の種類
- **label** 暗号化されたメッセージに関連付けるオプションのラベル *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
}
```

See: none

Return:

- size 復号化が成功すると、復号化されたメッセージのサイズが返されます。
- MEMORY_E: 必要な配列を Malloc に Malloc にするのに十分なメモリがない場合は返されます。
- RSA_PAD_E: パディングのエラーが発生した場合に返されます。
- BAD_PADDING_E: 過去のパディングの解析中にエラーが発生した場合に返されます。
- BAD_FUNC_ARG: 関数に渡された引数が渡された場合に返されます。

C.44.2.28 function wc_RsaFlattenPublicKey

```
int wc_RsaFlattenPublicKey(
    RsaKey * key,
    byte * e,
    word32 * eSz,
    byte * n,
    word32 * nSz
)
```

RSA アルゴリズムに使用される RsaKey 構造体の個々の要素（E、N）をバッファに取り出します。

Parameters:

- **key** 検証に使用する鍵。
- **e** e の値のバッファー。e は RSA モジュラ演算での大きな正の整数です。
- **eSz** e バッファのサイズ。
- **n** n の値のバッファー。N は RSA モジュラー演算では大きな正の整数です。*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.
}
```

See:

- wc_InitRsaKey
- wc_InitRsaKey_ex
- wc_MakeRsaKey

Return:

- 0 関数が正常に実行された場合は、エラーなしで返されます。
- BAD_FUNC_ARG: いずれかのパラメータが NULL 値で渡された場合に返されます。
- RSA_BUFFER_E: 渡された e または n バッファが正しいサイズではない場合に返されます。
- MP_MEM: 内部関数にメモリエラーがある場合に返されます。
- MP_VAL: 内部関数引数が無効な場合に返されます。

C.44.2.29 function wc_RsaKeyToPublicDer

```
int wc_RsaKeyToPublicDer(
    RsaKey * key,
    byte * output,
    word32 inLen
)
```

RSA 公開鍵を DER フォーマットに変換します。出力に書き込み、書き込まれたバイト数を返します。

Parameters:

- **key** 変換する RsaKey 構造体。
- **output** 保留された出力バッファー。（NULL が長さのみを返す場合）*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
}
```

See:

- wc_RsaPublicKeyDerSize
- wc_RsaKeyToPublicDer_ex
- wc_InitRsaKey

Return:

- 0 成功、書かれたバイト数。
- BAD_FUNC_ARG キーまたは出力が NULL の場合に返されます。
- MEMORY_E エラー割り当てメモリが発生したときに返されます。
- <0 エラー

C.44.2.30 function wc_RsaKeyToPublicDer_ex

```
int wc_RsaKeyToPublicDer_ex(
    RsaKey * key,
    byte * output,
    word32 inLen,
    int with_header
)
```

RSA 公開鍵を DER フォーマットに変換します。出力に書き込み、書き込まれたバイト数を返します。with_header が 0 の場合（seq + n + e）だけが ASN.1 Der フォーマットで返され、ヘッダーを除外します。

Parameters:

- **key** 変換する RsaKey 構造体。
- **output** 保留された出力バッファー。（NULL が長さのみを返す場合）*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
}
```

See:

- wc_RsaPublicKeyDerSize
- wc_RsaKeyToPublicDer
- wc_InitRsaKey

Return:

- 0 成功、書かれたバイト数。
- BAD_FUNC_ARG キーまたは出力が NULL の場合に返されます。
- MEMORY_E エラー割り当てメモリが発生したときに返されます。
- <0 エラー

C.44.2.31 function wc_MakeRsaKey

```
int wc_MakeRsaKey(
    RsaKey * key,
    int size,
    long e,
    WC_RNG * rng
)
```

この関数は、長さサイズ（ビット単位）の RSA 秘密鍵を生成し、指数（e）を指定します。次に、このキーを提供された RsaKey 構造体に格納するため、暗号化/復号化に使用できます。E に使用するセキュア番号は 65537 です。サイズは、RSA_MIN_SIZE よりも大きく、RSA_MAX_SIZE よりも大きくなる必要があります。この機能が利用可能であるため、コンパイル時にオプション wolfssl_key_gen を有効にする必要があります。これは、- を使用してください./configure を使用する場合は、-enable-keygen で実現できます。

Parameters:

- **key** 生成された秘密鍵を保存する RSAKEY 構造体へのポインタ
- **size** ビット単位の希望のキー長。rsa_min_size より大きく、rsa_max_size よりも大きくなる必要があります。
- **e** キーを生成するために使用する指数パラメータ。安全な選択は 65537 です *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
}
```

See: none

Return:

- 0 RSA 秘密鍵の生成に成功したら返されました
- BAD_FUNC_ARG 入力引数のいずれかが NULL の場合、サイズパラメータは必要な範囲外にあるか、e が誤って選択されている場合
- RNG_FAILURE_E 提供された RNG 構造体を使用してランダムブロックを生成するエラーがある場合
- MP_INIT_E
- MP_READ_E RSA キーの生成中に使用された数学ライブラリにエラーがある場合に返された RSA キーの生成中に使用された数学ライブラリにエラーがある場合に返される可能性があります。
- MP_CMP_E RSA キーの生成中に使用されている数学ライブラリにエラーがある場合は返される可能性があります。
- MP_INVMOD_E RSA キーの生成中に使用されている数学ライブラリにエラーがある場合は返される可能性があります。

- MP_EXPTMOD_E RSA キーの生成中に使用されている数学ライブラリにエラーがある場合は返される可能性があります。
- MP_MOD_E RSA キーの生成中に使用されている数学ライブラリにエラーがある場合は返される可能性があります。
- MP_MUL_E RSA キーの生成中に使用されている数学ライブラリにエラーがある場合は返される可能性があります。
- MP_ADD_E RSA キーの生成中に使用されている数学ライブラリにエラーがある場合は返される可能性があります。
- MP_MULMOD_E RSA キーの生成中に使用されている数学ライブラリにエラーがある場合は返される可能性があります。
- MP_TO_E RSA キーの生成中に使用されている数学ライブラリにエラーがある場合は返される可能性があります。
- MP_MEM RSA キーの生成中に使用されている数学ライブラリにエラーがある場合は返される可能性があります。
- MP_ZERO_E RSA キーの生成中に使用されている数学ライブラリにエラーがある場合は返される可能性があります。

C.44.2.32 function wc_RsaSetNonBlock

```
int wc_RsaSetNonBlock(
    RsaKey * key,
    RsaNb * nb
)
```

この関数は、ブロックされていない RSA コンテキストを設定します。RSANB コンテキストが設定されている場合、RSA 関数を多くの小さな操作に分割する高速数学ベースの非ブロッキング EXPTMOD が可能になります。wc_rsa_nonblock が定義されているときに有効になっています。

Parameters:

- **key** RSA キー構造 *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);
```

See: wc_RsaSetNonBlockTime

Return:

- 0 成功
- BAD_FUNC_ARG キーまたは NB が NULL の場合に返されます。

C.44.2.33 function wc_RsaSetNonBlockTime

```
int wc_RsaSetNonBlockTime(
    RsaKey * key,
    word32 maxBlockUs,
    word32 cpuMHz
)
```

この関数は最大ブロック時間の最大ブロック時間をマイクロ秒単位で設定します。それは、メガヘルツの CPU 速度と共に事前計算されたテーブル（TFM.cexptModnbinst を参照）を使用して、提供された最大ブロック時間内に次の動作を完了できるかどうかを判断します。wc_rsa_nonblock_time が定義されているときに有効になります。

Parameters:

- **key** RsaKey 構造体
- **maxBlockUs** マイクロ秒をブロックする最大時間。*Example*

```
RsaKey key;
RsaNb  nb;

wc_InitRsaKey(&key, NULL);
wc_RsaSetNonBlock(key, &nb);
wc_RsaSetNonBlockTime(&key, 4000, 160); // Block Max = 4 ms, CPU = 160MHz
```

See: wc_RsaSetNonBlock

Return:

- 0 成功
- BAD_FUNC_ARG キーが NULL の場合、または WC_RSASETNONBLOCK が以前に呼び出され、キー -> NB は NULL の場合に返されます。

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_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-ja/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)

	Name
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.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-ja/sha256.h

C.46.1 Functions

	Name
int	**wc_InitSha256**(wc_Sha256 *) この関数は SHA256 を初期化します。これは WC_SHA256HASH によって自動的に呼び出されます。
int	**wc_Sha256Update**(wc_Sha256 * sha, const byte * data, word32 len) 長さ LEN の提供されたバイト配列を絶えずハッシュするように呼び出すことができます。
int	**wc_Sha256Final**(wc_Sha256 * sha256, byte * hash) データのハッシュを確定します。結果はハッシュに入れられます。SHA256 構造体の状態をリセットします。
void	**wc_Sha256Free**(wc_Sha256 *)SHA256 構造をリセットします。注：これは、wolfssl_ti_hash が定義されている場合にのみサポートされています。

	Name
int	**wc_Sha256GetHash**(wc_Sha256 * sha256, byte * hash) ハッシュデータを取得します。結果はハッシュに入れられます。SHA256 構造体の状態をリセットしません。
int	**wc_InitSha224**(wc_Sha224 *)SHA224 構造を初期化するために使用されます。
int	**wc_Sha224Update**(wc_Sha224 * sha224, const byte * data, word32 len) 長さ LEN の提供されたバイト配列を絶えずハッシュするように呼び出すことができます。
int	**wc_Sha224Final**(wc_Sha224 * sha224, byte * hash) データのハッシュを確定します。結果はハッシュに入れられます。SHA224 構造体の状態をリセットします。

C.46.2 Functions Documentation

C.46.2.1 function wc_InitSha256

```
int wc_InitSha256(
    wc_Sha256 *
)
```

この関数は SHA256 を初期化します。これは WC_SHA256HASH によって自動的に呼び出されます。

See:

- wc_Sha256Hash
- wc_Sha256Update
- wc_Sha256Final

Return: 0 初期化に成功したときに返されます *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
)
```

長さ LEN の提供されたバイト配列を絶えずハッシュするように呼び出すことができます。

Parameters:

- **sha256** 暗号化に使用する SHA256 構造へのポインタ
- **data** ハッシュするデータ *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);
}
```

See:

- wc_Sha256Hash
- wc_Sha256Final
- wc_InitSha256

Return: 0 データをダイジェストに正常に追加すると返されます。

C.46.2.3 function wc_Sha256Final

```
int wc_Sha256Final(
    wc_Sha256 * sha256,
    byte * hash
)
```

データのハッシュを確定します。結果はハッシュに入れられます。SHA256 構造体の状態をリセットします。

Parameters:

- **sha256** 暗号化に使用する SHA256 構造へのポインタ *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);
}
```

See:

- wc_Sha256Hash
- wc_Sha256GetHash
- wc_InitSha256

Return: 0 ファイナライズに成功したときに返されます。

C.46.2.4 function wc_Sha256Free

```
void wc_Sha256Free(
    wc_Sha256 *
)
```

SHA256 構造をリセットします。注：これは、wolfssl_ti_hash が定義されている場合にのみサポートされています。

See:

- wc_InitSha256
- wc_Sha256Update
- wc_Sha256Final

Return: none いいえ返します。*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
)
```

ハッシュデータを取得します。結果はハッシュに入れられます。SHA256 構造体の状態をリセットしません。

Parameters:

- **sha256** 暗号化に使用する SHA256 構造へのポインタ *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);
}
```

See:

- wc_Sha256Hash
- wc_Sha256Final
- wc_InitSha256

Return: 0 ファイナライズに成功したときに返されます。

C.46.2.6 function wc_InitSha224

```
int wc_InitSha224(
    wc_Sha224 *
)
```

SHA224 構造を初期化するために使用されます。

See:

- wc_Sha224Hash
- wc_Sha224Update
- wc_Sha224Final

Return:

- 0 成功
- 1 SHA224 が NULL なので、エラーが返されました。*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
)
```

長さ LEN の提供されたバイト配列を絶えずハッシュするように呼び出すことができます。

Parameters:

- **sha224** 暗号化に使用する SHA224 構造へのポインタ。
- **data** ハッシュするデータ。*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);
}
```

See:

- wc_InitSha224
- wc_Sha224Final
- wc_Sha224Hash

Return:

- 0 成功
- 1 関数が失敗した場合はエラーが返されます。
- BAD_FUNC_ARG SHA224 またはデータが NULL の場合、エラーが返されます。

C.46.2.8 function wc_Sha224Final

```
int wc_Sha224Final(
    wc_Sha224 * sha224,
    byte * hash
)
```

データのハッシュを確定します。結果はハッシュに入れられます。SHA224 構造体の状態をリセットします。

Parameters:

- **sha224** 暗号化に使用する SHA224 構造へのポインタ *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);
}
```

See:

- wc_InitSha224
- wc_Sha224Hash
- wc_Sha224Update

Return:

- 0 成功
- <0 エラー

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-ja/sha512.h

C.47.1 Functions

	Name
int	**wc_InitSha512**(wc_Sha512 *) この関数は SHA512 を初期化します。これは WC_SHA512HASH によって自動的に呼び出されます。
int	**wc_Sha512Update**(wc_Sha512 * sha, const byte * data, word32 len) 長さ LEN の提供されたバイト配列を絶えずハッシュするように呼び出すことができます。
int	**wc_Sha512Final**(wc_Sha512 * sha512, byte * hash) データのハッシュを確定します。結果はハッシュに入れられます。
int	**wc_InitSha384**(wc_Sha384 *) この関数は SHA384 を初期化します。これは WC_SHA384HASH によって自動的に呼び出されます。
int	**wc_Sha384Update**(wc_Sha384 * sha, const byte * data, word32 len) 長さ LEN の提供されたバイト配列を絶えずハッシュするように呼び出すことができます。
int	**wc_Sha384Final**(wc_Sha384 * sha384, byte * hash) データのハッシュを確定します。結果はハッシュに入れられます。

C.47.2 Functions Documentation

C.47.2.1 function wc_InitSha512

```
int wc_InitSha512(
    wc_Sha512 *
)
```

この関数は SHA512 を初期化します。これは WC_SHA512HASH によって自動的に呼び出されます。

See:

- wc_Sha512Hash
- wc_Sha512Update
- wc_Sha512Final

Return: 0 初期化に成功したときに返されます *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
)
```

長さ LEN の提供されたバイト配列を絶えずハッシュするように呼び出すことができます。

Parameters:

- **sha512** 暗号化に使用する SHA512 構造へのポインタ
- **data** ハッシュするデータ *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);
}
```

See:

- wc_Sha512Hash
- wc_Sha512Final
- wc_InitSha512

Return: 0 データをダイジェストに正常に追加すると返されます。

C.47.2.3 function wc_Sha512Final

```
int wc_Sha512Final(
    wc_Sha512 * sha512,
    byte * hash
)
```

データのハッシュを確定します。結果はハッシュに入れられます。

Parameters:

- **sha512** 暗号化に使用する SHA512 構造へのポインタ *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);
}
```

See:

- wc_Sha512Hash
- wc_Sha512Final
- wc_InitSha512

Return: 0 ハッシュを確定するとうまく返されました。

C.47.2.4 function wc_InitSha384

```
int wc_InitSha384(
    wc_Sha384 *
)
```

この関数は SHA384 を初期化します。これは WC_SHA384HASH によって自動的に呼び出されます。

See:

- wc_Sha384Hash
- wc_Sha384Update
- wc_Sha384Final

Return: 0 初期化に成功したときに返されます *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
)
```

長さ LEN の提供されたバイト配列を絶えずハッシュするように呼び出すことができます。

Parameters:

- **sha384** 暗号化に使用する SHA384 構造へのポインタ
- **data** ハッシュするデータ *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);
}
```

See:

- wc_Sha384Hash
- wc_Sha384Final
- wc_InitSha384

Return: 0 データをダイジェストに正常に追加すると返されます。

C.47.2.6 function wc_Sha384Final

```
int wc_Sha384Final(
    wc_Sha384 * sha384,
    byte * hash
)
```

データのハッシュを確定します。結果はハッシュに入れられます。

Parameters:

- **sha384** 暗号化に使用する SHA384 構造へのポインタ *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);
}
```

See:

- wc_Sha384Hash
- wc_Sha384Final
- wc_InitSha384

Return: 0 ファイナライズに成功したときに返されます。

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-ja/sha.h

C.48.1 Functions

	Name
int	**wc_InitSha**(wc_Sha *) この関数は SHA を初期化します。これは自動的に WC_Shahash によって呼び出されます。
int	**wc_ShaUpdate**(wc_Sha * sha, const byte * data, word32 len) 長さ LEN の提供されたバイト配列を絶えずハッシュするように呼び出すことができます。
int	**wc_ShaFinal**(wc_Sha * sha, byte * hash) データのハッシュを確定します。結果はハッシュに入れられます。SHA 構造体の状態をリセットします。
void	**wc_ShaFree**(wc_Sha *) 初期化された SHA 構造体によって使用されるメモリをクリーンアップするために使用されます。注：これは、wolfssl_ti_hash が定義されている場合にのみサポートされています。
int	**wc_ShaGetHash**(wc_Sha * sha, byte * hash) ハッシュデータを取得します。結果はハッシュに入れられます。SHA 構造体の状態をリセットしません。

C.48.2 Functions Documentation

C.48.2.1 function wc_InitSha

```
int wc_InitSha(
    wc_Sha *
)
```

この関数は SHA を初期化します。これは自動的に WC_Shahash によって呼び出されます。

See:

- wc_ShaHash
- wc_ShaUpdate
- wc_ShaFinal

Return: 0 初期化に成功したときに返されます *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
)
```

長さ LEN の提供されたバイト配列を絶えずハッシュするように呼び出すことができます。

Parameters:

- **sha** 暗号化に使用する SHA 構造へのポインタ
- **data** ハッシュするデータ *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);
}
```

See:

- wc_ShaHash
- wc_ShaFinal
- wc_InitSha

Return: 0 データをダイジェストに正常に追加すると返されます。

C.48.2.3 function wc_ShaFinal

```
int wc_ShaFinal(
    wc_Sha * sha,
    byte * hash
)
```

データのハッシュを確定します。結果はハッシュに入れられます。SHA 構造体の状態をリセットします。

Parameters:

- **sha** 暗号化に使用する SHA 構造へのポインタ *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);
}
```

See:

- wc_ShaHash
- wc_InitSha
- wc_ShaGetHash

Return: 0 ファイナライズに成功したときに返されます。

C.48.2.4 function wc_ShaFree

```
void wc_ShaFree(
    wc_Sha *
)
```

初期化された SHA 構造体によって使用されるメモリをクリーンアップするために使用されます。注：これは、wolfssl_ti_hash が定義されている場合にのみサポートされています。

See:

- wc_InitSha
- wc_ShaUpdate
- wc_ShaFinal

Return: No 戻り値。*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
)
```

ハッシュデータを取得します。結果はハッシュに入れられます。SHA 構造体の状態をリセットしません。

Parameters:

- **sha** 暗号化に使用する SHA 構造へのポインタ *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);
}
```

See:

- wc_ShaHash
- wc_ShaFinal
- wc_InitSha

Return: 0 ファイナライズに成功したときに返されます。

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-ja/signature.h

C.49.1 Functions

	Name
int	**wc_SignatureGetSize**(enum wc_SignatureType sig_type, const void * key, word32 key_len) この関数は、結果のシグネチャの最大サイズを返します。
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) この関数は、データをハッシュし、結果のハッシュとキーを使用して署名を使用して署名を使用して署名を検証します。
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) この関数は、キーを使用してデータから署名を生成します。まずデータのハッシュを作成し、キーを使用してハッシュに署名します。

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
)
```

この関数は、結果のシグネチャの最大サイズを返します。

Parameters:

- **sig_type** wc_signature_type_ecc または wc_signature_type_rsa などの署名型列挙型値。
- **key** ECC_KEY や RSAKEY などのキー構造へのポインタ。*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
}
```

See:

- wc_HashGetDigestSize
- wc_SignatureGenerate
- wc_SignatureVerify

Return: Returns sig_type_e sig_type がサポートされていない場合 sig_type が無効な場合は bad_func_arg を返します。正の戻り値は、署名の最大サイズを示します。

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
)
```

この関数は、データをハッシュし、結果のハッシュとキーを使用して署名を使用して署名を使用して署名を検証します。

Parameters:

- **hash_type** “wc_hash_type_sha256” などの “enum wc_hashtype” からのハッシュ型。
- **sig_type** wc_signature_type_ecc または wc_signature_type_rsa などの署名型列挙型値。
- **data** ハッシュへのデータを含むバッファへのポインタ。
- **data_len** データバッファの長さ。
- **sig** 署名を出力するためのバッファへのポインタ。
- **sig_len** シグネチャ出力バッファの長さ。
- **key** ECC_KEY や RSAKEY などのキー構造へのポインタ。*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);
```

See:

- wc_SignatureGetSize
- wc_SignatureGenerate

Return:

- 0 成功
- SIG_TYPE_E -231、署名タイプが有効/利用可能です
- BAD_FUNC_ARG -173、関数の不良引数が提供されています
- BUFFER_E -132、出力バッファが小さすぎたり入力が大きすぎたりします。

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
)
```

この関数は、キーを使用してデータから署名を生成します。まずデータのハッシュを作成し、キーを使用してハッシュに署名します。

Parameters:

- **hash_type** “wc_hash_type_sha256” などの “enum wc_hashtype” からのハッシュ型。
- **sig_type** wc_signature_type_ecc または wc_signature_type_rsa などの署名型列挙型値。
- **data** ハッシュへのデータを含むバッファへのポインタ。
- **data_len** データバッファの長さ。
- **sig** 署名を出力するためのバッファへのポインタ。
- **sig_len** シグネチャ出力バッファの長さ。
- **key** ECC_KEY や RSAKEY などのキー構造へのポインタ。
- **key_len** キー構造のサイズ *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);
```

See:

- wc_SignatureGetSize
- wc_SignatureVerify

Return:

- 0 成功
- SIG_TYPE_E -231、署名タイプが有効/利用可能です
- BAD_FUNC_ARG -173、関数の不良引数が提供されています
- BUFFER_E -132、出力バッファが小さすぎたり入力が大きすぎたりします。

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-ja/siphash.h

C.50.1 Functions

	Name
int	**wc_InitSipHash**(SipHash * siphash, const unsigned char * key, unsigned char outSz) この関数は、Mac サイズのキーで Siphash を初期化します。
int	**wc_SipHashUpdate**(SipHash * siphash, const unsigned char * in, word32 inSz) 長さ LEN の提供されたバイト配列を絶えずハッシュするように呼び出すことができます。
int	**wc_SipHashFinal**(SipHash * siphash, unsigned char * out, unsigned char outSz) データの Macing を確定します。結果が出入りする。

	Name
int	**wc_SipHash**(const unsigned char * key, const unsigned char * in, word32 inSz, unsigned char * out, unsigned char outSz) この機能は Siphash を使用してデータを 1 ショットして、キーに基づいて MAC を計算します。

C.50.2 Functions Documentation

C.50.2.1 function wc_InitSipHash

```
int wc_InitSipHash(
    SipHash * siphash,
    const unsigned char * key,
    unsigned char outSz
)
```

この関数は、Mac サイズのキーで Siphash を初期化します。

Parameters:

- **siphash** Macing に使用するサイプハッシュ構造へのポインタ
- **key** 16 バイト配列へのポインタ *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");
}
```

See:

- wc_SipHash
- wc_SipHashUpdate
- wc_SipHashFinal

Return:

- 0 初期化に成功したときに返されます
- BAD_FUNC_ARG Siphash またはキーが NULL のときに返されます
- BAD_FUNC_ARG OUTSZ が 8 でも 16 でもない場合に返されます

C.50.2.2 function wc_SipHashUpdate

```
int wc_SipHashUpdate(
    SipHash * siphash,
    const unsigned char * in,
    word32 inSz
)
```

長さ LEN の提供されたバイト配列を絶えずハッシュするように呼び出すことができます。

Parameters:

- **siphash** Macing に使用するサイプハッシュ構造へのポインタ
- **in** マイートするデータ *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");
}
```

See:

- wc_SipHash
- wc_InitSipHash
- wc_SipHashFinal

Return:

- 0 Mac にデータを追加したら、返されます
- BAD_FUNC_ARG Siphash が null のとき返されました
- BAD_FUNC_ARG inne が null のとき返され、Insz はゼロではありません

C.50.2.3 function wc_SipHashFinal

```
int wc_SipHashFinal(
    SipHash * siphash,
    unsigned char * out,
    unsigned char outSz
)
```

データの Macing を確定します。結果が出入りする。

Parameters:

- **siphash** Macing に使用するサイプハッシュ構造へのポインタ
- **out** MAC 値を保持するためのバイト配列 *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");
}
```

See:

- wc_SipHash
- wc_InitSipHash
- wc_SipHashUpdate

Return:

- 0 ファイナライズに成功したときに返されます。
- BAD_FUNC_ARG Siphash の OUT が NULL のときに返されます
- BAD_FUNC_ARG OUTSZ が初期化された値と同じではない場合に返されます

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
)
```

この機能は Siphash を使用してデータを 1 ショットして、キーに基づいて MAC を計算します。

Parameters:

- **key** 16 バイト配列へのポインタ
- **in** マイートするデータ
- **inSz** マイクされるデータのサイズ
- **out** MAC 値を保持するためのバイト配列 *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");
}
```

See:

- wc_InitSipHash
- wc_SipHashUpdate
- wc_SipHashFinal

Return:

- 0 Macing に成功したときに返されました
- BAD_FUNC_ARG キーまたは OUT が NULL のときに返されます
- BAD_FUNC_ARG inne が null のとき返され、Insz はゼロではありません
- BAD_FUNC_ARG OUTSZ が 8 でも 16 でもない場合に返されます

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-ja/srp.h

C.51.1 Functions

	Name
int	**wc_SrpInit**(Srp * srp, SrpType type, SrpSide side) 使用方法のために SRP 構造体を初期化します。
void	**wc_SrpTerm**(Srp * srp) 使用後に SRP 構造リソースを解放します。
int	**wc_SrpSetUsername**(Srp * srp, const byte * username, word32 size) ユーザー名を設定します。この関数は、wc_srpinit の後に呼び出す必要があります。
int	**wc_SrpSetParams**(Srp * srp, const byte * N, word32 nSz, const byte * g, word32 gSz, const byte * salt, word32 saltSz) ユーザー名に基づいて SRP パラメータを設定します.. wc_srpsetuserName の後に呼び出す必要があります。
int	**wc_SrpSetPassword**(Srp * srp, const byte * password, word32 size) パスワードを設定します。パスワードを設定しても、SRP 構造内のパスワードデータが消去されません。クライアントは、x = h（salt + h（user：pswd））を計算し、それを認証フィールドに格納します。この関数は、wc_srpsetparams の後に呼び出されなければならず、クライアント側のみです。
int	**wc_SrpSetVerifier**(Srp * srp, const byte * verifier, word32 size) 検証者を設定します。この関数は、wc_srpsetparams の後に呼び出され、サーバー側のみです。
int	**wc_SrpGetVerifier**(Srp * srp, byte * verifier, word32 * size) 検証者を取得します。クライアントは V = g ^ x%N で検証者を計算します。この関数は、wc_srpsetpassword の後に呼び出され、クライアント側のみです。

	Name
int	**wc_SrpSetPrivate**(Srp * srp, const byte * priv, word32 size) プライベートのエフェラル値を設定します。プライベートの一時的な値は、クライアント側の A として知られています。サーバー側の and random () b。b = random () この関数は、ユニットテストケース、または開発者が外部ランダムソースを使用してエフェメラル値を設定したい場合は便利です。この関数は、WC_SRPGetPublic の前に呼び出されることがあります。
int	**wc_SrpGetPublic**(Srp * srp, byte * pub, word32 * size) 公共の一時的な値を取得します。公共の一時的な値は、クライアント側の A として知られています。サーバ側の A = g ^ A%n b。B =（k * v + (g b%n)）%n wc_srpsetpassword または wc_srpsetverifier の後に呼び出す必要があります。関数 WC_SRPSetPrivate は、WC_SRPGetPublic の前に呼び出されることがあります。
int	**wc_SrpComputeKey**(Srp * srp, byte * clientPubKey, word32 clientPubKeySz, byte * serverPubKey, word32 serverPubKeySz) セッションキーを計算します。成功後に SRP-> キーでキーをアクセスできます。
int	**wc_SrpGetProof**(Srp * srp, byte * proof, word32 * size) 証明を取得します。この関数は、wc_srpcomputekey の後に呼び出す必要があります。
int	**wc_SrpVerifyPeersProof**(Srp * srp, byte * proof, word32 size) ピアプルーフを確認します。この関数は、WC_SRPGetSessionKey の前に呼び出す必要があります。

C.51.2 Functions Documentation

C.51.2.1 function wc_SrpInit

```
int wc_SrpInit(
    Srp * srp,
    SrpType type,
    SrpSide side
)
```

使用方法のために SRP 構造体を初期化します。

Parameters:

- **srp** 初期化される SRP 構造。
- **type** 使用するハッシュ型。*Example*

```
Srp srp;
if (wc_SrpInit(&srp, SRP_TYPE_SHA, SRP_CLIENT_SIDE) != 0)
{
    // Initialization error
}
```

```
else
{
    wc_SrpTerm(&srp);
}
```

See:

- wc_SrpTerm
- wc_SrpSetUsername

Return:

- 0 成功しています。
- BAD_FUNC_ARG SRP などの引数が NULL または SRPSIDE の問題がある場合は、SRP_CLIENT_SIES または SRP_SERVER_SIED では問題がある場合に返します。
- NOT_COMPILED_IN タイプが引数として渡されたが、WolfCrypt ビルドに設定されていない場合。
- <0 エラー時に。

C.51.2.2 function wc_SrpTerm

```
void wc_SrpTerm(
    Srp * srp
)
```

使用後に SRP 構造リソースを解放します。

See: wc_SrpInit

Return: none いいえ返します。*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
)
```

ユーザー名を設定します。この関数は、wc_srpinit の後に呼び出す必要があります。

Parameters:

- **srp** SRP 構造
- **username** ユーザー名を含むバッファ。*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);
```

See:

- wc_SrpInit
- wc_SrpSetParams
- wc_SrpTerm

Return:

- 0 ユーザー名は正常に設定されました。
- BAD_FUNC_ARG: srp または username が null の場合に返します。
- MEMORY_E: SRP-> ユーザーにメモリを割り当てる問題がある場合
- < 0：エラー。

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
)
```

ユーザー名に基づいて SRP パラメータを設定します.. wc_srpsetuserName の後に呼び出す必要があります。

Parameters:

- **srp** SRP 構造
- **N** 弾性率 n = 2q + 1、[q、n] はプリムです。
- **nSz** n サイズをバイト単位で。
- **g** ジェネレータモジュロ N.
- **gSz** バイト数の G サイズ
- **salt** 小さいランダムなソルト。各ユーザー名に特有のものです。*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);
```

See:

- wc_SrpInit
- wc_SrpSetUsername

- wc_SrpTerm

Return:

- 0 成功
- BAD_FUNC_ARG SRP、N、G、または SALT が NULL の場合、または NSZ <GSZ の場合は返します。
- SRP_CALL_ORDER_E wc_srpsetuserName の前に wc_srpsetparams が呼び出された場合、返します。
- <0 エラー

C.51.2.5 function wc_SrpSetPassword

```
int wc_SrpSetPassword(
    Srp * srp,
    const byte * password,
    word32 size
)
```

パスワードを設定します。パスワードを設定しても、SRP 構造内のパスワードデータが消去されません。クライアントは、x = h（salt + h（user：pswd））を計算し、それを認証フィールドに格納します。この関数は、wc_srpsetparams の後に呼び出されなければならず、クライアント側のみです。

Parameters:

- **srp** SRP 構造
- **password** パスワードを含むバッファ。*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);
```

See:

- wc_SrpInit
- wc_SrpSetUsername
- wc_SrpSetParams

Return:

- 0 成功
- BAD_FUNC_ARG srp または password が null の場合、または srp-> side が srp_client_side に設定されていない場合。
- SRP_CALL_ORDER_E WC_SRPSETPASSWORD が順不同で呼び出されたときに戻ります。

- <0 エラー

C.51.2.6 function wc_SrpSetVerifier

```
int wc_SrpSetVerifier(
    Srp * srp,
    const byte * verifier,
    word32 size
)
```

検証者を設定します。この関数は、wc_srpsetparams の後に呼び出され、サーバー側のみです。

Parameters:

- **srp** SRP 構造
- **verifier** 検証者を含む構造体。*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);
```

See:

- wc_SrpInit
- wc_SrpSetParams
- wc_SrpGetVerifier

Return:

- 0 成功
- BAD_FUNC_ARG SRP または Verifier が NULL または SRP-> IS の場合、SRP_SERVER_SIED ではなく返されます。
- <0 エラー

C.51.2.7 function wc_SrpGetVerifier

```
int wc_SrpGetVerifier(
    Srp * srp,
    byte * verifier,
    word32 * size
)
```

検証者を取得します。クライアントは V = g ^ x%N で検証者を計算します。この関数は、wc_srpsetpassword の後に呼び出され、クライアント側のみです。

Parameters:

- **srp** SRP 構造
- **verifier** 検証者を書き込むためのバッファー。*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);
```

See:

- wc_SrpSetVerifier
- wc_SrpSetPassword

Return:

- 0 成功
- BAD_FUNC_ARG SRP、Verifier、または Size が NULL の場合、または SRP-> SIDE が SRP_CLIENT_SIED ではない場合に返されます。
- SRP_CALL_ORDER_E WC_SRPGetverifier が順不同で呼び出された場合に返されます。
- <0 エラー

C.51.2.8 function wc_SrpSetPrivate

```
int wc_SrpSetPrivate(
    Srp * srp,
    const byte * priv,
    word32 size
)
```

プライベートのエフェラル値を設定します。プライベートの一時的な値は、クライアント側の A として知られています。サーバー側の and random () b。b = random () この関数は、ユニットテストケース、または開発者が外部ランダムソースを使用してエフェメラル値を設定したい場合は便利です。この関数は、WC_SRPGetPublic の前に呼び出されることがあります。

Parameters:

- **srp** SRP 構造
- **priv** 一時的な値。*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);
```

See: wc_SrpGetPublic

Return:

- 0 成功
- BAD_FUNC_ARG SRP、Private、または Size が NULL の場合に返されます。
- SRP_CALL_ORDER_E WC_SRPSetPrivate が順不同で呼び出された場合に返されます。
- <0 エラー

C.51.2.9 function wc_SrpGetPublic

```
int wc_SrpGetPublic(
    Srp * srp,
    byte * pub,
    word32 * size
)
```

公共の一時的な値を取得します。公共の一時的な値は、クライアント側の A として知られています。サーバ側の A = g ^ A%n b。B = (k * v + (g b%n))%n wc_srpsetpassword または wc_srpsetverifier の後に呼び出す必要があります。関数 WC_SRPSetPrivate は、WC_SRPGetPublic の前に呼び出されることがあります。

Parameters:

- **srp** SRP 構造
- **pub** パブリックエフェラル値を書き込むためのバッファ。*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);
```

See:

- wc_SrpSetPrivate
- wc_SrpSetPassword
- wc_SrpSetVerifier

Return:

- 0 成功
- BAD_FUNC_ARG srp、pub、または size が null の場合に返されます。
- SRP_CALL_ORDER_E WC_SRPGetPublic が順不同で呼び出された場合に返されます。
- BUFFER_E サイズ <srp.n の場合は返しました。
- <0 エラー

C.51.2.10 function wc_SrpComputeKey

```
int wc_SrpComputeKey(
    Srp * srp,
    byte * clientPubKey,
    word32 clientPubKeySz,
    byte * serverPubKey,
    word32 serverPubKeySz
)
```

セッションキーを計算します。成功後に SRP-> キーでキーをアクセスできます。

Parameters:

- **srp** SRP 構造
- **clientPubKey** クライアントの公共の一時的な価値。
- **clientPubKeySz** クライアントの公共の一時的な値のサイズ。
- **serverPubKey** サーバーの一般の一時的な値。*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);
```

See: wc_SrpGetPublic

Return:

- 0 成功
- BAD_FUNC_ARG SRP、ClientPubKey、または ServerPubKey の場合、または ClientPubkeysz または ServerPubKeysz が 0 の場合に返されます。
- SRP_CALL_ORDER_E WC_SRPComputeKey が順不同で呼び出された場合に返されます。
- <0 エラー

C.51.2.11 function wc_SrpGetProof

```
int wc_SrpGetProof(
    Srp * srp,
    byte * proof,
    word32 * size
)
```

証明を取得します。この関数は、wc_srpcomputekey の後に呼び出す必要があります。

Parameters:

- **srp** SRP 構造
- **proof** ピアプルーフ。*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
}
```

See: wc_SrpComputeKey

Return:

- 0 成功
- BAD_FUNC_ARG SRP、PROV、または SIZE が NULL の場合に返します。
- BUFFER_E サイズが SRP-> Type のハッシュサイズより小さい場合に返します。
- <0 エラー

C.51.2.12 function wc_SrpVerifyPeersProof

```
int wc_SrpVerifyPeersProof(
    Srp * srp,
    byte * proof,
    word32 size
)
```

ピアプルーフを確認します。この関数は、WC_SRPGetSessionKey の前に呼び出す必要があります。

Parameters:

- **srp** SRP 構造
- **proof** ピアプルーフ。*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
}
```

See:

- wc_SrpGetSessionKey
- wc_SrpGetProof
- wc_SrpTerm

Return:

- 0 成功
- <0 エラー

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-ja/ssl.h

C.52.1 Functions

	Name
WOLFSSL_METHOD *	**wolfDTLSv1_2_client_method_ex**(void * heap) この関数は DTLS v1.2 クライアントメソッドを初期化します。
WOLFSSL_METHOD *	**wolfSSLv23_method**(void) この関数は、wolfSSLv23_client_method と同様に WOLFSSL_METHOD を返します（サーバー/クライアント）。
WOLFSSL_METHOD *	**wolfSSLv3_server_method関数は、アプリケーションがサーバーであることを示すために使用され、SSL3.0 プロトコルのみをサポートします。この関数は、wolfSSL_CTX_new() を使用して SSL/TLS コンテキストを作成するときに使用される新しい WOLFSSL_METHOD 構造体のメモリを割り当てて初期化します。
WOLFSSL_METHOD *	**wolfSSLv3_client_method関数は、アプリケーションがクライアントであり、SSL 3.0 プロトコルのみをサポートすることを示すために使用されます。この関数は、wolfSSL_CTX_new() を使用して SSL/TLS コンテキストを作成するときに使用される新しい WOLFSSL_METHOD 構造体のメモリを割り当てて初期化します。
WOLFSSL_METHOD *	**wolfTLSv1_server_method関数は、アプリケーションがサーバーであることを示すために使用され、TLS 1.0 プロトコルのみをサポートします。この関数は、wolfSSL_ctx_new() を使用して SSL/TLS コンテキストを作成するときに使用される新しい WOLFSSL_METHOD 構造体のメモリを割り当てて初期化します。

	Name
WOLFSSL_METHOD *	**wolfTLSv1_client_method**(void)wolftlsv1_client_method() 関数は、アプリケーションがクライアントであり、TLS 1.0 プロトコルのみをサポートすることを示すために使用されます。この関数は、wolfSSL_ctx_new() を使用して SSL/TLS コンテキストを作成するときに使用される新しい WOLFSSL_METHOD 構造体のメモリを割り当てて初期化します。
WOLFSSL_METHOD *	**wolfTLSv1_1_server_method関数は、アプリケーションがサーバーであることを示すために使用され、TLS 1.1 プロトコルのみをサポートします。この関数は、wolfSSL_ctx_new() を使用して SSL/TLS コンテキストを作成するときに使用される新しい WOLFSSL_METHOD 構造体のメモリを割り当てて初期化します。
WOLFSSL_METHOD *	**wolfTLSv1_1_client_method関数は、アプリケーションがクライアントであり、TLS 1.0 プロトコルのみをサポートすることを示すために使用されます。この関数は、wolfSSL_ctx_new() を使用して SSL/TLS コンテキストを作成するときに使用される新しい WOLFSSL_METHOD 構造体のメモリを割り当てて初期化します。
WOLFSSL_METHOD *	**wolfTLSv1_2_server_method関数は、アプリケーションがサーバーであることを示すために使用され、TLS 1.2 プロトコルのみをサポートします。この関数は、wolfSSL_ctx_new() を使用して SSL/TLS コンテキストを作成するときに使用される新しい WOLFSSL_METHOD 構造体のメモリを割り当てて初期化します。
WOLFSSL_METHOD *	**wolfTLSv1_2_client_method関数は、アプリケーションがクライアントであり、TLS 1.2 プロトコルのみをサポートすることを示すために使用されます。この関数は、wolfSSL_ctx_new() を使用して SSL/TLS コンテキストを作成するときに使用される新しい Wolfssl_method 構造体のメモリを割り当てて初期化します。
WOLFSSL_METHOD *	**wolfDTLSv1_client_method**(void)wolfdtlsv1_client_method() 関数は、アプリケーションがクライアントであり、DTLS 1.0 プロトコルのみをサポートすることを示すために使用されます。この関数は、wolfSSL_ctx_new() を使用して SSL/TLS コンテキストを作成するときに使用される新しい WOLFSSL_METHOD 構造体のメモリを割り当てて初期化します。この関数は、WolfSSL が DTLS サポート（–enable-dtls、または WOLFSSL_DTLS を定義することによって）ビルドされている場合にのみ使用できます。

	Name
WOLFSSL_METHOD *	**wolfDTLSv1_server_method関数は、アプリケーションがサーバーであることを示すために使用され、DTLS 1.0 プロトコルのみをサポートします。この関数は、wolfSSL_ctx_new() を使用して SSL/TLS コンテキストを作成するときに使用される新しい WOLFSSL_METHOD 構造体のメモリを割り当てて初期化します。この関数は、WolfSSL が DTLS サポート（–enable-dtls、または WOLFSSL_DTLS マクロを定義することによって）ビルドされている場合にのみ使用できます。
WOLFSSL_METHOD *	**wolfDTLSv1_2_server_method**(void)wolfDTLSv1_2_server_method() 関数はサーバ側用に WOLFSSL_METHOD 構造体のメモリを割り当てて初期化します。
WOLFSSL_METHOD *	**wolfDTLSv1_3_server_method関数はアプリケーションがサーバーであることを示すために使用され、DTLS 1.3 プロトコルのみをサポートします。この関数は、wolfSSL_ctx_new() を使用して SSL/TLS コンテキストを作成するときに使用される新しい WOLFSSL_METHOD 構造体のメモリを割り当てて初期化します。この関数は、WolfSSL が DTLS サポート（–enable-dtls13、または WOLFSSL_DTLS13 を定義することによって）ビルドされている場合にのみ使用できます。
WOLFSSL_METHOD *	**wolfDTLSv1_3_client_method関数はアプリケーションがクライアントであることを示すために使用され、DTLS 1.3 プロトコルのみをサポートします。この関数は、wolfSSL_ctx_new() を使用して SSL/TLS コンテキストを作成するときに使用される新しい WOLFSSL_METHOD 構造体のメモリを割り当てて初期化します。この関数は、WolfSSL が DTLS サポート（–enable-dtls13、または WOLFSSL_DTLS13 を定義することによって）ビルドされている場合にのみ使用できます。
WOLFSSL_METHOD *	**wolfDTLS_server_method関数はアプリケーションがサーバーであることを示すために使用され、可能な限り高いバージョン最小バージョンの DTLS プロトコルをサポートします。デフォルトの最小バージョンは WOLFSSL_MIN_DTLS_DOWNGRADE マクロでの指定をもとにしていて、実行時に wolfSSL_SetMinVersion() で変更することができます。この関数は、wolfSSL_ctx_new() を使用して SSL/TLS コンテキストを作成するときに使用される新しい WOLFSSL_METHOD 構造体のメモリを割り当てて初期化します。この関数は、WolfSSL が DTLS サポート（–enable-dtls、または WOLFSSL_DTLS を定義することによって）ビルドされている場合にのみ使用できます。

	Name
WOLFSSL_METHOD *	**wolfDTLS_client_method関数は アプリケーションがクライアントであることを示すために使用され、可能な限り高いバージョン最小バージョンの DTLS プロトコルをサポートします。デフォルトの最小バージョンは WOLFSSL_MIN_DTLS_DOWNGRADE マクロでの指定をもとにしていて、実行時に wolfSSL_SetMinVersion() で変更することができます。この関数は、wolfSSL_ctx_new() を使用して SSL/TLS コンテキストを作成するときに使用される新しい WOLFSSL_METHOD 構造体のメモリを割り当てて初期化します。この関数は、wolfSSL が DTLS サポート（–enable-dtls、または WOLFSSL_DTLS を定義することによって）ビルドされている場合にのみ使用できます。
int	**wolfSSL_use_old_poly**(WOLFSSL * ssl, int value)Chacha-Poly Aead Construction の最初のリリースと新しいバージョンの間にいくつかの違いがあるため、古いバージョンを使用してサーバー/クライアントと通信するオプションを追加しました。デフォルトでは、wolfSSL は新しいバージョンを使用します。
int	**wolfSSL_dtls_import**(WOLFSSL * ssl, unsigned char * buf, unsigned int sz)wolfSSL_dtls_import() 関数はシリアライズされたセッション状態を解析するために使われます。これにより、ハンドシェイクが完了した後に接続をピックアップすることができます。
int	**wolfSSL_tls_import**(WOLFSSL * ssl, const unsigned char * buf, unsigned int sz) シリアライズされた TLS セッションをインポートします。警告：buf には、状態に関する機密情報が含まれており、保存されている場合は保存する前に暗号化されるのが最善です。追加のデバッグ情報をマクロ WOLFSSL_SESSION_EXPORT_DEBUG を定義して表示できます。
int	**wolfSSL_CTX_dtls_set_export**(WOLFSSL_CTX * ctx, wc_dtls_export func)wolfSSL_CTX_dtls_set_export() 関数はセッションをエクスポートするためのコールバック関数を設定します。以前に格納されているエクスポート機能をクリアするためにパラメータ func に NULL を渡すことが許されます。サーバー側で使用され、ハンドシェイクが完了した直後に設定したコールバック関数が呼び出されます。

	Name
int	**wolfSSL_dtls_set_export**(WOLFSSL * ssl, wc_dtls_export func)wolfSSL_dtls_set_export() 関数はセッションをエクスポートする際に呼び出すコールバック関数を登録します。以前に登録されているエクスポート関数をクリアするために使うこともできます。サーバー側で使用され、ハンドシェイクが完了した直後に設定したコールバック関数が呼び出されます。
int	**wolfSSL_dtls_export**(WOLFSSL * ssl, unsigned char * buf, unsigned int * sz)wolfSSL_dtls_export() 関数は提供されたバッファへセッションをシリアル化します。セッションをエクスポートするための関数コールバックを使用するよりもメモリオーバーヘッドを減らすことができます。関数に渡された引数 buf が NULL の場合、sz には WolfSSL セッションのシリアライズに必要なバッファのサイズが設定されます。
int	**wolfSSL_tls_export**(WOLFSSL * ssl, unsigned char * buf, unsigned int * sz) シリアライズされた TLS セッションをエクスポートします。ほとんどの場合、wolfSSL_tls_export の代わりに wolfssl_get1_session を使用する必要があります。追加のデバッグ情報をマクロ WOLFSSL_SESSION_EXPORT_DEBUG を定義して表示できます。警告：buf には、状態に関する機密情報が含まれており、保存する場合は保存する前に暗号化されるのが最善です。
int	**wolfSSL_CTX_load_static_memory**(WOLFSSL_CTX ** ctx, wolfSSL_method_func method, unsigned char * buf, unsigned int sz, int flag, int max) この関数は CTX 用に静的メモリ領域を設定する目的に使用されます。設定された静的メモリ領域は CTX の有効期間および CTX から作成された全ての SSL オブジェクトに使用されます。引数 ctx に NULL を渡し、wolfSSL_method_func 関数を渡すことによって、CTX 自体の作成も静的メモリを使用します。wolfssl_method_func は次のシグネチャとなっています:wolfssl_method（ wolfssl_method_func)(void *heap)。引数 max に 0 を渡すと、設定されていないものとして動作し、最大の同時使用制限が適用されません。引数 flag に渡した値によって、メモリの使用方法と動作が決まります。利用可能なフラグ値は次のとおりです：0 - デフォルトの一般メモリ、WOLFMEM_IO_POOL - メッセージの受送信の際の入出力バッファとして使用され渡されたバッファ内のすべてのメモリが IO に使用されます、WOLFMEM_IO_FIXED - WOLFMEM_IO_POOL と同じですが、各 SSL は 2 つのバッファを自分のライフタイムの間保持して使用します。WOLFMEM_TRACK_STATS - 各 SSL は実行中にメモリ使用統計を追跡します。

	Name
int	**wolfSSL_CTX_is_static_memory**(WOLFSSL_CTX * ctx, WOLFSSL_MEM_STATS * mem_stats) この関数は現時点の接続に関する振る舞いの変更は行いません。静的メモリ使用量に関する情報を収集するためにのみ使用されます。
int	**wolfSSL_is_static_memory**(WOLFSSL * ssl, WOLFSSL_MEM_CONN_STATS * mem_stats)wolfSSL_is_static_memory 関数は SSL の静的メモリ使用量に関する情報を集めます。戻り値は、静的メモリが使用されているかどうかを示します。引数 ssl の上位の WOLFSSL_CTX に静的メモリを使用するように指定してあり、WOLFMEM_TRACK_STATS が定義されている場合に 引数 mem_stats に情報がセットされます。
int	**wolfSSL_CTX_use_certificate_file**(WOLFSSL_CTX * ctx, const char * file, int format) この関数は証明書ファイルを SSL コンテキスト (WOLFSSL_CTX) にロードします。ファイルは引数 file によって提供されます。引数 format は、ファイルのフォーマットタイプ（SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM）を指定します。適切な使用法の例をご覧ください。
int	**wolfSSL_CTX_use_PrivateKey_file**(WOLFSSL_CTX * ctx, const char * file, int format) この関数は、秘密鍵ファイルを SSL コンテキスト (WOLFSSL_CTX) にロードします。ファイルは引数 file によって提供されます。引数 format は、次のファイルのフォーマットタイプを指定します：SSL_FILETYPE_ASN1　あるいは SSL_FILETYPE_PEM。適切な使用法の例をご覧ください。外部キーストアを使用し、秘密鍵を持っていない場合は、代わりに公開鍵を入力して crypto コールバックを登録して署名を処理することができます。このためには、crypto コールバックまたは PK コールバックを使用したコンフィギュレーションでビルドします。crypto コールバックを有効にするには、–enable-cryptocb または WOLF_CRYPTO_CB マクロを使用し、wc_CryptoCb_RegisterDevice を使用して暗号コールバックを登録し、wolfSSL_CTX_SetDevId を使用して関連する devid を設定します。

	Name
int	**wolfSSL_CTX_load_verify_locations**(WOLFSSL_CTX * ctx, const char * file, const char * path) この関数は、PEM 形式の CA 証明書ファイルを SSL コンテキスト (WOLFSSL_CTX) にロードします。これらの証明書は、信頼できるルート証明書として扱われ、SSL ハンドシェイク中にピアから受信した証明書を検証するために使用されます。引数 file によって提供されるルート証明書ファイルは、単一の証明書または複数の証明書を含むファイルでの場合があります。複数の CA 証明書が同じファイルに含まれている場合、wolfSSL はファイルに表示されているのと同じ順序でそれらをロードします。引数 path は、信頼できるルート CA の証明書を含むディレクトリの名前へのポインタです。引数 file が NULL ではない場合、パスが必要でない場合は NULL として指定できます。引数 path が指定されていてかつ NO_WOLFSSL_DIR が定義されていない場合には、wolfSSL ライブラリは指定されたディレクトリに存在するすべての CA 証明書をロードします。この関数はディレクトリ内のすべてのファイルをロードしようとします。この関数は、ヘッダーに “—–BEGIN CERTIFICATE—–” を持つ PEM フォーマットされた CERT_TYPE ファイルを期待しています。
int	**wolfSSL_CTX_load_verify_locations_ex**(WOLFSSL_CTX * ctx, const char * file, const char * path, unsigned int flags) この関数は、PEM 形式の CA 証明書ファイルを SSL コンテキスト (WOLFSSL_CTX) にロードします。これらの証明書は、信頼できるルート証明書として扱われ、SSL ハンドシェイク中にピアから受信した証明書を検証するために使用されます。引数 file によって提供されるルート証明書ファイルは、単一の証明書または複数の証明書を含むファイルでの場合があります。複数の CA 証明書が同じファイルに含まれている場合、wolfSSL はファイルに表示されているのと同じ順序でそれらをロードします。引数 path は、信頼できるルート CA の証明書を含むディレクトリの名前へのポインタです。引数 file が NULL ではない場合、パスが必要でない場合は NULL として指定できます。引数 path が指定されていてかつ NO_WOLFSSL_DIR が定義されていない場合には、wolfSSL ライブラリは指定されたディレクトリに存在するすべての CA 証明書をロードします。この関数は引数 flags に基づいてディレクトリ内のすべてのファイルをロードしようとします。この関数は、ヘッダーに “—–BEGIN CERTIFICATE—–” を持つ PEM フォーマットされた CERT_TYPE ファイルを期待しています。

	Name
const char **	**wolfSSL_get_system_CA_dirs**(word32 * num) この関数は、wolfSSL_CTX_load_system_CA_certs が呼び出されたときに、wolfSSL がシステム CA 証明書を検索するディレクトリを表す文字列の配列へのポインタを返します。
int	**wolfSSL_CTX_load_system_CA_certs**(WOLFSSL_CTX * ctx) この関数は、CA 証明書を OS 依存の CA 証明書ストアから WOLFSSL_CTX にロードしようとします。ロードされた証明書は信頼されます。サポートおよびテストされているプラットフォームは、Linux(Debian、Ubuntu、Gentoo、Fedora、RHEL)、Windows 10/11、Android、Apple OS X、iOS です。
int	**wolfSSL_CTX_trust_peer_cert**(WOLFSSL_CTX * ctx, const char * file, int type) この関数は、TLS/SSL ハンドシェイクを実行するときにピアを検証するために使用する証明書をロードします。ハンドシェイク中に送信されたピア証明書は、この関数で指定された証明書の SKID と署名を比較することによって検証されます。これら 2 つのことが一致しない場合は、ピア証明書の検証にはロードされた CA 証明書が使用されます。この機能は WOLFSSL_TRUST_PEER_CERT マクロを定義することで機能を有効にできます。適切な使用法は例をご覧ください。
int	**wolfSSL_CTX_use_certificate_chain_file**(WOLFSSL_CTX * ctx, const char * file) この関数は、証明書チェーンを SSL コンテキスト (WOLFSSL_CTX) にロードします。証明書チェーンを含むファイルは引数 file によって提供され、PEM 形式の証明書を含める必要があります。この関数は、最大 MAX_CHAIN_DEPTH（既定で 9、internal.h で定義されている）数の証明書を処理します。この数にはサブジェクト証明書を含みます。
int	**wolfSSL_CTX_use_RSAPrivateKey_file**(WOLFSSL_CTX * ctx, const char * file, int format) この関数は、SSL 接続で使用されている RSA 秘密鍵を SSL コンテキスト (WOLFSSL_CTX) にロードします。この関数は、wolfSSL が OpenSSL 互換 API が有効（–enable_openSSLExtra、#define OPENSSL_EXTRA）でコンパイルされている場合にのみ利用可能で、より一般的に使用されている wolfSSL_CTX_use_PrivateKey_file() 関数と同じです。ファイル引数には、RSA 秘密鍵ファイルへのポインタが、引数 format で指定された形式で含まれています。
long	**wolfSSL_get_verify_depth**(WOLFSSL * ssl) この関数は、有効なセッション（NULL 以外の引数 ssl）が指定された場合に、デフォルトで 9 の最大チェーン深度を返します。

	Name
long	**wolfSSL_CTX_get_verify_depth**(WOLFSSL_CTX * ctx) この関数は、WOLFSSL_CTX 構造体構造を使用して証明書チェーン深度を取得します。
int	**wolfSSL_use_certificate_file**(WOLFSSL * ssl, const char * file, int format) この関数は証明書ファイルを SSL セッション（WOLFSSL 構造体）にロードします。証明書ファイルはファイル引数によって提供されます。引数 format は、ファイルのフォーマットタイプ（SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM）を指定します。
int	**wolfSSL_use_PrivateKey_file**(WOLFSSL * ssl, const char * file, int format) この関数は、秘密鍵ファイルを SSL セッション（WOLFSSL 構造体）にロードします。鍵ファイルは引数 file によって提供されます。引数 format は、ファイルのタイプ（SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM が指定可）を指定します。外部キーストアを使用し、秘密鍵を持っていない場合は、代わりに公開鍵を入力して CryPro コールバックを登録して署名を処理することができます。このためには、Crypto コールバックまたは PK コールバックを使用したコンフィグレーションでビルドします。Crypto コールバックを有効にするには、–enable-cryptocb または WOLF_CRYPTO_CB マクロを使用してビルドし、wc_CryptoCb_RegisterDevice を使用して暗号コールバックを登録し、wolfSSL_SetDevId を使用して関連する devId を設定します。
int	**wolfSSL_use_certificate_chain_file**(WOLFSSL * ssl, const char * file) この関数は、証明書チェーンを SSL セッション WOLFSSL 構造体）にロードします。証明書チェーンを含むファイルは引数 file によって提供され、PEM 形式の証明書を含める必要があります。この関数は、MAX_CHAIN_DEPTH（既定で 9、internal.h で定義されている）証明書に加えて、サブジェクト証明書を処理します。
int	**wolfSSL_use_RSAPrivateKey_file**(WOLFSSL * ssl, const char * file, int format) この関数は、SSL 接続で使用されている RSA 秘密鍵を SSL セッション（WOLFSSL 構造体）にロードします。この関数は、wolfSSL が OpenSSL 互換 API を有効（–enable_openSSlExtra、#define OPENSSL_EXTRA）でビルドされている場合にのみ利用可能で、より一般的に使用される wolfSSL_use_PrivateKey_file() 関数と同じです。引数 file には、RSA 秘密鍵ファイルへのポインタが、フォーマットで指定された形式で含まれています。

	Name
int	**wolfSSL_CTX_der_load_verify_locations**(WOLFSSL_CTX * ctx, const char * file, int format) この関数は wolfSSL_CTX_load_verify_locations と似ていますが、DER フォーマットされた CA ファイルを SSL コンテキスト（WOLFSSL_CTX）にロードすることを許可します。それはまだ PEM 形式の CA ファイルをロードするためにも使用されるかもしれません。これらの証明書は、信頼できるルート証明書として扱われ、SSL ハンドシェイク中にピアから受信した証明書を検証するために使用されます。ファイル引数によって提供されるルート証明書ファイルは、単一の証明書または複数の証明書を含むファイルでも可能。複数の CA 証明書が同じファイルに含まれている場合、wolfSSL はファイルに表示されているのと同じ順序でそれらをロードします。引数 format は、証明書が SSL_FILETYPE_PEM または SSL_FILETYPE_ASN1（DER）のいずれかにある形式を指定します。wolfSSL_CTX_load_verify_locations とは異なり、この関数は特定のディレクトリパスからの CA 証明書のロードを許可しません。この関数は、wolfSSL ライブラリが WOLFSSL_DER_LOAD マクロが定義された状態でビルドされたときにのみ利用可能です。
WOLFSSL_CTX *	**wolfSSL_CTX_new**(WOLFSSL_METHOD *) この関数は、所望の SSL/TLS プロトコル用メソッド構造体を引数に取って、新しい SSL コンテキストを作成します。
WOLFSSL *	**wolfSSL_new**(WOLFSSL_CTX *) この関数はすでに作成された SSL コンテキスト (WOLFSSL_CTX) を入力として、新しい SSL セッション (WOLFSSL) を作成します。
int	**wolfSSL_set_fd**(WOLFSSL * ssl, int fd) この関数は、SSL 接続の入出力機能としてファイル記述子 (fd) を割り当てます。通常これはソケットファイル記述子になります。
int	**wolfSSL_set_dtls_fd_connected**(WOLFSSL * ssl, int fd) この関数はファイルディスクリプタ (fd) を SSL コネクションの入出力手段として設定します。通常はソケットファイルディスクリプタが指定されます。この関数は DTLS 専用の API であり、ソケットは接続済みとマークされます。したがって、与えられた fd に対する recvfrom と sendto 呼び出しでの addr と addr_len は NULL に設定されます。

	Name
int	**wolfDTLS_SetChGoodCb**(WOLFSSL * ssl, ClientHelloGoodCb cb, void * user_ctx) この関数は DTLS ClientHello メッセージが正しく処理できた際に呼び出されるコールバック関数を設定します。クッキー交換メカニズムを使用する場合 (DTLS1.2 の HelloVerifyRequest か DTLS1.3 のクッキー拡張を伴った HelloRetryRequest のいずれかを使用する場合) には、クッキー交換が成功した時点でこのコールバック関数が呼び出されます。この機能はひとつの WOLFSSL オブジェクトを新たな接続を待ち受けるリスナーとして使い, ClientHello が検証された WOLFSSL オブジェクトから絶縁させることができます。この場合の検証はクッキー交換か ClientHello が正しいフォーマットになっているかのチェックによってなされます。
char *	**wolfSSL_get_cipher_list**(int priority) この関数は引数で渡された優先順位の暗号名 (Cipher) 文字列へのポインタを返します。
int	**wolfSSL_get_ciphers**(char * buf, int len) この関数は wolfSS で有効化されている暗号名 (Cipher) を取得します。
const char *	**wolfSSL_get_cipher_name**(WOLFSSL * ssl) この関数は、引数を wolfSSL_get_cipher_name_internal に渡すことによって、DHE-RSA の形式の暗号名を取得します。
int	**wolfSSL_get_fd**(const WOLFSSL *) この関数は、SSL 接続の入出力機能として使用されるファイル記述子 (fd) を返します。通常これはソケットファイル記述子になります。
void	**wolfSSL_set_using_nonblock**(WOLFSSL * ssl, int nonblock) この関数は、WOLFSSL オブジェクトに基礎となる I/O がノンブロックであることを通知します。アプリケーションが WOLFSSL オブジェクトを作成した後、ブロッキング以外のソケットで使用する場合は、wolfssl_set_using_nonblock() を呼び出します。これにより、wolfssl オブジェクトは、EWOULDBLOCK を受信することを意味します。
int	**wolfSSL_get_using_nonblock**(WOLFSSL *) この機能により、wolfSSL がノンブロッキング I/O を使用しているかどうかをアプリケーションが判断できます。wolfSSL がノンブロッキング I/O を使用している場合、この関数は 1 を返します。アプリケーションが WOLFSSL オブジェクトを生成した後に wolfSSL_set_using_nonblock() を呼び出してノンブロッキングソケットを使うとこの関数は１を返します。これにより、WOLFSSL オブジェクトは、recevfrom がタイムアウトせず代わりに EWOULDBLOCK を受信するようになります。

	Name
int	**wolfSSL_writeは、ブロックとノンブロッキング I/O の両方で動作します。基礎となる入出力がノンブロッキングに設定されている場合、wolfSSL_write() が要求を満たすことができなかったら wolfSSL_write() は関数呼び出しからすぐに戻ります。この場合、wolfSSL_get_error() の呼び出しは SSL_ERROR_WANT_READ または SSL_ERROR_WANT_WRITE のいずれかを返します。その結果、基礎となる I/O が準備ができたら、呼び出し側プロセスは wolfssl_write() への呼び出しを繰り返す必要があります。基礎となる入出力がブロックされている場合、WolfSSL_WRITE() は、サイズ SZ のバッファデータが完全に書かれたかエラーが発生したら、戻るだけです。
int	**wolfSSL_read**(WOLFSSL * ssl, void * data, int sz) この関数は、SSL セッション (ssl) の内部読み取りバッファから sz バイトをバッファデータに読み出します。読み取られたバイトは内部受信バッファから削除されます。必要に応じて、wolfSSL_read() の呼び出し時点ではまだ wolfSSL_connect() または wolfSSL_accept() がまだ呼び出されていない場合、SSL/TLS セッションをネゴシエートします。SSL/TLS プロトコルは、最大サイズの SSL レコードを使用します（最大レコードサイズは /wolfssl/internal.h）。そのため、wolfSSL は、レコードを処理および復号することができる前に、SSL レコード全体を内部的に読み取る必要があります。このため、wolfSSL_read() への呼び出しは、呼び出し時に復号された最大バッファサイズを返すことができます。検索され、次回の wolfSSL_read() への呼び出しで復号される内部 wolfSSL 受信バッファで待機していない追加の復号データがあるかもしれません。sz が内部読み取りバッファ内のバイト数より大きい場合、wolfSSL_read() は内部読み取りバッファで使用可能なバイトを返します。BYTES が内部読み取りバッファにバッファされていない場合は、wolfSSL_read() への呼び出しは次のレコードの処理をトリガーします。

	Name
int	**wolfSSL_peek**(WOLFSSL * ssl, void * data, int sz) この関数は SSL セッション（SSL）内部読み取りバッファから SZ バイトをバッファデータにコピーします。この関数は、内部 SSL セッション受信バッファ内のデータが削除されていないか変更されていないことを除いて、wolfssl_read() と同じです。必要に応じて、wolfssl_read() のように、wolfssl_peek() はまだ wolfssl_connect() または wolfssl_accept() によってまだ実行されていない場合、wolfssl_peek() は SSL / TLS セッションをネゴシエートします。SSL/TLS プロトコルは、最大サイズの SSL レコードを使用します（最大レコードサイズは /wolfssl/internal.h）。そのため、WolfSSL は、レコードを処理および復号化することができる前に、SSL レコード全体を内部的に読み取る必要があります。このため、wolfssl_peek() への呼び出しは、呼び出し時に復号化された最大バッファサイズを返すことができます。wolfssl_peek()/ wolfssl_read() への次の呼び出しで検索および復号化される内部 WolfSSL 受信バッファ内で待機していない追加の復号化データがあるかもしれません。SZ が内部読み取りバッファ内のバイト数よりも大きい場合、SSL_PEEK() は内部読み取りバッファで使用可能なバイトを返します。バイトが内部読み取りバッファにバッファされていない場合、Wolfssl_peek() への呼び出しは次のレコードの処理をトリガーします。
int	**wolfSSL_accept は、ブロックとノンブロッキング I/O の両方で動作します。基礎となる入出力がノンブロッキングである場合、wolfSSL_accept() は、基礎となる I/O が wolfSSL_accept の要求を満たすことができなかったときに戻ります。この場合、wolfSSL_get_error() への呼び出しは SSL_ERROR_WANT_READ または SSL_ERROR_WANT_WRITE のいずれかを生成します。呼び出しプロセスは、読み取り可能なデータが使用可能であり、wolfSSL が停止した場所を拾うときに、wolfSSL_accept の呼び出しを繰り返す必要があります。ノンブロッキングソケットを使用する場合は、何も実行する必要がありますが、select() を使用して必要な条件を確認できます。基礎となる I/O がブロックされている場合、wolfSSL_accept() はハンドシェイクが終了したら、またはエラーが発生したら戻ります。
void	**wolfSSL_CTX_free**(WOLFSSL_CTX *) この関数は、割り当てられた WOLFSSL_CTX オブジェクトを解放します。この関数は CTX 参照数を減らし、参照カウントが 0 に達したときにのみコンテキストを解放します。
void	**wolfSSL_free**(WOLFSSL *) この関数は割り当てられた WOLFSSL オブジェクトを解放します。

	Name
int	**wolfSSL_shutdownは、ブロックとノンブロッキング I/O の両方で動作します。下層の I/O がノンブロッキングの場合、wolfSSL_shutdown() が要求を満たすことができなかった場合、wolfSSL_shutdown() はエラーを返します。この場合、wolfSSL_get_error() への呼び出しは SSL_ERROR_WANT_READ または SSL_ERROR_WANT_WRITE のいずれかを生成します。その結果、下層の I/O が準備ができたら、呼び出し側プロセスは wolfSSL_shutdown() への呼び出しを繰り返す必要があります。
int	**wolfSSL_sendは、ブロックとノンブロッキング I/O の両方で動作します。基礎となる入出力がノンブロッキングに設定されている場合、wolfSSL_send() が要求を満たすことができなかったら wolfSSL_send() は関数呼び出しからすぐに戻ります。この場合、wolfSSL_get_error() の呼び出しは SSL_ERROR_WANT_READ または SSL_ERROR_WANT_WRITE のいずれかを返します。その結果、基礎となる I/O が準備ができたら、呼び出し側プロセスは wolfSSL_send() への呼び出しを繰り返す必要があります。基礎となる入出力がブロックされている場合、wolfSSL_send() は、サイズ SZ のバッファデータが完全に書かれたかエラーが発生したら、戻るだけです。
int	**wolfSSL_recvへの次の呼び出しで検索および復号される内部 wolfSSL 受信バッファで待機していない追加の復号化されたデータがあるかもしれません。引数 sz が内部読み取りバッファ内のバイト数よりも大きい場合、wolfSSL_recv() は内部読み取りバッファで使用可能なバイトを返します。バイトが内部読み取りバッファにバッファされていない場合は、wolfSSL_recv() への呼び出しは次のレコードの処理をトリガーします。
int	**wolfSSL_get_error**(WOLFSSL * ssl, int ret) この関数は、直前の API 関数呼び出し（wolfssl_connect、wolfssl_accept、wolfssl_read、wolfssl_write など）がエラーコード（SSL_FAILURE）を呼び出した理由を表す一意のエラーコードを返します。直前の関数の戻り値は、ret を介して wolfSSL_get_error に渡されます。wolfSSL_get_error は一意のエラーコードを返します。wolfSSL_err_error_string() を呼び出して人間が読めるエラー文字列を取得することができます。詳細については、wolfSSL_err_error_string() を参照してください。
int	**wolfSSL_get_alert_history**(WOLFSSL * ssl, WOLFSSL_ALERT_HISTORY * h) この関数はアラート履歴を取得します。
int	**wolfSSL_set_sessionによって返されたオブジェクトは、アプリケーションが使用後に解放する必要があります。

	Name
WOLFSSL_SESSION *	**wolfSSL_get_session**(WOLFSSL * ssl)NO_SESSION_CACHE_REF が定義されている場合、この関数は SSL で使用されている現在のセッション（WOLFSSL_SESSION）へのポインタを返します。この関数は、WOLFSSL_SESSION オブジェクトへの永続的なポインタを返します。返されるポインタは、wolfSSL_free が呼び出されたときに解放されます。この呼び出しは、現在のセッションを検査または変更するためにのみ使用されます。セッション再開に使用する場合は、wolfSSL_get1_session() を使用することをお勧めします。NO_SESSION_CACHE_REF が定義されていない場合の後方互換性のために、この関数はローカルキャッシュに格納されている永続セッションオブジェクトポインタを返します。キャッシュサイズは有限であり、アプリケーションが wolfSSL_set_session() を呼び出す時までにセッションオブジェクトが別の SSL 接続によって上書きされる危険性があります。アプリケーションに NO_SESSION_CACHE_REF を定義し、セッション再開に wolfSSL_get1_session() を使用することをお勧めします。
void	**wolfSSL_flush_sessions**(WOLFSSL_CTX * ctx, long tm) この機能は、期限切れになったセッションキャッシュからセッションをフラッシュします。時間比較には引数 tm が使用されます。wolfSSL は現在セッションに静的テーブルを使用しているため、フラッシングは不要です。そのため、この機能は現在スタブとして存在しています。この関数は、wolfssl が OpenSSL 互換層でコンパイルされているときの OpenSSL 互換性（ssl_flush_sessions）を提供します。
int	**wolfSSL_SetServerID**(WOLFSSL * ssl, const unsigned char * id, int len, int newSession) この関数はクライアントセッションをサーバー ID と関連付けます。引数 newSession がオンの場合、既存のセッションは再利用されません。
int	**wolfSSL_GetSessionIndex**(WOLFSSL * ssl) この関数は、WOLFSSL 構造体の指定セッションインデックス値を取得します。
int	**wolfSSL_GetSessionAtIndex**(int index, WOLFSSL_SESSION * session) この関数はセッションキャッシュの指定されたインデックスのセッションを取得し、それをメモリにコピーします。WOLFSSL_SESSION 構造体はセッション情報を保持します。
WOLFSSL_X509_CHAIN *	**wolfSSL_SESSION_get_peer_chain**(WOLFSSL_SESSION * session)WOLFSSL_SESSION 構造体からピア証明書チェーンを返します。

	Name
void	**wolfSSL_CTX_set_verify**(WOLFSSL_CTX * ctx, int mode, VerifyCallback verify_callback) この関数はリモートピアの検証方法を設定し、また証明書検証コールバック関数を SSL コンテキストに登録することもできます。検証コールバックは、検証障害が発生した場合にのみ呼び出されます。検証コールバックが必要な場合は、NULL ポインタを verify_callback に使用できます。ピア証明書の検証モードは、論理的またはフラグのリストです。可能なフラグ値は次のとおりです: SSL_VERIFY_NONE -クライアントモード：クライアントはサーバーから受信した証明書を検証せず、ハンドシェイクは通常どおり続きます。-サーバーモード：サーバーはクライアントに証明書要求を送信しません。そのため、クライアント検証は有効になりません。SSL_VERIFY_PEER -クライアントモード：クライアントはハンドシェイク中にサーバーから受信した証明書を検証します。これは wolfSSL ではデフォルトでオンにされます。したがって、このオプションを使用すると効果がありません。-サーバーモード：サーバーは証明書要求をクライアントに送信し、受信したクライアント証明書を確認します。 SSL_VERIFY_FAIL_IF_NO_PEER_CERT -クライアントモード：クライアント側で使用されていない場合は効果がありません。-サーバーモード：要求されたときにクライアントが証明書の送信に失敗した場合は、サーバー側で検証が失敗します（SSL サーバーの SSL_VERIFY_PEER を使用する場合）。 SSL_VERIFY_FAIL_EXCEPT_PSK -クライアントモード：クライアント側で使用されていない場合は効果がありません。-サーバーモード：PSK 接続の場合を除き、検証は SSL_VERIFY_FAIL_IF_NO_PEER_CERT と同じです。PSK 接続が行われている場合、接続はピア証明書なしで通過します。

	Name
void	**wolfSSL_set_verify**(WOLFSSL * ssl, int mode, VerifyCallback verify_callback) この関数はリモートピアの検証方法を設定し、また証明書検証コールバック関数を WOLFSSL オブジェクトに登録することもできます。検証コールバックは、検証障害が発生した場合にのみ呼び出されます。検証コールバックが必要な場合は、NULL ポインタを verify_callback に使用できます。ピア証明書の検証モードは、論理的またはフラグのリストです。可能なフラグ値は次のとおりです: SSL_VERIFY_NONE -クライアントモード：クライアントはサーバーから受信した証明書を検証せず、ハンドシェイクは通常どおり続きます。-サーバーモード：サーバーはクライアントに証明書要求を送信しません。そのため、クライアント検証は有効になりません。SSL_VERIFY_PEER -クライアントモード：クライアントはハンドシェイク中にサーバーから受信した証明書を検証します。これは wolfSSL ではデフォルトでオンにされます。したがって、このオプションを使用すると効果がありません。-サーバーモード：サーバーは証明書要求をクライアントに送信し、受信したクライアント証明書を確認します。 SSL_VERIFY_FAIL_IF_NO_PEER_CERT -クライアントモード：クライアント側で使用されていない場合は効果がありません。-サーバーモード：要求されたときにクライアントが証明書の送信に失敗した場合は、サーバー側で検証が失敗します（SSL サーバーの SSL_VERIFY_PEER を使用する場合）。 SSL_VERIFY_FAIL_EXCEPT_PSK -クライアントモード：クライアント側で使用されていない場合は効果がありません。-サーバーモード：PSK 接続の場合を除き、検証は SSL_VERIFY_FAIL_IF_NO_PEER_CERT と同じです。PSK 接続が行われている場合、接続はピア証明書なしで通過します。
void	**wolfSSL_SetCertCbCtx**(WOLFSSL * ssl, void * ctx) この関数は、検証コールバックのためのユーザー CTX オブジェクト情報を格納します。
void	**wolfSSL_CTX_SetCertCbCtx**(WOLFSSL_CTX * ctx, void * userCtx) この関数は、検証コールバックのためのユーザー CTX オブジェクト情報を格納します。
int	**wolfSSL_pending**(WOLFSSL *) この関数は、wolfSSL_read() によって読み取られる WOLFSSL オブジェクトでバッファされているバイト数を返します。
void	**wolfSSL_load_error_strings**(void) この機能は OpenSSL API（SSL_load_error_string）との互換性の目的みで提供してあり処理は行いません。

	Name
int	**wolfSSL_library_init**(void) この関数は wolfSSL_CTX_new() 内で内部的に呼び出されます。この関数は wolfSSL_Init() のラッパーで、wolfSSL が OpenSSL 互換層でコンパイルされたときの OpenSSL API（ssl_library_init）との互換性の為に存在します。wolfSSL_init() は、より一般的に使用されている wolfSSL 初期化機能です。
int	**wolfSSL_SetDevId**(WOLFSSL * ssl, int devId) この関数は WOLFSSL オブジェクトレベルで Device Id をセットします。
int	**wolfSSL_CTX_SetDevId**(WOLFSSL_CTX * ctx, int devId) この関数は WOLFSSL_CTX レベルで Device Id をセットします。
int	**wolfSSL_CTX_GetDevId**(WOLFSSL_CTX * ctx, WOLFSSL * ssl) この関数は WOLFSSL_CTX レベルで Device Id を取得します。
long	**wolfSSL_CTX_set_session_cache_mode**(WOLFSSL_CTX * ctx, long mode) この関数は SSL セッションキャッシュ機能を有効または無効にします。動作はモードに使用される値によって異なります。モードの値は次のとおりです：SSL_SESS_CACHE_OFF - セッションキャッシングを無効にします。デフォルトでセッションキャッシングがオンになっています。SSL_SESS_CACHE_NO_AUTO_CLEAR - セッションキャッシュのオートフラッシュを無効にします。デフォルトで自動フラッシングはオンになっています。
int	**wolfSSL_set_session_secret_cb**(WOLFSSL * ssl, SessionSecretCb cb, void * ctx) この関数はセッションシークレットコールバック関数をセットします。SessionSecretCb タイプは次のシグネチャとなっています：int（* sessioneCretcb）（wolfssl * ssl、void * secret、int * secretsz、void * ctx）。WOLFSSL 構造体の sessionSecretCb メンバーは引数 cb に設定されます。
int	**wolfSSL_save_session_cache**(const char * fname) この関数はセッションキャッシュをファイルに持続します。追加のメモリ使用のため、memsave は使用されません。
int	**wolfSSL_restore_session_cache**(const char * fname) この関数はファイルから永続セッションキャッシュを復元します。追加のメモリ使用のため、memstore は使用しません。
int	**wolfSSL_memsave_session_cache**(void * mem, int sz) この関数はセッションキャッシュをメモリに保持します。
int	**wolfSSL_memrestore_session_cache**(const void * mem, int sz) この関数はメモリから永続セッションキャッシュを復元します。
int	**wolfSSL_get_session_cache_memsize**(void) この関数は、セッションキャッシュ保存バッファをどのように大きくするかを返します。

	Name
int	**wolfSSL_CTX_save_cert_cache**(WOLFSSL_CTX * ctx, const char * fname) この関数は Cert キャッシュをメモリからファイルに書き込みます。
int	**wolfSSL_CTX_restore_cert_cache**(WOLFSSL_CTX * ctx, const char * fname) この関数はファイルから証明書キャッシュを担当します。
int	**wolfSSL_CTX_memsave_cert_cache**(WOLFSSL_CTX * ctx, void * mem, int sz, int * used) この関数は証明書キャッシュをメモリに持続します。
int	**wolfSSL_CTX_memrestore_cert_cache**(WOLFSSL_CTX * ctx, const void * mem, int sz) この関数は証明書キャッシュをメモリから復元します。
int	**wolfSSL_CTX_get_cert_cache_memsize**(WOLFSSL_CTX * ctx)Certificate Cache Save バッファが必要なサイズを返します。
int	**wolfSSL_CTX_set_cipher_listが呼び出される都度、特定の SSL コンテキストの暗号スイートリストを提供されたリストにリセットします。暗号スイートリストはヌル終端されたコロン区切りリストです。たとえば、リストの値が「DHE-RSA-AES256-SHA256：DHE-RSA-AES128-SHA256：AES256-SHA256」有効な暗号値は、src/internal.c の cipher_names [] 配列のフルネーム値です。（有効な暗号化値の明確なリストの場合は src/internal.c をチェックしてください）
int	**wolfSSL_set_cipher_listが呼び出される都度、特定の SSL コンテキストの暗号スイートリストを提供されたリストにリセットします。暗号スイートリストはヌル終端されたコロン区切りリストです。たとえば、リストの値が「DHE-RSA-AES256-SHA256：DHE-RSA-AES128-SHA256：AES256-SHA256」有効な暗号値は、src/internal.c の cipher_names [] 配列のフルネーム値です。（有効な暗号化値の明確なリストの場合は src/internal.c をチェックしてください）
void	**wolfSSL_dtls_set_using_nonblock**(WOLFSSL * ssl, int nonblock) この関数は WOLFSSL DTLS オブジェクトに下層の UDP I/O はノンブロッキングであることを通知します。アプリケーションが WOLFSSL オブジェクトを作成した後、ノンブロッキング UDP ソケットを使用する場合は、wolfSSL_dtls_set_using_nonblock() を呼び出します。これにより、WOLFSSL オブジェクトは、recvfrom 呼び出しがタイムアウトせずに EWOULDBLOCK を受信することを意味します。

	Name
int	**wolfSSL_dtls_get_using_nonblock**(WOLFSSL * ssl) この関数は WOLFSSL DTLS オブジェクトが下層に UDP ノンブロッキング I/O を使用しているか否かを取得します。WOLFSSL オブジェクトがノンブロッキング I/O を使用している場合、この関数は 1 を返します。これにより、WOLFSSL オブジェクトは、EWOULDBLOCK を受信することを意味します。この機能は DTLS セッションにとってのみ意味があります。
int	**wolfSSL_dtls_get_current_timeout**(WOLFSSL * ssl) この関数は現在のタイムアウト値を秒単位で返します。ノンブロッキングソケットを使用する場合、ユーザーコードでは、利用可能な recvV データの到着をチェックするタイミングや待つべき時間を知る必要があります。この関数によって返される値は、アプリケーションがどのくらい待機するかを示します。
int	**wolfSSL_dtls13_use_quick_timeout**(WOLFSSL * ssl) この関数はアプリケーションがより早いタイムアウト時間を設定する必要がある場合に true を返します。ノンブロッキングソケットを使用する場合でユーザーコードで受信データが到着しているか何時チェックするか、あるいはどのくらいの時間待てばよいのかを決める必要があります。この関数が true を返した場合、ライブラリはすでに通信の中断を検出しましたが、他のピアからのメッセージがまだ送信中の場合に備えて、もう少し待機する必要があることを意味します。このタイマーの値を微調整するのはアプリケーション次第ですが、dtls_get_current_timeout()/4 が最適です。
void	**wolfSSL_dtls13_set_send_more_acks**(WOLFSSL * ssl, int value) この関数は、ライブラリが中断を検出したときにすぐに他のピアに ACK を送信するかどうかを設定します。ACK をすぐに送信すると、遅延は最小限に抑えられますが、必要以上に多くの帯域幅が消費される可能性があります。アプリケーションが独自にタイマーを管理しており、このオプションが 0 に設定されている場合、アプリケーションコードは wolfSSL_dtls13_use_quick_timeout() を使用して、遅延した ACK を送信するためにより速いタイムアウトを設定する必要があるかどうかを判断できます。
int	**wolfSSL_dtls_set_timeout_init**(WOLFSSL * ssl, int) この関数は DTLS タイムアウトを設定します。
int	**wolfSSL_dtls_set_timeout_max**(WOLFSSL * ssl, int)

	Name
int	**wolfSSL_dtls_got_timeout**(WOLFSSL * ssl)DTLS でノンブロッキングソケットを使用する場合、この関数は送信がタイムアウトしたと考えられる場合に呼び出される必要があります。タイムアウト値の調整など、最後の送信を再試行するために必要なアクションを実行します。時間がかかりすぎると、失敗が返されます。
int	**wolfSSL_dtls_retransmit**(WOLFSSL * ssl)DTLS でノンブロッキングソケットを使用する場合、この関数は予想されるタイムアウト値と再送信回数を無視して最後のハンドシェイクフライトを再送信します。これは、DTLS を使用しており、タイムアウトや再試行回数も管理する必要があるアプリケーションに役立ちます。
int	**wolfSSL_dtls**(WOLFSSL * ssl)DTLS を使用するように構成されているかどうかを取得します。
int	**wolfSSL_dtls_set_peer**(WOLFSSL * ssl, void * peer, unsigned int peerSz) この関数は引数 peer で与えられるアドレスを DTLS のピアとしてセットします。
int	**wolfSSL_dtls_get_peer**(WOLFSSL * ssl, void * peer, unsigned int * peerSz) この関数は、現在の DTLS ピアの sockaddr_in(サイズ peerSz) を取得します。この関数は、peerSz を SSL セッションに保存されている実際の DTLS ピアサイズと比較します。ピアアドレスが peer に収まる場合は、peerSz がピアのサイズに設定されて、ピアの sockaddr_in が peer にコピーされます。
char *	**wolfSSL_ERR_error_string**(unsigned long errNumber, char * data) この関数は、wolfSSL_get_error() によって返されたエラーコードをより人間が読めるエラー文字列に変換します。引数 errNumber は、wolfSSL_get_error() によって返され、引数 data はエラー文字列が配置されるバッファへのポインタです。MAX_ERROR_SZ で定義されているように、データの最大長はデフォルトで 80 文字です。これは wolfssl/wolfcrypt/error.h で定義されています。
void	**wolfSSL_ERR_error_string_n**(unsigned long e, char * buf, unsigned long sz) この関数は、wolfssl_err_error_string() のバッファのサイズを指定するバージョンです。ここで、引数 len は引数 buf に書き込まれ得る最大文字数を指定します。wolfSSL_err_error_string() と同様に、この関数は wolfSSL_get_error() から返されたエラーコードをより人間が読めるエラー文字列に変換します。人間が読める文字列は buf に置かれます。
int	**wolfSSL_get_shutdown**(const WOLFSSL * ssl) この関数は、Options 構造体の closeNotify または connReset または sentNotify メンバーのシャットダウン条件をチェックします。Options 構造体は WOLFSSL 構造体内にあります。

	Name
int	**wolfSSL_session_reused**(WOLFSSL * ssl) この関数は、オプション構造体の再開メンバを返します。フラグはセッションを再利用するかどうかを示します。そうでなければ、新しいセッションを確立する必要があります。
int	**wolfSSL_is_init_finished**(WOLFSSL * ssl) この関数は、接続が確立されているかどうかを確認します。
const char *	**wolfSSL_get_version**(WOLFSSL * ssl) 文字列として使用されている SSL バージョンを返します。
int	**wolfSSL_get_current_cipher_suite**(WOLFSSL * ssl)SSL セッションで現在の暗号スイートを返します。
WOLFSSL_CIPHER *	**wolfSSL_get_current_cipher**(WOLFSSL * ssl) この関数は、SSL セッションの現在の暗号へのポインタを返します。
const char *	**wolfSSL_CIPHER_get_name**(const WOLFSSL_CIPHER * cipher) この関数は、SSL オブジェクト内の Cipher Suite と使用可能なスイートと一致し、文字列表現を返します。
const char *	**wolfSSL_get_cipher**(WOLFSSL * ssl) この関数は、SSL オブジェクト内の暗号スイートと使用可能なスイートと一致します。
WOLFSSL_SESSION *	**wolfSSL_get1_session**(WOLFSSL * ssl) この関数は、WOLFSSL 構造体から WOLFSSL_SESSION を参照型として返します。これには、wolfSSL_SESSION_free を呼び出してセッション参照を解除する必要があります。WOLFSSL_SESSION は、セッションの再開を実行するために必要なすべての必要な情報を含み、新しいハンドシェイクなしで接続を再確立します。セッションの再開の場合、wolfSSL_shutdown() をセッションオブジェクトに呼び出す前に、アプリケーションはオブジェクトから wolfssl_get1_session() を呼び出して保存する必要があります。これはセッションへのポインタを返します。その後、アプリケーションは新しい WOLFSSL オブジェクトを作成し、保存したセッションを wolfssl_set_session() に割り当てる必要があります。この時点で、アプリケーションは wolfssl_connect() を呼び出し、WolfSSL はセッションを再開しようとします。WolfSSL サーバーコードでは、デフォルトでセッションの再開を許可します。wolfssl_get1_session() によって返されたオブジェクトは、アプリケーションが使用後は解放される必要があります。

	Name
WOLFSSL_METHOD *	**wolfSSLv23_client_method**(void)wolfsslv23_client_method() 関数は、アプリケーションがクライアントであることを示すために使用され、SSL 3.0~TLS 1.3 の間でサーバーでサポートされている最高のプロトコルバージョンをサポートします。この関数は、wolfSSL_CTX_new() を使用して SSL / TLS コンテキストを作成するときに使用される新しい Wolfssl_method 構造体のメモリを割り当てて初期化します。WolfSSL クライアントとサーバーの両方が堅牢なバージョンのダウングレード機能を持っています。特定のプロトコルバージョンメソッドがどちらの側で使用されている場合は、そのバージョンのみがネゴシエートされたり、エラーが返されます。たとえば、TLSV1 を使用し、SSLv3 のみに接続しようとするクライアントは、TLSV1.1 に接続しても失敗します。この問題を解決するために、wolfsslv23_client_method() 関数を使用するクライアントは、サーバーでサポートされている最高のプロトコルバージョンを使用し、必要に応じて SSLv3 にダウングレードします。この場合、クライアントは SSLv3 - TLSv1.3 を実行しているサーバーに接続できるようになります。
int	**wolfSSL_BIO_get_mem_data**(WOLFSSL_BIO * bio, void * p) この関数は、内部メモリバッファの先頭へのバイトポインタを設定するために使用されます。
long	**wolfSSL_BIO_set_fd**(WOLFSSL_BIO * b, int fd, int flag) 使用する BIO のファイル記述子を設定します。
int	**wolfSSL_BIO_set_close**(WOLFSSL_BIO * b, long flag)BIO が解放されたときに I/O ストリームを閉じる必要があることを示すために使用されるクローズフラグを設定します。
WOLFSSL_BIO_METHOD *	**wolfSSL_BIO_s_socket**(void) この関数は BIO_SOCKET タイプの WOLFSSL_BIO_METHOD を取得するために使用されます。
int	**wolfSSL_BIO_set_write_buf_size**(WOLFSSL_BIO * b, long size) この関数は、WOLFSSL_BIO のライトバッファのサイズを設定するために使用されます。書き込みバッファが以前に設定されている場合、この関数はサイズをリセットするときに解放されます。読み書きインデックスを 0 にリセットするという点で、wolfSSL_BIO_reset に似ています。

	Name
int	**wolfSSL_BIO_make_bio_pair**(WOLFSSL_BIO * b1, WOLFSSL_BIO * b2) これは 2 つの BIOS を一緒にペアリングするために使用されます。一対の BIOS は、2 つの方法パイプと同様に、他方で読み取られることができ、その逆も同様である。BIOS の両方が同じスレッド内にあることが予想されます。この機能はスレッドセーフではありません。2 つの BIOS のうちの 1 つを解放すると、両方ともペアになっています。書き込みバッファサイズが以前に設定されていない場合、それはペアになる前に 17000（wolfssl_bio_size）のデフォルトサイズに設定されます。
int	**wolfSSL_BIO_ctrl_reset_read_request**(WOLFSSL_BIO * bio) この関数は、読み取り要求フラグを 0 に戻すために使用されます。
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) 関数によって返される数のバイトを書き込むためにバッファーへのポインタを取得します。返されるポインタに追加のバイトを書き込んだ場合、返された値は範囲外の書き込みにつながる可能性があります。
int	**wolfSSL_BIO_reset**(WOLFSSL_BIO * bio) バイオを初期状態にリセットします。タイプ BIO_BIO の例として、これは読み書きインデックスをリセットします。
int	**wolfSSL_BIO_seek**(WOLFSSL_BIO * bio, int ofs) この関数は、指定されたオフセットへファイルポインタを調整します。これはファイルの先頭からのオフセットです。
int	**wolfSSL_BIO_write_filename**(WOLFSSL_BIO * bio, char * name) これはファイルに設定および書き込むために使用されます。現在ファイル内のデータを上書きし、BIO が解放されたときにファイルを閉じるように設定されます。
long	**wolfSSL_BIO_set_mem_eof_return**(WOLFSSL_BIO * bio, int v) これはファイル値の終わりを設定するために使用されます。一般的な値は予想される正の値と混同されないように-1 です。
long	**wolfSSL_BIO_get_mem_ptr**(WOLFSSL_BIO * bio, WOLFSSL_BUF_MEM ** m) これは WolfSSL_BIO メモリポインタのゲッター関数です。
char *	**wolfSSL_X509_NAME_oneline**(WOLFSSL_X509_NAME * name, char * in, int sz) この関数は X509 の名前をバッファにコピーします。
WOLFSSL_X509_NAME *	**wolfSSL_X509_get_issuer_name**(WOLFSSL_X509 * cert) この関数は証明書発行者の名前を返します。

	Name
WOLFSSL_X509_NAME *	**wolfSSL_X509_get_subject_name**(WOLFSSL_X509 * cert) この関数は、wolfssl_x509 構造の件名メンバーを返します。
int	**wolfSSL_X509_get_isCA**(WOLFSSL_X509 * cert)WOLFSSL_X509 構造体の isCa メンバーをチェックして値を返します。
int	**wolfSSL_X509_NAME_get_text_by_NID**(WOLFSSL_X509_NAME * name, int nid, char * buf, int len) この関数は、渡された NID 値に関連するテキストを取得します。
int	**wolfSSL_X509_get_signature_type**(WOLFSSL_X509 * cert) この関数は、WOLFSSL_X509 構造体の sigOID メンバーに格納されている値を返します。
void	**wolfSSL_X509_free**(WOLFSSL_X509 * x509) この関数は WOLFSSL_X509 構造体を解放します。
int	**wolfSSL_X509_get_signature**(WOLFSSL_X509 * x509, unsigned char * buf, int * bufSz)x509 署名を取得し、それをバッファに保存します。
int	**wolfSSL_X509_STORE_add_cert**(WOLFSSL_X509_STORE * store, WOLFSSL_X509 * x509) この関数は、WOLFSSL_X509_STRE 構造体に証明書を追加します。
WOLFSSL_STACK *	**wolfSSL_X509_STORE_CTX_get_chain**(WOLFSSL_X509_STORE_CT * ctx) この関数は、WOLFSSL_X509_STORE_CTX 構造体のチェーン変数の getter 関数です。現在チェーンは取り込まれていません。
int	**wolfSSL_X509_STORE_set_flags**(WOLFSSL_X509_STORE * store, unsigned long flag) この関数は、渡された WOLFSSL_X509_STORE 構造体の動作を変更するためのフラグを取ります。使用されるフラグの例は WOLFSSL_CRL_CHECK です。
const byte *	**wolfSSL_X509_notBefore**(WOLFSSL_X509 * x509) この関数は BYTE アレイとして符号化された “not before” 要素を返します。
const byte *	**wolfSSL_X509_notAfter**(WOLFSSL_X509 * x509) この関数は、BYTE 配列として符号化された “not after” 要素を返します。
WOLFSSL_BIGNUM *	**wolfSSL_ASN1_INTEGER_to_BN**(const WOLFSSL_ASN1_INTEGER * ai, WOLFSSL_BIGNUM * bn) この関数は、WOLFSSL_ASN1_INTEGER 値を WOLFSSL_BIGNUM 構造体にコピーするために使用されます。
long	**wolfSSL_CTX_add_extra_chain_cert**(WOLFSSL_CTX * ctx, WOLFSSL_X509 * x509) この関数は、WOLFSSL_CTX 構造で構築されている内部チェーンに証明書を追加します。
int	**wolfSSL_CTX_get_read_ahead**(WOLFSSL_CTX * ctx) この関数は、WOLFSSL_CTX 構造から Get Read Hape フラグを返します。

	Name
int	**wolfSSL_CTX_set_read_ahead**(WOLFSSL_CTX * ctx, int v) この関数は、WOLFSSL_CTX 構造内の読み出し先のフラグを設定します。
long	**wolfSSL_CTX_set_tlsext_status_arg**(WOLFSSL_CTX * ctx, void * arg) この関数は OCSP で使用するオプション引数を設定します。
long	**wolfSSL_CTX_set_tlsext_opaque_prf_input_callback_arg**(WOLF * ctx, void * arg) この関数は、PRF コールバックに渡すオプションの引数を設定します。
long	**wolfSSL_set_options**(WOLFSSL * s, long op) この関数は、SSL のオプションマスクを設定します。いくつかの有効なオプションは、ssl_op_all、ssl_op_cookie_exchange、ssl_op_no_sslv2、ssl_op_no_sslv3、ssl_op_no_tlsv1_1、ssl_op_no_tlsv1_2、ssl_op_no_compression です。
long	**wolfSSL_get_options**(const WOLFSSL * ssl) この関数は現在のオプションマスクを返します。
long	**wolfSSL_set_tlsext_debug_arg**(WOLFSSL * ssl, void * arg) この関数は、渡されたデバッグ引数を設定するために使用されます。
long	**wolfSSL_set_tlsext_status_type**(WOLFSSL * s, int type) この関数は、サーバが OCSP ステータス応答（OCSP ステイプルとも呼ばれる）を送受信するクライアントアプリケーションが要求されたときに呼び出されます。
long	**wolfSSL_get_verify_result**(const WOLFSSL * ssl)
void	**wolfSSL_ERR_print_errors_fp**(XFILE fp, int err) この関数は、wolfSSL_get_error() によって返されたエラーコードをより多くの人間が読めるエラー文字列に変換し、その文字列を出力ファイルに印刷します。ERR は、WOLFSSL_GET_ERROR() によって返され、FP がエラー文字列が配置されるファイルであるエラーコードです。
void	**wolfSSL_ERR_print_errors_cb**(int*)(const char* str, size_t len, void *u) cb, void* u) この関数は提供されたコールバックを使用してエラー報告を処理します。コールバック関数はエラー回線ごとに実行されます。文字列、長さ、および userdata はコールバックパラメータに渡されます。
void	**wolfSSL_CTX_set_psk_client_callback**(WOLFSSL_CTX * ctx, wc_psk_client_callback cb) この関数は WOLFSSL_CTX 構造の client_psk_cb メンバーをセットします。
void	**wolfSSL_set_psk_client_callback**(WOLFSSL * ssl, wc_psk_client_callback)
const char *	**wolfSSL_get_psk_identity_hint**(const WOLFSSL *) この関数は PSK アイデンティティヒントを返します。

	Name
const char *	**wolfSSL_get_psk_identity**(const WOLFSSL *) 関数は、配列構造の Client_Identity メンバーへの定数ポインタを返します。
int	**wolfSSL_CTX_use_psk_identity_hint**(WOLFSSL_CTX * ctx, const char * hint) この関数は、WOLFSSL_CTX 構造体の server_hint メンバーに HINT 引数を格納します。
int	**wolfSSL_use_psk_identity_hint**(WOLFSSL * ssl, const char * hint) この関数は、wolfssl 構造内の配列構造の server_hint メンバーに HINT 引数を格納します。
void	**wolfSSL_CTX_set_psk_server_callback**(WOLFSSL_CTX * ctx, wc_psk_server_callback cb)WOLFSSL_CTX 構造体
void	**wolfSSL_set_psk_server_callback**(WOLFSSL * ssl, wc_psk_server_callback cb)WolfSSL 構造オプションメンバー。
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 *) この機能により、CTX 構造の HAVAnon メンバーがコンパイル中に定義されている場合は、CTX 構造の HABANON メンバーを有効にします。
WOLFSSL_METHOD *	**wolfSSLv23_server_method**(void)wolfsslv23_server_method() 関数は、アプリケーションがサーバーであることを示すために使用され、SSL 3.0 _ TLS 1.3 からプロトコルバージョンと接続するクライアントをサポートします。この関数は、wolfSSL_CTX_new() を使用して SSL / TLS コンテキストを作成するときに使用される新しい Wolfssl_method 構造体のメモリを割り当てて初期化します。
int	**wolfSSL_state**(WOLFSSL * ssl)
WOLFSSL_X509 *	**wolfSSL_get_peer_certificate**(WOLFSSL * ssl) この関数はピアの証明書を取得します。
int	**wolfSSL_want_read**(WOLFSSL *) この関数は、wolfSSL_get_error() を呼び出して ssl_error_want_read を取得するのと似ています。基礎となるエラー状態が SSL_ERROR_WANT_READ の場合、この関数は 1 を返しますが、それ以外の場合は 0 です。
int	**wolfSSL_want_write**(WOLFSSL *) この関数は、wolfSSL_get_error() を呼び出し、RETURS の SSL_ERROR_WANT_WRITE を取得するのと同じです。基礎となるエラー状態が SSL_ERROR_WANT_WRITE の場合、この関数は 1 を返しますが、それ以外の場合は 0 です。

	Name
int	**wolfSSL_check_domain_name**(WOLFSSL * ssl, const char * dn)wolfssl デフォルトでは、有効な日付範囲と検証済みの署名のためにピア証明書をチェックします。wolfssl_connect() または wolfssl_accept() の前にこの関数を呼び出すと、実行するチェックのリストにドメイン名チェックが追加されます。DN 受信時にピア証明書を確認するためのドメイン名を保持します。
int	**wolfSSL_Init**(void) 使用するために WolfSSL ライブラリを初期化します。アプリケーションごとに 1 回、その他のライブラリへの呼び出しの前に呼び出す必要があります。
int	**wolfSSL_Cleanup**(void) さらなる使用から WOLFSSL ライブラリを初期化します。ライブラリによって使用されるリソースを解放しますが、呼び出される必要はありません。
const char *	**wolfSSL_lib_version**(void) この関数は現在のライブラリーバージョンを返します。
word32	**wolfSSL_lib_version_hex**(void) この関数は、現在のライブラリーのバージョンを 16 進表記で返します。
int	**wolfSSL_negotiate**(WOLFSSL * ssl)SSL メソッドの側面に基づいて、実際の接続または承認を実行します。クライアント側から呼び出された場合、サーバ側から呼び出された場合に wolfssl_accept() が実行されている間に wolfssl_connect() が行われる。
int	**wolfSSL_set_compression**(WOLFSSL * ssl)SSL 接続に圧縮を使用する機能をオンにします。両側には圧縮がオンになっている必要があります。そうでなければ圧縮は使用されません。ZLIB ライブラリは実際のデータ圧縮を実行します。ライブラリにコンパイルするには、システムの設定システムに–with-libz を使用し、そうでない場合は hand_libz を定義します。送受信されるメッセージの実際のサイズを減らす前にデータを圧縮している間に、圧縮によって保存されたデータの量は通常、ネットワークの遅いすべてのネットワークを除いたものよりも分析に時間がかかります。
int	**wolfSSL_set_timeout**(WOLFSSL * ssl, unsigned int to) この関数は SSL セッションタイムアウト値を秒単位で設定します。
int	**wolfSSL_CTX_set_timeout**(WOLFSSL_CTX * ctx, unsigned int to) この関数は、指定された SSL コンテキストに対して、SSL セッションのタイムアウト値を秒単位で設定します。
WOLFSSL_X509_CHAIN *	**wolfSSL_get_peer_chain**(WOLFSSL * ssl) ピアの証明書チェーンを取得します。
int	**wolfSSL_get_chain_count**(WOLFSSL_X509_CHAIN * chain) ピアの証明書チェーン数を取得します。

	Name
int	**wolfSSL_get_chain_length**(WOLFSSL_X509_CHAIN * chain, int idx)Index（IDX）のピアの ASN1.DER 証明書長をバイト単位で取得します。
unsigned char *	**wolfSSL_get_chain_cert**(WOLFSSL_X509_CHAIN * chain, int idx) インデックス（IDX）でピアの ASN1.DER 証明書を取得します。
WOLFSSL_X509 *	**wolfSSL_get_chain_X509**(WOLFSSL_X509_CHAIN * chain, int idx) この関数は、証明書のチェーンからのピアの WOLFSSL_X509 構造体をインデックス（IDX）で取得します。
int	**wolfSSL_get_chain_cert_pem**(WOLFSSL_X509_CHAIN * chain, int idx, unsigned char * buf, int inLen, int * outLen) インデックス（IDX）でピアの PEM 証明書を取得します。
const unsigned char *	**wolfSSL_get_sessionID**(const WOLFSSL_SESSION * s) セッションの ID を取得します。セッション ID は常に 32 バイトの長さです。
int	**wolfSSL_X509_get_serial_number**(WOLFSSL_X509 * x509, unsigned char * in, int * inOutSz) ピアの証明書のシリアル番号を取得します。シリアル番号バッファ（IN）は少なくとも 32 バイト以上であり、入力として * INOUTSZ 引数として提供されます。関数を呼び出した後 * INOUTSZ は IN バッファに書き込まれた実際の長さをバイト単位で保持します。
char *	**wolfSSL_X509_get_subjectCN**(WOLFSSL_X509 *) 証明書から件名の共通名を返します。
const unsigned char *	**wolfSSL_X509_get_der**(WOLFSSL_X509 * x509, int * outSz) この関数は、wolfssl_x509 構造体の DER エンコードされた証明書を取得します。
WOLFSSL_ASN1_TIME *	**wolfSSL_X509_get_notAfter**(WOLFSSL_X509 *) この関数は、x509 が null のかどうかを確認し、そうでない場合は、x509 構造体のノッカスメンバーを返します。
int	**wolfSSL_X509_version**(WOLFSSL_X509 *) この関数は X509 証明書のバージョンを取得します。
WOLFSSL_X509 *	**wolfSSL_X509_d2i_fp**(WOLFSSL_X509 ** x509, FILE * file)no_stdio_filesystem が定義されている場合、この関数はヒープメモリを割り当て、wolfssl_x509 構造を初期化してそれにポインタを返します。
WOLFSSL_X509 *	**wolfSSL_X509_load_certificate_file**(const char * fname, int format) 関数は X509 証明書をメモリにロードします。
unsigned char *	**wolfSSL_X509_get_device_type**(WOLFSSL_X509 * x509, unsigned char * in, int * inOutSz) この関数は、デバイスの種類を X509 構造からバッファにコピーします。

	Name
unsigned char *	**wolfSSL_X509_get_hw_type**(WOLFSSL_X509 * x509, unsigned char * in, int * inOutSz) この関数は、wolfssl_x509 構造の HWType メンバーをバッファにコピーします。
unsigned char *	**wolfSSL_X509_get_hw_serial_number**(WOLFSSL_X509 * x509, unsigned char * in, int * inOutSz) この関数は X509 オブジェクトの hwserialNum メンバを返します。
int	**wolfSSL_connect_cert**(WOLFSSL * ssl) この関数はクライアント側で呼び出され、ピアの証明書チェーンを取得するのに十分な長さだけサーバーを持つ SSL / TLS ハンドシェイクを開始します。この関数が呼び出されると、基礎となる通信チャネルはすでに設定されています。 wolfssl_connect_cert() は、ブロックと非ブロック I / O の両方で動作します。基礎となる I / O がノンブロッキングである場合、 wolfsl_connect_cert() は、 wolfssl_connect_cert_cert() のニーズを満たすことができなかったときに戻ります。ハンドシェイクを続けます。この場合、wolfSSL_get_error() への呼び出しは SSL_ERROR_WANT_READ または SSL_ERROR_WANT_WRITE のいずれかを生成します。通話プロセスは、基礎となる I / O が準備ができて、wolfssl がオフになっているところを拾うときに、wolfssl_connect_cert() への呼び出しを繰り返す必要があります。ノンブロッキングソケットを使用する場合は、何も実行する必要がありますが、select() を使用して必要な条件を確認できます。基礎となる入出力がブロックされている場合、wolfssl_connect_cert() はピアの証明書チェーンが受信されたらのみ返されます。
WC_PKCS12 *	**wolfSSL_d2i_PKCS12_bio**(WOLFSSL_BIO * bio, WC_PKCS12 ** pkcs12)WOLFSSL_D2I_PKCS12_BIO（D2I_PKCS12_BIO）は、WOLFSSL_BIO から構造 WC_PKCS12 への PKCS12 情報にコピーされます。この情報は、オプションの MAC 情報を保持するための構造とともにコンテンツに関する情報のリストとして構造内に分割されています。構造体 WC_PKCS12 で情報がチャンク（ただし復号化されていない）に分割された後、それはその後、呼び出しによって解析および復号化され得る。
WC_PKCS12 *	**wolfSSL_i2d_PKCS12_bio**(WOLFSSL_BIO * bio, WC_PKCS12 * pkcs12)WOLFSSL_I2D_PKCS12_BIO（I2D_PKCS12_BIO）は、構造 WC_PKCS12 から WOLFSSL_BIO への証明書情報にコピーされます。

	Name
int	**wolfSSL_PKCS12_parse(WOLFSSL_X509)** ca)pkcs12 は、configure コマンドへの-enable-openSSLAXTRA を追加することで有効にできます。それは復号化のためにトリプル DES と RC4 を使うことができるので、OpenSSlextra（–enable-des3 -enable-arc4）を有効にするときにもこれらの機能を有効にすることをお勧めします。wolfssl は現在 RC2 をサポートしていませんので、RC2 での復号化は現在利用できません。これは、.p12 ファイルを作成するために OpenSSL コマンドラインで使用されるデフォルトの暗号化方式では注目すかもしれません。WOLFSSL_PKCS12_PARSE（PKCS12_PARSE）。この関数が最初に行っているのは、存在する場合は Mac が正しいチェックです。MAC が失敗した場合、関数は返され、保存されているコンテンツ情報のいずれかを復号化しようとしません。この関数は、バッグタイプを探している各コンテンツ情報を介して解析します。バッグタイプがわかっている場合は、必要に応じて復号化され、構築されている証明書のリストに格納されているか、見つかったキーとして保存されます。すべてのバッグを介して解析した後、見つかったキーは、一致するペアが見つかるまで証明書リストと比較されます。この一致するペアはキーと証明書として返され、オプションで見つかった証明書リストは stack_of 証明書として返されます。瞬間、CRL、秘密または安全なバッグがスキップされ、解析されません。デバッグプリントアウトを見ることで、これらまたは他の「不明」バッグがスキップされているかどうかがわかります。フレンドリー名などの追加の属性は、PKCS12 ファイルを解析するときにスキップされます。
int	**wolfSSL_SetTmpDH**(WOLFSSL * ssl, const unsigned char * p, int pSz, const unsigned char * g, int gSz) サーバー DIFFIE-HELLMAN エフェメラルパラメータ設定。この関数は、サーバーが DHE を使用する暗号スイートをネゴシエートしている場合に使用するグループパラメータを設定します。
int	**wolfSSL_SetTmpDH_buffer**(WOLFSSL * ssl, const unsigned char * b, long sz, int format) 関数は wolfssl_settmph_buffer_wrapper を呼び出します。これは Diffie-Hellman パラメータのラッパーです。
int	**wolfSSL_SetTmpDH_file**(WOLFSSL * ssl, const char * f, int format) この関数は、wolfssl_settmph_file_wrapper を呼び出してサーバ diffie-hellman パラメータを設定します。

	Name
int	**wolfSSL_CTX_SetTmpDH**(WOLFSSL_CTX * ctx, const unsigned char * p, int pSz, const unsigned char * g, int gSz) サーバー CTX Diffie-Hellman のパラメータを設定します。
int	**wolfSSL_CTX_SetTmpDH_buffer**(WOLFSSL_CTX * ctx, const unsigned char * b, long sz, int format)wolfssl_settmph_buffer_wrapper を呼び出すラッパー関数
int	**wolfSSL_CTX_SetTmpDH_file**(WOLFSSL_CTX * ctx, const char * f, int format) この関数は、wolfssl_settmph_file_wrapper を呼び出してサーバー Diffie-Hellman パラメータを設定します。
int	**wolfSSL_CTX_SetMinDhKey_Sz**(WOLFSSL_CTX * ctx, word16) この関数は、WOLFSSL_CTX 構造体の minkkeysz メンバーにアクセスして、Diffie Hellman 鍵サイズの最小サイズ（ビット単位）を設定します。
int	**wolfSSL_SetMinDhKey_Sz**(WOLFSSL * ssl, word16 keySz_bits)WolfSSL 構造の Diffie-Hellman 鍵の最小サイズ（ビット単位）を設定します。
int	**wolfSSL_CTX_SetMaxDhKey_Sz**(WOLFSSL_CTX * ctx, word16 keySz_bits) この関数は、WOLFSSL_CTX 構造体の maxdhkeysz メンバーにアクセスして、Diffie Hellman 鍵サイズの最大サイズ（ビット単位）を設定します。
int	**wolfSSL_SetMaxDhKey_Sz**(WOLFSSL * ssl, word16 keySz_bits)WolfSSL 構造の Diffie-Hellman 鍵の最大サイズ（ビット単位）を設定します。
int	**wolfSSL_GetDhKey_Sz**(WOLFSSL *) オプション構造のメンバーである DHKEYSZ（ビット内）の値を返します。この値は、Diffie-Hellman 鍵サイズをバイト単位で表します。
int	**wolfSSL_CTX_SetMinRsaKey_Sz**(WOLFSSL_CTX * ctx, short keySz)WOLFSSL_CTX 構造体と wolfssl_cert_manager 構造の両方で最小 RSA 鍵サイズを設定します。
int	**wolfSSL_SetMinRsaKey_Sz**(WOLFSSL * ssl, short keySz)WolfSSL 構造にある RSA のためのビットで最小許容鍵サイズを設定します。
int	**wolfSSL_CTX_SetMinEccKey_Sz**(WOLFSSL_CTX * ssl, short keySz)wolf_ctx 構造体と wolfssl_cert_manager 構造体の ECC 鍵の最小サイズをビット単位で設定します。
int	**wolfSSL_SetMinEccKey_Sz**(WOLFSSL * ssl, short keySz) オプション構造の MineCckeysz メンバーの値を設定します。オプション構造体は、WolfSSL 構造のメンバーであり、SSL パラメータを介してアクセスされます。

	Name
int	**wolfSSL_make_eap_keys**(WOLFSSL * ssl, void * key, unsigned int len, const char * label) この関数は、eap_tls と eap-ttls によって、マスターシークレットからキーイングマテリアルを導出します。
int	**wolfSSL_writev**(WOLFSSL * ssl, const struct iovec * iov, int iovcnt)Writev Semantics をシミュレートしますが、SSL_Write() の動作のために実際にはブロックしないため、フロント追加が小さくなる可能性があるため Writev を使いやすいソフトウェアに移植する。
int	**wolfSSL_CTX_UnloadCAs**(WOLFSSL_CTX *) この関数は CA 署名者リストをアンロードし、署名者全体のテーブルを解放します。
int	**wolfSSL_CTX_Unload_trust_peers**(WOLFSSL_CTX *) この関数は、以前にロードされたすべての信頼できるピア証明書をアンロードするために使用されます。マクロ wolfssl_trust_peer_cert を定義することで機能が有効になっています。
int	**wolfSSL_CTX_trust_peer_buffer**(WOLFSSL_CTX * ctx, const unsigned char * in, long sz, int format) この関数は、TLS / SSL ハンドシェイクを実行するときにピアを検証するために使用する証明書をロードします。ハンドシェイク中に送信されたピア証明書は、使用可能なときにスキッドを使用することによって比較されます。これら 2 つのことが一致しない場合は、ロードされた CAS が使用されます。ファイルの代わりにバッファーの場合は、wolfssl_ctx_trust_peer_cert と同じ機能です。特徴はマクロ wolfssl_trust_peer_cert を定義することによって有効になっています適切な使用法の例を参照してください。
int	**wolfSSL_CTX_load_verify_buffer**(WOLFSSL_CTX * ctx, const unsigned char * in, long sz, int format) この関数は CA 証明書バッファを WolfSSL コンテキストにロードします。バッファ以外のバージョンのように動作し、ファイルの代わりに入力としてバッファと呼ばれる機能が異なるだけです。バッファはサイズ SZ の引数によって提供されます。形式バッファのフォーマットタイプを指定します。SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM。フォーマットが PEM 内にある限り、バッファあたり複数の CA 証明書をロードすることができます。適切な使用法の例をご覧ください。

	Name
int	**wolfSSL_CTX_load_verify_buffer_ex**(WOLFSSL_CTX * ctx, const unsigned char * in, long sz, int format, int userChain, word32 flags) この関数は CA 証明書バッファを WolfSSL コンテキストにロードします。バッファ以外のバージョンのように動作し、ファイルの代わりに入力としてバッファと呼ばれる機能が異なるだけです。バッファはサイズ SZ の引数によって提供されます。形式バッファのフォーマットタイプを指定します。SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM。フォーマットが PEM 内にある限り、バッファあたり複数の CA 証明書をロードすることができます。_EX バージョンは PR 2413 に追加され、UserChain と Flags の追加の引数をサポートします。
int	**wolfSSL_CTX_load_verify_chain_buffer_format**(WOLFSSL_CTX * ctx, const unsigned char * in, long sz, int format) この関数は、CA 証明書チェーンバッファを WolfSSL コンテキストにロードします。バッファ以外のバージョンのように動作し、ファイルの代わりに入力としてバッファと呼ばれる機能が異なるだけです。バッファはサイズ SZ の引数によって提供されます。形式バッファのフォーマットタイプを指定します。SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM。フォーマットが PEM 内にある限り、バッファあたり複数の CA 証明書をロードすることができます。適切な使用法の例をご覧ください。
int	**wolfSSL_CTX_use_certificate_buffer**(WOLFSSL_CTX * ctx, const unsigned char * in, long sz, int format) この関数は証明書バッファを WolfSSL コンテキストにロードします。バッファ以外のバージョンのように動作し、ファイルの代わりに入力としてバッファと呼ばれる機能が異なるだけです。バッファはサイズ SZ の引数によって提供されます。形式バッファのフォーマットタイプを指定します。SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM。適切な使用法の例をご覧ください。
int	**wolfSSL_CTX_use_PrivateKey_buffer**(WOLFSSL_CTX * ctx, const unsigned char * in, long sz, int format) この関数は、秘密鍵バッファを SSL コンテキストにロードします。バッファ以外のバージョンのように動作し、ファイルの代わりに入力としてバッファと呼ばれる機能が異なるだけです。バッファはサイズ SZ の引数によって提供されます。形式バッファのフォーマットタイプを指定します。SSL_FILETYPE_ASN1OR SSL_FILETYPE_PEM。適切な使用法の例をご覧ください。

	Name
int	**wolfSSL_CTX_use_certificate_chain_buffer**(WOLFSSL_CTX * ctx, const unsigned char * in, long sz) この関数は、証明書チェーンバッファを WolfSSL コンテキストにロードします。バッファ以外のバージョンのように動作し、ファイルの代わりに入力としてバッファと呼ばれる機能が異なるだけです。バッファはサイズ SZ の引数によって提供されます。バッファは PEM 形式で、ルート証明書で終わる対象の証明書から始めてください。適切な使用法の例をご覧ください。
int	**wolfSSL_use_certificate_buffer**(WOLFSSL * ssl, const unsigned char * in, long sz, int format) この関数は、証明書バッファを WolfSSL オブジェクトにロードします。バッファ以外のバージョンのように動作し、ファイルの代わりに入力としてバッファと呼ばれる機能が異なるだけです。バッファはサイズ SZ の引数によって提供されます。形式バッファのフォーマットタイプを指定します。SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM。適切な使用法の例をご覧ください。
int	**wolfSSL_use_PrivateKey_buffer**(WOLFSSL * ssl, const unsigned char * in, long sz, int format) この関数は、秘密鍵バッファを WolfSSL オブジェクトにロードします。バッファ以外のバージョンのように動作し、ファイルの代わりに入力としてバッファと呼ばれる機能が異なるだけです。バッファはサイズ SZ の引数によって提供されます。形式バッファのフォーマットタイプを指定します。SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM。適切な使用法の例をご覧ください。
int	**wolfSSL_use_certificate_chain_buffer**(WOLFSSL * ssl, const unsigned char * in, long sz) この関数は、証明書チェーンバッファを WolfSSL オブジェクトにロードします。バッファ以外のバージョンのように動作し、ファイルの代わりに入力としてバッファと呼ばれる機能が異なるだけです。バッファはサイズ SZ の引数によって提供されます。バッファは PEM 形式で、ルート証明書で終わる対象の証明書から始めてください。適切な使用法の例をご覧ください。
int	**wolfSSL_UnloadCertsKeys**(WOLFSSL *) この関数は、SSL が所有する証明書または鍵をアンロードします。
int	**wolfSSL_CTX_set_group_messages**(WOLFSSL_CTX *) この機能は、可能な限りハンドシェイクメッセージのグループ化をオンにします。
int	**wolfSSL_set_group_messages**(WOLFSSL *) この機能は、可能な限りハンドシェイクメッセージのグループ化をオンにします。

	Name
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) この関数は、許可されている最小のダウングレードバージョンを設定します。接続が（wolfsslv23_client_method または wolfsslv23_server_method）を使用して、接続がダウングレードできる場合にのみ適用されます。
int	**wolfSSL_SetMinVersion**(WOLFSSL * ssl, int version) この関数は、許可されている最小のダウングレードバージョンを設定します。接続が（wolfsslv23_client_method または wolfsslv23_server_method）を使用して、接続がダウングレードできる場合にのみ適用されます。
int	**wolfSSL_GetObjectSize**(void) ビルドオプションと設定に依存します。WolfSSL を構築するときに show_sizes が定義されている場合、この関数は WolfSSL オブジェクト（スイート、暗号など）内の個々のオブジェクトのサイズも stdout に印刷されます。
int	**wolfSSL_GetOutputSize**(WOLFSSL * ssl, int inSz) アプリケーションがトランスポートレイヤ間で何バイトを送信したい場合は、指定された平文の入力サイズを指定してください。SSL / TLS ハンドシェイクが完了した後に呼び出す必要があります。
int	**wolfSSL_GetMaxOutputSize**(WOLFSSL *) プロトコル規格で指定されている最大 SSL / TLS レコードサイズのいずれかに対応します。この関数は、アプリケーションが wolfssl_getOutputSize() と呼ばれ、input_size_e エラーを受信したときに役立ちます。SSL / TLS ハンドシェイクが完了した後に呼び出す必要があります。
int	**wolfSSL_SetVersion**(WOLFSSL * ssl, int version) この関数は、バージョンで指定されたバージョンを使用して、指定された SSL セッション（WolfSSL オブジェクト）の SSL/TLS プロトコルバージョンを設定します。これにより、SSL セッション（SSL）のプロトコル設定が最初に定義され、SSL コンテキスト（wolfSSL_CTX_new()）メソッドの種類によって上書きされます。

	Name
void	**wolfSSL_CTX_SetMacEncryptCb**(WOLFSSL_CTX * ctx, CallbackMacEncrypti cb)MAC /暗号化コールバック。コールバックは成功の場合は 0 を返すか、エラーの場合は <0 です。SSL と CTX ポインタはユーザーの利便性に利用できます。MacOut は、MAC の結果を保存する必要がある出力バッファです。Macin は Mac 入力バッファーと Macinsz のサイズを注意しています。MacContent と Macverify は、Wolfssl_SettlShmacinner() に必要であり、そのまま通過します。Encout は、暗号化の結果を格納する必要がある出力バッファです。ENCIN は ENCSZ が入力のサイズである間は暗号化する入力バッファです。コールバックの例は、wolfssl / test.h mymacencryptcb() を見つけることができます。
void	**wolfSSL_SetMacEncryptCtx**(WOLFSSL * ssl, void * ctx)CTX へのコールバックコンテキスト。
void *	**wolfSSL_GetMacEncryptCtx**(WOLFSSL * ssl)Mac / Encrypt コールバックコンテキストは、wolfssl_setmacencryptx() で保存されていました。
void	**wolfSSL_CTX_SetDecryptVerifyCb**(WOLFSSL_CTX * ctx, CallbackDecryptVerify cb) コールバックを復号化/確認します。コールバックは成功の場合は 0 を返すか、エラーの場合は <0 です。SSL と CTX ポインタはユーザーの利便性に利用できます。DECOUT は、復号化の結果を格納する出力バッファです。DECIN は暗号化された入力バッファーと Decinsz のサイズを注意しています。コンテンツと検証は、WolfSSL_SettlShmacinner() に必要であり、そのまま通過します。PADSZ は、パディングの合計値で設定する出力変数です。つまり、MAC サイズとパディングバイトとパッドバイトを加えています。コールバックの例は、wolfssl / test.h mydecryptverifycb() を見つけることができます。
void	**wolfSSL_SetDecryptVerifyCtx**(WOLFSSL * ssl, void * ctx) コールバックコンテキストを CTX に復号化/検証します。
void *	**wolfSSL_GetDecryptVerifyCtx**(WOLFSSL * ssl)wolfssl_setdecryptverifyctx() で以前に保存されているコールバックコンテキストを復号化/検証します。
const unsigned char *	**wolfSSL_GetMacSecret**(WOLFSSL * ssl, int verify)VERIFY パラメーターは、これがピア・メッセージの検証のためのものであるかどうかを指定します。
const unsigned char *	**wolfSSL_GetClientWriteKey**(WOLFSSL *)
const unsigned char *	**wolfSSL_GetClientWriteIV**(WOLFSSL *) ハンドシェイクプロセスから。
const unsigned char *	**wolfSSL_GetServerWriteKey**(WOLFSSL *)

	Name
const unsigned char *	**wolfSSL_GetServerWriteIV**(WOLFSSL *) ハンドシェイクプロセスから。
int	**wolfSSL_GetKeySize**(WOLFSSL *)
int	**wolfSSL_GetIVSize**(WOLFSSL *)WolfSSL 構造体に保持されている Specs 構造体の IV_SIZE メンバーを返します。
int	**wolfSSL_GetSide**(WOLFSSL *)
int	**wolfSSL_IsTLSv1_1**(WOLFSSL *) 少なくとも TLS バージョン 1.1 以上です。
int	**wolfSSL_GetBulkCipher**(WOLFSSL *) ハンドシェイクから。
int	**wolfSSL_GetCipherBlockSize**(WOLFSSL *) ハンドシェイク。
int	**wolfSSL_GetAeadMacSize**(WOLFSSL *) ハンドシェーク。暗号タイプの wolfssl_aead_type の場合。
int	**wolfSSL_GetHmacSize**(WOLFSSL *) ハンドシェーク。wolfssl_aead_type 以外の暗号タイプの場合。
int	**wolfSSL_GetHmacType**(WOLFSSL *) ハンドシェーク。wolfssl_aead_type 以外の暗号タイプの場合。
int	**wolfSSL_GetCipherType**(WOLFSSL *) ハンドシェイクから。
int	**wolfSSL_SetTlsHmacInner**(WOLFSSL * ssl, byte * inner, word32 sz, int content, int verify) 送受信結果は、少なくとも wolfssl_gethmacsize() バイトであるべきである内部に書き込まれます。メッセージのサイズは SZ で指定され、内容はメッセージの種類であり、検証はこれがピアメッセージの検証であるかどうかを指定します。wolfssl_aead_type を除く暗号タイプに有効です。
void	**wolfSSL_CTX_SetEccSignCb**(WOLFSSL_CTX * ctx, CallbackEccSign cb) コールバックは成功の場合は 0 を返すか、エラーの場合は <0 です。SSL と CTX ポインタはユーザーの利便性に利用できます。INS は入力バッファーが入力の長さを表します。OUT は、署名の結果を保存する必要がある出力バッファです。OUTSZ は、呼び出し時に出力バッファのサイズを指定する入力/出力変数であり、署名の実際のサイズを戻す前に格納する必要があります。keyder は ASN1 フォーマットの ECC 秘密鍵であり、Keysz は鍵のキーの長さです。コールバックの例は、wolfssl / test.h myeccsign() を見つけることができます。
void	**wolfSSL_SetEccSignCtx**(WOLFSSL * ssl, void * ctx)CTX へのコンテキスト。
void *	**wolfSSL_GetEccSignCtx**(WOLFSSL * ssl) 以前に wolfssl_seteccsignctx() で保存されていたコンテキスト。

	Name
void	**wolfSSL_CTX_SetEccSignCtx**(WOLFSSL_CTX * ctx, void * userCtx)CTX へのコンテキスト。
void *	**wolfSSL_CTX_GetEccSignCtx**(WOLFSSL_CTX * ctx) 以前に wolfssl_seteccsignctx() で保存されていたコンテキスト。
void	**wolfSSL_CTX_SetEccVerifyCb**(WOLFSSL_CTX * ctx, CallbackEccVerify cb) コールバックは成功の場合は 0 を返すか、エラーの場合は <0 です。SSL と CTX ポインタはユーザーの利便性に利用できます。SIG は検証の署名であり、SIGSZ は署名の長さを表します。ハッシュはメッセージのダイジェストを含む入力バッファであり、HASHSZ はハッシュの長さを意味します。結果は、検証の結果を格納する出力変数、成功のために 1、失敗のために 0 を記憶する必要があります。keyder は ASN1 フォーマットの ECC 秘密鍵であり、Keysz はキーのキーの長さです。コールバックの例は、wolfssl / test.h myeccverify() を見つけることができます。
void	**wolfSSL_SetEccVerifyCtx**(WOLFSSL * ssl, void * ctx)CTX へのコンテキスト。
void *	**wolfSSL_GetEccVerifyCtx**(WOLFSSL * ssl) 以前に wolfssl_setecverifyctx() で保存されていたコンテキスト。
void	**wolfSSL_CTX_SetRsaSignCb**(WOLFSSL_CTX * ctx, CallbackRsaSign cb) コールバックは成功の場合は 0 を返すか、エラーの場合は <0 です。SSL と CTX ポインタはユーザーの利便性に利用できます。INS は入力バッファーが入力の長さを表します。OUT は、署名の結果を保存する必要がある出力バッファです。OUTSZ は、呼び出し時に出力バッファのサイズを指定する入力/出力変数であり、署名の実際のサイズを戻す前に格納する必要があります。keyder は ASN1 フォーマットの RSA 秘密鍵であり、Keysz はバイト数のキーの長さです。コールバックの例は、wolfssl / test.h myrsasign() を見つけることができます。
void	**wolfSSL_SetRsaSignCtx**(WOLFSSL * ssl, void * ctx)ctx に。
void *	**wolfSSL_GetRsaSignCtx**(WOLFSSL * ssl) 以前に wolfssl_setrsAsignctx() で保存されていたコンテキスト。
void	**wolfSSL_CTX_SetRsaVerifyCb**(WOLFSSL_CTX * ctx, CallbackRsaVerify cb) コールバックは、成功のための平文バイト数または <0 エラーの場合は <0 を返すべきです。SSL と CTX ポインタはユーザーの利便性に利用できます。SIG は検証の署名であり、SIGSZ は署名の長さを表します。復号化プロセスとパディングの後に検証バッファの先頭に設定する必要があります。keyder は ASN1 形式の RSA 公開鍵であり、Keysz はキーのキーの長さです。コールバックの例は、wolfssl / test.h myrsaverify() を見つけることができます。

	Name
void	**wolfSSL_SetRsaVerifyCtx**(WOLFSSL * ssl, void * ctx)CTX へのコンテキスト。
void *	**wolfSSL_GetRsaVerifyCtx**(WOLFSSL * ssl) 以前に wolfssl_setrsaverifyctx() で保存されていたコンテキスト。
void	**wolfSSL_CTX_SetRsaEncCb**(WOLFSSL_CTX * ctx, CallbackRsaEnc cb) 暗号化します。コールバックは成功の場合は 0 を返すか、エラーの場合は <0 です。SSL と CTX ポインタはユーザーの利便性に利用できます。IN は入力バッファですが、INSZ は入力の長さを表します。暗号化の結果を保存する必要がある出力バッファです。OUTSZ は、呼び出し時に出力バッファのサイズを指定する入力/出力変数であり、暗号化の実際のサイズは戻って前に格納されるべきです。keyder は ASN1 形式の RSA 公開鍵であり、Keysz はキーのキーの長さです。例コールバックの例は、wolfssl / test.h myrsaenc() を見つけることができます。
void	**wolfSSL_SetRsaEncCtx**(WOLFSSL * ssl, void * ctx)CTX へのコールバックコンテキスト。
void *	**wolfSSL_GetRsaEncCtx**(WOLFSSL * ssl) コールバックコンテキストは、wolfssl_setrsaencctx() で以前に保存されていました。
void	**wolfSSL_CTX_SetRsaDecCb**(WOLFSSL_CTX * ctx, CallbackRsaDec cb) 復号化します。コールバックは、成功のための平文バイト数または <0 エラーの場合は <0 を返すべきです。SSL と CTX ポインタはユーザーの利便性に利用できます。IN は、復号化する入力バッファが入力の長さを表します。復号化プロセスおよび任意のパディングの後、復号化バッファの先頭に設定する必要があります。keyder は ASN1 フォーマットの RSA 秘密鍵であり、Keysz はバイト数のキーの長さです。コールバックの例は、wolfssl / test.h myrsadec() を見つけることができます。
void	**wolfSSL_SetRsaDecCtx**(WOLFSSL * ssl, void * ctx)CTX へのコールバックコンテキスト。
void *	**wolfSSL_GetRsaDecCtx**(WOLFSSL * ssl) コールバックコンテキストは、wolfssl_setrsadecctx() で以前に保存されていました。
void	**wolfSSL_CTX_SetCACb**(WOLFSSL_CTX * ctx, CallbackCACache cb) 新しい CA 証明書が WolfSSL にロードされたときに呼び出される（WolfSSL_CTX）。コールバックには、符号化された証明書を持つバッファが与えられます。
WOLFSSL_CERT_MANAGER *	**wolfSSL_CertManagerNew_ex**(void * heap) 新しい証明書マネージャコンテキストを割り当てて初期化します。このコンテキストは、SSL のニーズとは無関係に使用できます。証明書をロードしたり、証明書を確認したり、失効状況を確認したりするために使用することができます。

	Name
WOLFSSL_CERT_MANAGER *	**wolfSSL_CertManagerNew**(void) 新しい証明書マネージャコンテキストを割り当てて初期化します。このコンテキストは、SSL のニーズとは無関係に使用できます。証明書をロードしたり、証明書を確認したり、失効状況を確認したりするために使用することができます。
void	**wolfSSL_CertManagerFree**(WOLFSSL_CERT_MANAGER *) 証明書マネージャのコンテキストに関連付けられているすべてのリソースを解放します。証明書マネージャを使用する必要がなくなるときにこれを呼び出します。
int	**wolfSSL_CertManagerLoadCA**(WOLFSSL_CERT_MANAGER * cm, const char * f, const char * d)Manager コンテキストへの CA 証明書のロードの場所を指定します。PEM 証明書カフェイルには、複数の信頼できる CA 証明書が含まれている可能性があります。capath が null でない場合、PEM 形式の CA 証明書を含むディレクトリを指定します。
int	**wolfSSL_CertManagerLoadCABuffer**(WOLFSSL_CERT_MANAGER * cm, const unsigned char * in, long sz, int format)wolfssl_ctx_load_verify_buffer を呼び出して、関数に渡された CM 内の情報を失うことなく一時的な CM を使用してその結果を返すことによって CA バッファをロードします。
int	**wolfSSL_CertManagerUnloadCAs**(WOLFSSL_CERT_MANAGER * cm) この関数は CA 署名者リストをアンロードします。
int	**wolfSSL_CertManagerUnload_trust_peers**(WOLFSSL_CERT_MA * cm) 関数は信頼できるピアリンクリストを解放し、信頼できるピアリストのロックを解除します。
int	**wolfSSL_CertManagerVerify**(WOLFSSL_CERT_MANAGER * cm, const char * f, int format) 証明書マネージャのコンテキストで確認する証明書を指定します。フォーマットは SSL_FILETYPE_PEM または SSL_FILETYPE_ASN1 にすることができます。
int	**wolfSSL_CertManagerVerifyBuffer**(WOLFSSL_CERT_MANAGER * cm, const unsigned char * buff, long sz, int format) 証明書マネージャのコンテキストを使用して確認する証明書バッファを指定します。フォーマットは SSL_FILETYPE_PEM または SSL_FILETYPE_ASN1 にすることができます。
void	**wolfSSL_CertManagerSetVerify**(WOLFSSL_CERT_MANAGER * cm, VerifyCallback vc) この関数は、証明書マネージャーの verifyCallback 関数を設定します。存在する場合、それはロードされた各 CERT に対して呼び出されます。検証エラーがある場合は、検証コールバックを使用してエラーを過度に乗り越えます。
int	**wolfSSL_CertManagerCheckCRL**(WOLFSSL_CERT_MANAGER * cm, unsigned char * der, int sz)CRL リスト。

	Name
int	**wolfSSL_CertManagerEnableCRL**(WOLFSSL_CERT_MANAGER * cm, int options) 証明書マネージャを使用して証明書を検証するときに証明書失効リストの確認をオンにします。デフォルトでは、CRL チェックはオフです。オプションには、wolfssl_crl_checkall が含まれます。これは、チェーン内の各証明書に対して CRL 検査を実行します。これはデフォルトであるリーフ証明書のみです。
int	**wolfSSL_CertManagerDisableCRL**(WOLFSSL_CERT_MANAGER *) 証明書マネージャを使用して証明書を検証するときに証明書失効リストの確認をオフにします。デフォルトでは、CRL チェックはオフです。この関数を使用して、この Certificate Manager コンテキストを使用して CRL 検査を一時的または恒久的に無効にして、以前は CRL 検査が有効になっていました。
int	**wolfSSL_CertManagerLoadCRL**(WOLFSSL_CERT_MANAGER * cm, const char * path, int type, int monitor) 証明書の失効確認のために証明書を CRL にロードする際にエラーチェックを行い、その後証明書を LoadCRL() へ渡します。
int	**wolfSSL_CertManagerLoadCRLBuffer**(WOLFSSL_CERT_MANAGE * cm, const unsigned char * buff, long sz, int type) この関数は、BufferLoadCRL を呼び出すことによって CRL ファイルをロードします。
int	**wolfSSL_CertManagerSetCRL_Cb**(WOLFSSL_CERT_MANAGER * cm, CbMissingCRL cb) この関数は CRL 証明書マネージャコールバックを設定します。LABE_CRL が定義されていて一致する CRL レコードが見つからない場合、CBMissingCRL は呼び出されます（WolfSSL_CertManagerSetCRL_CB を介して設定）。これにより、CRL を外部に検索してロードすることができます。
int	**wolfSSL_CertManagerFreeCRL**(WOLFSSL_CERT_MANAGER * cm) この関数は証明書マネジャーに保持されている CRL を解放します。アプリケーションは CRL を wolfSSL_CertManagerFreeCRL を呼び出して解放した後に、新しい CRL をロードすることができます。
int	**wolfSSL_CertManagerCheckOCSP**(WOLFSSL_CERT_MANAGER * cm, unsigned char * der, int sz) この機能により、OCSPENABLED OCSPENABLED が OCSP チェックオプションが有効になっていることを意味します。
int	**wolfSSL_CertManagerEnableOCSP**(WOLFSSL_CERT_MANAGER * cm, int options)OCSP がオフになっている場合は OCSP をオンにし、[設定] オプションを使用可能になっている場合。
int	**wolfSSL_CertManagerDisableOCSP**(WOLFSSL_CERT_MANAGER *)OCSP 証明書の失効を無効にします。

	Name
int	**wolfSSL_CertManagerSetOCSPOverrideURL**(WOLFSSL_CERT_M * cm, const char * url) この関数は、URL を wolfssl_cert_manager 構造体の OCSpoverrideURL メンバーにコピーします。
int	**wolfSSL_CertManagerSetOCSP_Cb**(WOLFSSL_CERT_MANAGER * cm, CbOCSPIO ioCb, CbOCSPRespFree respFreeCb, void * ioCbCtx) この関数は、wolfssl_cert_manager の OCSP コールバックを設定します。
int	**wolfSSL_CertManagerEnableOCSPStapling**(WOLFSSL_CERT_MA * cm) この関数は、オプションをオンにしないと OCSP ステープルをオンにします。オプションを設定します。
int	**wolfSSL_EnableCRL**(WOLFSSL * ssl, int options)
int	**wolfSSL_DisableCRL**(WOLFSSL * ssl)
int	**wolfSSL_LoadCRL**(WOLFSSL * ssl, const char * path, int type, int monitor) 失効検査の証明書
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)wolfssl_cert_manager 構造体。
int	**wolfSSL_SetOCSP_Cb**(WOLFSSL * ssl, CbOCSPIO ioCb, CbOCSPRespFree respFreeCb, void * ioCbCtx)wolfssl_cert_manager 構造体。
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)wolfssl_certmanagerLoadcr()。
int	**wolfSSL_CTX_SetCRL_Cb**(WOLFSSL_CTX * ctx, CbMissingCRL cb)wolfssl_certmanagersetCRL_CB を呼び出して、WolfSSL_CERT_MANAGER 構造のメンバー。
int	**wolfSSL_CTX_EnableOCSP**(WOLFSSL_CTX * ctx, int options)wolfssl の機能オプションの値が 1 つ以上のオプションで構成されている場合は、次のオプションを 1 つ以上にします.wolfssl_ocsp_enable _ OCSP ルックアップを有効にする wolfssl_ocsp_url_override _ 証明書の URL の代わりに URL をオーバーライドします。オーバーライド URL は、wolfssl_ctx_setocsp_overrideURL() 関数を使用して指定されます。この関数は、wolfssl が OCSP サポート（–enable-ocsp、#define hane_ocsp）でコンパイルされたときにのみ OCSP オプションを設定します。

	Name
int	**wolfSSL_CTX_DisableOCSP**(WOLFSSL_CTX *)wolfssl_cert_manager 構造体の OCSPENABLED メンバーに影響を与えます。
int	**wolfSSL_CTX_SetOCSP_OverrideURL**(WOLFSSL_CTX * ctx, const char * url)wolfssl_csp_url_override オプションが wolfssl_ctx_enableocsp を使用して設定されていない限り、OCSP は個々の証明書にある URL を使用します。
int	**wolfSSL_CTX_SetOCSP_Cb**(WOLFSSL_CTX * ctx, CbOCSPIO ioCb, CbOCSPRespFree respFreeCb, void * ioCbCtx)
int	**wolfSSL_CTX_EnableOCSPStapling**(WOLFSSL_CTX *)wolfssl_certmanagerEnableOcspStapling()。
void	**wolfSSL_KeepArrays**(WOLFSSL *) 通常、SSL ハンドシェイクの最後に、WolfSSL は一時的なアレイを解放します。ハンドシェイクが始まる前にこの関数を呼び出すと、WolfSSL は一時的な配列を解放するのを防ぎます。Wolfssl_get_keys() または PSK のヒントなどのものには、一時的な配列が必要になる場合があります。ユーザが一時的な配列で行われると、wolfssl_freearray() のいずれかが即座にリソースを解放することができ、あるいは、関連する SSL オブジェクトが解放されたときにリソースが解放されるようになる可能性がある。
void	**wolfSSL_FreeArrays**(WOLFSSL *) 通常、SSL ハンドシェイクの最後に、WolfSSL は一時的なアレイを解放します。wolfssl_keeparrays() がハンドシェイクの前に呼び出された場合、WolfSSL は一時的な配列を解放しません。この関数は一時的な配列を明示的に解放し、ユーザーが一時的な配列で行われたときに呼び出されるべきであり、SSL オブジェクトがこれらのリソースを解放するのを待ったくない。
int	**wolfSSL_UseSNI**(WOLFSSL * ssl, unsigned char type, const void * data, unsigned short size)'ssl' パラメータに渡されたオブジェクト。これは、WolfSSL クライアントによって SNI 拡張機能が ClientHello で送信され、WolfSSL Server は ServerHello + SNI または SNI ミスマッチの場合は致命的な Alert Hello + SNI を応答します。
int	**wolfSSL_CTX_UseSNI**(WOLFSSL_CTX * ctx, unsigned char type, const void * data, unsigned short size)SSL コンテキストから作成されたオブジェクトは'ctx' パラメータに渡されました。これは、WolfSSL クライアントによって SNI 拡張機能が ClientHello で送信され、WolfSSL サーバーは ServerHello + SNI または SNI の不一致の場合には致命的な ALERT Hello + SNI を応答します。

	Name
void	**wolfSSL_SNI_SetOptions**(WOLFSSL * ssl, unsigned char type, unsigned char options)'ssl' パラメータに渡された SSL オブジェクト内のサーバー名表示を使用した SSL セッションの動作。オプションを以下に説明します。
void	**wolfSSL_CTX_SNI_SetOptions**(WOLFSSL_CTX * ctx, unsigned char type, unsigned char options)SSL セッションを使用した SSL オブジェクトのサーバ名指示を使用して、SSL コンテキストから作成された SSL オブジェクトから作成されます。オプションを以下に説明します。
int	**wolfSSL_SNI_GetFromBuffer**(const unsigned char * clientHello, unsigned int helloSz, unsigned char type, unsigned char * sni, unsigned int * inOutSz) クライアントによってクライアントから提供された名前表示クライアントによって送信されたメッセージセッションを開始する。SNI を取得するためのコンテキストまたはセッション設定が必要ありません。
unsigned char	**wolfSSL_SNI_Status**(WOLFSSL * ssl, unsigned char type) この関数は SNI オブジェクトのステータスを取得します。
unsigned short	**wolfSSL_SNI_GetRequest**(WOLFSSL * ssl, unsigned char type, void ** data)SSL セッションでクライアントによって提供されるサーバー名の表示。
int	**wolfSSL_UseALPN**(WOLFSSL * ssl, char * protocol_name_list, unsigned int protocol_name_listSz, unsigned char options)wolfssl セッションに ALPN を設定します。
int	**wolfSSL_ALPN_GetProtocol**(WOLFSSL * ssl, char ** protocol_name, unsigned short * size) この関数は、サーバーによって設定されたプロトコル名を取得します。
int	**wolfSSL_ALPN_GetPeerProtocol**(WOLFSSL * ssl, char ** list, unsigned short * listSz) この関数は、alpn_client_list データを SSL オブジェクトからバッファにコピーします。
int	**wolfSSL_UseMaxFragment**(WOLFSSL * ssl, unsigned char mfl)'ssl' パラメータに渡された SSL オブジェクト内の最大フラグメント長。これは、最大フラグメント長拡張機能が WolfSSL クライアントによって ClientHello で送信されることを意味します。
int	**wolfSSL_CTX_UseMaxFragment**(WOLFSSL_CTX * ctx, unsigned char mfl)SSL コンテキストから作成された SSL オブジェクトの最大フラグメント長さ'ctx' パラメータに渡されました。これは、最大フラグメント長拡張機能が WolfSSL クライアントによって ClientHello で送信されることを意味します。

	Name
int	**wolfSSL_UseTruncatedHMAC**(WOLFSSL * ssl)'ssl' パラメータに渡された SSL オブジェクト内の truncated HMAC。これは、切り捨てられた HMAC 拡張機能が WolfSSL クライアントによって ClientHello で送信されることを意味します。
int	**wolfSSL_CTX_UseTruncatedHMAC**(WOLFSSL_CTX * ctx)'ctx' パラメータに渡された SSL コンテキストから作成された SSL オブジェクトのための Truncated HMAC。これは、切り捨てられた HMAC 拡張機能が WolfSSL クライアントによって ClientHello で送信されることを意味します。
int	**wolfSSL_UseOCSPStapling**(WOLFSSL * ssl, unsigned char status_type, unsigned char options)OCSP で提示された証明書失効チェックのコストを下げます。
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)OCSP ステイプルのために。
int	**wolfSSL_UseSupportedCurve**(WOLFSSL * ssl, word16 name) サポートされている楕円曲線拡張子は、'SSL' パラメータに渡された SSL オブジェクトでサポートされています。これは、サポートされているカーブが WolfSSL クライアントによって ClientHello で送信されることを意味します。この機能は複数の曲線を有効にするために複数の時間と呼ぶことができます。
int	**wolfSSL_CTX_UseSupportedCurve**(WOLFSSL_CTX * ctx, word16 name) サポートされている楕円曲線は、'ctx' パラメータに渡された SSL コンテキストから作成された SSL オブジェクトの拡張子です。これは、サポートされているカーブが WolfSSL クライアントによって ClientHello で送信されることを意味します。この機能は複数の曲線を有効にするために複数の時間と呼ぶことができます。
int	**wolfSSL_UseSecureRenegotiation**(WOLFSSL * ssl) この関数は、供給された WOLFSSL 構造の安全な再交渉を強制します。これはお勧めできません。
int	**wolfSSL_Rehandshake**(WOLFSSL * ssl) この関数は安全な再交渉ハンドシェイクを実行します。これは、WolfSSL がこの機能を妨げるように強制されます。

	Name
int	**wolfSSL_UseSessionTicket**(WOLFSSL * ssl) セッションチケットを使用するように WolfSSL 構造を強制します。定数 hous_session_ticket を定義し、定数 NO_WOLFSSL_CLIENT をこの関数を使用するように定義しないでください。
int	**wolfSSL_CTX_UseSessionTicket**(WOLFSSL_CTX * ctx) この関数は、セッションチケットを使用するように WolfSSL コンテキストを設定します。
int	**wolfSSL_get_SessionTicket**(WOLFSSL * ssl, unsigned char * buf, word32 * bufSz) この機能は、セッション構造のチケットメンバーをバッファにコピーします。
int	**wolfSSL_set_SessionTicket**(WOLFSSL * ssl, const unsigned char * buf, word32 bufSz) この関数は、WolfSSL 構造体内の wolfssl_session 構造体のチケットメンバーを設定します。関数に渡されたバッファはメモリにコピーされます。
int	**wolfSSL_set_SessionTicket_cb**(WOLFSSL * ssl, CallbackSessionTicket cb, void * ctx)CallbackSessionTicket は、int（* callbacksessionTicket）（wolfssl 、*const unsigned char* 、int、void *）の関数ポインタです。
int	**wolfSSL_send_SessionTicket**(WOLFSSL * ssl) この関数は TLS1.3 ハンドシェークが確立したあとでセッションチケットを送信します。
int	**wolfSSL_CTX_set_TicketEncCb**(WOLFSSL_CTX * ctx, SessionTicketEncCb)RFC 5077 で指定されているセッションチケットをサポートするためのサーバーが。
int	**wolfSSL_CTX_set_TicketHint**(WOLFSSL_CTX * ctx, int) サーバーサイドの使用のために。
int	**wolfSSL_CTX_set_TicketEncCtx**(WOLFSSL_CTX * ctx, void *) 折り返し電話。サーバーサイドの使用のために。
void *	**wolfSSL_CTX_get_TicketEncCtx**(WOLFSSL_CTX * ctx) 折り返し電話。サーバーサイドの使用のために。
int	**wolfSSL_SetHsDoneCb**(WOLFSSL * ssl, HandShakeDoneCb cb, void * user_ctx) この機能には、WolfSSL 構造の HSDonectx メンバーが設定されています。
int	**wolfSSL_PrintSessionStats**(void) この関数はセッションから統計を印刷します。
int	**wolfSSL_get_session_stats**(unsigned int * active, unsigned int * total, unsigned int * peak, unsigned int * maxSessions) この関数はセッションの統計を取得します。

	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) この関数は CR と SR の値をコピーしてから WC_PRF（疑似ランダム関数）に渡し、その値を返します。
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)TLS キーを導き出すための外部のラッパー。
int	**wolfSSL_connect_ex**(WOLFSSL * ssl, HandShakeCallBack hsCb, TimeoutCallBack toCb, WOLFSSL_TIMEVAL timeout) ハンドシェイクコールバックが設定されます。これは、デバッガが利用できず、スニッフィングが実用的ではない場合に、サポートをデバッグするための組み込みシステムで役立ちます。ハンドシェイクエラーが発生したか否かが呼び出されます。SSL パケットの最大数が既知であるため、動的メモリは使用されません。パケット名を PacketNames [] でアクセスできます。接続拡張機能は、タイムアウト値とともにタイムアウトコールバックを設定することもできます。これは、ユーザーが TCP スタックをタイムアウトするのを待ったくない場合に便利です。この拡張子は、コールバックのどちらか、またはどちらのコールバックも呼び出されません。
int	**wolfSSL_accept_ex**(WOLFSSL * ssl, HandShakeCallBacki hsCb, TimeoutCallBack toCb, WOLFSSL_TIMEVAL timeout) 設定する。これは、デバッガが利用できず、スニッフィングが実用的ではない場合に、サポートをデバッグするための組み込みシステムで役立ちます。ハンドシェイクエラーが発生したか否かが呼び出されます。SSL パケットの最大数が既知であるため、動的メモリは使用されません。パケット名を PacketNames [] でアクセスできます。接続拡張機能は、タイムアウト値とともにタイムアウトコールバックを設定することもできます。これは、ユーザーが TCP スタックをタイムアウトするのを待ったくない場合に便利です。この拡張子は、コールバックのどちらか、またはどちらのコールバックも呼び出されません。
long	**wolfSSL_BIO_set_fp**(WOLFSSL_BIO * bio, XFILE fp, int c) これは BIO の内部ファイルポインタを設定するために使用されます。
long	**wolfSSL_BIO_get_fp**(WOLFSSL_BIO * bio, XFILE * fp) この関数は、

	Name
int	**wolfSSL_check_private_key**(const WOLFSSL * ssl) この関数は、秘密鍵が使用されている証明書との一致であることを確認します。
int	**wolfSSL_X509_get_ext_by_NID**(const WOLFSSL_X509 * x509, int nid, int lastPos) この機能は、渡された NID 値に一致する拡張索引を探して返します。
void *	**wolfSSL_X509_get_ext_d2i**(const WOLFSSL_X509 * x509, int nid, int * c, int * idx) この関数は、渡された NID 値に合った拡張子を探して返します。
int	**wolfSSL_X509_digest**(const WOLFSSL_X509 * x509, const WOLFSSL_EVP_MD * digest, unsigned char * buf, unsigned int * len) この関数は DER 証明書のハッシュを返します。
int	**wolfSSL_use_certificate**(WOLFSSL * ssl, WOLFSSL_X509 * x509) ハンドシェイク中に使用するために、WolfSSL 構造の証明書を設定するために使用されます。
int	**wolfSSL_use_certificate_ASN1**(WOLFSSL * ssl, unsigned char * der, int derSz)
int	**wolfSSL_use_PrivateKey**(WOLFSSL * ssl, WOLFSSL_EVP_PKEY * pkey) これは WolfSSL 構造の秘密鍵を設定するために使用されます。
int	**wolfSSL_use_PrivateKey_ASN1**(int pri, WOLFSSL * ssl, unsigned char * der, long derSz) これは WolfSSL 構造の秘密鍵を設定するために使用されます。DER フォーマットのキーバッファが予想されます。
int	**wolfSSL_use_RSAPrivateKey_ASN1**(WOLFSSL * ssl, unsigned char * der, long derSz) これは WolfSSL 構造の秘密鍵を設定するために使用されます。DER フォーマットの RSA キーバッファが予想されます。
WOLFSSL_DH *	**wolfSSL_DSA_dup_DH**(const WOLFSSL_DSA * r) この関数は、DSA のパラメータを新しく作成された WOLFSSL_DH 構造体に重複しています。
int	**wolfSSL_SESSION_get_master_key**(const WOLFSSL_SESSION * ses, unsigned char * out, int outSz) これはハンドシェイクを完了した後にマスターキーを取得するために使用されます。
int	**wolfSSL_SESSION_get_master_key_length**(const WOLFSSL_SESSION * ses) これはマスター秘密鍵の長さを取得するために使用されます。
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) この関数は BIO から DER バッファを取得し、それを WolfSSL_X509 構造に変換します。
WOLFSSL_X509_STORE *	**wolfSSL_CTX_get_cert_store**(WOLFSSL_CTX * ctx)

	Name
size_t	**wolfSSL_BIO_ctrl_pending**(WOLFSSL_BIO * b) 保留中のバイト数を読み取る数を取得します。BIO タイプが BIO_BIO の場合、ペアから読み取る番号です。BIO に SSL オブジェクトが含まれている場合は、SSL オブジェクトからのデータを保留中です（WolfSSL_Pending（SSL））。bio_memory タイプがある場合は、メモリバッファのサイズを返します。
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) これは CTX で設定されたパスワードコールバックのゲッター関数です。
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) この関数は wolfssl_pem_read_bio_x509 と同じように動作します。AUX は、信頼できる/拒否されたユースケースや人間の読みやすさのためのフレンドリーな名前などの追加情報を含むことを意味します。
long	**wolfSSL_CTX_set_tmp_dh**(WOLFSSL_CTX * ctx, WOLFSSL_DH * dh)WOLFSSL_CTX 構造体の DH メンバーを diffie-hellman パラメータで初期化します。
WOLFSSL_DSA *	**wolfSSL_PEM_read_bio_DSAparams**(WOLFSSL_BIO * bp, WOLFSSL_DSA ** x, wc_pem_password_cb * cb, void * u) この関数は、BIO の PEM バッファから DSA パラメータを取得します。
unsigned long	**wolfSSL_ERR_peek_last_error**(void) この関数は、wolfssl_Error に遭遇した最後のエラーの絶対値を返します。 **WOLF_STACK_OF**(WOLFSSL_X509) const この関数はピアの証明書チェーンを取得します。
long	**wolfSSL_CTX_clear_options**(WOLFSSL_CTX * ctx, long opt) この関数は、WOLFSSL_CTX オブジェクトのオプションビットをリセットします。
int	**wolfSSL_set_jobject**(WOLFSSL * ssl, void * objPtr) この関数は、WolfSSL 構造の jobjectref メンバーを設定します。
void *	**wolfSSL_get_jobject**(WOLFSSL * ssl) この関数は、wolfssl 構造の jobjectref メンバーを返します。
int	**wolfSSL_set_msg_callback**(WOLFSSL * ssl, SSL_Msg_Cb cb) この関数は SSL 内のコールバックを設定します。コールバックはハンドシェイクメッセージを観察することです。CB の NULL 値はコールバックをリセットします。

	Name
int	**wolfSSL_set_msg_callback_arg**(WOLFSSL * ssl, void * arg) この関数は、SSL 内の関連コールバックコンテキスト値を設定します。値はコールバック引数に渡されます。
char *	**wolfSSL_X509_get_next_altname**(WOLFSSL_X509 * x509) この関数は、存在する場合は、ピア証明書から altname を返します。
WOLFSSL_ASN1_TIME *	**wolfSSL_X509_get_notBefore**(WOLFSSL_X509 * x509) 関数は、x509 が null のかどうかを確認し、そうでない場合は、WOLFSSL_X509 構造体の NotBefore メンバーを返します。
int	**wolfSSL_connectは、ブロッキングとノンブロッキング I/O の両方で動作します。下層の I/O がノンブロッキングの場合、wolfSSL_connect() は、下層の I/O が wolfSSL_connect の要求（送信データ、受信データ）を満たすことができなかったときには即戻ります。この場合、wolfSSL_get_error() の呼び出しで SSL_ERROR_WANT_READ または SSL_ERROR_WANT_WRITE のいずれかが返されます。呼び出したプロセスは、下層の I/O がが READY になった時点で、WOLFSSL が停止したときから再開できるように wolfSSL_connect() への呼び出しを繰り返す必要があります。これには select() を使用して必要な条件が整ったかどうかを確認できます。ブロッキング I/O を使用する場合は、ハンドシェークが終了するかエラーが発生するまで戻ってきません。wolfSSL は OpenSSL と比べて証明書検証に異なるアプローチを取ります。クライアントのデフォルトポリシーはサーバーを認証することです。これは、CA 証明書を読み込まない場合、サーバーを確認することができず”_155” のエラーコードが返されます。OpenSSL と同じ振る舞い（つまり、CA 証明書のロードなしでサーバー認証を成功させる）を取らせたい場合には、セキュリティ面でお勧めはしませんが、SSL_CTX_SET_VERIFY（ctx、SSL_VERIFY_NONE、0) を呼び出すことで可能となります。
int	**wolfSSL_send_hrr_cookie**(WOLFSSL * ssl, const unsigned char * secret, unsigned int secretSz) この関数はサーバー側で呼び出されて、HellORetryRequest メッセージに Cookie を含める必要があることを示します。Cookie は現在のトランスクリプトのハッシュを保持しているので、別のサーバープロセスは応答で ClientHello を処理できます。秘密は Cookie データの整合性チェックを Generting するときに使用されます。

	Name
int	**wolfSSL_disable_hrr_cookie**(WOLFSSL * ssl) この関数はサーバー側で呼び出され、HelloRetryRequest メッセージがクッキーを含んではならないこと、DTLSv1.3 が使用されている場合にはクッキーの交換がハンドシェークに含まれないことを表明します。DTLSv1.3 ではクッキー交換を行わないとサーバーが DoS/Amplification 攻撃を受けやすくなる可能性があることに留意してください。
int	**wolfSSL_CTX_no_ticket_TLSv13**(WOLFSSL_CTX * ctx) この関数はサーバー上で呼び出され、ハンドシェイク完了時にセッション再開のためのセッションチケットの送信を行わないようにします。
int	**wolfSSL_no_ticket_TLSv13**(WOLFSSL * ssl) ハンドシェイクが完了すると、この関数はサーバー上で再開セッションチケットの送信を停止するように呼び出されます。
int	**wolfSSL_CTX_no_dhe_psk**(WOLFSSL_CTX * ctx) この関数は、Authentication にプリシェアキーを使用している場合、DIFFIE-HELLMAN（DH）スタイルのキー交換を許可する TLS V1.3 WolfSSL コンテキストで呼び出されます。
int	**wolfSSL_no_dhe_psk**(WOLFSSL * ssl) この関数は、事前共有鍵を使用している TLS V1.3 クライアントまたはサーバーで、に Diffie-Hellman（DH）スタイルの鍵交換を許可しないように設定します。
int	**wolfSSL_update_keys**(WOLFSSL * ssl) この関数は、TLS v1.3 クライアントまたはサーバーの wolfssl で呼び出されて、キーのロールオーバーを強制します。KeyUpdate メッセージがピアに送信され、新しいキーが暗号化のために計算されます。ピアは KeyUpdate メッセージを送り、新しい復号化キー WIL を計算します。この機能は、ハンドシェイクが完了した後にのみ呼び出すことができます。
int	**wolfSSL_key_update_response**(WOLFSSL * ssl, int * required) この関数は、TLS v1.3 クライアントまたはサーバーの wolfssl で呼び出され、キーのロールオーバーが進行中かどうかを判断します。wolfssl_update_keys() が呼び出されると、KeyUpdate メッセージが送信され、暗号化キーが更新されます。復号化キーは、応答が受信されたときに更新されます。
int	**wolfSSL_CTX_allow_post_handshake_auth**(WOLFSSL_CTX * ctx) この関数は、TLS v1.3 クライアントの WolfSSL コンテキストで呼び出され、クライアントはサーバーからの要求に応じて Post Handshake を送信できるようにします。これは、クライアント認証などを必要としないページを持つ Web サーバーに接続するときに役立ちます。

	Name
int	**wolfSSL_allow_post_handshake_auth**(WOLFSSL * ssl) この関数は、TLS V1.3 クライアント WolfSSL で呼び出され、クライアントはサーバーからの要求に応じてハンドシェイクを送ります。handshake クライアント認証拡張機能は ClientHello で送信されます。これは、クライアント認証などを必要としないページを持つ Web サーバーに接続するときに役立ちます。
int	**wolfSSL_request_certificate**(WOLFSSL * ssl) この関数は、TLS v1.3 クライアントからクライアント証明書を要求します。これは、Web サーバーがクライアント認証やその他のものを必要とするページにサービスを提供している場合に役立ちます。接続で最大 256 の要求を送信できます。
int	**wolfSSL_CTX_set1_groups_list**(WOLFSSL_CTX * ctx, char * list) この関数は楕円曲線グループのリストを設定して、WolfSSL コンテキストを希望の順に設定します。リストはヌル終了したテキスト文字列、およびコロン区切りリストです。この関数を呼び出して、TLS v1.3 接続で使用する鍵交換楕円曲線パラメータを設定します。
int	**wolfSSL_set1_groups_list**(WOLFSSL * ssl, char * list) この関数は楕円曲線グループのリストを設定して、WolfSSL を希望の順に設定します。リストはヌル終了したテキスト文字列、およびコロン区切りリストです。この関数を呼び出して、TLS v1.3 接続で使用する鍵交換楕円曲線パラメータを設定します。
int	**wolfSSL_preferred_group**(WOLFSSL * ssl) この関数は、クライアントが TLS v1.3 ハンドシェイクで使用することを好む鍵交換グループを返します。この情報を完了した後にこの機能を呼び出して、サーバーがどのグループが予想されるようにこの情報が将来の接続で使用できるようになるかを決定するために、この情報が将来の接続で鍵交換のための鍵ペアを事前生成することができます。
int	**wolfSSL_CTX_set_groups**(WOLFSSL_CTX * ctx, int * groups, int count) この関数は楕円曲線グループのリストを設定して、WolfSSL コンテキストを希望の順に設定します。リストは、Count で指定された識別子の数を持つグループ識別子の配列です。この関数を呼び出して、TLS v1.3 接続で使用する鍵交換楕円曲線パラメータを設定します。
int	**wolfSSL_set_groups**(WOLFSSL * ssl, int * groups, int count) この関数は、wolfssl を許すために楕円曲線グループのリストを設定します。リストは、Count で指定された識別子の数を持つグループ識別子の配列です。この関数を呼び出して、TLS v1.3 接続で使用する鍵交換楕円曲線パラメータを設定します。

	Name
int	**wolfSSL_connect_TLSv13は、ブロックとノンブロック I/O の両方で動作します。下層 I/O がノンブロッキングの場合、wolfSSL_connect() は、下層 I/O が wolfssl_connect の要求を満たすことができなかったときに戻ります。この場合、wolfSSL_get_error() への呼び出しは SSL_ERROR_WANT_READ または SSL_ERROR_WANT_WRITE のいずれかを生成します。通話プロセスは、下層 I/O が READY および WOLFSSL が停止したときに wolfssl_connect() への呼び出しを繰り返す必要があります。ノンブロッキングソケットを使用する場合は、何も実行する必要がありますが、select() を使用して必要な条件を確認できます。基礎となる入出力がブロックされている場合、wolfssl_connect() はハンドシェイクが終了したら、またはエラーが発生したらのみ戻ります。WolfSSL は OpenSSL よりも証明書検証に異なるアプローチを取ります。クライアントのデフォルトポリシーはサーバーを確認することです。これは、CAS を読み込まない場合、サーバーを確認することができ、確認できません（_155）。SSL_CONNECT を持つことの OpenSSL の動作が成功した場合は、サーバーを検証してセキュリティを抑えることができます。SSL_CTX_SET_VERIFY（CTX、SSL_VERIFY_NONE、0）。ssl_new() を呼び出す前に。お勧めできませんが。 **wolfSSL_accept_TLSv13は、ブロックとノンブロッキング I/O の両方で動作します。下層の入出力がノンブロッキングである場合、wolfSSL_accept() は、下層の I/O が wolfSSL_accept の要求を満たすことができなかったときに戻ります。この場合、wolfSSL_get_error() への呼び出しは SSL_ERROR_WANT_READ または SSL_ERROR_WANT_WRITE のいずれかを生成します。通話プロセスは、読み取り可能なデータが使用可能であり、wolfssl が停止した場所を拾うときに、wolfssl_accept の呼び出しを繰り返す必要があります。ノンブロッキングソケットを使用する場合は、何も実行する必要がありますが、select() を使用して必要な条件を確認できます。下層の I/O がブロックされている場合、wolfssl_accept() はハンドシェイクが終了したら、またはエラーが発生したら戻ります。古いバージョンの ClientHello メッセージがサポートされていますが、TLS v1.3 接続を期待するときにこの関数を呼び出します。

	Name
int	**wolfSSL_CTX_set_max_early_data**(WOLFSSL_CTX * ctx, unsigned int sz) この関数は、WolfSSL コンテキストを使用して TLS V1.3 サーバーによって受け入れられるアーリーデータの最大量を設定します。この関数を呼び出して、再生攻撃を軽減するためのプロセスへのアーリーデータの量を制限します。初期のデータは、セッションチケットが送信されたこと、したがってセッションチケットが再開されるたびに同じ接続の鍵から派生した鍵によって保護されます。値は再開のためにセッションチケットに含まれています。ゼロの値は、セッションチケットを使用してクライアントによってアーリーデータを送信することを示します。アーリーデータバイト数をアプリケーションで実際には可能な限り低く保つことをお勧めします。
int	**wolfSSL_set_max_early_data**(WOLFSSL * ssl, unsigned int sz) この関数は、WolfSSL コンテキストを使用して TLS V1.3 サーバーによって受け入れられるアーリーデータの最大量を設定します。この関数を呼び出して、再生攻撃を軽減するためプロセスへのアーリーデータの量を制限します。初期のデータは、セッションチケットが送信されたこと、したがってセッションチケットが再開されるたびに同じ接続の鍵から派生した鍵によって保護されます。値は再開のためにセッションチケットに含まれています。ゼロの値は、セッションチケットを使用してクライアントによってアーリーデータを送信することを示します。アーリーデータバイト数をアプリケーションで実際には可能な限り低く保つことをお勧めします。
int	**wolfSSL_write_early_dataまたは wolfSSL_connect_tlsv13() の代わりにこの関数を呼び出して、サーバーに接続してハンドシェイクにデータを送ります。この機能はクライアントでのみ使用されます。
int	**wolfSSL_read_early_data**(WOLFSSL * ssl, void * data, int sz, int * outSz) この関数は、再開時にクライアントからの早期データを読み取ります。wolfssl_accept() または wolfssl_accept_tlsv13() の代わりにこの関数を呼び出して、クライアントを受け入れ、ハンドシェイク内の早期データを読み取ります。ハンドシェイクよりも早期データがない場合は、通常として処理されます。この機能はサーバーでのみ使用されます。
void	**wolfSSL_CTX_set_psk_client_tls13_callback**(WOLFSSL_CTX * ctx, wc_psk_client_tls13_callback cb) この関数は、TLS v1.3 接続のプレシェア鍵（PSK）クライアント側コールバックを設定します。コールバックは PSK アイデンティティを見つけ、そのキーと、ハンドシェイクに使用する暗号の名前を返します。この関数は、WOLFSSL_CTX 構造体の client_psk_tls13_cb メンバーを設定します。

	Name
void	**wolfSSL_set_psk_client_tls13_callback**(WOLFSSL * ssl, wc_psk_client_tls13_callback cb) この関数は、TLS v1.3 接続のプレシェアキー（PSK）クライアント側コールバックを設定します。コールバックは PSK アイデンティティを見つけ、そのキーと、ハンドシェイクに使用する暗号の名前を返します。この関数は、wolfssl 構造体の Options フィールドの client_psk_tls13_cb メンバーを設定します。
void	**wolfSSL_CTX_set_psk_server_tls13_callback**(WOLFSSL_CTX * ctx, wc_psk_server_tls13_callback cb) この関数は、TLS v1.3 接続用の事前共有鍵（PSK）サーバ側コールバックを設定します。コールバックは PSK アイデンティティを見つけ、そのキーと、ハンドシェイクに使用する暗号の名前を返します。この関数は、wolfssl_ctx 構造体の server_psk_tls13_cb メンバーを設定します。
void	**wolfSSL_set_psk_server_tls13_callback**(WOLFSSL * ssl, wc_psk_server_tls13_callback cb) この関数は、TLS v1.3 接続用の事前共有鍵（PSK）サーバ側コールバックを設定します。コールバックは PSK アイデンティティを見つけ、そのキーと、ハンドシェイクに使用する暗号の名前を返します。この関数は、wolfssl 構造体のオプションフィールドの server_psk_tls13_cb メンバーを設定します。
int	**wolfSSL_UseKeyShare**(WOLFSSL * ssl, word16 group) この関数は、キーペアの生成を含むグループからキーシェアエントリを作成します。Keyshare エクステンションには、鍵交換のための生成されたすべての公開鍵が含まれています。この関数が呼び出されると、指定されたグループのみが含まれます。優先グループがサーバーに対して以前に確立されているときにこの関数を呼び出します。
int	**wolfSSL_NoKeyShares**(WOLFSSL * ssl) この関数は、ClientHello で鍵共有が送信されないように呼び出されます。これにより、ハンドシェイクに鍵交換が必要な場合は、サーバーが HelloretryRequest で応答するように強制します。予想される鍵交換グループが知られておらず、キーの生成を不必要に回避するときにこの機能を呼び出します。鍵交換が必要なときにハンドシェイクを完了するために追加の往復が必要になることに注意してください。
WOLFSSL_METHOD *	**wolfTLSv1_3_server_method_ex**(void * heap) この関数は、アプリケーションがサーバーであることを示すために使用され、TLS 1.3 プロトコルのみをサポートします。この関数は、wolfSSL_CTX_new() を使用して SSL / TLS コンテキストを作成するときに使用される新しい Wolfssl_method 構造体のメモリを割り当てて初期化します。

	Name
WOLFSSL_METHOD *	**wolfTLSv1_3_client_method_ex**(void * heap) この関数は、アプリケーションがクライアントであることを示すために使用され、TLS 1.3 プロトコルのみをサポートします。この関数は、wolfSSL_CTX_new() を使用して SSL / TLS コンテキストを作成するときに使用される新しい Wolfssl_method 構造体のメモリを割り当てて初期化します。
WOLFSSL_METHOD *	**wolfTLSv1_3_server_method**(void) この関数は、アプリケーションがサーバーであることを示すために使用され、TLS 1.3 プロトコルのみをサポートします。この関数は、wolfSSL_CTX_new() を使用して SSL / TLS コンテキストを作成するときに使用される新しい Wolfssl_method 構造体のメモリを割り当てて初期化します。
WOLFSSL_METHOD *	**wolfTLSv1_3_client_method**(void) この関数は、アプリケーションがクライアントであることを示すために使用され、TLS 1.3 プロトコルのみをサポートします。この関数は、wolfSSL_CTX_new() を使用して SSL / TLS コンテキストを作成するときに使用される新しい Wolfssl_method 構造体のメモリを割り当てて初期化します。
WOLFSSL_METHOD *	**wolfTLSv1_3_method_ex**(void * heap) この関数は、まだどちらの側（サーバ/クライアント）を決定していないことを除いて、Wolftlsv1_3_client_method と同様の wolfssl_method を返します。
WOLFSSL_METHOD *	**wolfTLSv1_3_method**(void) この関数は、まだどちらの側（サーバ/クライアント）を決定していないことを除いて、Wolftlsv1_3_client_method と同様の wolfssl_method を返します。
int	**wolfSSL_CTX_set_client_cert_type**(WOLFSSL_CTX * ctx, const char * buf, int len) この関数はクライアント側で呼び出される場合には、サーバー側に Certificate メッセージで送信できる証明書タイプを設定します。サーバー側で呼び出される場合には、受入れ可能なクライアント証明書タイプを設定します。Raw Public Key 証明書を送受信したい場合にはこの関数を使って証明書タイプを設定しなければなりません。設定する証明書タイプは優先度順に格納したバイト配列として渡します。設定するバッファアドレスに NULL を渡すか、あるいはバッファサイズに 0 を渡すと規定値にもどすことができます。規定値は X509 証明書（WOLFSSL_CERT_TYPE_X509）のみを扱う設定となっています。

	Name
int	**wolfSSL_CTX_set_server_cert_type**(WOLFSSL_CTX * ctx, const char * buf, int len) この関数はサーバー側で呼び出される場合には、クライアント側にCertificate メッセージで送信できる証明書タイプを設定します。クライアント側で呼び出される場合には、受入れ可能なサーバー証明書タイプを設定します。Raw Public Key 証明書を送受信したい場合にはこの関数を使って証明書タイプを設定しなければなりません。設定する証明書タイプは優先度順に格納したバイト配列として渡します。設定するバッファアドレスに NULL を渡すか、あるいはバッファサイズに 0 を渡すと規定値にもどすことができます。規定値は X509 証明書（WOLFSSL_CERT_TYPE_X509）のみを扱う設定となっています。
int	**wolfSSL_set_client_cert_type**(WOLFSSL * ssl, const char * buf, int len) この関数はクライアント側で呼び出される場合には、サーバー側にCertificate メッセージで送信できる証明書タイプを設定します。サーバー側で呼び出される場合には、受入れ可能なクライアント証明書タイプを設定します。Raw Public Key 証明書を送受信したい場合にはこの関数を使って証明書タイプを設定しなければなりません。設定する証明書タイプは優先度順に格納したバイト配列として渡します。設定するバッファアドレスに NULL を渡すか、あるいはバッファサイズに 0 を渡すと規定値にもどすことができます。規定値は X509 証明書（WOLFSSL_CERT_TYPE_X509）のみを扱う設定となっています。
int	**wolfSSL_set_server_cert_type**(WOLFSSL * ssl, const char * buf, int len) この関数はサーバー側で呼び出される場合には、クライアント側にCertificate メッセージで送信できる証明書タイプを設定します。クライアント側で呼び出される場合には、受入れ可能なサーバー証明書タイプを設定します。Raw Public Key 証明書を送受信したい場合にはこの関数を使って証明書タイプを設定しなければなりません。設定する証明書タイプは優先度順に格納したバイト配列として渡します。設定するバッファアドレスに NULL を渡すか、あるいはバッファサイズに 0 を渡すと規定値にもどすことができます。規定値は X509 証明書（WOLFSSL_CERT_TYPE_X509）のみを扱う設定となっています。

	Name
int	**wolfSSL_get_negotiated_client_cert_type**(WOLFSSL * ssl, int * tp) この関数はハンドシェーク終了後に呼び出し、相手とのネゴシエーションの結果得られたクライアント証明書のタイプを返します。ネゴシエーションが発生しない場合には戻り値として WOLFSSL_SUCCESS が返されますが、証明書タイプとしては WOLFSSL_CERT_TYPE_UNKNOWN が返されます。
int	**wolfSSL_get_negotiated_server_cert_type**(WOLFSSL * ssl, int * tp) この関数はハンドシェーク終了後に呼び出し、相手とのネゴシエーションの結果得られたサーバー証明書のタイプを返します。ネゴシエーションが発生しない場合には戻り値として WOLFSSL_SUCCESS が返されますが、証明書タイプとしては WOLFSSL_CERT_TYPE_UNKNOWN が返されます。
int	**wolfSSL_CTX_set_ephemeral_key**(WOLFSSL_CTX * ctx, int keyAlgo, const char * key, unsigned int keySz, int format) この関数はテストのための固定/静的なエフェラルキーを設定します。
int	**wolfSSL_set_ephemeral_key**(WOLFSSL * ssl, int keyAlgo, const char * key, unsigned int keySz, int format) この関数はテストのための固定/静的なエフェラルキーを設定します。
int	**wolfSSL_CTX_get_ephemeral_key**(WOLFSSL_CTX * ctx, int keyAlgo, const unsigned char ** key, unsigned int * keySz) この関数は ASN.1/DER としてロードされたキーへのポインタを返します
int	**wolfSSL_get_ephemeral_key**(WOLFSSL * ssl, int keyAlgo, const unsigned char ** key, unsigned int * keySz) この関数は ASN.1/DER としてロードされた鍵へのポインタを返します
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) 選択したメッセージダイジェスト、パディング、および RSA キーを使用してメッセージに署名します。
int	**wolfSSL_dtls13_has_pending_msg**(WOLFSSL * ssl)DTLSv1.3 送信済みだがまだ相手からアクノリッジを受けとっていないメッセージがあるか調べます。
unsigned int	**wolfSSL_SESSION_get_max_early_data**(const WOLFSSL_SESSION * s) アーリーデータの最大サイズを取得します。
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_dtls_cid_use**(WOLFSSL * ssl) コネクション ID 拡張を有効にします。RFC9146 と RFC9147 を参照してください。

	Name
int	**wolfSSL_dtls_cid_is_enabled**(WOLFSSL * ssl) この関数はハンドシェークが完了した後に呼び出されると、コネクション ID がネゴシエートされたかどうか確認することができます。RFC9146 と RFC9147 を参照してください。
int	**wolfSSL_dtls_cid_set**(WOLFSSL * ssl, unsigned char * cid, unsigned int size) このコネクションで他のピアに対してレコードを送信するためのコネクション ID をセットします。RFC9146 と RFC9147 を参照してください。コネクション ID は最大値が DTLS_CID_MAX_SIZE でなければなりません。DTLS_CID_MAX_SIZE はビルド時に値を指定が可能ですが 255 バイトをこえることはできません。
int	**wolfSSL_dtls_cid_get_rx_size**(WOLFSSL * ssl, unsigned int * size) コネクション ID のサイズを取得します。RFC9146 と RFC9147 を参照してください。
int	**wolfSSL_dtls_cid_get_rx**(WOLFSSL * ssl, unsigned char * buffer, unsigned int bufferSz) コネクション ID を引数 buffer で指定されたバッファにコピーします。RFC9146 と RFC9147 を参照してください。バッファのサイズは引数 bufferSz で指定してください。
int	**wolfSSL_dtls_cid_get_tx_size**(WOLFSSL * ssl, unsigned int * size) コネクション ID のサイズを取得します。c サイズは引数 size 変数に格納されます。
int	**wolfSSL_dtls_cid_get_tx**(WOLFSSL * ssl, unsigned char * buffer, unsigned int bufferSz) コネクション ID を引き数 buffer で指定されるバッファにコピーします。RFC9146 と RFC9147 を参照してください。バッファのサイズは引き数 bufferSz で指定します。

C.52.2 Functions Documentation

C.52.2.1 function wolfDTLSv1_2_client_method_ex

```
WOLFSSL_METHOD * wolfDTLSv1_2_client_method_ex(
    void * heap
)
```

この関数は DTLS v1.2 クライアントメソッドを初期化します。

See:

- wolfSSL_Init
- wolfSSL_CTX_new

Return:

- 作成に成功した場合は、WOLFSSL_METHOD ポインタを返します。
- メモリ割り当てエラーまたはメソッドの作成の失敗の場合は NULL を返します。

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
)
```

この関数は、wolfSSLv23_client_method と同様に WOLFSSL_METHOD を返します（サーバー/クライアント）。

See:

- wolfSSL_new
- wolfSSL_free

Return:

- 作成に成功した場合は、WOLFSSL_METHOD ポインタを返します。
- メモリ割り当てエラーまたはメソッドの作成の失敗の場合は NULL を返します。

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
)
```

wolfSSLv3_server_method()関数は、アプリケーションがサーバーであることを示すために使用され、SSL3.0 プロトコルのみをサポートします。この関数は、wolfSSL_CTX_new() を使用して SSL/TLS コンテキストを作成するときに使用される新しい WOLFSSL_METHOD 構造体のメモリを割り当てて初期化します。

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:

- 成功した場合、新しく作成された WOLFSSL_METHOD 構造体へのポインタを返します。
- XMALLOC を呼び出すときにメモリ割り当てが失敗した場合、基礎となる Malloc() 実装の失敗値が返されます（通常は errno が ENOMEM に設定されます）。

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
)
```

wolfSSLv3_client_method()関数は、アプリケーションがクライアントであり、SSL 3.0 プロトコルのみをサポートすることを示すために使用されます。この関数は、wolfSSL_CTX_new() を使用して SSL/TLS コンテキストを作成するときに使用される新しい WOLFSSL_METHOD 構造体のメモリを割り当てて初期化します。

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:

- 成功した場合、新しく作成された WOLFSSL_METHOD 構造体へのポインタを返します。
- XMALLOC を呼び出すときにメモリ割り当てが失敗した場合、基礎となる Malloc() 実装の失敗値が返されます（通常は errno が ENOMEM に設定されます）。

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
)
```

wolfTLSv1_server_method()関数は、アプリケーションがサーバーであることを示すために使用され、TLS 1.0 プロトコルのみをサポートします。この関数は、wolfSSL_ctx_new() を使用して SSL/TLS コンテキストを作成するときに使用される新しい WOLFSSL_METHOD 構造体のメモリを割り当てて初期化します。

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:

- 成功した場合、新しく作成された WOLFSSL_METHOD 構造体へのポインタを返します。
- XMALLOC を呼び出すときにメモリ割り当てが失敗した場合、基礎となる Malloc() 実装の失敗値が返されます（通常は errno が ENOMEM に設定されます）。

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
)
```

wolftlsv1_client_method() 関数は、アプリケーションがクライアントであり、TLS 1.0 プロトコルのみをサポートすることを示すために使用されます。この関数は、wolfSSL_ctx_new() を使用して SSL/TLS コンテキストを作成するときに使用される新しい WOLFSSL_METHOD 構造体のメモリを割り当てて初期化します。

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:

- 成功した場合、新しく作成された WOLFSSL_METHOD 構造体へのポインタを返します。
- XMALLOC を呼び出すときにメモリ割り当てが失敗した場合、基礎となる Malloc() 実装の失敗値が返されます（通常は errno が ENOMEM に設定されます）。

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
)
```

wolfTLSv1_1_server_method()関数は、アプリケーションがサーバーであることを示すために使用され、TLS 1.1 プロトコルのみをサポートします。この関数は、wolfSSL_ctx_new() を使用して SSL/TLS コンテキストを作成するときに使用される新しい WOLFSSL_METHOD 構造体のメモリを割り当てて初期化します。

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:

- 成功した場合、新しく作成された WOLFSSL_METHOD 構造体へのポインタを返します。
- XMALLOC を呼び出すときにメモリ割り当てが失敗した場合、基礎となる Malloc() 実装の失敗値が返されます（通常は errno が ENOMEM に設定されます）。

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
)
```

wolfTLSv1_1_client_method()関数は、アプリケーションがクライアントであり、TLS 1.0 プロトコルのみをサポートすることを示すために使用されます。この関数は、wolfSSL_ctx_new() を使用して SSL/TLS コンテキストを作成するときに使用される新しい WOLFSSL_METHOD 構造体のメモリを割り当てて初期化します。

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:

- 成功した場合、新しく作成された WOLFSSL_METHOD 構造体へのポインタを返します。
- XMALLOC を呼び出すときにメモリ割り当てが失敗した場合、基礎となる Malloc() 実装の失敗値が返されます（通常は errno が ENOMEM に設定されます）。

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
)
```

wolfTLSv1_2_server_method()関数は、アプリケーションがサーバーであることを示すために使用され、TLS 1.2 プロトコルのみをサポートします。この関数は、wolfSSL_ctx_new() を使用して SSL/TLS コンテキストを作成するときに使用される新しい WOLFSSL_METHOD 構造体のメモリを割り当てて初期化します。

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:

- 成功した場合、新しく作成された WOLFSSL_METHOD 構造体へのポインタを返します。
- XMALLOC を呼び出すときにメモリ割り当てが失敗した場合、基礎となる Malloc() 実装の失敗値が返されます（通常は errno が ENOMEM に設定されます）。

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
)
```

wolfTLSv1_2_client_method()関数は、アプリケーションがクライアントであり、TLS 1.2 プロトコルのみをサポートすることを示すために使用されます。この関数は、wolfSSL_ctx_new() を使用して SSL/TLS コンテキストを作成するときに使用される新しい Wolfssl_method 構造体のメモリを割り当てて初期化します。

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:

- 成功した場合、新しく作成された WOLFSSL_METHOD 構造体へのポインタを返します。
- XMALLOC を呼び出すときにメモリ割り当てが失敗した場合、基礎となる Malloc() 実装の失敗値が返されます（通常は errno が ENOMEM に設定されます）。

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
)
```

wolfdtlsv1_client_method() 関数は、アプリケーションがクライアントであり、DTLS 1.0 プロトコルのみをサポートすることを示すために使用されます。この関数は、wolfSSL_ctx_new() を使用して SSL/TLS コンテキストを作成するときに使用される新しい WOLFSSL_METHOD 構造体のメモリを割り当てて初期化します。この関数は、WolfSSL が DTLS サポート（–enable-dtls、または WOLFSSL_DTLS を定義することによって）ビルドされている場合にのみ使用できます。

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:

- 成功した場合、新しく作成された WOLFSSL_METHOD 構造体へのポインタを返します。
- XMALLOC を呼び出すときにメモリ割り当てが失敗した場合、基礎となる Malloc() 実装の失敗値が返されます（通常は errno が ENOMEM に設定されます）。

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
)
```

wolfDTLSv1_server_method()関数は、アプリケーションがサーバーであることを示すために使用され、DTLS 1.0 プロトコルのみをサポートします。この関数は、wolfSSL_ctx_new() を使用して SSL/TLS コンテキストを作成するときに使用される新しい WOLFSSL_METHOD 構造体のメモリを割り当てて初期化します。この関数は、WolfSSL が DTLS サポート（–enable-dtls、または WOLFSSL_DTLS マクロを定義することによって）ビルドされている場合にのみ使用できます。

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:

- 成功した場合、新しく作成された WOLFSSL_METHOD 構造体へのポインタを返します。
- XMALLOC を呼び出すときにメモリ割り当てが失敗した場合、基礎となる Malloc() 実装の失敗値が返されます（通常は errno が ENOMEM に設定されます）。

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_2_server_method

```
WOLFSSL_METHOD * wolfDTLSv1_2_server_method(
    void
)
```

wolfDTLSv1_2_server_method() 関数はサーバ側用に WOLFSSL_METHOD 構造体のメモリを割り当てて初期化します。

Parameters:

- **なし** *Example*

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(wolfDTLSv1_2_server_method());
WOLFSSL* ssl = WOLFSSL_new(ctx);
…
```

See:

- wolfSSL_CTX_new
- wolfSSL_CTX_new

Return:

- 成功した場合、新しく作成された WOLFSSL_METHOD 構造体へのポインタを返します。
- 成功した場合、新しく作成された WOLFSSL_METHOD 構造体へのポインタを返します。

この関数はサーバー側用に WOLFSSL_METHOD 構造体を生成して初期化します。

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(wolfDTLSv1_2_server_method());
WOLFSSL* ssl = WOLFSSL_new(ctx);
…
```

C.52.2.14 function wolfDTLSv1_3_server_method

```
WOLFSSL_METHOD * wolfDTLSv1_3_server_method(
    void
)
```

wolfDTLSv1_3_server_method()関数はアプリケーションがサーバーであることを示すために使用され、DTLS 1.3 プロトコルのみをサポートします。この関数は、wolfSSL_ctx_new() を使用して SSL/TLS コンテキストを作成するときに使用される新しい WOLFSSL_METHOD 構造体のメモリを割り当てて初期化します。この関数は、WolfSSL が DTLS サポート（–enable-dtls13、または WOLFSSL_DTLS13 を定義することによって）ビルドされている場合にのみ使用できます。

Parameters:

- **なし** *Example*

```
WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfDTLSv1_3_server_method();
if (method == NULL) {
    // unable to get method
}

ctx = wolfSSL_CTX_new(method);
...
```

See: wolfDTLSv1_3_client_method

Return:

- 成功した場合、新しく作成された WOLFSSL_METHOD 構造体へのポインタを返します。
- XMALLOC を呼び出すときにメモリ割り当てが失敗した場合、基礎となる Malloc() 実装の失敗値が返されます（通常は errno が ENOMEM に設定されます）。

C.52.2.15 function wolfDTLSv1_3_client_method

```
WOLFSSL_METHOD * wolfDTLSv1_3_client_method(
    void
)
```

wolfDTLSv1_3_client_method()関数はアプリケーションがクライアントであることを示すために使用され、DTLS 1.3 プロトコルのみをサポートします。この関数は、wolfSSL_ctx_new() を使用して SSL/TLS コンテキストを作成するときに使用される新しい WOLFSSL_METHOD 構造体のメモリを割り当てて初期化します。この関数は、WolfSSL が DTLS サポート（–enable-dtls13、または WOLFSSL_DTLS13 を定義することによって）ビルドされている場合にのみ使用できます。

Parameters:

- **なし**

See: wolfDTLSv1_3_server_method

Return:

- 成功した場合、新しく作成された WOLFSSL_METHOD 構造体へのポインタを返します。
- XMALLOC を呼び出すときにメモリ割り当てが失敗した場合、基礎となる Malloc() 実装の失敗値が返されます（通常は errno が ENOMEM に設定されます）。

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.52.2.16 function wolfDTLS_server_method

```
WOLFSSL_METHOD * wolfDTLS_server_method(
    void
)
```

wolfDTLS_server_method()関数はアプリケーションがサーバーであることを示すために使用され、可能な限り高いバージョン最小バージョンの DTLS プロトコルをサポートします。デフォルトの最小バージョンは WOLFSSL_MIN_DTLS_DOWNGRADE マクロでの指定をもとにしていて、実行時に wolfSSL_SetMinVersion() で変更することができます。この関数は、wolfSSL_ctx_new() を使用して SSL/TLS コンテキストを作成するときに使用される新しい WOLFSSL_METHOD 構造体のメモリを割り当てて初期化します。この関数は、WolfSSL が DTLS サポート（–enable-dtls、または WOLFSSL_DTLS を定義することによって）ビルドされている場合にのみ使用できます。

Parameters:

- **なし** *Example*

```
WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfDTLS_server_method();
if (method == NULL) {
    // unable to get method
}

ctx = wolfSSL_CTX_new(method);
...
```

See:

- wolfDTLS_client_method
- wolfSSL_SetMinVersion

Return:

- 成功した場合、新しく作成された WOLFSSL_METHOD 構造体へのポインタを返します。
- XMALLOC を呼び出すときにメモリ割り当てが失敗した場合、基礎となる Malloc() 実装の失敗値が返されます（通常は errno が ENOMEM に設定されます）。

C.52.2.17 function wolfDTLS_client_method

```
WOLFSSL_METHOD * wolfDTLS_client_method(
    void
)
```

wolfDTLS_client_method()関数は アプリケーションがクライアントであることを示すために使用され、可能な限り高いバージョン最小バージョンの DTLS プロトコルをサポートします。デフォルトの最小

バージョンは WOLFSSL_MIN_DTLS_DOWNGRADE マクロでの指定をもとにしていて、実行時に wolfSSL_SetMinVersion() で変更することができます。この関数は、wolfSSL_ctx_new() を使用して SSL/TLS コンテキストを作成するときに使用される新しい WOLFSSL_METHOD 構造体のメモリを割り当てて初期化します。この関数は、wolfSSL が DTLS サポート（–enable-dtls、または WOLFSSL_DTLS を定義することによって）ビルドされている場合にのみ使用できます。

Parameters:

- **なし**

See:

- wolfDTLS_server_method
- wolfSSL_SetMinVersion

Return:

- 成功した場合、新しく作成された WOLFSSL_METHOD 構造体へのポインタを返します。
- XMALLOC を呼び出すときにメモリ割り当てが失敗した場合、基礎となる Malloc() 実装の失敗値が返されます（通常は errno が ENOMEM に設定されます）。

Example

```
WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfDTLS_client_method();
if (method == NULL) {
    // unable to get method
}

ctx = wolfSSL_CTX_new(method);
...
```

C.52.2.18 function wolfSSL_use_old_poly

```
int wolfSSL_use_old_poly(
    WOLFSSL * ssl,
    int value
)
```

Chacha-Poly Aead Construction の最初のリリースと新しいバージョンの間にいくつかの違いがあるため、古いバージョンを使用してサーバー/クライアントと通信するオプションを追加しました。デフォルトでは、wolfSSL は新しいバージョンを使用します。

Parameters:

- **ssl** wolfSSL_new()を使用して作成した WOLFSSL 構造体へのポインタ。

See: none

Return: 0 成功の場合に返されます。

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
)
```

wolfSSL_dtls_import() 関数はシリアライズされたセッション状態を解析するために使われます。これにより、ハンドシェイクが完了した後に接続をピックアップすることができます。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ。
- **buf** インポートするシリアル化されたセッション情報を格納するバッファへのポインタ。
- **sz** バッファのサイズ

See:

- wolfSSL_new
- wolfSSL_CTX_new
- wolfSSL_CTX_dtls_set_export

Return:

- 成功した場合、読み取ったバッファの量が返されます。
- すべての失敗した戻り値は 0 未満になります。
- VERSION_ERROR バージョンの不一致が見つかった場合、(すなわち、DTLS v1 と CTX が DTLS v1.2 に設定された場合)、Version_Error が返されます。

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
)
```

シリアライズされた TLS セッションをインポートします。警告：buf には、状態に関する機密情報が含まれており、保存されている場合は保存する前に暗号化されるのが最善です。追加のデバッグ情報をマクロ WOLFSSL_SESSION_EXPORT_DEBUG を定義して表示できます。

Parameters:

- **ssl** セッションをインポートするための WOLFSSL 構造体へのポインタ
- **buf** シリアル化されたセッションを含むバッファへのポインタ
- **sz** バッファのサイズ

See:

- wolfSSL_dtls_import
- wolfSSL_tls_export

Return: バッファ'buf' から読み込まれたバイト数を返します。

C.52.2.21 function wolfSSL_CTX_dtls_set_export

```
int wolfSSL_CTX_dtls_set_export(
    WOLFSSL_CTX * ctx,
    wc_dtls_export func
)
```

wolfSSL_CTX_dtls_set_export() 関数はセッションをエクスポートするためのコールバック関数を設定します。以前に格納されているエクスポート機能をクリアするためにパラメータ func に NULL を渡すことが許されます。サーバー側で使用され、ハンドシェイクが完了した直後に設定したコールバック関数が呼び出されます。

Parameters:

- **ctx** wolfSSL_CTX_new()で作成された WOLFSSL_CTX 構造体へのポインタ。
- **func** セッションをエクスポートする際に呼び出す関数ポインタ

See:

- wolfSSL_new
- wolfSSL_CTX_new
- wolfSSL_dtls_set_export
- Static buffer use

Return:

- SSL_SUCCESS 成功時に返されます。
- BAD_FUNC_ARG NULL または予想されない引数が渡された場合に返されます。

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
)
```

wolfSSL_dtls_set_export() 関数はセッションをエクスポートする際に呼び出すコールバック関数を登録します。以前に登録されているエクスポート関数をクリアするために使うこともできます。サーバー側で使用され、ハンドシェイクが完了した直後に設定したコールバック関数が呼び出されます。

Parameters:

- **ssl** wolfssl_new() を使用して作成された WOLFSSL 構造体へのポインタ。
- **func** セッションをエクスポートする際に呼び出す関数ポインタ

See:

- wolfSSL_new
- wolfSSL_CTX_new
- wolfSSL_CTX_dtls_set_export

Return:

- SSL_SUCCESS 成功時に返されます。
- BAD_FUNC_ARG NULL または予想されない引数が渡された場合に返されます。

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
)
```

wolfSSL_dtls_export() 関数は提供されたバッファへセッションをシリアル化します。セッションをエクスポートするための関数コールバックを使用するよりもメモリオーバーヘッドを減らすことができます。関数に渡された引数 buf が NULL の場合、sz には WolfSSL セッションのシリアライズに必要なバッファのサイズが設定されます。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ。
- **buf** シリアライズしたセッションを保持するためのバッファ。
- **sz** バッファのサイズ

See:

- wolfSSL_new
- wolfSSL_CTX_new
- wolfSSL_CTX_dtls_set_export
- wolfSSL_dtls_import

Return:

- 成功した場合、使用されるバッファサイズが返されます。
- すべての失敗した戻り値は 0 未満になります。

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
)
```

シリアライズされた TLS セッションをエクスポートします。ほとんどの場合、wolfSSL_tls_export の代わりに wolfssl_get1_session を使用する必要があります。追加のデバッグ情報をマクロ WOLFSSL_SESSION_EXPORT_DEBUG を定義して表示できます。警告：buf には、状態に関する機密情報が含まれており、保存する場合は保存する前に暗号化されるのが最善です。

Parameters:

- **ssl** セッションをエクスポートするための WOLFSSL 構造体へのポインタ
- **buf** シリアライズされたセッションの出力先バッファへのポインタ
- **sz** 出力先バッファのサイズ

See:

- wolfSSL_dtls_import
- wolfSSL_tls_import

Return: バッファ'buf' に書き込まれたバイト数

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
)
```

この関数は CTX 用に静的メモリ領域を設定する目的に使用されます。設定された静的メモリ領域は CTX の有効期間および CTX から作成された全ての SSL オブジェクトに使用されます。引数 ctx に NULL を渡し、wolfSSL_method_func 関数を渡すことによって、CTX 自体の作成も静的メモリを使用します。wolfssl_method_func は次のシグネチャとなっています:wolfssl_method （ wolfssl_method_func)(void *heap)。引数 max に 0 を渡すと、設定されていないものとして動作し、最大の同時使用制限が適用されません。引数 flag に渡した値によって、メモリの使用方法と動作が決まります。利用可能なフラグ値は次のとおりです：0 - デフォルトの一般メモリ、WOLFMEM_IO_POOL - メッセージの受送信の際の入出力バッファとして使用され渡されたバッファ内のすべてのメモリが IO に使用されます、WOLFMEM_IO_FIXED - WOLFMEM_IO_POOL と同じですが、各 SSL は 2 つのバッファを自分のライフタイムの間保持して使用します。WOLFMEM_TRACK_STATS - 各 SSL は実行中にメモリ使用統計を追跡します。

Parameters:

- **ctx** WOLFSSL_CTX 構造体へのポインタのポインタ
- **method** メソッド関数（例えば、wolfSSLv23_server_method_ex）で ctx が NULL でない場合は NULL にする必要があります。
- **buf** すべての操作に使用するメモリバッファへのポインタ。
- **sz** 渡されているメモリバッファのサイズ。
- **flag** メモリの使用タイプ
- **max** 同時使用の最大値

See:

- wolfSSL_CTX_new
- wolfSSL_CTX_is_static_memory
- wolfSSL_is_static_memory

Return:

- SSL_SUCCESS 成功した場合に返されます。に返されます。
- SSL_FAILURE 失敗した場合に返されます。

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
)
```

この関数は現時点の接続に関する振る舞いの変更は行いません。静的メモリ使用量に関する情報を収集するためにのみ使用されます。

Parameters:

- **ctx** wolfSSL_CTX_new()を使用して作成された WOLFSSL_CTX 構造体へのポインタ。
- **mem_stats** 静的メモリの使用量に関する情報を保持する WOLFSSL_MEM_STATS 構造体へのポインタ

See:

- wolfSSL_CTX_new
- wolfSSL_CTX_load_static_memory
- wolfSSL_is_static_memory

Return:

- 1 CTX の静的メモリを使用している場合に返されます。
- 0 静的メモリを使用していない場合に返されます。

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 関数は SSL の静的メモリ使用量に関する情報を集めます。戻り値は、静的メモリが使用されているかどうかを示します。引数 ssl の上位の WOLFSSL_CTX に静的メモリを使用するように指定してあり、WOLFMEM_TRACK_STATS が定義されている場合に 引数 mem_stats に情報がセットされます。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ。
- **mem_stats** 静的メモリの使用量に関する情報を保持する WOLFSSL_MEM_STATS 構造体へのポインタ

See:

- wolfSSL_new
- wolfSSL_CTX_is_static_memory

Return:

- 1 静的メモリを使用している場合に返されます。
- 0 静的メモリを使用していない場合に返されます。

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
)
```

この関数は証明書ファイルを SSL コンテキスト (WOLFSSL_CTX) にロードします。ファイルは引数 file によって提供されます。引数 format は、ファイルのフォーマットタイプ（SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM）を指定します。適切な使用法の例をご覧ください。

Parameters:

- **ctx** wolfSSL_CTX_new()を使用して作成された WOLFSSL_CTX 構造体へのポインタ
- **file** ロードする証明書を含むファイルパス文字列。
- **format** ロードする証明書のフォーマット：SSL_FILETYPE_ASN1 あるいは SSL_FILETYPE_PEM

See:

- wolfSSL_CTX_use_certificate_buffer
- wolfSSL_use_certificate_file
- wolfSSL_use_certificate_buffer

Return:

- SSL_SUCCESS 成功した場合に返されます。に返されます。
- SSL_FAILURE 失敗時に返されます。失敗した場合の可能な原因としては、ファイルが誤った形式の場合、または引数 format を使用して誤ったフォーマットが指定されている、あるいはファイルが存在しない、あるいは読み取ることができない、または破損している、メモリ不足が発生、Base16 のデコードに失敗しているなどの原因が考えられます。

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
)
```

この関数は、秘密鍵ファイルを SSL コンテキスト (WOLFSSL_CTX) にロードします。ファイルは引数 file によって提供されます。引数 format は、次のファイルのフォーマットタイプを指定します：SSL_FILETYPE_ASN1 あるいは SSL_FILETYPE_PEM。適切な使用法の例をご覧ください。外部キーストアを使用し、秘密鍵を持っていない場合は、代わりに公開鍵を入力して crypto コールバックを登録して署名を処理することができます。このためには、crypto コールバックまたは PK コールバックを使用したコンフィギュレーションでビルドします。crypto コールバックを有効にするには、–enable-cryptocb または WOLF_CRYPTO_CB マクロを使用し、wc_CryptoCb_RegisterDevice を使用して暗号コールバックを登録し、wolfSSL_CTX_SetDevId を使用して関連する devid を設定します。

Parameters:

- **なし** *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
}
...
```

See:

- wolfSSL_CTX_use_PrivateKey_buffer
- wolfSSL_use_PrivateKey_file
- wolfSSL_use_PrivateKey_buffer
- wc_CryptoCb_RegisterDevice
- wolfSSL_CTX_SetDevId

Return:

- SSL_SUCCESS 成功した場合に返されます。に返されます。
- SSL_FAILURE 関数呼び出しが失敗した場合の可能な原因としては、ファイルが誤った形式の場合、または引数 format を使用して誤ったフォーマットが指定されている、あるいはファイルが存在しない、あるいは読み取ることができない、または破損している、メモリ不足が発生、Base16 のデコードに失敗しているなどの原因が考えられます

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
)
```

この関数は、PEM 形式の CA 証明書ファイルを SSL コンテキスト (WOLFSSL_CTX) にロードします。これらの証明書は、信頼できるルート証明書として扱われ、SSL ハンドシェイク中にピアから受信した証明書を検証するために使用されます。引数 file によって提供されるルート証明書ファイルは、単一の証明書または複数の証明書を含むファイルでの場合があります。複数の CA 証明書が同じファイルに含まれている場合、wolfSSL はファイルに表示されているのと同じ順序でそれらをロードします。引数 path は、信頼できるルート CA の証明書を含むディレクトリの名前へのポインタです。引数 file が NULL ではない場合、パスが必要でない場合は NULL として指定できます。引数 path が指定されていてかつ NO_WOLFSSL_DIR が定義されていない場合には、wolfSSL ライブラリは指定されたディレクトリに存在するすべての CA 証明書をロードします。この関数はディレクトリ内のすべてのファイルをロードしようとします。この関数は、ヘッダーに “—–BEGIN CERTIFICATE—–” を持つ PEM フォーマットされた CERT_TYPE ファイルを期待しています。

Parameters:

- **ctx** wolfSSL_CTX_new()で作成された SSL コンテキストへのポインタ。
- **file** PEM 形式の CA 証明書を含むファイルの名前へのポインタ。
- **path** CA 証明書を含んでいるディレクトリのディレクトリの名前へのポインタ。

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 成功した場合に返されます。に返されます。
- SSL_FAILURE CTX が NULL の場合、またはファイルとパスの両方が NULL の場合に返されます。
- SSL_BAD_FILETYPE ファイルが間違った形式である場合に返されます。
- SSL_BAD_FILE ファイルが存在しない場合、読み込めない場合、または破損している場合に返されます。
- MEMORY_E メモリ不足状態が発生した場合に返されます。
- ASN_INPUT_E base16 デコードがファイルに対して失敗した場合に返されます。
- ASN_BEFORE_DATE_E 現在の日付が使用開始日より前の場合に返されます。
- ASN_AFTER_DATE_E 現在の日付が使用期限後より後の場合に返されます。
- BUFFER_E チェーンバッファが受信バッファよりも大きい場合に返されます。
- BAD_PATH_ERROR opendir() がパスを開こうとして失敗した場合に返されます。

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
)
```

この関数は、PEM 形式の CA 証明書ファイルを SSL コンテキスト (WOLFSSL_CTX) にロードします。これらの証明書は、信頼できるルート証明書として扱われ、SSL ハンドシェイク中にピアから受信した証明書を検証するために使用されます。引数 file によって提供されるルート証明書ファイルは、単一の証明書または複数の証明書を含むファイルでの場合があります。複数の CA 証明書が同じファイルに含まれている場合、wolfSSL はファイルに表示されているのと同じ順序でそれらをロードします。引数 path は、信頼できるルート CA の証明書を含むディレクトリの名前へのポインタです。引数 file が NULL ではない場合、パスが必要でない場合は NULL として指定できます。引数 path が指定されていてかつ NO_WOLFSSL_DIR が定義されていない場合には、wolfSSL ライブラリは指定されたディレクトリに存在するすべての CA 証明書をロードします。この関数は引数 flags に基づいてディレクトリ内のすべてのファイルをロードしようとします。この関数は、ヘッダーに “—–BEGIN CERTIFICATE—–” を持つ PEM フォーマットされた CERT_TYPE ファイルを期待しています。

Parameters:

- **ctx** wolfSSL_CTX_new()で作成された SSL コンテキストへのポインタ。
- **file** PEM 形式の CA 証明書を含むファイルの名前へのポインタ。
- **path** CA 証明書を含んでいるディレクトリのフォルダーパス
- **flags** 指定可能なマスク値: WOLFSSL_LOAD_FLAG_IGNORE_ERR, WOLFSSL_LOAD_FLAG_DATE_ERR_OKAY, 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 成功した場合に返されます。に返されます。
- SSL_FAILURE CTX が NULL の場合、またはファイルとパスの両方が NULL の場合に返されます。
- SSL_BAD_FILETYPE ファイルが間違った形式である場合に返されます。
- SSL_BAD_FILE ファイルが存在しない場合、読み込めない場合、または破損している場合に返されます。
- MEMORY_E メモリ不足状態が発生した場合に返されます。
- ASN_INPUT_E base16 デコードがファイルに対して失敗した場合に返されます。
- ASN_BEFORE_DATE_E 現在の日付が使用開始日より前の場合に返されます。
- ASN_AFTER_DATE_E 現在の日付が使用期限後より後の場合に返されます。
- BUFFER_E チェーンバッファが受信バッファよりも大きい場合に返されます。
- BAD_PATH_ERROR opendir() がパスを開こうとして失敗した場合に返されます。

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
)
```

この関数は、wolfSSL_CTX_load_system_CA_certs が呼び出されたときに、wolfSSL がシステム CA 証明書を検索するディレクトリを表す文字列の配列へのポインタを返します。

Parameters:

- **num** word32 型変数へのポインタ。文字列配列の長さを格納します。

See:

- wolfSSL_CTX_load_system_CA_certs
- wolfSSL_CTX_load_verify_locations
- wolfSSL_CTX_load_verify_locations_ex

Return:

- 成功時には文字列配列へのポインタを返します。
- NULL 失敗時に返します。

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
)
```

この関数は、CA 証明書を OS 依存の CA 証明書ストアから WOLFSSL_CTX にロードしようとします。ロードされた証明書は信頼されます。サポートおよびテストされているプラットフォームは、Linux(Debian、Ubuntu、Gentoo、Fedora、RHEL)、Windows 10/11、Android、Apple OS X、iOS です。

Parameters:

- **ctx** wolfSSL_CTX_new()で生成された WOLFSSL_CTX 構造体へのポインタ。

See:

- wolfSSL_get_system_CA_dirs
- wolfSSL_CTX_load_verify_locations
- wolfSSL_CTX_load_verify_locations_ex

Return:

- WOLFSSL_SUCCESS 成功時に返されます。
- WOLFSSL_BAD_PATH システム CA 証明書がロードできなかった場合に返されます。
- WOLFSSL_FAILURE そのほかのエラー発生時（Windows 証明書ストアが正常にクローズされない等）

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
)
```

この関数は、TLS/SSL ハンドシェイクを実行するときにピアを検証するために使用する証明書をロードします。ハンドシェイク中に送信されたピア証明書は、この関数で指定された証明書の SKID と署名を比較することによって検証されます。これら 2 つのことが一致しない場合は、ピア証明書の検証にはロードされた CA 証明書が使用されます。この機能は WOLFSSL_TRUST_PEER_CERT マクロを定義することで機能を有効にできます。適切な使用法は例をご覧ください。

Parameters:

- **ctx** wolfSSL_CTX_new()で生成された WOLFSSL_CTX 構造体へのポインタ。
- **file** 証明書を含むファイルの名前へのポインタ

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 成功時に返されます。
- SSL_FAILURE CTX が NULL の場合、または両方のファイルと種類が無効な場合に返されます。
- SSL_BAD_FILETYPE ファイルが間違った形式である場合に返されます。
- SSL_BAD_FILE ファイルが存在しない場合に返されます。読み込め、または破損していません。
- MEMORY_E メモリ不足状態が発生した場合に返されます。
- ASN_INPUT_E base16 デコードがファイルに対して失敗した場合に返されます。

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
)
```

この関数は、証明書チェーンを SSL コンテキスト (WOLFSSL_CTX) にロードします。証明書チェーンを含むファイルは引数 file によって提供され、PEM 形式の証明書を含める必要があります。この関数は、最大 MAX_CHAIN_DEPTH（既定で 9、internal.h で定義されている）数の証明書を処理します。この数にはサブジェクト証明書を含みます。

Parameters:

- **ctx** wolfSSL_CTX_new()で生成された WOLFSSL_CTX 構造体へのポインタ。

See:

- wolfSSL_CTX_use_certificate_file
- wolfSSL_CTX_use_certificate_buffer
- wolfSSL_use_certificate_file
- wolfSSL_use_certificate_buffer

Return:

- SSL_SUCCESS 成功時に返されます。
- SSL_FAILURE 関数呼び出しが失敗した場合、可能な原因としては：誤った形式である場合、または「format」引数を使用して誤ったフォーマットが指定されている場合、ファイルが存在しない、読み取れない、または破損している、メモリ枯渇などが考えられます。

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
)
```

この関数は、SSL 接続で使用されている RSA 秘密鍵を SSL コンテキスト (WOLFSSL_CTX) にロードします。この関数は、wolfSSL が OpenSSL 互換 API が有効（–enable-openSSLExtra、#define OPENSSL_EXTRA）でコンパイルされている場合にのみ利用可能で、より一般的に使用されている wolfSSL_CTX_use_PrivateKey_file() 関数と同じです。ファイル引数には、RSA 秘密鍵ファイルへのポインタが、引数 format で指定された形式で含まれています。

Parameters:

- **ctx** wolfSSL_CTX_new()を使用して作成された WOLFSSL_CTX 構造体へのポインタ
- **file** フォーマットで指定された形式で、WolfSSL SSL コンテキストにロードされる RSA 秘密鍵を含むファイルの名前へのポインタ。
- **format** RSA 秘密鍵のエンコード形式を指定します。指定可能なフォーマット値は：SSL_FILETYPE_PEM と 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 成功時に返されます。
- SSL_FAILURE 関数呼び出しが失敗した場合に返されます。失敗の原因には次が考えられます：入力鍵ファイルが誤った形式である、または引数 format を使用して誤った形式が与えられている場合、ファイルが存在しない、読み込めない、または破損してる、メモリ不足状態が発生。

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
)
```

この関数は、有効なセッション（NULL 以外の引数 ssl）が指定された場合に、デフォルトで 9 の最大チェーン深度を返します。

See: wolfSSL_CTX_get_verify_depth

Return:

- MAX_CHAIN_DEPTH WOLFSSL 構造体が NULL ではない場合に返されます。デフォルトでは値は 9 です。
- BAD_FUNC_ARG WOLFSSL 構造体が NULL の場合に返されます。

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
)
```

この関数は、WOLFSSL_CTX 構造体構造を使用して証明書チェーン深度を取得します。

See:

- wolfSSL_CTX_use_certificate_chain_file
- wolfSSL_get_verify_depth

Return:

- MAX_CHAIN_DEPTH WOLFSSL_CTX 構造体が NULL ではない場合に返されます。最大証明書チェーンピア深度の定数表現。
- BAD_FUNC_ARG WOLFSSL_CTX 構造体が NULL の場合に返されます。

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
)
```

この関数は証明書ファイルを SSL セッション（WOLFSSL 構造体）にロードします。証明書ファイルはファイル引数によって提供されます。引数 format は、ファイルのフォーマットタイプ（SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM）を指定します。

Parameters:

- **ssl** wolfSSL_new()で作成された WOLFSSL 構造体へのポインタ。

- **file** WOLFSSL 構造体にロードされる証明書を含むファイルの名前へのポインタ
- **format** 証明書ファイルのエンコード形式を指定します。指定可能なフォーマット値は：SSL_FILETYPE_PEM と SSL_FILETYPE_ASN1

See:

- wolfSSL_CTX_use_certificate_buffer
- wolfSSL_CTX_use_certificate_file
- wolfSSL_use_certificate_buffer

Return:

- SSL_SUCCESS 成功時に返されます。
- SSL_FAILURE 関数呼び出しが失敗した場合に返されます。可能な原因には次のようなものがあります。ファイルが誤った形式、または引数 format を使用して誤った形式が与えられた、メモリ不足状態が発生した、ファイルで Base16 のデコードが失敗した

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
)
```

この関数は、秘密鍵ファイルを SSL セッション（WOLFSSL 構造体）にロードします。鍵ファイルは引数 file によって提供されます。引数 format は、ファイルのタイプ（SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM が指定可）を指定します。外部キーストアを使用し、秘密鍵を持っていない場合は、代わりに公開鍵を入力して CryPro コールバックを登録して署名を処理することができます。このためには、Crypto コールバックまたは PK コールバックを使用したコンフィグレーションでビルドします。Crypto コールバックを有効にするには、–enable-cryptocb または WOLF_CRYPTO_CB マクロを使用してビルドし、wc_CryptoCb_RegisterDevice を使用して暗号コールバックを登録し、wolfSSL_SetDevId を使用して関連する devId を設定します。

Parameters:

- **ssl** wolfSSL_new()で作成された WOLFSSL 構造体へのポインタ。
- **file** WOLFSSL 構造体にロードされる証明書を含むファイルの名前へのポインタ
- **format** 秘密鍵ファイルのエンコード形式を指定します。指定可能なフォーマット値は：SSL_FILETYPE_PEM と 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 成功時に返されます。
- SSL_FAILURE 関数呼び出しが失敗した場合に返されます。可能な原因には次のようなものがあります。ファイルが誤った形式、または引数 format を使用して誤った形式が与えられた、メモリ不足状態が発生した、ファイルで Base16 のデコードが失敗した

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
)
```

この関数は、証明書チェーンを SSL セッション WOLFSSL 構造体）にロードします。証明書チェーンを含むファイルは引数 file によって提供され、PEM 形式の証明書を含める必要があります。この関数は、MAX_CHAIN_DEPTH（既定で 9、internal.h で定義されている）証明書に加えて、サブジェクト証明書を処理します。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ
- **file** WOLFSSL 構造体にロードされる証明書を含むファイルの名前へのポインタ。証明書は PEM 形式でなければなりません。

See:

- wolfSSL_CTX_use_certificate_chain_file
- wolfSSL_CTX_use_certificate_chain_buffer
- wolfSSL_use_certificate_chain_buffer

Return:

- SSL_SUCCESS 成功時に返されます。
- SSL_FAILURE 関数呼び出しが失敗した場合に返されます。可能な原因には次のようなものがあります：ファイルが誤った形式、または引数 format を使用して誤った形式が与えられた、メモリ不足状態が発生した、ファイルで base16 のデコードが失敗した

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
)
```

この関数は、SSL 接続で使用されている RSA 秘密鍵を SSL セッション（WOLFSSL 構造体）にロードします。この関数は、wolfSSL が OpenSSL 互換 API を有効（–enable-openSSlExtra、#define OPENSSL_EXTRA）でビルドされている場合にのみ利用可能で、より一般的に使用される wolfSSL_use_PrivateKey_file() 関数と同じです。引数 file には、RSA 秘密鍵ファイルへのポインタが、フォーマットで指定された形式で含まれています。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ

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 成功時に返されます。
- SSL_FAILURE 関数呼び出しが失敗した場合に返されます。可能な原因には次のようなものがあります：ファイルが誤った形式、または引数 format を使用して誤った形式が与えられた、メモリ不足状態が発生した、ファイルで Base16 のデコードが失敗した

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
)
```

この関数は wolfSSL_CTX_load_verify_locations と似ていますが、DER フォーマットされた CA ファイルを SSL コンテキスト（WOLFSSL_CTX）にロードすることを許可します。それはまだ PEM 形式の CA ファイルをロードするためにも使用されるかもしれません。これらの証明書は、信頼できるルート証明書として扱われ、SSL ハンドシェイク中にピアから受信した証明書を検証するために使用されます。ファイル引数によって提供されるルート証明書ファイルは、単一の証明書または複数の証明書を含むファイルでも可能。複数の CA 証明書が同じファイルに含まれている場合、wolfSSL はファイルに表示されているのと同じ順序でそれらをロードします。引数 format は、証明書が SSL_FILETYPE_PEM または SSL_FILETYPE_ASN1

(DER) のいずれかにある形式を指定します。wolfSSL_CTX_load_verify_locations とは異なり、この関数は特定のディレクトリパスからの CA 証明書のロードを許可しません。この関数は、wolfSSL ライブラリが WOLFSSL_DER_LOAD マクロが定義された状態でビルドされたときにのみ利用可能です。

Parameters:

- **ctx** wolfSSL_CTX_new()を使用して作成された WOLFSSL_CTX 構造体へのポインタ
- **file** wolfssl SSL コンテキストにロードされる CA 証明書を含むファイルの名前をフォーマットで指定された形式で指定します。

See:

- wolfSSL_CTX_load_verify_locations
- wolfSSL_CTX_load_verify_buffer

Return:

- SSL_SUCCESS 成功時に返されます。
- SSL_FAILURE 失敗すると返されます。

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 *
)
```

この関数は、所望の SSL/TLS プロトコル用メソッド構造体を引数に取って、新しい SSL コンテキストを作成します。

See: wolfSSL_new

Return:

- pointer 成功した場合、新しく作成された WOLFSSL_CTX 構造体へのポインタを返します。
- NULL 失敗時に返されます。

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 *
)
```

この関数はすでに作成された SSL コンテキスト (WOLFSSL_CTX) を入力として、新しい SSL セッション (WOLFSSL) を作成します。

See: wolfSSL_CTX_new

Return:

- 成功した場合、新しく作成された WOLFSSL 構造体へのポインタを返します。
- NULL 失敗時に返されます。

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
)
```

この関数は、SSL 接続の入出力機能としてファイル記述子 (fd) を割り当てます。通常これはソケットファイル記述子になります。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ
- **fd** SSL/TLS 接続に使用するファイルディスクリプタ

See:

- wolfSSL_CTX_SetIOSend
- wolfSSL_CTX_SetIORecv
- wolfSSL_SetIOReadCtx
- wolfSSL_SetIOWriteCtx

Return:

- SSL_SUCCESS 成功時に返されます。

- BAD_FUNC_ARG 失敗時に返されます。

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
)
```

この関数はファイルディスクリプタ (fd) を SSL コネクションの入出力手段として設定します。通常はソケットファイルディスクリプタが指定されます。この関数は DTLS 専用の API であり、ソケットは接続済みとマークされます。したがって、与えられた fd に対する recvfrom と sendto 呼び出しでの addr と addr_len は NULL に設定されます。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ
- **fd** SSL/TLS コネクションに使用するファイルディスクリプタ。

See:

- wolfSSL_CTX_SetIOSend
- wolfSSL_CTX_SetIORecv
- wolfSSL_SetIOReadCtx
- wolfSSL_SetIOWriteCtx
- wolfDTLS_SetChGoodCb

Return:

- SSL_SUCCESS 成功時に返されます。
- BAD_FUNC_ARG 失敗時に返されます。

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
)
```

この関数は DTLS ClientHello メッセージが正しく処理できた際に呼び出されるコールバック関数を設定します。クッキー交換メカニズムを使用する場合 (DTLS1.2 の HelloVerifyRequest か DTLS1.3 のクッキー拡張を伴った HelloRetryRequest のいずれかを使用する場合) には、クッキー交換が成功した時点でこのコールバック関数が呼び出されます。この機能はひとつの WOLFSSL オブジェクトを新たな接続を待ち受けるリスナーとして使い, ClientHello が検証された WOLFSSL オブジェクトから絶縁させることができます。この場合の検証はクッキー交換か ClientHello が正しいフォーマットになっているかのチェックによってなされます。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ
- **fd** SSL/TLS コネクションに使用するファイルディスクリプタ。

See: wolfSSL_set_dtls_fd_connected

Return:

- SSL_SUCCESS 成功時に返されます。
- BAD_FUNC_ARG 失敗時に返されます。

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

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
)
```

この関数は引数で渡された優先順位の暗号名 (Cipher) 文字列へのポインタを返します。

Parameters:

- **priority** 整数値で指定する優先順位

See:

- wolfSSL_CIPHER_get_name
- wolfSSL_get_current_cipher

Return:

- 成功時には暗号名 (Cipher) 文字列へのポインタを返します。

- 0 引数で渡された優先順位が範囲外かあるいは無効な値であった場合に返されます。

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
)
```

この関数は wolfSS で有効化されている暗号名 (Cipher) を取得します。

Parameters:

- **buf** 文字列を格納するバッファへのポインタ。
- **len** バッファのサイズ

See:

- GetCipherNames
- wolfSSL_get_cipher_list
- ShowCiphers

Return:

- SSL_SUCCESS 関数がエラーなしで実行された場合に返されます。
- BAD_FUNC_ARG 引数 buf が NULL の場合、または引数 len がゼロ以下の場合に返されます。
- BUFFER_E バッファが十分に大きくなく、オーバーフローする可能性がある場合に返されます。

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
)
```

この関数は、引数を wolfSSL_get_cipher_name_internal に渡すことによって、DHE-RSA の形式の暗号名を取得します。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ

See:

- wolfSSL_CIPHER_get_name
- wolfSSL_get_current_cipher
- wolfSSL_get_cipher_name_internal

Return:

- 成功時には一致した暗号スイートの文字列表現を返します。
- NULL エラーまたは暗号が見つからない場合に返されます。

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 *
)
```

この関数は、SSL 接続の入出力機能として使用されるファイル記述子 (fd) を返します。通常これはソケットファイル記述子になります。

See: wolfSSL_set_fd

Return: fd 成功時には SSL セッションに関連つけられているファイル記述子を返します。

Example

```
int sockfd;
WOLFSSL* ssl = 0;
...
sockfd = wolfSSL_get_fd(ssl);
...
```

C.52.2.53 function wolfSSL_set_using_nonblock

```
void wolfSSL_set_using_nonblock(
    WOLFSSL * ssl,
    int nonblock
)
```

この関数は、WOLFSSL オブジェクトに基礎となる I/O がノンブロックであることを通知します。アプリケーションが WOLFSSL オブジェクトを作成した後、ブロッキング以外のソケットで使用する場合は、wolfssl_set_using_nonblock() を呼び出します。これにより、wolfssl オブジェクトは、EWOULDBLOCK を受信することを意味します。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ
- **nonblock** WOLFSSL オブジェクトにノンブロッキング I/O を使用することを通知するフラグ。１を指定することでノンブロッキング I/O を使用することを指定する。

See:

- wolfSSL_get_using_nonblock
- wolfSSL_dtls_got_timeout

- wolfSSL_dtls_get_current_timeout

Return: なし

Example

```
WOLFSSL* ssl = 0;
...
wolfSSL_set_using_nonblock(ssl, 1);
```

C.52.2.54 function wolfSSL_get_using_nonblock

```
int wolfSSL_get_using_nonblock(
    WOLFSSL *
)
```

この機能により、wolfSSL がノンブロッキング I/O を使用しているかどうかをアプリケーションが判断できます。wolfSSL がノンブロッキング I/O を使用している場合、この関数は 1 を返します。アプリケーションが WOLFSSL オブジェクトを生成した後に wolfSSL_set_using_nonblock() を呼び出してノンブロッキングソケットを使うとこの関数は 1 を返します。これにより、WOLFSSL オブジェクトは、recevfrom がタイムアウトせず代わりに EWOULDBLOCK を受信するようになります。

See: wolfSSL_set_session

Return:

- 0 基礎となる I/O がブロックされています。
- 1 基礎となる I/O は非ブロッキングです。

Example

```
int ret = 0;
WOLFSSL* ssl = 0;
...
ret = wolfSSL_get_using_nonblock(ssl);
if (ret == 1) {
    // underlying I/O is non-blocking
}
...
```

C.52.2.55 function wolfSSL_write

```
int wolfSSL_write(
    WOLFSSL * ssl,
    const void * data,
    int sz
)
```

この関数は、バッファあるいはデータから、SSL 接続に対して、sz バイトを書き込みます。必要に応じて、wolfSSL_write() の呼び出し時点ではまだ wolfSSL_connect() または wolfSSL_accept() がまだ呼び出されていない場合、SSL/TLS セッションをネゴシエートします。wolfSSL_write()は、ブロックとノンブロッキング I/O の両方で動作します。基礎となる入出力がノンブロッキングに設定されている場合、wolfSSL_write() が要求を満たすことができなかったら wolfSSL_write() は関数呼び出しからすぐに戻ります。この場合、wolfSSL_get_error() の呼び出しは SSL_ERROR_WANT_READ または SSL_ERROR_WANT_WRITE のいずれかを返します。その結果、基礎となる I/O が準備ができたら、呼び出し側プロセスは wolfssl_write() への呼び出しを繰り返す必要があります。基礎となる入出力がブロックされている場合、WolfSSL_WRITE() は、サイズ SZ のバッファデータが完全に書かれたかエラーが発生したら、戻るだけです。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ
- **data** ピアに送信されるデータを含んでいるバッファへのポインタ。
- **sz** 送信データを含んでいるバッファのサイズ

See:

- wolfSSL_send
- wolfSSL_read
- wolfSSL_recv

Return:

- 成功時には書き込んだバイト数（1 以上）を返します。
- 0 失敗したときに返されます。特定のエラーコードについて wolfSSL_get_error() を呼び出します。
- SSL_FATAL_ERROR エラーが発生したとき、または非ブロッキングソケットを使用するときには、SSL_ERROR_WANT_READ または SSL_ERROR_WANT_WRITE エラーが受信され、再度 WOLFSSL_WRITE() を呼び出す必要がある場合は、障害が発生します。特定のエラーコードを取得するには、wolfSSL_get_error() を使用してください。

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.52.2.56 function wolfSSL_read

```
int wolfSSL_read(
    WOLFSSL * ssl,
    void * data,
    int sz
)
```

この関数は、SSL セッション (ssl) の内部読み取りバッファから sz バイトをバッファデータに読み出します。読み取られたバイトは内部受信バッファから削除されます。必要に応じて、wolfSSL_read() の呼び出し時点ではまだ wolfSSL_connect() または wolfSSL_accept() がまだ呼び出されていない場合、SSL/TLS セッションをネゴシエートします。SSL/TLS プロトコルは、最大サイズの SSL レコードを使用します（最大レコードサイズは /wolfssl/internal.h）。そのため、wolfSSL は、レコードを処理および復号することができる前に、SSL レコード全体を内部的に読み取る必要があります。このため、wolfSSL_read() への呼び出しは、呼び出し時に復号された最大バッファサイズを返すことができます。検索され、次回の wolfSSL_read() への呼び出しで復号される内部 wolfSSL 受信バッファで待機していない追加の復号データがあるかもしれません。sz が内部読み取りバッファ内のバイト数より大きい場合、wolfSSL_read() は内部読み取りバッファで使用可能なバイトを返します。BYTES が内部読み取りバッファにバッファされていない場合は、wolfSSL_read() への呼び出しは次のレコードの処理をトリガーします。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ
- **data** wolfSSL_read()が読み取るデータを格納するバッファへのポインタ。
- **sz** バッファに読み取るデータのサイズ

See:

- wolfSSL_recv
- wolfSSL_write
- wolfSSL_peek
- wolfSSL_pending

Return:

- 成功時には読み取られたバイト数（1 以上）を返します。
- 0 失敗したときに返されます。これは、クリーン（通知アラートを閉じる）シャットダウンまたはピアが接続を閉じただけであることによって発生する可能性があります。特定のエラーコードについて wolfSSL_get_error() を呼び出します。
- SSL_FATAL_ERROR エラーが発生したとき、またはノンブロッキングソケットを使用するときに、SSL_ERROR_WANT_READ または SSL_ERROR_WANT_WRITE エラーが受信され、再度 wolfSSL_read() を呼び出す必要がある場合は、障害が発生します。特定のエラーコードを取得するには、wolfSSL_get_error() を使用してください。

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.52.2.57 function wolfSSL_peek

```
int wolfSSL_peek(
    WOLFSSL * ssl,
    void * data,
    int sz
)
```

この関数は SSL セッション（SSL）内部読み取りバッファから SZ バイトをバッファデータにコピーします。この関数は、内部 SSL セッション受信バッファ内のデータが削除されていないか変更されていないことを除いて、wolfssl_read() と同じです。必要に応じて、wolfssl_read() のように、wolfssl_peek() はまだ wolfssl_connect() または wolfssl_accept() によってまだ実行されていない場合、wolfssl_peek() は SSL / TLS セッションをネゴシエートします。SSL/TLS プロトコルは、最大サイズの SSL レコードを使用します（最大レコードサイズは /wolfssl/internal.h）。そのため、WolfSSL は、レコードを処理および復号化することができる前に、SSL レコード全体を内部的に読み取る必要があります。このため、wolfssl_peek() への呼び出しは、呼び出し時に復号化された最大バッファサイズを返すことができます。wolfssl_peek()/wolfssl_read() への次の呼び出しで検索および復号化される内部 WolfSSL 受信バッファ内で待機していない追加の復号化データがあるかもしれません。SZ が内部読み取りバッファ内のバイト数よりも大きい場合、SSL_PEEK() は内部読み取りバッファで使用可能なバイトを返します。バイトが内部読み取りバッファにバッファされていない場合、Wolfssl_peek() への呼び出しは次のレコードの処理をトリガーします。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ
- **data** wolfSSL_peek()がデータを読み取るバッファー。
- **sz** バッファに読み取るデータのサイズ

See: wolfSSL_read

Return:

- 成功時には読み取られたバイト数（1 以上）を返します。
- 0 失敗したときに返されます。これは、クリーン（通知アラートを閉じる）シャットダウンまたはピアが接続を閉じただけであることによって発生する可能性があります。特定のエラーコードについて wolfSSL_get_error() を呼び出します。
- SSL_FATAL_ERROR エラーが発生したとき、またはノンブロッキングソケットを使用するときに、SSL_ERROR_WANT_READ または SSL_ERROR_WANT_WRITE エラーが受信され、再度 wolfSSL_peek() を呼び出す必要がある場合は、障害が発生します。特定のエラーコードを取得するには、wolfSSL_get_error() を使用してください。

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.52.2.58 function wolfSSL_accept

```
int wolfSSL_accept(
    WOLFSSL *
)
```

この関数はサーバー側で呼び出され、SSL クライアントが SSL/TLS ハンドシェイクを開始するのを待ちます。この関数が呼び出されると、基礎となる通信チャネルはすでに設定されています。wolfSSL_accept()は、ブロックとノンブロッキング I/O の両方で動作します。基礎となる入出力がノンブロッキングである場合、wolfSSL_accept() は、基礎となる I/O が wolfSSL_accept の要求を満たすことができなかったときに戻ります。この場合、wolfSSL_get_error() への呼び出しは SSL_ERROR_WANT_READ または SSL_ERROR_WANT_WRITE のいずれかを生成します。呼び出しプロセスは、読み取り可能なデータが使用可能であり、wolfSSL が停止した場所を拾うときに、wolfSSL_accept の呼び出しを繰り返す必要があります。ノンブロッキングソケットを使用する場合は、何も実行する必要がありますが、select() を使用して必要な条件を確認できます。基礎となる I/O がブロックされている場合、wolfSSL_accept() はハンドシェイクが終了したら、またはエラーが発生したら戻ります。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ

See:

- wolfSSL_get_error
- wolfSSL_connect

Return:

- SSL_SUCCESS 成功時に返されます。
- SSL_FATAL_ERROR エラーが発生した場合に返されます。より詳細なエラーコードを取得するには、wolfSSL_get_error() を呼び出します。

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.52.2.59 function wolfSSL_CTX_free

```
void wolfSSL_CTX_free(
    WOLFSSL_CTX *
)
```

この関数は、割り当てられた WOLFSSL_CTX オブジェクトを解放します。この関数は CTX 参照数を減らし、参照カウントが 0 に達したときにのみコンテキストを解放します。

Parameters:

- **ctx** wolfSSL_CTX_new()を使用して作成された WOLFSSL_CTX 構造体へのポインタ

See:

- wolfSSL_CTX_new
- wolfSSL_new
- wolfSSL_free

Return: なし

Example

```
WOLFSSL_CTX* ctx = 0;
...
wolfSSL_CTX_free(ctx);
```

C.52.2.60 function wolfSSL_free

```
void wolfSSL_free(
    WOLFSSL *
)
```

この関数は割り当てられた WOLFSSL オブジェクトを解放します。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ

See:

- wolfSSL_CTX_new
- wolfSSL_new
- wolfSSL_CTX_free

Return: なし

Example

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

WOLFSSL* ssl = 0;
```

```
...
wolfSSL_free(ssl);
```

C.52.2.61 function wolfSSL_shutdown

```
int wolfSSL_shutdown(
    WOLFSSL *
)
```

この関数は、引数 ssl の SSL セッションに対してアクティブな SSL/TLS 接続をシャットダウンします。この関数は、ピアに “Close Notify” アラートを送信しようとします。呼び出し側アプリケーションは、Peer がその “Close Notify” アラートを応答として送信してくるのを待つか、または wolfSSL_shutdown から呼び出しが戻った時点で（リソースを保存するために）下層の接続を切断するのを待つことができます。どちらのオプションも TLS 仕様で許されています。シャットダウンした後に下層の接続を再び別のセッションで使用する予定ならば、ピア間で同期を保つために完全な 2 方向のシャットダウン手順を実行する必要があります。wolfSSL_shutdown()は、ブロックとノンブロッキング I/O の両方で動作します。下層の I/O がノンブロッキングの場合、wolfSSL_shutdown() が要求を満たすことができなかった場合、wolfSSL_shutdown() はエラーを返します。この場合、wolfSSL_get_error() への呼び出しは SSL_ERROR_WANT_READ または SSL_ERROR_WANT_WRITE のいずれかを生成します。その結果、下層の I/O が準備ができたら、呼び出し側プロセスは wolfSSL_shutdown() への呼び出しを繰り返す必要があります。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ

See:

- wolfSSL_free
- wolfSSL_CTX_free

Return:

- SSL_SUCCESS 成功時に返されます。
- SSL_SHUTDOWN_NOT_DONE シャットダウンが終了していない場合に返され、関数を再度呼び出す必要があります。
- SSL_FATAL_ERROR 失敗したときに返されます。より具体的なエラーコードは wolfSSL_get_error() を呼び出します。

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.52.2.62 function wolfSSL_send

```
int wolfSSL_send(
    WOLFSSL * ssl,
    const void * data,
    int sz,
    int flags
)
```

この関数は、書き込み操作のために指定されたフラグを使用してバッファあるいはデータから、SSL 接続に対して、sz バイトを書き込みます。必要に応じて、wolfSSL_send() の呼び出し時点ではまだ wolfSSL_connect() または wolfSSL_accept() がまだ呼び出されていない場合、SSL/TLS セッションをネゴシエートします。wolfSSL_send()は、ブロックとノンブロッキング I/O の両方で動作します。基礎となる入出力がノンブロッキングに設定されている場合、wolfSSL_send() が要求を満たすことができなかったら wolfSSL_send() は関数呼び出しからすぐに戻ります。この場合、wolfSSL_get_error() の呼び出しは SSL_ERROR_WANT_READ または SSL_ERROR_WANT_WRITE のいずれかを返します。その結果、基礎となる I/O が準備ができたら、呼び出し側プロセスは wolfSSL_send() への呼び出しを繰り返す必要があります。基礎となる入出力がブロックされている場合、wolfSSL_send() は、サイズ SZ のバッファデータが完全に書かれたかエラーが発生したら、戻るだけです。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ
- **data** ピアに送信されるデータを含んでいるバッファへのポインタ。
- **sz** 送信データを含んでいるバッファのサイズ
- **flags** 下層の I/O の send に対して指定するフラグ

See:

- wolfSSL_write
- wolfSSL_read
- wolfSSL_recv

Return:

- 成功時には書き込んだバイト数（1 以上）を返します。
- 0 失敗したときに返されます。特定のエラーコードについて wolfSSL_get_error() を呼び出します。
- SSL_FATAL_ERROR エラーが発生したとき、または非ブロッキングソケットを使用するときには、SSL_ERROR_WANT_READ または SSL_ERROR_WANT_WRITE エラーが受信され、再度 WOLFSSL_WRITE() を呼び出す必要がある場合は、障害が発生します。特定のエラーコードを取得するには、wolfSSL_get_error() を使用してください。

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.52.2.63 function wolfSSL_recv

```
int wolfSSL_recv(
    WOLFSSL * ssl,
    void * data,
    int sz,
    int flags
)
```

この関数は、基礎となる RECV 動作のために指定されたフラグを使用して、SSL セッション（ssl）内部読み取りバッファから sz バイトをバッファデータに読み出します。読み取られたバイトは内部受信バッファから削除されます。この関数は wolfssl_read() と同じです。ただし、アプリケーションが基礎となる読み取り操作の RECV フラグを設定できることを許可します。必要に応じて wolfssl_recv() が wolfssl_connect()

または wolfssl_accept() によってハンドシェイクがまだ実行されていない場合は、SSL/TLS セッションをネゴシエートします。SSL/TLS プロトコルは、最大サイズの SSL レコードを使用します（最大レコードサイズは /wolfssl/internal.h）。そのため、wolfSSL は、レコードを処理および復号することができる前に、SSL レコード全体を内部的に読み取る必要があります。このため、wolfSSL_recv() への呼び出しは、呼び出し時に復号された最大バッファサイズを返すことができるだけです。wolfSSL_recv()への次の呼び出しで検索および復号される内部 wolfSSL 受信バッファで待機していない追加の復号化されたデータがあるかもしれません。引数 sz が内部読み取りバッファ内のバイト数よりも大きい場合、wolfSSL_recv() は内部読み取りバッファで使用可能なバイトを返します。バイトが内部読み取りバッファにバッファされていない場合は、wolfSSL_recv() への呼び出しは次のレコードの処理をトリガーします。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ
- **data** wolfSSL_recv()がデータを読み取るバッファー。
- **sz** データを読み込むためのバイト数。

See:

- wolfSSL_read
- wolfSSL_write
- wolfSSL_peek
- wolfSSL_pending

Return:

- 成功時には読み取られたバイト数 (1 以上) を返します。
- 0 失敗したときに返されます。これは、クリーン（通知アラートを閉じる）シャットダウンまたはピアが接続を閉じただけであることによって発生する可能性があります。特定のエラーコードについて wolfSSL_get_error() を呼び出します。
- SSL_FATAL_ERROR エラーが発生した場合、または非ブロッキングソケットを使用するときには、SSL_ERROR_WANT_READ または SSL_ERROR_WANT_WRITE エラーが発生し、アプリケーションが再び WOLFSSL_RECV() を呼び出す必要があります。特定のエラーコードを取得するには、wolfSSL_get_error() を使用してください。

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.52.2.64 function wolfSSL_get_error

```
int wolfSSL_get_error(
    WOLFSSL * ssl,
    int ret
)
```

この関数は、直前の API 関数呼び出し（wolfssl_connect、wolfssl_accept、wolfssl_read、wolfssl_write など）がエラーコード（SSL_FAILURE）を呼び出した理由を表す一意のエラーコードを返します。直前の関数の戻り値は、ret を介して wolfSSL_get_error に渡されます。wolfSSL_get_error は一意のエラーコードを返します。wolfSSL_err_error_string() を呼び出して人間が読めるエラー文字列を取得することができます。詳細については、wolfSSL_err_error_string() を参照してください。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ

See:

- wolfSSL_ERR_error_string
- wolfSSL_ERR_error_string_n
- wolfSSL_ERR_print_errors_fp
- wolfSSL_load_error_strings

Return:

- 呼び出し成功時、この関数は、直前の関数が失敗した理由を説明する固有のエラーコードを返します。
- SSL_ERROR_NONE 引数 ret が 0 より大きい場合に返されます。ret が 0 以下の場合、直前の API がエラーコードを返すが実際に発生しなかった場合にこの値を返す場合があります。例としては、引数 sz に 0 を渡して wolfSSL_read() を呼び出す場合に発生します。wolfssl_read() が 0 を戻した場合は通常エラーを示しますが、この場合はエラーは発生していません。従って、wolfSSL_get_error() がその後呼び出された場合、ssl_error_none が返されます。

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.52.2.65 function wolfSSL_get_alert_history

```
int wolfSSL_get_alert_history(
    WOLFSSL * ssl,
    WOLFSSL_ALERT_HISTORY * h
)
```

この関数はアラート履歴を取得します。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WolfSSL 構造へのポインタ。
- **h** WOLFSSL 構造体の “alert_history member” の値が格納される、WOLFSSL_ALERT_HISTORY 構造体へのポインタ。

See: wolfSSL_get_error

Return: SSL_SUCCESS 関数が正常に完了したときに返されます。警告履歴があったか、またはいずれにも、戻り値は SSL_SUCCESS です。

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.52.2.66 function wolfSSL_set_session

```
int wolfSSL_set_session(
    WOLFSSL * ssl,
    WOLFSSL_SESSION * session
)
```

この関数は、SSL オブジェクト SSL が SSL/TLS 接続を確立する目的で使用するセッションを設定します。セッション再開を行う場合、wolfSSL_shutdown() を呼び出す前に wolfSSL_get1_session() を呼び出してセッションオブジェクトを取得し、セッション ID を保存しておく必要があります。後で、アプリケーションは新しい WOLFSSL オブジェクトを作成し、保存したセッションを wolfSSL_set_session() に渡す必要があります。その後アプリケーションは wolfSSL_connect() を呼び出し、wolfSSL はセッション再開を試みます。wolfSSL サーバーコードでは、デフォルトでセッション再開を許可します。wolfSSL_get1_session()によって返されたオブジェクトは、アプリケーションが使用後に解放する必要があります。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ
- **session** WOLFSSL_SESSION 構造体へのポインタ。

See: wolfSSL_get1_session

Return:

- SSL_SUCCESS セッションを正常に設定すると返されます。
- SSL_FAILURE 失敗した場合に返されます。これはセッションキャッシュが無効になっている、またはセッションがタイムアウトした場合によって発生する可能性があります。
- OPENSSL_EXTRA と WOLFSSL_ERROR_CODE_OPENSSL が定義されている場合には、セッションがタイムアウトしていても SSL_SUCCESS が返されます。

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.52.2.67 function wolfSSL_get_session

```
WOLFSSL_SESSION * wolfSSL_get_session(
    WOLFSSL * ssl
)
```

NO_SESSION_CACHE_REF が定義されている場合、この関数は SSL で使用されている現在のセッション（WOLFSSL_SESSION）へのポインタを返します。この関数は、WOLFSSL_SESSION オブジェクトへの永続的なポインタを返します。返されるポインタは、wolfSSL_free が呼び出されたときに解放されます。この呼び出しは、現在のセッションを検査または変更するためにのみ使用されます。セッション再開に使用する場合は、wolfSSL_get1_session() を使用することをお勧めします。NO_SESSION_CACHE_REF が定義されて

いない場合の後方互換性のために、この関数はローカルキャッシュに格納されている永続セッションオブジェクトポインタを返します。キャッシュサイズは有限であり、アプリケーションが wolfSSL_set_session() を呼び出す時までにセッションオブジェクトが別の SSL 接続によって上書きされる危険性があります。アプリケーションに NO_SESSION_CACHE_REF を定義し、セッション再開に wolfSSL_get1_session() を使用することをお勧めします。

See:

- wolfSSL_get1_session
- wolfSSL_set_session

Return:

- 現在の SSL セッションオブジェクトへのポインタを返します。
- NULL ssl が NULL の場合、SSL セッションキャッシュが無効になっている場合、wolfSSL はセッション ID を使用できない、またはミューテックス関数が失敗した場合に返されます。

Example

```
WOLFSSL* ssl;
WOLFSSL_SESSION* session;
...
session = wolfSSL_get_session(ssl);
if (session == NULL) {
    // failed to get session pointer
}
...
```

C.52.2.68 function wolfSSL_flush_sessions

```
void wolfSSL_flush_sessions(
    WOLFSSL_CTX * ctx,
    long tm
)
```

この機能は、期限切れになったセッションキャッシュからセッションをフラッシュします。時間比較には引数 tm が使用されます。wolfSSL は現在セッションに静的テーブルを使用しているため、フラッシングは不要です。そのため、この機能は現在スタブとして存在しています。この関数は、wolfssl が OpenSSL 互換層でコンパイルされているときの OpenSSL 互換性（ssl_flush_sessions）を提供します。

Parameters:

- **ctx** wolfSSL_CTX_new()を使用して作成された WOLFSSL_CTX 構造体へのポインタ。
- **tm** セッションの有効期限の比較で使用される時間

See:

- wolfSSL_get1_session
- wolfSSL_set_session

Return: なし

Example

```
WOLFSSL_CTX* ssl;
...
wolfSSL_flush_sessions(ctx, time(0));
```

C.52.2.69 function wolfSSL_SetServerID

```
int wolfSSL_SetServerID(
    WOLFSSL * ssl,
    const unsigned char * id,
    int len,
    int newSession
)
```

この関数はクライアントセッションをサーバー ID と関連付けます。引数 newSession がオンの場合、既存のセッションは再利用されません。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ。
- **id** WOLFSSL_SESSION 構造体の ServerID メンバーにコピーされるサーバー ID データへのポインタ。
- **len** サーバー ID データのサイズ
- **newSession** セッションを再利用するか否かを指定するフラグ。オンの場合、既存のセッションは再利用されません。

See: wolfSSL_set_session

Return:

- SSL_SUCCESS 関数がエラーなしで実行された場合に返されます。
- BAD_FUNC_ARG 引数 ssl または引数 id が NULL の場合、または引数 len がゼロ以下の場合に返されます。

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.52.2.70 function wolfSSL_GetSessionIndex

```
int wolfSSL_GetSessionIndex(
    WOLFSSL * ssl
)
```

この関数は、WOLFSSL 構造体の指定セッションインデックス値を取得します。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ。

See: wolfSSL_GetSessionAtIndex

Return: この関数は、WOLFSSL 構造体内の SessionIndex を表す int 型の値を返します。

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.52.2.71 function wolfSSL_GetSessionAtIndex

```
int wolfSSL_GetSessionAtIndex(
    int index,
    WOLFSSL_SESSION * session
)
```

この関数はセッションキャッシュの指定されたインデックスのセッションを取得し、それをメモリにコピーします。WOLFSSL_SESSION 構造体はセッション情報を保持します。

Parameters:

- **idx** セッションインデックス値
- **session** WOLFSSL_SESSION 構造体へのポインタ

See:

- UnLockMutex
- LockMutex
- wolfSSL_GetSessionIndex

Return:

- SSL_SUCCESS 関数が正常に実行され、エラーがスローされなかった場合に返されます。
- BAD_MUTEX_E アンロックまたはロックミューテックスエラーが発生した場合に返されます。
- SSL_FAILURE 関数が正常に実行されなかった場合に返されます。

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.52.2.72 function wolfSSL_SESSION_get_peer_chain

```
WOLFSSL_X509_CHAIN * wolfSSL_SESSION_get_peer_chain(
    WOLFSSL_SESSION * session
)
```

WOLFSSL_SESSION 構造体からピア証明書チェーンを返します。

Parameters:

- **session** WOLFSSL_SESSION 構造体へのポインタ

See:

- wolfSSL_GetSessionAtIndex
- wolfSSL_GetSessionIndex
- AddSession

Example

```
WOLFSSL_SESSION* session;
WOLFSSL_X509_CHAIN* chain;
...
chain = wolfSSL_SESSION_get_peer_chain(session);
if(!chain){
    // There was no chain. Failure case.
}
```

C.52.2.73 function wolfSSL_CTX_set_verify

```
void wolfSSL_CTX_set_verify(
    WOLFSSL_CTX * ctx,
    int mode,
    VerifyCallback verify_callback
)
```

この関数はリモートピアの検証方法を設定し、また証明書検証コールバック関数を SSL コンテキストに登録することもできます。検証コールバックは、検証障害が発生した場合にのみ呼び出されます。検証コールバックが必要な場合は、NULL ポインタを verify_callback に使用できます。ピア証明書の検証モードは、論理的またはフラグのリストです。可能なフラグ値は次のとおりです: SSL_VERIFY_NONE -クライアントモード：クライアントはサーバーから受信した証明書を検証せず、ハンドシェイクは通常どおり続きます。-サーバーモード：サーバーはクライアントに証明書要求を送信しません。そのため、クライアント検証は有効になりません。SSL_VERIFY_PEER -クライアントモード：クライアントはハンドシェイク中にサーバーから受信した証明書を検証します。これは wolfSSL ではデフォルトでオンにされます。したがって、このオプションを使用すると効果がありません。-サーバーモード：サーバーは証明書要求をクライアントに送信し、受信したクライアント証明書を確認します。SSL_VERIFY_FAIL_IF_NO_PEER_CERT -クライアントモード：クライアント側で使用されていない場合は効果がありません。-サーバーモード：要求されたときにクライアントが証明書の送信に失敗した場合は、サーバー側で検証が失敗します（SSL サーバーの SSL_VERIFY_PEER を使用する場合）。SSL_VERIFY_FAIL_EXCEPT_PSK -クライアントモード：クライアント側で使用されていない場合は効果がありません。-サーバーモード：PSK 接続の場合を除き、検証は SSL_VERIFY_FAIL_IF_NO_PEER_CERT と同じです。PSK 接続が行われている場合、接続はピア証明書なしで通過します。

Parameters:

- **ctx** wolfSSL_CTX_new()で作成された SSL コンテキストへのポインタ。
- **mode** ピアの証明書をどのように検証するかを示すフラグ値
- **verify_callback** 証明書検証が失敗した際に呼び出されるコールバック関数。必要がないなら NULL を指定すること。

See: wolfSSL_set_verify

Return: なし

Example

```
WOLFSSL_CTX*    ctx    = 0;
...
wolfSSL_CTX_set_verify(ctx, (WOLFSSL_VERIFY_PEER |
                        WOLFSSL_VERIFY_FAIL_IF_NO_PEER_CERT), NULL);
```

C.52.2.74 function wolfSSL_set_verify

```
void wolfSSL_set_verify(
    WOLFSSL * ssl,
    int mode,
    VerifyCallback verify_callback
)
```

この関数はリモートピアの検証方法を設定し、また証明書検証コールバック関数を WOLFSSL オブジェクトに登録することもできます。検証コールバックは、検証障害が発生した場合にのみ呼び出されます。検証コールバックが必要な場合は、NULL ポインタを verify_callback に使用できます。ピア証明書の検証モードは、論理的またはフラグのリストです。可能なフラグ値は次のとおりです: SSL_VERIFY_NONE -クライアントモード：クライアントはサーバーから受信した証明書を検証せず、ハンドシェイクは通常どおり続きます。-サーバーモード：サーバーはクライアントに証明書要求を送信しません。そのため、クライアント検証は有効になりません。SSL_VERIFY_PEER -クライアントモード：クライアントはハンドシェイク中にサーバーから受信した証明書を検証します。これは wolfSSL ではデフォルトでオンにされます。したがって、このオプションを使用すると効果がありません。-サーバーモード：サーバーは証明書要求をクライアントに送信し、受信したクライアント証明書を確認します。SSL_VERIFY_FAIL_IF_NO_PEER_CERT -クライアントモード：クライアント側で使用されていない場合は効果がありません。-サーバーモード：要求されたときにクライアントが証明書の送信に失敗した場合は、サーバー側で検証が失敗します（SSL サーバーの SSL_VERIFY_PEER を使用する場合）。SSL_VERIFY_FAIL_EXCEPT_PSK -クライアントモード：クライアント側で使用されていない場合は効果がありません。-サーバーモード：PSK 接続の場合を除き、検証は SSL_VERIFY_FAIL_IF_NO_PEER_CERT と同じです。PSK 接続が行われている場合、接続はピア証明書なしで通過します。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ
- **mode** ピアの証明書をどのように検証するかを示すフラグ値
- **verify_callback** 証明書検証が失敗した際に呼び出されるコールバック関数。必要がないなら NULL を指定すること。

See: wolfSSL_CTX_set_verify

Return: なし

Example

```
WOLFSSL* ssl = 0;
...
wolfSSL_set_verify(ssl, SSL_VERIFY_PEER | SSL_VERIFY_FAIL_IF_NO_PEER_CERT, 0);
```

C.52.2.75 function wolfSSL_SetCertCbCtx

```
void wolfSSL_SetCertCbCtx(
    WOLFSSL * ssl,
    void * ctx
)
```

この関数は、検証コールバックのためのユーザー CTX オブジェクト情報を格納します。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ。
- **ctx** ボイドポインタ。WOLFSSL 構造体の verifyCbCtx メンバーにセットされます。

See:

- wolfSSL_CTX_save_cert_cache
- wolfSSL_CTX_restore_cert_cache
- wolfSSL_CTX_set_verify

Return: なし

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.52.2.76 function wolfSSL_CTX_SetCertCbCtx

```
void wolfSSL_CTX_SetCertCbCtx(
    WOLFSSL_CTX * ctx,
    void * userCtx
)
```

この関数は、検証コールバックのためのユーザー CTX オブジェクト情報を格納します。

Parameters:

- **ctx** WOLFSSL_CTX 構造体へのポインタ。
- **ctx** ボイドポインタ。WOLFSSL_CTX 構造体の verifyCbCtx メンバーにセットされます。

See:

- wolfSSL_CTX_save_cert_cache
- wolfSSL_CTX_restore_cert_cache
- wolfSSL_CTX_set_verify

Return: なし

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.52.2.77 function wolfSSL_pending

```
int wolfSSL_pending(
    WOLFSSL *
)
```

この関数は、wolfSSL_read() によって読み取られる WOLFSSL オブジェクトでバッファされているバイト数を返します。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ

See:

- wolfSSL_recv
- wolfSSL_read
- wolfSSL_peek

Return: この関数は、保留中のバイト数を返します。

Example

```
int pending = 0;
WOLFSSL* ssl = 0;
...

pending = wolfSSL_pending(ssl);
printf("There are %d bytes buffered and available for reading", pending);
```

C.52.2.78 function wolfSSL_load_error_strings

```
void wolfSSL_load_error_strings(
    void
)
```

この機能は OpenSSL API（SSL_load_error_string）との互換性の目的みで提供してあり処理は行いません。

Parameters:

- **なし** *Example*

```
wolfSSL_load_error_strings();
```

See:

- wolfSSL_get_error
- wolfSSL_ERR_error_string
- wolfSSL_ERR_error_string_n
- wolfSSL_ERR_print_errors_fp
- wolfSSL_load_error_strings

Return: なし

C.52.2.79 function wolfSSL_library_init

```
int wolfSSL_library_init(
    void
)
```

この関数は wolfSSL_CTX_new() 内で内部的に呼び出されます。この関数は wolfSSL_Init() のラッパーで、wolfSSL が OpenSSL 互換層でコンパイルされたときの OpenSSL API（ssl_library_init）との互換性の為に存在します。wolfSSL_init() は、より一般的に使用されている wolfSSL 初期化機能です。

See:

- wolfSSL_Init
- wolfSSL_Cleanup

Return:

- SSL_SUCCESS 成功した場合に返されます。に返されます。
- SSL_FATAL_ERROR 失敗したときに返されます。

Example

```
int ret = 0;
ret = wolfSSL_library_init();
if (ret != SSL_SUCCESS) {
    failed to initialize wolfSSL
}
...
```

C.52.2.80 function wolfSSL_SetDevId

```
int wolfSSL_SetDevId(
    WOLFSSL * ssl,
    int devId
)
```

この関数は WOLFSSL オブジェクトレベルで Device Id をセットします。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ
- **devId** ハードウエアと共に使用する際に指定する ID

See:

- wolfSSL_CTX_SetDevId
- wolfSSL_CTX_GetDevId

Return:

- WOLFSSL_SUCCESS 成功時に返されます。
- BAD_FUNC_ARG ssl が NULL の場合に返されます。

Example

```
WOLFSSL* ssl;
int DevId = -2;

wolfSSL_SetDevId(ssl, devId);
```

C.52.2.81 function wolfSSL_CTX_SetDevId

```
int wolfSSL_CTX_SetDevId(
    WOLFSSL_CTX * ctx,
    int devId
)
```

この関数は WOLFSSL_CTX レベルで Device Id をセットします。

Parameters:

- **ctx** wolfSSL_CTX_new()で作成された SSL コンテキストへのポインタ。
- **devId** ハードウエアと共に使用する際に指定する ID

See:

- wolfSSL_SetDevId
- wolfSSL_CTX_GetDevId

Return:

- WOLFSSL_SUCCESS 成功時に返されます。
- BAD_FUNC_ARG ssl が NULL の場合に返されます。

Example

```
WOLFSSL_CTX* ctx;
int DevId = -2;

wolfSSL_CTX_SetDevId(ctx, devId);
```

C.52.2.82 function wolfSSL_CTX_GetDevId

```
int wolfSSL_CTX_GetDevId(
    WOLFSSL_CTX * ctx,
    WOLFSSL * ssl
)
```

この関数は WOLFSSL_CTX レベルで Device Id を取得します。

Parameters:

- **ctx** wolfSSL_CTX_new()で作成された SSL コンテキストへのポインタ。
- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ

See:

- wolfSSL_SetDevId
- wolfSSL_CTX_SetDevId

Return:

- devId 成功時に返されます。
- INVALID_DEVID SSL と CTX の両方が NULL の場合に返されます。

Example

```
WOLFSSL_CTX* ctx;

wolfSSL_CTX_GetDevId(ctx, ssl);
```

C.52.2.83 function wolfSSL_CTX_set_session_cache_mode

```
long wolfSSL_CTX_set_session_cache_mode(
    WOLFSSL_CTX * ctx,
    long mode
)
```

この関数は SSL セッションキャッシュ機能を有効または無効にします。動作はモードに使用される値によって異なります。モードの値は次のとおりです：SSL_SESS_CACHE_OFF - セッションキャッシングを無効にします。デフォルトでセッションキャッシングがオンになっています。SSL_SESS_CACHE_NO_AUTO_CLEAR - セッションキャッシュのオートフラッシュを無効にします。デフォルトで自動フラッシングはオンになっています。

Parameters:

- **ctx** wolfSSL_CTX_new()で作成された SSL コンテキストへのポインタ。
- **mode** セッションキャッシュの振る舞いを変更する為に使用します。

See:

- wolfSSL_flush_sessions
- wolfSSL_get1_session
- wolfSSL_set_session
- wolfSSL_get_sessionID
- wolfSSL_CTX_set_timeout

Return: SSL_SUCCESS 成功に戻ります。

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.52.2.84 function wolfSSL_set_session_secret_cb

```
int wolfSSL_set_session_secret_cb(
    WOLFSSL * ssl,
    SessionSecretCb cb,
    void * ctx
)
```

この関数はセッションシークレットコールバック関数をセットします。SessionSecretCb タイプは次のシグネチャとなっています：int（* sessioneCretcb）（wolfssl * ssl、void * secret、int * secretsz、void * ctx）。WOLFSSL 構造体の sessionSecretCb メンバーは引数 cb に設定されます。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ
- **cb** セッションシークレットコールバック関数ポインタ。
- **ctx** セッションシークレットコールバック関数に渡されるユーザーコンテキスト。

See: SessionSecretCb

Return:

- SSL_SUCCESS 関数の実行がエラーを返されなかった場合に返されます。
- SSL_FATAL_ERROR WOLFSSL 構造が NULL の場合に返されます。

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.52.2.85 function wolfSSL_save_session_cache

```
int wolfSSL_save_session_cache(
    const char * fname
)
```

この関数はセッションキャッシュをファイルに持続します。追加のメモリ使用のため、memsave は使用されません。

Parameters:

- **fname** 書き込み対象ファイル名へのポインタ。

See:

- XFWRITE
- wolfSSL_restore_session_cache
- wolfSSL_memrestore_session_cache

Return:

- SSL_SUCCESS 関数がエラーなしで実行された場合に返されます。セッションキャッシュはファイルに書き込まれました。
- SSL_BAD_FILE FNAME を開くことができないか、それ以外の場合は破損した場合に返されます。
- FWRITE_ERROR XfWrite がファイルへの書き込みに失敗した場合に返されます。
- BAD_MUTEX_E ミューテックスロック障害が発生した場合に返されます。

Example

```
const char* fname;
...
if(wolfSSL_save_session_cache(fname) != SSL_SUCCESS){
    // Fail to write to file.
}
```

C.52.2.86 function wolfSSL_restore_session_cache

```
int wolfSSL_restore_session_cache(
    const char * fname
)
```

この関数はファイルから永続セッションキャッシュを復元します。追加のメモリ使用のため、memstore は使用しません。

Parameters:

- **fname** キャッシュを読み取るためのファイル名へのポインタ。

See:

- XFREAD
- XFOPEN

Return:

- SSL_SUCCESS 関数がエラーなしで実行された場合に返されます。
- SSL_BAD_FILE 関数に渡されたファイルが破損していて XFOPEN によって開くことができなかった場合に返されます。
- FREAD_ERROR ファイルに XFREAD から読み取りエラーが発生した場合に返されます。
- CACHE_MATCH_ERROR セッションキャッシュヘッダの一致が失敗した場合に返されます。
- BAD_MUTEX_E ミューテックスロック障害が発生した場合に返されます。

Example

```
const char *fname;
...
if(wolfSSL_restore_session_cache(fname) != SSL_SUCCESS){
    // Failure case. The function did not return SSL_SUCCESS.
}
```

C.52.2.87 function wolfSSL_memsave_session_cache

```
int wolfSSL_memsave_session_cache(
    void * mem,
    int sz
)
```

この関数はセッションキャッシュをメモリに保持します。

Parameters:

- **mem** セッションキャッシュのコピー先バッファへのポインタ

- **sz** コピー先バッファのサイズ

See:

- XMEMCPY
- wolfSSL_get_session_cache_memsize

Return:

- SSL_SUCCESS 関数がエラーなしで実行された場合に返されます。セッションキャッシュはメモリに正常に永続化されました。
- BAD_MUTEX_E ミューテックスロックエラーが発生した場合に返されます。
- BUFFER_E バッファサイズが小さすぎると返されます。

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.52.2.88 function wolfSSL_memrestore_session_cache

```
int wolfSSL_memrestore_session_cache(
    const void * mem,
    int sz
)
```

この関数はメモリから永続セッションキャッシュを復元します。

Parameters:

- **mem** セッションキャッシュを保持しているバッファへのポインタ。
- **sz** バッファのサイズ

See: wolfSSL_save_session_cache

Return:

- SSL_SUCCESS 関数がエラーなしで実行された場合に返されます。
- BUFFER_E メモリバッファが小さすぎると返されます。
- BAD_MUTEX_E セッションキャッシュミューテックスロックが失敗した場合に返されます。
- CACHE_MATCH_ERROR セッションキャッシュヘッダの一致が失敗した場合に返されます。

Example

```
const void* memoryFile;
int szMf;
...
if(wolfSSL_memrestore_session_cache(memoryFile, szMf) != SSL_SUCCESS){
    // Failure case. SSL_SUCCESS was not returned.
}
```

C.52.2.89 function wolfSSL_get_session_cache_memsize

```
int wolfSSL_get_session_cache_memsize(
    void
)
```

この関数は、セッションキャッシュ保存バッファをどのように大きくするかを返します。

See: wolfSSL_memrestore_session_cache

Return: この関数は、セッションキャッシュ保存バッファのサイズを表す整数を返します。

Example

```
int sz = // Minimum size for error checking;
...
if(sz < wolfSSL_get_session_cache_memsize()){
    // Memory buffer is too small
}
```

C.52.2.90 function wolfSSL_CTX_save_cert_cache

```
int wolfSSL_CTX_save_cert_cache(
    WOLFSSL_CTX * ctx,
    const char * fname
)
```

この関数は Cert キャッシュをメモリからファイルに書き込みます。

Parameters:

- **ctx** WOLFSSL_CTX 構造体へのポインタ、証明書情報を保持します。
- **fname** 出力先ファイル名へのポインタ

See:

- CM_SaveCertCache
- DoMemSaveCertCache

Return:

- SSL_SUCCESS CM_SaveCertCache が正常に終了した場合。
- BAD_FUNC_ARG 引数のいずれかの引数が NULL の場合に返されます。
- SSL_BAD_FILE 証明書キャッシュ保存ファイルを開くことができなかった場合。
- BAD_MUTEX_E ロックミューテックスが失敗した場合
- MEMORY_E メモリの割り当てに失敗しました。
- FWRITE_ERROR 証明書キャッシュファイルの書き込みに失敗しました。

Example

```
WOLFSSL_CTX* ctx = WOLFSSL_CTX_new( protocol def );
const char* fname;
...
if(wolfSSL_CTX_save_cert_cache(ctx, fname)){
    // file was written.
}
```

C.52.2.91 function wolfSSL_CTX_restore_cert_cache

```
int wolfSSL_CTX_restore_cert_cache(
    WOLFSSL_CTX * ctx,
    const char * fname
)
```

この関数はファイルから証明書キャッシュを担当します。

Parameters:

- **ctx** WOLFSSL_CTX 構造体へのポインタ、証明書情報を保持します。
- **fname** 証明書キャッシュを読み取るファイル名へのポインタ。

See:

- CM_RestoreCertCache
- XFOPEN

Return:

- SSL_SUCCESS 正常に実行された場合に返されます。
- SSL_BAD_FILE XFOPEN が XBADFILE を返すと返されます。ファイルが破損しています。
- MEMORY_E TEMP バッファの割り当てられたメモリが失敗した場合に返されます。
- BAD_FUNC_ARG 引数 fname または引数 ctx が NULL である場合に返されます。

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.52.2.92 function wolfSSL_CTX_memsave_cert_cache

```
int wolfSSL_CTX_memsave_cert_cache(
    WOLFSSL_CTX * ctx,
    void * mem,
    int sz,
    int * used
)
```

この関数は証明書キャッシュをメモリに持続します。

Parameters:

- **ctx** wolfSSL_CTX_new()を使用して作成された WOLFSSL_CTX 構造体へのポインタ。
- **mem** 宛先への void ポインタ（出力バッファ）。
- **sz** 出力バッファのサイズ。
- **used** 証明書キャッシュヘッダーのサイズを格納する変数へのポインタ。

See:

- DoMemSaveCertCache
- GetCertCacheMemSize
- CM_MemRestoreCertCache
- CM_GetCertCacheMemSize

Return:

- SSL_SUCCESS 機能の実行に成功したことに戻ります。エラーが投げられていません。
- BAD_MUTEX_E WOLFSSL_CERT_MANAGER 構造体の caLock メンバー 0（ゼロ）ではなかった。
- BAD_FUNC_ARG 引数 ctx、mem が NULL の場合、または sz が 0 以下の場合に返されます。
- BUFFER_E 出力バッファ MEM が小さすぎました。

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.52.2.93 function wolfSSL_CTX_memrestore_cert_cache

```
int wolfSSL_CTX_memrestore_cert_cache(
    WOLFSSL_CTX * ctx,
    const void * mem,
    int sz
)
```

この関数は証明書キャッシュをメモリから復元します。

Parameters:

- **ctx** wolfSSL_CTX_new()を使用して作成された WOLFSSL_CTX 構造体へのポインタ。
- **mem** 証明書キャッシュに復元される値を保持しているバッファへのポインタ。
- **sz** バッファのサイズ

See: CM_MemRestoreCertCache

Return:

- SSL_SUCCESS 関数とサブルーチンがエラーなしで実行された場合に返されます。
- BAD_FUNC_ARG CTX または MEM パラメータが NULL または SZ パラメータがゼロ以下の場合に返されます。
- BUFFER_E CERT キャッシュメモリバッファが小さすぎると戻ります。
- CACHE_MATCH_ERROR CERT キャッシュヘッダーの不一致があった場合に返されます。
- BAD_MUTEX_E ロックミューテックスが失敗した場合に返されます。

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.52.2.94 function wolfSSL_CTX_get_cert_cache_memsize

```
int wolfSSL_CTX_get_cert_cache_memsize(
    WOLFSSL_CTX * ctx
)
```

Certificate Cache Save バッファが必要なサイズを返します。

Parameters:

- **ctx** wolfSSL_CTX_new()で作成された SSL コンテキストへのポインタ。

See: CM_GetCertCacheMemSize

Return:

- メモリサイズを返します。

- BAD_FUNC_ARG WOLFSSL_CTX 構造体が NULL の場合に返されます。
- BAD_MUTEX_E ミューテックスロックエラーが発生した場合に返されます。

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.52.2.95 function wolfSSL_CTX_set_cipher_list

```
int wolfSSL_CTX_set_cipher_list(
    WOLFSSL_CTX * ctx,
    const char * list
)
```

この関数は、与えられた WOLFSSL_CTX に暗号スイートリストを設定します。この暗号スイートリストは、このコンテキストを使用して作成された新しい SSL セッション（WolfSSL）のデフォルトリストになります。リスト内の暗号は、優先度の高いものの順に順にソートされるべきです。wolfSSL_CTX_set_cipher_list()が呼び出される都度、特定の SSL コンテキストの暗号スイートリストを提供されたリストにリセットします。暗号スイートリストはヌル終端されたコロン区切りリストです。たとえば、リストの値が「DHE-RSA-AES256-SHA256：DHE-RSA-AES128-SHA256：AES256-SHA256」有効な暗号値は、src/internal.c の cipher_names [] 配列のフルネーム値です。（有効な暗号化値の明確なリストの場合は src/internal.c をチェックしてください）

Parameters:

- **ctx** wolfSSL_CTX_new()で作成された SSL コンテキストへのポインタ。
- **list** ヌル終端されたコロン区切りの暗号スイートリスト文字列へのポインタ。

See:

- wolfSSL_set_cipher_list
- wolfSSL_CTX_new

Return:

- SSL_SUCCESS 成功時に返されます。
- SSL_FAILURE 失敗した場合に返されます。

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.52.2.96 function wolfSSL_set_cipher_list

```
int wolfSSL_set_cipher_list(
    WOLFSSL * ssl,
    const char * list
)
```

この関数は、特定の WolfSSL オブジェクト（SSL セッション）の暗号スイートリストを設定します。この暗号スイートリストは、このコンテキストを使用して作成された新しい SSL セッション（WolfSSL）のデフォルトリストになります。リスト内の暗号は、優先度の高いものの順に順にソートされるべきです。wolfSSL_CTX_set_cipher_list()が呼び出される都度、特定の SSL コンテキストの暗号スイートリストを提供されたリストにリセットします。暗号スイートリストはヌル終端されたコロン区切りリストです。たとえば、リストの値が「DHE-RSA-AES256-SHA256：DHE-RSA-AES128-SHA256：AES256-SHA256」有効な暗号値は、src/internal.c の cipher_names [] 配列のフルネーム値です。（有効な暗号化値の明確なリストの場合は src/internal.c をチェックしてください）

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ
- **list** ヌル終端されたコロン区切りの暗号スイートリスト文字列へのポインタ。

See:

- wolfSSL_CTX_set_cipher_list
- wolfSSL_new

Return:

- SSL_SUCCESS 機能完了に成功したときに返されます。
- SSL_FAILURE 失敗した場合に返されます。

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.52.2.97 function wolfSSL_dtls_set_using_nonblock

```
void wolfSSL_dtls_set_using_nonblock(
    WOLFSSL * ssl,
    int nonblock
)
```

この関数は WOLFSSL DTLS オブジェクトに下層の UDP I/O はノンブロッキングであることを通知します。アプリケーションが WOLFSSL オブジェクトを作成した後、ノンブロッキング UDP ソケットを使用する場合は、wolfSSL_dtls_set_using_nonblock() を呼び出します。これにより、WOLFSSL オブジェクトは、recvfrom 呼び出しがタイムアウトせずに EWOULDBLOCK を受信することを意味します。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ
- **nonblock** WOLFSSL 構造体にノンブロッキング I/O を使用していることを指定するフラグ。ノンブロッキングを使用している場合には 1 を指定、それ以外は 0 を指定してください。

See:

- wolfSSL_dtls_get_using_nonblock
- wolfSSL_dtls_got_timeout
- wolfSSL_dtls_get_current_timeout

Return: なし

Example

```
WOLFSSL* ssl = 0;
...
wolfSSL_dtls_set_using_nonblock(ssl, 1);
```

C.52.2.98 function wolfSSL_dtls_get_using_nonblock

```
int wolfSSL_dtls_get_using_nonblock(
    WOLFSSL * ssl
)
```

この関数は WOLFSSL DTLS オブジェクトが下層に UDP ノンブロッキング I/O を使用しているか否かを取得します。WOLFSSL オブジェクトがノンブロッキング I/O を使用している場合、この関数は 1 を返します。これにより、WOLFSSL オブジェクトは、EWOULDBLOCK を受信することを意味します。この機能は DTLS セッションにとってのみ意味があります。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ

See:

- wolfSSL_dtls_set_using_nonblock
- wolfSSL_dtls_got_timeout
- wolfSSL_dtls_set_using_nonblock

Return:

- 0 基礎となる I/O がブロックされています。
- 1 基礎となる I/O はノンブロッキングです。

Example

```
int ret = 0;
WOLFSSL* ssl = 0;
...
ret = wolfSSL_dtls_get_using_nonblock(ssl);
if (ret == 1) {
    // underlying I/O is non-blocking
}
...
```

C.52.2.99 function wolfSSL_dtls_get_current_timeout

```
int wolfSSL_dtls_get_current_timeout(
    WOLFSSL * ssl
)
```

この関数は現在のタイムアウト値を秒単位で返します。ノンブロッキングソケットを使用する場合、ユーザーコードでは、利用可能な recvV データの到着をチェックするタイミングや待つべき時間を知る必要があります。この関数によって返される値は、アプリケーションがどのくらい待機するかを示します。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ

See:

- wolfSSL_dtls
- wolfSSL_dtls_get_peer
- wolfSSL_dtls_got_timeout
- wolfSSL_dtls_set_peer

Return:

- seconds 現在の DTLS タイムアウト値（秒）
- NOT_COMPILED_IN wolfSSL が DTLS サポートで構築されていない場合。

Example

```
int timeout = 0;
WOLFSSL* ssl;
...
timeout = wolfSSL_get_dtls_current_timeout(ssl);
printf("DTLS timeout (sec) = %d\n", timeout);
```

C.52.2.100 function wolfSSL_dtls13_use_quick_timeout

```
int wolfSSL_dtls13_use_quick_timeout(
    WOLFSSL * ssl
)
```

この関数はアプリケーションがより早いタイムアウト時間を設定する必要がある場合に true を返します。ノンブロッキングソケットを使用する場合でユーザーコードで受信データが到着しているか何時チェックするか、あるいはどのくらいの時間待てばよいのかを決める必要があります。この関数が true を返した場合、ライブラリはすでに通信の中断を検出しましたが、他のピアからのメッセージがまだ送信中の場合に備えて、もう少し待機する必要があることを意味します。このタイマーの値を微調整するのはアプリケーション次第ですが、dtls_get_current_timeout()/4 が最適です。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ

See:

- wolfSSL_dtls
- wolfSSL_dtls_get_peer
- wolfSSL_dtls_got_timeout
- wolfSSL_dtls_set_peer
- wolfSSL_dtls13_set_send_more_acks

Return: true アプリケーションがより早いタイムアウトを設定する必要がある場合に返されます。

C.52.2.101 function wolfSSL_dtls13_set_send_more_acks

```
void wolfSSL_dtls13_set_send_more_acks(
    WOLFSSL * ssl,
    int value
)
```

この関数は、ライブラリが中断を検出したときにすぐに他のピアに ACK を送信するかどうかを設定します。ACK をすぐに送信すると、遅延は最小限に抑えられますが、必要以上に多くの帯域幅が消費される可能性があります。アプリケーションが独自にタイマーを管理しており、このオプションが 0 に設定されている場合、アプリケーションコードは wolfSSL_dtls13_use_quick_timeout() を使用して、遅延した ACK を送信するためにより速いタイムアウトを設定する必要があるかどうかを判断できます。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ
- **value** 設定を行う場合には 1 を行わない場合には 0 を設定します。

See:

- wolfSSL_dtls

- wolfSSL_dtls_get_peer
- wolfSSL_dtls_got_timeout
- wolfSSL_dtls_set_peer
- wolfSSL_dtls13_use_quick_timeout

C.52.2.102 function wolfSSL_dtls_set_timeout_init

```
int wolfSSL_dtls_set_timeout_init(
    WOLFSSL * ssl,
    int
)
```

この関数は DTLS タイムアウトを設定します。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ
- **value** タイムアウトオプションを有効にする場合には 1 を指定し、無効にする場合には 0 を指定します。

See:

- wolfSSL_dtls_set_timeout_max
- wolfSSL_dtls_got_timeout

Return:

- SSL_SUCCESS 関数がエラーなしで実行された場合に返されます。SSL の DTLS_TIMEOUT_INIT と DTLS_TIMEOUT メンバーが設定されています。
- BAD_FUNC_ARG 引数 ssl が NULL の場合、またはタイムアウトが 0 以下の場合に返されます。タイムアウト引数が許可されている最大値を超えている場合にも返されます。

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.52.2.103 function wolfSSL_dtls_set_timeout_max

```
int wolfSSL_dtls_set_timeout_max(
    WOLFSSL * ssl,
    int
)
```

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ
- **timeout** 最大タイムアウト時間

See:

- wolfSSL_dtls_set_timeout_init
- wolfSSL_dtls_got_timeout

Return:

- SSL_SUCCESS 関数がエラーなしで実行された場合に返されます。
- BAD_FUNC_ARG wolfssl 構造体が NULL の場合、または TIMEOUT 引数がゼロ以下である場合、または WolfSSL 構造体の DTLS_TIMEOUT_INIT メンバーよりも小さい場合に返されます。

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.52.2.104 function wolfSSL_dtls_got_timeout

```
int wolfSSL_dtls_got_timeout(
    WOLFSSL * ssl
)
```

DTLS でノンブロッキングソケットを使用する場合、この関数は送信がタイムアウトしたと考えられる場合に呼び出される必要があります。タイムアウト値の調整など、最後の送信を再試行するために必要なアクションを実行します。時間がかかりすぎると、失敗が返されます。

See:

- wolfSSL_dtls_get_current_timeout
- wolfSSL_dtls_get_peer
- wolfSSL_dtls_set_peer
- wolfSSL_dtls

Return:

- SSL_SUCCESS 成功時に戻ります
- SSL_FATAL_ERROR ピアからの応答を得ることなく、再送信/タイムアウトが多すぎる場合に返されます。
- NOT_COMPILED_IN wolfSSL が DTLS サポートでコンパイルされていない場合に返されます。

Example

```
See the following files for usage examples:
<wolfssl_root>/examples/client/client.c
<wolfssl_root>/examples/server/server.c
```

C.52.2.105 function wolfSSL_dtls_retransmit

```
int wolfSSL_dtls_retransmit(
    WOLFSSL * ssl
)
```

DTLS でノンブロッキングソケットを使用する場合、この関数は予想されるタイムアウト値と再送信回数を無視して最後のハンドシェイクフライトを再送信します。これは、DTLS を使用しており、タイムアウトや再試行回数も管理する必要があるアプリケーションに役立ちます。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ

See:

- wolfSSL_dtls_get_current_timeout
- wolfSSL_dtls_got_timeout
- wolfSSL_dtls

Return:

- SSL_SUCCESS 成功時に戻ります
- SSL_FATAL_ERROR ピアからの応答が得られないまま再送信/タイムアウトが多すぎる場合に返されます。

Example

```
int ret = 0;
WOLFSSL* ssl;
...
ret = wolfSSL_dtls_retransmit(ssl);
```

C.52.2.106 function wolfSSL_dtls

```
int wolfSSL_dtls(
    WOLFSSL * ssl
)
```

DTLS を使用するように構成されているかどうかを取得します。

See:

- wolfSSL_dtls_get_current_timeout
- wolfSSL_dtls_get_peer
- wolfSSL_dtls_got_timeout
- wolfSSL_dtls_set_peer

Return:

- 1 SSL セッション（SSL）が DTLS を使用するように設定されている場合、この関数は 1 を返します。
- 0 そうでない場合に返されます。

Example

```
int ret = 0;
WOLFSSL* ssl;
...
ret = wolfSSL_dtls(ssl);
if (ret) {
    // SSL session has been configured to use DTLS
}
```

C.52.2.107 function wolfSSL_dtls_set_peer

```
int wolfSSL_dtls_set_peer(
    WOLFSSL * ssl,
    void * peer,
    unsigned int peerSz
)
```

この関数は引数 peer で与えられるアドレスを DTLS のピアとしてセットします。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ

- **peer** ピアのアドレスを含む sockaddr_in 構造体へのポインタ。
- **peerSz** sockaddr_in 構造体のサイズ。0 が指定された場合には ssl に設定されているピアの情報をクリアします。

See:

- wolfSSL_dtls_get_current_timeout
- wolfSSL_dtls_get_peer
- wolfSSL_dtls_got_timeout
- wolfSSL_dtls

Return:

- SSL_SUCCESS 成功時に返されます。
- SSL_FAILURE 失敗時に返されます。
- SSL_NOT_IMPLEMENTED wolfSSL が DTLS をサポートするようにコンパイルされていない場合に返されます。

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.52.2.108 function wolfSSL_dtls_get_peer

```
int wolfSSL_dtls_get_peer(
    WOLFSSL * ssl,
    void * peer,
    unsigned int * peerSz
)
```

この関数は、現在の DTLS ピアの sockaddr_in(サイズ peerSz) を取得します。この関数は、peerSz を SSL セッションに保存されている実際の DTLS ピアサイズと比較します。ピアアドレスが peer に収まる場合は、peerSz がピアのサイズに設定されて、ピアの sockaddr_in が peer にコピーされます。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ
- **peer** ピアの sockaddr_in 構造体を保存するためのバッファへのポインタ。
- **peerSz** サイズを格納する変数。入力時には引数 peer で示されるバッファのサイズを指定してください。出力時には実際の sockaddr_in 構造体のサイズを返します。

See:

- wolfSSL_dtls_get_current_timeout
- wolfSSL_dtls_got_timeout
- wolfSSL_dtls_set_peer
- wolfSSL_dtls

Return:

- SSL_SUCCESS 成功時に返されます。
- SSL_FAILURE 失敗時に返されます。
- SSL_NOT_IMPLEMENTED wolfSSL が DTLS をサポートするようにコンパイルされていない場合に返されます。

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.52.2.109 function wolfSSL_ERR_error_string

```
char * wolfSSL_ERR_error_string(
    unsigned long errNumber,
    char * data
)
```

この関数は、wolfSSL_get_error() によって返されたエラーコードをより人間が読めるエラー文字列に変換します。引数 errNumber は、wolfSSL_get_error() によって返され、引数 data はエラー文字列が配置されるバッファへのポインタです。MAX_ERROR_SZ で定義されているように、データの最大長はデフォルトで 80 文字です。これは wolfssl/wolfcrypt/error.h で定義されています。

Parameters:

- **errNumber** wolfSSL_get_error()によって返されたエラーコード。
- **data** 人間が読めるエラー文字列を格納したバッファへのポインタ

See:

- wolfSSL_get_error
- wolfSSL_ERR_error_string_n
- wolfSSL_ERR_print_errors_fp
- wolfSSL_load_error_strings

Return:

- success 正常に完了すると、この関数は data に返されるのと同じ文字列を返します。
- failure 失敗すると、この関数は適切な障害理由、MSG を持つ文字列を返します。

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.52.2.110 function wolfSSL_ERR_error_string_n

```
void wolfSSL_ERR_error_string_n(
    unsigned long e,
    char * buf,
    unsigned long sz
)
```

この関数は、wolfssl_err_error_string() のバッファのサイズを指定するバージョンです。ここで、引数 len は引数 buf に書き込まれ得る最大文字数を指定します。wolfSSL_err_error_string() と同様に、この関数は

wolfSSL_get_error() から返されたエラーコードをより人間が読めるエラー文字列に変換します。人間が読める文字列は buf に置かれます。

Parameters:

- **e** wolfSSL_get_error()によって返されたエラーコード。
- **buff** e と一致する人間が読めるエラー文字列を含む出力バッファ。
- **len** 出力バッファのサイズ

See:

- wolfSSL_get_error
- wolfSSL_ERR_error_string
- wolfSSL_ERR_print_errors_fp
- wolfSSL_load_error_strings

Return: なし

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.52.2.111 function wolfSSL_get_shutdown

```
int wolfSSL_get_shutdown(
    const WOLFSSL * ssl
)
```

この関数は、Options 構造体の closeNotify または connReset または sentNotify メンバーのシャットダウン条件をチェックします。Options 構造体は WOLFSSL 構造体内にあります。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ

See: wolfSSL_SESSION_free

Return:

- 1 SSL_SENT_SHUTDOWN が返されます。
- 2 SSL_RECEIVED_SHUTDOWN が返されます。

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.52.2.112 function wolfSSL_session_reused

```
int wolfSSL_session_reused(
    WOLFSSL * ssl
)
```

この関数は、オプション構造体の再開メンバを返します。フラグはセッションを再利用するかどうかを示します。そうでなければ、新しいセッションを確立する必要があります。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ

See:

- wolfSSL_SESSION_free
- wolfSSL_GetSessionIndex
- wolfSSL_memsave_session_cache

Return: This 関数セッションの再利用のフラグを表すオプション構造に保持されている int 型を返します。

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
...
if(!wolfSSL_session_reused(sslResume)){
    // No session reuse allowed.
}
```

C.52.2.113 function wolfSSL_is_init_finished

```
int wolfSSL_is_init_finished(
    WOLFSSL * ssl
)
```

この関数は、接続が確立されているかどうかを確認します。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ

See:

- wolfSSL_set_accept_state
- wolfSSL_get_keys
- wolfSSL_set_shutdown

Return:

- 0 接続が確立されていない場合、すなわち WolfSSL 構造体が NULL またはハンドシェイクが行われていない場合に返されます。
- 1 接続が確立されていない場合は返されます.WolfSSL 構造体は NULL またはハンドシェイクが行われていません。

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.52.2.114 function wolfSSL_get_version

```
const char * wolfSSL_get_version(
    WOLFSSL * ssl
)
```

文字列として使用されている SSL バージョンを返します。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ

See: wolfSSL_lib_version

Return:

- “SSLv3” SSLv3 を使う
- “TLSv1” TLSV1 を使用する
- “TLSv1.1” TLSV1.1 を使用する
- “TLSv1.2” TLSV1.2 を使用する
- “TLSv1.3” TLSV1.3 を使用する
- “DTLS”: DTLS を使う
- “DTLSv1.2” DTLSV1.2 を使用する
- “unknown” どのバージョンの TLS が使用されているかを判断するという問題がありました。

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.52.2.115 function wolfSSL_get_current_cipher_suite

```
int wolfSSL_get_current_cipher_suite(
    WOLFSSL * ssl
)
```

SSL セッションで現在の暗号スイートを返します。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ

See:

- wolfSSL_CIPHER_get_name
- wolfSSL_get_current_cipher
- wolfSSL_get_cipher_list

Return:

- ssl->options.cipherSuite 現在の暗号スイートを表す整数。

- 0 提供されている SSL セッションは NULL です。

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.52.2.116 function wolfSSL_get_current_cipher

```
WOLFSSL_CIPHER * wolfSSL_get_current_cipher(
    WOLFSSL * ssl
)
```

この関数は、SSL セッションの現在の暗号へのポインタを返します。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ

See:

- wolfSSL_get_cipher
- wolfSSL_get_cipher_name_internal
- wolfSSL_get_cipher_name

Return:

- The 関数 WolfSSL 構造体の暗号メンバーのアドレスを返します。これは wolfssl_icipher 構造へのポインタです。
- NULL WolfSSL 構造が NULL の場合に返されます。

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.52.2.117 function wolfSSL_CIPHER_get_name

```
const char * wolfSSL_CIPHER_get_name(
    const WOLFSSL_CIPHER * cipher
)
```

この関数は、SSL オブジェクト内の Cipher Suite と使用可能なスイートと一致し、文字列表現を返します。

Parameters:

- **cipher** WOLFSSL_CIPHER 構造体へのポインタ

See:

- wolfSSL_get_cipher
- wolfSSL_get_current_cipher
- wolfSSL_get_cipher_name_internal
- wolfSSL_get_cipher_name

Return:

- string この関数は、一致した暗号スイートの文字列表現を返します。
- none スイートが一致していない場合は「なし」を返します。

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.52.2.118 function wolfSSL_get_cipher

```
const char * wolfSSL_get_cipher(
    WOLFSSL * ssl
)
```

この関数は、SSL オブジェクト内の暗号スイートと使用可能なスイートと一致します。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ

See:

- wolfSSL_CIPHER_get_name
- wolfSSL_get_current_cipher

Return: This 関数 Suite が一致させた String 値を返します。スイートが一致していない場合は「なし」を返します。

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.52.2.119 function wolfSSL_get1_session

```
WOLFSSL_SESSION * wolfSSL_get1_session(
    WOLFSSL * ssl
)
```

この関数は、WOLFSSL 構造体から WOLFSSL_SESSION を参照型として返します。これには、wolfSSL_SESSION_free を呼び出してセッション参照を解除する必要があります。WOLFSSL_SESSION は、セッションの再開を実行するために必要なすべての必要な情報を含み、新しいハンドシェイクなしで接続を再確立します。セッションの再開の場合、wolfSSL_shutdown() をセッションオブジェクトに呼び出す前に、アプリケーションはオブジェクトから wolfssl_get1_session() を呼び出して保存する必要があります。これはセッションへのポインタを返します。その後、アプリケーションは新しい WOLFSSL オブジェクトを作成し、保存したセッションを wolfssl_set_session() に割り当てる必要があります。この時点で、アプリケーションは wolfssl_connect() を呼び出し、WolfSSL はセッションを再開しようとします。WolfSSL サーバーコードでは、デフォルトでセッションの再開を許可します。wolfssl_get1_session() によって返されたオブジェクトは、アプリケーションが使用後は解放される必要があります。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ

See:

- wolfSSL_new
- wolfSSL_free
- wolfSSL_SESSION_free

Return:

- WOLFSSL_SESSION 成功の場合はセッションポインタを返します。
- NULL ssl が NULL の場合、SSL セッションキャッシュが無効になっている場合、WolfSSL はセッション ID を使用できない、またはミューテックス関数が失敗します。

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.52.2.120 function wolfSSLv23_client_method

```
WOLFSSL_METHOD * wolfSSLv23_client_method(
    void
)
```

wolfsslv23_client_method() 関数は、アプリケーションがクライアントであることを示すために使用され、SSL 3.0~TLS 1.3 の間でサーバーでサポートされている最高のプロトコルバージョンをサポートします。この関数は、wolfSSL_CTX_new() を使用して SSL / TLS コンテキストを作成するときに使用される新しい Wolfssl_method 構造体のメモリを割り当てて初期化します。WolfSSL クライアントとサーバーの両方が堅

牢なバージョンのダウングレード機能を持っています。特定のプロトコルバージョンメソッドがどちらの側で使用されている場合は、そのバージョンのみがネゴシエートされたり、エラーが返されます。たとえば、TLSV1 を使用し、SSLv3 のみに接続しようとするクライアントは、TLSV1.1 に接続しても失敗します。この問題を解決するために、wolfsslv23_client_method() 関数を使用するクライアントは、サーバーでサポートされている最高のプロトコルバージョンを使用し、必要に応じて SSLv3 にダウングレードします。この場合、クライアントは SSLv3 - TLSv1.3 を実行しているサーバーに接続できるようになります。

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 成功すると、wolfssl_method へのポインタが返されます。
- Failure xmalloc を呼び出すときにメモリ割り当てが失敗した場合、基礎となる Malloc() 実装の失敗値が返されます（通常は errno が ENOMEM に設定されます）。

Example

```
WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;
method = wolfSSLv23_client_method();
if (method == NULL) {
// unable to get method
}

ctx = wolfSSL_CTX_new(method);
...
```

C.52.2.121 function wolfSSL_BIO_get_mem_data

```
int wolfSSL_BIO_get_mem_data(
    WOLFSSL_BIO * bio,
    void * p
)
```

この関数は、内部メモリバッファの先頭へのバイトポインタを設定するために使用されます。

Parameters:

- **bio** のメモリバッファを取得するための WOLFSSL_BIO 構造体。
- **p** メモリバッファへのポインタ。

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_s_mem
- wolfSSL_BIO_set_fp
- wolfSSL_BIO_free

Return:

- size 成功すると、バッファのサイズが返されます
- SSL_FATAL_ERROR エラーケースに遭遇した場合

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.52.2.122 function wolfSSL_BIO_set_fd

```
long wolfSSL_BIO_set_fd(
    WOLFSSL_BIO * b,
    int fd,
    int flag
)
```

使用する BIO のファイル記述子を設定します。

Parameters:

- **bio** FD を設定するための WOLFSSL_BIO 構造。
- **fd** 使用するファイル記述子。
- **closeF** fd をクローズする際のふるまいを指定するフラグ

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_free

Return: SSL_SUCCESS(1) 成功時に返されます。

Example

```
WOLFSSL_BIO* bio;
int fd;
// setup bio
wolfSSL_BIO_set_fd(bio, fd, BIO_NOCLOSE);
```

C.52.2.123 function wolfSSL_BIO_set_close

```
int wolfSSL_BIO_set_close(
    WOLFSSL_BIO * b,
    long flag
)
```

BIO が解放されたときに I/O ストリームを閉じる必要があることを示すために使用されるクローズフラグを設定します。

Parameters:

- **bio** WOLFSSL_BIO 構造体。
- **flag** I/O ストリームを閉じる必要があることを示すために使用されるクローズフラグ

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_free

Return: SSL_SUCCESS(1) 成功時に返されます。

Example

```
WOLFSSL_BIO* bio;
// setup bio
wolfSSL_BIO_set_close(bio, BIO_NOCLOSE);
```

C.52.2.124 function wolfSSL_BIO_s_socket

```
WOLFSSL_BIO_METHOD * wolfSSL_BIO_s_socket(
    void
)
```

この関数は BIO_SOCKET タイプの WOLFSSL_BIO_METHOD を取得するために使用されます。

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_s_mem

Return: WOLFSSL_BIO_METHOD ソケットタイプである WOLFSSL_BIO_METHOD 構造体へのポインタ

Example

```
WOLFSSL_BIO* bio;
bio = wolfSSL_BIO_new(wolfSSL_BIO_s_socket);
```

C.52.2.125 function wolfSSL_BIO_set_write_buf_size

```
int wolfSSL_BIO_set_write_buf_size(
    WOLFSSL_BIO * b,
    long size
)
```

この関数は、WOLFSSL_BIO のライトバッファのサイズを設定するために使用されます。書き込みバッファが以前に設定されている場合、この関数はサイズをリセットするときに解放されます。読み書きインデックスを 0 にリセットするという点で、wolfSSL_BIO_reset に似ています。

Parameters:

- **bio** FD を設定するための WOLFSSL_BIO 構造。
- **size** バッファサイズ

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_s_mem
- wolfSSL_BIO_free

Return:

- SSL_SUCCESS 書き込みバッファの設定に成功しました。
- SSL_FAILURE エラーケースに遭遇した場合

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.52.2.126 function wolfSSL_BIO_make_bio_pair

```
int wolfSSL_BIO_make_bio_pair(
    WOLFSSL_BIO * b1,
    WOLFSSL_BIO * b2
)
```

これは 2 つの BIOS を一緒にペアリングするために使用されます。一対の BIOS は、2 つの方法パイプと同様に、他方で読み取られることができ、その逆も同様である。BIOS の両方が同じスレッド内にあることが予想されます。この機能はスレッドセーフではありません。2 つの BIOS のうちの 1 つを解放すると、両方ともペアになっています。書き込みバッファサイズが以前に設定されていない場合、それはペアになる前に 17000（wolfssl_bio_size）のデフォルトサイズに設定されます。

Parameters:

- **b1** ペアを設定するための第一の WOLFSSL_BIO 構造体へのポインタ。
- **b2** 第二のの WOLFSSL_BIO 構造体へのポインタ。

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_s_mem
- wolfSSL_BIO_free

Return:

- SSL_SUCCESS 2 つの BIOS をうまくペアリングします。
- SSL_FAILURE エラーケースに遭遇した場合

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.52.2.127 function wolfSSL_BIO_ctrl_reset_read_request

```
int wolfSSL_BIO_ctrl_reset_read_request(
    WOLFSSL_BIO * bio
)
```

この関数は、読み取り要求フラグを 0 に戻すために使用されます。

Parameters:

- **bio** WOLFSSL_BIO 構造体へのポインタ。

See:

- wolfSSL_BIO_new, wolfSSL_BIO_s_mem
- wolfSSL_BIO_new, wolfSSL_BIO_free

Return:

- SSL_SUCCESS 値を正常に設定します。
- SSL_FAILURE エラーケースに遭遇した場合

Example

```
WOLFSSL_BIO* bio;
int ret;
...
ret = wolfSSL_BIO_ctrl_reset_read_request(bio);
// check ret value
```

C.52.2.128 function wolfSSL_BIO_nread0

```
int wolfSSL_BIO_nread0(
    WOLFSSL_BIO * bio,
    char ** buf
)
```

Parameters:

- **bio** WOLFSSL_BIO 構造体へのポインタ。
- **buf** 読み取り用バッファへのポインタのポインタ

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_nwrite0

Return: >=0 成功すると、読み取るバイト数を返します

bri f この関数は、読み取り用のバッファポインタを取得するために使用されます。wolfSSL_BIO_nread とは異なり、内部読み取りインデックスは関数呼び出しから返されたサイズ分進みません。返される値を超えて読み取ると、アレイの境界から読み出される可能性があります。*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.52.2.129 function wolfSSL_BIO_nread

```
int wolfSSL_BIO_nread(
    WOLFSSL_BIO * bio,
    char ** buf,
    int num
)
```

Parameters:

- **bio** WOLFSSL_BIO 構造体へのポインタ。
- **buf** 読み取り配列の先頭に設定するポインタ。
- **num** 読み取りサイズ

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_nwrite

Return:

- =0 成功すると、読み取るバイト数を返します
- WOLFSSL_BIO_ERROR(-1) Return -1 を読むものではないエラーケースについて

biie れは、この関数は、読み取り用のバッファポインタを取得するために使用されます。内部読み取りインデックスは、読み取り元のバッファの先頭に指されている BUF を使用して、関数呼び出しから返されるサイズ分進みます。数 num で要求された値よりもバイトが少ない場合、より少ない値が返されます。返される値を超えて読み取ると、アレイの境界から読み出される可能性があります。*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.52.2.130 function wolfSSL_BIO_nwrite

```
int wolfSSL_BIO_nwrite(
    WOLFSSL_BIO * bio,
    char ** buf,
    int num
)
```

関数によって返される数のバイトを書き込むためにバッファーへのポインタを取得します。返されるポインタに追加のバイトを書き込んだ場合、返された値は範囲外の書き込みにつながる可能性があります。

Parameters:

- **bio** WOLFSSL_BIO 構造に書き込む構造。
- **buf** 書き込むためのバッファへのポインタ。
- **num** 書き込みたいサイズ

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_free
- wolfSSL_BIO_nread

Return:

- int 返されたバッファポインタに書き込むことができるバイト数を返します。
- WOLFSSL_BIO_UNSET(-2) バイオペアの一部ではない場合
- WOLFSSL_BIO_ERROR(-1) に書くべき部屋がこれ以上ない場合

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.52.2.131 function wolfSSL_BIO_reset

```
int wolfSSL_BIO_reset(
    WOLFSSL_BIO * bio
)
```

バイオを初期状態にリセットします。タイプ BIO_BIO の例として、これは読み書きインデックスをリセットします。

Parameters:

- **bio** WOLFSSL_BIO 構造体へのポインタ。

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_free

Return:

- 0 バイオのリセットに成功しました。
- WOLFSSL_BIO_ERROR(-1) 不良入力または失敗したリセットで返されます。

Example

```
WOLFSSL_BIO* bio;
// setup bio
wolfSSL_BIO_reset(bio);
//use pt
```

C.52.2.132 function wolfSSL_BIO_seek

```
int wolfSSL_BIO_seek(
    WOLFSSL_BIO * bio,
    int ofs
)
```

この関数は、指定されたオフセットへファイルポインタを調整します。これはファイルの先頭からのオフセットです。

Parameters:

- **bio** 設定する WOLFSSL_BIO 構造体へのポインタ。
- **ofs** ファイルの先頭からのオフセット

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_s_mem
- wolfSSL_BIO_set_fp
- wolfSSL_BIO_free

Return:

- 0 正常に探しています。
- -1 エラーケースに遭遇した場合

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.52.2.133 function wolfSSL_BIO_write_filename

```
int wolfSSL_BIO_write_filename(
    WOLFSSL_BIO * bio,
    char * name
)
```

これはファイルに設定および書き込むために使用されます。現在ファイル内のデータを上書きし、BIO が解放されたときにファイルを閉じるように設定されます。

Parameters:

- **bio** ファイルを設定する WOLFSSL_BIO 構造体体。
- **name** 書き込み先ファイル名へのポインタ

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_s_file
- wolfSSL_BIO_set_fp
- wolfSSL_BIO_free

Return:

- SSL_SUCCESS ファイルの開きと設定に成功しました。
- SSL_FAILURE エラーケースに遭遇した場合

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.52.2.134 function wolfSSL_BIO_set_mem_eof_return

```
long wolfSSL_BIO_set_mem_eof_return(
    WOLFSSL_BIO * bio,
    int v
)
```

これはファイル値の終わりを設定するために使用されます。一般的な値は予想される正の値と混同されないように-1 です。

Parameters:

- **bio** ファイル値の終わりを設定するための WOLFSSL_BIO 構造体体。
- **v** bio にセットする値。

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_s_mem
- wolfSSL_BIO_set_fp
- wolfSSL_BIO_free

Return: 0 完了に戻りました

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.52.2.135 function wolfSSL_BIO_get_mem_ptr

```
long wolfSSL_BIO_get_mem_ptr(
    WOLFSSL_BIO * bio,
    WOLFSSL_BUF_MEM ** m
)
```

これは WolfSSL_BIO メモリポインタのゲッター関数です。

Parameters:

- **bio** メモリポインタを取得するための WOLFSSL_BIO 構造体へのポインタ。
- **ptr** WOLFSSL_BUF_MEM 構造体へのポインタ（現在は char* となっている）

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_s_mem

Return:

- SSL_SUCCESS ポインタ SSL_SUCCESS を返す正常に（現在 1 の値）。
- SSL_FAILURE null 引数が渡された場合（現在 0 の値）に渡された場合に返されます。

Example

```
WOLFSSL_BIO* bio;
WOLFSSL_BUF_MEM* pt;
// setup bio
wolfSSL_BIO_get_mem_ptr(bio, &pt);
//use pt
```

C.52.2.136 function wolfSSL_X509_NAME_oneline

```
char * wolfSSL_X509_NAME_oneline(
    WOLFSSL_X509_NAME * name,
    char * in,
    int sz
)
```

この関数は X509 の名前をバッファにコピーします。

Parameters:

- **name** wolfssl_x509 構造へのポインタ。
- **in** WOLFSSL_X509_NAME 構造体からコピーされた名前を保持するためのバッファ。
- **sz** バッファの最大サイズ

See:

- wolfSSL_X509_get_subject_name
- wolfSSL_X509_get_issuer_name
- wolfSSL_X509_get_isCA
- wolfSSL_get_peer_certificate
- wolfSSL_X509_version

Return: A WOLFSSL_X509_NAME 構造名メンバーのデータが正常に実行された場合、name メンバーのデータが返されます。

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.52.2.137 function wolfSSL_X509_get_issuer_name

```
WOLFSSL_X509_NAME * wolfSSL_X509_get_issuer_name(
    WOLFSSL_X509 * cert
)
```

この関数は証明書発行者の名前を返します。

Parameters:

- **cert** WOLFSSL_X509 構造体へのポインタ

See:

- wolfSSL_X509_get_subject_name
- wolfSSL_X509_get_isCA
- wolfSSL_get_peer_certificate
- wolfSSL_X509_NAME_oneline

Return:

- point WOLFSSL_X509 構造体の発行者メンバーへのポインタが返されます。
- NULL 渡された証明書が NULL の場合

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.52.2.138 function wolfSSL_X509_get_subject_name

```
WOLFSSL_X509_NAME * wolfSSL_X509_get_subject_name(
    WOLFSSL_X509 * cert
)
```

この関数は、wolfssl_x509 構造の件名メンバーを返します。

Parameters:

- **cert** WOLFSSL_X509 構造体へのポインタ

See:

- wolfSSL_X509_get_issuer_name
- wolfSSL_X509_get_isCA
- wolfSSL_get_peer_certificate

Return: pointer wolfssl_x509_name 構造へのポインタ。WOLFSSL_X509 構造体が NULL の場合、または構造体の件名メンバーが NULL の場合、ポインタは NULL になることがあります。

Example

```
WOLFSSL_X509* cert;
WOLFSSL_X509_NAME name;
...
name = wolfSSL_X509_get_subject_name(cert);
if(name == NULL){
    // Deal with the NULL cacse
}
```

C.52.2.139 function wolfSSL_X509_get_isCA

```
int wolfSSL_X509_get_isCA(
    WOLFSSL_X509 * cert
)
```

WOLFSSL_X509 構造体の isCa メンバーをチェックして値を返します。

Parameters:

- **cert** WOLFSSL_X509 構造体へのポインタ

See:

- wolfSSL_X509_get_issuer_name
- wolfSSL_X509_get_isCA

Return:

- isCA WOLFSSL_X509 構造体の isCa メンバーの値を返します。
- 0 有効な WOLFSSL_X509 構造体が渡されない場合に返されます。

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.52.2.140 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
)
```

この関数は、渡された NID 値に関連するテキストを取得します。

Parameters:

- **name** wolfssl_x509_name テキストを検索する。
- **nid** 検索する NID。
- **buf** 見つかったときにテキストを保持するためのバッファー。
- **len** バッファのサイズ

See: none

Return: int テキストバッファのサイズを返します。

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.52.2.141 function wolfSSL_X509_get_signature_type

```
int wolfSSL_X509_get_signature_type(
    WOLFSSL_X509 * cert
)
```

この関数は、WOLFSSL_X509 構造体の sigOID メンバーに格納されている値を返します。

Parameters:

- **cert** WOLFSSL_X509 構造体へのポインタ

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 WOLFSSL_X509 構造体が NULL の場合に返されます。
- int x509 オブジェクトから取得された整数値が返されます。

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.52.2.142 function wolfSSL_X509_free

```
void wolfSSL_X509_free(
    WOLFSSL_X509 * x509
)
```

この関数は WOLFSSL_X509 構造体を解放します。

Parameters:

- **x509** WOLFSSL_X509 構造体へのポインタ

See:

- wolfSSL_X509_get_signature
- wolfSSL_X509_version
- wolfSSL_X509_get_der
- wolfSSL_X509_get_serial_number
- wolfSSL_X509_notBefore
- wolfSSL_X509_notAfter

Return: なし

Example

```
WOLFSSL_X509* x509 = (WOLFSSL_X509*)XMALOC(sizeof(WOLFSSL_X509), NULL,
DYNAMIC_TYPE_X509) ;

wolfSSL_X509_free(x509);
```

C.52.2.143 function wolfSSL_X509_get_signature

```
int wolfSSL_X509_get_signature(
    WOLFSSL_X509 * x509,
    unsigned char * buf,
    int * bufSz
)
```

x509 署名を取得し、それをバッファに保存します。

Parameters:

- **x509** wolfssl_x509 構造へのポインタ。
- **buf** バッファへの文字ポインタ。
- **bufSz** バッファサイズを格納する int 型変数へのポインタ

See:

- wolfSSL_X509_get_serial_number
- wolfSSL_X509_get_signature_type
- wolfSSL_X509_get_device_type

Return:

- SSL_SUCCESS 関数が正常に実行された場合に返されます。署名がバッファにロードされます。
- SSL_FATAL_ERRROR X509 構造体または BUFSZ メンバーが NULL の場合に返します。SIG 構造の長さメンバのチェックもある（SIG は X509 のメンバーである）。

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.52.2.144 function wolfSSL_X509_STORE_add_cert

```
int wolfSSL_X509_STORE_add_cert(
    WOLFSSL_X509_STORE * store,
    WOLFSSL_X509 * x509
)
```

この関数は、WOLFSSL_X509_STRE 構造体に証明書を追加します。

Parameters:

- **str** 証明書を追加する証明書ストア。
- **x509** 追加する WOLFSSL_X509 構造体へのポインタ

See: wolfSSL_X509_free

Return:

- SSL_SUCCESS 証明書が正常に追加された場合。
- SSL_FATAL_ERROR: 証明書が正常に追加されない場合

Example

```
WOLFSSL_X509_STORE* str;
WOLFSSL_X509* x509;
int ret;
ret = wolfSSL_X509_STORE_add_cert(str, x509);
//check ret value
```

C.52.2.145 function wolfSSL_X509_STORE_CTX_get_chain

```
WOLFSSL_STACK * wolfSSL_X509_STORE_CTX_get_chain(
    WOLFSSL_X509_STORE_CTX * ctx
)
```

この関数は、WOLFSSL_X509_STORE_CTX 構造体のチェーン変数の getter 関数です。現在チェーンは取り込まれていません。

Parameters:

- **ctx** WOLFSSL_X509_STORE_CTX 構造体へのポインタ

See: wolfSSL_sk_X509_free

Return:

- pointer 成功した場合 WOLFSSL_STACK（STACK_OF(WOLFSSL_X509)）ポインタと同じ
- Null 失敗した場合に返されます。

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.52.2.146 function wolfSSL_X509_STORE_set_flags

```
int wolfSSL_X509_STORE_set_flags(
    WOLFSSL_X509_STORE * store,
    unsigned long flag
)
```

この関数は、渡された WOLFSSL_X509_STORE 構造体の動作を変更するためのフラグを取ります。使用されるフラグの例は WOLFSSL_CRL_CHECK です。

Parameters:

- **str** フラグを設定する証明書ストア。
- **flag** フラグ

See:

- wolfSSL_X509_STORE_new
- wolfSSL_X509_STORE_free

Return:

- SSL_SUCCESS フラグを設定するときにエラーが発生しなかった場合。
- <0 障害の際に負の値が返されます。

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.52.2.147 function wolfSSL_X509_notBefore

```
const byte * wolfSSL_X509_notBefore(
    WOLFSSL_X509 * x509
)
```

この関数は BYTE アレイとして符号化された “not before” 要素を返します。

Parameters:

- **x509** WOLFSSL_X509 構造体へのポインタ。

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 WOLFSSL_X509 構造体が NULL の場合に返されます。
- byte NetBeforEdata を含むバッファへのポインタが返されます。

Example

```
WOLFSSL_X509* x509 = (WOLFSSL_X509*)XMALLOC(sizeof(WOLFSSL_X509), NULL,
                        DYNAMIC_TYPE_X509);
...
byte* notBeforeData = wolfSSL_X509_notBefore(x509);
```

C.52.2.148 function wolfSSL_X509_notAfter

```
const byte * wolfSSL_X509_notAfter(
    WOLFSSL_X509 * x509
)
```

この関数は、BYTE 配列として符号化された “not after” 要素を返します。

Parameters:

- **x509** WOLFSSL_X509 構造体へのポインタ。

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 WOLFSSL_X509 構造体が NULL の場合に返されます。
- byte notAfterData を含むバッファへのポインタが返されます。

Example

```
WOLFSSL_X509* x509 = (WOLFSSL_X509*)XMALLOC(sizeof(WOLFSSL_X509), NULL,
                        DYNAMIC_TYPE_X509);
...
byte* notAfterData = wolfSSL_X509_notAfter(x509);
```

C.52.2.149 function wolfSSL_ASN1_INTEGER_to_BN

```
WOLFSSL_BIGNUM * wolfSSL_ASN1_INTEGER_to_BN(
    const WOLFSSL_ASN1_INTEGER * ai,
    WOLFSSL_BIGNUM * bn
)
```

この関数は、WOLFSSL_ASN1_INTEGER 値を WOLFSSL_BIGNUM 構造体にコピーするために使用されます。

Parameters:

- **ai** WOLFSSL_ASN1_INTEGER 構造体へのポインタ
- **bn** もし、既存の WOLFSSL_BIGNUM 構造体にコピーしたい場合そのポインタをこの引数で指定します。NULL を指定すると新たに WOLFSSL_BIGNUM 構造体が生成されて使用されます。

See: none

Return:

- pointer WOLFSSL_ASN1_INTEGER 値を正常にコピーすると、WOLFSSL_BIGNUM ポインタが返されます。
- Null 失敗時に返されます。

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.52.2.150 function wolfSSL_CTX_add_extra_chain_cert

```
long wolfSSL_CTX_add_extra_chain_cert(
    WOLFSSL_CTX * ctx,
    WOLFSSL_X509 * x509
)
```

この関数は、WOLFSSL_CTX 構造で構築されている内部チェーンに証明書を追加します。

Parameters:

- **ctx** 証明書を追加するための WOLFSSL_CTX 構造。
- **x509** WOLFSSL_X509 構造体へのポインタ。

See:

- wolfSSL_CTX_new
- wolfSSL_CTX_free

Return:

- SSL_SUCCESS 証明書の追加に成功したら。
- SSL_FAILURE チェーンに証明書を追加することが失敗した場合。

Example

```
WOLFSSL_CTX* ctx;
WOLFSSL_X509* x509;
int ret;
// create ctx
ret = wolfSSL_CTX_add_extra_chain_cert(ctx, x509);
// check ret value
```

C.52.2.151 function wolfSSL_CTX_get_read_ahead

```
int wolfSSL_CTX_get_read_ahead(
    WOLFSSL_CTX * ctx
)
```

この関数は、WOLFSSL_CTX 構造から Get Read Hape フラグを返します。

Parameters:

- **ctx** WOLFSSL_CTX 構造体へのポインタ

See:

- wolfSSL_CTX_new
- wolfSSL_CTX_free
- wolfSSL_CTX_set_read_ahead

Return:

- flag 成功すると、読み取り先のフラグを返します。
- SSL_FAILURE ctx が null の場合、ssl_failure が返されます。

Example

```
WOLFSSL_CTX* ctx;
int flag;
// setup ctx
flag = wolfSSL_CTX_get_read_ahead(ctx);
//check flag
```

C.52.2.152 function wolfSSL_CTX_set_read_ahead

```
int wolfSSL_CTX_set_read_ahead(
    WOLFSSL_CTX * ctx,
    int v
)
```

この関数は、WOLFSSL_CTX 構造内の読み出し先のフラグを設定します。

Parameters:

- **ctx** WOLFSSL_CTX 構造体へのポインタ
- **v** 先読みフラグ

See:

- wolfSSL_CTX_new
- wolfSSL_CTX_free
- wolfSSL_CTX_get_read_ahead

Return:

- SSL_SUCCESS ctx が先読みフラグを設定した場合。
- SSL_FAILURE ctx が NULL の場合に返されます。

Example

```
WOLFSSL_CTX* ctx;
int flag;
int ret;
// setup ctx
ret = wolfSSL_CTX_set_read_ahead(ctx, flag);
// check return value
```

C.52.2.153 function wolfSSL_CTX_set_tlsext_status_arg

```
long wolfSSL_CTX_set_tlsext_status_arg(
    WOLFSSL_CTX * ctx,
    void * arg
)
```

この関数は OCSP で使用するオプション引数を設定します。

Parameters:

- **ctx** WOLFSSL_CTX 構造へのポインタ
- **arg** ユーザー引数

See:

- wolfSSL_CTX_new
- wolfSSL_CTX_free

Return:

- SSL_FAILURE CTX または IT の CERT Manager が NULL の場合。
- SSL_SUCCESS 正常に設定されている場合。

Example

```
WOLFSSL_CTX* ctx;
void* data;
int ret;
// setup ctx
ret = wolfSSL_CTX_set_tlsext_status_arg(ctx, data);

//check ret value
```

C.52.2.154 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
)
```

この関数は、PRF コールバックに渡すオプションの引数を設定します。

Parameters:

- **ctx** WOLFSSL_CTX 構造へのポインタ
- **arg** ユーザー引数

See:

- wolfSSL_CTX_new
- wolfSSL_CTX_free

Return:

- SSL_FAILURE CTX が NULL の場合
- SSL_SUCCESS 正常に設定されている場合。

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.52.2.155 function wolfSSL_set_options

```
long wolfSSL_set_options(
    WOLFSSL * s,
    long op
)
```

この関数は、SSL のオプションマスクを設定します。いくつかの有効なオプションは、ssl_op_all、ssl_op_cookie_exchange、ssl_op_no_sslv2、ssl_op_no_sslv3、ssl_op_no_tlsv1_1、ssl_op_no_tlsv1_2、ssl_op_no_compression です。

Parameters:

- **s** オプションマスクを設定するための WolfSSL 構造。
- **op** オプションマスク。以下の値が指定可能です：

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 SSL に格納されている更新されたオプションマスク値を返します。

Example

```
WOLFSSL* ssl;
unsigned long mask;
mask = SSL_OP_NO_TLSv1
mask  = wolfSSL_set_options(ssl, mask);
// check mask
```

C.52.2.156 function wolfSSL_get_options

```
long wolfSSL_get_options(
    const WOLFSSL * ssl
)
```

この関数は現在のオプションマスクを返します。

Parameters:

- **ssl** WOLFSSL 構造体へのポインタ

See:

- wolfSSL_new
- wolfSSL_free
- wolfSSL_set_options

Return: val SSL に格納されているマスク値を返します。

Example

```
WOLFSSL* ssl;
unsigned long mask;
mask  = wolfSSL_get_options(ssl);
// check mask
```

C.52.2.157 function wolfSSL_set_tlsext_debug_arg

```
long wolfSSL_set_tlsext_debug_arg(
    WOLFSSL * ssl,
    void * arg
)
```

この関数は、渡されたデバッグ引数を設定するために使用されます。

Parameters:

- **ssl** 引数を設定するための WolfSSL 構造。
- **arg** デバッグ引数

See:

- wolfSSL_new
- wolfSSL_free

Return:

- SSL_SUCCESS 成功時に返されます。
- SSL_FAILURE NULL SSL が渡された場合。

Example

```
WOLFSSL* ssl;
void* args;
int ret;
// create ssl object
ret  = wolfSSL_set_tlsext_debug_arg(ssl, args);
// check ret value
```

C.52.2.158 function wolfSSL_set_tlsext_status_type

```
long wolfSSL_set_tlsext_status_type(
    WOLFSSL * s,
    int type
)
```

この関数は、サーバが OCSP ステータス応答（OCSP ステイプルとも呼ばれる）を送受信するクライアントアプリケーションが要求されたときに呼び出されます。

Parameters:

- **s** ssl_new() 関数によって作成された WOLFSSL 構造体へのポインタ
- **type** ssl 拡張タイプ。TLSEXT_STATUSTYPE_ocsp のみ指定可。

See:

- wolfSSL_new
- wolfSSL_CTX_new
- wolfSSL_free

- wolfSSL_CTX_free

Return:

- 1 成功時に返されます。
- 0 エラー時に返されます。

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.52.2.159 function wolfSSL_get_verify_result

```
long wolfSSL_get_verify_result(
    const WOLFSSL * ssl
)
```

Parameters:

- **ssl** WOLFSSL 構造体へのポインタ

See:

- wolfSSL_new
- wolfSSL_free

Return:

- X509_V_OK 成功した検証について
- SSL_FAILURE NULL SSL が渡された場合。

bri f この関数は、れは、ピアの証明書を確認しようとした後に結果を取得するために使用されます。*Example*

```
WOLFSSL* ssl;
long ret;
// attempt/complete handshake
ret  = wolfSSL_get_verify_result(ssl);
// check ret value
```

C.52.2.160 function wolfSSL_ERR_print_errors_fp

```
void wolfSSL_ERR_print_errors_fp(
    XFILE fp,
    int err
)
```

この関数は、wolfSSL_get_error() によって返されたエラーコードをより多くの人間が読めるエラー文字列に変換し、その文字列を出力ファイルに印刷します。ERR は、WOLFSSL_GET_ERROR() によって返され、FP がエラー文字列が配置されるファイルであるエラーコードです。

Parameters:

- **fp** に書き込まれる人間が読めるエラー文字列の出力ファイル。
- **err** wolfSSL_get_error()で返されるエラーコード。

See:

- wolfSSL_get_error
- wolfSSL_ERR_error_string
- wolfSSL_ERR_error_string_n
- wolfSSL_load_error_strings

Return: なし

Example

```
int err = 0;
WOLFSSL* ssl;
FILE* fp = ...
...
err = wolfSSL_get_error(ssl, 0);
wolfSSL_ERR_print_errors_fp(fp, err);
```

C.52.2.161 function wolfSSL_ERR_print_errors_cb

```
void wolfSSL_ERR_print_errors_cb(
    int(*)(const char *str, size_t len, void *u) cb,
    void * u
)
```

この関数は提供されたコールバックを使用してエラー報告を処理します。コールバック関数はエラー回線ごとに実行されます。文字列、長さ、および userdata はコールバックパラメータに渡されます。

Parameters:

- **cb** コールバック関数
- **u** コールバック関数に渡される userdata

See:

- wolfSSL_get_error
- wolfSSL_ERR_error_string
- wolfSSL_ERR_error_string_n
- wolfSSL_load_error_strings

Return: なし

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.52.2.162 function wolfSSL_CTX_set_psk_client_callback

```
void wolfSSL_CTX_set_psk_client_callback(
    WOLFSSL_CTX * ctx,
    wc_psk_client_callback cb
)
```

この関数は WOLFSSL_CTX 構造の client_psk_cb メンバーをセットします。

Parameters:

- **ctx** wolfSSL_CTX_new()を使用して作成された WOLFSSL_CTX 構造体へのポインタ。

- **cb** wc_psk_client_callback はコールバック関数ポインタで WOLFSSL_CTX 構造体に格納されます。戻り値は成功時には鍵長を返し、エラー時には 0 を返します。unsigned int (_wc_psk_client_callback) PSK クライアントコールバック関数の引数：

WOLFSSL_ ssl - WOLFSSL 構造体へのポインタ

const char* hint - ユーザーに対して表示されるヒント文字列

char* identity - ID

unsigned int id_max_len - ID バッファのサイズ

unsigned char* key - 格納される鍵

unsigned int key_max_len - 鍵の最大サイズ

See:

- wolfSSL_set_psk_client_callback
- wolfSSL_set_psk_server_callback
- wolfSSL_CTX_set_psk_server_callback
- wolfSSL_CTX_set_psk_client_callback

Return: なし

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.52.2.163 function wolfSSL_set_psk_client_callback

```
void wolfSSL_set_psk_client_callback(
    WOLFSSL * ssl,
    wc_psk_client_callback
)
```

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WolfSSL 構造体へのポインタ

See:

- wolfSSL_CTX_set_psk_client_callback
- wolfSSL_CTX_set_psk_server_callback
- wolfSSL_set_psk_server_callback

Return: none いいえ返します。

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.52.2.164 function wolfSSL_get_psk_identity_hint

```
const char * wolfSSL_get_psk_identity_hint(
    const WOLFSSL *
)
```

この関数は PSK アイデンティティヒントを返します。

See: wolfSSL_get_psk_identity

Return:

- pointer WolfSSL 構造の配列メンバーに格納されている値への const char ポインタが返されます。
- NULL WOLFSSL または配列構造が NULL の場合に返されます。

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.52.2.165 function wolfSSL_get_psk_identity

```
const char * wolfSSL_get_psk_identity(
    const WOLFSSL *
)
```

関数は、配列構造の Client_Identity メンバーへの定数ポインタを返します。

See:

- wolfSSL_get_psk_identity_hint
- wolfSSL_use_psk_identity_hint

Return:

- string 配列構造の client_identity メンバの文字列値。
- NULL WOLFSSL 構造が NULL の場合、または WOLFSSL 構造の配列メンバーが NULL の場合。

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.52.2.166 function wolfSSL_CTX_use_psk_identity_hint

```
int wolfSSL_CTX_use_psk_identity_hint(
    WOLFSSL_CTX * ctx,
    const char * hint
)
```

この関数は、WOLFSSL_CTX 構造体の server_hint メンバーに HINT 引数を格納します。

Parameters:

- **ctx** wolfSSL_CTX_new()を使用して作成された WOLFSSL_CTX 構造体へのポインタ。

See: wolfSSL_use_psk_identity_hint

Return: SSL_SUCCESS 機能の実行が成功したために返されます。

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.52.2.167 function wolfSSL_use_psk_identity_hint

```
int wolfSSL_use_psk_identity_hint(
    WOLFSSL * ssl,
    const char * hint
)
```

この関数は、wolfssl 構造内の配列構造の server_hint メンバーに HINT 引数を格納します。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WolfSSL 構造体へのポインタ

See: wolfSSL_CTX_use_psk_identity_hint

Return:

- SSL_SUCCESS ヒントが WolfSSL 構造に正常に保存された場合に返されます。
- SSL_FAILURE WOLFSSL または配列構造が NULL の場合に返されます。

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
const char* hint;
...
if(wolfSSL_use_psk_identity_hint(ssl, hint) != SSL_SUCCESS){
    // Handle failure case.
}
```

C.52.2.168 function wolfSSL_CTX_set_psk_server_callback

```
void wolfSSL_CTX_set_psk_server_callback(
    WOLFSSL_CTX * ctx,
    wc_psk_server_callback cb
)
```

WOLFSSL_CTX 構造体

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WolfSSL 構造体へのポインタ

See:

- wc_psk_server_callback
- wolfSSL_set_psk_client_callback
- wolfSSL_set_psk_server_callback
- wolfSSL_CTX_set_psk_client_callback

Return: none いいえ返します。

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.52.2.169 function wolfSSL_set_psk_server_callback

```
void wolfSSL_set_psk_server_callback(
    WOLFSSL * ssl,
    wc_psk_server_callback cb
)
```

WolfSSL 構造オプションメンバー。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WolfSSL 構造体へのポインタ

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 いいえ返します。

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.52.2.170 function wolfSSL_set_psk_callback_ctx

```
int wolfSSL_set_psk_callback_ctx(
    WOLFSSL * ssl,
    void * psk_ctx
)
```

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WolfSSL 構造体へのポインタ

See:

- wolfSSL_get_psk_callback_ctx
- wolfSSL_CTX_set_psk_callback_ctx
- wolfSSL_CTX_get_psk_callback_ctx

Return: WOLFSSL_SUCCESS または wolfssl_failure.

C.52.2.171 function wolfSSL_CTX_set_psk_callback_ctx

```
int wolfSSL_CTX_set_psk_callback_ctx(
    WOLFSSL_CTX * ctx,
    void * psk_ctx
)
```

Parameters:

- **ctx** wolfSSL_CTX_new()を使用して作成された WOLFSSL_CTX 構造体へのポインタ。

See:

- wolfSSL_set_psk_callback_ctx
- wolfSSL_get_psk_callback_ctx
- wolfSSL_CTX_get_psk_callback_ctx

Return: WOLFSSL_SUCCESS または wolfssl_failure.

C.52.2.172 function wolfSSL_get_psk_callback_ctx

```
void * wolfSSL_get_psk_callback_ctx(
    WOLFSSL * ssl
)
```

See:

- wolfSSL_set_psk_callback_ctx
- wolfSSL_CTX_set_psk_callback_ctx
- wolfSSL_CTX_get_psk_callback_ctx

Return: void ユーザー PSK コンテキストへのポインタ

C.52.2.173 function wolfSSL_CTX_get_psk_callback_ctx

```
void * wolfSSL_CTX_get_psk_callback_ctx(
    WOLFSSL_CTX * ctx
)
```

See:

- wolfSSL_CTX_set_psk_callback_ctx
- wolfSSL_set_psk_callback_ctx
- wolfSSL_get_psk_callback_ctx

Return: void ユーザー PSK コンテキストへのポインタ

C.52.2.174 function wolfSSL_CTX_allow_anon_cipher

```
int wolfSSL_CTX_allow_anon_cipher(
    WOLFSSL_CTX *
)
```

この機能により、CTX 構造の HAVAnon メンバーがコンパイル中に定義されている場合は、CTX 構造の HABANON メンバーを有効にします。

See: none

Return:

- SSL_SUCCESS 機能が正常に実行され、CTX の Haveannon メンバーが 1 に設定されている場合に返されます。
- SSL_FAILURE CTX 構造が NULL の場合に返されます。

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.52.2.175 function wolfSSLv23_server_method

```
WOLFSSL_METHOD * wolfSSLv23_server_method(
    void
)
```

wolfsslv23_server_method() 関数は、アプリケーションがサーバーであることを示すために使用され、SSL 3.0 - TLS 1.3 からプロトコルバージョンと接続するクライアントをサポートします。この関数は、wolfSSL_CTX_new() を使用して SSL / TLS コンテキストを作成するときに使用される新しい Wolfssl_method 構造体のメモリを割り当てて初期化します。

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 成功した場合、呼び出しは新しく作成された wolfssl_method 構造へのポインタを返します。
- Failure xmalloc を呼び出すときにメモリ割り当てが失敗した場合、基礎となる Malloc() 実装の失敗値が返されます（通常は errno が enomeem に設定されます）。

Example

```
WOLFSSL_METHOD* method;
WOLFSSL_CTX* ctx;

method = wolfSSLv23_server_method();
if (method == NULL) {
    // unable to get method
}

ctx = wolfSSL_CTX_new(method);
...
```

C.52.2.176 function wolfSSL_state

```
int wolfSSL_state(
    WOLFSSL * ssl
)
```

See:

- wolfSSL_new
- wolfSSL_free

Return:

- wolfssl_error SSL エラー状態、通常はマイナスを返します
- BAD_FUNC_ARG ssl が NULL の場合

bri f この関数は、れは、WolfSSL 構造体の内部エラー状態を取得するために使用されます。*Example*

```
WOLFSSL* ssl;
int ret;
// create ssl object
ret  = wolfSSL_state(ssl);
// check ret value
```

C.52.2.177 function wolfSSL_get_peer_certificate

```
WOLFSSL_X509 * wolfSSL_get_peer_certificate(
    WOLFSSL * ssl
)
```

この関数はピアの証明書を取得します。

See:

- wolfSSL_X509_get_issuer_name
- wolfSSL_X509_get_subject_name
- wolfSSL_X509_get_isCA

Return:

- pointer WOLFSSL_X509 構造の PECRERT メンバーへのポインタが存在する場合は。
- 0 ピア証明書発行者サイズが定義されていない場合に返されます。

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.52.2.178 function wolfSSL_want_read

```
int wolfSSL_want_read(
    WOLFSSL *
)
```

この関数は、wolfSSL_get_error() を呼び出して ssl_error_want_read を取得するのと似ています。基礎となるエラー状態が SSL_ERROR_WANT_READ の場合、この関数は 1 を返しますが、それ以外の場合は 0 です。

See:

- wolfSSL_want_write
- wolfSSL_get_error

Return:

- 1 WOLFSSL_GET_ERROR() は SSL_ERROR_WANT_READ を返し、基礎となる I / O には読み取り可能なデータがあります。
- 0 SSL_ERROR_WANT_READ エラー状態はありません。

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.52.2.179 function wolfSSL_want_write

```
int wolfSSL_want_write(
    WOLFSSL *
)
```

この関数は、wolfSSL_get_error() を呼び出し、RETURS の SSL_ERROR_WANT_WRITE を取得するのと同じです。基礎となるエラー状態が SSL_ERROR_WANT_WRITE の場合、この関数は 1 を返しますが、それ以外の場合は 0 です。

See:

- wolfSSL_want_read
- wolfSSL_get_error

Return:

- 1 WOLFSSL_GET_ERROR() は SSL_ERROR_WANT_WRITE を返します。基礎となる I / O は、基礎となる SSL 接続で進行状況を行うために書き込まれるデータを必要とします。
- 0 ssl_error_want_write エラー状態はありません。

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.52.2.180 function wolfSSL_check_domain_name

```
int wolfSSL_check_domain_name(
    WOLFSSL * ssl,
    const char * dn
)
```

wolfssl デフォルトでは、有効な日付範囲と検証済みの署名のためにピア証明書をチェックします。wolfssl_connect() または wolfssl_accept() の前にこの関数を呼び出すと、実行するチェックのリストにドメイン名チェックが追加されます。DN 受信時にピア証明書を確認するためのドメイン名を保持します。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WolfSSL 構造体へのポインタ

See: none

Return:

- SSL_SUCCESS 成功時に返されます。
- SSL_FAILURE メモリエラーが発生した場合に返されます。

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.52.2.181 function wolfSSL_Init

```
int wolfSSL_Init(
    void
)
```

使用するために WolfSSL ライブラリを初期化します。アプリケーションごとに 1 回、その他のライブラリへの呼び出しの前に呼び出す必要があります。

See: wolfSSL_Cleanup

Return:

- SSL_SUCCESS 成功した場合に返されます。、通話が戻ります。
- BAD_MUTEX_E 返される可能性があるエラーです。

Example

```
int ret = 0;
ret = wolfSSL_Init();
if (ret != SSL_SUCCESS) {
    failed to initialize wolfSSL library
}
```

C.52.2.182 function wolfSSL_Cleanup

```
int wolfSSL_Cleanup(
    void
)
```

さらなる使用から WOLFSSL ライブラリを初期化します。ライブラリによって使用されるリソースを解放しますが、呼び出される必要はありません。

See: wolfSSL_Init

Return: SSL_SUCCESS エラーを返しません。

Example

```
wolfSSL_Cleanup();
```

C.52.2.183 function wolfSSL_lib_version

```
const char * wolfSSL_lib_version(
    void
)
```

この関数は現在のライブラリーバージョンを返します。

See: word32_wolfSSL_lib_version_hex

Return: LIBWOLFSSL_VERSION_STRING バージョンを定義する const char ポインタ。

Example

```
char version[MAXSIZE];
version = wolfSSL_KeepArrays();
…
if(version != ExpectedVersion){
```

```
    // Handle the mismatch case
}
```

C.52.2.184 function wolfSSL_lib_version_hex

```
word32 wolfSSL_lib_version_hex(
    void
)
```

この関数は、現在のライブラリーのバージョンを 16 進表記で返します。

See: wolfSSL_lib_version

Return: LILBWOLFSSL_VERSION_HEX wolfssl / version.h で定義されている 16 進数バージョンを返します。

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.52.2.185 function wolfSSL_negotiate

```
int wolfSSL_negotiate(
    WOLFSSL * ssl
)
```

SSL メソッドの側面に基づいて、実際の接続または承認を実行します。クライアント側から呼び出された場合、サーバ側から呼び出された場合に wolfssl_accept() が実行されている間に wolfssl_connect() が行われる。

See:

- SSL_connect
- SSL_accept

Return:

- SSL_SUCCESS 成功した場合に返されます。に返却されます。(注意、古いバージョンは 0 を返します)
- SSL_FATAL_ERROR 基礎となる呼び出しがエラーになった場合に返されます。特定のエラーコードを取得するには、wolfSSL_get_error() を使用してください。

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.52.2.186 function wolfSSL_set_compression

```
int wolfSSL_set_compression(
    WOLFSSL * ssl
)
```

SSL 接続に圧縮を使用する機能をオンにします。両側には圧縮がオンになっている必要があります。そうでなければ圧縮は使用されません。ZLIB ライブラリは実際のデータ圧縮を実行します。ライブラリにコンパイルするには、システムの設定システムに–with-libz を使用し、そうでない場合は hand_libz を定義します。送受信されるメッセージの実際のサイズを減らす前にデータを圧縮している間に、圧縮によって保存されたデータの量は通常、ネットワークの遅いすべてのネットワークを除いたものよりも分析に時間がかかります。

See: none

Return:

- SSL_SUCCESS 成功時に返されます。
- NOT_COMPILED_IN 圧縮サポートがライブラリに組み込まれていない場合に返されます。

Example

```
int ret = 0;
WOLFSSL* ssl = 0;
...
ret = wolfSSL_set_compression(ssl);
if (ret == SSL_SUCCESS) {
    // successfully enabled compression for SSL session
}
```

C.52.2.187 function wolfSSL_set_timeout

```
int wolfSSL_set_timeout(
    WOLFSSL * ssl,
    unsigned int to
)
```

この関数は SSL セッションタイムアウト値を秒単位で設定します。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WolfSSL 構造体へのポインタ

See:

- wolfSSL_get1_session
- wolfSSL_set_session

Return:

- SSL_SUCCESS セッションを正常に設定すると返されます。
- BAD_FUNC_ARG ssl が NULL の場合に返されます。

Example

```
int ret = 0;
WOLFSSL* ssl = 0;
...

ret = wolfSSL_set_timeout(ssl, 500);
if (ret != SSL_SUCCESS) {
    // failed to set session timeout value
```

```
}
...
```

C.52.2.188 function wolfSSL_CTX_set_timeout

```
int wolfSSL_CTX_set_timeout(
    WOLFSSL_CTX * ctx,
    unsigned int to
)
```

この関数は、指定された SSL コンテキストに対して、SSL セッションのタイムアウト値を秒単位で設定します。

Parameters:

- **ctx** wolfSSL_CTX_new()で作成された SSL コンテキストへのポインタ。

See:

- wolfSSL_flush_sessions
- wolfSSL_get1_session
- wolfSSL_set_session
- wolfSSL_get_sessionID
- wolfSSL_CTX_set_session_cache_mode

Return:

- the wolfssl_error_code_openssl の場合、以前のタイムアウト値
- defined 成功しています。定義されていない場合、SSL_SUCCESS は返されます。
- BAD_FUNC_ARG 入力コンテキスト（CTX）が NULL のときに返されます。

Example

```
WOLFSSL_CTX*    ctx    = 0;
...
ret = wolfSSL_CTX_set_timeout(ctx, 500);
if (ret != SSL_SUCCESS) {
    // failed to set session timeout value
}
```

C.52.2.189 function wolfSSL_get_peer_chain

```
WOLFSSL_X509_CHAIN * wolfSSL_get_peer_chain(
    WOLFSSL * ssl
)
```

ピアの証明書チェーンを取得します。

See:

- wolfSSL_get_chain_count
- wolfSSL_get_chain_length
- wolfSSL_get_chain_cert
- wolfSSL_get_chain_cert_pem

Return:

- chain 正常にコールがピアの証明書チェーンを返します。
- 0 無効な WolfSSL ポインタが関数に渡されると返されます。

Example

none

C.52.2.190 function wolfSSL_get_chain_count

```
int wolfSSL_get_chain_count(
    WOLFSSL_X509_CHAIN * chain
)
```

ピアの証明書チェーン数を取得します。

See:

- wolfSSL_get_peer_chain
- wolfSSL_get_chain_length
- wolfSSL_get_chain_cert
- wolfSSL_get_chain_cert_pem

Return:

- Success 正常にコールがピアの証明書チェーン数を返します。
- 0 無効なチェーンポインタが関数に渡されると返されます。

Example

none

C.52.2.191 function wolfSSL_get_chain_length

```
int wolfSSL_get_chain_length(
    WOLFSSL_X509_CHAIN * chain,
    int idx
)
```

Index（IDX）のピアの ASN1.DER 証明書長をバイト単位で取得します。

Parameters:

- **chain** 有効な wolfssl_x509_chain 構造へのポインタ。

See:

- wolfSSL_get_peer_chain
- wolfSSL_get_chain_count
- wolfSSL_get_chain_cert
- wolfSSL_get_chain_cert_pem

Return:

- Success 正常にコールがインデックス別にピアの証明書長をバイト単位で返します。
- 0 無効なチェーンポインタが関数に渡されると返されます。

Example

none

C.52.2.192 function wolfSSL_get_chain_cert

```
unsigned char * wolfSSL_get_chain_cert(
    WOLFSSL_X509_CHAIN * chain,
    int idx
)
```

インデックス（IDX）でピアの ASN1.DER 証明書を取得します。

Parameters:

- **chain** 有効な wolfssl_x509_chain 構造へのポインタ。

See:

- wolfSSL_get_peer_chain
- wolfSSL_get_chain_count
- wolfSSL_get_chain_length
- wolfSSL_get_chain_cert_pem

Return:

- Success 正常にコールがインデックスでピアの証明書を返します。
- 0 無効なチェーンポインタが関数に渡されると返されます。

Example

none

C.52.2.193 function wolfSSL_get_chain_X509

```
WOLFSSL_X509 * wolfSSL_get_chain_X509(
    WOLFSSL_X509_CHAIN * chain,
    int idx
)
```

この関数は、証明書のチェーンからのピアの WOLFSSL_X509 構造体をインデックス（IDX）で取得します。

Parameters:

- **chain** 動的メモリ session_cache の場合に使用される WOLFSSL_X509_CHAIN へのポインタ。

See:

- InitDecodedCert
- ParseCertRelative
- CopyDecodedToX509

Return: pointer WOLFSSL_X509 構造体へのポインタを返します。

注意：本関数から返された構造体を wolfSSL_FreeX509() を呼び出して解放するのはユーザーの責任です。

Example

```
WOLFSSL_X509_CHAIN* chain = &session->chain;
int idx = 999; // set idx
...
WOLFSSL_X509* 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.52.2.194 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
)
```

インデックス（IDX）でピアの PEM 証明書を取得します。

Parameters:

- **chain** 有効な wolfssl_x509_chain 構造へのポインタ。

See:

- wolfSSL_get_peer_chain
- wolfSSL_get_chain_count
- wolfSSL_get_chain_length
- wolfSSL_get_chain_cert

Return:

- Success 正常にコールがインデックスでピアの証明書を返します。
- 0 無効なチェーンポインタが関数に渡されると返されます。

Example

none

C.52.2.195 function wolfSSL_get_sessionID

```
const unsigned char * wolfSSL_get_sessionID(
    const WOLFSSL_SESSION * s
)
```

セッションの ID を取得します。セッション ID は常に 32 バイトの長さです。

See: SSL_get_session

Return: id セッション ID。

Example

none

C.52.2.196 function wolfSSL_X509_get_serial_number

```
int wolfSSL_X509_get_serial_number(
    WOLFSSL_X509 * x509,
    unsigned char * in,
    int * inOutSz
)
```

ピアの証明書のシリアル番号を取得します。シリアル番号バッファ（IN）は少なくとも 32 バイト以上であり、入力として * INOUTSZ 引数として提供されます。関数を呼び出した後 * INOUTSZ は IN バッファに書き込まれた実際の長さをバイト単位で保持します。

Parameters:

- **in** シリアル番号バッファは少なくとも 32 バイトの長さであるべきです

See: SSL_get_peer_certificate

Return:

- SSL_SUCCESS 成功時に返されます。
- BAD_FUNC_ARG 関数の不良引数が見つかった場合に返されます。

Example

none

C.52.2.197 function wolfSSL_X509_get_subjectCN

```
char * wolfSSL_X509_get_subjectCN(
    WOLFSSL_X509 *
)
```

証明書から件名の共通名を返します。

See:

- wolfSSL_X509_Name_get_entry
- wolfSSL_X509_get_next_altname
- wolfSSL_X509_get_issuer_name
- wolfSSL_X509_get_subject_name

Return:

- NULL X509 構造が NULL の場合に返されます
- string サブジェクトの共通名の文字列表現は成功に返されます

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.52.2.198 function wolfSSL_X509_get_der

```
const unsigned char * wolfSSL_X509_get_der(
    WOLFSSL_X509 * x509,
    int * outSz
)
```

この関数は、wolfssl_x509 構造体の DER エンコードされた証明書を取得します。

Parameters:

- **x509** 証明書情報を含む WolfSSL_X509 構造へのポインタ。

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 この関数は Derbuffer 構造体のバッファメンバーを返します。これはバイト型です。
- NULL x509 または outsz パラメーターが null の場合に返されます。

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.52.2.199 function wolfSSL_X509_get_notAfter

```
WOLFSSL_ASN1_TIME * wolfSSL_X509_get_notAfter(
    WOLFSSL_X509 *
)
```

この関数は、x509 が null のかどうかを確認し、そうでない場合は、x509 構造体のノッカスメンバーを返します。

See: wolfSSL_X509_get_notBefore

Return:

- pointer ASN1_TIME を使用して X509 構造体のノカフターメンバーに構造体を表明します。
- NULL X509 オブジェクトが NULL の場合に返されます。

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.52.2.200 function wolfSSL_X509_version

```
int wolfSSL_X509_version(
    WOLFSSL_X509 *
)
```

この関数は X509 証明書のバージョンを取得します。

See:

- wolfSSL_X509_get_subject_name
- wolfSSL_X509_get_issuer_name
- wolfSSL_X509_get_isCA
- wolfSSL_get_peer_certificate

Return:

- 0 X509 構造が NULL の場合に返されます。

- version X509 構造に保存されているバージョンが返されます。

Example

```
WOLFSSL_X509* x509;
int version;
...
version = wolfSSL_X509_version(x509);
if(!version){
    // The function returned 0, failure case.
}
```

C.52.2.201 function wolfSSL_X509_d2i_fp

```
WOLFSSL_X509 * wolfSSL_X509_d2i_fp(
    WOLFSSL_X509 ** x509,
    FILE * file
)
```

no_stdio_filesystem が定義されている場合、この関数はヒープメモリを割り当て、wolfssl_x509 構造を初期化してそれにポインタを返します。

Parameters:

- **x509** wolfssl_x509 ポインタへのポインタ。

See:

- wolfSSL_X509_d2i
- XFTELL
- XREWIND
- XFSEEK

Return:

- *WOLFSSL_X509 関数が正常に実行された場合、WolfSSL_X509 構造ポインタが返されます。
- NULL Xftell マクロの呼び出しが負の値を返す場合。

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.52.2.202 function wolfSSL_X509_load_certificate_file

```
WOLFSSL_X509 * wolfSSL_X509_load_certificate_file(
    const char * fname,
    int format
)
```

関数は X509 証明書をメモリにロードします。

Parameters:

- **fname** ロードする証明書ファイル。

See:

- InitDecodedCert
- PemToDer
- wolfSSL_get_certificate
- AssertNotNull

Return:

- pointer 実行された実行は、wolfssl_x509 構造へのポインタを返します。
- NULL 証明書が書き込まれなかった場合に返されます。

Example

```
#define cliCert    “certs/client-cert.pem”
…
X509* x509;
…
x509 = wolfSSL_X509_load_certificate_file(cliCert, SSL_FILETYPE_PEM);
AssertNotNull(x509);
```

C.52.2.203 function wolfSSL_X509_get_device_type

```
unsigned char * wolfSSL_X509_get_device_type(
    WOLFSSL_X509 * x509,
    unsigned char * in,
    int * inOutSz
)
```

この関数は、デバイスの種類を X509 構造からバッファにコピーします。

Parameters:

- **x509** wolfssl_x509_new() で作成された wolfssl_x509 構造へのポインタ。
- **in** デバイスタイプ（バッファ）を保持するバイトタイプへのポインタ。

See:

- wolfSSL_X509_get_hw_type
- wolfSSL_X509_get_hw_serial_number
- wolfSSL_X509_d2i

Return:

- pointer X509 構造からデバイスの種類を保持するバイトポインタを返します。
- NULL バッファサイズが NULL の場合に返されます。

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.52.2.204 function wolfSSL_X509_get_hw_type

```
unsigned char * wolfSSL_X509_get_hw_type(
    WOLFSSL_X509 * x509,
    unsigned char * in,
    int * inOutSz
)
```

この関数は、wolfssl_x509 構造の HWType メンバーをバッファにコピーします。

Parameters:

- **x509** 証明書情報を含む WolfSSL_X509 構造へのポインタ。
- **in** バッファを表すバイトを入力するポインタ。

See:

- wolfSSL_X509_get_hw_serial_number
- wolfSSL_X509_get_device_type

Return:

- byte この関数は、wolfssl_x509 構造の HWType メンバーに以前に保持されているデータのバイトタイプを返します。
- NULL inoutsz が null の場合に返されます。

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.52.2.205 function wolfSSL_X509_get_hw_serial_number

```
unsigned char * wolfSSL_X509_get_hw_serial_number(
    WOLFSSL_X509 * x509,
    unsigned char * in,
    int * inOutSz
)
```

この関数は X509 オブジェクトの hwserialNum メンバを返します。

Parameters:

- **x509** 証明書情報を含む WOLFSSL_X509 構造へのポインタ。
- **in** コピーされるバッファへのポインタ。

See:

- wolfSSL_X509_get_subject_name
- wolfSSL_X509_get_issuer_name
- wolfSSL_X509_get_isCA
- wolfSSL_get_peer_certificate
- wolfSSL_X509_version

Return: pointer この関数は、X509 オブジェクトからロードされたシリアル番号を含む IN バッファへのバイトポインタを返します。

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.52.2.206 function wolfSSL_connect_cert

```
int wolfSSL_connect_cert(
    WOLFSSL * ssl
)
```

この関数はクライアント側で呼び出され、ピアの証明書チェーンを取得するのに十分な長さだけサーバーを持つ SSL / TLS ハンドシェイクを開始します。この関数が呼び出されると、基礎となる通信チャネルはすでに設定されています。wolfssl_connect_cert() は、ブロックと非ブロック I / O の両方で動作します。基礎となる I / O がノンブロッキングである場合、wolfsl_connect_cert() は、wolfssl_connect_cert_cert() のニーズを満たすことができなかったときに戻ります。ハンドシェイクを続けます。この場合、wolfSSL_get_error() への呼び出しは SSL_ERROR_WANT_READ または SSL_ERROR_WANT_WRITE のいずれかを生成します。通話プロセスは、基礎となる I / O が準備ができて、wolfssl がオフになっているところを拾うときに、wolfssl_connect_cert() への呼び出しを繰り返す必要があります。ノンブロッキングソケットを使用する場合は、何も実行する必要がありますが、select() を使用して必要な条件を確認できます。基礎となる入出力がブロックされている場合、wolfssl_connect_cert() はピアの証明書チェーンが受信されたらのみ返されます。

See:

- wolfSSL_get_error
- wolfSSL_connect
- wolfSSL_accept

Return:

- SSL_SUCCESS 成功時に返されます。
- SSL_FAILURE SSL セッションパラメータが NULL の場合、返されます。
- SSL_FATAL_ERROR エラーが発生した場合に返されます。より詳細なエラーコードを取得するには、wolfSSL_get_error() を呼び出します。

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.52.2.207 function wolfSSL_d2i_PKCS12_bio

```
WC_PKCS12 * wolfSSL_d2i_PKCS12_bio(
    WOLFSSL_BIO * bio,
    WC_PKCS12 ** pkcs12
)
```

WOLFSSL_D2I_PKCS12_BIO（D2I_PKCS12_BIO）は、WOLFSSL_BIO から構造 WC_PKCS12 への PKCS12 情報にコピーされます。この情報は、オプションの MAC 情報を保持するための構造とともにコンテンツに関する情報のリストとして構造内に分割されています。構造体 WC_PKCS12 で情報がチャンク（ただし復号化されていない）に分割された後、それはその後、呼び出しによって解析および復号化され得る。

Parameters:

- **bio** PKCS12 バッファを読み取るための WOLFSSL_BIO 構造。

See:

- wolfSSL_PKCS12_parse
- wc_PKCS12_free

Return:

- WC_PKCS12 WC_PKCS12 構造へのポインタ。
- Failure 関数に失敗した場合は NULL を返します。

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.52.2.208 function wolfSSL_i2d_PKCS12_bio

```
WC_PKCS12 * wolfSSL_i2d_PKCS12_bio(
    WOLFSSL_BIO * bio,
    WC_PKCS12 * pkcs12
)
```

WOLFSSL_I2D_PKCS12_BIO（I2D_PKCS12_BIO）は、構造 WC_PKCS12 から WOLFSSL_BIO への証明書情報にコピーされます。

Parameters:

- **bio** PKCS12 バッファを書き込むための WOLFSSL_BIO 構造。

See:

- wolfSSL_PKCS12_parse
- wc_PKCS12_free

Return:

- 1 成功のために。
- 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.52.2.209 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 は、configure コマンドへの-enable-openSSLAXTRA を追加することで有効にできます。それは復号化のためにトリプル DES と RC4 を使うことができるので、OpenSSlextra（–enable-des3 -enable-arc4）を有効にするときにもこれらの機能を有効にすることをお勧めします。wolfssl は現在 RC2 をサポートしていませんので、RC2 での復号化は現在利用できません。これは、.p12 ファイルを作成するために OpenSSL コマンドラインで使用されるデフォルトの暗号化方式では注目すかもしれません。WOLFSSL_PKCS12_PARSE（PKCS12_PARSE）。この関数が最初に行っているのは、存在する場合は Mac が正しいチェックです。MAC が失敗した場合、関数は返され、保存されているコンテンツ情報のいずれかを復号化しようとしません。この関数は、バッグタイプを探している各コンテンツ情報を介して解析します。バッグタイプがわかっている場合は、必要に応じて復号化され、構築されている証明書のリストに格納されているか、見つかったキーとして保存されます。すべてのバッグを介して解析した後、見つかったキーは、一致するペアが見つかるまで証明書リストと比較されます。この一致するペアはキーと証明書として返され、オプションで見つかった証明書リストは stack_of 証明書として返されます。瞬間、CRL、秘密または安全なバッグがスキップされ、解析されません。デバッグプリントアウトを見ることで、これらまたは他の「不明」バッグがスキップされているかどうかがわかります。フレンドリー名などの追加の属性は、PKCS12 ファイルを解析するときにスキップされます。

Parameters:

- **pkcs12** wc_pkcs12 解析する構造
- **paswd** PKCS12 を復号化するためのパスワード。
- **pkey** PKCS12 からデコードされた秘密鍵を保持するための構造。
- **cert** PKCS12 から復号された証明書を保持する構造

See:

- wolfSSL_d2i_PKCS12_bio
- wc_PKCS12_free

Return:

- SSL_SUCCESS PKCS12 の解析に成功しました。
- SSL_FAILURE エラーケースに遭遇した場合

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.52.2.210 function wolfSSL_SetTmpDH

```
int wolfSSL_SetTmpDH(
    WOLFSSL * ssl,
    const unsigned char * p,
    int pSz,
    const unsigned char * g,
    int gSz
)
```

サーバー DIFFIE-HELLMAN エフェメラルパラメータ設定。この関数は、サーバーが DHE を使用する暗号スイートをネゴシエートしている場合に使用するグループパラメータを設定します。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WolfSSL 構造へのポインタ。
- **p** Diffie-Hellman 素数パラメータ。
- **pSz** p のサイズ。
- **g** Diffie-Hellman “Generator” パラメータ。

See: SSL_accept

Return:

- SSL_SUCCESS 成功時に返されます。
- MEMORY_ERROR メモリエラーが発生した場合に返されます。
- SIDE_ERROR この関数が SSL サーバではなく SSL クライアントで呼び出されると返されます。

Example

```
WOLFSSL* ssl;
static unsigned char p[] = {...};
static unsigned char g[] = {...};
...
wolfSSL_SetTmpDH(ssl, p, sizeof(p), g, sizeof(g));
```

C.52.2.211 function wolfSSL_SetTmpDH_buffer

```
int wolfSSL_SetTmpDH_buffer(
    WOLFSSL * ssl,
    const unsigned char * b,
    long sz,
```

```
    int format
)
```

関数は wolfssl_settmph_buffer_wrapper を呼び出します。これは Diffie-Hellman パラメータのラッパーです。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WolfSSL 構造へのポインタ。
- **buf** wolfssl_settmph_file_wrapper から渡された割り当てバッファー。
- **sz** ファイルのサイズ（wolfssl_settmph_file_wrapper 内の fname）を保持するロング int。

See:

- wolfSSL_SetTmpDH_buffer_wrapper
- wc_DhParamsLoad
- wolfSSL_SetTmpDH
- PemToDer
- wolfSSL_CTX_SetTmpDH
- wolfSSL_CTX_SetTmpDH_file

Return:

- SSL_SUCCESS 実行に成功した場合。
- SSL_BAD_FILETYPE ファイルの種類が pem ではなく、asn.1 ではない場合 WC_DHParamSLOAD が正常に戻っていない場合は、も返されます。
- SSL_NO_PEM_HEADER PEM ヘッダーがない場合は PemToder から返します。
- SSL_BAD_FILE PemToder にファイルエラーがある場合に返されます。
- SSL_FATAL_ERROR コピーエラーが発生した場合は PemToder から返されました。
- MEMORY_E - メモリ割り当てエラーが発生した場合
- BAD_FUNC_ARG wolfssl 構造体が null の場合、またはそうでない場合はサブルーチンに渡された場合に返されます。
- DH_KEY_SIZE_E wolfssl_settmph() または WOLFSSL_CTX_settmph() の鍵サイズエラーがある場合に返されます。
- SIDE_ERROR wolfssl_settmph のサーバー側ではない場合に返されます。

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.52.2.212 function wolfSSL_SetTmpDH_file

```
int wolfSSL_SetTmpDH_file(
    WOLFSSL * ssl,
    const char * f,
    int format
)
```

この関数は、wolfssl_settmph_file_wrapper を呼び出してサーバ diffie-hellman パラメータを設定します。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ
- **fname** 証明書を保持している定数の文字ポインタ。

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 この機能の正常な完了とそのサブルーチンの完了に戻りました。
- MEMORY_E この関数またはサブルーチンにメモリ割り当てが失敗した場合に返されます。
- SIDE_ERROR WolfSSL 構造体にあるオプション構造のサイドメンバーがサーバー側ではない場合。
- SSL_BAD_FILETYPE 証明書が一連のチェックに失敗した場合は返します。
- DH_KEY_SIZE_E DH パラメーターの鍵サイズが WolfSSL 構造体の MinkKeysz メンバーの値より小さい場合に返されます。
- DH_KEY_SIZE_E DH パラメータの鍵サイズが wolfssl 構造体の MAXDHKEYSZ メンバーの値よりも大きい場合に返されます。
- BAD_FUNC_ARG wolfssl 構造など、引数値が null の場合に返します。

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
const char* dhParam;
...
AssertIntNE(SSL_SUCCESS,
wolfSSL_SetTmpDH_file(ssl, dhParam, SSL_FILETYPE_PEM));
```

C.52.2.213 function wolfSSL_CTX_SetTmpDH

```
int wolfSSL_CTX_SetTmpDH(
    WOLFSSL_CTX * ctx,
    const unsigned char * p,
    int pSz,
    const unsigned char * g,
    int gSz
)
```

サーバー CTX Diffie-Hellman のパラメータを設定します。

Parameters:

- **ctx** wolfSSL_CTX_new()を使用して作成された WOLFSSL_CTX 構造体へのポインタ。
- **p** ServerDH_P 構造体のバッファメンバーにロードされた定数の符号なし文字ポインタ。
- **pSz** p のサイズを表す int 型は、max_dh_size に初期化されます。
- **g** ServerDh_g 構造体のバッファメンバーにロードされた定数の符号なし文字ポインタ。

See:

- wolfSSL_SetTmpDH
- wc_DhParamsLoad

Return:

- SSL_SUCCESS 関数とすべてのサブルーチンがエラーなしで戻った場合に返されます。
- BAD_FUNC_ARG CTX、P、または G パラメーターが NULL の場合に返されます。
- DH_KEY_SIZE_E DH パラメータの鍵サイズが WOLFSSL_CTX 構造体の MindHKEYSZ メンバーの値より小さい場合に返されます。

- DH_KEY_SIZE_E DH パラメータの鍵サイズが WOLFSSL_CTX 構造体の MaxDhkeySZ メンバーの値よりも大きい場合に返されます。
- MEMORY_E この関数またはサブルーチンにメモリの割り当てが失敗した場合に返されます。

Example

```
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.52.2.214 function wolfSSL_CTX_SetTmpDH_buffer

```
int wolfSSL_CTX_SetTmpDH_buffer(
    WOLFSSL_CTX * ctx,
    const unsigned char * b,
    long sz,
    int format
)
```

wolfssl_settmph_buffer_wrapper を呼び出すラッパー関数

Parameters:

- **ctx** wolfSSL_CTX_new()を使用して作成された WolfSSL 構造へのポインタ。
- **buf** バッファとして割り当てられ、wolfssl_settmpdh_buffer_wrapper に渡された定数の符号なし文字型へのポインタ。
- **sz** wolfssl_settmph_file_wrapper() の FNAME パラメータから派生した長い整数型。

See:

- wolfSSL_SetTmpDH_buffer_wrapper
- wolfSSL_SetTMpDH_buffer
- wolfSSL_SetTmpDH_file_wrapper
- wolfSSL_CTX_SetTmpDH_file

Return:

- 0 実行が成功するために返されました。
- BAD_FUNC_ARG CTX パラメータまたは BUF パラメータが NULL の場合に返されます。
- MEMORY_E メモリ割り当てエラーがある場合
- SSL_BAD_FILETYPE フォーマットが正しくない場合に返されます。

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.52.2.215 function wolfSSL_CTX_SetTmpDH_file

```
int wolfSSL_CTX_SetTmpDH_file(
    WOLFSSL_CTX * ctx,
    const char * f,
    int format
)
```

この関数は、wolfssl_settmph_file_wrapper を呼び出してサーバー Diffie-Hellman パラメータを設定します。

Parameters:

- **ctx** wolfSSL_CTX_new()を使用して作成された WOLFSSL_CTX 構造体へのポインタ。
- **fname** 証明書ファイルへの定数文字ポインタ。

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 wolfssl_settmph_file_wrapper またはそのサブルーチンのいずれかが正常に戻った場合に返されます。
- MEMORY_E 動的メモリの割り当てがサブルーチンで失敗した場合に返されます。
- BAD_FUNC_ARG CTX または FNAME パラメータが NULL またはサブルーチンが NULL 引数に渡された場合に返されます。
- SSL_BAD_FILE 証明書ファイルが開くことができない場合、またはファイルの一連のチェックが wolfssl_settmpdh_file_wrapper から失敗した場合に返されます。
- SSL_BAD_FILETYPE フォーマットが wolfssl_settmph_buffer_wrapper() から PEM または ASN.1 ではない場合に返されます。
- DH_KEY_SIZE_E DH パラメータの鍵サイズが WOLFSSL_CTX 構造体の MindHKEYSZ メンバーの値より小さい場合に返されます。
- DH_KEY_SIZE_E DH パラメータの鍵サイズが WOLFSSL_CTX 構造体の MaxDhkeySZ メンバーの値よりも大きい場合に返されます。
- SIDE_ERROR wolfssl_settmph() で返されたサイドがサーバー終了ではない場合。
- SSL_NO_PEM_HEADER PEM ヘッダーがない場合は PemToder から返されます。
- SSL_FATAL_ERROR メモリコピーの失敗がある場合は PemToder から返されます。

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.52.2.216 function wolfSSL_CTX_SetMinDhKey_Sz

```
int wolfSSL_CTX_SetMinDhKey_Sz(
    WOLFSSL_CTX * ctx,
    word16
)
```

この関数は、WOLFSSL_CTX 構造体の minkkeysz メンバーにアクセスして、Diffie Hellman 鍵サイズの最小サイズ（ビット単位）を設定します。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WolfSSL 構造へのポインタ。

See:

- wolfSSL_SetMinDhKey_Sz
- wolfSSL_CTX_SetMaxDhKey_Sz
- wolfSSL_SetMaxDhKey_Sz
- wolfSSL_GetDhKey_Sz
- wolfSSL_CTX_SetTMpDH_file

Return:

- SSL_SUCCESS 関数が正常に完了した場合に返されます。
- BAD_FUNC_ARG WOLFSSL_CTX 構造体が null の場合、またはキー z_BITS が 16,000 を超えるか、または 8 によって割り切れない場合に返されます。

Example

```
public static int CTX_SetMinDhKey_Sz(IntPtr ctx, short minDhKey){
...
return wolfSSL_CTX_SetMinDhKey_Sz(local_ctx, minDhKeyBits);
```

C.52.2.217 function wolfSSL_SetMinDhKey_Sz

```
int wolfSSL_SetMinDhKey_Sz(
    WOLFSSL * ssl,
    word16 keySz_bits
)
```

WolfSSL 構造の Diffie-Hellman 鍵の最小サイズ（ビット単位）を設定します。

Parameters:

- **ssl** wolfssl_new() を使用して作成された WolfSSL 構造へのポインタ。

See:

- wolfSSL_CTX_SetMinDhKey_Sz
- wolfSSL_GetDhKey_Sz

Return:

- SSL_SUCCESS 最小サイズは正常に設定されました。
- BAD_FUNC_ARG wolfssl 構造は NULL、または Keysz_BITS が 16,000 を超えるか、または 8 によって割り切れない場合

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
word16 keySz_bits;
...
if(wolfSSL_SetMinDhKey_Sz(ssl, keySz_bits) != SSL_SUCCESS){
    // Failed to set.
}
```

C.52.2.218 function wolfSSL_CTX_SetMaxDhKey_Sz

```
int wolfSSL_CTX_SetMaxDhKey_Sz(
    WOLFSSL_CTX * ctx,
    word16 keySz_bits
)
```

この関数は、WOLFSSL_CTX 構造体の maxdhkeysz メンバーにアクセスして、Diffie Hellman 鍵サイズの最大サイズ（ビット単位）を設定します。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ

See:

- wolfSSL_SetMinDhKey_Sz
- wolfSSL_CTX_SetMinDhKey_Sz
- wolfSSL_SetMaxDhKey_Sz
- wolfSSL_GetDhKey_Sz
- wolfSSL_CTX_SetTMpDH_file

Return:

- SSL_SUCCESS 関数が正常に完了した場合に返されます。
- BAD_FUNC_ARG WOLFSSL_CTX 構造体が null の場合、またはキー z_BITS が 16,000 を超えるか、または 8 によって割り切れない場合に返されます。

Example

```
public static int CTX_SetMaxDhKey_Sz(IntPtr ctx, short maxDhKey){
…
return wolfSSL_CTX_SetMaxDhKey_Sz(local_ctx, keySz_bits);
```

C.52.2.219 function wolfSSL_SetMaxDhKey_Sz

```
int wolfSSL_SetMaxDhKey_Sz(
    WOLFSSL * ssl,
    word16 keySz_bits
)
```

WolfSSL 構造の Diffie-Hellman 鍵の最大サイズ（ビット単位）を設定します。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ

See:

- wolfSSL_CTX_SetMaxDhKey_Sz
- wolfSSL_GetDhKey_Sz

Return:

- SSL_SUCCESS 最大サイズは正常に設定されました。

- BAD_FUNC_ARG WOLFSSL 構造は NULL または KEYSZ パラメータは許容サイズより大きかったか、または 8 によって割り切れませんでした。

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
word16 keySz;
...
if(wolfSSL_SetMaxDhKey(ssl, keySz) != SSL_SUCCESS){
    // Failed to set.
}
```

C.52.2.220 function wolfSSL_GetDhKey_Sz

```
int wolfSSL_GetDhKey_Sz(
    WOLFSSL *
)
```

オプション構造のメンバーである DHKEYSZ（ビット内）の値を返します。この値は、Diffie-Hellman 鍵サイズをバイト単位で表します。

See:

- wolfSSL_SetMinDhKey_sz
- wolfSSL_CTX_SetMinDhKey_Sz
- wolfSSL_CTX_SetTmpDH
- wolfSSL_SetTmpDH
- wolfSSL_CTX_SetTmpDH_file

Return:

- dhKeySz サイズを表す整数値である ssl-> options.dhkeysz で保持されている値を返します。
- BAD_FUNC_ARG wolfssl 構造体が NULL の場合に返します。

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.52.2.221 function wolfSSL_CTX_SetMinRsaKey_Sz

```
int wolfSSL_CTX_SetMinRsaKey_Sz(
    WOLFSSL_CTX * ctx,
    short keySz
)
```

WOLFSSL_CTX 構造体と wolfssl_cert_manager 構造の両方で最小 RSA 鍵サイズを設定します。

Parameters:

- **ctx** wolfSSL_CTX_new()を使用して作成された WOLFSSL_CTX 構造体へのポインタ。

See: wolfSSL_SetMinRsaKey_Sz

Return:

- SSL_SUCCESS 機能の実行に成功したことに戻ります。
- BAD_FUNC_ARG CTX 構造が NULL の場合、または KEYSZ がゼロより小さいか、または 8 によって割り切れない場合に返されます。

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.52.2.222 function wolfSSL_SetMinRsaKey_Sz

```
int wolfSSL_SetMinRsaKey_Sz(
    WOLFSSL * ssl,
    short keySz
)
```

WolfSSL 構造にある RSA のためのビットで最小許容鍵サイズを設定します。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ

See: wolfSSL_CTX_SetMinRsaKey_Sz

Return:

- SSL_SUCCESS 最小値が正常に設定されました。
- BAD_FUNC_ARG SSL 構造が NULL の場合、または KSYSZ がゼロより小さい場合、または 8 によって割り切れない場合に返されます。

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
short keySz;
…

int isSet =  wolfSSL_SetMinRsaKey_Sz(ssl, keySz);
if(isSet != SSL_SUCCESS){
    Failed to set.
}
```

C.52.2.223 function wolfSSL_CTX_SetMinEccKey_Sz

```
int wolfSSL_CTX_SetMinEccKey_Sz(
    WOLFSSL_CTX * ssl,
    short keySz
)
```

wolf_ctx 構造体と wolfssl_cert_manager 構造体の ECC 鍵の最小サイズをビット単位で設定します。

Parameters:

- **ctx** wolfSSL_CTX_new()を使用して作成された WOLFSSL_CTX 構造体へのポインタ。

See: wolfSSL_SetMinEccKey_Sz

Return:

- SSL_SUCCESS 実行が成功したために返され、MineCkeysz メンバーが設定されます。
- BAD_FUNC_ARG WOLFSSL_CTX 構造体が null の場合、または鍵が負の場合、または 8 によって割り切れない場合に返されます。

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.52.2.224 function wolfSSL_SetMinEccKey_Sz

```
int wolfSSL_SetMinEccKey_Sz(
    WOLFSSL * ssl,
    short keySz
)
```

オプション構造の MineCckeysz メンバーの値を設定します。オプション構造体は、WolfSSL 構造のメンバーであり、SSL パラメータを介してアクセスされます。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ

See:

- wolfSSL_CTX_SetMinEccKey_Sz
- wolfSSL_CTX_SetMinRsaKey_Sz
- wolfSSL_SetMinRsaKey_Sz

Return:

- SSL_SUCCESS 関数がオプション構造の MineCckeysz メンバーを正常に設定した場合。
- BAD_FUNC_ARG WOLFSSL_CTX 構造体が null の場合、または鍵サイズ（keysz）が 0（ゼロ）未満の場合、または 8 で割り切れない場合。

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.52.2.225 function wolfSSL_make_eap_keys

```
int wolfSSL_make_eap_keys(
    WOLFSSL * ssl,
    void * key,
    unsigned int len,
```

```
    const char * label
)
```

この関数は、eap_tls と eap-ttls によって、マスターシークレットからキーイングマテリアルを導出します。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ
- **msk** p_hash 関数の結果を保持する void ポインタ変数。
- **len** MSK 変数の長さを表す符号なし整数。

See:

- wc_PRF
- wc_HmacFinal
- wc_HmacUpdate

Return:

- BUFFER_E バッファの実際のサイズが許容最大サイズを超える場合に返されます。
- MEMORY_E メモリ割り当てにエラーがある場合に返されます。

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);;
void* msk;
unsigned int len;
const char* label;
...
return wolfSSL_make_eap_keys(ssl, msk, len, label);
```

C.52.2.226 function wolfSSL_writev

```
int wolfSSL_writev(
    WOLFSSL * ssl,
    const struct iovec * iov,
    int iovcnt
)
```

Writev Semantics をシミュレートしますが、SSL_Write() の動作のために実際にはブロックしないため、フロント追加が小さくなる可能性があるため Writev を使いやすいソフトウェアに移植する。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ
- **iov** 書き込みへの I / O ベクトルの配列

See: wolfSSL_write

Return:

- 0 成功時に書かれたバイト数。
- 0 失敗したときに返されます。特定のエラーコードについて wolfSSL_get_error() を呼び出します。
- MEMORY_ERROR メモリエラーが発生した場合に返されます。
- SSL_FATAL_ERROR エラーが発生したとき、または非ブロッキングソケットを使用するときには、SSL_ERROR_WANT_READ または SSL_ERROR_WANT_WRITE エラーが受信され、再度 WOLFSSL_WRITE() を呼び出す必要がある場合は、障害が発生します。特定のエラーコードを取得するには、wolfSSL_get_error() を使用してください。

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.52.2.227 function wolfSSL_CTX_UnloadCAs

```
int wolfSSL_CTX_UnloadCAs(
    WOLFSSL_CTX *
)
```

この関数は CA 署名者リストをアンロードし、署名者全体のテーブルを解放します。

See:

- wolfSSL_CertManagerUnloadCAs
- LockMutex
- FreeSignerTable
- UnlockMutex

Return:

- SSL_SUCCESS 機能の実行に成功したことに戻ります。
- BAD_FUNC_ARG WOLFSSL_CTX 構造体が null の場合、または他の方法では未解決の引数値がサブルーチンに渡された場合に返されます。
- BAD_MUTEX_E ミューテックスエラーが発生した場合に返されます。lockmutex() は 0 を返しませんでした。

Example

```
WOLFSSL_METHOD method = wolfTLSv1_2_client_method();
WOLFSSL_CTX* ctx = wolfSSL_CTX_new(method);
…
if(!wolfSSL_CTX_UnloadCAs(ctx)){
    // The function did not unload CAs
}
```

C.52.2.228 function wolfSSL_CTX_Unload_trust_peers

```
int wolfSSL_CTX_Unload_trust_peers(
    WOLFSSL_CTX *
)
```

この関数は、以前にロードされたすべての信頼できるピア証明書をアンロードするために使用されます。マクロ wolfssl_trust_peer_cert を定義することで機能が有効になっています。

See:

- wolfSSL_CTX_trust_peer_buffer

- wolfSSL_CTX_trust_peer_cert

Return:

- SSL_SUCCESS 成功時に返されます。
- BAD_FUNC_ARG CTX が NULL の場合に返されます。
- SSL_BAD_FILE ファイルが存在しない場合に返されます。読み込め、または破損していません。
- MEMORY_E メモリ不足状態が発生した場合に返されます。

Example

```
int ret = 0;
WOLFSSL_CTX* ctx;
...
ret = wolfSSL_CTX_Unload_trust_peers(ctx);
if (ret != SSL_SUCCESS) {
    // error unloading trusted peer certs
}
...
```

C.52.2.229 function wolfSSL_CTX_trust_peer_buffer

```
int wolfSSL_CTX_trust_peer_buffer(
    WOLFSSL_CTX * ctx,
    const unsigned char * in,
    long sz,
    int format
)
```

この関数は、TLS / SSL ハンドシェイクを実行するときにピアを検証するために使用する証明書をロードします。ハンドシェイク中に送信されたピア証明書は、使用可能なときにスキッドを使用することによって比較されます。これら 2 つのことが一致しない場合は、ロードされた CAS が使用されます。ファイルの代わりにバッファーの場合は、wolfssl_ctx_trust_peer_cert と同じ機能です。特徴はマクロ wolfssl_trust_peer_cert を定義することによって有効になっています適切な使用法の例を参照してください。

Parameters:

- **ctx** wolfSSL_CTX_new()で作成された SSL コンテキストへのポインタ。
- **buffer** 証明書を含むバッファへのポインタ。
- **sz** バッファ入力の長さ。

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 成功すると
- SSL_FAILURE CTX が NULL の場合、または両方のファイルと種類が無効な場合に返されます。
- SSL_BAD_FILETYPE ファイルが間違った形式である場合に返されます。
- SSL_BAD_FILE ファイルが存在しない場合に返されます。読み込め、または破損していません。
- MEMORY_E メモリ不足状態が発生した場合に返されます。

- ASN_INPUT_E base16 デコードがファイルに対して失敗した場合に返されます。

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.52.2.230 function wolfSSL_CTX_load_verify_buffer

```
int wolfSSL_CTX_load_verify_buffer(
    WOLFSSL_CTX * ctx,
    const unsigned char * in,
    long sz,
    int format
)
```

この関数は CA 証明書バッファを WolfSSL コンテキストにロードします。バッファ以外のバージョンのように動作し、ファイルの代わりに入力としてバッファと呼ばれる機能が異なるだけです。バッファはサイズ SZ の引数によって提供されます。形式バッファのフォーマットタイプを指定します。SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM。フォーマットが PEM 内にある限り、バッファあたり複数の CA 証明書をロードすることができます。適切な使用法の例をご覧ください。

Parameters:

- **ctx** wolfSSL_CTX_new()で作成された SSL コンテキストへのポインタ。
- **in** CA 証明書バッファへのポインタ。
- **sz** 入力 CA 証明書バッファのサイズ、IN。

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 成功すると
- SSL_BAD_FILETYPE ファイルが間違った形式である場合に返されます。
- SSL_BAD_FILE ファイルが存在しない場合に返されます。読み込め、または破損していません。
- MEMORY_E メモリ不足状態が発生した場合に返されます。
- ASN_INPUT_E base16 デコードがファイルに対して失敗した場合に返されます。
- BUFFER_E チェーンバッファが受信バッファよりも大きい場合に返されます。

Example

```
int ret = 0;
int sz = 0;
WOLFSSL_CTX* ctx;
```

```
byte certBuff[...];
...

ret = wolfSSL_CTX_load_verify_buffer(ctx, certBuff, sz, SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
    // error loading CA certs from buffer
}
...
```

C.52.2.231 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
)
```

この関数は CA 証明書バッファを WolfSSL コンテキストにロードします。バッファ以外のバージョンのように動作し、ファイルの代わりに入力としてバッファと呼ばれる機能が異なるだけです。バッファはサイズ SZ の引数によって提供されます。形式バッファのフォーマットタイプを指定します。SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM。フォーマットが PEM 内にある限り、バッファあたり複数の CA 証明書をロードすることができます。_EX バージョンは PR 2413 に追加され、UserChain と Flags の追加の引数をサポートします。

Parameters:

- **ctx** wolfSSL_CTX_new()で作成された SSL コンテキストへのポインタ。
- **in** CA 証明書バッファへのポインタ。
- **sz** 入力 CA 証明書バッファのサイズ、IN。
- **format** バッファ証明書の形式、SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM。
- **userChain** フォーマット wolfssl_filetype_asn1 を使用する場合、このセットはゼロ以外のセットを示しています.Der のチェーンが表示されています。

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 成功すると
- SSL_BAD_FILETYPE ファイルが間違った形式である場合に返されます。
- SSL_BAD_FILE ファイルが存在しない場合に返されます。読み込め、または破損していません。
- MEMORY_E メモリ不足状態が発生した場合に返されます。
- ASN_INPUT_E base16 デコードがファイルに対して失敗した場合に返されます。
- BUFFER_E チェーンバッファが受信バッファよりも大きい場合に返されます。

Example

```
int ret = 0;
int sz = 0;
WOLFSSL_CTX* ctx;
byte 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.52.2.232 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
)
```

この関数は、CA 証明書チェーンバッファを WolfSSL コンテキストにロードします。バッファ以外のバージョンのように動作し、ファイルの代わりに入力としてバッファと呼ばれる機能が異なるだけです。バッファはサイズ SZ の引数によって提供されます。形式バッファのフォーマットタイプを指定します。SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM。フォーマットが PEM 内にある限り、バッファあたり複数の CA 証明書をロードすることができます。適切な使用法の例をご覧ください。

Parameters:

- **ctx** wolfSSL_CTX_new()で作成された SSL コンテキストへのポインタ。
- **in** CA 証明書バッファへのポインタ。
- **sz** 入力 CA 証明書バッファのサイズ、IN。

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 成功すると
- SSL_BAD_FILETYPE ファイルが間違った形式である場合に返されます。
- SSL_BAD_FILE ファイルが存在しない場合に返されます。読み込め、または破損していません。
- MEMORY_E メモリ不足状態が発生した場合に返されます。
- ASN_INPUT_E base16 デコードがファイルに対して失敗した場合に返されます。
- BUFFER_E チェーンバッファが受信バッファよりも大きい場合に返されます。

Example

```
int ret = 0;
int sz = 0;
WOLFSSL_CTX* ctx;
```

```
byte 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.52.2.233 function wolfSSL_CTX_use_certificate_buffer

```
int wolfSSL_CTX_use_certificate_buffer(
    WOLFSSL_CTX * ctx,
    const unsigned char * in,
    long sz,
    int format
)
```

この関数は証明書バッファを WolfSSL コンテキストにロードします。バッファ以外のバージョンのように動作し、ファイルの代わりに入力としてバッファと呼ばれる機能が異なるだけです。バッファはサイズ SZ の引数によって提供されます。形式バッファのフォーマットタイプを指定します。SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM。適切な使用法の例をご覧ください。

Parameters:

- **ctx** wolfSSL_CTX_new()で作成された SSL コンテキストへのポインタ。
- **in** ロードする証明書を含む入力バッファ。
- **sz** 入力バッファのサイズ。

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 成功すると
- SSL_BAD_FILETYPE ファイルが間違った形式である場合に返されます。
- SSL_BAD_FILE ファイルが存在しない場合に返されます。読み込め、または破損していません。
- MEMORY_E メモリ不足状態が発生した場合に返されます。
- ASN_INPUT_E base16 デコードがファイルに対して失敗した場合に返されます。

Example

```
int ret = 0;
int sz = 0;
WOLFSSL_CTX* ctx;
byte certBuff[...];
...
ret = wolfSSL_CTX_use_certificate_buffer(ctx, certBuff, sz, SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
    // error loading certificate from buffer
}
...
```

C.52.2.234 function wolfSSL_CTX_use_PrivateKey_buffer

```
int wolfSSL_CTX_use_PrivateKey_buffer(
    WOLFSSL_CTX * ctx,
    const unsigned char * in,
    long sz,
    int format
)
```

この関数は、秘密鍵バッファを SSL コンテキストにロードします。バッファ以外のバージョンのように動作し、ファイルの代わりに入力としてバッファと呼ばれる機能が異なるだけです。バッファはサイズ SZ の引数によって提供されます。形式バッファのフォーマットタイプを指定します。SSL_FILETYPE_ASN1OR SSL_FILETYPE_PEM。適切な使用法の例をご覧ください。

Parameters:

- **ctx** wolfSSL_CTX_new()で作成された SSL コンテキストへのポインタ。
- **in** ロードする秘密鍵を含む入力バッファ。
- **sz** 入力バッファのサイズ。

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 成功すると
- SSL_BAD_FILETYPE ファイルが間違った形式である場合に返されます。
- SSL_BAD_FILE ファイルが存在しない場合に返されます。読み込め、または破損していません。
- MEMORY_E メモリ不足状態が発生した場合に返されます。
- ASN_INPUT_E base16 デコードがファイルに対して失敗した場合に返されます。
- NO_PASSWORD 鍵ファイルが暗号化されているがパスワードが提供されていない場合に返されます。

Example

```
int ret = 0;
int sz = 0;
WOLFSSL_CTX* ctx;
byte keyBuff[...];
...
ret = wolfSSL_CTX_use_PrivateKey_buffer(ctx, keyBuff, sz, SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
    // error loading private key from buffer
}
...
```

C.52.2.235 function wolfSSL_CTX_use_certificate_chain_buffer

```
int wolfSSL_CTX_use_certificate_chain_buffer(
    WOLFSSL_CTX * ctx,
    const unsigned char * in,
    long sz
)
```

この関数は、証明書チェーンバッファを WolfSSL コンテキストにロードします。バッファ以外のバージョンのように動作し、ファイルの代わりに入力としてバッファと呼ばれる機能が異なるだけです。バッファはサイズ SZ の引数によって提供されます。バッファは PEM 形式で、ルート証明書で終わる対象の証明書から始めてください。適切な使用法の例をご覧ください。

Parameters:

- **ctx** wolfSSL_CTX_new()で作成された SSL コンテキストへのポインタ。
- **in** ロードされる PEM 形式の証明書チェーンを含む入力バッファ。

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 成功すると
- SSL_BAD_FILETYPE ファイルが間違った形式である場合に返されます。
- SSL_BAD_FILE ファイルが存在しない場合に返されます。読み込め、または破損していません。
- MEMORY_E メモリ不足状態が発生した場合に返されます。
- ASN_INPUT_E base16 デコードがファイルに対して失敗した場合に返されます。
- BUFFER_E チェーンバッファが受信バッファよりも大きい場合に返されます。

Example

```
int ret = 0;
int sz = 0;
WOLFSSL_CTX* ctx;
byte certChainBuff[...];
...
ret = wolfSSL_CTX_use_certificate_chain_buffer(ctx, certChainBuff, sz);
if (ret != SSL_SUCCESS) {
    // error loading certificate chain from buffer
}
...
```

C.52.2.236 function wolfSSL_use_certificate_buffer

```
int wolfSSL_use_certificate_buffer(
    WOLFSSL * ssl,
    const unsigned char * in,
    long sz,
    int format
)
```

この関数は、証明書バッファを WolfSSL オブジェクトにロードします。バッファ以外のバージョンのように動作し、ファイルの代わりに入力としてバッファと呼ばれる機能が異なるだけです。バッファはサイズ SZ の引数によって提供されます。形式バッファのフォーマットタイプを指定します。SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM。適切な使用法の例をご覧ください。

Parameters:

- **ssl** wolfSSL_new()で作成された SSL セッションへのポインタ。
- **in** ロードする証明書を含むバッファ。
- **sz** バッファにある証明書のサイズ。

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 成功時に返されます。
- SSL_BAD_FILETYPE ファイルが間違った形式である場合に返されます。
- SSL_BAD_FILE ファイルが存在しない場合に返されます。読み込め、または破損していません。
- MEMORY_E メモリ不足状態が発生した場合に返されます。
- ASN_INPUT_E base16 デコードがファイルに対して失敗した場合に返されます。

Example

```
int buffSz;
int ret;
byte certBuff[...];
WOLFSSL* ssl = 0;
...

ret = wolfSSL_use_certificate_buffer(ssl, certBuff, buffSz, SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
    // failed to load certificate from buffer
}
```

C.52.2.237 function wolfSSL_use_PrivateKey_buffer

```
int wolfSSL_use_PrivateKey_buffer(
    WOLFSSL * ssl,
    const unsigned char * in,
    long sz,
    int format
)
```

この関数は、秘密鍵バッファを WolfSSL オブジェクトにロードします。バッファ以外のバージョンのように動作し、ファイルの代わりに入力としてバッファと呼ばれる機能が異なるだけです。バッファはサイズ SZ の引数によって提供されます。形式バッファのフォーマットタイプを指定します。SSL_FILETYPE_ASN1 または SSL_FILETYPE_PEM。適切な使用法の例をご覧ください。

Parameters:

- **ssl** wolfssl_new() で作成された SSL セッションへのポインタ。
- **in** ロードする秘密鍵を含むバッファ。
- **sz** バッファにある秘密鍵のサイズ。

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 成功時に返されます。
- SSL_BAD_FILETYPE ファイルが間違った形式である場合に返されます。
- SSL_BAD_FILE ファイルが存在しない場合に返されます。読み込め、または破損していません。
- MEMORY_E メモリ不足状態が発生した場合に返されます。
- ASN_INPUT_E base16 デコードがファイルに対して失敗した場合に返されます。
- NO_PASSWORD 鍵ファイルが暗号化されているがパスワードが提供されていない場合に返されます。

Example

```
int buffSz;
int ret;
byte keyBuff[...];
WOLFSSL* ssl = 0;
...
ret = wolfSSL_use_PrivateKey_buffer(ssl, keyBuff, buffSz, SSL_FILETYPE_PEM);
if (ret != SSL_SUCCESS) {
    // failed to load private key from buffer
}
```

C.52.2.238 function wolfSSL_use_certificate_chain_buffer

```
int wolfSSL_use_certificate_chain_buffer(
    WOLFSSL * ssl,
    const unsigned char * in,
    long sz
)
```

この関数は、証明書チェーンバッファを WolfSSL オブジェクトにロードします。バッファ以外のバージョンのように動作し、ファイルの代わりに入力としてバッファと呼ばれる機能が異なるだけです。バッファはサイズ SZ の引数によって提供されます。バッファは PEM 形式で、ルート証明書で終わる対象の証明書から始めてください。適切な使用法の例をご覧ください。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ
- **in** ロードする証明書を含むバッファ。

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 成功時に返されます。
- SSL_BAD_FILETYPE ファイルが間違った形式である場合に返されます。
- SSL_BAD_FILE ファイルが存在しない場合に返されます。読み込め、または破損していません。
- MEMORY_E メモリ不足状態が発生した場合に返されます。
- ASN_INPUT_E base16 デコードがファイルに対して失敗した場合に返されます。
- BUFFER_E チェーンバッファが受信バッファよりも大きい場合に返されます。

Example

```
int buffSz;
int ret;
byte certChainBuff[...];
WOLFSSL* ssl = 0;
...
ret = wolfSSL_use_certificate_chain_buffer(ssl, certChainBuff, buffSz);
if (ret != SSL_SUCCESS) {
    // failed to load certificate chain from buffer
}
```

C.52.2.239 function wolfSSL_UnloadCertsKeys

```
int wolfSSL_UnloadCertsKeys(
    WOLFSSL *
)
```

この関数は、SSL が所有する証明書または鍵をアンロードします。

See: wolfSSL_CTX_UnloadCAs

Return:

- SSL_SUCCESS - 関数が正常に実行された場合に返されます。
- BAD_FUNC_ARG - wolfssl オブジェクトが null の場合に返されます。

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
...
int unloadKeys = wolfSSL_UnloadCertsKeys(ssl);
if(unloadKeys != SSL_SUCCESS){
    // Failure case.
}
```

C.52.2.240 function wolfSSL_CTX_set_group_messages

```
int wolfSSL_CTX_set_group_messages(
    WOLFSSL_CTX *
)
```

この機能は、可能な限りハンドシェイクメッセージのグループ化をオンにします。

See:

- wolfSSL_set_group_messages
- wolfSSL_CTX_new

Return:

- SSL_SUCCESS 成功に戻ります。
- BAD_FUNC_ARG 入力コンテキストが NULL の場合、返されます。

Example

```
WOLFSSL_CTX* ctx = 0;
...
ret = wolfSSL_CTX_set_group_messages(ctx);
if (ret != SSL_SUCCESS) {
    // failed to set handshake message grouping
}
```

C.52.2.241 function wolfSSL_set_group_messages

```
int wolfSSL_set_group_messages(
    WOLFSSL *
)
```

この機能は、可能な限りハンドシェイクメッセージのグループ化をオンにします。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ

See:

- wolfSSL_CTX_set_group_messages
- wolfSSL_new

Return:

- SSL_SUCCESS 成功に戻ります。
- BAD_FUNC_ARG 入力コンテキストが NULL の場合、返されます。

Example

```
WOLFSSL* ssl = 0;
...
ret = wolfSSL_set_group_messages(ssl);
if (ret != SSL_SUCCESS) {
// failed to set handshake message grouping
}
```

C.52.2.242 function wolfSSL_SetFuzzerCb

```
void wolfSSL_SetFuzzerCb(
    WOLFSSL * ssl,
    CallbackFuzzer cbf,
    void * fCtx
)
```

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WolfSSL 構造へのポインタ。
- **cbf** フォームの関数ポインタである CallBackFozzer タイプ：int（* callbackfuzzer）（wolfssl * ssl、consigned char * buf、int sz、int 型、void * fuzzctx）;
- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ

See: CallbackFuzzer

Return: none いいえ返します。

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.52.2.243 function wolfSSL_DTLS_SetCookieSecret

```
int wolfSSL_DTLS_SetCookieSecret(
    WOLFSSL * ssl,
    const unsigned char * secret,
    unsigned int secretSz
)
```

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ
- **secret** 秘密バッファを表す定数バイトポインタ。

See:

- ForceZero
- wc_RNG_GenerateBlock

Return:

- 0 関数がエラーなしで実行された場合に返されます。
- BAD_FUNC_ARG 許容できない値で関数に渡された引数があった場合に返されます。
- COOKIE_SECRET_SZ 秘密サイズが 0 の場合に返されます。
- MEMORY_ERROR 新しい Cookie Secret にメモリを割り当てる問題がある場合は返されました。

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.52.2.244 function wolfSSL_GetRNG

```
WC_RNG * wolfSSL_GetRNG(
    WOLFSSL * ssl
)
```

See: wolfSSL_CTX_new_rng

Return:

- rng 成功時に返されます。
- NULL ssl が NULL の場合 *Example*

```
WOLFSSL* ssl;

wolfSSL_GetRNG(ssl);
```

C.52.2.245 function wolfSSL_CTX_SetMinVersion

```
int wolfSSL_CTX_SetMinVersion(
    WOLFSSL_CTX * ctx,
    int version
)
```

この関数は、許可されている最小のダウングレードバージョンを設定します。接続が（wolfsslv23_client_method または wolfsslv23_server_method）を使用して、接続がダウングレードできる場合にのみ適用されます。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ

See: SetMinVersionHelper

Return:

- SSL_SUCCESS エラーなしで返された関数と最小バージョンが設定されている場合に返されます。
- BAD_FUNC_ARG WOLFSSL_CTX 構造が NULL の場合、または最小バージョンがサポートされていない場合に返されます。

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.52.2.246 function wolfSSL_SetMinVersion

```
int wolfSSL_SetMinVersion(
    WOLFSSL * ssl,
    int version
)
```

この関数は、許可されている最小のダウングレードバージョンを設定します。接続が（wolfsslv23_client_method または wolfsslv23_server_method）を使用して、接続がダウングレードできる場合にのみ適用されます。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ

See: SetMinVersionHelper

Return:

- SSL_SUCCESS この関数とそのサブルーチンがエラーなしで実行された場合に返されます。
- BAD_FUNC_ARG SSL オブジェクトが NULL の場合に返されます。サブルーチンでは、良いバージョンが一致しない場合、このエラーはスローされます。

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.52.2.247 function wolfSSL_GetObjectSize

```
int wolfSSL_GetObjectSize(
    void
)
```

ビルドオプションと設定に依存します。WolfSSL を構築するときに show_sizes が定義されている場合、この関数は WolfSSL オブジェクト（スイート、暗号など）内の個々のオブジェクトのサイズも stdout に印刷されます。

See: wolfSSL_new

Return: size この関数は、WolfSSL オブジェクトのサイズを返します。

Example

```
int size = 0;
size = wolfSSL_GetObjectSize();
printf("sizeof(WOLFSSL) = %d\n", size);
```

C.52.2.248 function wolfSSL_GetOutputSize

```
int wolfSSL_GetOutputSize(
    WOLFSSL * ssl,
    int inSz
)
```

アプリケーションがトランスポートレイヤ間で何バイトを送信したい場合は、指定された平文の入力サイズを指定してください。SSL / TLS ハンドシェイクが完了した後に呼び出す必要があります。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ

See: wolfSSL_GetMaxOutputSize

Return:

- size 成功すると、要求されたサイズが返されます
- INPUT_SIZE_E 入力サイズが最大 TLS フラグメントサイズより大きい場合は返されます（WOLFSSL_GETMAXOUTPUTSIZE()）。
- BAD_FUNC_ARG 無効な関数引数に戻り、または SSL / TLS ハンドシェイクがまだ完了していない場合

Example

none

C.52.2.249 function wolfSSL_GetMaxOutputSize

```
int wolfSSL_GetMaxOutputSize(
    WOLFSSL *
)
```

プロトコル規格で指定されている最大 SSL / TLS レコードサイズのいずれかに対応します。この関数は、アプリケーションが wolfssl_getOutputSize() と呼ばれ、input_size_e エラーを受信したときに役立ちます。SSL / TLS ハンドシェイクが完了した後に呼び出す必要があります。

See: wolfSSL_GetOutputSize

Return:

- size 成功すると、最大出力サイズが返されます

- BAD_FUNC_ARG 無効な関数引数のときに返されるか、SSL / TLS ハンドシェイクがまだ完了していない場合。

Example

none

C.52.2.250 function wolfSSL_SetVersion

```
int wolfSSL_SetVersion(
    WOLFSSL * ssl,
    int version
)
```

この関数は、バージョンで指定されたバージョンを使用して、指定された SSL セッション（WolfSSL オブジェクト）の SSL/TLS プロトコルバージョンを設定します。これにより、SSL セッション（SSL）のプロトコル設定が最初に定義され、SSL コンテキスト（wolfSSL_CTX_new()）メソッドの種類によって上書きされます。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WolfSSL 構造へのポインタ。

See: wolfSSL_CTX_new

Return:

- SSL_SUCCESS 成功時に返されます。
- BAD_FUNC_ARG 入力 SSL オブジェクトが NULL または誤ったプロトコルバージョンがバージョンで指定されている場合に返されます。

Example

```
int ret = 0;
WOLFSSL* ssl;
...

ret = wolfSSL_SetVersion(ssl, WOLFSSL_TLSV1);
if (ret != SSL_SUCCESS) {
    // failed to set SSL session protocol version
}
```

C.52.2.251 function wolfSSL_CTX_SetMacEncryptCb

```
void wolfSSL_CTX_SetMacEncryptCb(
    WOLFSSL_CTX * ctx,
    CallbackMacEncrypti cb
)
```

MAC /暗号化コールバック。コールバックは成功の場合は 0 を返すか、エラーの場合は <0 です。SSL と CTX ポインタはユーザーの利便性に利用できます。MacOut は、MAC の結果を保存する必要がある出力バッファです。Macin は Mac 入力バッファーと Macinsz のサイズを注意しています。MacContent と Macverify は、Wolfssl_SettlShmacinner() に必要であり、そのまま通過します。Encout は、暗号化の結果を格納する必要がある出力バッファです。ENCIN は ENCSZ が入力のサイズである間は暗号化する入力バッファです。コールバックの例は、wolfssl / test.h mymacencryptcb() を見つけることができます。

See:

- wolfSSL_SetMacEncryptCtx
- wolfSSL_GetMacEncryptCtx

Return: none 返品不可。

Example

none

C.52.2.252 function wolfSSL_SetMacEncryptCtx

```
void wolfSSL_SetMacEncryptCtx(
    WOLFSSL * ssl,
    void * ctx
)
```

CTX へのコールバックコンテキスト。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ

See:

- wolfSSL_CTX_SetMacEncryptCb
- wolfSSL_GetMacEncryptCtx

Return: none 返品不可。

Example

none

C.52.2.253 function wolfSSL_GetMacEncryptCtx

```
void * wolfSSL_GetMacEncryptCtx(
    WOLFSSL * ssl
)
```

Mac / Encrypt コールバックコンテキストは、wolfssl_setmacencryptx() で保存されていました。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ

See:

- wolfSSL_CTX_SetMacEncryptCb
- wolfSSL_SetMacEncryptCtx

Return:

- pointer 正常にコールがコンテキストへの有効なポインタを返します。
- NULL 空白のコンテキストのために返されます。

Example

none

C.52.2.254 function wolfSSL_CTX_SetDecryptVerifyCb

```
void wolfSSL_CTX_SetDecryptVerifyCb(
    WOLFSSL_CTX * ctx,
    CallbackDecryptVerify cb
)
```

コールバックを復号化/確認します。コールバックは成功の場合は 0 を返すか、エラーの場合は <0 です。SSL と CTX ポインタはユーザーの利便性に利用できます。DECOUT は、復号化の結果を格納する出力バッファです。DECIN は暗号化された入力バッファーと Decinsz のサイズを注意しています。コンテンツと検証は、WolfSSL_SettlShmacinner() に必要であり、そのまま通過します。PADSZ は、パディングの合計値で設定する出力変数です。つまり、MAC サイズとパディングバイトとパッドバイトを加えています。コールバックの例は、wolfssl / test.h mydecryptverifycb() を見つけることができます。

See:

- wolfSSL_SetMacEncryptCtx
- wolfSSL_GetMacEncryptCtx

Return: none いいえ返します。

Example

none

C.52.2.255 function wolfSSL_SetDecryptVerifyCtx

```
void wolfSSL_SetDecryptVerifyCtx(
    WOLFSSL * ssl,
    void * ctx
)
```

コールバックコンテキストを CTX に復号化/検証します。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ

See:

- wolfSSL_CTX_SetDecryptVerifyCb
- wolfSSL_GetDecryptVerifyCtx

Return: none いいえ返します。

Example

none

C.52.2.256 function wolfSSL_GetDecryptVerifyCtx

```
void * wolfSSL_GetDecryptVerifyCtx(
    WOLFSSL * ssl
)
```

wolfssl_setdecryptverifyctx() で以前に保存されているコールバックコンテキストを復号化/検証します。

See:

- wolfSSL_CTX_SetDecryptVerifyCb
- wolfSSL_SetDecryptVerifyCtx

Return:

- pointer 正常にコールがコンテキストへの有効なポインタを返します。
- NULL 空白のコンテキストのために返されます。

Example

none

C.52.2.257 function wolfSSL_GetMacSecret

```
const unsigned char * wolfSSL_GetMacSecret(
    WOLFSSL * ssl,
    int verify
)
```

VERIFY パラメーターは、これがピア・メッセージの検証のためのものであるかどうかを指定します。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ

See: wolfSSL_GetHmacSize

Return:

- pointer 正常にコールが秘密に有効なポインタを返します。秘密のサイズは、Wolfssl_gethmacsize() から入手できます。
- NULL エラー状態に戻ります。

Example

none

C.52.2.258 function wolfSSL_GetClientWriteKey

```
const unsigned char * wolfSSL_GetClientWriteKey(
    WOLFSSL *
)
```

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ

See:

- wolfSSL_GetKeySize
- wolfSSL_GetClientWriteIV

Return:

- pointer 正常にコールがキーへの有効なポインタを返します。鍵のサイズは、wolfssl_getkeysize() から取得できます。
- NULL エラー状態に戻ります。

Example

none

C.52.2.259 function wolfSSL_GetClientWriteIV

```
const unsigned char * wolfSSL_GetClientWriteIV(
    WOLFSSL *
)
```

ハンドシェイクプロセスから。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ

See:

- wolfSSL_GetCipherBlockSize()

- wolfSSL_GetClientWriteKey()

Return:

- pointer 正常にコールが IV への有効なポインタを返します。IV のサイズは、wolfssl_getCipherBlockSize() から取得できます。
- NULL エラー状態に戻ります。

Example

none

C.52.2.260 function wolfSSL_GetServerWriteKey

```
const unsigned char * wolfSSL_GetServerWriteKey(
    WOLFSSL *
)
```

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ

See:

- wolfSSL_GetKeySize
- wolfSSL_GetServerWriteIV

Return:

- pointer 正常にコールが鍵への有効なポインタを返します。鍵のサイズは、wolfssl_getkeysize() から取得できます。
- NULL エラー状態に戻ります。

Example

none

C.52.2.261 function wolfSSL_GetServerWriteIV

```
const unsigned char * wolfSSL_GetServerWriteIV(
    WOLFSSL *
)
```

ハンドシェイクプロセスから。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ

See:

- wolfSSL_GetCipherBlockSize
- wolfSSL_GetClientWriteKey

Return:

- pointer 正常にコールが IV への有効なポインタを返します。IV のサイズは、wolfssl_getCipherBlockSize() から取得できます。
- NULL エラー状態に戻ります。

C.52.2.262 function wolfSSL_GetKeySize

```
int wolfSSL_GetKeySize(
    WOLFSSL *
)
```

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ

See:

- wolfSSL_GetClientWriteKey
- wolfSSL_GetServerWriteKey

Return:

- size 正常にコールが鍵サイズをバイト単位で返します。
- BAD_FUNC_ARG エラー状態に戻ります。

Example

none

C.52.2.263 function wolfSSL_GetIVSize

```
int wolfSSL_GetIVSize(
    WOLFSSL *
)
```

WolfSSL 構造体に保持されている Specs 構造体の IV_SIZE メンバーを返します。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ

See:

- wolfSSL_GetKeySize
- wolfSSL_GetClientWriteIV
- wolfSSL_GetServerWriteIV

Return:

- iv_size ssl-> specs.iv_size で保持されている値を返します。
- BAD_FUNC_ARG WolfSSL 構造が NULL の場合に返されます。

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.52.2.264 function wolfSSL_GetSide

```
int wolfSSL_GetSide(
    WOLFSSL *
)
```

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ

See:

- wolfSSL_GetClientWriteKey
- wolfSSL_GetServerWriteKey

Return:

- success 成功した場合、呼び出しが WolfSSL オブジェクトの側面に応じて wolfssl_server_end または wolfssl_client_end を返します。
- BAD_FUNC_ARG エラー状態に戻ります。

Example

none

C.52.2.265 function wolfSSL_IsTLSv1_1

```
int wolfSSL_IsTLSv1_1(
    WOLFSSL *
)
```

少なくとも TLS バージョン 1.1 以上です。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ

See: wolfSSL_GetSide

Return:

- true/false 成功した場合、呼び出しが TRUE または 0 の場合は 0 を返します。
- BAD_FUNC_ARG エラー状態に戻ります。

Example

none

C.52.2.266 function wolfSSL_GetBulkCipher

```
int wolfSSL_GetBulkCipher(
    WOLFSSL *
)
```

ハンドシェイクから。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ

See:

- wolfSSL_GetCipherBlockSize
- wolfSSL_GetKeySize

Return:

- If コールが成功すると、wolfssl_cipher_null、wolfssl_des、wolfssl_triple_des、wolfssl_aes、wolfssl_aes_gcm、wolfssl_aes_ccm、wolfssl_camellia。
- BAD_FUNC_ARG エラー状態に戻ります。

Example

none

C.52.2.267 function wolfSSL_GetCipherBlockSize

```
int wolfSSL_GetCipherBlockSize(
    WOLFSSL *
)
```

ハンドシェイク。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ

See:

- wolfSSL_GetBulkCipher
- wolfSSL_GetKeySize

Return:

- size 正常にコールが暗号ブロックサイズのサイズをバイト単位で戻します。
- BAD_FUNC_ARG エラー状態に戻ります。

Example

none

C.52.2.268 function wolfSSL_GetAeadMacSize

```
int wolfSSL_GetAeadMacSize(
    WOLFSSL *
)
```

ハンドシェーク。暗号タイプの wolfssl_aead_type の場合。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ

See:

- wolfSSL_GetBulkCipher
- wolfSSL_GetKeySize

Return:

- size 正常にコールが EAD MAC サイズのサイズをバイト単位で戻します。
- BAD_FUNC_ARG エラー状態に戻ります。

Example

none

C.52.2.269 function wolfSSL_GetHmacSize

```
int wolfSSL_GetHmacSize(
    WOLFSSL *
)
```

ハンドシェーク。wolfssl_aead_type 以外の暗号タイプの場合。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ

See:

- wolfSSL_GetBulkCipher
- wolfSSL_GetHmacType

Return:

- size 正常にコールが（H）MAC サイズのサイズをバイト単位で戻します。
- BAD_FUNC_ARG エラー状態に戻ります。

Example

none

C.52.2.270 function wolfSSL_GetHmacType

```
int wolfSSL_GetHmacType(
    WOLFSSL *
)
```

ハンドシェーク。wolfssl_aead_type 以外の暗号タイプの場合。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ

See:

- wolfSSL_GetBulkCipher
- wolfSSL_GetHmacSize

Return:

- If コールが成功すると、次のいずれかが返されます.MD5、SHA、SHA256、SHA384。
- BAD_FUNC_ARG エラー状態に対して返される可能性があります。
- SSL_FATAL_ERROR エラー状態にも返される可能性があります。

Example

none

C.52.2.271 function wolfSSL_GetCipherType

```
int wolfSSL_GetCipherType(
    WOLFSSL *
)
```

ハンドシェイクから。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ

See:

- wolfSSL_GetBulkCipher
- wolfSSL_GetHmacType

Return:

- If 正常にコールは次のいずれかを返します.WolfSSL_BLOCK_TYPE、WOLFSSL_STREAM_TYPE、WOLFSSL_AEAD_TYPE。
- BAD_FUNC_ARG エラー状態に戻ります。

Example

none

C.52.2.272 function wolfSSL_SetTlsHmacInner

```
int wolfSSL_SetTlsHmacInner(
    WOLFSSL * ssl,
    byte * inner,
    word32 sz,
    int content,
    int verify
)
```

送受信結果は、少なくとも wolfssl_gethmacsize() バイトであるべきである内部に書き込まれます。メッセージのサイズは SZ で指定され、内容はメッセージの種類であり、検証はこれがピアメッセージの検証であるかどうかを指定します。wolfssl_aead_type を除く暗号タイプに有効です。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ

See:

- wolfSSL_GetBulkCipher
- wolfSSL_GetHmacType

Return:

- 1 成功時に返されます。
- BAD_FUNC_ARG エラー状態に戻ります。

Example

none

C.52.2.273 function wolfSSL_CTX_SetEccSignCb

```
void wolfSSL_CTX_SetEccSignCb(
    WOLFSSL_CTX * ctx,
    CallbackEccSign cb
)
```

コールバックは成功の場合は 0 を返すか、エラーの場合は <0 です。SSL と CTX ポインタはユーザーの利便性に利用できます。INS は入力バッファーが入力の長さを表します。OUT は、署名の結果を保存する必要がある出力バッファです。OUTSZ は、呼び出し時に出力バッファのサイズを指定する入力/出力変数であり、署名の実際のサイズを戻す前に格納する必要があります。keyder は ASN1 フォーマットの ECC 秘密鍵であり、Keysz は鍵のキーの長さです。コールバックの例は、wolfssl / test.h myeccsign() を見つけることができます。

See:

- wolfSSL_SetEccSignCtx

- wolfSSL_GetEccSignCtx

Return: none いいえ返します。

Example

none

C.52.2.274 function wolfSSL_SetEccSignCtx

```
void wolfSSL_SetEccSignCtx(
    WOLFSSL * ssl,
    void * ctx
)
```

CTX へのコンテキスト。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ

See:

- wolfSSL_CTX_SetEccSignCb
- wolfSSL_GetEccSignCtx

Return: none いいえ返します。

Example

none

C.52.2.275 function wolfSSL_GetEccSignCtx

```
void * wolfSSL_GetEccSignCtx(
    WOLFSSL * ssl
)
```

以前に wolfssl_seteccsignctx() で保存されていたコンテキスト。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ

See:

- wolfSSL_CTX_SetEccSignCb
- wolfSSL_SetEccSignCtx

Return:

- pointer 正常にコールがコンテキストへの有効なポインタを返します。
- NULL 空白のコンテキストのために返されます。

Example

none

C.52.2.276 function wolfSSL_CTX_SetEccSignCtx

```
void wolfSSL_CTX_SetEccSignCtx(
    WOLFSSL_CTX * ctx,
    void * userCtx
)
```

CTX へのコンテキスト。

Parameters:

- **ctx** wolfSSL_CTX_new()で作成された WOLFSSL_CTX 構造体へのポインタ。

See:

- wolfSSL_CTX_SetEccSignCb
- wolfSSL_CTX_GetEccSignCtx

Return: none いいえ返します。

Example

none

C.52.2.277 function wolfSSL_CTX_GetEccSignCtx

```
void * wolfSSL_CTX_GetEccSignCtx(
    WOLFSSL_CTX * ctx
)
```

以前に wolfssl_seteccsignctx() で保存されていたコンテキスト。

See:

- wolfSSL_CTX_SetEccSignCb
- wolfSSL_CTX_SetEccSignCtx

Return:

- pointer 正常にコールがコンテキストへの有効なポインタを返します。
- NULL 空白のコンテキストのために返されます。

Example

none

C.52.2.278 function wolfSSL_CTX_SetEccVerifyCb

```
void wolfSSL_CTX_SetEccVerifyCb(
    WOLFSSL_CTX * ctx,
    CallbackEccVerify cb
)
```

コールバックは成功の場合は 0 を返すか、エラーの場合は <0 です。SSL と CTX ポインタはユーザーの利便性に利用できます。SIG は検証の署名であり、SIGSZ は署名の長さを表します。ハッシュはメッセージのダイジェストを含む入力バッファであり、HASHSZ はハッシュの長さを意味します。結果は、検証の結果を格納する出力変数、成功のために 1、失敗のために 0 を記憶する必要があります。keyder は ASN1 フォーマットの ECC 秘密鍵であり、Keysz はキーのキーの長さです。コールバックの例は、wolfssl / test.h myeccverify() を見つけることができます。

See:

- wolfSSL_SetEccVerifyCtx
- wolfSSL_GetEccVerifyCtx

Return: none いいえ返します。

Example

none

C.52.2.279 function wolfSSL_SetEccVerifyCtx

```
void wolfSSL_SetEccVerifyCtx(
    WOLFSSL * ssl,
    void * ctx
)
```

CTX へのコンテキスト。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ

See:

- wolfSSL_CTX_SetEccVerifyCb
- wolfSSL_GetEccVerifyCtx

Return: none いいえ返します。

Example

none

C.52.2.280 function wolfSSL_GetEccVerifyCtx

```
void * wolfSSL_GetEccVerifyCtx(
    WOLFSSL * ssl
)
```

以前に wolfssl_setecverifyctx() で保存されていたコンテキスト。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ

See:

- wolfSSL_CTX_SetEccVerifyCb
- wolfSSL_SetEccVerifyCtx

Return:

- pointer 正常にコールがコンテキストへの有効なポインタを返します。
- NULL 空白のコンテキストのために返されます。

Example

none

C.52.2.281 function wolfSSL_CTX_SetRsaSignCb

```
void wolfSSL_CTX_SetRsaSignCb(
    WOLFSSL_CTX * ctx,
    CallbackRsaSign cb
)
```

コールバックは成功の場合は 0 を返すか、エラーの場合は <0 です。SSL と CTX ポインタはユーザーの利便性に利用できます。INS は入力バッファーが入力の長さを表します。OUT は、署名の結果を保存する必要がある出力バッファです。OUTSZ は、呼び出し時に出力バッファのサイズを指定する入力/出力変数であり、署名の実際のサイズを戻す前に格納する必要があります。keyder は ASN1 フォーマットの RSA 秘密鍵であり、Keysz はバイト数のキーの長さです。コールバックの例は、wolfssl / test.h myrsasign() を見つけることができます。

See:

- wolfSSL_SetRsaSignCtx
- wolfSSL_GetRsaSignCtx

Return: none いいえ返します。

Example

none

C.52.2.282 function wolfSSL_SetRsaSignCtx

```
void wolfSSL_SetRsaSignCtx(
    WOLFSSL * ssl,
    void * ctx
)
```

ctx に。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ

See:

- wolfSSL_CTX_SetRsaSignCb
- wolfSSL_GetRsaSignCtx

Return: none いいえ返します。

Example

none

C.52.2.283 function wolfSSL_GetRsaSignCtx

```
void * wolfSSL_GetRsaSignCtx(
    WOLFSSL * ssl
)
```

以前に wolfssl_setrsAsignctx() で保存されていたコンテキスト。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ

See:

- wolfSSL_CTX_SetRsaSignCb
- wolfSSL_SetRsaSignCtx

Return:

- pointer 正常にコールがコンテキストへの有効なポインタを返します。
- NULL 空白のコンテキストのために返されます。

Example

none

C.52.2.284 function wolfSSL_CTX_SetRsaVerifyCb

```
void wolfSSL_CTX_SetRsaVerifyCb(
    WOLFSSL_CTX * ctx,
    CallbackRsaVerify cb
)
```

コールバックは、成功のための平文バイト数または <0 エラーの場合は <0 を返すべきです。SSL と CTX ポインタはユーザーの利便性に利用できます。SIG は検証の署名であり、SIGSZ は署名の長さを表します。復号化プロセスとパディングの後に検証バッファの先頭に設定する必要があります。keyder は ASN1 形式の RSA 公開鍵であり、Keysz はキーのキーの長さです。コールバックの例は、wolfssl / test.h myrsaverify() を見つけることができます。

See:

- wolfSSL_SetRsaVerifyCtx
- wolfSSL_GetRsaVerifyCtx

Return: none いいえ返します。

C.52.2.285 function wolfSSL_SetRsaVerifyCtx

```
void wolfSSL_SetRsaVerifyCtx(
    WOLFSSL * ssl,
    void * ctx
)
```

CTX へのコンテキスト。

See:

- wolfSSL_CTX_SetRsaVerifyCb
- wolfSSL_GetRsaVerifyCtx

Return: none いいえ返します。

Example

none

C.52.2.286 function wolfSSL_GetRsaVerifyCtx

```
void * wolfSSL_GetRsaVerifyCtx(
    WOLFSSL * ssl
)
```

以前に wolfssl_setrsaverifyctx() で保存されていたコンテキスト。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ

See:

- wolfSSL_CTX_SetRsaVerifyCb
- wolfSSL_SetRsaVerifyCtx

Return:

- pointer 正常にコールがコンテキストへの有効なポインタを返します。
- NULL 空白のコンテキストのために返されます。

Example

none

C.52.2.287 function wolfSSL_CTX_SetRsaEncCb

```
void wolfSSL_CTX_SetRsaEncCb(
    WOLFSSL_CTX * ctx,
    CallbackRsaEnc cb
)
```

暗号化します。コールバックは成功の場合は 0 を返すか、エラーの場合は <0 です。SSL と CTX ポインタはユーザーの利便性に利用できます。IN は入力バッファですが、INSZ は入力の長さを表します。暗号化の結果を保存する必要がある出力バッファです。OUTSZ は、呼び出し時に出力バッファのサイズを指定する入力/出力変数であり、暗号化の実際のサイズは戻って前に格納されるべきです。keyder は ASN1 形式の RSA 公開鍵であり、Keysz はキーのキーの長さです。例コールバックの例は、wolfssl / test.h myrsaenc() を見つけることができます。

See:

- wolfSSL_SetRsaEncCtx
- wolfSSL_GetRsaEncCtx

Return: none いいえ返します。

Example

none

C.52.2.288 function wolfSSL_SetRsaEncCtx

```
void wolfSSL_SetRsaEncCtx(
    WOLFSSL * ssl,
    void * ctx
)
```

CTX へのコールバックコンテキスト。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ

See:

- wolfSSL_CTX_SetRsaEncCb
- wolfSSL_GetRsaEncCtx

Return: none いいえ返します。

Example

none

C.52.2.289 function wolfSSL_GetRsaEncCtx

```
void * wolfSSL_GetRsaEncCtx(
    WOLFSSL * ssl
)
```

コールバックコンテキストは、wolfssl_setrsaencctx() で以前に保存されていました。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ

See:

- wolfSSL_CTX_SetRsaEncCb
- wolfSSL_SetRsaEncCtx

Return:

- pointer 正常にコールがコンテキストへの有効なポインタを返します。
- NULL 空白のコンテキストのために返されます。

Example

none

C.52.2.290 function wolfSSL_CTX_SetRsaDecCb

```
void wolfSSL_CTX_SetRsaDecCb(
    WOLFSSL_CTX * ctx,
    CallbackRsaDec cb
)
```

復号化します。コールバックは、成功のための平文バイト数または <0 エラーの場合は <0 を返すべきです。SSL と CTX ポインタはユーザーの利便性に利用できます。IN は、復号化する入力バッファが入力の長さを表します。復号化プロセスおよび任意のパディングの後、復号化バッファの先頭に設定する必要があります。keyder は ASN1 フォーマットの RSA 秘密鍵であり、Keysz はバイト数のキーの長さです。コールバックの例は、wolfssl / test.h myrsadec() を見つけることができます。

See:

- wolfSSL_SetRsaDecCtx
- wolfSSL_GetRsaDecCtx

Return: none いいえ返します。

Example

none

C.52.2.291 function wolfSSL_SetRsaDecCtx

```
void wolfSSL_SetRsaDecCtx(
    WOLFSSL * ssl,
    void * ctx
)
```

CTX へのコールバックコンテキスト。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ

See:

- wolfSSL_CTX_SetRsaDecCb
- wolfSSL_GetRsaDecCtx

Return: none いいえ返します。

Example

none

C.52.2.292 function wolfSSL_GetRsaDecCtx

```
void * wolfSSL_GetRsaDecCtx(
    WOLFSSL * ssl
)
```

コールバックコンテキストは、wolfssl_setrsadecctx() で以前に保存されていました。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ

See:

- wolfSSL_CTX_SetRsaDecCb
- wolfSSL_SetRsaDecCtx

Return:

- pointer 正常にコールがコンテキストへの有効なポインタを返します。
- NULL 空白のコンテキストのために返されます。

Example

none

C.52.2.293 function wolfSSL_CTX_SetCACb

```
void wolfSSL_CTX_SetCACb(
    WOLFSSL_CTX * ctx,
    CallbackCACache cb
)
```

新しい CA 証明書が WolfSSL にロードされたときに呼び出される（WolfSSL_CTX)。コールバックには、符号化された証明書を持つバッファが与えられます。

Parameters:

- **ctx** wolfSSL_CTX_new()で作成された SSL コンテキストへのポインタ。

See: wolfSSL_CTX_load_verify_locations

Return: none 返品不可。

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.52.2.294 function wolfSSL_CertManagerNew_ex

```
WOLFSSL_CERT_MANAGER * wolfSSL_CertManagerNew_ex(
    void * heap
)
```

新しい証明書マネージャコンテキストを割り当てて初期化します。このコンテキストは、SSL のニーズとは無関係に使用できます。証明書をロードしたり、証明書を確認したり、失効状況を確認したりするために使用することができます。

See: wolfSSL_CertManagerFree

Return:

- WOLFSSL_CERT_MANAGER 正常にコールが有効な wolfssl_cert_manager ポインタを返します。
- NULL エラー状態に戻ります。

C.52.2.295 function wolfSSL_CertManagerNew

```
WOLFSSL_CERT_MANAGER * wolfSSL_CertManagerNew(
    void
)
```

新しい証明書マネージャコンテキストを割り当てて初期化します。このコンテキストは、SSL のニーズとは無関係に使用できます。証明書をロードしたり、証明書を確認したり、失効状況を確認したりするために使用することができます。

See: wolfSSL_CertManagerFree

Return:

- WOLFSSL_CERT_MANAGER 正常にコールが有効な wolfssl_cert_manager ポインタを返します。
- NULL エラー状態に戻ります。

Example

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

WOLFSSL_CERT_MANAGER* cm;
cm = wolfSSL_CertManagerNew();
if (cm == NULL) {
// error creating new cert manager
}
```

C.52.2.296 function wolfSSL_CertManagerFree

```
void wolfSSL_CertManagerFree(
    WOLFSSL_CERT_MANAGER *
)
```

証明書マネージャのコンテキストに関連付けられているすべてのリソースを解放します。証明書マネージャを使用する必要がなくなるときにこれを呼び出します。

See: wolfSSL_CertManagerNew

Return: none

Example

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

WOLFSSL_CERT_MANAGER* cm;
```

```
...
wolfSSL_CertManagerFree(cm);
```

C.52.2.297 function wolfSSL_CertManagerLoadCA

```
int wolfSSL_CertManagerLoadCA(
    WOLFSSL_CERT_MANAGER * cm,
    const char * f,
    const char * d
)
```

Manager コンテキストへの CA 証明書のロードの場所を指定します。PEM 証明書カフェイルには、複数の信頼できる CA 証明書が含まれている可能性があります。capath が null でない場合、PEM 形式の CA 証明書を含むディレクトリを指定します。

Parameters:

- **cm** wolfssl_certmanagernew() を使用して作成された wolfssl_cert_manager 構造体へのポインタ。
- **file** ロードする CA 証明書を含むファイルの名前へのポインタ。

See: wolfSSL_CertManagerVerify

Return:

- SSL_SUCCESS 成功した場合に返されます。、通話が戻ります。
- SSL_BAD_FILETYPE ファイルが間違った形式である場合に返されます。
- SSL_BAD_FILE ファイルが存在しない場合に返されます。読み込め、または破損していません。
- MEMORY_E メモリ不足状態が発生した場合に返されます。
- ASN_INPUT_E base16 デコードがファイルに対して失敗した場合に返されます。
- BAD_FUNC_ARG ポインタが提供されていない場合に返されるエラーです。
- SSL_FATAL_ERROR - 失敗時に返されます。

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.52.2.298 function wolfSSL_CertManagerLoadCABuffer

```
int wolfSSL_CertManagerLoadCABuffer(
    WOLFSSL_CERT_MANAGER * cm,
    const unsigned char * in,
    long sz,
    int format
)
```

wolfssl_ctx_load_verify_buffer を呼び出して、関数に渡された CM 内の情報を失うことなく一時的な CM を使用してその結果を返すことによって CA バッファをロードします。

Parameters:

- **cm** wolfssl_certmanagernew() を使用して作成された wolfssl_cert_manager 構造体へのポインタ。
- **in** CERT 情報用のバッファー。

- **sz** バッファの長さ。

See:

- wolfSSL_CTX_load_verify_buffer
- ProcessChainBuffer
- ProcessBuffer
- cm_pick_method

Return:

- SSL_FATAL_ERROR wolfssl_cert_manager 構造体が NULL の場合、または wolfSSL_CTX_new() が NULL を返す場合に返されます。
- SSL_SUCCESS 実行が成功するために返されます。

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.52.2.299 function wolfSSL_CertManagerUnloadCAs

```
int wolfSSL_CertManagerUnloadCAs(
    WOLFSSL_CERT_MANAGER * cm
)
```

この関数は CA 署名者リストをアンロードします。

See:

- FreeSignerTable
- UnlockMutex

Return:

- SSL_SUCCESS 機能の実行に成功したことに戻ります。
- BAD_FUNC_ARG wolfssl_cert_manager が null の場合に返されます。
- BAD_MUTEX_E ミューテックスエラーが発生した場合に返されます。

Example

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

WOLFSSL_CTX* ctx = wolfSSL_CTX_new(protocol method);
WOLFSSL_CERT_MANAGER* cm = wolfSSL_CertManagerNew();
...
if(wolfSSL_CertManagerUnloadCAs(ctx->cm) != SSL_SUCCESS){
    Failure case.
}
```

C.52.2.300 function wolfSSL_CertManagerUnload_trust_peers

```
int wolfSSL_CertManagerUnload_trust_peers(
    WOLFSSL_CERT_MANAGER * cm
)
```

関数は信頼できるピアリンクリストを解放し、信頼できるピアリストのロックを解除します。

See: UnLockMutex

Return:

- SSL_SUCCESS 関数が正常に完了した場合
- BAD_FUNC_ARG wolfssl_cert_manager が null の場合
- BAD_MUTEX_E ミューテックスエラー TPLOCK では、WOLFSSL_CERT_MANAGER 構造体のメンバーは 0（ニル）です。

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.52.2.301 function wolfSSL_CertManagerVerify

```
int wolfSSL_CertManagerVerify(
    WOLFSSL_CERT_MANAGER * cm,
    const char * f,
    int format
)
```

証明書マネージャのコンテキストで確認する証明書を指定します。フォーマットは SSL_FILETYPE_PEM または SSL_FILETYPE_ASN1 にすることができます。

Parameters:

- **cm** wolfssl_certmanagernew() を使用して作成された wolfssl_cert_manager 構造体へのポインタ。
- **fname** 検証する証明書を含むファイルの名前へのポインタ。

See:

- wolfSSL_CertManagerLoadCA
- wolfSSL_CertManagerVerifyBuffer

Return:

- SSL_SUCCESS 成功した場合に返されます。
- ASN_SIG_CONFIRM_E 署名が検証できなかった場合に返されます。
- ASN_SIG_OID_E 署名の種類がサポートされていない場合に返されます。
- CRL_CERT_REVOKED この証明書が取り消された場合に返されるエラーです。
- CRL_MISSING 現在の発行者 CRL が利用できない場合に返されるエラーです。
- ASN_BEFORE_DATE_E 現在の日付が前日の前にある場合に返されます。
- ASN_AFTER_DATE_E 現在の日付が後の日付の後の場合に返されます。
- SSL_BAD_FILETYPE ファイルが間違った形式である場合に返されます。
- SSL_BAD_FILE ファイルが存在しない場合に返されます。読み込め、または破損していません。
- MEMORY_E メモリ不足状態が発生した場合に返されます。
- ASN_INPUT_E base16 デコードがファイルに対して失敗した場合に返されます。
- BAD_FUNC_ARG ポインタが提供されていない場合に返されるエラーです。

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.52.2.302 function wolfSSL_CertManagerVerifyBuffer

```
int wolfSSL_CertManagerVerifyBuffer(
    WOLFSSL_CERT_MANAGER * cm,
    const unsigned char * buff,
    long sz,
    int format
)
```

証明書マネージャのコンテキストを使用して確認する証明書バッファを指定します。フォーマットは SSL_FILETYPE_PEM または SSL_FILETYPE_ASN1 にすることができます。

Parameters:

- **cm** wolfssl_certmanagernew() を使用して作成された wolfssl_cert_manager 構造体へのポインタ。
- **buff** 検証する証明書を含むバッファ。
- **sz** バッファのサイズ、BUF。

See:

- wolfSSL_CertManagerLoadCA
- wolfSSL_CertManagerVerify

Return:

- SSL_SUCCESS 成功した場合に返されます。
- ASN_SIG_CONFIRM_E 署名が検証できなかった場合に返されます。
- ASN_SIG_OID_E 署名の種類がサポートされていない場合に返されます。
- CRL_CERT_REVOKED この証明書が取り消された場合に返されるエラーです。
- CRL_MISSING 現在の発行者 CRL が利用できない場合に返されるエラーです。
- ASN_BEFORE_DATE_E 現在の日付が前日の前にある場合に返されます。
- ASN_AFTER_DATE_E 現在の日付が後の日付の後の場合に返されます。
- SSL_BAD_FILETYPE ファイルが間違った形式である場合に返されます。
- SSL_BAD_FILE ファイルが存在しない場合に返されます。読み込め、または破損していません。
- MEMORY_E メモリ不足状態が発生した場合に返されます。
- ASN_INPUT_E base16 デコードがファイルに対して失敗した場合に返されます。
- BAD_FUNC_ARG ポインタが提供されていない場合に返されるエラーです。

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.52.2.303 function wolfSSL_CertManagerSetVerify

```
void wolfSSL_CertManagerSetVerify(
    WOLFSSL_CERT_MANAGER * cm,
    VerifyCallback vc
)
```

この関数は、証明書マネージャーの verifyCallback 関数を設定します。存在する場合、それはロードされた各 CERT に対して呼び出されます。検証エラーがある場合は、検証コールバックを使用してエラーを過度に乗り越えます。

Parameters:

- **cm** wolfssl_certmanagernew() を使用して作成された wolfssl_cert_manager 構造体へのポインタ。

See: wolfSSL_CertManagerVerify

Return: none 返品不可。

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.52.2.304 function wolfSSL_CertManagerCheckCRL

```
int wolfSSL_CertManagerCheckCRL(
    WOLFSSL_CERT_MANAGER * cm,
    unsigned char * der,
    int sz
)
```

CRL リスト。

Parameters:

- **cm** wolfssl_cert_manager 構造体へのポインタ。
- **der** DER フォーマット証明書へのポインタ。

See:

- CheckCertCRL
- ParseCertRelative
- wolfSSL_CertManagerSetCRL_CB
- InitDecodedCert

Return:

- SSL_SUCCESS 関数が予想どおりに返された場合は返します。wolfssl_cert_manager 構造体の CRLENABLED メンバーがオンになっている場合。
- MEMORY_E 割り当てられたメモリが失敗した場合は返します。
- BAD_FUNC_ARG wolfssl_cert_manager が null の場合

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.52.2.305 function wolfSSL_CertManagerEnableCRL

```
int wolfSSL_CertManagerEnableCRL(
    WOLFSSL_CERT_MANAGER * cm,
    int options
)
```

証明書マネージャを使用して証明書を検証するときに証明書失効リストの確認をオンにします。デフォルトでは、CRL チェックはオフです。オプションには、wolfssl_crl_checkall が含まれます。これは、チェーン内の各証明書に対して CRL 検査を実行します。これはデフォルトであるリーフ証明書のみです。

Parameters:

- **cm** wolfssl_certmanagernew() を使用して作成された wolfssl_cert_manager 構造体へのポインタ。

See: wolfSSL_CertManagerDisableCRL

Return:

- SSL_SUCCESS 成功した場合に返されます。、通話が戻ります。
- NOT_COMPILED_IN WolfSSL が CRL を有効にして構築されていない場合に返されます。
- MEMORY_E メモリ不足状態が発生した場合に返されます。
- BAD_FUNC_ARG ポインタが提供されていない場合に返されるエラーです。
- SSL_FAILURE CRL コンテキストを正しく初期化できない場合に返されます。

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.52.2.306 function wolfSSL_CertManagerDisableCRL

```
int wolfSSL_CertManagerDisableCRL(
    WOLFSSL_CERT_MANAGER *
)
```

証明書マネージャを使用して証明書を検証するときに証明書失効リストの確認をオフにします。デフォルトでは、CRL チェックはオフです。この関数を使用して、この Certificate Manager コンテキストを使用して CRL 検査を一時的または恒久的に無効にして、以前は CRL 検査が有効になっていました。

See: wolfSSL_CertManagerEnableCRL

Return:

- SSL_SUCCESS 成功した場合に返されます。、通話が戻ります。
- BAD_FUNC_ARG 関数ポインタが提供されていない場合に返されるエラーです。

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.52.2.307 function wolfSSL_CertManagerLoadCRL

```
int wolfSSL_CertManagerLoadCRL(
    WOLFSSL_CERT_MANAGER * cm,
    const char * path,
    int type,
    int monitor
)
```

証明書の失効確認のために証明書を CRL にロードする際にエラーチェックを行い、その後証明書を LoadCRL() へ渡します。

Parameters:

- **cm** wolfSSL_CertManagerNew()を使用して作成された WOLFSSL_CERT_MANAGER 構造体へのポインタ。
- **path** CRL へのパスを保持しているバッファーへのポインタ。
- **type** ロードする証明書の種類。

See:

- wolfSSL_CertManagerEnableCRL
- wolfSSL_LoadCRL

Return:

- SSL_SUCCESS wolfSSL_CertManagerLoadCRL でエラーが発生せず、loadCRL が成功で戻る場合に返されます。
- BAD_FUNC_ARG WOLFSSL_CERT_MANAGER 構造体が NULL の場合
- SSL_FATAL_ERROR wolfSSL_CertManagerEnableCRL が SSL_SUCCESS 以外のを返す場合。
- BAD_PATH_ERROR path が NULL の場合
- MEMORY_E LOADCRL がヒープメモリの割り当てに失敗した場合。

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.52.2.308 function wolfSSL_CertManagerLoadCRLBuffer

```
int wolfSSL_CertManagerLoadCRLBuffer(
    WOLFSSL_CERT_MANAGER * cm,
    const unsigned char * buff,
    long sz,
    int type
)
```

この関数は、BufferLoadCRL を呼び出すことによって CRL ファイルをロードします。

Parameters:

- **cm** wolfssl_cert_manager 構造体へのポインタ。
- **buff** 定数バイトタイプとバッファです。
- **sz** バッファのサイズを表す長い int。

See:

- BufferLoadCRL
- wolfSSL_CertManagerEnableCRL

Return:

- SSL_SUCCESS 関数がエラーなしで完了した場合に返されます。
- BAD_FUNC_ARG wolfssl_cert_manager が null の場合に返されます。
- SSL_FATAL_ERROR wolfssl_cert_manager に関連付けられているエラーがある場合に返されます。

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.52.2.309 function wolfSSL_CertManagerSetCRL_Cb

```
int wolfSSL_CertManagerSetCRL_Cb(
    WOLFSSL_CERT_MANAGER * cm,
    CbMissingCRL cb
)
```

この関数は CRL 証明書マネージャコールバックを設定します。LABE_CRL が定義されていて一致する CRL レコードが見つからない場合、CBMissingCRL は呼び出されます（WolfSSL_CertManagerSetCRL_CB を介して設定）。これにより、CRL を外部に検索してロードすることができます。

Parameters:

- **cm** 証明書の情報を保持している WOLFSSL_CERT_MANAGER 構造。

See:

- CbMissingCRL
- wolfSSL_SetCRL_Cb

Return:

- SSL_SUCCESS 関数とサブルーチンの実行が成功したら返されます。
- BAD_FUNC_ARG wolfssl_cert_manager 構造体が NULL の場合に返されます。

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.52.2.310 function wolfSSL_CertManagerFreeCRL

```
int wolfSSL_CertManagerFreeCRL(
    WOLFSSL_CERT_MANAGER * cm
)
```

この関数は証明書マネジャーに保持されている CRL を解放します。アプリケーションは CRL を wolfSSL_CertManagerFreeCRL を呼び出して解放した後に、新しい CRL をロードすることができます。

Parameters:

- **cm** wolfSSL_CertManagerNew()で生成された WOLFSSL_CERT_MANAGER 構造体へのポインター。

See: wolfSSL_CertManagerLoadCRL

Return:

- SSL_SUCCESS 関数の実行に成功した場合に返されます。
- BAD_FUNC_ARG WOLFSSL_CERT_MANAGER 構造体へのポインターが NULL で渡された場合に返されます。

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.52.2.311 function wolfSSL_CertManagerCheckOCSP

```
int wolfSSL_CertManagerCheckOCSP(
    WOLFSSL_CERT_MANAGER * cm,
    unsigned char * der,
    int sz
)
```

この機能により、OCSPENABLED OCSPENABLED が OCSP チェックオプションが有効になっていることを意味します。

Parameters:

- **cm** wolfssl_certmanagernew() を使用して作成された wolfssl_cert_manager 構造体へのポインタ。
- **der** 証明書へのバイトポインタ。

See:

- ParseCertRelative
- CheckCertOCSP

Return:

- SSL_SUCCESS 機能の実行に成功したことに戻ります。wolfssl_cert_manager の OCSPENABLED メンバーが有効になっています。
- BAD_FUNC_ARG wolfssl_cert_manager 構造体が null の場合、または許可されていない引数値がサブルーチンに渡された場合に返されます。
- MEMORY_E この関数内にメモリを割り当てるエラーまたはサブルーチンがある場合に返されます。

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.52.2.312 function wolfSSL_CertManagerEnableOCSP

```
int wolfSSL_CertManagerEnableOCSP(
    WOLFSSL_CERT_MANAGER * cm,
    int options
)
```

OCSP がオフになっている場合は OCSP をオンにし、[設定] オプションを使用可能になっている場合。

Parameters:

- **cm** wolfssl_certmanagernew() を使用して作成された wolfssl_cert_manager 構造体へのポインタ。

See: wolfSSL_CertManagerNew

Return:

- SSL_SUCCESS 関数呼び出しが成功した場合に返されます。
- BAD_FUNC_ARG cm 構造体が null の場合
- MEMORY_E wolfssl_ocsp struct 値が null の場合
- SSL_FAILURE WOLFSSL_OCSP 構造体の初期化は初期化に失敗します。
- NOT_COMPILED_IN 正しい機能を有効にしてコンパイルされていないビルド。

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.52.2.313 function wolfSSL_CertManagerDisableOCSP

```
int wolfSSL_CertManagerDisableOCSP(
    WOLFSSL_CERT_MANAGER *
)
```

OCSP 証明書の失効を無効にします。

See: wolfSSL_DisableCRL

Return:

- SSL_SUCCESS WolfSSL_CertMangerDisableCRL は、WolfSSL_CERT_MANAGER 構造体の CRLEnabled メンバを無効にしました。
- BAD_FUNC_ARG WOLFSSL 構造はヌルでした。

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.52.2.314 function wolfSSL_CertManagerSetOCSPOverrideURL

```
int wolfSSL_CertManagerSetOCSPOverrideURL(
    WOLFSSL_CERT_MANAGER * cm,
    const char * url
)
```

この関数は、URL を wolfssl_cert_manager 構造体の OCSpoverrideURL メンバーにコピーします。

See:

- ocspOverrideURL
- wolfSSL_SetOCSP_OverrideURL

Return:

- SSL_SUCCESS この機能は期待どおりに実行できました。
- BAD_FUNC_ARG wolfssl_cert_manager 構造体は null です。
- MEMEORY_E 証明書マネージャの OCSPoverRideURL メンバーにメモリを割り当てることができませんでした。

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.52.2.315 function wolfSSL_CertManagerSetOCSP_Cb

```
int wolfSSL_CertManagerSetOCSP_Cb(
    WOLFSSL_CERT_MANAGER * cm,
    CbOCSPIO ioCb,
    CbOCSPRespFree respFreeCb,
    void * ioCbCtx
)
```

この関数は、wolfssl_cert_manager の OCSP コールバックを設定します。

Parameters:

- **cm** wolfssl_cert_manager 構造体へのポインタ。
- **ioCb** CBocSpio 型の関数ポインタ。
- **respFreeCb** - CBOCSPRESPFREAS 型の関数ポインタ。

See:

- wolfSSL_CertManagerSetOCSPOverrideURL
- wolfSSL_CertManagerCheckOCSP
- wolfSSL_CertManagerEnableOCSPStapling
- wolfSSL_ENableOCSP
- wolfSSL_DisableOCSP
- wolfSSL_SetOCSP_Cb

Return:

- SSL_SUCCESS 実行に成功したことに戻ります。引数は wolfssl_cert_manager 構造体に保存されます。
- BAD_FUNC_ARG wolfssl_cert_manager が null の場合に返されます。

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.52.2.316 function wolfSSL_CertManagerEnableOCSPStapling

```
int wolfSSL_CertManagerEnableOCSPStapling(
    WOLFSSL_CERT_MANAGER * cm
)
```

この関数は、オプションをオンにしないと OCSP ステープルをオンにします。オプションを設定します。

See: wolfSSL_CTX_EnableOCSPStapling

Return:

- SSL_SUCCESS エラーがなく、関数が正常に実行された場合に返されます。
- BAD_FUNC_ARG wolfssl_cert_manager 構造体が NULL またはそうでない場合は、サブルーチンに渡された未解決の引数値があった場合に返されます。
- MEMORY_E メモリ割り当てがある問題が発生した場合に返されます。
- SSL_FAILURE OCSP 構造体の初期化が失敗した場合に返されます。
- NOT_COMPILED_IN wolfssl が haber_certificate_status_request オプションでコンパイルされていない場合に返されます。

Example

```
int wolfSSL_CTX_EnableOCSPStapling(WOLFSSL_CTX* ctx){
…
return wolfSSL_CertManagerEnableOCSPStapling(ctx->cm);
```

C.52.2.317 function wolfSSL_EnableCRL

```
int wolfSSL_EnableCRL(
    WOLFSSL * ssl,
    int options
)
```

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WolfSSL 構造へのポインタ。

See:

- wolfSSL_CertManagerEnableCRL
- InitCRL

Return:

- SSL_SUCCESS 関数とサブルーチンはエラーなしで返されました。
- BAD_FUNC_ARG WolfSSL 構造が NULL の場合に返されます。
- MEMORY_E メモリの割り当てが失敗した場合に返されます。
- SSL_FAILURE initcrl 関数が正常に戻されない場合に返されます。
- NOT_COMPILED_IN have_crl はコンパイル中に有効になっていませんでした。

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.52.2.318 function wolfSSL_DisableCRL

```
int wolfSSL_DisableCRL(
    WOLFSSL * ssl
)
```

See:

- wolfSSL_CertManagerDisableCRL
- wolfSSL_CertManagerDisableOCSP

Return:

- SSL_SUCCESS WolfSSL_CertMangerDisableCRL は、WolfSSL_CERT_MANAGER 構造体の CRLEnabled メンバを無効にしました。
- BAD_FUNC_ARG WOLFSSL 構造はヌルでした。

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
...
if(wolfSSL_DisableCRL(ssl) != SSL_SUCCESS){
    // Failure case
}
```

C.52.2.319 function wolfSSL_LoadCRL

```
int wolfSSL_LoadCRL(
    WOLFSSL * ssl,
    const char * path,
    int type,
    int monitor
)
```

失効検査の証明書

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WolfSSL 構造へのポインタ。
- **path** CRL ファイルへのパスを保持する定数文字ポインタ。
- **type** 証明書の種類を表す整数。

See:

- wolfSSL_CertManagerLoadCRL
- wolfSSL_CertManagerEnableCRL
- LoadCRL

Return:

- WOLFSSL_SUCCESS 関数とすべてのサブルーチンがエラーなしで実行された場合に返されます。
- SSL_FATAL_ERROR サブルーチンの 1 つが正常に戻されない場合に返されます。
- BAD_FUNC_ARG wolfssl_cert_manager または wolfssl 構造が null の場合

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.52.2.320 function wolfSSL_SetCRL_Cb

```
int wolfSSL_SetCRL_Cb(
    WOLFSSL * ssl,
    CbMissingCRL cb
)
```

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WolfSSL 構造へのポインタ。

See:

- CbMissingCRL
- wolfSSL_CertManagerSetCRL_Cb

Return:

- SSL_SUCCESS 関数またはサブルーチンがエラーなしで実行された場合に返されます。wolfssl_cert_manager の CBMissingCRL メンバーが設定されています。
- BAD_FUNC_ARG WOLFSSL または WOLFSSL_CERT_MANAGER 構造体が NULL の場合に返されます。

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.52.2.321 function wolfSSL_EnableOCSP

```
int wolfSSL_EnableOCSP(
    WOLFSSL * ssl,
    int options
)
```

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WolfSSL 構造へのポインタ。

See: wolfSSL_CertManagerEnableOCSP

Return:

- SSL_SUCCESS 関数とサブルーチンがエラーなしで実行された場合に返されます。
- BAD_FUNC_ARG この関数またはサブルーチンの引数が無効な引数値を受信した場合に返されます。
- MEMORY_E 構造体やその他の変数にメモリを割り当てるエラーが発生した場合に返されます。
- NOT_COMPILED_IN wolfssl が hane_ocsp オプションでコンパイルされていない場合に返されます。

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.52.2.322 function wolfSSL_DisableOCSP

```
int wolfSSL_DisableOCSP(
    WOLFSSL *
)
```

See: wolfSSL_CertManagerDisableOCSP

Return:

- SSL_SUCCESS 関数とそのサブルーチンがエラーなしで戻った場合に返されます。wolfssl_cert_manager 構造体の OCSPENABLED メンバーは正常に設定されました。
- BAD_FUNC_ARG WolfSSL 構造が NULL の場合に返されます。

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
…
if(wolfSSL_DisableOCSP(ssl) != SSL_SUCCESS){
    // Returned with an error. Failure case in this block.
}
```

C.52.2.323 function wolfSSL_SetOCSP_OverrideURL

```
int wolfSSL_SetOCSP_OverrideURL(
    WOLFSSL * ssl,
    const char * url
)
```

wolfssl_cert_manager 構造体。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WolfSSL 構造へのポインタ。

See: wolfSSL_CertManagerSetOCSPOverrideURL

Return:

- SSL_SUCCESS 機能の実行に成功したことに戻ります。
- BAD_FUNC_ARG wolfssl 構造体が null の場合、または未解決の引数がサブルーチンに渡された場合に返されます。
- MEMORY_E サブルーチンにメモリを割り当てるエラーが発生した場合に返されます。

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.52.2.324 function wolfSSL_SetOCSP_Cb

```
int wolfSSL_SetOCSP_Cb(
    WOLFSSL * ssl,
    CbOCSPIO ioCb,
    CbOCSPRespFree respFreeCb,
    void * ioCbCtx
)
```

wolfssl_cert_manager 構造体。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WolfSSL 構造へのポインタ。
- **ioCb** CBocSpio を入力するための関数ポインタ。
- **respFreeCb** 応答メモリを解放するための呼び出しである CBocSpreSpFree を入力するための関数ポインタ。

See:

- wolfSSL_CertManagerSetOCSP_Cb
- CbOCSPIO
- CbOCSPRespFree

Return:

- SSL_SUCCESS 関数がエラーなしで実行された場合に返されます。CM の OCSPIOCB、OCSPRESPFREECB、および OCSPIOCTX メンバーが設定されています。
- BAD_FUNC_ARG WOLFSSL または WOLFSSL_CERT_MANAGER 構造が NULL の場合に返されます。

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.52.2.325 function wolfSSL_CTX_EnableCRL

```
int wolfSSL_CTX_EnableCRL(
    WOLFSSL_CTX * ctx,
```

```
    int options
)
```

See:

- wolfSSL_CertManagerEnableCRL
- InitCRL
- wolfSSL_CTX_DisableCRL

Return:

- SSL_SUCCESS この関数とそれがサブルーチンの場合はエラーなしで実行されます。
- BAD_FUNC_ARG CTX 構造体が NULL の場合、またはその他の点ではサブルーチンに無効な引数があった場合に返されます。
- MEMORY_E 関数の実行中にメモリの割り当てエラーが発生した場合に返されます。
- SSL_FAILURE wolfssl_cert_manager の CRL メンバーが正しく初期化されなかった場合に返されます。
- NOT_COMPILED_IN wolfssl は hane_crl オプションでコンパイルされませんでした。

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.52.2.326 function wolfSSL_CTX_DisableCRL

```
int wolfSSL_CTX_DisableCRL(
    WOLFSSL_CTX * ctx
)
```

See: wolfSSL_CertManagerDisableCRL

Return:

- SSL_SUCCESS 関数がエラーなしで実行された場合に返されます。wolfssl_cert_manager 構造体の CRLEnabled メンバーは 0 に設定されています。
- BAD_FUNC_ARG CTX 構造体または CM 構造体に NULL 値がある場合に返されます。

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
...
if(wolfSSL_CTX_DisableCRL(ssl->ctx) != SSL_SUCCESS){
    // Failure case.
}
```

C.52.2.327 function wolfSSL_CTX_LoadCRL

```
int wolfSSL_CTX_LoadCRL(
    WOLFSSL_CTX * ctx,
    const char * path,
    int type,
    int monitor
)
```

wolfssl_certmanagerLoadcr()。

Parameters:

- **ctx** wolfSSL_CTX_new()を使用して作成された WOLFSSL_CTX 構造体へのポインタ。
- **path** 証明書へのパス。
- **type** 証明書の種類を保持する整数変数。

See:

- wolfSSL_CertManagerLoadCRL
- LoadCRL

Return:

- SSL_SUCCESS - 関数とそのサブルーチンがエラーなしで実行された場合に返されます。
- BAD_FUNC_ARG - この関数またはサブルーチンが NULL 構造に渡された場合に返されます。
- BAD_PATH_ERROR - パス変数が null として開くと戻ります。
- MEMORY_E - メモリの割り当てが失敗した場合に返されます。

Example

```
WOLFSSL_CTX* ctx;
const char* path;
…
return wolfSSL_CTX_LoadCRL(ctx, path, SSL_FILETYPE_PEM, 0);
```

C.52.2.328 function wolfSSL_CTX_SetCRL_Cb

```
int wolfSSL_CTX_SetCRL_Cb(
    WOLFSSL_CTX * ctx,
    CbMissingCRL cb
)
```

wolfssl_certmanagersetCRL_CB を呼び出して、WolfSSL_CERT_MANAGER 構造のメンバー。

Parameters:

- **ctx** wolfSSL_CTX_new()で作成された WOLFSSL_CTX 構造体へのポインタ。

See:

- wolfSSL_CertManagerSetCRL_Cb
- CbMissingCRL

Return:

- SSL_SUCCESS 実行が成功するために返されました。WOLFSSL_CERT_MANAGER 構造体の CBMSSINGCRL は CB に正常に設定されました。
- BAD_FUNC_ARG wolfssl_ctx または wolfssl_cert_manager が NULL の場合に返されます。

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.52.2.329 function wolfSSL_CTX_EnableOCSP

```
int wolfSSL_CTX_EnableOCSP(
    WOLFSSL_CTX * ctx,
    int options
)
```

wolfssl の機能オプションの値が 1 つ以上のオプションで構成されている場合は、次のオプションを 1 つ以上にします.wolfssl_ocsp_enable - OCSP ルックアップを有効にする wolfssl_ocsp_url_override - 証明書の URL の代わりに URL をオーバーライドします。オーバーライド URL は、wolfssl_ctx_setocsp_overrideURL() 関数を使用して指定されます。この関数は、wolfssl が OCSP サポート（–enable-ocsp、#define hane_ocsp）でコンパイルされたときにのみ OCSP オプションを設定します。

Parameters:

- **ctx** wolfSSL_CTX_new()で作成された SSL コンテキストへのポインタ。

See: wolfSSL_CTX_OCSP_set_override_url

Return:

- SSL_SUCCESS 成功したときに返されます。
- SSL_FAILURE 失敗したときに返されます。
- NOT_COMPILED_IN この関数が呼び出されたときに返されますが、wolfssl がコンパイルされたときに OCSP サポートは有効になっていませんでした。

Example

```
WOLFSSL_CTX* ctx = 0;
...
wolfSSL_CTX_OCSP_set_options(ctx, WOLFSSL_OCSP_ENABLE);
```

C.52.2.330 function wolfSSL_CTX_DisableOCSP

```
int wolfSSL_CTX_DisableOCSP(
    WOLFSSL_CTX *
)
```

wolfssl_cert_manager 構造体の OCSPENABLED メンバーに影響を与えます。

See:

- wolfSSL_DisableOCSP
- wolfSSL_CertManagerDisableOCSP

Return:

- SSL_SUCCESS 関数がエラーなしで実行された場合に返されます。CM の OCSPENABLED メンバーは無効になっています。
- BAD_FUNC_ARG WOLFSSL_CTX 構造が null の場合に返されます。

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( method );
WOLFSSL* ssl = wolfSSL_new(ctx);
...
if(!wolfSSL_CTX_DisableOCSP(ssl->ctx)){
    // OCSP is not disabled
}
```

C.52.2.331 function wolfSSL_CTX_SetOCSP_OverrideURL

```
int wolfSSL_CTX_SetOCSP_OverrideURL(
    WOLFSSL_CTX * ctx,
    const char * url
)
```

wolfssl_csp_url_override オプションが wolfssl_ctx_enableocsp を使用して設定されていない限り、OCSP は個々の証明書にある URL を使用します。

Parameters:

- **ctx** wolfSSL_CTX_new()で作成された SSL コンテキストへのポインタ。

See: wolfSSL_CTX_OCSP_set_options

Return:

- SSL_SUCCESS 成功したときに返されます。
- SSL_FAILURE 失敗したときに返されます。
- NOT_COMPILED_IN この関数が呼び出されたときに返されますが、wolfssl がコンパイルされたときに OCSP サポートは有効になっていませんでした。

Example

```
WOLFSSL_CTX* ctx = 0;
...
wolfSSL_CTX_OCSP_set_override_url(ctx, “custom-url-here”);
```

C.52.2.332 function wolfSSL_CTX_SetOCSP_Cb

```
int wolfSSL_CTX_SetOCSP_Cb(
    WOLFSSL_CTX * ctx,
    CbOCSPIO ioCb,
    CbOCSPRespFree respFreeCb,
    void * ioCbCtx
)
```

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WolfSSL 構造へのポインタ。
- **ioCb** 関数ポインタである CBocSpio 型。
- **respFreeCb** 関数ポインタである CBocSprepSprepFree 型。

See:

- wolfSSL_CertManagerSetOCSP_Cb
- CbOCSPIO
- CbOCSPRespFree

Return:

- SSL_SUCCESS 関数が正常に実行された場合に返されます。CM 内の OCSPIOCB、OCSPRESPFREECB、および OCSPIOCTX メンバーは正常に設定されました。
- BAD_FUNC_ARG WOLFSSL_CTX または wolfssl_cert_manager 構造体が null の場合に返されます。

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.52.2.333 function wolfSSL_CTX_EnableOCSPStapling

```
int wolfSSL_CTX_EnableOCSPStapling(
    WOLFSSL_CTX *
)
```

wolfssl_certmanagerEnableOcspStapling()。

See:

- wolfSSL_CertManagerEnableOCSPStapling
- InitOCSP

Return:

- SSL_SUCCESS エラーがなく、関数が正常に実行された場合に返されます。
- BAD_FUNC_ARG WOLFSSL_CTX 構造体が NULL またはそうでない場合は、サブルーチンに渡された未解決の引数値があった場合に返されます。
- MEMORY_E メモリ割り当てがある問題が発生した場合に返されます。
- SSL_FAILURE OCSP 構造体の初期化が失敗した場合に返されます。
- NOT_COMPILED_IN wolfssl が haber_certificate_status_request オプションでコンパイルされていない場合に返されます。

Example

```
WOLFSSL* ssl = wolfSSL_new();
ssl->method.version; // set to desired protocol
...
if(!wolfSSL_CTX_EnableOCSPStapling(ssl->ctx)){
    // OCSP stapling is not enabled
}
```

C.52.2.334 function wolfSSL_KeepArrays

```
void wolfSSL_KeepArrays(
    WOLFSSL *
)
```

通常、SSL ハンドシェイクの最後に、WolfSSL は一時的なアレイを解放します。ハンドシェイクが始まる前にこの関数を呼び出すと、WolfSSL は一時的な配列を解放するのを防ぎます。Wolfssl_get_keys() または PSK のヒントなどのものには、一時的な配列が必要になる場合があります。ユーザが一時的な配列で行われると、wolfssl_freearray() のいずれかが即座にリソースを解放することができ、あるいは、関連する SSL オブジェクトが解放されたときにリソースが解放されるようになる可能性がある。

See: wolfSSL_FreeArrays

Return: none 返品不可。

Example

```
WOLFSSL* ssl;
...
wolfSSL_KeepArrays(ssl);
```

C.52.2.335 function wolfSSL_FreeArrays

```
void wolfSSL_FreeArrays(
    WOLFSSL *
)
```

通常、SSL ハンドシェイクの最後に、WolfSSL は一時的なアレイを解放します。wolfssl_keeparrays() がハンドシェイクの前に呼び出された場合、WolfSSL は一時的な配列を解放しません。この関数は一時的な配列を明示的に解放し、ユーザーが一時的な配列で行われたときに呼び出されるべきであり、SSL オブジェクトがこれらのリソースを解放するのを待ったくない。

See: wolfSSL_KeepArrays

Return: none 返品不可。

Example

```
WOLFSSL* ssl;
...
wolfSSL_FreeArrays(ssl);
```

C.52.2.336 function wolfSSL_UseSNI

```
int wolfSSL_UseSNI(
    WOLFSSL * ssl,
    unsigned char type,
    const void * data,
    unsigned short size
)
```

'ssl' パラメータに渡されたオブジェクト。これは、WolfSSL クライアントによって SNI 拡張機能が ClientHello で送信され、WolfSSL Server は ServerHello + SNI または SNI ミスマッチの場合は致命的な Alert Hello + SNI を応答します。

Parameters:

- **ssl** wolfSSL_new()で作成された SSL オブジェクトへのポインタ。
- **type** どの種類のサーバー名がデータに渡されたかを示します。既知の型は次のとおりです。enum {wolfssl_sni_host_name = 0};
- **data** サーバー名データへのポインタ。

See:

- wolfSSL_new
- wolfSSL_CTX_UseSNI

Return:

- WOLFSSL_SUCCESS 成功時に返されます。
- BAD_FUNC_ARG 次のいずれかの場合で返されるエラーです.SSL は NULL、データは NULL、タイプは不明な値です。(下記参照)
- MEMORY_E 十分なメモリがないときにエラーが返されます。

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.52.2.337 function wolfSSL_CTX_UseSNI

```
int wolfSSL_CTX_UseSNI(
    WOLFSSL_CTX * ctx,
    unsigned char type,
    const void * data,
    unsigned short size
)
```

SSL コンテキストから作成されたオブジェクトは'ctx' パラメータに渡されました。これは、WolfSSL クライアントによって SNI 拡張機能が ClientHello で送信され、WolfSSL サーバーは ServerHello + SNI または SNI の不一致の場合には致命的な ALERT Hello + SNI を応答します。

Parameters:

- **ctx** wolfSSL_CTX_new()で作成された SSL コンテキストへのポインタ。
- **type** どの種類のサーバー名がデータに渡されたかを示します。既知の型は次のとおりです。enum {wolfssl_sni_host_name = 0};
- **data** サーバー名データへのポインタ。

See:

- wolfSSL_CTX_new
- wolfSSL_UseSNI

Return:

- WOLFSSL_SUCCESS 成功時に返されます。
- BAD_FUNC_ARG 次のいずれかの場合で返されるエラーです.CTX は NULL、データは NULL、タイプは不明な値です。(下記参照)
- MEMORY_E 十分なメモリがないときにエラーが返されます。

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.52.2.338 function wolfSSL_SNI_SetOptions

```
void wolfSSL_SNI_SetOptions(
    WOLFSSL * ssl,
    unsigned char type,
    unsigned char options
)
```

'ssl' パラメータに渡された SSL オブジェクト内のサーバー名表示を使用した SSL セッションの動作。オプションを以下に説明します。

Parameters:

- **ssl** wolfSSL_new()で作成された SSL オブジェクトへのポインタ。
- **type** どの種類のサーバー名がデータに渡されたかを示します。既知の型は次のとおりです。enum {wolfssl_sni_host_name = 0};
- **options** 選択されたオプションを持つビット単位のセマフォ。利用可能なオプションは次のとおりです。enum {wolfssl_sni_continue_on_mismatch = 0x01、wolfssl_sni_answer_on_mismatch = 0x02}; 通常、サーバーは、クライアントによって提供されたホスト名がサーバーと表示されているホスト名がサーバーで提供されている場合、サーバーは handshake を中止します。
- **WOLFSSL_SNI_CONTINUE_ON_MISMATCH** このオプションを設定すると、サーバーはセッションを中止する代わりに SNI 応答を送信しません。

See:

- wolfSSL_new
- wolfSSL_UseSNI
- wolfSSL_CTX_SNI_SetOptions

Return: none いいえ返します。

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.52.2.339 function wolfSSL_CTX_SNI_SetOptions

```
void wolfSSL_CTX_SNI_SetOptions(
    WOLFSSL_CTX * ctx,
```

```
    unsigned char type,
    unsigned char options
)
```

SSL セッションを使用した SSL オブジェクトのサーバ名指示を使用して、SSL コンテキストから作成された SSL オブジェクトから作成されます。オプションを以下に説明します。

Parameters:

- **ctx** wolfSSL_CTX_new()で作成された SSL コンテキストへのポインタ。
- **type** どの種類のサーバー名がデータに渡されたかを示します。既知の型は次のとおりです。enum {wolfssl_sni_host_name = 0};
- **options** 選択されたオプションを持つビット単位のセマフォ。利用可能なオプションは次のとおりです。enum {wolfssl_sni_continue_on_mismatch = 0x01、wolfssl_sni_answer_on_mismatch = 0x02}; 通常、サーバーは、クライアントによって提供されたホスト名がサーバーと表示されているホスト名がサーバーで提供されている場合、サーバーは handshake を中止します。
- **WOLFSSL_SNI_CONTINUE_ON_MISMATCH** このオプションを設定すると、サーバーはセッションを中止する代わりに SNI 応答を送信しません。

See:

- wolfSSL_CTX_new
- wolfSSL_CTX_UseSNI
- wolfSSL_SNI_SetOptions

Return: none いいえ返します。

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.52.2.340 function wolfSSL_SNI_GetFromBuffer

```
int wolfSSL_SNI_GetFromBuffer(
    const unsigned char * clientHello,
    unsigned int helloSz,
    unsigned char type,
    unsigned char * sni,
    unsigned int * inOutSz
)
```

クライアントによってクライアントから提供された名前表示クライアントによって送信されたメッセージセッションを開始する。SNI を取得するためのコンテキストまたはセッション設定が必要ありません。

Parameters:

- **buffer** クライアントから提供されたデータへのポインタ（クライアント hello）。
- **bufferSz** クライアント hello メッセージのサイズ。

- **type** どの種類のサーバー名がバッファーから取得されているかを示します。既知の型は次のとおりです。enum {wolfssl_sni_host_name = 0};
- **sni** 出力が保存される場所へのポインタ。

See:

- wolfSSL_UseSNI
- wolfSSL_CTX_UseSNI
- wolfSSL_SNI_GetRequest

Return:

- WOLFSSL_SUCCESS 成功時に返されます。
- BAD_FUNC_ARG このケースで返されるエラーは、次のいずれかの場合で返されます。バッファは NULL、BUFFERSZ <= 0、SNI は NULL、INOUTSZ は NULL または <= 0 です。
- BUFFER_ERROR 不正なクライアント hello メッセージがあるときにエラーが返されます。
- INCOMPLETE_DATA 抽出を完了するのに十分なデータがない場合に返されるエラーです。

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.52.2.341 function wolfSSL_SNI_Status

```
unsigned char wolfSSL_SNI_Status(
    WOLFSSL * ssl,
    unsigned char type
)
```

この関数は SNI オブジェクトのステータスを取得します。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WolfSSL 構造へのポインタ。

See:

- TLSX_SNI_Status
- TLSX_SNI_find
- TLSX_Find

Return:

- value SNI が NULL でない場合、この関数は SNI 構造体のステータスメンバーのバイト値を返します。
- 0 SNI オブジェクトが NULL の場合

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.52.2.342 function wolfSSL_SNI_GetRequest

```
unsigned short wolfSSL_SNI_GetRequest(
    WOLFSSL * ssl,
    unsigned char type,
    void ** data
)
```

SSL セッションでクライアントによって提供されるサーバー名の表示。

Parameters:

- **ssl** wolfSSL_new()で作成された SSL オブジェクトへのポインタ。
- **type** どの種類のサーバー名がデータ内で取得されているかを示します。既知の型は次のとおりです。 enum {wolfssl_sni_host_name = 0};

See:

- wolfSSL_UseSNI
- wolfSSL_CTX_UseSNI

Return: size 提供された SNI データのサイズ。

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.52.2.343 function wolfSSL_UseALPN

```
int wolfSSL_UseALPN(
    WOLFSSL * ssl,
    char * protocol_name_list,
    unsigned int protocol_name_listSz,
    unsigned char options
)
```

wolfssl セッションに ALPN を設定します。

Parameters:

- **ssl** 使用する WolfSSL セッション。
- **protocol_name_list** 使用するプロトコル名のリスト。カンマ区切り文字列が必要です。
- **protocol_name_listSz** プロトコル名のリストのサイズ。

See: TLSX_UseALPN

Return:

- WOLFSSL_SUCCESS: 成功時に返されます。
- BAD_FUNC_ARG SSL または PROTOCOL_NAME_LIST が NULL または PROTOCOL_NAME_LISTSZ が大きすぎたり、オプションがサポートされていないものを含みます。
- MEMORY_ERROR プロトコルリストのメモリの割り当て中にエラーが発生しました。
- SSL_FAILURE 失敗時に返されます。

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.52.2.344 function wolfSSL_ALPN_GetProtocol

```
int wolfSSL_ALPN_GetProtocol(
    WOLFSSL * ssl,
    char ** protocol_name,
    unsigned short * size
)
```

この関数は、サーバーによって設定されたプロトコル名を取得します。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WolfSSL 構造へのポインタ。
- **protocol_name** プロトコル名を表す CHAR へのポインタは、ALPN 構造に保持されます。

See:

- TLSX_ALPN_GetRequest
- TLSX_Find

Return:

- SSL_SUCCESS エラーが投げられていない正常な実行に戻りました。
- SSL_FATAL_ERROR 拡張子が見つからなかった場合、またはピアとプロトコルが一致しなかった場合に返されます。2 つ以上のプロトコル名が受け入れられている場合は、スローされたエラーもあります。
- SSL_ALPN_NOT_FOUND ピアとプロトコルの一致が見つからなかったことを示す返されました。
- BAD_FUNC_ARG 関数に渡された null 引数があった場合に返されます。

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.52.2.345 function wolfSSL_ALPN_GetPeerProtocol

```
int wolfSSL_ALPN_GetPeerProtocol(
    WOLFSSL * ssl,
    char ** list,
    unsigned short * listSz
)
```

この関数は、alpn_client_list データを SSL オブジェクトからバッファにコピーします。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WolfSSL 構造へのポインタ。
- **list** バッファへのポインタ。SSL オブジェクトからのデータがコピーされます。

See: wolfSSL_UseALPN

Return:

- SSL_SUCCESS 関数がエラーなしで実行された場合に返されます。SSL オブジェクトの ALPN_CLIENT_LIST メンバーが LIST パラメータにコピーされました。
- BAD_FUNC_ARG list または listsz パラメーターが null の場合に返されます。
- BUFFER_ERROR リストバッファに問題がある場合は（NULL またはサイズが 0 の場合）に問題がある場合に返されます。
- MEMORY_ERROR メモリを動的に割り当てる問題がある場合に返されます。

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.52.2.346 function wolfSSL_UseMaxFragment

```
int wolfSSL_UseMaxFragment(
    WOLFSSL * ssl,
    unsigned char mfl
)
```

'ssl' パラメータに渡された SSL オブジェクト内の最大フラグメント長。これは、最大フラグメント長拡張機能が WolfSSL クライアントによって ClientHello で送信されることを意味します。

Parameters:

- **ssl** wolfSSL_new()で作成された SSL オブジェクトへのポインタ。

See:

- wolfSSL_new
- wolfSSL_CTX_UseMaxFragment

Return:

- SSL_SUCCESS 成功時に返されます。
- BAD_FUNC_ARG 次のいずれかの場合に返されるエラーです.SSL は NULL、MFL は範囲外です。
- MEMORY_E 十分なメモリがないときにエラーが返されます。

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.52.2.347 function wolfSSL_CTX_UseMaxFragment

```
int wolfSSL_CTX_UseMaxFragment(
    WOLFSSL_CTX * ctx,
    unsigned char mfl
)
```

SSL コンテキストから作成された SSL オブジェクトの最大フラグメント長さ'ctx' パラメータに渡されました。これは、最大フラグメント長拡張機能が WolfSSL クライアントによって ClientHello で送信されることを意味します。

Parameters:

- **ctx** wolfSSL_CTX_new()で作成された SSL コンテキストへのポインタ。

See:

- wolfSSL_CTX_new

- wolfSSL_UseMaxFragment

Return:

- SSL_SUCCESS 成功時に返されます。
- BAD_FUNC_ARG 次のいずれかの場合に返されるエラーです.CTX は NULL、MFL は範囲外です。
- MEMORY_E 十分なメモリがないときにエラーが返されます。

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.52.2.348 function wolfSSL_UseTruncatedHMAC

```
int wolfSSL_UseTruncatedHMAC(
    WOLFSSL * ssl
)
```

'ssl' パラメータに渡された SSL オブジェクト内の truncated HMAC。これは、切り捨てられた HMAC 拡張機能が WolfSSL クライアントによって ClientHello で送信されることを意味します。

See:

- wolfSSL_new
- wolfSSL_CTX_UseMaxFragment

Return:

- SSL_SUCCESS 成功時に返されます。
- BAD_FUNC_ARG 次のいずれかの場合に返されるエラーです.SSL は NULL です
- MEMORY_E 十分なメモリがないときにエラーが返されます。

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.52.2.349 function wolfSSL_CTX_UseTruncatedHMAC

```
int wolfSSL_CTX_UseTruncatedHMAC(
    WOLFSSL_CTX * ctx
)
```

'ctx' パラメータに渡された SSL コンテキストから作成された SSL オブジェクトのための Truncated HMAC。これは、切り捨てられた HMAC 拡張機能が WolfSSL クライアントによって ClientHello で送信されることを意味します。

See:

- wolfSSL_CTX_new
- wolfSSL_UseMaxFragment

Return:

- SSL_SUCCESS 成功時に返されます。
- BAD_FUNC_ARG 次のいずれかの場合に返されるエラーです.CTX は NULL
- MEMORY_E 十分なメモリがないときにエラーが返されます。

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.52.2.350 function wolfSSL_UseOCSPStapling

```
int wolfSSL_UseOCSPStapling(
    WOLFSSL * ssl,
    unsigned char status_type,
    unsigned char options
)
```

OCSP で提示された証明書失効チェックのコストを下げます。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WolfSSL 構造へのポインタ。
- **status_type** tlsx_usecertificateSrequest() に渡され、CertificateStatusRequest 構造体に格納されているバイトタイプ。

See:

- TLSX_UseCertificateStatusRequest
- wolfSSL_CTX_UseOCSPStapling

Return:

- SSL_SUCCESS tlsx_usecertificateStatusRequest がエラーなしで実行された場合に返されます。
- MEMORY_E メモリの割り当てにエラーがある場合に返されます。
- BAD_FUNC_ARG NULL またはその他の点では、関数に渡された値が渡される引数がある場合に返されます。

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
…
if (wolfSSL_UseOCSPStapling(ssl, WOLFSSL_CSR2_OCSP,
WOLFSSL_CSR2_OCSP_USE_NONCE) != SSL_SUCCESS){
    // Failed case.
}
```

C.52.2.351 function wolfSSL_CTX_UseOCSPStapling

```
int wolfSSL_CTX_UseOCSPStapling(
    WOLFSSL_CTX * ctx,
    unsigned char status_type,
    unsigned char options
)
```

Parameters:

- **ctx** wolfSSL_CTX_new()を使用して作成された WOLFSSL_CTX 構造体へのポインタ。
- **status_type** tlsx_usecertificateSrequest() に渡され、CertificateStatusRequest 構造体に格納されているバイトタイプ。

See:

- wolfSSL_UseOCSPStaplingV2
- wolfSSL_UseOCSPStapling
- TLSX_UseCertificateStatusRequest

Return:

- SSL_SUCCESS 関数とサブルーチンがエラーなしで実行された場合に返されます。
- BAD_FUNC_ARG 未解決の値がサブルーチンに渡された場合、WOLFSSL_CTX 構造体が NULL またはそうでない場合に返されます。
- MEMORY_E 関数またはサブルーチンがメモリを正しく割り振ることができなかった場合に返されます。

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.52.2.352 function wolfSSL_UseOCSPStaplingV2

```
int wolfSSL_UseOCSPStaplingV2(
    WOLFSSL * ssl,
    unsigned char status_type,
    unsigned char options
)
```

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WolfSSL 構造へのポインタ。
- **status_type** OCSP ステータスタイプをロードするバイトタイプ。

See:

- TLSX_UseCertificatStatusRequestV2
- wolfSSL_SNI_SetOptions
- wolfSSL_CTX_SNI_SetOptions

Return:

- SSL_SUCCESS - 関数とサブルーチンがエラーなしで実行された場合に返されます。
- MEMORY_E - メモリエラーの割り当てがあった場合に返されます。
- BAD_FUNC_ARG - NULL またはそれ以外の場合は解読されていない引数が関数またはサブルーチンに渡された場合に返されます。

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.52.2.353 function wolfSSL_CTX_UseOCSPStaplingV2

```
int wolfSSL_CTX_UseOCSPStaplingV2(
    WOLFSSL_CTX * ctx,
    unsigned char status_type,
    unsigned char options
)
```

OCSP ステイプルのために。

Parameters:

- **ctx** wolfSSL_CTX_new()を使用して作成された WOLFSSL_CTX 構造体へのポインタ。
- **status_type** CertificatStatusRequest 構造体にあるバイトタイプで、wolfssl_csr2_ocsp または wolfssl_csr2_ocsp_multi でなければなりません。

See:

- TLSX_UseCertificateStatusRequestV2
- wc_RNG_GenerateBlock
- TLSX_Push

Return:

- SSL_SUCCESS 関数とサブルーチンがエラーなしで実行された場合。
- BAD_FUNC_ARG WOLFSSL_CTX 構造が null の場合、または側数変数がクライアント側ではない場合に返されます。
- MEMORY_E メモリの割り当てが失敗した場合に返されます。

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.52.2.354 function wolfSSL_UseSupportedCurve

```
int wolfSSL_UseSupportedCurve(
    WOLFSSL * ssl,
    word16 name
)
```

サポートされている楕円曲線拡張子は、'SSL' パラメータに渡された SSL オブジェクトでサポートされています。これは、サポートされているカーブが WolfSSL クライアントによって ClientHello で送信されることを意味します。この機能は複数の曲線を有効にするために複数の時間と呼ぶことができます。

Parameters:

- **ssl** wolfSSL_new()で作成された SSL オブジェクトへのポインタ。

See:

- wolfSSL_CTX_new
- wolfSSL_CTX_UseSupportedCurve

Return:

- SSL_SUCCESS 成功時に返されます。
- BAD_FUNC_ARG 次のいずれかの場合に返されるエラーです.SSL は NULL です。名前は未知の値です。(下記参照)
- MEMORY_E 十分なメモリがないときにエラーが返されます。

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.52.2.355 function wolfSSL_CTX_UseSupportedCurve

```
int wolfSSL_CTX_UseSupportedCurve(
    WOLFSSL_CTX * ctx,
    word16 name
)
```

サポートされている楕円曲線は、'ctx' パラメータに渡された SSL コンテキストから作成された SSL オブジェクトの拡張子です。これは、サポートされているカーブが WolfSSL クライアントによって ClientHello で送信されることを意味します。この機能は複数の曲線を有効にするために複数の時間と呼ぶことができます。

Parameters:

- **ctx** wolfSSL_CTX_new()で作成された SSL コンテキストへのポインタ。

See:

- wolfSSL_CTX_new
- wolfSSL_UseSupportedCurve

Return:

- SSL_SUCCESS 成功時に返されます。
- BAD_FUNC_ARG 次のいずれかの場合に返されるエラーです.CTX は NULL、名前は未知の値です。(下記参照)
- MEMORY_E 十分なメモリがないときにエラーが返されます。

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.52.2.356 function wolfSSL_UseSecureRenegotiation

```
int wolfSSL_UseSecureRenegotiation(
    WOLFSSL * ssl
)
```

この関数は、供給された WOLFSSL 構造の安全な再交渉を強制します。これはお勧めできません。

See:

- TLSX_Find
- TLSX_UseSecureRenegotiation

Return:

- SSL_SUCCESS 安全な再ネゴシエーションを正常に設定します。
- BAD_FUNC_ARG ssl が NULL の場合、エラーを返します。
- MEMORY_E 安全な再交渉のためにメモリを割り当てることができない場合、エラーを返します。

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.52.2.357 function wolfSSL_Rehandshake

```
int wolfSSL_Rehandshake(
    WOLFSSL * ssl
)
```

この関数は安全な再交渉ハンドシェイクを実行します。これは、WolfSSL がこの機能を妨げるように強制されます。

See:

- wolfSSL_negotiate
- wc_InitSha512
- wc_InitSha384
- wc_InitSha256
- wc_InitSha
- wc_InitMd5

Return:

- SSL_SUCCESS 関数がエラーなしで実行された場合に返されます。
- BAD_FUNC_ARG wolfssl 構造が null またはそうでなければ、許容できない引数がサブルーチンに渡された場合に返されます。
- SECURE_RENEGOTIATION_E ハンドシェイクを再ネゴシエーションすることにエラーが発生した場合に返されます。
- SSL_FATAL_ERROR サーバーまたはクライアント構成にエラーが発生した場合は、再ネゴシエーションが完了できなかった場合に返されます。wolfssl_negotiate() を参照してください。

Example

```
WOLFSSL* ssl = wolfSSL_new(ctx);
...
if(wolfSSL_Rehandshake(ssl) != SSL_SUCCESS){
    // There was an error and the rehandshake is not successful.
}
```

C.52.2.358 function wolfSSL_UseSessionTicket

```
int wolfSSL_UseSessionTicket(
    WOLFSSL * ssl
)
```

セッションチケットを使用するように WolfSSL 構造を強制します。定数 hous_session_ticket を定義し、定数 NO_WOLFSSL_CLIENT をこの関数を使用するように定義しないでください。

See: TLSX_UseSessionTicket

Return:

- SSL_SUCCESS セッションチケットを使用したセットに成功しました。
- BAD_FUNC_ARG ssl が NULL の場合に返されます。
- MEMORY_E セッションチケットを設定するためのメモリの割り当て中にエラーが発生しました。

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.52.2.359 function wolfSSL_CTX_UseSessionTicket

```
int wolfSSL_CTX_UseSessionTicket(
    WOLFSSL_CTX * ctx
)
```

この関数は、セッションチケットを使用するように WolfSSL コンテキストを設定します。

See: TLSX_UseSessionTicket

Return:

- SSL_SUCCESS 関数は正常に実行されます。
- BAD_FUNC_ARG CTX が NULL の場合に返されます。
- MEMORY_E 内部関数内のメモリの割り当て中にエラーが発生しました。

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.52.2.360 function wolfSSL_get_SessionTicket

```
int wolfSSL_get_SessionTicket(
    WOLFSSL * ssl,
    unsigned char * buf,
    word32 * bufSz
)
```

この機能は、セッション構造のチケットメンバーをバッファにコピーします。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WolfSSL 構造へのポインタ。
- **buf** メモリバッファを表すバイトポインタ。

See:

- wolfSSL_UseSessionTicket
- wolfSSL_set_SessionTicket

Return:

- SSL_SUCCESS 関数がエラーなしで実行された場合に返されます。
- BAD_FUNC_ARG 引数の 1 つが NULL の場合、または bufsz 引数が 0 の場合に返されます。

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.52.2.361 function wolfSSL_set_SessionTicket

```
int wolfSSL_set_SessionTicket(
    WOLFSSL * ssl,
    const unsigned char * buf,
    word32 bufSz
)
```

この関数は、WolfSSL 構造体内の wolfssl_session 構造体のチケットメンバーを設定します。関数に渡されたバッファはメモリにコピーされます。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WolfSSL 構造へのポインタ。
- **buf** セッション構造のチケットメンバーにロードされるバイトポインタ。

See: wolfSSL_set_SessionTicket_cb

Return:

- SSL_SUCCESS 機能の実行に成功したことに戻ります。関数はエラーなしで返されました。
- BAD_FUNC_ARG WolfSSL 構造が NULL の場合に返されます。BUF 引数が NULL の場合は、これはスローされますが、bufsz 引数はゼロではありません。

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.52.2.362 function wolfSSL_set_SessionTicket_cb

```
int wolfSSL_set_SessionTicket_cb(
    WOLFSSL * ssl,
    CallbackSessionTicket cb,
    void * ctx
)
```

CallbackSessionTicket は、int（* callbacksessionTicket）（wolfssl 、*const unsigned char* 、int、void *）の関数ポインタです。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WolfSSL 構造へのポインタ。

- **cb** Type CallbackSessionTicket への関数ポインタ。

See:

- wolfSSL_set_SessionTicket
- CallbackSessionTicket
- sessionTicketCB

Return:

- SSL_SUCCESS 関数がエラーなしで実行された場合に返されます。
- BAD_FUNC_ARG WolfSSL 構造が NULL の場合に返されます。

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.52.2.363 function wolfSSL_send_SessionTicket

```
int wolfSSL_send_SessionTicket(
    WOLFSSL * ssl
)
```

この関数は TLS1.3 ハンドシェークが確立したあとでセッションチケットを送信します。

Parameters:

- **ssl** wolfSSL_new()を使って生成された WOLFSSL 構造体へのポインタ。

See:

- wolfSSL_get_SessionTicket
- CallbackSessionTicket
- sessionTicketCB

Return:

- WOLFSSL_SUCCESS セッションチケットが送信された場合に返されます。
- BAD_FUNC_ARG WOLFSSL 構造体が NULL, あるいは TLS v1.3 を使用しない場合に返されます。
- SIDE_ERROR returned サーバー側でない場合に返されます。
- NOT_READY_ERROR ハンドシェークが完了しない場合に返されます。
- WOLFSSL_FATAL_ERROR メッセージの生成か送信に失敗した際に返されます。

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.52.2.364 function wolfSSL_CTX_set_TicketEncCb

```
int wolfSSL_CTX_set_TicketEncCb(
    WOLFSSL_CTX * ctx,
    SessionTicketEncCb
)
```

RFC 5077 で指定されているセッションチケットをサポートするためのサーバーが。

Parameters:

- **ctx** wolfSSL_CTX_new()で作成された WOLFSSL_CTX オブジェクトへのポインタ。
- **cb** セッションチケットを暗号化/復号化するためのユーザーコールバック関数
- **ssl(Callback)** wolfSSL_new()で作成された WolfSSL オブジェクトへのポインタ
- **key_name(Callback)** このチケットコンテキストの一意のキー名はランダムに生成されるべきです
- **iv(Callback)** ユニークな IV このチケットの場合、最大 128 ビット、ランダムに生成されるべきです
- **mac(Callback)** このチケットの最大 256 ビット MAC
- **enc(Callback)** この暗号化パラメータが true の場合、ユーザーはキーコード、IV、Mac を記入し、チケットを長さのインレルの範囲内に暗号化し、結果として生じる出力長を * outreen に設定する必要があります。wolfssl_ticket_ret_ok を返す暗号化が成功したことを WolfSSL に指示します。この暗号化パラメータが false の場合、key_name、iv、および mac を使用して、リングインレーンの範囲内のチケットの復号化を実行する必要があります。結果の復号長は * outreen に設定する必要があります。wolfssl_ticket_ret_ok を返すと、復号化されたチケットの使用を続行するように WolfSSL に指示します。wolfssl_ticket_ret_create を返すと、復号化されたチケットを使用するだけでなく、クライアントに送信するための新しいものを生成するように指示し、最近ロールされている場合に役立つ、フルハンドシェイクを強制したくない。wolfssl_ticket_ret_reject を返すと、WolfSSL にこのチケットを拒否し、フルハンドシェイクを実行し、通常のセッション再開のための新しい標準セッション ID を作成します。wolfssl_ticket_ret_fatal を返すと、致命的なエラーで接続の試みを終了するように WolfSSL に指示します。
- **ticket(Callback)** 暗号化チケットの入出力バッファ。ENC パラメータを参照してください
- **inLen(Callback)** チケットパラメータの入力長
- **outLen(Callback)** チケットパラメータの結果の出力長。コールバック outlen を入力すると、チケットバッファで使用可能な最大サイズが表示されます。

See:

- wolfSSL_CTX_set_TicketHint
- wolfSSL_CTX_set_TicketEncCtx

Return:

- SSL_SUCCESS セッションを正常に設定すると返されます。
- BAD_FUNC_ARG 失敗した場合に返されます。これは、無効な引数を関数に渡すことによって発生します。

Example

```
See wolfssl/test.h myTicketEncCb() used by the example
server and example echoserver.
```

C.52.2.365 function wolfSSL_CTX_set_TicketHint

```
int wolfSSL_CTX_set_TicketHint(
    WOLFSSL_CTX * ctx,
    int
)
```

サーバーサイドの使用のために。

Parameters:

- **ctx** wolfSSL_CTX_new()で作成された WOLFSSL_CTX オブジェクトへのポインタ。

See: wolfSSL_CTX_set_TicketEncCb

Return:

- SSL_SUCCESS セッションを正常に設定すると返されます。
- BAD_FUNC_ARG 失敗した場合に返されます。これは、無効な引数を関数に渡すことによって発生します。

Example

none

C.52.2.366 function wolfSSL_CTX_set_TicketEncCtx

```
int wolfSSL_CTX_set_TicketEncCtx(
    WOLFSSL_CTX * ctx,
    void *
)
```

折り返し電話。サーバーサイドの使用のために。

Parameters:

- **ctx** wolfSSL_CTX_new()で作成された WOLFSSL_CTX オブジェクトへのポインタ。

See: wolfSSL_CTX_set_TicketEncCb

Return:

- SSL_SUCCESS セッションを正常に設定すると返されます。
- BAD_FUNC_ARG 失敗した場合に返されます。これは、無効な引数を関数に渡すことによって発生します。

Example

none

C.52.2.367 function wolfSSL_CTX_get_TicketEncCtx

```
void * wolfSSL_CTX_get_TicketEncCtx(
    WOLFSSL_CTX * ctx
)
```

折り返し電話。サーバーサイドの使用のために。

See: wolfSSL_CTX_set_TicketEncCtx

Return:

- userCtx セッションを正常に取得すると返されます。
- NULL 失敗した場合に返されます。これは、無効な引数を関数に渡すことによって、またはユーザーコンテキストが設定されていないときに発生します。

Example

none

C.52.2.368 function wolfSSL_SetHsDoneCb

```
int wolfSSL_SetHsDoneCb(
    WOLFSSL * ssl,
    HandShakeDoneCb cb,
    void * user_ctx
)
```

この機能には、WolfSSL 構造の HSDonectx メンバーが設定されています。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WolfSSL 構造へのポインタ。
- **cb** int（* HandshakedOneCB）（wolfssl 、*void* ）の署名を持つタイプ HandshakedOneCB の関数ポインタ。

See: HandShakeDoneCb

Return:

- SSL_SUCCESS 関数がエラーなしで実行された場合に返されます。WolfSSL 構造体の HSDONECB と HSDonectx メンバーが設定されています。
- BAD_FUNC_ARG wolfssl 構造体が NULL の場合に返されます。

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.52.2.369 function wolfSSL_PrintSessionStats

```
int wolfSSL_PrintSessionStats(
    void
)
```

この関数はセッションから統計を印刷します。

See: wolfSSL_get_session_stats

Return:

- SSL_SUCCESS 関数とサブルーチンがエラーなしで戻った場合に返されます。セッション統計は正常に取得され印刷されました。
- BAD_FUNC_ARG サブルーチン wolfssl_get_session_stats() が許容できない引数に渡された場合に返されます。
- BAD_MUTEX_E サブルーチンにミューテックスエラーがあった場合に返されます。

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.52.2.370 function wolfSSL_get_session_stats

```
int wolfSSL_get_session_stats(
    unsigned int * active,
    unsigned int * total,
    unsigned int * peak,
    unsigned int * maxSessions
)
```

この関数はセッションの統計を取得します。

Parameters:

- **active** 現在のセッションの合計を表す Word32 ポインタ。
- **total** 総セッションを表す Word32 ポインタ。
- **peak** ピークセッションを表す Word32 ポインタ。

See: wolfSSL_PrintSessionStats

Return:

- SSL_SUCCESS 関数とサブルーチンがエラーなしで戻った場合に返されます。セッション統計は正常に取得され印刷されました。
- BAD_FUNC_ARG サブルーチン wolfssl_get_session_stats() が許容できない引数に渡された場合に返されます。
- BAD_MUTEX_E サブルーチンにミューテックスエラーがあった場合に返されます。

Example

```
int wolfSSL_PrintSessionStats(void){
…
ret = wolfSSL_get_session_stats(&totalSessionsNow,
&totalSessionsSeen, &peak, &maxSessions);
…
return ret;
```

C.52.2.371 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
)
```

この関数は CR と SR の値をコピーしてから WC_PRF（疑似ランダム関数）に渡し、その値を返します。

Parameters:

- **ms** マスターシークレットはアレイ構造に保持されています。
- **msLen** マスターシークレットの長さ。
- **pms** マスター前の秘密はアレイ構造に保持されています。
- **pmsLen** マスタープレマスターシークレットの長さ。
- **cr** クライアントのランダム
- **sr** サーバーのランダムです。
- **tls1_2** バージョンが少なくとも TLS バージョン 1.2 であることを意味します。

See:

- wc_PRF
- MakeTlsMasterSecret

Return:

- 0 成功した
- BUFFER_E バッファのサイズにエラーが発生した場合に返されます。
- MEMORY_E サブルーチンが動的メモリを割り当てることができなかった場合に返されます。

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.52.2.372 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
)
```

TLS キーを導き出すための外部のラッパー。

Parameters:

- **key_data** DeriveTlSkeys に割り当てられ、最終ハッシュを保持するために WC_PRF に渡されたバイトポインタ。
- **keyLen** WOLFSSL 構造体のスペックメンバーからの Derivetlskeys で派生した Word32 タイプ。
- **ms** WolfSSL 構造内でアレイ構造に保持されているマスターシークレットを保持する定数ポインタ型。
- **msLen** 列挙された定義で、マスターシークレットの長さを保持する Word32 タイプ。
- **sr** WOLFSSL 構造内の配列構造の ServerRandom メンバーへの定数バイトポインタ。
- **cr** WolfSSL 構造内の配列構造の ClientRandom メンバーへの定数バイトポインタ。
- **tls1_2** ISATLEASTLSV1_2() から返された整数型。

See:

- wc_PRF
- DeriveTlsKeys

- IsAtLeastTLSv1_2

Return:

- 0 成功に戻りました。
- BUFFER_E LABLEN と SEADLEN の合計（合計サイズを計算）が最大サイズを超えると返されます。
- MEMORY_E メモリの割り当てが失敗した場合に返されます。

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.52.2.373 function wolfSSL_connect_ex

```
int wolfSSL_connect_ex(
    WOLFSSL * ssl,
    HandShakeCallBack hsCb,
    TimeoutCallBack toCb,
    WOLFSSL_TIMEVAL timeout
)
```

ハンドシェイクコールバックが設定されます。これは、デバッガが利用できず、スニッフィングが実用的ではない場合に、サポートをデバッグするための組み込みシステムで役立ちます。ハンドシェイクエラーが発生したか否かが呼び出されます。SSL パケットの最大数が既知であるため、動的メモリは使用されません。パケット名を PacketNames [] でアクセスできます。接続拡張機能は、タイムアウト値とともにタイムアウトコールバックを設定することもできます。これは、ユーザーが TCP スタックをタイムアウトするのを待ったくない場合に便利です。この拡張子は、コールバックのどちらか、またはどちらのコールバックも呼び出されません。

See: wolfSSL_accept_ex

Return:

- SSL_SUCCESS 成功時に返されます。
- GETTIME_ERROR gettimeofday() がエラーを検出した場合、返されます。
- SETITIMER_ERROR setItimer() がエラーを検出した場合、返されます。
- SIGACT_ERROR sigAction() がエラーを検出した場合、返されます。
- SSL_FATAL_ERROR 基になる ssl_connect() 呼び出しがエラーを検出した場合に返されます。

Example

none

C.52.2.374 function wolfSSL_accept_ex

```
int wolfSSL_accept_ex(
    WOLFSSL * ssl,
    HandShakeCallBacki hsCb,
    TimeoutCallBack toCb,
    WOLFSSL_TIMEVAL timeout
)
```

設定する。これは、デバッガが利用できず、スニッフィングが実用的ではない場合に、サポートをデバッグするための組み込みシステムで役立ちます。ハンドシェイクエラーが発生したか否かが呼び出されます。SSL パケットの最大数が既知であるため、動的メモリは使用されません。パケット名を PacketNames [] でアクセスできます。接続拡張機能は、タイムアウト値とともにタイムアウトコールバックを設定することもできます。これは、ユーザーが TCP スタックをタイムアウトするのを待ったくない場合に便利です。この拡張子は、コールバックのどちらか、またはどちらのコールバックも呼び出されません。

See: wolfSSL_connect_ex

Return:

- SSL_SUCCESS 成功時に返されます。
- GETTIME_ERROR gettimeofday() がエラーを検出した場合、返されます。
- SETITIMER_ERROR setItimer() がエラーを検出した場合、返されます。
- SIGACT_ERROR sigAction() がエラーを検出した場合、返されます。
- SSL_FATAL_ERROR 基礎となる ssl_accept() 呼び出しがエラーを検出した場合に返されます。

Example

none

C.52.2.375 function wolfSSL_BIO_set_fp

```
long wolfSSL_BIO_set_fp(
    WOLFSSL_BIO * bio,
    XFILE fp,
    int c
)
```

これは BIO の内部ファイルポインタを設定するために使用されます。

Parameters:

- **bio** ペアを設定するための WOLFSSL_BIO 構造体。
- **fp** バイオで設定するファイルポインタ。

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_s_mem
- wolfSSL_BIO_get_fp
- wolfSSL_BIO_free

Return:

- SSL_SUCCESS ファイルポインタを正常に設定します。
- SSL_FAILURE エラーケースに遭遇した場合

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.52.2.376 function wolfSSL_BIO_get_fp

```
long wolfSSL_BIO_get_fp(
    WOLFSSL_BIO * bio,
```

```
    XFILE * fp
)
```

この関数は、

Parameters:

- **bio** ペアを設定するための WOLFSSL_BIO 構造体。

See:

- wolfSSL_BIO_new
- wolfSSL_BIO_s_mem
- wolfSSL_BIO_set_fp
- wolfSSL_BIO_free

Return:

- SSL_SUCCESS ファイルポインタを正常に取得します。
- SSL_FAILURE エラーケースに遭遇した場合

```
ingroup IO
```

これは、BIO の内部ファイルポインタを取得するために使用されます。*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.52.2.377 function wolfSSL_check_private_key

```
int wolfSSL_check_private_key(
    const WOLFSSL * ssl
)
```

この関数は、秘密鍵が使用されている証明書との一致であることを確認します。

See:

- wolfSSL_new
- wolfSSL_free

Return:

- SSL_SUCCESS うまく一致します。
- SSL_FAILURE エラーケースに遭遇した場合
- <0 ssl_failure 以外のすべてのエラーケースは負の値です。

Example

```
WOLFSSL* ssl;
int ret;
// create and set up ssl
ret  = wolfSSL_check_private_key(ssl);
// check ret value
```

C.52.2.378 function wolfSSL_X509_get_ext_by_NID

```
int wolfSSL_X509_get_ext_by_NID(
    const WOLFSSL_X509 * x509,
    int nid,
    int lastPos
)
```

この機能は、渡された NID 値に一致する拡張索引を探して返します。

Parameters:

- **x509** 拡張のために解析する証明書。
- **nid** 見つかる拡張 OID。

Return:

- = 0 拡張インデックスが成功した場合に返されます。
- -1 拡張が見つからないかエラーが発生した場合

Example

```
const WOLFSSL_X509* x509;
int lastPos = -1;
int idx;

idx = wolfSSL_X509_get_ext_by_NID(x509, NID_basic_constraints, lastPos);
```

C.52.2.379 function wolfSSL_X509_get_ext_d2i

```
void * wolfSSL_X509_get_ext_d2i(
    const WOLFSSL_X509 * x509,
    int nid,
    int * c,
    int * idx
)
```

この関数は、渡された NID 値に合った拡張子を探して返します。

Parameters:

- **x509** 拡張のために解析する証明書。
- **nid** 見つかる拡張 OID。
- **c** not null が複数の拡張子に-2 に設定されていない場合は-1 が見つかりませんでした。

See: wolfSSL_sk_ASN1_OBJECT_free

Return:

- pointer STACK_OF（wolfssl_asn1_object）ポインタが成功した場合に返されます。
- NULL 拡張が見つからないかエラーが発生した場合

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.52.2.380 function wolfSSL_X509_digest

```
int wolfSSL_X509_digest(
    const WOLFSSL_X509 * x509,
    const WOLFSSL_EVP_MD * digest,
    unsigned char * buf,
    unsigned int * len
)
```

この関数は DER 証明書のハッシュを返します。

Parameters:

- **x509** ハッシュを得るための証明書。
- **digest** 使用するハッシュアルゴリズム
- **buf** ハッシュを保持するためのバッファ。

See: none

Return:

- SSL_SUCCESS ハッシュの作成に成功しました。
- SSL_FAILURE 不良入力または失敗したハッシュに戻りました。

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.52.2.381 function wolfSSL_use_certificate

```
int wolfSSL_use_certificate(
    WOLFSSL * ssl,
    WOLFSSL_X509 * x509
)
```

ハンドシェイク中に使用するために、WolfSSL 構造の証明書を設定するために使用されます。

Parameters:

- **ssl** 証明書を設定するための WolfSSL 構造。

See:

- wolfSSL_new
- wolfSSL_free

Return:

- SSL_SUCCESS 設定の成功した引数について。
- SSL_FAILURE NULL 引数が渡された場合。

Example

```
 WOLFSSL* ssl;
WOLFSSL_X509* x509
int ret;
// create ssl object and x509
```

```
ret  = wolfSSL_use_certificate(ssl, x509);
// check ret value
```

C.52.2.382 function wolfSSL_use_certificate_ASN1

```
int wolfSSL_use_certificate_ASN1(
    WOLFSSL * ssl,
    unsigned char * der,
    int derSz
)
```

Parameters:

- **ssl** 証明書を設定するための WolfSSL 構造。
- **der** 使用する証明書。

See:

- wolfSSL_new
- wolfSSL_free

Return:

- SSL_SUCCESS 設定の成功した引数について。
- SSL_FAILURE NULL 引数が渡された場合。

bii こ f は、この関数は、handshake の間に使用するために WolfSSL 構造の証明書を設定するために使用されます。DER フォーマットバッファが予想されます。*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.52.2.383 function wolfSSL_use_PrivateKey

```
int wolfSSL_use_PrivateKey(
    WOLFSSL * ssl,
    WOLFSSL_EVP_PKEY * pkey
)
```

これは WolfSSL 構造の秘密鍵を設定するために使用されます。

Parameters:

- **ssl** 引数を設定するための WolfSSL 構造。

See:

- wolfSSL_new
- wolfSSL_free

Return:

- SSL_SUCCESS 設定の成功した引数について。
- SSL_FAILURE NULL SSL が渡された場合。すべてのエラーケースは負の値になります。

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.52.2.384 function wolfSSL_use_PrivateKey_ASN1

```
int wolfSSL_use_PrivateKey_ASN1(
    int pri,
    WOLFSSL * ssl,
    unsigned char * der,
    long derSz
)
```

これは WolfSSL 構造の秘密鍵を設定するために使用されます。DER フォーマットのキーバッファが予想されます。

Parameters:

- **pri** 秘密鍵の種類。
- **ssl** 引数を設定するための WolfSSL 構造。
- **der** バッファー保持 DER キー。

See:

- wolfSSL_new
- wolfSSL_free
- wolfSSL_use_PrivateKey

Return:

- SSL_SUCCESS 秘密鍵の構文解析と設定に成功した場合。
- SSL_FAILURE NULL SSL が渡された場合。すべてのエラーケースは負の値になります。

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.52.2.385 function wolfSSL_use_RSAPrivateKey_ASN1

```
int wolfSSL_use_RSAPrivateKey_ASN1(
    WOLFSSL * ssl,
    unsigned char * der,
    long derSz
)
```

これは WolfSSL 構造の秘密鍵を設定するために使用されます。DER フォーマットの RSA キーバッファが予想されます。

Parameters:

- **ssl** 引数を設定するための WolfSSL 構造。

- **der** バッファー保持 DER キー。

See:

- wolfSSL_new
- wolfSSL_free
- wolfSSL_use_PrivateKey

Return:

- SSL_SUCCESS 秘密鍵の構文解析と設定に成功した場合。
- SSL_FAILURE NULL SSL が渡された場合。すべてのエラーケースは負の値になります。

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.52.2.386 function wolfSSL_DSA_dup_DH

```
WOLFSSL_DH * wolfSSL_DSA_dup_DH(
    const WOLFSSL_DSA * r
)
```

この関数は、DSA のパラメータを新しく作成された WOLFSSL_DH 構造体に重複しています。

See: none

Return:

- WOLFSSL_DH 重複した場合は WolfSSL_DH 構造体を返す場合
- NULL 失敗すると

Example

```
WOLFSSL_DH* dh;
WOLFSSL_DSA* dsa;
// set up dsa
dh = wolfSSL_DSA_dup_DH(dsa);

// check dh is not null
```

C.52.2.387 function wolfSSL_SESSION_get_master_key

```
int wolfSSL_SESSION_get_master_key(
    const WOLFSSL_SESSION * ses,
    unsigned char * out,
    int outSz
)
```

これはハンドシェイクを完了した後にマスターキーを取得するために使用されます。

Parameters:

- **ses** マスターシークレットバッファを取得するための WolfSSL_SESSION 構造。
- **out** データを保持するためのバッファ。

See:

- wolfSSL_new
- wolfSSL_free

Return:

- 0 データの取得に成功した場合、0 より大きい値を返します。
- 0 ランダムなデータバッファまたはエラー状態が返されない場合は 0
- max 渡された OUTSZ が 0 の場合、必要な最大バッファサイズが返されます。

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.52.2.388 function wolfSSL_SESSION_get_master_key_length

```
int wolfSSL_SESSION_get_master_key_length(
    const WOLFSSL_SESSION * ses
)
```

これはマスター秘密鍵の長さを取得するために使用されます。

See:

- wolfSSL_new
- wolfSSL_free

Return: size マスターシークレットキーサイズを返します。

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.52.2.389 function wolfSSL_CTX_set_cert_store

```
void wolfSSL_CTX_set_cert_store(
    WOLFSSL_CTX * ctx,
    WOLFSSL_X509_STORE * str
)
```

Parameters:

- **ctx** Cert Store ポインタを設定するための WolfSSL_CTX 構造体へのポインタ。

See:

- wolfSSL_CTX_new
- wolfSSL_CTX_free

Return: none 返品不可。

bri f この関数は、れは、CTX の WOLFSSL_X509_STORE 構造の設定機能です。*Example*

```
WOLFSSL_CTX ctx;
WOLFSSL_X509_STORE* st;
// setup ctx and st
st = wolfSSL_CTX_set_cert_store(ctx, st);
//use st
```

C.52.2.390 function wolfSSL_d2i_X509_bio

```
WOLFSSL_X509 * wolfSSL_d2i_X509_bio(
    WOLFSSL_BIO * bio,
    WOLFSSL_X509 ** x509
)
```

この関数は BIO から DER バッファを取得し、それを WolfSSL_X509 構造に変換します。

Parameters:

- **bio** DER 証明書バッファを持つ WOLFSSL_BIO 構造体体へのポインタ。

See: none

Return:

- pointer 成功した wolfssl_x509 構造ポインタを返します。
- Null 失敗時に NULL を返します

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.52.2.391 function wolfSSL_CTX_get_cert_store

```
WOLFSSL_X509_STORE * wolfSSL_CTX_get_cert_store(
    WOLFSSL_CTX * ctx
)
```

See:

- wolfSSL_CTX_new
- wolfSSL_CTX_free
- wolfSSL_CTX_set_cert_store

Return:

- WOLFSSL_X509_STORE* ポインタを正常に入手します。
- NULL NULL 引数が渡された場合に返されます。

bri f この関数は、れは、CTX の WOLFSSL_X509_STORE 構造のゲッター関数です。*Example*

```
WOLFSSL_CTX ctx;
WOLFSSL_X509_STORE* st;
// setup ctx
st = wolfSSL_CTX_get_cert_store(ctx);
//use st
```

C.52.2.392 function wolfSSL_BIO_ctrl_pending

```
size_t wolfSSL_BIO_ctrl_pending(
    WOLFSSL_BIO * b
)
```

保留中のバイト数を読み取る数を取得します。BIO タイプが BIO_BIO の場合、ペアから読み取る番号です。BIO に SSL オブジェクトが含まれている場合は、SSL オブジェクトからのデータを保留中です（WolfSSL_Pending（SSL））。bio_memory タイプがある場合は、メモリバッファのサイズを返します。

See:

- wolfSSL_BIO_make_bio_pair
- wolfSSL_BIO_new

Return: >=0 保留中のバイト数。

Example

```
WOLFSSL_BIO* bio;
int pending;
bio = wolfSSL_BIO_new();
…
pending = wolfSSL_BIO_ctrl_pending(bio);
```

C.52.2.393 function wolfSSL_get_server_random

```
size_t wolfSSL_get_server_random(
    const WOLFSSL * ssl,
    unsigned char * out,
    size_t outlen
)
```

Parameters:

- **ssl** クライアントのランダムデータバッファを取得するための WolfSSL 構造。
- **out** ランダムデータを保持するためのバッファ。

See:

- wolfSSL_new
- wolfSSL_free

Return:

- 0 データの取得に成功した場合、0 より大きい値を返します。
- 0 ランダムなデータバッファまたはエラー状態が返されない場合は 0
- max 渡された OUTSZ が 0 の場合、必要な最大バッファサイズが返されます。

biief は、この関数は、ハンドシェイク中にサーバーによって送信されたランダムなデータを取得するために使用されます。*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.52.2.394 function wolfSSL_get_client_random

```
size_t wolfSSL_get_client_random(
    const WOLFSSL * ssl,
    unsigned char * out,
    size_t outSz
)
```

Parameters:

- **ssl** クライアントのランダムデータバッファを取得するための WolfSSL 構造。
- **out** ランダムデータを保持するためのバッファ。

See:

- wolfSSL_new
- wolfSSL_free

Return:

- 0 データの取得に成功した場合、0 より大きい値を返します。
- 0 ランダムなデータバッファまたはエラー状態が返されない場合は 0
- max 渡された OUTSZ が 0 の場合、必要な最大バッファサイズが返されます。

biief は、この関数は、ハンドシェイク中にクライアントによって送信されたランダムなデータを取得するために使用されます。*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.52.2.395 function wolfSSL_CTX_get_default_passwd_cb

```
wc_pem_password_cb * wolfSSL_CTX_get_default_passwd_cb(
    WOLFSSL_CTX * ctx
)
```

これは CTX で設定されたパスワードコールバックのゲッター関数です。

See:

- wolfSSL_CTX_new
- wolfSSL_CTX_free

Return:

- func 成功すると、コールバック関数を返します。
- NULL CTX が NULL の場合、NULL が返されます。

Example

```
WOLFSSL_CTX* ctx;
wc_pem_password_cb cb;
// setup ctx
cb = wolfSSL_CTX_get_default_passwd_cb(ctx);
//use cb
```

C.52.2.396 function wolfSSL_CTX_get_default_passwd_cb_userdata

```
void * wolfSSL_CTX_get_default_passwd_cb_userdata(
    WOLFSSL_CTX * ctx
)
```

See:

- wolfSSL_CTX_new
- wolfSSL_CTX_free

Return:

- pointer 成功すると、ユーザーデータポインタを返します。
- NULL CTX が NULL の場合、NULL が返されます。

bri f この関数は、れは、CTX で設定されているパスワードコールバックユーザーデータの取得機能です。
Example

```
WOLFSSL_CTX* ctx;
void* data;
// setup ctx
data = wolfSSL_CTX_get_default_passwd_cb(ctx);
//use data
```

C.52.2.397 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
)
```

この関数は wolfssl_pem_read_bio_x509 と同じように動作します。AUX は、信頼できる/拒否されたユースケースや人間の読みやすさのためのフレンドリーな名前などの追加情報を含むことを意味します。

Parameters:

- **bp** WOLFSSL_BIO 構造体体から PEM バッファを取得します。
- **x** wolfssl_x509 を機能副作用で設定する場合
- **cb** パスワードコールバック

See: wolfSSL_PEM_read_bio_X509

Return:

- WOLFSSL_X509 PEM バッファの解析に成功した場合、wolfssl_x509 構造が返されます。
- Null PEM バッファの解析に失敗した場合。

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.52.2.398 function wolfSSL_CTX_set_tmp_dh

```
long wolfSSL_CTX_set_tmp_dh(
    WOLFSSL_CTX * ctx,
    WOLFSSL_DH * dh
)
```

WOLFSSL_CTX 構造体の DH メンバーを diffie-hellman パラメータで初期化します。

Parameters:

- **ctx** wolfSSL_CTX_new()を使用して作成された WOLFSSL_CTX 構造体へのポインタ。

See: wolfSSL_BN_bn2bin

Return:

- SSL_SUCCESS 関数が正常に実行された場合に返されます。
- BAD_FUNC_ARG CTX または DH 構造体が NULL の場合に返されます。
- SSL_FATAL_ERROR 構造値を設定するエラーが発生した場合に返されます。
- MEMORY_E メモリを割り当てることができなかった場合に返されます。

Example

```
WOLFSSL_CTX* ctx = wolfSSL_CTX_new( protocol method );
WOLFSSL_DH* dh;
...
return wolfSSL_CTX_set_tmp_dh(ctx, dh);
```

C.52.2.399 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
)
```

この関数は、BIO の PEM バッファから DSA パラメータを取得します。

Parameters:

- **bio** PEM メモリポインタを取得するための WOLFSSL_BIO 構造体体へのポインタ。
- **x** 新しい WolfSSL_DSA 構造に設定するポインタ。
- **cb** パスワードコールバック関数

See: none

Return:

- WOLFSSL_DSA PEM バッファの解析に成功した場合、WolfSSL_DSA 構造が作成され、返されます。
- Null PEM バッファの解析に失敗した場合。

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.52.2.400 function wolfSSL_ERR_peek_last_error

```
unsigned long wolfSSL_ERR_peek_last_error(
    void
)
```

この関数は、wolfssl_Error に遭遇した最後のエラーの絶対値を返します。

See: wolfSSL_ERR_print_errors_fp

Return: error 最後のエラーの絶対値を返します。

Example

```
unsigned long err;
...
err = wolfSSL_ERR_peek_last_error();
// inspect err value
```

C.52.2.401 function WOLF_STACK_OF

```
WOLF_STACK_OF(
    WOLFSSL_X509
) const
```

この関数はピアの証明書チェーンを取得します。

See:

- wolfSSL_X509_get_issuer_name
- wolfSSL_X509_get_subject_name
- wolfSSL_X509_get_isCA

Return:

- pointer ピアの証明書スタックへのポインタを返します。
- NULL ピア証明書がない場合に返されます。

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.52.2.402 function wolfSSL_CTX_clear_options

```
long wolfSSL_CTX_clear_options(
    WOLFSSL_CTX * ctx,
```

```
    long opt
)
```

この関数は、WOLFSSL_CTX オブジェクトのオプションビットをリセットします。

See:

- wolfSSL_CTX_new
- wolfSSL_new
- wolfSSL_free

Return: option 新しいオプションビット

Example

```
WOLFSSL_CTX* ctx = 0;
...
wolfSSL_CTX_clear_options(ctx, SSL_OP_NO_TLSv1);
```

C.52.2.403 function wolfSSL_set_jobject

```
int wolfSSL_set_jobject(
    WOLFSSL * ssl,
    void * objPtr
)
```

この関数は、WolfSSL 構造の jobjectref メンバーを設定します。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WolfSSL 構造へのポインタ。

See: wolfSSL_get_jobject

Return:

- SSL_SUCCESS jobjectref が objptr に正しく設定されている場合に返されます。
- SSL_FAILURE 関数が正しく実行されず、jobjectref が設定されていない場合に返されます。

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.52.2.404 function wolfSSL_get_jobject

```
void * wolfSSL_get_jobject(
    WOLFSSL * ssl
)
```

この関数は、wolfssl 構造の jobjectref メンバーを返します。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WolfSSL 構造へのポインタ。

See: wolfSSL_set_jobject

Return:

- value wolfssl 構造体が null でない場合、関数は jobjectref 値を返します。
- NULL wolfssl 構造体が NULL の場合に返されます。

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.52.2.405 function wolfSSL_set_msg_callback

```
int wolfSSL_set_msg_callback(
    WOLFSSL * ssl,
    SSL_Msg_Cb cb
)
```

この関数は SSL 内のコールバックを設定します。コールバックはハンドシェイクメッセージを観察することです。CB の NULL 値はコールバックをリセットします。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WolfSSL 構造へのポインタ。

See: wolfSSL_set_msg_callback_arg

Return:

- SSL_SUCCESS 成功しています。
- SSL_FAILURE NULL SSL が渡された場合。

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.52.2.406 function wolfSSL_set_msg_callback_arg

```
int wolfSSL_set_msg_callback_arg(
    WOLFSSL * ssl,
    void * arg
)
```

この関数は、SSL 内の関連コールバックコンテキスト値を設定します。値はコールバック引数に渡されます。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WolfSSL 構造へのポインタ。

See: wolfSSL_set_msg_callback

Return: none 返品不可。

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.52.2.407 function wolfSSL_X509_get_next_altname

```
char * wolfSSL_X509_get_next_altname(
    WOLFSSL_X509 * x509
)
```

この関数は、存在する場合は、ピア証明書から altname を返します。

See:

- wolfSSL_X509_get_issuer_name
- wolfSSL_X509_get_subject_name

Return:

- NULL 次の AltName がない場合。
- cert->altNamesNext->name wolfssl_x509 から、AltName リストからの文字列値である構造が存在する場合に返されます。

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.52.2.408 function wolfSSL_X509_get_notBefore

```
WOLFSSL_ASN1_TIME * wolfSSL_X509_get_notBefore(
    WOLFSSL_X509 * x509
)
```

関数は、x509 が null のかどうかを確認し、そうでない場合は、WOLFSSL_X509 構造体の NotBefore メンバーを返します。

Parameters:

- **x509** WOLFSSL_X509 構造体へのポインタ

See: wolfSSL_X509_get_notAfter

Return:

- pointer WOLFSSL_ASN1_TIME へのポインタ（WOLFSSL_X509 構造体の NotBefore メンバーへのポインタ）を返します。
- NULL WOLFSSL_X509 構造体が NULL の場合に返されます。

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.52.2.409 function wolfSSL_connect

```
int wolfSSL_connect(
    WOLFSSL * ssl
)
```

この関数はクライアント側で呼び出され、サーバーとの SSL/TLS ハンドシェイクを開始します。この関数が呼び出されるまでに下層の通信チャネルはすでに設定されている必要があります。wolfSSL_connect()は、ブロッキングとノンブロッキング I/O の両方で動作します。下層の I/O がノンブロッキングの場合、wolfSSL_connect() は、下層の I/O が wolfSSL_connect の要求（送信データ、受信データ）を満たすことができなかったときには即戻ります。この場合、wolfSSL_get_error() の呼び出しで SSL_ERROR_WANT_READ または SSL_ERROR_WANT_WRITE のいずれかが返されます。呼び出したプロセスは、下層の I/O ががREADY になった時点で、WOLFSSL が停止したときから再開できるように wolfSSL_connect() への呼び出しを繰り返す必要があります。これには select() を使用して必要な条件が整ったかどうかを確認できます。ブロッキング I/O を使用する場合は、ハンドシェークが終了するかエラーが発生するまで戻ってきません。wolfSSL は OpenSSL と比べて証明書検証に異なるアプローチを取ります。クライアントのデフォルトポリシーはサーバーを認証することです。これは、CA 証明書を読み込まない場合、サーバーを確認することができず”_155” のエラーコードが返されます。OpenSSL と同じ振る舞い（つまり、CA 証明書のロードなしでサーバー認証を成功させる）を取らせたい場合には、セキュリティ面でお勧めはしませんが、SSL_CTX_SET_VERIFY（ctx、SSL_VERIFY_NONE、0) を呼び出すことで可能となります。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WolfSSL 構造へのポインタ。

See:

- wolfSSL_get_error
- wolfSSL_accept

Return:

- SSL_SUCCESS 成功した場合に返されます。
- SSL_FATAL_ERROR エラーが発生した場合に返されます。より詳細なエラーコードを取得するには、wolfSSL_get_error() を呼び出します。

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.52.2.410 function wolfSSL_send_hrr_cookie

```
int wolfSSL_send_hrr_cookie(
    WOLFSSL * ssl,
    const unsigned char * secret,
    unsigned int secretSz
)
```

この関数はサーバー側で呼び出されて、HellORetryRequest メッセージに Cookie を含める必要があることを示します。Cookie は現在のトランスクリプトのハッシュを保持しているので、別のサーバープロセスは応答で ClientHello を処理できます。秘密は Cookie データの整合性チェックを Generting するときに使用されます。

Parameters:

- **ssl** l wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ。
- **秘密を保持しているバッファへのポインタを秘密にします。渡す NULL は、新しいランダムシークレットを生成することを示します。**
- **シークスのサイズをバイト単位でサイズ。0 を渡すと、デフォルトのサイズを使用することを示します.WC_SHA256_DIGEST_SIZE（または SHA-256 が使用できない場合は WC_SHA_DIGEST_SIZE)。**

See: wolfSSL_new

Return:

- BAD_FUNC_ARG ssl が NULL の場合、または TLS v1.3 を使用していない場合。
- SIDE_ERROR クライアントで呼び出された場合。
- WOLFSSL_SUCCESS 成功した場合に返されます。
- MEMORY_ERROR 秘密を保存するために動的メモリを割り当てる場合に失敗しました。

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.52.2.411 function wolfSSL_disable_hrr_cookie

```
int wolfSSL_disable_hrr_cookie(
    WOLFSSL * ssl
)
```

この関数はサーバー側で呼び出され、HelloRetryRequest メッセージがクッキーを含んではならないこと、DTLSv1.3 が使用されている場合にはクッキーの交換がハンドシェークに含まれないことを表明します。DTLSv1.3 ではクッキー交換を行わないとサーバーが DoS/Amplification 攻撃を受けやすくなる可能性があることに留意してください。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ。

See: wolfSSL_send_hrr_cookie

Return:

- WOLFSSL_SUCCESS 成功時に返されます。
- BAD_FUNC_ARG ssl が NULL あるいは TLS v1.3 を使用していない場合に返されます。

- SIDE_ERROR クライアント側でこの関数が呼び出された場合に返されます。

C.52.2.412 function wolfSSL_CTX_no_ticket_TLSv13

```
int wolfSSL_CTX_no_ticket_TLSv13(
    WOLFSSL_CTX * ctx
)
```

この関数はサーバー上で呼び出され、ハンドシェイク完了時にセッション再開のためのセッションチケットの送信を行わないようにします。

Parameters:

- **ctx** wolfSSL_CTX_new()で作成された WOLFSSL_CTX 構造体へのポインタ。

See: wolfSSL_no_ticket_TLSv13

Return:

- BAD_FUNC_ARG CTX が NULL の場合、または TLS v1.3 を使用していない場合。
- SIDE_ERROR クライアントで呼び出された場合。

Example

```
int ret;
WOLFSSL_CTX* ctx;
...
ret = wolfSSL_CTX_no_ticket_TLSv13(ctx);
if (ret != 0) {
    // failed to set no ticket
}
```

C.52.2.413 function wolfSSL_no_ticket_TLSv13

```
int wolfSSL_no_ticket_TLSv13(
    WOLFSSL * ssl
)
```

ハンドシェイクが完了すると、この関数はサーバー上で再開セッションチケットの送信を停止するように呼び出されます。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ。

See: wolfSSL_CTX_no_ticket_TLSv13

Return:

- BAD_FUNC_ARG ssl が NULL の場合、または TLS v1.3 を使用していない場合。
- SIDE_ERROR クライアントで呼び出された場合。

Example

```
int ret;
WOLFSSL* ssl;
...
ret = wolfSSL_no_ticket_TLSv13(ssl);
if (ret != 0) {
    // failed to set no ticket
}
```

C.52.2.414 function wolfSSL_CTX_no_dhe_psk

```
int wolfSSL_CTX_no_dhe_psk(
    WOLFSSL_CTX * ctx
)
```

この関数は、Authentication にプリシェアキーを使用している場合、DIFFIE-HELLMAN（DH）スタイルのキー交換を許可する TLS V1.3 WolfSSL コンテキストで呼び出されます。

Parameters:

- **ctx** wolfSSL_CTX_new()で作成された WOLFSSL_CTX 構造体へのポインタ。

See: wolfSSL_no_dhe_psk

Return: BAD_FUNC_ARG CTX が NULL の場合、または TLS v1.3 を使用していない場合。

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.52.2.415 function wolfSSL_no_dhe_psk

```
int wolfSSL_no_dhe_psk(
    WOLFSSL * ssl
)
```

この関数は、事前共有鍵を使用している TLS V1.3 クライアントまたはサーバーで、に Diffie-Hellman（DH）スタイルの鍵交換を許可しないように設定します。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ。

See: wolfSSL_CTX_no_dhe_psk

Return: BAD_FUNC_ARG ssl が NULL の場合、または TLS v1.3 を使用していない場合。

Example

```
int ret;
WOLFSSL* ssl;
...
ret = wolfSSL_no_dhe_psk(ssl);
if (ret != 0) {
    // failed to set no DHE for PSK handshakes
}
```

C.52.2.416 function wolfSSL_update_keys

```
int wolfSSL_update_keys(
    WOLFSSL * ssl
)
```

この関数は、TLS v1.3 クライアントまたはサーバーの wolfssl で呼び出されて、キーのロールオーバーを強制します。KeyUpdate メッセージがピアに送信され、新しいキーが暗号化のために計算されます。ピアは

KeyUpdate メッセージを送り、新しい復号化キー WIL を計算します。この機能は、ハンドシェイクが完了した後にのみ呼び出すことができます。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ。

See: wolfSSL_write

Return:

- BAD_FUNC_ARG ssl が NULL の場合、または TLS v1.3 を使用していない場合。
- WANT_WRITE 書き込みが準備ができていない場合

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.52.2.417 function wolfSSL_key_update_response

```
int wolfSSL_key_update_response(
    WOLFSSL * ssl,
    int * required
)
```

この関数は、TLS v1.3 クライアントまたはサーバーの wolfssl で呼び出され、キーのロールオーバーが進行中かどうかを判断します。wolfssl_update_keys() が呼び出されると、KeyUpdate メッセージが送信され、暗号化キーが更新されます。復号化キーは、応答が受信されたときに更新されます。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ。
- **キー更新応答が必要ない場合は必須 0。1 キー更新応答が必要ない場合。**

See: wolfSSL_update_keys

Return: 0 成功した。

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.52.2.418 function wolfSSL_CTX_allow_post_handshake_auth

```
int wolfSSL_CTX_allow_post_handshake_auth(
    WOLFSSL_CTX * ctx
)
```

この関数は、TLS v1.3 クライアントの WolfSSL コンテキストで呼び出され、クライアントはサーバーからの要求に応じて Post Handshake を送信できるようにします。これは、クライアント認証などを必要としないページを持つ Web サーバーに接続するときに役立ちます。

Parameters:

- **ctx** wolfSSL_CTX_new()で作成された WOLFSSL_CTX 構造体へのポインタ。

See:

- wolfSSL_allow_post_handshake_auth
- wolfSSL_request_certificate

Return:

- BAD_FUNC_ARG CTX が NULL の場合、または TLS v1.3 を使用していない場合。
- SIDE_ERROR サーバーで呼び出された場合。

Example

```
int ret;
WOLFSSL_CTX* ctx;
...
ret = wolfSSL_allow_post_handshake_auth(ctx);
if (ret != 0) {
    // failed to allow post handshake authentication
}
```

C.52.2.419 function wolfSSL_allow_post_handshake_auth

```
int wolfSSL_allow_post_handshake_auth(
    WOLFSSL * ssl
)
```

この関数は、TLS V1.3 クライアント WolfSSL で呼び出され、クライアントはサーバーからの要求に応じてハンドシェイクを送ります。handshake クライアント認証拡張機能は ClientHello で送信されます。これは、クライアント認証などを必要としないページを持つ Web サーバーに接続するときに役立ちます。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ。

See:

- wolfSSL_CTX_allow_post_handshake_auth
- wolfSSL_request_certificate

Return:

- BAD_FUNC_ARG ssl が NULL の場合、または TLS v1.3 を使用していない場合。
- SIDE_ERROR サーバーで呼び出された場合。

Example

```
int ret;
WOLFSSL* ssl;
...
ret = wolfSSL_allow_post_handshake_auth(ssl);
```

```
if (ret != 0) {
    // failed to allow post handshake authentication
}
```

C.52.2.420 function wolfSSL_request_certificate

```
int wolfSSL_request_certificate(
    WOLFSSL * ssl
)
```

この関数は、TLS v1.3 クライアントからクライアント証明書を要求します。これは、Web サーバーがクライアント認証やその他のものを必要とするページにサービスを提供している場合に役立ちます。接続で最大 256 の要求を送信できます。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ。

See:

- wolfSSL_allow_post_handshake_auth
- wolfSSL_write

Return:

- BAD_FUNC_ARG ssl が NULL の場合、または TLS v1.3 を使用していない場合。
- WANT_WRITE 書き込みが準備ができていない場合
- SIDE_ERROR クライアントで呼び出された場合。
- NOT_READY_ERROR ハンドシェイクが終了していないときに呼び出された場合。
- POST_HAND_AUTH_ERROR 送付後認証が許可されていない場合。
- MEMORY_E 動的メモリ割り当てが失敗した場合

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.52.2.421 function wolfSSL_CTX_set1_groups_list

```
int wolfSSL_CTX_set1_groups_list(
    WOLFSSL_CTX * ctx,
    char * list
)
```

この関数は楕円曲線グループのリストを設定して、WolfSSL コンテキストを希望の順に設定します。リストはヌル終了したテキスト文字列、およびコロン区切りリストです。この関数を呼び出して、TLS v1.3 接続で使用する鍵交換楕円曲線パラメータを設定します。

Parameters:

- **ctx** wolfSSL_CTX_new()で作成された WOLFSSL_CTX 構造体へのポインタ。
- **list** 楕円曲線グループのコロン区切りリストである文字列をリストします。

See:

- wolfSSL_set1_groups_list
- wolfSSL_CTX_set_groups
- wolfSSL_set_groups
- wolfSSL_UseKeyShare
- wolfSSL_preferred_group

Return: WOLFSSL_FAILURE ポインタパラメータが NULL の場合、wolfssl_max_group_count グループが多い場合は、グループ名が認識されないか、TLS v1.3 を使用していません。

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.52.2.422 function wolfSSL_set1_groups_list

```
int wolfSSL_set1_groups_list(
    WOLFSSL * ssl,
    char * list
)
```

この関数は楕円曲線グループのリストを設定して、WolfSSL を希望の順に設定します。リストはヌル終了したテキスト文字列、およびコロン区切りリストです。この関数を呼び出して、TLS v1.3 接続で使用する鍵交換楕円曲線パラメータを設定します。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ。
- **list** 鍵交換グループのコロン区切りリストである文字列をリストします。

See:

- wolfSSL_CTX_set1_groups_list
- wolfSSL_CTX_set_groups
- wolfSSL_set_groups
- wolfSSL_UseKeyShare
- wolfSSL_preferred_group

Return: WOLFSSL_FAILURE ポインタパラメータが NULL の場合、wolfssl_max_group_count グループが多い場合は、グループ名が認識されないか、TLS v1.3 を使用していません。

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.52.2.423 function wolfSSL_preferred_group

```
int wolfSSL_preferred_group(
    WOLFSSL * ssl
)
```

この関数は、クライアントが TLS v1.3 ハンドシェイクで使用することを好む鍵交換グループを返します。この情報を完了した後にこの機能を呼び出して、サーバーがどのグループが予想されるようにこの情報が将来の接続で使用できるようになるかを決定するために、この情報が将来の接続で鍵交換のための鍵ペアを事前生成することができます。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ。

See:

- wolfSSL_UseKeyShare
- wolfSSL_CTX_set_groups
- wolfSSL_set_groups
- wolfSSL_CTX_set1_groups_list
- wolfSSL_set1_groups_list

Return:

- BAD_FUNC_ARG ssl が NULL の場合、または TLS v1.3 を使用していない場合。
- SIDE_ERROR サーバーで呼び出された場合。
- NOT_READY_ERROR ハンドシェイクが完了する前に呼び出された場合。

Example

```
int ret;
int group;
WOLFSSL* ssl;
...
ret = wolfSSL_CTX_set1_groups_list(ssl)
if (ret < 0) {
    // failed to get group
}
group = ret;
```

C.52.2.424 function wolfSSL_CTX_set_groups

```
int wolfSSL_CTX_set_groups(
    WOLFSSL_CTX * ctx,
    int * groups,
    int count
)
```

この関数は楕円曲線グループのリストを設定して、WolfSSL コンテキストを希望の順に設定します。リストは、Count で指定された識別子の数を持つグループ識別子の配列です。この関数を呼び出して、TLS v1.3 接続で使用する鍵交換楕円曲線パラメータを設定します。

Parameters:

- **ctx** wolfSSL_CTX_new()で作成された WOLFSSL_CTX 構造体へのポインタ。
- **groups** 識別子によって鍵交換グループのリストをグループ化します。
- **count** グループ内の鍵交換グループの数を数えます。

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 ポインタパラメータが NULL の場合、グループ数は wolfssl_max_group_count を超えているか、TLS v1.3 を使用していません。

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.52.2.425 function wolfSSL_set_groups

```
int wolfSSL_set_groups(
    WOLFSSL * ssl,
    int * groups,
    int count
)
```

この関数は、wolfssl を許すために楕円曲線グループのリストを設定します。リストは、Count で指定された識別子の数を持つグループ識別子の配列です。この関数を呼び出して、TLS v1.3 接続で使用する鍵交換楕円曲線パラメータを設定します。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ。
- **groups** 識別子によって鍵交換グループのリストをグループ化します。
- **count** グループ内の鍵交換グループの数を数えます。

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 ポインタパラメータが NULL の場合､グループ数が WolfSSL_MAX_GROUP_COUNT を超えている場合、任意の識別子は認識されないか、TLS v1.3 を使用していません。

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.52.2.426 function wolfSSL_connect_TLSv13

```
int wolfSSL_connect_TLSv13(
    WOLFSSL * ssl
)
```

この関数はクライアント側で呼び出され、サーバーとの TLS v1.3 ハンドシェイクを開始します。この関数が呼び出されると、下層の通信チャネルはすでに設定されています。wolfSSL_connect()は、ブロックとノンブロック I/O の両方で動作します。下層 I/O がノンブロッキングの場合、wolfSSL_connect() は、下層 I/O が wolfssl_connect の要求を満たすことができなかったときに戻ります。この場合、wolfSSL_get_error() への呼び出しは SSL_ERROR_WANT_READ または SSL_ERROR_WANT_WRITE のいずれかを生成します。通話プロセスは、下層 I/O が READY および WOLFSSL が停止したときに wolfssl_connect() への呼び出しを繰り返す必要があります。ノンブロッキングソケットを使用する場合は、何も実行する必要がありますが、select() を使用して必要な条件を確認できます。基礎となる入出力がブロックされている場合、wolfssl_connect() はハンドシェイクが終了したら、またはエラーが発生したらのみ戻ります。WolfSSL は OpenSSL よりも証明書検証に異なるアプローチを取ります。クライアントのデフォルトポリシーはサーバーを確認することです。これは、CAS を読み込まない場合、サーバーを確認することができ、確認できません（_155）。SSL_CONNECT を持つことの OpenSSL の動作が成功した場合は、サーバーを検証してセキュリティを抑えることができます。SSL_CTX_SET_VERIFY（CTX、SSL_VERIFY_NONE、0）。ssl_new() を呼び出す前に。お勧めできませんが。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ。

See:

- wolfSSL_get_error
- wolfSSL_connect
- wolfSSL_accept_TLSv13
- wolfSSL_accept

Return:

- SSL_SUCCESS 成功時に返されます。
- SSL_FATAL_ERROR エラーが発生した場合に返されます。より詳細なエラーコードを取得するには、wolfSSL_get_error() を呼び出します。

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.52.2.427 function wolfSSL_accept_TLSv13

```
wolfSSL_accept_TLSv13(
    WOLFSSL * ssl
)
```

この関数はサーバー側で呼び出され、SSL/TLS クライアントが SSL/TLS ハンドシェイクを開始するのを待ちうけます。この関数が呼び出されると、下層の通信チャネルはすでに設定されています。wolfSSL_accept()は、ブロックとノンブロッキング I/O の両方で動作します。下層の入出力がノンブロッキングである場合、wolfSSL_accept() は、下層の I/O が wolfSSL_accept の要求を満たすことができなかったときに戻ります。この場合、wolfSSL_get_error() への呼び出しは SSL_ERROR_WANT_READ または SSL_ERROR_WANT_WRITE のいずれかを生成します。通話プロセスは、読み取り可能なデータが使用可能であり、wolfssl が停止した場所を拾うときに、wolfssl_accept の呼び出しを繰り返す必要があります。ノンブロッキングソケットを使用する場合は、何も実行する必要がありますが、select() を使用して必要な条件を確認できます。下層の I/O がブロックされている場合、wolfssl_accept() はハンドシェイクが終了したら、またはエラーが発生したら戻ります。古いバージョンの ClientHello メッセージがサポートされていますが、TLS v1.3 接続を期待するときにこの関数を呼び出します。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ。

See:

- wolfSSL_get_error
- wolfSSL_connect_TLSv13
- wolfSSL_connect
- wolfSSL_accept_TLSv13
- wolfSSL_accept

Return:

- SSL_SUCCESS 成功時に返されます。
- SSL_FATAL_ERROR エラーが発生した場合に返されます。より詳細なエラーコードを取得するには、wolfSSL_get_error() を呼び出します。

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.52.2.428 function wolfSSL_CTX_set_max_early_data

```
int wolfSSL_CTX_set_max_early_data(
    WOLFSSL_CTX * ctx,
    unsigned int sz
)
```

この関数は、WolfSSL コンテキストを使用して TLS V1.3 サーバーによって受け入れられるアーリーデータの最大量を設定します。この関数を呼び出して、再生攻撃を軽減するためのプロセスへのアーリーデータの量を制限します。初期のデータは、セッションチケットが送信されたこと、したがってセッションチケット

が再開されるたびに同じ接続の鍵から派生した鍵によって保護されます。値は再開のためにセッションチケットに含まれています。ゼロの値は、セッションチケットを使用してクライアントによってアーリーデータを送信することを示します。アーリーデータバイト数をアプリケーションで実際には可能な限り低く保つことをお勧めします。

Parameters:

- **ctx** wolfSSL_CTX_new()で作成された WOLFSSL_CTX 構造体へのポインタ。
- **sz** バイト単位で受け入れるアーリーデータのサイズ。

See:

- wolfSSL_set_max_early_data
- wolfSSL_write_early_data
- wolfSSL_read_early_data

Return:

- BAD_FUNC_ARG CTX が NULL の場合、または TLS v1.3 を使用していない場合。
- SIDE_ERROR クライアントで呼び出された場合。

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.52.2.429 function wolfSSL_set_max_early_data

```
int wolfSSL_set_max_early_data(
    WOLFSSL * ssl,
    unsigned int sz
)
```

この関数は、WolfSSL コンテキストを使用して TLS V1.3 サーバーによって受け入れられるアーリーデータの最大量を設定します。この関数を呼び出して、再生攻撃を軽減するためプロセスへのアーリーデータの量を制限します。初期のデータは、セッションチケットが送信されたこと、したがってセッションチケットが再開されるたびに同じ接続の鍵から派生した鍵によって保護されます。値は再開のためにセッションチケットに含まれています。ゼロの値は、セッションチケットを使用してクライアントによってアーリーデータを送信することを示します。アーリーデータバイト数をアプリケーションで実際には可能な限り低く保つことをお勧めします。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ。
- **SZ クライアントからバイト単位で受け入れるアーリーデータのサイズ。**

See:

- wolfSSL_CTX_set_max_early_data
- wolfSSL_write_early_data
- wolfSSL_read_early_data

Return:

- BAD_FUNC_ARG ssl が NULL の場合、または TLS v1.3 を使用していない場合。
- SIDE_ERROR クライアントで呼び出された場合。

Example

```
int ret;
WOLFSSL* ssl;
...
ret = wolfSSL_set_max_early_data(ssl, 128);
if (ret != WOLFSSL_SUCCESS) {
    // failed to set group list
}
```

C.52.2.430 function wolfSSL_write_early_data

```
int wolfSSL_write_early_data(
    WOLFSSL * ssl,
    const void * data,
    int sz,
    int * outSz
)
```

この関数は、セッション再開時にサーバーにアーリーデータを書き込みます。wolfSSL_connect()またはwolfSSL_connect_tlsv13() の代わりにこの関数を呼び出して、サーバーに接続してハンドシェイクにデータを送ります。この機能はクライアントでのみ使用されます。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ。
- **data** アーリーデータを保持しているバッファへのポインタ。
- **sz** 書き込むアーリーデータのサイズ
- **outSz** 書き込んだアーリーデータのサイズ

See:

- wolfSSL_read_early_data
- wolfSSL_connect
- wolfSSL_connect_TLSv13

Return:

- BAD_FUNC_ARG ポインタパラメータが NULL の場合に返されます。sz は 0 未満または TLSV1.3 を使用しない場合にも返されます。
- SIDE_ERROR サーバーで呼び出された場合に返されます。
- WOLFSSL_FATAL_ERROR 接続が行われていない場合に返されます。

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.52.2.431 function wolfSSL_read_early_data

```
int wolfSSL_read_early_data(
    WOLFSSL * ssl,
    void * data,
    int sz,
    int * outSz
)
```

この関数は、再開時にクライアントからの早期データを読み取ります。wolfssl_accept() または wolfssl_accept_tlsv13() の代わりにこの関数を呼び出して、クライアントを受け入れ、ハンドシェイク内の早期データを読み取ります。ハンドシェイクよりも早期データがない場合は、通常として処理されます。この機能はサーバーでのみ使用されます。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ。
- **データはクライアントから読み込まれた早期データを保持するためのバッファ。**
- **バッファの SZ サイズバイト数。**
- **OUTSZ 初期データのバイト数。**

See:

- wolfSSL_write_early_data
- wolfSSL_accept
- wolfSSL_accept_TLSv13

Return:

- BAD_FUNC_ARG ポインタパラメータが NULL の場合、SZ は 0 未満または TLSV1.3 を使用しない。
- SIDE_ERROR クライアントで呼び出された場合。
- WOLFSSL_FATAL_ERROR 接続を受け入れると失敗した場合

Example

```
int ret = 0;
int err = 0;
WOLFSSL* ssl;
byte earlyData[128];
int outSz;
char buffer[80];
...

ret = wolfSSL_read_early_data(ssl, earlyData, sizeof(earlyData), &outSz);
if (ret != SSL_SUCCESS) {
    err = wolfSSL_get_error(ssl, ret);
    printf("error = %d, %s\n", err, wolfSSL_ERR_error_string(err, buffer));
}
if (outSz > 0) {
    // early data available
```

```
}
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.52.2.432 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
)
```

この関数は、TLS v1.3 接続のプレシェア鍵（PSK）クライアント側コールバックを設定します。コールバックは PSK アイデンティティを見つけ、そのキーと、ハンドシェイクに使用する暗号の名前を返します。この関数は、WOLFSSL_CTX 構造体の client_psk_tls13_cb メンバーを設定します。

Parameters:

- **ctx** wolfSSL_CTX_new()で作成された WOLFSSL_CTX 構造体へのポインタ。

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.52.2.433 function wolfSSL_set_psk_client_tls13_callback

```
void wolfSSL_set_psk_client_tls13_callback(
    WOLFSSL * ssl,
    wc_psk_client_tls13_callback cb
)
```

この関数は、TLS v1.3 接続のプレシェアキー（PSK）クライアント側コールバックを設定します。コールバックは PSK アイデンティティを見つけ、そのキーと、ハンドシェイクに使用する暗号の名前を返します。この関数は、wolfssl 構造体の Options フィールドの client_psk_tls13_cb メンバーを設定します。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ。

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.52.2.434 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
)
```

この関数は、TLS v1.3 接続用の事前共有鍵（PSK）サーバ側コールバックを設定します。コールバックは PSK アイデンティティを見つけ、そのキーと、ハンドシェイクに使用する暗号の名前を返します。この関数は、wolfssl_ctx 構造体の server_psk_tls13_cb メンバーを設定します。

Parameters:

- **ctx** wolfSSL_CTX_new()で作成された WOLFSSL_CTX 構造体へのポインタ。

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.52.2.435 function wolfSSL_set_psk_server_tls13_callback

```
void wolfSSL_set_psk_server_tls13_callback(
    WOLFSSL * ssl,
    wc_psk_server_tls13_callback cb
)
```

この関数は、TLS v1.3 接続用の事前共有鍵（PSK）サーバ側コールバックを設定します。コールバックは PSK アイデンティティを見つけ、そのキーと、ハンドシェイクに使用する暗号の名前を返します。この関数は、wolfssl 構造体のオプションフィールドの server_psk_tls13_cb メンバーを設定します。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ。

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.52.2.436 function wolfSSL_UseKeyShare

```
int wolfSSL_UseKeyShare(
    WOLFSSL * ssl,
    word16 group
)
```

この関数は、キーペアの生成を含むグループからキーシェアエントリを作成します。Keyshare エクステンションには、鍵交換のための生成されたすべての公開鍵が含まれています。この関数が呼び出されると、指定されたグループのみが含まれます。優先グループがサーバーに対して以前に確立されているときにこの関数を呼び出します。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ。
- **キー交換グループ識別子をグループ化します。**

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 ssl が NULL の場合に返されます。
- MEMORY_E 動的メモリ割り当てに失敗すると返されます。

Example

```
int ret;
WOLFSSL* ssl;
...
ret = wolfSSL_UseKeyShare(ssl, WOLFSSL_ECC_X25519);
if (ret != WOLFSSL_SUCCESS) {
    // failed to set key share
}
```

C.52.2.437 function wolfSSL_NoKeyShares

```
int wolfSSL_NoKeyShares(
    WOLFSSL * ssl
)
```

この関数は、ClientHello で鍵共有が送信されないように呼び出されます。これにより、ハンドシェイクに鍵交換が必要な場合は、サーバーが HelloretryRequest で応答するように強制します。予想される鍵交換グループが知られておらず、キーの生成を不必要に回避するときにこの機能を呼び出します。鍵交換が必要なときにハンドシェイクを完了するために追加の往復が必要になることに注意してください。

Parameters:

- **ssl** wolfSSL_new()を使用して作成された WOLFSSL 構造体へのポインタ。

See: wolfSSL_UseKeyShare

Return:

- BAD_FUNC_ARG ssl が NULL の場合に返されます。
- SIDE_ERROR サーバーで呼び出された場合。

Example

```
int ret;
WOLFSSL* ssl;
...
ret = wolfSSL_NoKeyShares(ssl);
```

```
if (ret != WOLFSSL_SUCCESS) {
    // failed to set no key shares
}
```

C.52.2.438 function wolfTLSv1_3_server_method_ex

```
WOLFSSL_METHOD * wolfTLSv1_3_server_method_ex(
    void * heap
)
```

この関数は、アプリケーションがサーバーであることを示すために使用され、TLS 1.3 プロトコルのみをサポートします。この関数は、wolfSSL_CTX_new() を使用して SSL / TLS コンテキストを作成するときに使用される新しい Wolfssl_method 構造体のメモリを割り当てて初期化します。

Parameters:

- **ヒープ静的メモリ割り当て中に静的メモリ割り当て器が使用するバッファへのポインタを使用します。**

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: 新しく作成された wWOLFSSL_METHOS 構造体へのポインタを返します。

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.52.2.439 function wolfTLSv1_3_client_method_ex

```
WOLFSSL_METHOD * wolfTLSv1_3_client_method_ex(
    void * heap
)
```

この関数は、アプリケーションがクライアントであることを示すために使用され、TLS 1.3 プロトコルのみをサポートします。この関数は、wolfSSL_CTX_new() を使用して SSL / TLS コンテキストを作成するときに使用される新しい Wolfssl_method 構造体のメモリを割り当てて初期化します。

Parameters:

- **ヒープ静的メモリ割り当て中に静的メモリ割り当て器が使用するバッファへのポインタを使用します。**

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: 新しく作成された wWOLFSSL_METHOS 構造体へのポインタを返します。

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.52.2.440 function wolfTLSv1_3_server_method

```
WOLFSSL_METHOD * wolfTLSv1_3_server_method(
    void
)
```

この関数は、アプリケーションがサーバーであることを示すために使用され、TLS 1.3 プロトコルのみをサポートします。この関数は、wolfSSL_CTX_new() を使用して SSL / TLS コンテキストを作成するときに使用される新しい Wolfssl_method 構造体のメモリを割り当てて初期化します。

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: 新しく作成された wWOLFSSL_METHOS 構造体へのポインタを返します。

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.52.2.441 function wolfTLSv1_3_client_method

```
WOLFSSL_METHOD * wolfTLSv1_3_client_method(
    void
)
```

この関数は、アプリケーションがクライアントであることを示すために使用され、TLS 1.3 プロトコルのみをサポートします。この関数は、wolfSSL_CTX_new() を使用して SSL / TLS コンテキストを作成するときに使用される新しい Wolfssl_method 構造体のメモリを割り当てて初期化します。

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: 新しく作成された wWOLFSSL_METHOS 構造体へのポインタを返します。

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.52.2.442 function wolfTLSv1_3_method_ex

```
WOLFSSL_METHOD * wolfTLSv1_3_method_ex(
    void * heap
)
```

この関数は、まだどちらの側(サーバ/クライアント)を決定していないことを除いて、Wolftlsv1_3_client_method と同様の wolfssl_method を返します。

Parameters:

- **ヒープ静的メモリ割り当て中に静的メモリ割り当て器が使用するバッファへのポインタを使用します。**

See:

- wolfSSL_new
- wolfSSL_free

Return: WOLFSSL_METHOD 成功した作成では、wolfssl_method ポインタを返します

Example

```
WOLFSSL* ctx;
ctx  = wolfSSL_CTX_new(wolfTLSv1_3_method_ex(NULL));
// check ret value
```

C.52.2.443 function wolfTLSv1_3_method

```
WOLFSSL_METHOD * wolfTLSv1_3_method(
    void
)
```

この関数は、まだどちらの側(サーバ/クライアント)を決定していないことを除いて、Wolftlsv1_3_client_methodと同様の wolfssl_method を返します。

See:

- wolfSSL_new
- wolfSSL_free

Return: WOLFSSL_METHOD 成功した作成では、wolfssl_method ポインタを返します

Example

```
WOLFSSL* ctx;
ctx  = wolfSSL_CTX_new(wolfTLSv1_3_method());
// check ret value
```

C.52.2.444 function wolfSSL_CTX_set_client_cert_type

```
int wolfSSL_CTX_set_client_cert_type(
    WOLFSSL_CTX * ctx,
    const char * buf,
    int len
)
```

この関数はクライアント側で呼び出される場合には、サーバー側に Certificate メッセージで送信できる証明書タイプを設定します。サーバー側で呼び出される場合には、受入れ可能なクライアント証明書タイプを設定します。Raw Public Key 証明書を送受信したい場合にはこの関数を使って証明書タイプを設定しなければなりません。設定する証明書タイプは優先度順に格納したバイト配列として渡します。設定するバッファアドレスに NULL を渡すか、あるいはバッファサイズに 0 を渡すと規定値にもどすことができます。規定値は X509 証明書（WOLFSSL_CERT_TYPE_X509）のみを扱う設定となっています。

Parameters:

- **ctx** wolfssl_ctx コンテキストポインタ
- **ctype** 証明書タイプを格納したバッファへのポインタ
- **len** 証明書タイプを格納したバッファのサイズ（バイト数）*Example*

```
int ret;
WOLFSSL_CTX* ctx;
char ctype[] = {WOLFSSL_CERT_TYPE_RPK, WOLFSSL_CERT_TYPE_X509};
```

```
int len = sizeof(ctype)/sizeof(byte);
...

ret = wolfSSL_CTX_set_client_cert_type(ctx, ctype, 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 成功
- BAD_FUNC_ARG ctx として NULL を渡した、あるいは不正な証明書タイプを指定した、あるいは MAX_CLIENT_CERT_TYPE_CNT 以上のバッファサイズを指定した、あるいは指定の証明書タイプに重複がある

C.52.2.445 function wolfSSL_CTX_set_server_cert_type

```
int wolfSSL_CTX_set_server_cert_type(
    WOLFSSL_CTX * ctx,
    const char * buf,
    int len
)
```

この関数はサーバー側で呼び出される場合には、クライアント側に Certificate メッセージで送信できる証明書タイプを設定します。クライアント側で呼び出される場合には、受入れ可能なサーバー証明書タイプを設定します。Raw Public Key 証明書を送受信したい場合にはこの関数を使って証明書タイプを設定しなければなりません。設定する証明書タイプは優先度順に格納したバイト配列として渡します。設定するバッファアドレスに NULL を渡すか、あるいはバッファサイズに 0 を渡すと規定値にもどすことができます。規定値は X509 証明書（WOLFSSL_CERT_TYPE_X509）のみを扱う設定となっています。

Parameters:

- **ctx** wolfssl_ctx コンテキストポインタ
- **ctype** 証明書タイプを格納したバッファへのポインタ
- **len** 証明書タイプを格納したバッファのサイズ（バイト数）*Example*

```
int ret;
WOLFSSL_CTX* ctx;
char ctype[] = {WOLFSSL_CERT_TYPE_RPK, WOLFSSL_CERT_TYPE_X509};
int len = sizeof(ctype)/sizeof(byte);
...

ret = wolfSSL_CTX_set_server_cert_type(ctx, ctype, 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 成功

- BAD_FUNC_ARG ctx として NULL を渡した、あるいは不正な証明書タイプを指定した、あるいは MAX_SERVER_CERT_TYPE_CNT 以上のバッファサイズを指定した、あるいは指定の証明書タイプに重複がある

C.52.2.446 function wolfSSL_set_client_cert_type

```
int wolfSSL_set_client_cert_type(
    WOLFSSL * ssl,
    const char * buf,
    int len
)
```

この関数はクライアント側で呼び出される場合には、サーバー側に Certificate メッセージで送信できる証明書タイプを設定します。サーバー側で呼び出される場合には、受入れ可能なクライアント証明書タイプを設定します。Raw Public Key 証明書を送受信したい場合にはこの関数を使って証明書タイプを設定しなければなりません。設定する証明書タイプは優先度順に格納したバイト配列として渡します。設定するバッファアドレスに NULL を渡すか、あるいはバッファサイズに 0 を渡すと規定値にもどすことができます。規定値は X509 証明書（WOLFSSL_CERT_TYPE_X509）のみを扱う設定となっています。

Parameters:

- **ssl** WOLFSSL 構造体へのポインタ
- **ctype** 証明書タイプを格納したバッファへのポインタ
- **len** 証明書タイプを格納したバッファのサイズ（バイト数）*Example*

```
int ret;
WOLFSSL* ssl;
char ctype[] = {WOLFSSL_CERT_TYPE_RPK, WOLFSSL_CERT_TYPE_X509};
int len = sizeof(ctype)/sizeof(byte);
...

ret = wolfSSL_set_client_cert_type(ssl, ctype, 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 成功
- BAD_FUNC_ARG ssl として NULL を渡した、あるいは不正な証明書タイプを指定した、あるいは MAX_CLIENT_CERT_TYPE_CNT 以上のバッファサイズを指定した、あるいは指定の証明書タイプに重複がある

C.52.2.447 function wolfSSL_set_server_cert_type

```
int wolfSSL_set_server_cert_type(
    WOLFSSL * ssl,
    const char * buf,
    int len
)
```

この関数はサーバー側で呼び出される場合には、クライアント側に Certificate メッセージで送信できる証明書タイプを設定します。クライアント側で呼び出される場合には、受入れ可能なサーバー証明書タイプを設定します。Raw Public Key 証明書を送受信したい場合にはこの関数を使って証明書タイプを設定しなけ

ればなりません。設定する証明書タイプは優先度順に格納したバイト配列として渡します。設定するバッファアドレスに NULL を渡すか、あるいはバッファサイズに 0 を渡すと規定値にもどすことができます。規定値は X509 証明書（WOLFSSL_CERT_TYPE_X509）のみを扱う設定となっています。

Parameters:

- **ssl** WOLFSSL 構造体へのポインタ
- **ctype** 証明書タイプを格納したバッファへのポインタ
- **len** 証明書タイプを格納したバッファのサイズ（バイト数）*Example*

```
int ret;
WOLFSSL* ssl;
char ctype[] = {WOLFSSL_CERT_TYPE_RPK, WOLFSSL_CERT_TYPE_X509};
int len = sizeof(ctype)/sizeof(byte);
...

ret = wolfSSL_set_server_cert_type(ssl, ctype, 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 成功
- BAD_FUNC_ARG ctx として NULL を渡した、あるいは不正な証明書タイプを指定した、あるいは MAX_SERVER_CERT_TYPE_CNT 以上のバッファサイズを指定した、あるいは指定の証明書タイプに重複がある

C.52.2.448 function wolfSSL_get_negotiated_client_cert_type

```
int wolfSSL_get_negotiated_client_cert_type(
    WOLFSSL * ssl,
    int * tp
)
```

この関数はハンドシェーク終了後に呼び出し、相手とのネゴシエーションの結果得られたクライアント証明書のタイプを返します。ネゴシエーションが発生しない場合には戻り値として WOLFSSL_SUCCESS が返されますが、証明書タイプとしては WOLFSSL_CERT_TYPE_UNKNOWN が返されます。

Parameters:

- **ssl** WOLFSSL 構造体へのポインタ
- **tp** 証明書タイプが返されるバッファへのポインタ *Example*

```
int ret;
WOLFSSL* ssl;
int tp;
...

ret = wolfSSL_get_negotiated_client_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_server_cert_type

Return:

- WOLFSSL_SUCCESS 成功時にかえります。tp に返された証明書タイプは WOLFSSL_CERT_TYPE_X509, WOLFSSL_CERT_TYPE_RPK あるいは WOLFSSL_CERT_TYPE_UNKNOWN のいずれかとなります。
- BAD_FUNC_ARG ssl として NULL を渡した、あるいは tp として NULL を渡した

C.52.2.449 function wolfSSL_get_negotiated_server_cert_type

```
int wolfSSL_get_negotiated_server_cert_type(
    WOLFSSL * ssl,
    int * tp
)
```

この関数はハンドシェーク終了後に呼び出し、相手とのネゴシエーションの結果得られたサーバー証明書のタイプを返します。ネゴシエーションが発生しない場合には戻り値として WOLFSSL_SUCCESS が返されますが、証明書タイプとしては WOLFSSL_CERT_TYPE_UNKNOWN が返されます。

Parameters:

- **ssl** WOLFSSL 構造体へのポインタ
- **tp** 証明書タイプが返されるバッファへのポインタ *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 成功時にかえります。tp に返された証明書タイプは WOLFSSL_CERT_TYPE_X509, WOLFSSL_CERT_TYPE_RPK あるいは WOLFSSL_CERT_TYPE_UNKNOWN のいずれかとなります。
- BAD_FUNC_ARG ssl として NULL を渡した、あるいは tp として NULL を渡した

C.52.2.450 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
)
```

この関数はテストのための固定/静的なエフェラルキーを設定します。

Parameters:

- **ctx** WOLFSSL_CTX コンテキストポインタ

- **keyAlgo** WC_PK_TYPE_DH および WC_PK_TYPE_ECDH のような enum wc_pktype
- **key** キーファイルパス（Keysz == 0）または実際のキーバッファ（PEM または ASN.1）
- **keySz** キーサイズ（「キー」arg はファイルパスの場合は 0 になります）

See: wolfSSL_CTX_get_ephemeral_key

Return: 0 成功時に返されます。

C.52.2.451 function wolfSSL_set_ephemeral_key

```
int wolfSSL_set_ephemeral_key(
    WOLFSSL * ssl,
    int keyAlgo,
    const char * key,
    unsigned int keySz,
    int format
)
```

この関数はテストのための固定/静的なエフェラルキーを設定します。

Parameters:

- **ssl** WOLFSSL 構造体へのポインタ
- **keyAlgo** WC_PK_TYPE_DH および WC_PK_TYPE_ECDH のような enum wc_pktype
- **key** キーファイルパス（Keysz == 0）または実際のキーバッファ（PEM または ASN.1）
- **keySz** キーサイズ（「キー」arg はファイルパスの場合は 0 になります）

See: wolfSSL_get_ephemeral_key

Return: 0 成功時に返されます。

C.52.2.452 function wolfSSL_CTX_get_ephemeral_key

```
int wolfSSL_CTX_get_ephemeral_key(
    WOLFSSL_CTX * ctx,
    int keyAlgo,
    const unsigned char ** key,
    unsigned int * keySz
)
```

この関数は ASN.1/DER としてロードされたキーへのポインタを返します

Parameters:

- **ctx** wolfssl_ctx コンテキストポインタ
- **keyAlgo** WC_PK_TYPE_DH および WC_PK_TYPE_ECDH のような enum wc_pktype
- **key** キーバッファポインタ

See: wolfSSL_CTX_set_ephemeral_key

Return: 0 成功時に返されます。

C.52.2.453 function wolfSSL_get_ephemeral_key

```
int wolfSSL_get_ephemeral_key(
    WOLFSSL * ssl,
    int keyAlgo,
    const unsigned char ** key,
    unsigned int * keySz
)
```

この関数は ASN.1/DER としてロードされた鍵へのポインタを返します

Parameters:

- **ssl** WOLFSSL 構造体へのポインタ
- **keyAlgo** WC_PK_TYPE_DH および WC_PK_TYPE_ECDH のような enum wc_pktype
- **key** キーバッファポインタ

See: wolfSSL_set_ephemeral_key

Return: 0 成功時に返されます。

C.52.2.454 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
)
```

選択したメッセージダイジェスト、パディング、および RSA キーを使用してメッセージに署名します。

Parameters:

- **type** ハッシュ NID
- **m** 署名するメッセージ。これは署名するメッセージのダイジェスト
- **mLen** 署名するメッセージの長さ
- **sigRet** 出力バッファへのポインタ
- **sigLen** 入力時には sigRet の長さを指定します。出力時には sigRet に書き込まれたデータの長さを格納します。
- **rsa** 入力に署名するために使用される RSA 鍵
- **flag** 1：シグニチャ 0：未パワード署名を比較する値を出力します。注：RSA_PKCS1_PSS_PADDING の場合は、wc_rsapss_checkpadding_ex 関数を使用して * VERIFY * 関数の出力を確認する必要があります。
- **padding** パディング

Return:

- WOLFSSL_SUCCESS 成功時に返されます。
- WOLFSSL_FAILURE エラー発生時に返されます。

C.52.2.455 function wolfSSL_dtls13_has_pending_msg

```
int wolfSSL_dtls13_has_pending_msg(
    WOLFSSL * ssl
)
```

DTLSv1.3 送信済みだがまだ相手からアクノリッジを受けとっていないメッセージがあるか調べます。

Parameters:

- **ssl** WOLFSSL 構造体へのポインタ。

Return: 1 ペンディングのメッセージがある場合に返されます。それ以外は 0 が返されます。

C.52.2.456 function wolfSSL_SESSION_get_max_early_data

```
unsigned int wolfSSL_SESSION_get_max_early_data(
    const WOLFSSL_SESSION * s
)
```

アーリーデータの最大サイズを取得します。

Parameters:

- **s** WOLFSSL_SESSION 構造体へのポインタ
- **s** WOLFSSL_SESSION 構造体へのポインタ

See:

- wolfSSL_set_max_early_data
- wolfSSL_write_early_data
- wolfSSL_read_early_data

Return: アーリーデータの最大サイズ（max_early_data）

C.52.2.457 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.52.2.458 function wolfSSL_dtls_cid_use

```
int wolfSSL_dtls_cid_use(
    WOLFSSL * ssl
)
```

コネクション ID 拡張を有効にします。RFC9146 と RFC9147 を参照してください。

Parameters:

- **ssl** WOLFSSL 構造体へのポインタ。

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 成功時に返されます。それ以外はエラーコードが返されます。

C.52.2.459 function wolfSSL_dtls_cid_is_enabled

```
int wolfSSL_dtls_cid_is_enabled(
    WOLFSSL * ssl
)
```

この関数はハンドシェークが完了した後に呼び出されると、コネクション ID がネゴシエートされたかどうか確認することができます。RFC9146 と RFC9147 を参照してください。

Parameters:

- **ssl** WOLFSSL 構造体へのポインタ。

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 コネクション ID がネゴシエートされた場合に返されます。それ以外は 0 が返されます。

C.52.2.460 function wolfSSL_dtls_cid_set

```
int wolfSSL_dtls_cid_set(
    WOLFSSL * ssl,
    unsigned char * cid,
    unsigned int size
)
```

このコネクションで他のピアに対してレコードを送信するためのコネクション ID をセットします。RFC9146 と RFC9147 を参照してください。コネクション ID は最大値が DTLS_CID_MAX_SIZE でなければなりません。DTLS_CID_MAX_SIZE はビルド時に値を指定が可能ですが 255 バイトをこえることはできません。

Parameters:

- **ssl** WOLFSSL 構造体へのポインタ。
- **cid** コネクション ID
- **size** コネクション ID のサイズ

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 コネクション ID がセットできた場合に返されます。それ以外はエラーコードが返されます。

C.52.2.461 function wolfSSL_dtls_cid_get_rx_size

```
int wolfSSL_dtls_cid_get_rx_size(
    WOLFSSL * ssl,
    unsigned int * size
)
```

コネクション ID のサイズを取得します。RFC9146 と RFC9147 を参照してください。

Parameters:

- **ssl** WOLFSSL 構造体へのポインタ。
- **size** コネクション ID のサイズを格納する int 型変数へのポインタ。

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 コネクション ID が取得できた場合に返されます。それ以外はエラーコードが返されます。

C.52.2.462 function wolfSSL_dtls_cid_get_rx

```
int wolfSSL_dtls_cid_get_rx(
    WOLFSSL * ssl,
    unsigned char * buffer,
    unsigned int bufferSz
)
```

コネクション ID を引数 buffer で指定されたバッファにコピーします。RFC9146 と RFC9147 を参照してください。バッファのサイズは引数 bufferSz で指定してください。

Parameters:

- **ssl** WOLFSSL 構造体へのポインタ。
- **buffer** コネクション ID がコピーされる先のバッファへのポインタ。
- **bufferSz** バッファのサイズ

See:

- 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 コネクション ID が取得できた場合に返されます。それ以外はエラーコードが返されます。

C.52.2.463 function wolfSSL_dtls_cid_get_tx_size

```
int wolfSSL_dtls_cid_get_tx_size(
    WOLFSSL * ssl,
    unsigned int * size
)
```

コネクション ID のサイズを取得します。 c サイズは引数 size 変数に格納されます。

Parameters:

- **ssl** WOLFSSL 構造体へのポインタ。
- **size** コネクション ID のサイズを格納する int 型変数へのポインタ。

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 コネクション ID のサイズが取得できた場合に返されます。それ以外はエラーコードが返されます。

C.52.2.464 function wolfSSL_dtls_cid_get_tx

```
int wolfSSL_dtls_cid_get_tx(
    WOLFSSL * ssl,
    unsigned char * buffer,
    unsigned int bufferSz
)
```

コネクション ID を引き数 buffer で指定されるバッファにコピーします。RFC9146 と RFC9147 を参照してください。バッファのサイズは引き数 bufferSz で指定します。

Parameters:

- **ssl** WOLFSSL 構造体へのポインタ。
- **buffer** ConnectionID がコピーされるバッファへのポインタ。
- **bufferSz** バッファのサイズ

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_size

Return: WOLFSSL_SUCCESS ConnectionID が正常にコピーされた際に返されます。それ以外はエラーコードが返されます。

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_2_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*);

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*);

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 index, 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_get_peer(WOLFSSL* ssl, 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* ssl);

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_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_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_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* x509);

WOLFSSL_ASN1_TIME* wolfSSL_X509_get_notBefore(WOLFSSL_X509* 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* ssl);

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);

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_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_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_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_get_tx_size(WOLFSSL* ssl, unsigned int* size);
```

```
int wolfSSL_dtls_cid_get_tx(WOLFSSL* ssl, unsigned char* buffer,
    unsigned int bufferSz);
```

C.53 dox_comments/header_files-ja/tfm.h

C.53.1 Functions

	Name
word32	**CheckRunTimeFastMath**(void) この関数は、整数の最大サイズのランタイム FastMath 設定をチェックします。FP_SIZE が正しく機能するために、FP_SIZE が各ライブラリーに一致しなければならないため、ユーザーが WolfCrypt ライブラリを独立して使用している場合に重要です。このチェックは CheckFastMathSettings () として定義されています。これは、CheckRuntimeFastMath と FP_SIZE を比較するだけで、ミスマッチがある場合は 0 を返します。

C.53.2 Functions Documentation

C.53.2.1 function CheckRunTimeFastMath

```
word32 CheckRunTimeFastMath(
    void
)
```

この関数は、整数の最大サイズのランタイム FastMath 設定をチェックします。FP_SIZE が正しく機能するために、FP_SIZE が各ライブラリーに一致しなければならないため、ユーザーが WolfCrypt ライブラリを独立して使用している場合に重要です。このチェックは CheckFastMathSettings () として定義されています。これは、CheckRuntimeFastMath と FP_SIZE を比較するだけで、ミスマッチがある場合は 0 を返します。

See: CheckRunTimeSettings

Return: FP_SIZE 数学ライブラリで利用可能な最大サイズに対応する FP_SIZE を返します。*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-ja/types.h

C.54.1 Functions

	Name
void *	**XMALLOC**(size_t n, void * heap, int type) これは実際には関数ではなく、むしろプリプロセッサマクロであり、ユーザーは自分の Malloc、Realloc、および標準の C メモリ関数の代わりに自由な関数に置き換えることができます。外部メモリ機能を使用するには、xmalloc_user を定義します。これにより、メモリ機能をフォームの外部関数に置き換えます.extern void * xmalloc（size_t n、void * heap、int 型）; extern void * XrealLoc（void * p、size_t n、void ヒープ、*int* 型）。*extern void xfree*（*void* p、void * heap、int 型）; wolfssl_malloc、wolfssl_realloc、wolfssl_free の代わりに基本的な C メモリ機能を使用するには、NO_WOLFSSL_MEMORY を定義します。これにより、メモリ関数が次のものに置き換えられます。::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)）これらのオプションのどれも選択されていない場合、システムはデフォルトで使用されます。WolfSSL メモリ機能ユーザーはコールバックフックを介してカスタムメモリ機能を設定できます (Wolfssl_Malloc、WolfSSL_Realloc、wolfssl_free を参照)。このオプションは、メモリ関数を次のものに置き換えます。::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) これは実際には関数ではなく、むしろプリプロセッサマクロであり、ユーザーは自分の Malloc、Realloc、および標準の C メモリ関数の代わりに自由な関数に置き換えることができます。外部メモリ機能を使用するには、xmalloc_user を定義します。これにより、メモリ機能をフォームの外部関数に置き換えます.extern void * xmalloc（size_t n、void * heap、int 型）; extern void * XrealLoc（void * p、size_t n、void ヒープ、*int* 型）。*extern void xfree*（*void* p、void * heap、int 型）; wolfssl_malloc、wolfssl_realloc、wolfssl_free の代わりに基本的な C メモリ機能を使用するには、NO_WOLFSSL_MEMORY を定義します。これにより、メモリ関数が次のものに置き換えられます。::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)）これらのオプションのどれも選択されていない場合、システムはデフォルトで使用されます。WolfSSL メモリ機能ユーザーはコールバックフックを介してカスタムメモリ機能を設定できます（Wolfssl_Malloc、WolfSSL_Realloc、wolfssl_free を参照）。このオプションは、メモリ関数を次のものに置き換えます。::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) これは実際には関数ではなく、むしろプリプロセッサマクロであり、ユーザーは自分の Malloc、Realloc、および標準の C メモリ関数の代わりに自由な関数に置き換えることができます。外部メモリ機能を使用するには、xmalloc_user を定義します。これにより、メモリ機能をフォームの外部関数に置き換えます.extern void * xmalloc（size_t n、void * heap、int 型）; extern void * XrealLoc（void * p、size_t n、void ヒープ、*int* 型）。*extern void xfree*（*void* p、void * heap、int 型）; wolfssl_malloc、wolfssl_realloc、wolfssl_free の代わりに基本的な C メモリ機能を使用するには、NO_WOLFSSL_MEMORY を定義します。これにより、メモリ関数が次のものに置き換えられます。::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)）これらのオプションのどれも選択されていない場合、システムはデフォルトで使用されます。WolfSSL メモリ機能ユーザーはコールバックフックを介してカスタムメモリ機能を設定できます (Wolfssl_Malloc、WolfSSL_Realloc、wolfssl_free を参照)。このオプションは、メモリ関数を次のものに置き換えます。::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) この関数はコンパイル時クラスの設定をチェックします。設定が正しく機能するためのライブラリ間のライブラリ間で一致する必要があるため、ユーザーが WolfCrypt ライブラリを独立して使用している場合は重要です。このチェックは CheckCtcSettings () として定義されています。これは、CheckRuntimeSettings と CTC_Settings を比較するだけで、ミスマッチがある場合は 0、または 1 が一致した場合は 1 を返します。

C.54.2 Functions Documentation

C.54.2.1 function XMALLOC

```
void * XMALLOC(
    size_t n,
    void * heap,
    int type
)
```

これは実際には関数ではなく、むしろプリプロセッサマクロであり、ユーザーは自分の Malloc、Realloc、および標準の C メモリ関数の代わりに自由な関数に置き換えることができます。外部メモリ機能を使用する

には、xmalloc_user を定義します。これにより、メモリ機能をフォームの外部関数に置き換えます.extern void * xmalloc（size_t n、void * heap、int 型）; extern void * XrealLoc（void * p、size_t n、void ヒープ、*int* 型）。*extern void xfree*（*void* p、void * heap、int 型）; wolfssl_malloc、wolfssl_realloc、wolfssl_free の代わりに基本的な C メモリ機能を使用するには、NO_WOLFSSL_MEMORY を定義します。これにより、メモリ関数が次のものに置き換えられます。::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））これらのオプションのどれも選択されていない場合、システムはデフォルトで使用されます。WolfSSL メモリ機能ユーザーはコールバックフックを介してカスタムメモリ機能を設定できます（Wolfssl_Malloc、WolfSSL_Realloc、wolfssl_free を参照）。このオプションは、メモリ関数を次のものに置き換えます。::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** 割り当てるメモリのサイズ
- **h** （カスタム XMalloc 関数で使用されています）使用するヒープへのポインタ *Example*

```
int* tenInts = XMALLOC(sizeof(int)*10, NULL, DYNAMIC_TYPE_TMP_BUFFER);
if (tenInts == NULL) {
    // error allocating space
    return MEMORY_E;
}
```

See:

- wolfSSL_Malloc
- wolfSSL_Realloc
- wolfSSL_Free
- wolfSSL_SetAllocators

Return:

- pointer 成功したメモリへのポインタを返します
- NULL 失敗した

C.54.2.2 function XREALLOC

```
void * XREALLOC(
    void * p,
    size_t n,
    void * heap,
    int type
)
```

これは実際には関数ではなく、むしろプリプロセッサマクロであり、ユーザーは自分の Malloc、Realloc、および標準の C メモリ関数の代わりに自由な関数に置き換えることができます。外部メモリ機能を使用するには、xmalloc_user を定義します。これにより、メモリ機能をフォームの外部関数に置き換えます.extern void * xmalloc（size_t n、void * heap、int 型）; extern void * XrealLoc（void * p、size_t n、void ヒープ、*int* 型）。*extern void xfree*（*void* p、void * heap、int 型）; wolfssl_malloc、wolfssl_realloc、wolfssl_free の代わりに基本的な C メモリ機能を使用するには、NO_WOLFSSL_MEMORY を定義します。これにより、メモリ関数が次のものに置き換えられます。::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））これらのオプションのどれも選択されていない場合、システムはデフォルトで使用されます。WolfSSL メモリ機能ユーザーはコールバックフックを介してカスタムメモリ機能を設定できます（Wolfssl_Malloc、WolfSSL_Realloc、wolfssl_free を参照）。このオプションは、メモリ関数を次のものに置き換えます。::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** Reallocate へのアドレスへのポインタ
- **n** 割り当てるメモリのサイズ
- **h** （カスタム Xrealloc 関数で使用されています）使用するヒープへのポインタ *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);
```

See:

- wolfSSL_Malloc
- wolfSSL_Realloc
- wolfSSL_Free
- wolfSSL_SetAllocators

Return:

- Return 成功したメモリを割り当てるポインタ
- NULL 失敗した

C.54.2.3 function XFREE

```
void XFREE(
    void * p,
    void * heap,
    int type
)
```

これは実際には関数ではなく、むしろプリプロセッサマクロであり、ユーザーは自分の Malloc、Realloc、および標準の C メモリ関数の代わりに自由な関数に置き換えることができます。外部メモリ機能を使用するには、xmalloc_user を定義します。これにより、メモリ機能をフォームの外部関数に置き換えます.extern void * xmalloc（size_t n、void * heap、int 型）; extern void * XrealLoc（void * p、size_t n、void ヒープ、*int* 型）。*extern void xfree*（*void* p、void * heap、int 型）; wolfssl_malloc、wolfssl_realloc、wolfssl_free の代わりに基本的な C メモリ機能を使用するには、NO_WOLFSSL_MEMORY を定義します。これにより、メモリ関数が次のものに置き換えられます。::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)）これらのオプションのどれも選択されていない場合、システムはデフォルトで使用されます。WolfSSL メモリ機能ユーザーはコールバックフックを介してカスタムメモリ機能を設定できます(Wolfssl_Malloc、WolfSSL_Realloc、wolfssl_free を参照)。このオプションは、メモリ関数を次のものに置き換えます。::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** 無料のアドレスへのポインタ
- **h** 使用するヒープへの（カスタム XFree 関数で使用されています)。*Example*

```
int* tenInts = XMALLOC(sizeof(int) * 10, NULL, DYNAMIC_TYPE_TMP_BUFFER);
if (tenInts == NULL) {
    // error allocating space
    return MEMORY_E;
}
```

See:

- wolfSSL_Malloc
- wolfSSL_Realloc
- wolfSSL_Free
- wolfSSL_SetAllocators

Return: none いいえ返します。

C.54.2.4 function CheckRunTimeSettings

```
word32 CheckRunTimeSettings(
    void
)
```

この関数はコンパイル時クラスの設定をチェックします。設定が正しく機能するためのライブラリ間のライブラリ間で一致する必要があるため、ユーザーが WolfCrypt ライブラリを独立して使用している場合は重要です。このチェックは CheckCtcSettings（）として定義されています。これは、CheckRuntimeSettings と CTC_Settings を比較するだけで、ミスマッチがある場合は 0、または 1 が一致した場合は 1 を返します。

See: CheckRunTimeFastMath

Return: settings 実行時 CTC_SETTINGS（コンパイル時設定）を返します。*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-ja/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) 入力バッファーから暗号を復号化し、AES で Cipher Block Chaining を使用して出力バッファに出力バッファーに入れます。この関数は、AES 構造を初期化する必要はありません。代わりに、キーと IV（初期化ベクトル）を取り、これらを使用して AES オブジェクトを初期化してから暗号テキストを復号化します。

	Name
int	**wc_Des_CbcDecryptWithKey**(byte * out, const byte * in, word32 sz, const byte * key, const byte * iv) この関数は入力暗号文を復号化し、結果の平文を出力バッファーに出力します。暗号ブロックチェーンチェーン（CBC）モードで DES 暗号化を使用します。この関数は、wc_des_cbcdecrypt の代わりに、ユーザーが DES 構造体を直接インスタンス化せずにメッセージを復号化できるようにします。
int	**wc_Des_CbcEncryptWithKey**(byte * out, const byte * in, word32 sz, const byte * key, const byte * iv) この関数は入力平文を暗号化し、結果の暗号文を出力バッファーに出力します。暗号ブロックチェーンチェーン（CBC）モードで DES 暗号化を使用します。この関数は、WC_DES_CBCENCRYPT の代わりになり、ユーザーが DES 構造を直接インスタンス化せずにメッセージを暗号化できます。
int	**wc_Des3_CbcEncryptWithKey**(byte * out, const byte * in, word32 sz, const byte * key, const byte * iv) この関数は入力平文を暗号化し、結果の暗号文を出力バッファーに出力します。暗号ブロックチェーン（CBC）モードでトリプル DES（3DES）暗号化を使用します。この関数は、WC_DES3_CBCENCRYPT の代わりになり、ユーザーが DES3 構造を直接インスタンス化せずにメッセージを暗号化できます。
int	**wc_Des3_CbcDecryptWithKey**(byte * out, const byte * in, word32 sz, const byte * key, const byte * iv) この関数は入力暗号文を復号化し、結果の平文を出力バッファーに出力します。暗号ブロックチェーン（CBC）モードでトリプル DES（3DES）暗号化を使用します。この関数は、wc_des3_cbcdecrypt の代わりに、ユーザーが DES3 構造を直接インスタンス化せずにメッセージを復号化できるようにします。

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
)
```

入力バッファーから暗号を復号化し、AES で Cipher Block Chaining を使用して出力バッファに出力バッファーに入れます。この関数は、AES 構造を初期化する必要はありません。代わりに、キーと IV（初期化ベクトル）を取り、これらを使用して AES オブジェクトを初期化してから暗号テキストを復号化します。

Parameters:

- **out** 復号化されたメッセージのプレーンテキストを保存する出力バッファへのポインタ
- **in** 復号化される暗号テキストを含む入力バッファへのポインタ
- **inSz** 入力メッセージのサイズ
- **key** 復号化のための 16,24、または 32 バイトの秘密鍵 *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
}
```

See:

- wc_AesSetKey
- wc_AesSetIV
- wc_AesCbcEncrypt
- wc_AesCbcDecrypt

Return:

- 0 メッセージの復号化に成功しました
- BAD_ALIGN_E ブロック整列エラーに戻りました
- BAD_FUNC_ARG aesetiv の間にキーの長さが無効な場合、または AES オブジェクトが NULL の場合
- MEMORY_E wolfssl_small_stack が有効になっていて、xmalloc が AES オブジェクトのインスタンス化に失敗した場合に返されます。

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
)
```

この関数は入力暗号文を復号化し、結果の平文を出力バッファーに出力します。暗号ブロックチェーンチェーン（CBC）モードで DES 暗号化を使用します。この関数は、wc_des_cbcdecrypt の代わりに、ユーザーが DES 構造体を直接インスタンス化せずにメッセージを復号化できるようにします。

Parameters:

- **out** 復号化された平文を保存するバッファへのポインタ
- **in** 暗号化された暗号文を含む入力バッファへのポインタ
- **sz** 復号化する暗号文の長さ
- **key** 復号化に使用する 8 バイトのキーを含むバッファへのポインタ *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
}
```

See: wc_Des_CbcDecrypt

Return:

- 0 与えられた暗号文を正常に復号化したときに返されました
- MEMORY_E DES 構造体の割り当てスペースが割り当てられている場合に返された

3

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
)
```

この関数は入力平文を暗号化し、結果の暗号文を出力バッファーに出力します。暗号ブロックチェーンチェーン（CBC）モードで DES 暗号化を使用します。この関数は、WC_DES_CBCENCRYPT の代わりになり、ユーザーが DES 構造を直接インスタンス化せずにメッセージを暗号化できます。

Parameters:

- **out** 最終暗号化データ
- **in** 暗号化されるデータは、DES ブロックサイズに埋められなければなりません。
- **sz** 入力バッファのサイズ
- **key** 暗号化に使用するキーへのポインタ。*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
}
```

See:

- wc_Des_CbcDecryptWithKey
- wc_Des_CbcEncrypt

Return:

- 0 データの暗号化に成功した後に返されます。
- MEMORY_E DES 構造体にメモリを割り当てるエラーがある場合は返されます。
- <0 暗号化中に任意のエラーに戻ります。

3

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
)
```

この関数は入力平文を暗号化し、結果の暗号文を出力バッファーに出力します。暗号ブロックチェーン（CBC）モードでトリプル DES（3DES）暗号化を使用します。この関数は、WC_DES3_CBCENCRYPT の代わりになり、ユーザーが DES3 構造を直接インスタンス化せずにメッセージを暗号化できます。

Parameters:

- **out** 最終暗号化データ
- **in** 暗号化されるデータは、DES ブロックサイズに埋められなければなりません。
- **sz** 入力バッファのサイズ
- **key** 暗号化に使用するキーへのポインタ。*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
}
```

See:

- wc_Des3_CbcDecryptWithKey
- wc_Des_CbcEncryptWithKey
- wc_Des_CbcDecryptWithKey

Return:

- 0 データの暗号化に成功した後に返されます。
- MEMORY_E DES 構造体にメモリを割り当てるエラーがある場合は返されます。
- <0 暗号化中に任意のエラーに戻ります。

3

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
)
```

この関数は入力暗号文を復号化し、結果の平文を出力バッファーに出力します。暗号ブロックチェーン（CBC）モードでトリプル DES（3DES）暗号化を使用します。この関数は、wc_des3_cbcdecrypt の代わりに、ユーザーが DES3 構造を直接インスタンス化せずにメッセージを復号化できるようにします。

Parameters:

- **out** 復号化された平文を保存するバッファへのポインタ
- **in** 暗号化された暗号文を含む入力バッファへのポインタ
- **sz** 復号化する暗号文の長さ
- **key** 復号化に使用する 24 バイトのキーを含むバッファへのポインタ *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
}
```

See: wc_Des3_CbcDecrypt

Return:

- 0 与えられた暗号文を正常に復号化したときに返されました
- MEMORY_E DES 構造体の割り当てスペースが割り当てられている場合に返された

3

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-ja/wc_port.h

C.56.1 Functions

	Name
int	**wolfCrypt_Init**(void)WolfCrypt によって使用されるリソースを初期化するために使用されます。
int	**wolfCrypt_Cleanup**(void)WolfCrypt によって使用されるリソースをクリーンアップするために使用されます。

C.56.2 Functions Documentation

C.56.2.1 function wolfCrypt_Init

```
int wolfCrypt_Init(
    void
)
```

WolfCrypt によって使用されるリソースを初期化するために使用されます。

See: wolfCrypt_Cleanup

Return:

- 0 成功すると。
- <0 init リソースが失敗すると。*Example*

```
...
if (wolfCrypt_Init() != 0) {
    WOLFSSL_MSG("Error with wolfCrypt_Init call");
}
```

C.56.2.2 function wolfCrypt_Cleanup

```
int wolfCrypt_Cleanup(
    void
)
```

WolfCrypt によって使用されるリソースをクリーンアップするために使用されます。

See: wolfCrypt_Init

Return:

- 0 成功すると。
- <0 リソースのクリーンアップが失敗したとき。*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-ja/wolfio.h

C.57.1 Functions

	Name
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) データ。デフォルトでは、WolfSSL はシステムの TCP RECV () 関数を使用するコールバックとして EmbedReceive () を使用します。ユーザは、メモリ、他のネットワークモジュール、またはどこからでも入力するように機能を登録できます。関数の機能とエラーコードのためのガイドとして、src / io.c の埋め込み Receive () 関数を参照してください。特に、データが準備ができていないときに、IO_ERR_WANT_READ を非ブロック受信用に返す必要があります。
void	**wolfSSL_SetIOReadCtx**(WOLFSSL * ssl, void * ctx) コールバック関数デフォルトでは、WolfSSL は、WolfSSL がシステムの TCP ライブラリを使用している場合、wolfssl_set_fd () に渡されたファイル記述子をコンテキストとして設定します。自分の受信コールバックを登録した場合は、セッションの特定のコンテキストを設定することができます。たとえば、メモリバッファを使用している場合、コンテキストは、メモリバッファーのどこにありかを説明する構造へのポインタであり得る。
void	**wolfSSL_SetIOWriteCtx**(WOLFSSL * ssl, void * ctx) コールバック関数デフォルトでは、WolfSSL は、WolfSSL がシステムの TCP ライブラリを使用している場合、wolfssl_set_fd () に渡されたファイル記述子をコンテキストとして設定します。独自の送信コールバックを登録した場合は、セッションの特定のコンテキストを設定することができます。たとえば、メモリバッファを使用している場合、コンテキストは、メモリバッファーのどこにありかを説明する構造へのポインタであり得る。
void *	**wolfSSL_GetIOReadCtx**(WOLFSSL * ssl) この関数は、WolfSSL 構造体の IOCB_READCTX メンバーを返します。
void *	**wolfSSL_GetIOWriteCtx**(WOLFSSL * ssl) この関数は、WolfSSL 構造の IOCB_WRITECTX メンバーを返します。

	Name
void	**wolfSSL_SetIOReadFlags**(WOLFSSL * ssl, int flags) 与えられた SSL セッション受信コールバックは、デフォルトの wolfssl 埋め込み受信コールバック、またはユーザによって指定されたカスタムコールバックであり得る (wolfssl_ctx_setiorecv を参照)。デフォルトのフラグ値は、WolfSSL によって wolfssl によって 0 の値に設定されます。デフォルトの WolfSSL 受信コールバックは RECV () 関数を使用してソケットからデータを受信します。「Recv ()」ページから：「Recv () 関数への flags 引数は、1 つ以上の値を OR 処理するか、MSG_OOB プロセス帯域外データ、MSG_PEEK PEEK、MSG_PEEK PEEK、MSG_WAITALL がフルを待っています要求またはエラー。MSG_OOB フラグは、通常のデータストリームで受信されないであろう帯域外データの受信を要求します。一部のプロトコルは通常のデータキューの先頭に迅速なデータを配置し、このフラグをそのようなプロトコルで使用することはできません。MSG_PEEK フラグは、受信操作によって受信キューの先頭からのデータをキューから削除することなくデータを返します。したがって、以降の受信呼び出しは同じデータを返します。MSG_WAITALL フラグは、完全な要求が満たされるまで操作ブロックを要求します。ただし、信号がキャッチされている場合は、呼び出し側よりも少ないデータが少なく、エラーまたは切断が発生するか、または受信されるデータが返されるものとは異なるタイプのデータを返します。
void	**wolfSSL_SetIOWriteFlags**(WOLFSSL * ssl, int flags)SSL セッションを考えると送信コールバックは、デフォルトの WolfSSL EmbedEnd コールバック、またはユーザーによって指定されたカスタムコールバックのいずれかです (WolfSSL_CTX_SetiosEnd を参照)。デフォルトのフラグ値は、wolfssl によって 0 の値に設定されます。デフォルトの WolfSSL Send Callback は send () 関数を使用してソケットからデータを送信します。send () man ページから：”flags パラメータには、次のうち 1 つ以上が含まれていてもよい。フラグ MSG_OOB は、この概念（例えば SOCK_STREAM）をサポートするソケットに「帯域外」データを送信するために使用される。基礎となるプロトコルは、「帯域外」のデータもサポートする必要があります。MSG_DONTROUTE は通常、診断プログラムまたはルーティングプログラムによってのみ使用されます。」
void	**wolfSSL_SetIO_NetX**(WOLFSSL * ssl, NX_TCP_SOCKET * nxsocket, ULONG waitoption) この関数は、wolfssl 構造内の nxctx 構造体の NxSocket メンバーと NXWAIT メンバーを設定します。

	Name
void	**wolfSSL_CTX_SetGenCookie**(WOLFSSL_CTX * ctx, CallbackGenCookie cb)wolfssl_ctx 構造 CallBackGencookie Type は関数ポインタで、署名：int（* callbackgencookie）（wolfssl * ssl、unsigned char * buf、int sz、void * ctx）を持っています。
void *	**wolfSSL_GetCookieCtx**(WOLFSSL * ssl) この関数は、WolfSSL 構造の IOCB_COOKIECTX メンバーを返します。
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) この関数は、WolfSSL が WolfSSL_ISOTP でコンパイルされている場合に使用する場合は、WolfSSL の場合は ISO-TP コンテキストを設定します。

C.57.2 Functions Documentation

C.57.2.1 function EmbedReceive

```
int EmbedReceive(
    WOLFSSL * ssl,
    char * buf,
    int sz,
    void * ctx
)
```

Parameters:

- **ssl** wolfssl_new（）を使用して作成された WolfSSL 構造へのポインタ。
- **buf** バッファのチャーポインタ表現。
- **sz** バッファのサイズ。*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.
}
```

See:

- EmbedSend
- wolfSSL_CTX_SetIORecv
- wolfSSL_SSLSetIORecv

Return:

- Success この関数は、読み取られたバイト数を返します。
- WOLFSSL_CBIO_ERR_WANT_READ 最後のエラーが socket_ewouldbolcok または socket_eagain であれば、メッセージを返されます。

- WOLFSSL_CBIO_ERR_TIMEOUT “Socket Timeout” メッセージを返しました。
- WOLFSSL_CBIO_ERR_CONN_RST 最後のエラーが socket_econnreset の場合、“Connection Reset” メッセージで返されます。
- WOLFSSL_CBIO_ERR_ISR 最後のエラーが socket_eintr の場合、“Socket Interrupted” メッセージが返されます。
- WOLFSSL_CBIO_ERR_WANT_READ 最後のエラーが socket_econneRefused の場合、「接続拒否」メッセージを返しました。
- WOLFSSL_CBIO_ERR_CONN_CLOSE 最後のエラーが SOCKET_ECONNABORTED の場合、「接続中止」メッセージで返されます。
- WOLFSSL_CBIO_ERR_GENERAL 最後のエラーが指定されていない場合は、「一般的なエラー」メッセージで返されます。

C.57.2.2 function EmbedSend

```
int EmbedSend(
    WOLFSSL * ssl,
    char * buf,
    int sz,
    void * ctx
)
```

Parameters:

- **ssl** wolfssl_new （） を使用して作成された WolfSSL 構造へのポインタ。
- **buf** バッファを表す文字ポインタ。
- **sz** バッファのサイズ。*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.
}
```

See:

- EmbedReceive
- wolfSSL_CTX_SetIOSend
- wolfSSL_SSLSetIOSend

Return:

- Success この関数は送信されたバイト数を返します。
- WOLFSSL_CBIO_ERR_WANT_WRITE 最後のエラーが socket_ewouldblock または socket_eagain であれば、“Block” メッセージを返します。
- WOLFSSL_CBIO_ERR_CONN_RST 最後のエラーが socket_econnreset の場合、“Connection Reset” メッセージで返されます。
- WOLFSSL_CBIO_ERR_ISR 最後のエラーが socket_eintr の場合、“Socket Interrupted” メッセージが返されます。
- WOLFSSL_CBIO_ERR_CONN_CLOSE 最後のエラーが socket_epipe の場合、“Socket Epipe” メッセージを返しました。
- WOLFSSL_CBIO_ERR_GENERAL 最後のエラーが指定されていない場合は、「一般的なエラー」メッセージで返されます。

C.57.2.3 function EmbedReceiveFrom

```
int EmbedReceiveFrom(
    WOLFSSL * ssl,
    char * buf,
    int sz,
    void *
)
```

Parameters:

- **ssl** wolfssl_new （） を使用して作成された WolfSSL 構造へのポインタ。
- **buf** バッファへの定数の文字ポインタ。
- **sz** バッファのサイズを表す int 型。*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
}
```

See:

- EmbedSendTo
- wolfSSL_CTX_SetIORecv
- wolfSSL_SSLSetIORecv
- wolfSSL_dtls_get_current_timeout

Return:

- Success この関数は、実行が成功した場合に読み込まれた NB バイトを返します。
- WOLFSSL_CBIO_ERR_WANT_READ 接続が拒否された場合、または「ブロック」エラーが発生した場合は機能にスローされました。
- WOLFSSL_CBIO_ERR_TIMEOUT ソケットがタイムアウトした場合は返されます。
- WOLFSSL_CBIO_ERR_CONN_RST 接続がリセットされている場合は返されます。
- WOLFSSL_CBIO_ERR_ISR ソケットが中断された場合は返されます。
- WOLFSSL_CBIO_ERR_GENERAL 一般的なエラーがあった場合に返されます。

C.57.2.4 function EmbedSendTo

```
int EmbedSendTo(
    WOLFSSL * ssl,
    char * buf,
    int sz,
    void * ctx
)
```

Parameters:

- **ssl** wolfssl_new （） を使用して作成された WolfSSL 構造へのポインタ。
- **buf** バッファを表す文字ポインタ。
- **sz** バッファのサイズ。*Example*

```
WOLFSSL* ssl;
…
char* buf;
```

```
int sz;
void* ctx;

int sEmbed = EmbedSendto(ssl, buf, sz, ctx);
if(sEmbed <= 0){
    // No bytes sent. Failure case.
}
```

See:

- EmbedReceiveFrom
- wolfSSL_CTX_SetIOSend
- wolfSSL_SSLSetIOSend

Return:

- Success この関数は送信されたバイト数を返します。
- WOLFSSL_CBIO_ERR_WANT_WRITE 最後のエラーが socket_ewouldblock または socket_eagain エラーの場合、“Block” メッセージを返します。
- WOLFSSL_CBIO_ERR_CONN_RST 最後のエラーが socket_econnreset の場合、“Connection Reset” メッセージで返されます。
- WOLFSSL_CBIO_ERR_ISR 最後のエラーが socket_eintr の場合、“Socket Interrupted” メッセージが返されます。
- WOLFSSL_CBIO_ERR_CONN_CLOSE 最後のエラーが wolfssl_cbio_err_conn_crose の場合、“Socket Epipe” メッセージを返しました。
- WOLFSSL_CBIO_ERR_GENERAL 最後のエラーが指定されていない場合は、「一般的なエラー」メッセージで返されます。

C.57.2.5 function EmbedGenerateCookie

```
int EmbedGenerateCookie(
    WOLFSSL * ssl,
    unsigned char * buf,
    int sz,
    void *
)
```

Parameters:

- **ssl** wolfssl_new（）を使用して作成された WolfSSL 構造へのポインタ。
- **buf** バッファを表すバイトポインタ。xmemcpy（）からの宛先です。
- **sz** バッファのサイズ。*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
}
```

See: wolfSSL_CTX_SetGenCookie

Return:

- Success この関数は、バッファにコピーされたバイト数を返します。
- GEN_COOKIE_E getPeername が EmbedGenerateCookie に失敗した場合に返されます。

C.57.2.6 function EmbedOcspRespFree

```
void EmbedOcspRespFree(
    void * ctx,
    byte * resp
)
```

Parameters:

- **ctx** ヒープヒントへの void ポインタ。*Example*

```
void* ctx;
byte* resp; // Response buffer.
…
EmbedOcspRespFree(ctx, resp);
```

See:

- wolfSSL_CertManagerSetOCSP_Cb
- wolfSSL_CertManagerEnableOCSPStapling
- wolfSSL_CertManagerEnableOCSP

Return: none いいえ返します。

C.57.2.7 function wolfSSL_CTX_SetIORecv

```
void wolfSSL_CTX_SetIORecv(
    WOLFSSL_CTX * ctx,
    CallbackIORecv CBIORecv
)
```

データ。デフォルトでは、WolfSSL はシステムの TCP RECV () 関数を使用するコールバックとして EmbedReceive () を使用します。ユーザは、メモリ、他のネットワークモジュール、またはどこからでも入力するように機能を登録できます。関数の機能とエラーコードのためのガイドとして、src / io.c の埋め込み Receive () 関数を参照してください。特に、データが準備ができていないときに、IO_ERR_WANT_READ を非ブロック受信用に返す必要があります。

Parameters:

- **ctx** wolfssl_ctx_new () で作成された SSL コンテキストへのポインタ。*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
}
```

See:

- wolfSSL_CTX_SetIOSend
- wolfSSL_SetIOReadCtx
- wolfSSL_SetIOWriteCtx

Return: none いいえ返します。

C.57.2.8 function wolfSSL_SetIOReadCtx

```
void wolfSSL_SetIOReadCtx(
    WOLFSSL * ssl,
    void * ctx
)
```

コールバック関数デフォルトでは、WolfSSL は、WolfSSL がシステムの TCP ライブラリを使用している場合、wolfssl_set_fd () に渡されたファイル記述子をコンテキストとして設定します。自分の受信コールバックを登録した場合は、セッションの特定のコンテキストを設定することができます。たとえば、メモリバッファを使用している場合、コンテキストは、メモリバッファーのどこにありかを説明する構造へのポインタであり得る。

Parameters:

- **ssl** wolfssl_new () で作成された SSL セッションへのポインタ。*Example*

```
int sockfd;
WOLFSSL* ssl = 0;
...
// Manually setting the socket fd as the receive CTX, for example
wolfSSL_SetIOReadCtx(ssl, &sockfd);
...
```

See:

- wolfSSL_CTX_SetIORecv
- wolfSSL_CTX_SetIOSend
- wolfSSL_SetIOWriteCtx

Return: none いいえ返します。

C.57.2.9 function wolfSSL_SetIOWriteCtx

```
void wolfSSL_SetIOWriteCtx(
    WOLFSSL * ssl,
    void * ctx
)
```

コールバック関数デフォルトでは、WolfSSL は、WolfSSL がシステムの TCP ライブラリを使用している場合、wolfssl_set_fd () に渡されたファイル記述子をコンテキストとして設定します。独自の送信コールバックを登録した場合は、セッションの特定のコンテキストを設定することができます。たとえば、メモリバッファを使用している場合、コンテキストは、メモリバッファーのどこにありかを説明する構造へのポインタであり得る。

Parameters:

- **ssl** wolfssl_new () で作成された SSL セッションへのポインタ。*Example*

```
int sockfd;
WOLFSSL* ssl = 0;
...
// Manually setting the socket fd as the send CTX, for example
wolfSSL_SetIOWriteCtx(ssl, &sockfd);
...
```

See:

- wolfSSL_CTX_SetIORecv
- wolfSSL_CTX_SetIOSend
- wolfSSL_SetIOReadCtx

Return: none いいえ返します。

C.57.2.10 function wolfSSL_GetIOReadCtx

```
void * wolfSSL_GetIOReadCtx(
    WOLFSSL * ssl
)
```

この関数は、WolfSSL 構造体の IOCB_READCTX メンバーを返します。

See:

- wolfSSL_GetIOWriteCtx
- wolfSSL_SetIOReadFlags
- wolfSSL_SetIOWriteCtx
- wolfSSL_SetIOReadCtx
- wolfSSL_CTX_SetIOSend

Return:

- pointer この関数は、wolfssl 構造体の iocb_readctx メンバーへの void ポインタを返します。
- NULL wolfssl 構造体が NULL の場合に返されます。*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
)
```

この関数は、WolfSSL 構造の IOCB_WRITECTX メンバーを返します。

See:

- wolfSSL_GetIOReadCtx
- wolfSSL_SetIOWriteCtx
- wolfSSL_SetIOReadCtx
- wolfSSL_CTX_SetIOSend

Return:

- pointer この関数は、WolfSSL 構造の IOCB_WRITECTX メンバーへの void ポインタを返します。
- NULL wolfssl 構造体が NULL の場合に返されます。*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
)
```

与えられた SSL セッション受信コールバックは、デフォルトの wolfssl 埋め込み受信コールバック、またはユーザによって指定されたカスタムコールバックであり得る（wolfssl_ctx_setiorecv を参照）。デフォルトのフラグ値は、WolfSSL によって wolfssl によって 0 の値に設定されます。デフォルトの WolfSSL 受信コールバックは RECV () 関数を使用してソケットからデータを受信します。「Recv ()」ページから：「Recv () 関数への flags 引数は、1 つ以上の値を OR 処理するか、MSG_OOB プロセス帯域外データ、MSG_PEEK PEEK、MSG_PEEK PEEK、MSG_WAITALL がフルを待っています要求またはエラー。MSG_OOB フラグは、通常のデータストリームで受信されないであろう帯域外データの受信を要求します。一部のプロトコルは通常のデータキューの先頭に迅速なデータを配置し、このフラグをそのようなプロトコルで使用することはできません。MSG_PEEK フラグは、受信操作によって受信キューの先頭からのデータをキューから削除することなくデータを返します。したがって、以降の受信呼び出しは同じデータを返します。MSG_WAITALL フラグは、完全な要求が満たされるまで操作ブロックを要求します。ただし、信号がキャッチされている場合は、呼び出し側よりも少ないデータが少なく、エラーまたは切断が発生するか、または受信されるデータが返されるものとは異なるタイプのデータを返します。

Parameters:

- **ssl** wolfssl_new () で作成された SSL セッションへのポインタ。*Example*

```
WOLFSSL* ssl = 0;
...
// Manually setting recv flags to 0
wolfSSL_SetIOReadFlags(ssl, 0);
...
```

See:

- wolfSSL_CTX_SetIORecv
- wolfSSL_CTX_SetIOSend
- wolfSSL_SetIOReadCtx

Return: none いいえ返します。

C.57.2.13 function wolfSSL_SetIOWriteFlags

```
void wolfSSL_SetIOWriteFlags(
    WOLFSSL * ssl,
    int flags
)
```

SSL セッションを考えると送信コールバックは、デフォルトの WolfSSL EmbedEnd コールバック、またはユーザーによって指定されたカスタムコールバックのいずれかです（WolfSSL_CTX_SetiosEnd を参照）。デフォルトのフラグ値は、wolfssl によって 0 の値に設定されます。デフォルトの WolfSSL Send Callback は send () 関数を使用してソケットからデータを送信します。send () man ページから：”flags パラメータには、次のうち 1 つ以上が含まれていてもよい。フラグ MSG_OOB は、この概念（例えば SOCK_STREAM）をサポートするソケットに「帯域外」データを送信するために使用される。基礎となるプロトコルは、「帯域外」のデータもサポートする必要があります。MSG_DONTROUTE は通常、診断プログラムまたはルーティングプログラムによってのみ使用されます。」

Parameters:

- **ssl** wolfssl_new () で作成された SSL セッションへのポインタ。*Example*

```
WOLFSSL* ssl = 0;
...
// Manually setting send flags to 0
wolfSSL_SetIOWriteFlags(ssl, 0);
...
```

See:

- wolfSSL_CTX_SetIORecv
- wolfSSL_CTX_SetIOSend
- wolfSSL_SetIOReadCtx

Return: none いいえ返します。

C.57.2.14 function wolfSSL_SetIO_NetX

```
void wolfSSL_SetIO_NetX(
    WOLFSSL * ssl,
    NX_TCP_SOCKET * nxsocket,
    ULONG waitoption
)
```

この関数は、wolfssl 構造内の nxctx 構造体の NxSocket メンバーと NXWAIT メンバーを設定します。

Parameters:

- **ssl** wolfssl_new（）を使用して作成された WolfSSL 構造へのポインタ。
- **nxSocket** NXCTX 構造の NXSOCTOCK メンバーに設定されている NX_TCP_SOCKET を入力するためのポインタ。*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.
}
```

See:

- set_fd
- NetX_Send
- NetX_Receive

Return: none いいえ返します。

C.57.2.15 function wolfSSL_CTX_SetGenCookie

```
void wolfSSL_CTX_SetGenCookie(
    WOLFSSL_CTX * ctx,
    CallbackGenCookie cb
)
```

wolfssl_ctx 構造 CallBackGencookie Type は関数ポインタで、署名：int（* callbackgencookie）（wolfssl * ssl、unsigned char * buf、int sz、void * ctx）を持っています。

Parameters:

- **ssl** wolfssl_new（）を使用して作成された WolfSSL 構造へのポインタ。*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);
```

See: CallbackGenCookie

Return: none いいえ返します。

C.57.2.16 function wolfSSL_GetCookieCtx

```
void * wolfSSL_GetCookieCtx(
    WOLFSSL * ssl
)
```

この関数は、WolfSSL 構造の IOCB_COOKIECTX メンバーを返します。

See:

- wolfSSL_SetCookieCtx
- wolfSSL_CTX_SetGenCookie

Return:

- pointer この関数は、iocb_cookiectx に格納されている void ポインタ値を返します。
- NULL WolfSSL 構造体が NULL の場合 *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
)
```

この関数は、WolfSSL が WolfSSL_ISOTP でコンパイルされている場合に使用する場合は、WolfSSL の場合は ISO-TP コンテキストを設定します。

Parameters:

- **ssl** wolfssl コンテキスト

- **ctx** ユーザーはこの関数が初期化される ISOTP コンテキストを作成しました
- **recv_fn** ユーザーはバスを受信できます
- **send_fn** ユーザーはバスを送ることができます
- **delay_fn** ユーザーマイクロ秒の粒度遅延関数
- **receive_delay** 各 CAN バスパケットを遅らせるためのマイクロ秒のセット数
- **receive_buffer** ユーザーがデータを受信するためのバッファーが提供され、ISOTP_DEFAULT_BUFFER_SIZE バイトに割り当てられていることをお勧めします。
- **receive_buffer_size** - receive_buffer のサイズ *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);
```

Return: 0 成功すると、故障の wolfssl_cbio_err_general

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);
```

D　SSL/TLS の概要

D.1　全体アーキテクチャ

組み込み向け SSL/TLS ライブラリである wolfSSL(旧称：Cyassl) は、SSL 3.0、TLS 1.0、TLS 1.1、TLS 1.2、および TLS 1.3 プロトコルを実装しています。TLS 1.3 は現在、標準化されている最も安全な最新バージョンです。wolfSSL は、数年間不安定であるという事実により、SSL 2.0 をサポートしていません。

wolfSSL の TLS プロトコルは、RFC 5246 (https://tools.ietf.org/html/rfc5246). で定義されているとおりに実装しています.TLS には、メッセージ層とハンドシェーク層の 2 つのレコード層プロトコルが存在します。ハンドシェイクメッセージは、共通の暗号スイートのネゴシエーション、シークレットの作成、および安全な接続の有効化に使用されます。メッセージレイヤーはハンドシェイクレイヤーをカプセル化すると同時に、アラート処理とアプリケーションデータ転送もサポートします。

SSL プロトコルが既存のプロトコルにどのように適合するかを、**図 1** に示します。SSL は、OSI モデルのトランスポート層とアプリケーション層の間に位置し、多くのプロトコル (TCP/IP、Bluetooth などを含む) がトランスポートメディアとして機能します。アプリケーションプロトコルは、SSL (HTTP、FTP、SMTP など) の上に階層化されています。

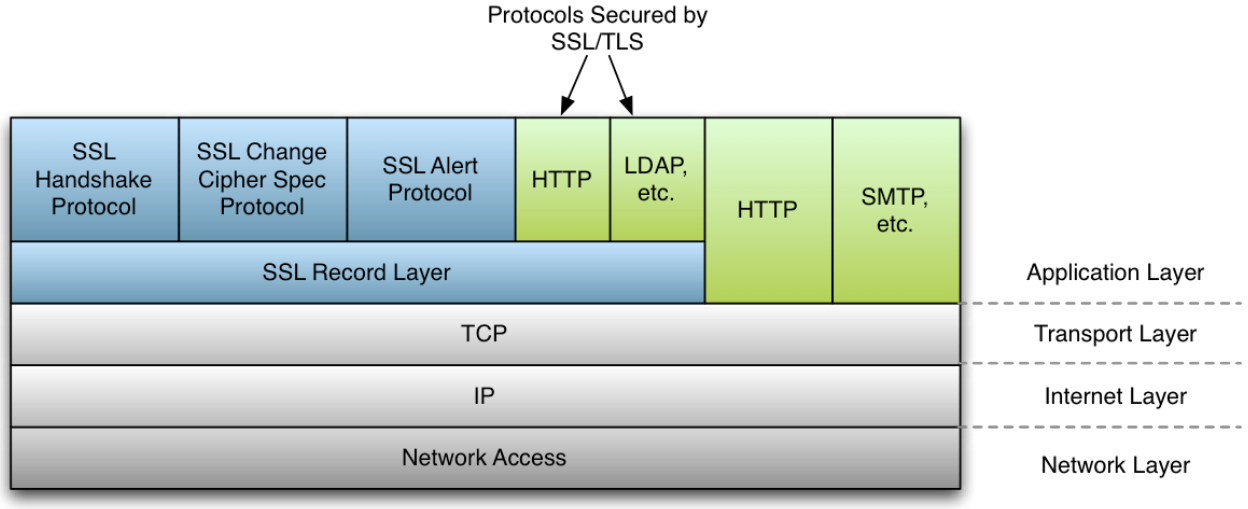

Figure 4: SSL Protocol Diagram

D.2　SSL ハンドシェイク

SSL ハンドシェイクプロセスの簡略を以下の**図 2** に示します。なお、SSL クライアントとサーバーの構成オプションによってはいくつかのステップは実行されません。

D.3　SSL プロトコルバージョンと TLS プロトコルバージョンの違い

SSL (Secure Sockets Layer) と TLS (Transport Security Layer) はどちらも、ネットワーク上で安全な通信を提供する暗号化プロトコルです。これら 2 つのプロトコル (および、それぞれのいくつかのバージョン) は、今日、Web ブラウジングから電子メール、インスタントメッセージング、VoIP に至るまで、さまざまなアプリケーションで広く使用されています。SSL と TLS、そしてそれぞれに含まれる各バージョン間にはいくつかの違いがあります。

以下に、SSL および TLS プロトコルの各バージョンについての解説と主な相違点を示します。各プロトコルの具体的な詳細については、記載されている RFC 仕様を参照してください。

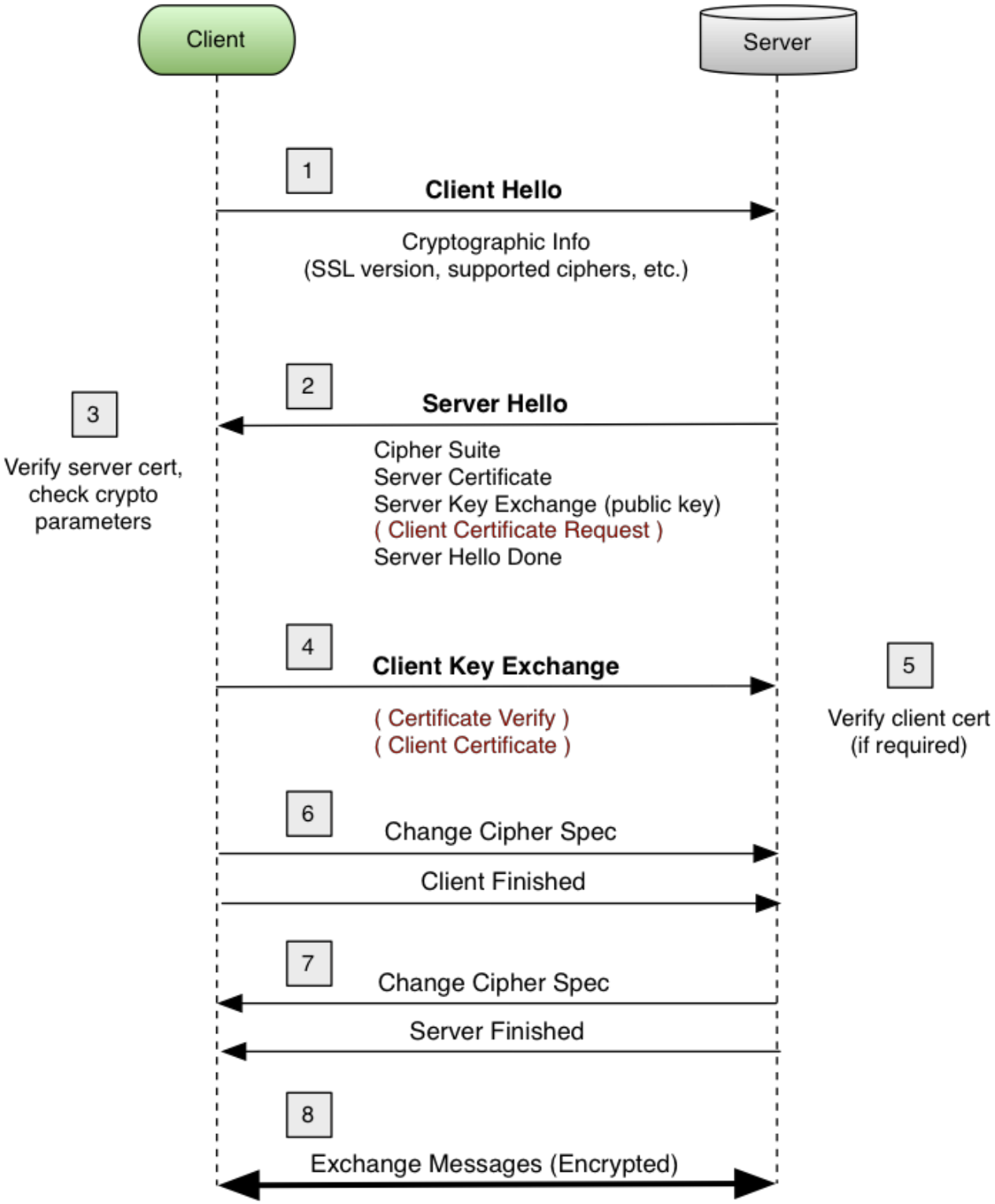

Figure 5: SSL Handshake Diagram

D.3.1 SSL 3.0

このプロトコルは Netscape によって開発された SSL 1.0 から始まり、1996 年にリリースされました。バージョン 1.0 はリリースされておらず、バージョン 2.0 には多くのセキュリティ上の欠陥があり、SSL 3.0 のリリースにつながりました。SSL 2.0 に対する SSL 3.0 のいくつかの主要な改善点は次のとおりです。

- メッセージ層からのデータ転送の分離
- Export Cipher を使用している場合でも、128 ビットのキーイングマテリアルを使用
- クライアントとサーバーが証明書のチェーンを送信し、組織は 2 つ以上の証明書の証明書階層を使用可能に
- 一般化された鍵交換プロトコルを実装し、Diffie-Hellman と Fortezza の鍵交換と非 RSA 証明書を許可
- レコードの圧縮と解凍を可能に
- 2.0 クライアントが検出されたときに SSL 2.0 にフォールバック

D.3.2 TLS 1.0

このプロトコルは 1999 年 1 月に RFC 2246 で最初に定義されました。これは SSL 3.0 からのアップグレードであり劇的な違いはありませんが、SSL 3.0 と TLS 1.0 は相互運用できない程度の変更を含んでいます。SSL 3.0 と TLS 1.0 の間の大きな違いとして、以下が挙げられます。

- 鍵導出関数の変更
- MAC の変更 - SSL 3.0 は初期 HMAC の変更を使用し、TLS 1.0 は HMAC を使用します。
- 完了 (Finished) メッセージの変更
- アラートの増加
- DSS/DH サポートの要求

D.3.3 TLS 1.1

このプロトコルは、2006 年 4 月に TLS 1.0 の後継として RFC 4346 で定義されました。主な変更点は次のとおりです。

- 暗黙の初期化ベクトル (IV) は、暗号ブロック連鎖 (CBC) 攻撃から保護するために明示的な IV に置き換えられました。
- パディングエラーの取り扱いは、CBC 攻撃から保護するために decryption_failed アラートではなく bad_record_mac アラートを使用するよう変更されました。
- IANA レジストリは、プロトコルパラメーター用に定義されました。
- 早期終了によってセッションが再開できなくなることがなくなりました。

D.3.4 TLS 1.2

このプロトコルは、2008 年 8 月に RFC 5246 で定義されました。TLS 1.1 をベースとして、いくつかの改善が行われました。主な相違点は次のとおりです。

- 擬似ランダム関数 (PRF) の MD5/SHA-1 の組み合わせを、暗号スイート指定 PRF に置き換えました。
- デジタル署名要素の MD5/SHA-1 の組み合わせは、単一のハッシュに置き換えられました。署名された要素には、使用されるハッシュアルゴリズムを明示的に指定するフィールドが含まれます。
- クライアントとサーバーが受け入れるハッシュおよび署名アルゴリズムの組み合わせが整理されました。

- 追加のデータモードを使用した、認証された暗号化のためのサポートが追加されました。
- TLS 拡張機能の定義と AES 暗号スイートがマージされました。
- EncryptedPremasterSecret バージョン番号が厳しくチェックされるようになりました。
- 多くの要件が厳しくなりました。
- `Verify_data` の長さは暗号スイートに依存します
- Bleichenbacher/Dlima 攻撃防御の説明が簡潔になりました。

D.3.5 TLS 1.3

このプロトコルは、2018 年 8 月に RFC 8446 で定義されました。主に、セキュリティ性能とスピードが向上しています。主な違いは次のとおりです。

- サポートされている対称アルゴリズムのリストは、すべての従来のアルゴリズムから整理されました。残りのアルゴリズムはすべて、認証タグ付き暗号 (AEAD) アルゴリズムを使用します。
- ゼロ RTT (0-RTT) モードが追加され、一部のセキュリティ属性を犠牲にすることで、一部のアプリケーションデータのための接続時のラウンドトリップが削減されました。
- ServerHello の後のすべてのハンドシェイクメッセージが暗号化されるようになりました。
- HMAC ベースの抽出および拡張鍵導出機能 (HKDF) がプリミティブとして使用されているため、鍵導出機能が再設計されました。
- ハンドシェイクステートマシンが再構築され、一貫性が向上し、余分なメッセージが削除されました。
- ECC は基本仕様になり、新しい署名アルゴリズムが含まれるようになりました。各曲線の単一のポイント形式を支持して、ポイント形式のネゴシエーションを削除しました。
- 圧縮、カスタム DHE グループ、および DSA が削除されました、RSA パディングは PSS を使用するようになりました。
- TLS 1.2 バージョンネゴシエーション検証メカニズムは廃止され、拡張機能のバージョンリストが採用されました。
- サーバー側の状態の有無にかかわらず、セッションの再開と、TLS の以前のバージョンの PSK ベースの暗号スイートは、単一の新しい PSK 交換に置き換えられました。

E　RFC、仕様、および参照

E.1　プロトコル

- SSL v3.0 - IETF ドラフト
- TLS V1.0-RFC2246
- TLS V1.1-RFC4346
- TLS V1.2-RFC5246
- TLS V1.3 - RFC8446.
- DTLS - RFC4347 仕様文書
- IPv4 -ウィキペディア
- IPv6 - ウィキペディア

E.2　ストリーム暗号

- ストリーム暗号情報-ウィキペディア
- RC4/ARC4 - IETF ドラフト ウィキペディア

E.3　ブロック暗号

- ブロック暗号情報 - ウィキペディア
- AES - NIST 出版 ウィキペディア
- AES-GCM - NIST 仕様
- AES -NI -インテルソフトウェアネットワーク
- DES/3DES -NIST 出版 ウィキペディア

E.4　ハッシュ機能

- SHA -NIST FIPS180-1 発表 NIST FIPS180-2 出版物 ウィキペディア
- MD4 -RFC1320
- MD5 -RFC1321
- RIPEMD -160-仕様文書

E.5　公開鍵暗号

- Diffie -Hellman -ウィキペディア
- RSA - MIT 紙 ウィキペディア
- DSA/DSS -NIST FIPS186-3
- ECDSA - 仕様書
- X.509-RFC3279
- ASN.1 - 仕様書 ウィキペディア
- PSK -RFC4279

E.6 その他

- PKCS # 5、PBKDF1、PBKDF2-RFC2898
- PKCS # 8-RFC5208.
- PKCS # 12 - ウィキペディア

F エラーコード

F.1 wolfSSL エラーコード

wolfSSL(以前の Cyassl) エラーコードは wolfssl/ssl.h にあります。次のエラーの詳細な説明については、SSL_get_error(man SSL_get_error) の OpenSSL Man ページを参照してください。

エラーコード列挙	エラーコード	エラー説明
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	

追加の wolfSSL エラーコードは wolfssl/error-ssl.h にあります

エラーコード列挙	エラーコード	エラー説明
INPUT_CASE_ERROR	-301	プロセス入力状態エラー
PREFIX_ERROR	-302	キーラウンドの悪いインデックス
MEMORY_ERROR	-303	メモリ確保失敗
VERIFY_FINISHED_ERROR	-304	Finished メッセージ検証に失敗
VERIFY_MAC_ERROR	-305	Mac 検証に失敗
PARSE_ERROR	-306	ヘッダーの解析エラー
UNKNOWN_HANDSHAKE_TYPE	-307	不明なハンドシェークタイプ
SOCKET_ERROR_E	-308	ソケットがエラー状態
SOCKET_NODATA	-309	予想されるデータがソケットにない
INCOMPLETE_DATA	-310	タスク完了のための十分なデータがない
UNKNOWN_RECORD_TYPE	-311	レコード HDR に未知のタイプ
DECRYPT_ERROR	-312	復号中のエラー
FATAL_ERROR	-313	致命的なエラーアラートを受信
ENCRYPT_ERROR	-314	暗号化中のエラー
FREAD_ERROR	-315	fread でエラー
NO_PEER_KEY	-316	ピアの鍵が存在しない
NO_PRIVATE_KEY	-317	秘密鍵が存在しない
RSA_PRIVATE_ERROR	-318	RSA 秘密鍵操作中のエラー
NO_DH_PARAMS	-319	サーバーが DH パラメータを送ってこない
BUILD_MSG_ERROR	-320	メッセージの作成に失敗
BAD_HELLO	-321	ClientHello メッセージが不正
DOMAIN_NAME_MISMATCH	-322	ピアのサブジェクト名がミスマッチ
WANT_READ	-323	データ再読み取りが必要
NOT_READY_ERROR	-324	ハンドシェイクレイヤーが Ready でない
VERSION_ERROR	-326	レコードレイヤーバージョンエラー
WANT_WRITE	-327	データ再送が必要
BUFFER_ERROR	-328	不正なバッファー入力
VERIFY_CERT_ERROR	-329	verify cert エラ
VERIFY_SIGN_ERROR	-330	verify sign エラー
CLIENT_ID_ERROR	-331	PSK クライアント ID エラー
SERVER_HINT_ERROR	-332	PSK サーバーヒントエラー

エラーコード列挙	エラーコード	エラー説明
PSK_KEY_ERROR	-333	PSK 鍵エラー
GETTIME_ERROR	-337	GetTimeOfday が失敗
GETITIMER_ERROR	-338	Getitimer が失敗
SIGACT_ERROR	-339	sigaction が失敗
SETITIMER_ERROR	-340	setitimer が失敗
LENGTH_ERROR	-341	レコードレイヤーの長さが不正
PEER_KEY_ERROR	-342	ピアの鍵をデコードできず
ZERO_RETURN	-343	ピアが close notify を送信
SIDE_ERROR	-344	クライアント/サーバータイプが不正
NO_PEER_CERT	-345	ピアが鍵を送信しなかった
ECC_CURVETYPE_ERROR	-350	ECC カーブタイプが不正
ECC_CURVE_ERROR	-351	ECC 曲線が不正
ECC_PEERKEY_ERROR	-352	ピア ECC 鍵が不正
ECC_MAKEKEY_ERROR	-353	Make ECC 鍵が不正
ECC_EXPORT_ERROR	-354	ECC エクスポート鍵が不正
ECC_SHARED_ERROR	-355	ECC シェアードシークレットが不正
NOT_CA_ERROR	-357	CA 証明書ではない
BAD_CERT_MANAGER_ERROR	-359	Cert Manager が不正
OCSP_CERT_REVOKED	-360	OCSP 証明書が取り消されました
CRL_CERT_REVOKED	-361	CRL 証明書が取り消されました
CRL_MISSING	-362	CRL ロードされていない
MONITOR_SETUP_E	-363	CRL モニターセットアップエラー
THREAD_CREATE_E	-364	スレッド作成でエラー
OCSP_NEED_URL	-365	OCSP
OCSP_CERT_UNKNOWN	-366	OCSP レスポンダーが不明
OCSP_LOOKUP_FAIL	-367	OCSP ルックアップが失敗
MAX_CHAIN_ERROR	-368	最大チェーンの深さを超えた
COOKIE_ERROR	-369	DTLS クッキーエラー
SEQUENCE_ERROR	-370	DTLS シーケンスエラー
SUITES_ERROR	-371	スイートポインタエラー
OUT_OF_ORDER_E	-373	Out of Order メッセージ受信
BAD_KEA_TYPE_E	-374	悪い KEA タイプが見つかりました
SANITY_CIPHER_E	-375	SANITY チェック暗号エラー
RECV_OVERFLOW_E	-376	受信コールバックが要求以上のデータを返却した
GEN_COOKIE_E	-377	クッキーエラーを生成
NO_PEER_VERIFY	-378	検証すべきピア証明書が無い
FWRITE_ERROR	-379	fwrite エラー
CACHE_MATCH_ERROR	-380	キャッシュ HRD 不一致
UNKNOWN_SNI_HOST_NAME_E	-381	認識されていないホスト名
UNKNOWN_MAX_FRAG_LEN_E	-382	認識されない最大フラグ長
KEYUSE_SIGNATURE_E	-383	KeyUse digSignature エラー
KEYUSE_ENCIPHER_E	-385	KeyUse KeyEncipher エラー
EXTKEYUSE_AUTH_E	-386	ExtKeyUse server
SEND_OOB_READ_E	-387	送信コールバックが帯域外データ受信
SECURE_RENEGOTIATION_E	-388	無効な再ネゴシエーション情報
SESSION_TICKET_LEN_E	-389	セッションチケットが大きすぎ
SESSION_TICKET_EXPECT_E	-390	セッションチケットがありません
SCR_DIFFERENT_CERT_E	-391	SCR 異なる証明書エラー
SESSION_SECRET_CB_E	-392	セッションシークレット CB FCN 障害
NO_CHANGE_CIPHER_E	-393	Cipher を変更する前に終了
SANITY_MSG_E	-394	メッセージ順のサニティチェックでエラー
DUPLICATE_MST_E	-395	重複メッセージエラー

エラーコード列挙	エラーコード	エラー説明
SNI_UNSUPPORTED	-396	SSL 3.0 は SNI をサポートしません
SOCKET_PEER_CLOSED_E	-397	トランスポート層がクローズされた
BAD_TICKET_KEY_CB_SZ	-398	セッションチケットキー CB サイズ不正
BAD_TICKET_MSG_SZ	-399	セッションチケット MSG サイズ不正
BAD_TICKET_ENCRYPT	-400	ユーザーチケット暗号化不正
DH_KEY_SIZE_E	-401	DH キーが小さすぎる
SNI_ABSENT_ERROR	-402	SNI リクエストはありません
RSA_SIGN_FAULT	-403	RSA 署名障害
HANDSHAKE_SIZE_ERROR	-404	ハンドシェークメッセージが大きすぎる
UNKNOWN_ALPN_PROTOCOL_NAME_E	-405	認識されていないプロトコル名エラー
BAD_CERTIFICATE_STATUS_ERROR	-406	証明書ステータスメッセージ
OCSP_INVALID_STATUS	-407	OCSP ステータスが無効です
OCSP_WANT_READ	-408	OCSP コールバック応答
RSA_KEY_SIZE_E	-409	RSA キーが小さすぎる
ECC_KEY_SIZE_E	-410	ECC キーが小さすぎる
DTLS_EXPORT_VER_E	-411	バージョンエラーをエクスポートする
INPUT_SIZE_E	-412	入力サイズ大きすぎます
CTX_INIT_MUTEX_E	-413	CTX ミューテックスエラーを初期化します
EXT_MASTER_SECRET_NEEDED_E	-414	EMS が再開することができます
DTLS_POOL_SZ_E	-415	DTLS プールサイズを超えました
DECODE_E	-416	ハンドシェイクメッセージエラーをデコードします
HTTP_TIMEOUT	-417	OCSP または CRL REQ の HTTP タイムアウト
WRITE_DUP_READ_E	-418	書き込み dup write side は読めない
WRITE_DUP_WRITE_E	-419	dup 読み取り side は書くことができません
INVALID_CERT_CTX_E	-420	TLS CERT CTX は一致していません
BAD_KEY_SHARE_DATA	-421	キー共有データ無効
MISSING_HANDSHAKE_DATA	-422	ハンドシェイクメッセージ欠落データ
BAD_BINDER	-423	バインダーが一致しません
EXT_NOT_ALLOWED	-424	MSG では許可されていない拡張
INVALID_PARAMETER	-425	セキュリティパラメーター無効
MCAST_HIGHWATER_CB_E	-426	マルチキャストハイウォーター CB err
ALERT_COUNT_E	-427	アラート数を超えました
EXT_MISSING	-428	必要な拡張機能が見つかりません
UNSUPPORTED_EXTENSION	-429	TLSX はクライアントから要求されていません
PRF_MISSING	-430	
DTLS_RETX_OVER_TX	-431	DTLS フライトを再送信
DH_PARAMS_NOT_FFDHE_E	-432	ffdhe ではなくサーバーからの dh パラメーション
TCA_INVALID_ID_TYPE	-433	TLSX TCA ID タイプ無効
TCA_ABSENT_ERROR	-434	TLSX TCA ID 応答なし

ネゴシエーションパラメーターエラー

エラーコード列挙	エラーコード	エラー説明
UNSUPPORTED_SUITE	-500	サポートされていない暗号スイート
MATCH_SUITE_ERROR	-501	暗号スイートと一致することはできません
COMPRESSION_ERROR	-502	圧縮ミスマッチ
KEY_SHARE_ERROR	-503	キーシェアミスマッチ
POST_HAND_AUTH_ERROR	-504	クライアントはポストハンド認証を行いません
HRR_COOKIE_ERROR	-505	HRR MSG クッキーミスマッチ

F.2 wolfCrypt エラーコード

wolfCrypt エラーコードは wolfssl/wolfcrypt/error.h にあります。

エラーコード列挙	エラーコード	エラー説明
OPEN_RAN_E	-101	ランダムデバイスを開く際にエラー
READ_RAN_E	-102	ランダムデバイスのリード時にエラー
WINCRYPT_E	-103	Windows Crypt init エラー
CRYPTGEN_E	-104	Windows 暗号化エラー
RAN_BLOCK_E	-105	ランダムデバイスの読み取りで would block
BAD_MUTEX_E	-106	ミューテックス操作で失敗
MP_INIT_E	-110	mp_init エラー状態
MP_READ_E	-111	mp_read エラー状態
MP_EXPTMOD_E	-112	MP_EXPTMOD エラー状態
MP_TO_E	-113	MP_TO_XXX エラー状態、変換できません
MP_SUB_E	-114	MP_SUB エラー状態、減算できません
MP_ADD_E	-115	MP_ADD エラー状態、追加できません
MP_MUL_E	-116	MP_MUL エラー状態、
MP_MULMOD_E	-117	MP_MULMOD エラー状態、Morply MOD
MP_MOD_E	-118	mp_mod エラー状態、mod
MP_INVMOD_E	-119	MP_INVMOD エラー状態、INV MOD
MP_CMP_E	-120	MP_CMP エラー状態
MP_ZERO_E	-121	予想されない、MP ゼロ結果を得た
MEMORY_E	-125	メモリのエラー
RSA_WRONG_TYPE_E	-130	RSA 関数の RSA 間違ったブロックタイプ
RSA_BUFFER_E	-131	RSA バッファーエラー、出力が小さすぎる、または入力が大き
BUFFER_E	-132	出力バッファーが小さすぎるか、入力が大きすぎる
ALGO_ID_E	-133	ALGO ID エラーの設定
PUBLIC_KEY_E	-134	公開キーエラーの設定
DATE_E	-135	日付有効性エラー
SUBJECT_E	-136	件名名の設定エラー
ISSUER_E	-137	発行者名エラーの設定
CA_TRUE_E	-138	CA Basic Constraint True Error
EXTENSIONS_E	-139	拡張機能の設定エラー
ASN_PARSE_E	-140	ASN 解析エラー、無効な入力
ASN_VERSION_E	-141	ASN バージョンエラー、無効な数値
ASN_GETINT_E	-142	ASN は大きな INT エラーを取得し、データが無効なデータ
ASN_RSA_KEY_E	-143	ASN キー INIT エラー、無効な入力
ASN_OBJECT_ID_E	-144	ASN オブジェクト ID エラー、無効な ID
ASN_TAG_NULL_E	-145	asn タグエラー、null
ASN_EXPECT_0_E	-146	asn はゼロではなく、エラーを期待します
ASN_BITSTR_E	-147	ASN ビット文字列エラー、間違った ID
ASN_UNKNOWN_OID_E	-148	ASN OID エラー、不明な合計 ID
ASN_DATE_SZ_E	-149	ASN 日付エラー、悪いサイズ
ASN_BEFORE_DATE_E	-150	ASN 日付エラー、現在の日付
ASN_AFTER_DATE_E	-151	ASN 日付エラー、現在の日付
ASN_SIG_OID_E	-152	ASN シグネチャエラー、OID の不一致
ASN_TIME_E	-153	ASN タイムエラー、不明な時間タイプ
ASN_INPUT_E	-154	ASN 入力エラー、十分なデータがありません
ASN_SIG_CONFIRM_E	-155	ASN SIG エラー、失敗の確認
ASN_SIG_HASH_E	-156	ASN SIG エラー、サポートされていないハッシュタイプ
ASN_SIG_KEY_E	-157	ASN SIG エラー、サポートされていないキータイプ
ASN_DH_KEY_E	-158	ASN キー INIT エラー、無効な入力
ASN_CRIT_EXT_E	-160	サポートされていない批判的拡張

エラーコード列挙	エラーコード	エラー説明
ECC_BAD_ARG_E	-170	間違ったタイプの ECC 入力引数
ASN_ECC_KEY_E	-171	ASN ECC 不良入力
ECC_CURVE_OID_E	-172	サポートされていない ECC OID カーブタイプ
BAD_FUNC_ARG	-173	提供された悪い関数引数
NOT_COMPILED_IN	-174	機能がコンパイルされていない
UNICODE_SIZE_E	-175	Unicode パスワード
NO_PASSWORD	-176	ユーザーが提供するパスワードはありません
ALT_NAME_E	-177	Alt Name Size の問題、大きすぎます
AES_GCM_AUTH_E	-180	AES-GCM 認証チェックの失敗
AES_CCM_AUTH_E	-181	AES-CCM 認証チェック不良
CAVIUM_INIT_E	-182	キャビウム init タイプエラー
COMPRESS_INIT_E	-183	init Error を圧縮します
COMPRESS_E	-184	エラーを圧縮する
DECOMPRESS_INIT_E	-185	init エラーを減圧します
DECOMPRESS_E	-186	解凍エラー
BAD_ALIGN_E	-187	操作のための悪いアライメント、Alloc のない
ASN_NO_SIGNER_E	-188	ASN SIG エラー、証明書を検証するための CA 署名者はありま
ASN_CRL_CONFIRM_E	-189	ASN CRL 失敗を確認する署名者はありません
ASN_CRL_NO_SIGNER_E	-190	ASN CRL 失敗を確認する署名者はありません
ASN_OCSP_CONFIRM_E	-191	ASN OCSP シグネチャーの確認失敗
BAD_ENC_STATE_E	-192	悪い ECC ENC 状態操作
BAD_PADDING_E	-193	悪いパディング、MSG は正しい長さ
REQ_ATTRIBUTE_E	-194	CERT 要求属性の設定エラー
PKCS7_OID_E	-195	PKCS # 7、不一致の OID エラー
PKCS7_RECIP_E	-196	PKCS # 7、受信者エラー
FIPS_NOT_ALLOWED_E	-197	FIPS は許可されていません
ASN_NAME_INVALID_E	-198	ASN 名制約エラー
RNG_FAILURE_E	-199	RNG が失敗し、再生
HMAC_MIN_KEYLEN_E	-200	FIPS モード HMAC 最小キー長エラー
RSA_PAD_E	-201	RSA パディングエラー
LENGTH_ONLY_E	-202	出力の長さのみを返す
IN_CORE_FIPS_E	-203	コアの整合性チェックの失敗 (障害)
AES_KAT_FIPS_E	-204	AES カット失敗
DES3_KAT_FIPS_E	-205	DES3 KAT 障害
HMAC_KAT_FIPS_E	-206	HMAC KAT 障害
RSA_KAT_FIPS_E	-207	RSA KAT 失敗
DRBG_KAT_FIPS_E	-208	ハッシュ DRBG KAT 障害
DRBG_CONT_FIPS_E	-209	ハッシュ DRBG 連続テスト障害
AESGCM_KAT_FIPS_E	-210	AESGCM KAT 障害
THREAD_STORE_KEY_E	-211	スレッドローカルストレージキーは障害を作成します
THREAD_STORE_SET_E	-212	スレッドローカルストレージキーセット障害
MAC_CMP_FAILED_E	-213	Mac の比較に失敗しました
IS_POINT_E	-214	ECC は曲線上のポイントに失敗しました
ECC_INF_E	-215	ECC ポイントインフィニティエラー
ECC_PRIV_KEY_E	-216	ECC 秘密鍵が無効なエラー
SRP_CALL_ORDER_E	-217	SRP 機能が間違っている
SRP_VERIFY_E	-218	SRP プルーフ検証失敗
SRP_BAD_KEY_E	-219	SRP の悪い一時的な値
ASN_NO_SKID	-220	ASN 主題のキー識別子
ASN_NO_AKID	-221	ASN 認証鍵識別子を見つけません
ASN_NO_KEYUSAGE	-223	ASN キー使用量は見つかりませんでした
SKID_E	-224	件名キー識別子エラーの設定

エラーコード列挙	エラーコード	エラー説明
AKID_E	-225	権限キー識別子エラー
KEYUSAGE_E	-226	悪いキー使用率値
CERTPOLICIES_E	-227	証明書ポリシーの設定エラー
WC_INIT_E	-228	wolfCrypt は初期化に失敗しました
SIG_VERIFY_E	-229	wolfCrypt シグネチャーの検証エラー
BAD_PKCS7_SIGNEEDS_CHECKCOND_E	-230	悪条件変数演算
SIG_TYPE_E	-231	署名タイプが有効/利用可能な
HASH_TYPE_E	-232	ハッシュタイプは有効/利用可能ではありません
WC_KEY_SIZE_E	-234	キーサイズエラー、小さすぎるか大きすぎる
ASN_COUNTRY_SIZE_E	-235	ASN CERT GEN、無効な国コードサイズ
MISSING_RNG_E	-236	RNG は必要ですが、提供されていません
ASN_PATHLEN_SIZE_E	-237	ASN CA パスの長さ大きすぎます
ASN_PATHLEN_INV_E	-238	ASN CA パスの長さ反転誤差
BAD_KEYWRAP_ALG_E	-239	KeyWrap のアルゴリズムエラー
BAD_KEYWRAP_IV_E	-240	復号化された AES キーラップ IV 不正
WC_CLEANUP_E	-241	wolfCrypt のクリーンアップに失敗しました
ECC_CDH_KAT_FIPS_E	-242	ECC CDH 既知の回答テスト失敗
DH_CHECK_PUB_E	-243	DH 公開キーエラーを確認します
BAD_PATH_ERROR	-244	Opendir の悪い道
ASYNC_OP_E	-245	非同期操作エラー
ECC_PRIVATEONLY_E	-246	プライベートのみの ECC キーの無効な使用
EXTKEYUSAGE_E	-247	大幅なキー使用率値
WC_HW_E	-248	ハードウェア暗号を使用するエラー
WC_HW_WAIT_E	-249	リソースを待っているハードウェア
PSS_SALTLEN_E	-250	PSS の塩の長さはハッシュには長すぎます
PRIME_GEN_E	-251	プライムを見つける障害
BER_INDEF_E	-252	無期限の長さの BER を復号することはできません
RSA_OUT_OF_RANGE_E	-253	範囲外の暗号化への暗号文
RSAPSS_PAT_FIPS_E	-254	RSA-PSS パット障害
ECDSA_PAT_FIPS_E	-255	ECDSA PAT FALION
DH_KAT_FIPS_E	-256	DH KAT 障害
AESCCM_KAT_FIPS_E	-257	AESCCM KAT 障害
SHA3_KAT_FIPS_E	-258	SHA-3 KAT 障害
ECDHE_KAT_FIPS_E	-259	ECDHE KAT 失敗
AES_GCM_OVERFLOW_E	-260	AES-GCM 呼び出しカウンターオーバーフロー
AES_CCM_OVERFLOW_E	-261	AES-CCM 呼び出しカウンタオーバーフロー
RSA_KEY_PAIR_E	-262	RSA キーペアワイズコンシステンシチェックフェイル
DH_CHECK_PRIVE_E	-263	DH 秘密鍵エラーを確認します
WC_AFALG_SOCK_E	-264	AF_ALG ソケットエラー
WC_DEVCRYPTO_E	-265	/dev/crypto エラー
ZLIB_INIT_ERROR	-266	ZLIB INIT エラー
ZLIB_COMPRESS_ERROR	-267	Zlib 圧縮エラー
ZLIB_DECOMPRESS_ERROR	-268	ZLIB 伸張エラー
PKCS7_NO_SIGNER_E	-269	PKCS7 署名データ MSG の署名者はいません
WC_PKCS7_WANT_READ_E	-270	PKCS7 ストリーム操作では、より多くの入力が必要です
CRYPTOCB_UNAVAILABLE	-271	Crypto Callback が利用できません
PKCS7_SIGNEEDS_CHECK	-272	発信者によって検証された署名のニーズ
ASN_SELF_SIGNED_E	-275	ASN 自己署名証明書エラー
MIN_CODE_E	-300	エラー-101- -299

F.3 一般的なエラーコードとその解決策

アプリケーションを wolfSSL で起動して実行するときに一般的に起こるエラーコードがいくつかあります。

F.3.1 ASN_NO_SIGNER_E (-188)

このエラーは証明書を使用して署名 CA 証明書をロードしていない場合に発生します。これは、wolfSSL サンプルクライアントを使用して Google への接続など、wolfSSL のサンプルサーバーまたはクライアントを別のクライアントまたはサーバーに使用して見ることができます。

```
./examples/client/client -g -h www.google.com -p 443
```

Google の CA 証明書には「-A」コマンドラインオプションがロードされていないため、これはエラー-188 で失敗します。

F.3.2 WANT_READ (-323)

WANT_READ エラーは、非ブロッキングソケットを使用する場合に頻繁に発生し、非ブロッキングソケットを使用する場合は実際にはエラーではありませんが、エラーとして発信者に渡されます。I/O コールバックからデータを受信する呼び出しが現在受信できるデータがないため、ブロックされると、I/O コールバックは WANT_READ を返します。発信者は、後で待って再度受信してみてください。これは通常、wolfSSL_read()、wolfSSL_negotiate()、wolfSSL_accept()、およびwolfSSL_connect()への呼び出しから見られます。クライアントとサーバーのサンプルは、デバッグが有効になっているときの WANT_READ インシデントを示します。

G　ポスト量子暗号の実験

wolfSSL チームは、実験的なポスト量子暗号アルゴリズムを wolfSSL ライブラリに統合しました。これは、Open Quantum Safe チームの Liboqs と統合することで行われました。それらの詳細については、https://openquantumsafe.org を参照してください。

この付録は、TLS 1.3 のコンテキストで、ポスト量子暗号について学び、実験したい人を対象としています。そポスト量子アルゴリズムが重要である理由、量子の脅威に対応して私たちが行ったこと、これらの新しいアルゴリズムの実験を開始する方法について説明します。

NOTE：liboq が提供するポスト量子アルゴリズムは標準化されておらず、実験的なものです。それらが生産環境では使用しないことを強くお勧めします。すべての OID、コードポイント、アーティファクト形式は一時的なものであり、将来変更される予定です。下位互換性がないことに注意してください。

注：wolfssl が`--with-liboqs` フラグで構成されていない場合は、これらの実験的アルゴリズムは有効になっていません。

G.1　ポスト量子暗号をわかりやすく紹介

G.1.1　なぜポスト量子暗号？

今日では、量子コンピューターの開発にますます多くのリソースが割かれるようになりました。そのため、クラウド量子コンピューティングリソースの商業化がすでに始まっています。現時点では未だ実用レベルの暗号が解かれた報告はありません。しかし、「あらかじめデータを収集、蓄積し、後に時間をかけて解読を進めていく」といった脅威モデルが存在します。すなわち、暗号の復号に特化した量子コンピューターが出現するよりも早いうちに準備が必要です。

NIST が、量子コンピューターに対して脆弱になる公開鍵暗号アルゴリズムを置き換えるように設計された新しいクラスのアルゴリズムの標準化を進めています。この章の執筆時点で、NIST は PQC 標準化プロセスの第 3 ラウンドをほぼ完了させており、2022 年初頭に標準化されるアルゴリズムを発表する予定です。その後、プロトコルとデータ形式を記述した標準文書を作成するプロセスにはさらに 1 年かかると予測されています。さらにその後には、FIPS のような規制の策定が開始される可能性があります。

G.1.2　私たちは自分自身をどのように守るのですか？

大まかに言えば、すべての TLS 1.3 接続において、認証と機密性は各接続を保護する重要な目標です。認証は、ECDSA などの署名スキームによって維持されます。機密性は、ECDHE などのキー確立アルゴリズムによって維持され、確立されたキーと AES などの対称暗号化アルゴリズムを使用して通信ストリームを暗号化します。したがって、TLS 1.3 プロトコルのセキュリティは、次の 3 種類の暗号化アルゴリズムに分解できます。

- 認証アルゴリズム
- キー確立アルゴリズム
- 対称暗号アルゴリズム

量子コンピュータが従来の暗号に及ぼす脅威には 2 つの形態があります。グローバーのアルゴリズムは、最新の対称暗号アルゴリズムのセキュリティを半分に低下させ、ショアのアルゴリズムは、最新の認証および鍵確立アルゴリズムのセキュリティを完全に破壊します。その結果、対称暗号アルゴリズムの強度を 2 倍にし、従来の認証および鍵確立アルゴリズムをポスト量子アルゴリズムに置き換えることで、通信を保護し続けることができます。TLS 1.3 ハンドシェイク中、暗号スイートは接続中に使用される対称暗号を指定します。AES-128 は一般的に十分であると認められているため、AES_256_GCM_SHA384 暗号スイートを使用することで強度を 2 倍にすることができます。鍵確立と認証には、ポスト量子 KEM (Key Encapsulation Mechanisms) と署名スキームがあります。

これらは量子コンピュータへの耐性のため、従来のアルゴリズムとは異なる種類の数学を使用して特別に設計されます。私たちが統合することを選択した認証アルゴリズムと KEM はすべて格子ベースのアルゴリズムです。

- ダイリチウム署名スキーム
- ファルコン署名スキーム
- KYBER KEM

注: SABRE KEM と NTRU KEM は非推奨となり、標準化に移行しなかったため削除されました。

注: KYBER KEM 90s の亜種は非推奨となり、NIST が標準化を検討していないため削除されました。

注: ダイリチウム署名方式の AES バリアントは非推奨となり、NIST が標準化を検討していないため削除されました。

格子ベースの暗号化の説明はこのドキュメントの範囲外のため、これらのアルゴリズムに関する詳細は、NIST の公開文書 https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions をご確認ください。

残念ながら、この章を記している時点においてこれらのアルゴリズムが量子コンピューターからの攻撃に耐えられるかどうかは、まだ分かっていません。従来型コンピューターにおける攻撃耐性も同様です。可能性はますます低くなっていますが、誰かが格子ベースの暗号化を破る可能性があります。ただし、暗号化は常にこのような経緯で機能してきた歴史があります。アルゴリズムは使い始めたときは優れていますが、弱点や脆弱性が発見され、テクノロジーは順次改善されます。ポスト量子アルゴリズムは比較的新しいため、コミュニティからもう少し注目する必要があるかもしれません。

解決策の 1 つは、これらの新しいアルゴリズムを完全には信頼しないことです。今のところ、ポスト量子 KEM を、信頼している従来のアルゴリズムとハイブリッド化することで、このリスクを回避することができます。FIPS 準拠を第一に考えると、NIST 標準曲線を使用した ECC は良い選択肢になります。このため、ポスト量子 KEM を統合するだけでなく、NIST が承認した曲線上の ECDSA とハイブリッド化しました。詳しくは以下のハイブリッドグループのリストを参照してください。

G.2 w olfSSL の Liboqs 統合を始めましょう

ここでは、まっさらな Linux 環境から安全な TLS 1.3 接続を実行できるようにするまでの手順を示します。

G.2.1 ビルド手順

wolfSSL リポジトリの INSTALL ファイル (https://github.com/wolfSSL/wolfssl/blob/master/INSTALL) を参照してください。

項目 15 (TLS 1.3 用の liboq を使用したビルド [実験的]) には、構成とビルドの方法に関する説明があります。

- liboqs
- wolfssl
- OQS の OpenSSL フォークにパッチを適用

ポスト量子暗号鍵と署名を使用して X.509 証明書を生成するには、パッチを適用した OQS OpenSSL フォークが必要です。手順は https://github.com/wolfSSL/osp/tree/master/oqs/README.md にあります。ポスト量子署名スキームを使用したくない場合は、OpenSSL を構築するステップをスキップできます。

G.2.2 量子安全な TLS 接続を確立する

次のようにして、サーバーとクライアントを別々のターミナルで実行します。

```
examples/server/server -v 4 -l TLS_AES_256_GCM_SHA384 \
  -A certs/falcon_level5_root_cert.pem \
  -c certs/falcon_level1_entity_cert.pem \
  -k certs/falcon_level1_entity_key.pem \
  --oqs P521_KYBER_LEVEL5

examples/client/client -v 4 -l TLS_AES_256_GCM_SHA384 \
  -A certs/falcon_level1_root_cert.pem \
  -c certs/falcon_level5_entity_cert.pem \
  -k certs/falcon_level5_entity_key.pem \
  --oqs P521_KYBER_LEVEL5
```

対称暗号化に AES-256、認証に FALCON 署名方式、キー確立に KYBER KEM とハイブリッド化された ECDHE を使用して、完全に量子安全な TLS 1.3 接続を実現しました。他のポスト量子の例についての詳細は、https://github.com/wolfSSL/wolfssl-examples/blob/master/pq/README.md で見つけることができます。

G.3 wolfSSL と OQS のフォークの OpenSSL の間の命名規則マッピング

NIST PQC コンテストに応募したすべてのチームは、NIST が定義する複数のセキュリティレベルをサポートしていました。https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/evaluation-criteria/security-(evaluation-criteria

そのため、各チームはバリアントを識別する方法を考え出す必要があり、各チームが独自のバリアント命名スキームを考え出しました。次の表からわかるように、この方法についてチーム間で調整はありませんでした。wolfSSL ライブラリは、バリアントの NIST レベルベースの命名規則を使用します。OQS チームは、各応募論文の命名規則に従うことを選択しました。次の表で、当社の命名規則と応募論文の命名規則をマッピングしています。

ポスト量子署名方式の命名規則:：

wolfSSL バリアント名	PQC 提出バリアント名
FALCON_LEVEL1	FALCON512
FALCON_LEVEL5	FALCON1024
DILITHIUM_LEVEL2	DILITHIUM2
DILITHIUM_LEVEL3	DILITHIUM3
DILITHIUM_LEVEL5	DILITHIUM5
SPHINCS_FAST_LEVEL1	SPHINCS+-SHAKE256-128f-simple
SPHINCS_FAST_LEVEL3	SPHINCS+-SHAKE256-192f-simple
SPHINCS_FAST_LEVEL5	SPHINCS+-SHAKE256-256f-simple
SPHINCS_SMALL_LEVEL1	SPHINCS+-SHAKE256-128s-simple
SPHINCS_SMALL_LEVEL3	SPHINCS+-SHAKE256-192s-simple
SPHINCS_SMALL_LEVEL5	SPHINCS+-SHAKE256-256s-simple

ポスト量子 KEM 命名規則：

wolfSSL バリアント名	PQC 提出バリアント名
KYBER_LEVEL1	KYBER512
KYBER_LEVEL3	KYBER768
KYBER_LEVEL5	KYBER1024

ポスト量子ハイブリッド KEM 命名規則：

wolfSSL バリアント名	NIST ECC 曲線と PQC 提出バリアント名
P256_KYBER_LEVEL1	ECDSA P-256 and KYBER512
P384_KYBER_LEVEL3	ECDSA P-384 and KYBER768
P521_KYBER_LEVEL5	ECDSA P-521 and KYBER1024

G.4 コードポイントと OID

当社がサポートする耐量子署名アルゴリズムと KEM は、OQS プロジェクトの OpenSSL フォークでもサポートされています。命名規則は当社のものとは異なりますが、同じ数値 OID とコードポイントを使用し、暗号化アーティファクトが同じライブラリ (liboqs) によって生成および処理されるという点で、完全な相互運用性があります。コードポイントは、TLS 1.3 の sigalgs およびサポートされるグループ拡張で使用されます。OID は、公開鍵、秘密鍵、署名の識別子として証明書と秘密鍵で使用されます。

TLS 1.3 のための量子コードポイント

wolfSSL バリアント名	コードポイント
FALCON_LEVEL1	65035
FALCON_LEVEL5	65038
DILITHIUM_LEVEL2	65184
DILITHIUM_LEVEL3	65187
DILITHIUM_LEVEL5	65189
KYBER_LEVEL1	570
KYBER_LEVEL3	572
KYBER_LEVEL5	573
P256_KYBER_LEVEL1	12090
P384_KYBER_LEVEL3	12092
P521_KYBER_LEVEL5	12093

証明書の Post-Quantum OID：

wolfSSL バリアント名	oid
FALCON_LEVEL1	1.3.9999.3.1
FALCON_LEVEL5	1.3.9999.3.4
DILITHIUM_LEVEL2	1.3.6.1.4.1.2.267.7.4.4
DILITHIUM_LEVEL3	1.3.6.1.4.1.2.267.7.6.5
DILITHIUM_LEVEL5	1.3.6.1.4.1.2.267.7.8.7
SPHINCS_FAST_LEVEL1	1.3.9999.6.7.4
SPHINCS_FAST_LEVEL3	1.3.9999.6.8.3
SPHINCS_FAST_LEVEL5	1.3.9999.6.9.3
SPHINCS_SMALL_LEVEL1	1.3.9999.6.7.10
SPHINCS_SMALL_LEVEL3	1.3.9999.6.8.7
SPHINCS_SMALL_LEVEL5	1.3.9999.6.9.7

G.5 暗号化アーティファクトサイズ

以下に示すサイズの単位はバイトです。

量子署名方式のアーティファクトサイズ：

wolfSSL バリアント名	公開鍵サイズ	秘密鍵サイズ	最大署名サイズ
FALCON_LEVEL1	897	1281	690
FALCON_LEVEL5	1793	2305	1330
DILITHIUM_LEVEL2	1312	2528	2420
DILITHIUM_LEVEL3	1952	4000	3293
DILITHIUM_LEVEL5	2592	4864	4595
SPHINCS_FAST_LEVEL1	32	64	17088
SPHINCS_FAST_LEVEL3	48	96	35664
SPHINCS_FAST_LEVEL5	64	128	49856
SPHINCS_SMALL_LEVEL1	32	64	7856
SPHINCS_SMALL_LEVEL3	48	96	16224
SPHINCS_SMALL_LEVEL5	64	128	29792

注：Falcon には、いくつかの署名サイズがあります。

耐量子 KEM アーティファクトのサイズ：

wolfSSL バリアント名	公開鍵サイズ	秘密鍵サイズ	暗号文サイズ	共有秘密のサイズ
KYBER_LEVEL1	800	1632	768	32
KYBER_LEVEL3	1184	2400	1088	32
KYBER_LEVEL5	1568	3168	1568	32
KYBER_90S_LEVEL1	800	1632	768	32
KYBER_90S_LEVEL3	1184	2400	1088	32
KYBER_90S_LEVEL5	1568	3168	1568	32

G.6 統計的データ

以下の統計とベンチマークは、Ubuntu 21.10 を実行している第 11 世代 Intel Core i7-1165G7@3-GHz(8 コア) で取得しました。liboqs は、`0.7.0` の古いコードとのコンパイラの非互換性のため、メインブランチで ba5b61a779a0db364f0e691a0a0bc8ac42e73f1b を使用しています。特記のない限り、構成オプションは以下のとおりです。

liboqs：

```
CFLAGS="-Os" cmake -DOQS_USE_OPENSSL=0 -DOQS_MINIMAL_BUILD="
    OQS_ENABLE_KEM_saber_saber;OQS_ENABLE_KEM_saber_lightsaber;
    OQS_ENABLE_KEM_saber_firesaber;OQS_ENABLE_KEM_kyber_1024;
    OQS_ENABLE_KEM_kyber_1024_90s;OQS_ENABLE_KEM_kyber_768;
    OQS_ENABLE_KEM_kyber_768_90s;OQS_ENABLE_KEM_kyber_512;
    OQS_ENABLE_KEM_kyber_512_90s;OQS_ENABLE_KEM_ntru_hps2048509;
    OQS_ENABLE_KEM_ntru_hps2048677;OQS_ENABLE_KEM_ntru_hps4096821;
    OQS_ENABLE_KEM_ntru_hrss701;OQS_ENABLE_SIG_falcon_1024;
    OQS_ENABLE_SIG_falcon_512;OQS_ENABLE_SIG_dilithium_2;
    OQS_ENABLE_SIG_dilithium_3;OQS_ENABLE_SIG_dilithium_5;
    OQS_ENABLE_SIG_dilithium_2_aes;OQS_ENABLE_SIG_dilithium_3_aes;
    OQS_ENABLE_SIG_dilithium_5_aes" ..
```

wolfssl：

```
./configure --with-liboqs \
            --disable-psk \
            --disable-shared \
            --enable-intelasm \
```

```
--enable-aesni \
--enable-sp-math-all \
--enable-sp-asm \
CFLAGS="-Os"
```

注：主に耐量子アルゴリズムをベンチマークしていますが、比較目的のために従来のアルゴリズムを残しています。

G.6.1 ランタイムバイナリサイズ

`tls_bench` サンプルプログラムのバイナリファイルは約 2.4MB、`--with-liboqs` を使用しない場合には 559kB です。約 1.9MB の違いがあります。

G.6.2 TLS 1.3 データ送信サイズ

サンプルサーバーとクライアントを実行し、送信されるすべての情報を Wireshark で記録することによって取得した値を以下に示します。これには、相互認証による TLS 1.3 ハンドシェイク、“hello wolfssl!”、“I hear you fa shizzle!” メッセージが含まれます。すべてのパケットの `tcp.len` を合計しました。

ciphersuite	認証	キー施設	合計バイト
TLS_AES_256_GCM_SHA384	RSA 2048 bit	ECC SECP256R1	5455
TLS_AES_256_GCM_SHA384	RSA 2048 bit	KYBER_LEVEL1	6633
TLS_AES_256_GCM_SHA384	RSA 2048 bit	KYBER_LEVEL3	7337
TLS_AES_256_GCM_SHA384	RSA 2048 bit	KYBER_LEVEL5	8201
TLS_AES_256_GCM_SHA384	RSA 2048 bit	KYBER_90S_LEVEL1	6633
TLS_AES_256_GCM_SHA384	RSA 2048 bit	KYBER_90S_LEVEL3	7337
TLS_AES_256_GCM_SHA384	RSA 2048 bit	KYBER_90S_LEVEL5	8201
TLS_AES_256_GCM_SHA384	RSA 2048 bit	P256_KYBER_LEVEL1	6763
TLS_AES_256_GCM_SHA384	RSA 2048 bit	P384_KYBER_LEVEL3	7531
TLS_AES_256_GCM_SHA384	RSA 2048 bit	P521_KYBER_LEVEL5	8467
TLS_AES_256_GCM_SHA384	RSA 2048 bit	P256_KYBER90S_LEVEL1	6763
TLS_AES_256_GCM_SHA384	RSA 2048 bit	P384_KYBER90S_LEVEL3	7531
TLS_AES_256_GCM_SHA384	RSA 2048 bit	P521_KYBER90S_LEVEL5	8467
TLS_AES_256_GCM_SHA384	FALCON_LEVEL1	ECC SECP256R1	6997
TLS_AES_256_GCM_SHA384	FALCON_LEVEL5	ECC SECP256R1	11248
TLS_AES_256_GCM_SHA384	FALCON_LEVEL1	KYBER_LEVEL1	8180
TLS_AES_256_GCM_SHA384	FALCON_LEVEL1	P256_KYBER_LEVEL1	8308
TLS_AES_256_GCM_SHA384	FALCON_LEVEL5	KYBER_LEVEL5	14007
TLS_AES_256_GCM_SHA384	FALCON_LEVEL5	P521_KYBER_LEVEL5	14257
TLS_AES_256_GCM_SHA384	DILITHIUM_LEVEL2	ECC SECP256R1	7918
TLS_AES_256_GCM_SHA384	DILITHIUM_LEVEL3	ECC SECP256R1	10233
TLS_AES_256_GCM_SHA384	DILITHIUM_LEVEL5	ECC SECP256R1	13477

G.6.3 ヒープとスタックの使用

これらの統計は、次の構成フラグを追加して取得しました。`--enable-trackmemory --enable-stacksize`

クライアントのサーバー認証なしのサーバー署名とクライアント検証、鍵交換用の TLS13-AES256-GCM-SHA384 暗号スイートおよび ECC SECP256R1 におけるメモリ使用量を以下に示します。

```
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
```

KEM グループのメモリ使用。サーバーのクライアント認証には TLS13-AES256-GCM-SHA384 暗号スイートおよび RSA-2048 を使用し、クライアントのサーバー認証は使用しません。

KEM グループのメモリ使用量を示します。サーバーのクライアント認証には TLS13-AES256-GCM-SHA384 と RSA-2048 を使用し、クライアントのサーバー認証は行いません。

```
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.6.4 Liboqs の KEMS のベンチマーク

操作	反復	トータルタイム (S)	時間 (米国)：平均	ポップ。stdev	CPU サイクル：平均	ポップ。st
Kyber512						
keygen	443212	3.000	6.769	3.282	20223	9715
encaps	339601	3.000	8.834	4.557	26411	13574
decaps	479954	3.000	6.251	3.594	18672	10669
Kyber768						
keygen	277967	3.000	10.793	5.490	32274	16375
encaps	225082	3.000	13.329	6.301	39871	18812
decaps	306782	3.000	9.779	5.063	29240	15097
Kyber1024						
keygen	216179	3.000	13.877	6.734	41513	20108
encaps	164469	3.000	18.241	8.353	54579	24968
decaps	217755	3.000	13.777	6.831	41210	20396
Kyber512-90s						
keygen	526948	3.000	5.693	2.795	17001	8235
encaps	380383	3.000	7.887	4.225	23570	12569
decaps	638653	3.000	4.697	2.896	14020	8543
Kyber768-90s						
keygen	394138	3.000	7.612	4.117	22746	12249
encaps	271196	3.000	11.062	5.881	33080	17557
decaps	424172	3.000	7.073	4.189	21132	12457
Kyber1024-90s						
keygen	278748	3.000	10.762	5.507	32182	16420
encaps	202208	3.000	14.836	7.486	44385	22368
decaps	299571	3.000	10.014	5.489	29945	16383

G.6.5 ベンチマーク

次のベンチマークは、次の設定フラグを使用して取得しました。

```
./configure --with-liboqs \
            --disable-psk \
            --disable-shared \
            --enable-intelasm \
            --enable-aesni \
            --enable-sp \
            --enable-sp-math \
            --enable-sp-asm \
            CFLAGS="-Os -DECC_USER_CURVES -DHAVE_ECC256 -DHAVE_ECC384"
```

G.6.5.1 wolfCrypt のベンチマーク **注**：シングルコアで測定したものです。

```
ECC   SECP256R1 key gen    95600 ops took 1.000 sec, avg 0.010 ms, 95555.939
   ops/sec
ECDHE SECP256R1 agree      26100 ops took 1.002 sec, avg 0.038 ms, 26038.522
   ops/sec
```

```
ECDSA SECP256R1 sign       63400 ops took 1.001 sec, avg 0.016 ms, 63320.787
    ops/sec
ECDSA SECP256R1 verify     24000 ops took 1.000 sec, avg 0.042 ms, 23994.983
    ops/sec
FALCON_level1   sign        5000 ops took 1.008 sec, avg 0.202 ms, 4961.637
    ops/sec
FALCON_level1   verify     27400 ops took 1.001 sec, avg 0.037 ms, 27361.394
    ops/sec
FALCON_level5   sign        2600 ops took 1.030 sec, avg 0.396 ms, 2523.187
    ops/sec
FALCON_level5   verify     14400 ops took 1.002 sec, avg 0.070 ms, 14376.179
    ops/sec
DILITHIUM_level2 sign      16200 ops took 1.003 sec, avg 0.062 ms, 16150.689
    ops/sec
DILITHIUM_level2 verify    44500 ops took 1.000 sec, avg 0.022 ms, 44478.388
    ops/sec
DILITHIUM_level3 sign      10200 ops took 1.002 sec, avg 0.098 ms, 10179.570
    ops/sec
DILITHIUM_level3 verify    27100 ops took 1.003 sec, avg 0.037 ms, 27017.485
    ops/sec
DILITHIUM_level5 sign       8400 ops took 1.009 sec, avg 0.120 ms, 8321.684
    ops/sec
DILITHIUM_level5 verify    17000 ops took 1.004 sec, avg 0.059 ms, 16933.788
    ops/sec
kyber_level1-kg           143608 ops took 1.000 sec, avg 0.007 ms, 143607.555
    ops/sec
kyber_level1-ed            64800 ops took 1.001 sec, avg 0.015 ms, 64725.835
    ops/sec
kyber_level3-kg            89790 ops took 1.000 sec, avg 0.011 ms, 89789.550
    ops/sec
kyber_level3-ed            42200 ops took 1.000 sec, avg 0.024 ms, 42190.886
    ops/sec
kyber_level5-kg            69362 ops took 1.000 sec, avg 0.014 ms, 69361.587
    ops/sec
kyber_level5-ed            31700 ops took 1.003 sec, avg 0.032 ms, 31606.130
    ops/sec
kyber90s_level1-kg        173655 ops took 1.000 sec, avg 0.006 ms, 173654.131
    ops/sec
kyber90s_level1-ed         77500 ops took 1.001 sec, avg 0.013 ms, 77424.888
    ops/sec
kyber90s_level3-kg        125138 ops took 1.000 sec, avg 0.008 ms, 125138.000
    ops/sec
kyber90s_level3-ed         55200 ops took 1.001 sec, avg 0.018 ms, 55153.726
    ops/sec
kyber90s_level5-kg         92773 ops took 1.000 sec, avg 0.011 ms, 92772.359
    ops/sec
kyber90s_level5-ed         39300 ops took 1.000 sec, avg 0.025 ms, 39283.188
    ops/sec
```

G.6.5.2 wolfSSL のベンチマーク

注：2 コアで測定したものです。

```
wolfSSL Server Benchmark on TLS13-AES256-GCM-SHA384 with group ECC_SECP256R1:
    Total       : 209715200 bytes
    Num Conns   :       801
```

```
    Rx Total    :   238.549 ms
    Tx Total    :    80.893 ms
    Rx          :   419.200 MB/s
    Tx          :  1236.204 MB/s
    Connect     :   552.092 ms
    Connect Avg :     0.689 ms
wolfSSL Client Benchmark on TLS13-AES256-GCM-SHA384 with group ECC_SECP256R1:
    Total       : 209715200 bytes
    Num Conns   :       801
    Rx Total    :   264.171 ms
    Tx Total    :    77.399 ms
    Rx          :   378.542 MB/s
    Tx          :  1292.002 MB/s
    Connect     :   550.630 ms
    Connect Avg :     0.687 ms

wolfSSL Server Benchmark on TLS13-AES256-GCM-SHA384 with group ECC_SECP384R1:
    Total       : 164626432 bytes
    Num Conns   :       629
    Rx Total    :   207.183 ms
    Tx Total    :    68.783 ms
    Rx          :   378.892 MB/s
    Tx          :  1141.270 MB/s
    Connect     :   508.584 ms
    Connect Avg :     0.809 ms
wolfSSL Client Benchmark on TLS13-AES256-GCM-SHA384 with group ECC_SECP384R1:
    Total       : 164626432 bytes
    Num Conns   :       629
    Rx Total    :   228.902 ms
    Tx Total    :    65.852 ms
    Rx          :   342.942 MB/s
    Tx          :  1192.073 MB/s
    Connect     :   506.299 ms
    Connect Avg :     0.805 ms

wolfSSL Server Benchmark on TLS13-AES256-GCM-SHA384 with group FFDHE_2048:
    Total       : 125829120 bytes
    Num Conns   :       481
    Rx Total    :   158.742 ms
    Tx Total    :    53.102 ms
    Rx          :   377.971 MB/s
    Tx          :  1129.896 MB/s
    Connect     :   579.937 ms
    Connect Avg :     1.206 ms
wolfSSL Client Benchmark on TLS13-AES256-GCM-SHA384 with group FFDHE_2048:
    Total       : 125829120 bytes
    Num Conns   :       481
    Rx Total    :   175.313 ms
    Tx Total    :    50.565 ms
    Rx          :   342.245 MB/s
    Tx          :  1186.597 MB/s
    Connect     :   582.023 ms
    Connect Avg :     1.210 ms
```

```
wolfSSL Server Benchmark on TLS13-AES256-GCM-SHA384 with group KYBER_LEVEL1:
    Total       : 225968128 bytes
    Num Conns   :       863
    Rx Total    :   258.872 ms
    Tx Total    :    87.586 ms
    Rx          :   416.229 MB/s
    Tx          :  1230.220 MB/s
    Connect     :   580.184 ms
    Connect Avg :     0.672 ms
wolfSSL Client Benchmark on TLS13-AES256-GCM-SHA384 with group KYBER_LEVEL1:
    Total       : 225968128 bytes
    Num Conns   :       863
    Rx Total    :   285.086 ms
    Tx Total    :    84.362 ms
    Rx          :   377.956 MB/s
    Tx          :  1277.233 MB/s
    Connect     :   574.039 ms
    Connect Avg :     0.665 ms

wolfSSL Server Benchmark on TLS13-AES256-GCM-SHA384 with group KYBER_LEVEL3:
    Total       : 214171648 bytes
    Num Conns   :       818
    Rx Total    :   241.450 ms
    Tx Total    :    80.798 ms
    Rx          :   422.965 MB/s
    Tx          :  1263.960 MB/s
    Connect     :   603.945 ms
    Connect Avg :     0.738 ms
wolfSSL Client Benchmark on TLS13-AES256-GCM-SHA384 with group KYBER_LEVEL3:
    Total       : 214171648 bytes
    Num Conns   :       818
    Rx Total    :   263.357 ms
    Tx Total    :    81.142 ms
    Rx          :   387.781 MB/s
    Tx          :  1258.593 MB/s
    Connect     :   596.085 ms
    Connect Avg :     0.729 ms

wolfSSL Server Benchmark on TLS13-AES256-GCM-SHA384 with group KYBER_LEVEL5:
    Total       : 206307328 bytes
    Num Conns   :       788
    Rx Total    :   249.636 ms
    Tx Total    :    84.465 ms
    Rx          :   394.073 MB/s
    Tx          :  1164.683 MB/s
    Connect     :   589.028 ms
    Connect Avg :     0.747 ms
wolfSSL Client Benchmark on TLS13-AES256-GCM-SHA384 with group KYBER_LEVEL5:
    Total       : 206307328 bytes
    Num Conns   :       788
    Rx Total    :   276.059 ms
    Tx Total    :    81.856 ms
    Rx          :   356.355 MB/s
    Tx          :  1201.798 MB/s
```

```
    Connect     :   580.463 ms
    Connect Avg :     0.737 ms

wolfSSL Server Benchmark on TLS13-AES256-GCM-SHA384 with group
   KYBER_90S_LEVEL1:
    Total       : 226754560 bytes
    Num Conns   :       866
    Rx Total    :   249.504 ms
    Tx Total    :    86.285 ms
    Rx          :   433.360 MB/s
    Tx          :  1253.120 MB/s
    Connect     :   590.655 ms
    Connect Avg :     0.682 ms
wolfSSL Client Benchmark on TLS13-AES256-GCM-SHA384 with group
   KYBER_90S_LEVEL1:
    Total       : 226754560 bytes
    Num Conns   :       866
    Rx Total    :   274.258 ms
    Tx Total    :    83.674 ms
    Rx          :   394.246 MB/s
    Tx          :  1292.214 MB/s
    Connect     :   585.395 ms
    Connect Avg :     0.676 ms

wolfSSL Server Benchmark on TLS13-AES256-GCM-SHA384 with group
   KYBER_90S_LEVEL3:
    Total       : 208666624 bytes
    Num Conns   :       797
    Rx Total    :   253.840 ms
    Tx Total    :    86.227 ms
    Rx          :   391.979 MB/s
    Tx          :  1153.925 MB/s
    Connect     :   584.268 ms
    Connect Avg :     0.733 ms
wolfSSL Client Benchmark on TLS13-AES256-GCM-SHA384 with group
   KYBER_90S_LEVEL3:
    Total       : 208666624 bytes
    Num Conns   :       797
    Rx Total    :   279.104 ms
    Tx Total    :    83.607 ms
    Rx          :   356.499 MB/s
    Tx          :  1190.096 MB/s
    Connect     :   580.950 ms
    Connect Avg :     0.729 ms

wolfSSL Server Benchmark on TLS13-AES256-GCM-SHA384 with group
   KYBER_90S_LEVEL5:
    Total       : 205783040 bytes
    Num Conns   :       786
    Rx Total    :   255.324 ms
    Tx Total    :    85.233 ms
    Rx          :   384.316 MB/s
    Tx          :  1151.260 MB/s
    Connect     :   583.899 ms
```

```
    Connect Avg :     0.743 ms
wolfSSL Client Benchmark on TLS13-AES256-GCM-SHA384 with group
   KYBER_90S_LEVEL5:
    Total       : 205783040 bytes
    Num Conns   :       786
    Rx Total    :   281.997 ms
    Tx Total    :    82.461 ms
    Rx          :   347.964 MB/s
    Tx          :  1189.958 MB/s
    Connect     :   579.312 ms
    Connect Avg :     0.737 ms

wolfSSL Server Benchmark on TLS13-AES256-GCM-SHA384 with group
   P256_KYBER_LEVEL1:
    Total       : 182190080 bytes
    Num Conns   :       696
    Rx Total    :   219.789 ms
    Tx Total    :    75.536 ms
    Rx          :   395.266 MB/s
    Tx          :  1150.114 MB/s
    Connect     :   641.859 ms
    Connect Avg :     0.922 ms
wolfSSL Client Benchmark on TLS13-AES256-GCM-SHA384 with group
   P256_KYBER_LEVEL1:
    Total       : 182190080 bytes
    Num Conns   :       696
    Rx Total    :   241.393 ms
    Tx Total    :    72.367 ms
    Rx          :   359.890 MB/s
    Tx          :  1200.483 MB/s
    Connect     :   581.373 ms
    Connect Avg :     0.835 ms

wolfSSL Server Benchmark on TLS13-AES256-GCM-SHA384 with group
   P384_KYBER_LEVEL3:
    Total       : 133431296 bytes
    Num Conns   :       510
    Rx Total    :   152.666 ms
    Tx Total    :    53.693 ms
    Rx          :   416.760 MB/s
    Tx          :  1184.982 MB/s
    Connect     :   743.577 ms
    Connect Avg :     1.458 ms
wolfSSL Client Benchmark on TLS13-AES256-GCM-SHA384 with group
   P384_KYBER_LEVEL3:
    Total       : 133431296 bytes
    Num Conns   :       510
    Rx Total    :   169.131 ms
    Tx Total    :    50.632 ms
    Rx          :   376.188 MB/s
    Tx          :  1256.605 MB/s
    Connect     :   611.105 ms
    Connect Avg :     1.198 ms
```

```
wolfSSL Server Benchmark on TLS13-AES256-GCM-SHA384 with group
    P256_KYBER_90S_LEVEL1:
    Total       : 191102976 bytes
    Num Conns   :       730
    Rx Total    :   211.835 ms
    Tx Total    :    72.819 ms
    Rx          :   430.170 MB/s
    Tx          :  1251.386 MB/s
    Connect     :   651.010 ms
    Connect Avg :     0.892 ms
wolfSSL Client Benchmark on TLS13-AES256-GCM-SHA384 with group
    P256_KYBER_90S_LEVEL1:
    Total       : 191102976 bytes
    Num Conns   :       730
    Rx Total    :   233.104 ms
    Tx Total    :    70.994 ms
    Rx          :   390.919 MB/s
    Tx          :  1283.561 MB/s
    Connect     :   589.063 ms
    Connect Avg :     0.807 ms

wolfSSL Server Benchmark on TLS13-AES256-GCM-SHA384 with group
    P384_KYBER_90S_LEVEL3:
    Total       : 136052736 bytes
    Num Conns   :       520
    Rx Total    :   168.780 ms
    Tx Total    :    57.603 ms
    Rx          :   384.376 MB/s
    Tx          :  1126.236 MB/s
    Connect     :   723.880 ms
    Connect Avg :     1.392 ms
wolfSSL Client Benchmark on TLS13-AES256-GCM-SHA384 with group
    P384_KYBER_90S_LEVEL3:
    Total       : 136052736 bytes
    Num Conns   :       520
    Rx Total    :   189.078 ms
    Tx Total    :    52.841 ms
    Rx          :   343.112 MB/s
    Tx          :  1227.747 MB/s
    Connect     :   594.282 ms
    Connect Avg :     1.143 ms
```

次のベンチマークは、次の設定フラグを使用して取得しました。

```
./configure --with-liboqs \
            --disable-psk \
            --disable-shared \
            --enable-intelasm \
            --enable-aesni \
            --enable-sp \
            --enable-sp-math-all \
            CFLAGS="-Os -DECC_USER_CURVES -DHAVE_ECC521"
```

注：2 コアで測定したものです。

```
wolfSSL Server Benchmark on TLS13-AES256-GCM-SHA384 with group ECC_SECP521R1:
```

```
    Total       :  22806528 bytes
    Num Conns   :        88
    Rx Total    :    29.526 ms
    Tx Total    :     9.423 ms
    Rx          :   368.325 MB/s
    Tx          :  1154.060 MB/s
    Connect     :   447.201 ms
    Connect Avg :     5.082 ms
wolfSSL Client Benchmark on TLS13-AES256-GCM-SHA384 with group ECC_SECP521R1:
    Total       :  22806528 bytes
    Num Conns   :        88
    Rx Total    :    32.363 ms
    Tx Total    :     9.206 ms
    Rx          :   336.028 MB/s
    Tx          :  1181.257 MB/s
    Connect     :   442.915 ms
    Connect Avg :     5.033 ms

wolfSSL Server Benchmark on TLS13-AES256-GCM-SHA384 with group
   P521_KYBER_LEVEL5:
    Total       :  10747904 bytes
    Num Conns   :        42
    Rx Total    :     8.199 ms
    Tx Total    :    30.942 ms
    Rx          :   625.096 MB/s
    Tx          :   165.633 MB/s
    Connect     :   958.292 ms
    Connect Avg :    22.816 ms
wolfSSL Client Benchmark on TLS13-AES256-GCM-SHA384 with group
   P521_KYBER_LEVEL5:
    Total       :  10747904 bytes
    Num Conns   :        42
    Rx Total    :     9.919 ms
    Tx Total    :     3.685 ms
    Rx          :   516.689 MB/s
    Tx          :  1390.684 MB/s
    Connect     :   679.437 ms
    Connect Avg :    16.177 ms

wolfSSL Server Benchmark on TLS13-AES256-GCM-SHA384 with group
   P521_KYBER_90S_LEVEL5:
    Total       :  13107200 bytes
    Num Conns   :        51
    Rx Total    :    19.132 ms
    Tx Total    :     6.887 ms
    Rx          :   326.680 MB/s
    Tx          :   907.481 MB/s
    Connect     :   976.107 ms
    Connect Avg :    19.139 ms
wolfSSL Client Benchmark on TLS13-AES256-GCM-SHA384 with group
   P521_KYBER_90S_LEVEL5:
    Total       :  13107200 bytes
    Num Conns   :        51
    Rx Total    :    23.578 ms
```

```
Tx Total    :     5.039 ms
Rx          :   265.078 MB/s
Tx          :  1240.273 MB/s
Connect     :   673.107 ms
Connect Avg :    13.198 ms
```

G.7 ドキュメンテーション

技術文書や既知の回答テストなどのその他のリソースは、NIST PQC Web サイトにあります。

https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions

アルゴリズム固有のベンチマーク情報については、OQS プロジェクトの Web サイトに掲載されています。

https://openquantumsafe.org/benchmarking/

G.8 ポスト量子ステートフルハッシュベース署名

このセクションでは、最近 wolfSSL がサポートを開始した LMS/HSS などのポスト量子ステートフルハッシュベース署名 (HBS) スキームについて記します。

G.8.1 動機づけ

ステートフル HBS スキームは、さまざまな理由から関心が高まっています。ステートフル HBS スキームの主な目的は、量子セキュリティの強化です。前述したように、ショアのアルゴリズムにより、量子コンピューターは大きな整数を効率的に因数分解し、離散対数を計算することができます。これによって、RSA や ECC などの公開鍵暗号スキームを完全に破ることができます。

対照的に、ステートフル HBS スキームは、その基礎となるハッシュ関数とマークルツリー (通常 SHA256 で実装) のセキュリティに基づいており、暗号に関連する量子コンピューターの登場によって破られることは予想されていません。これらの理由から、ステートフル HBS スキームは NIST SP 800-208 および NSA の CNSA 2.0 スイートで推奨されています。詳細については、次の 2 つのリンクをご参照ください。

- 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

さらに CNSA 2.0 のタイムラインでは、2030 年までにポスト量子ステートフル HBS スキームのみを使用する必要があり、採用は「直ちに」開始する必要があると規定されています。LMS の採用は、CNSA 2.0 スイートのタイムラインで最も早い要件です。

ただし、ステートフル HBS スキームの性質上、その使用と状態の追跡には細心の注意を払う必要があります。ステートフル HBS システムでは、秘密鍵は実際にはワンタイム署名 (OTS) キーの有限セットであり、再利用されることはありません。同じ OTS キーを使用して 2 つの異なるメッセージを署名すると、攻撃者が署名を偽造できる可能性があり、スキーム全体のセキュリティが崩壊します。したがって、ステートフル HBS スキームは、パブリックインターネットなどの一般的な使用には適していません。

LMS/HSS などのステートフル HBS スキームは、特に長い運用寿命が期待され、暗号に関連する量子コンピューターに対して耐性が求められる組み込みシステムや制約付きシステムでのオフラインファームウェア認証と署名検証に特に役立ちます。

G.8.2 LMS/HSS 署名

wolfSSL は、wolfCrypt 組み込み暗号エンジンに LMS/HSS ハッシュベースの署名スキームのサポートを追加しています。これは、以前の libOQS 統合と同様に、hash-sigsLMS/HSS ライブラリ (https://github.com/cisco/hash-sigs) との実験的な統合によって実現されます。

Leighton-Micali Signatures(LMS) とそのマルチツリーのバリアントである Hierarchical Signature System(HSS) は、ポスト量子、ステートフルハッシュベース署名スキームです。公開鍵と秘密鍵が小さく、署名と検証が速いことで知られています。署名のサイズは大きくなりますが、Winternitz パラメーターを介して調整できます。詳細については、RFC8554 の次の 2 つのリンクを参照してください：

- LMS: https://datatracker.ietf.org/doc/html/rfc8554
- HSS: https://datatracker.ietf.org/doc/html/rfc8554#section-6

前述したように、LMS/HSS 署名システムは有限数のワンタイム署名 (OTS) 鍵で構成されているため、有限数の署名しか安全に生成できません。ただし、署名の数と署名のサイズは、次に説明する一連の定義済みパラメーターを介して調整できます。

G.8.2.1 サポートしているパラメータ LMS/HSS 署名は 3 つのパラメータによって定義されます。- levels: マークルツリーのレベル数 - height: 個々のマークルツリーの高さ - Winternitz: ウィンターニッツチェーンで使用されるハッシュのビット数。署名サイズの時空間トレードオフとして使用されます。

wolfSSL は、RFC8554 で定義されているすべての LMS/HSS パラメータをサポートします：

- levels = {1..8}
- height = {5, 10, 15, 20, 25}
- Winternitz = {1, 2, 4, 8}

利用可能な署名の数: - N = 2 ** (levels * height)

便宜上、一部のパラメータセットは列挙型 `wc_LmsParm` で事前定義されています。その値を以下の表に示します：

パラメータセット	意味
WC_LMS_PARM_NONE	設定されていません。デフォルトを使用します (WC_LMS_PARM_L1_H15_W2)
WC_LMS_PARM_L1_H15_W2	level:1, height:15, Winternitz:2
WC_LMS_PARM_L1_H15_W4	level:1, height:15, Winternitz:4
WC_LMS_PARM_L2_H10_W2	level:2, height:10, Winternitz:2
WC_LMS_PARM_L2_H10_W4	level:2, height:10, Winternitz:4
WC_LMS_PARM_L2_H10_W8	level:2, height:10, Winternitz:8
WC_LMS_PARM_L3_H5_W2	level:3, height:5, Winternitz:2
WC_LMS_PARM_L3_H5_W4	level:3, height:5, Winternitz:4
WC_LMS_PARM_L3_H5_W8	level:3, height:5, Winternitz:8
WC_LMS_PARM_L3_H10_W4	level:3, height:10, Winternitz:4
WC_LMS_PARM_L4_H5_W8	level:4, height:5, Winternitz:8

ここで設定したパラメータに対する署名のサイズと署名の数を表示します：

パラメータセット	署名サイズ	署名数
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

表からわかるように、署名のサイズは主にレベルと Winternitz 値、および比較的影響は小さいですが高さによって決まります。

- レベル値を大きくすると、署名サイズは大幅に増加します。
- 高さの値を大きくすると、署名のサイズは増加します。
- Winternitz 値を大きくすると、署名のサイズは小さくなりますが、鍵の生成と署名/検証にかかる時間が長くなります。

鍵の生成時間は、第 1 レベルのツリーの高さによって大きく決まります。使用可能な署名の数が同じであっても、レベル 3、高さ 5 のツリーは、初期鍵生成時にレベル 1、高さ 15 のツリーよりもはるかに高速です

G.8.2.2 LMS/HSS ビルド方法 wolfSSL リポジトリの INSTALL ファイル (https://github.com/wolfSSL/wolfssl/blob/mas
を参照してください。項目 17(LMS/HSS サポートのための hash-sigs ライブラリを使用した構築 [実験]) には、wolfSSL と hash-sigs LMS/HSS ライブラリを設定および構築する方法についての手順が記載されています。

G.8.2.3 ベンチマークデータ 次のベンチマークデータは、Fedora 38(`6.2.9-300.fc38.x86_64`) 上の 8 コア Intel i7-8700 CPU@3.20GHz で取得されました。マルチスレッドの例では 4 スレッドと 4 コアが使用されましたが、シングルスレッドの例では 1 コアのみが使用されました。

INSTALL ファイルの項目 17 で説明したように、hash-sigs ライブラリは 2 つの静的ライブラリを提供します。- `hss_lib.a`: シングルスレッド - `hss_lib_thread.a`: マルチスレッド

マルチスレッドバージョンではワーカースレッドが生成され、鍵生成などの CPU を集中的に使用するタスクが高速化されます。これにより、主にすべてのパラメータ値に対する鍵の生成と署名が高速化され、より大きなレベル値の検証も多少高速化されます。

なお、以下のベンチマークは次の構成オプションを有効化して取得しました。

```
  ./configure \
    --enable-static \
    --disable-shared \
    --enable-lms=yes \
    --with-liblms=<path to hash sigs install>
```

マルチスレッドベンチマーク

以下は、集中的なタスクを並列化するために 4 スレッドを使用し、4 コアを使用したマルチスレッド `hss_lib_thread.a` と共にビルドして取得したベンチマークデータです。

```
./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
```

シングルスレッドベンチマーク

以下は、シングルスレッドの hss_lib.a と共にビルドして取得したベンチマークデータです これは単一のコアのみを使用します。

```
$ ./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.8.3 XMSS/XMSS^MT 署名

wolfSSL は、XMSS/XMSS^MT ステートフルハッシュベース署名のサポートを追加しています。LMS と同様に、これは RFC 8391 (https://www.rfc-editor.org/rfc/rfc8391.html) の xmss-reference リポジトリ (https://github.com/XMSS/xmss-reference.git) との実験的な統合によって実現されています。

xmss-reference は、xmss_core_fast および xmss_core 実装をサポートしています。xmss_core_fast 実装は、トレードオフとしてより大きな秘密鍵サイズでパフォーマンスを優先するように設計されています。

当社の統合では xmss_core_fast を使用していますが、パッチが適用されているため、代わりに wolfCrypt SHA256 実装を使用できます。

パッチは、wolfssl-examples リポジトリの

で公開しています。https://github.com/wolfSSL/wolfssl-examples

全体的に、XMSS/XMSS^MTはLMS/HSSに似ています。より詳細な比較については、「LMS vs XMSS: 2 つのハッシュベース署名標準の比較」(https://eprint.iacr.org/2017/349.pdf) を参照してください。

XMSS^MTはXMSSのマルチツリー一般化であり、HSS with LMSに似ていますが、XMSS/XMSS^MTではWinternitz値が `w=16` に固定されている点が異なります。公開鍵はXMSS/XMSS^MTでは若干大きくなり(XMSS/XMSS^MTでは68 バイト、LMS/HSSでは60バイト)、署名は若干小さくなります。

サポートしているパラメータ

wolfSSLは、NIST SP 800-208 (https://csrc.nist.gov/pubs/sp/800/208/final) の表10および11のSHA256 XMSS/XMSS^MTパラメータセットをサポートしています。

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

上記の表で、`n` は HASH 関数のバイト数、`w` はWinternitz値、`h` はツリーシステムの合計高さ、`d` はツリーのレベルを示します。

鍵生成時間は第1レベルのツリーの高さ(または`h/d`) によって大きく左右されますが、署名の長さは主に`d` (ハイパーツリーレベルの数)とともに増加します。

LMS/HSSと同様に、使用可能な署名の数は `2**h` に比例して増加します。ここで `h` はツリー システムの合計高さです。

ベンチマークデータ

以下では、Intel x86_64およびaarch64のいくつかのXMSS/XMSS^MTパラメータセットのベンチマーク データを示します。これらのシステムでのSHA256パフォーマンスも参考として記載されています。必要なハッシュチェーンの数が多いため、XMSS/XMSS^MTのCPU 作業の大部分が計算されるためです。さらに、xmss-referenceへのパッチはwolfCryptのSHA256実装を置き換えるため、同じASMの高速化のメリットが得られます。

前述のように、XMSS統合ではxmss-referenceの`xmss_core_fast`実装を使用しています。この実装は、秘密鍵のサイズが大きいというトレードオフで、より高速なパフォーマンスを実現します。

x86_64

以下のx86_64ベンチマークデータは、Fedora 38 (`6.2.9-300.fc38.x86_64`) が動作するIntel i7-8700 CPU @ 3.20GHz(8コア)で取得しました。このCPUには`avx avx2`フラグがあり、`--enable-intelasm` を有効化することでハッシュ操作を高速化できます。

`--enable-intelasm` を使用する場合

```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
```

--enable-intelasm を使用しない場合

```
$./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

以下の aarch64 データは、CPU フラグ sha1 sha2 sha3 sha512 を使用して Apple M1 上で実行されている Ubuntu(5.15.0-71-generic) で取得されました。--enable-armasm を使用してビルドすると、特に SHA ハッシュ操作が大幅に高速化されます。

--enable-armasm を使用する場合

```
$ ./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
```

--enable-armasm を使用しない場合

```
$ ./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
```

H　wolfSSL 移植ガイド

H.1　目的

このガイドは、軽量 SSL/TLS ライブラリ wolfSSL を新しい組み込みプラットフォーム、オペレーティングシステム、トランスポートメディア (TCP/IP、Bluetooth など) に移植するエンジニア向けのリファレンスを提供します。wolfSSL を移植する際に通常変更が必要となる wolfSSL コードベースの領域を説明しています。本稿はあくまで「ガイド」で、技術の進化に伴い変更が必要になることもあります。不足していると感じる箇所があれば、気兼ねなくお知らせください。

H.2　対象読者

このガイドは、wolfSSL および wolfCrypt を、現在サポートされていない新しいプラットフォームまたは環境に移植するエンジニアを対象としています。

H.3　序章

組み込みプラットフォームで wolfSSL を実行するには、いくつかの手順を繰り返す必要があります。これらの手順の一部は、2.4 章 で概説しています。

wolfSSL ドキュメント第 2 章に示した手順に加えて、特定のプラットフォームに対応するために移植または変更が必要なコード領域があります。wolfSSL はこれらの領域の多くを抽象化して、新しいプラットフォームへできるだけ簡単に移植できるようにしています。

`./wolfssl/wolfcrypt/settings.h` ファイルには、さまざまなオペレーティングシステム、TCP/IP スタック、およびチップセット (例: MBED、FREESCALE_MQX、MICROCHIP_PIC32、MICRIUM、EBSNET など) に固有の定義がいくつかあります。wolfSSL をコンパイルして新しいプラットフォームに移植するときに、`#defines` を配置する主な場所は 2 つあります。

1. オペレーティングシステムまたは TCP/IP スタックに対応するための新たなマクロ定義は、通常、wolfSSL のポーティングが完了すると、`settings.h` ファイルに追加しています。これにより、機能のオン/オフを簡単に切り替えたり、そのビルドの「デフォルト」となるビルド設定をカスタマイズしたりできます。wolfSSL を新しいプラットフォームに移植する際にも、このファイルに新しいカスタム定義を追加することでお使いいただけます。新しいプラットフォームへの移植が完了した際，もし差し支えなければ，wolfSSL の Git リポジトリ に Pull Request を送信していただけると嬉しく思います。これにより、より多くの環境で wolfSSL を使用しやすくなります。
2. wolfSSL 自体に変更を加えたくない場合、または追加のプリプロセッサ定義を使用して wolfSSL ビルドをカスタマイズしたい場合、wolfSSL はカスタムヘッダーファイル `user_settings.h` の使用を推奨します。wolfSSL ソースファイルをコンパイルするときに `WOLFSSL_USER_SETTINGS` が定義されている場合、wolfSSL はカスタムヘッダーファイル `user_settings.h` を自動的にインクルードします。このヘッダーはユーザーが作成し、インクルードパスに配置する必要があります。これにより、ユーザーは wolfSSL ビルド用に 1 つのファイルのみを管理すればよく、wolfSSL の新しいバージョンへの更新がはるかに簡単になります。

wolfSSL では、直接のメール (info@wolfssl.jp) または GitHub Pull Request を通じてパッチとコード変更をご送信いただくことを推奨しています。

H.4　wolfSSL の移植

H.4.1　データ型

Q: どんな場合にこの章の内容が役立ちますか?

A: プラットフォームに適したデータ型のサイズを設定することは常に重要です。

wolfSSL は、64 ビット型を利用できることで速度面でメリットを得ています。プラットフォームの sizeof(long) と sizeof(long long) の結果と一致するように SIZEOF_LONG と SIZEOF_LONG_LONG を定義します。これは、settings.h または user_settings.h のカスタム定義に追加できます。たとえば、MY_NEW_PLATFORM のサンプル定義の下の settings.h では次のように示します。

```
#ifdef MY_NEW_PLATFORM
    #define SIZEOF_LONG 4
    #define SIZEOF_LONG_LONG 8
...
#endif
```

wolfSSL と wolfCrypt では、word32 と word16 という 2 つの追加データ型を使用します。これらのデフォルトの型マッピングは次のとおりです。

```
#ifndef WOLFSSL_TYPES
#ifndef byte
    typedef unsigned char byte
#endif
    typedef unsigned short word16;
    typedef unsigned int word32;
    typedef byte word24[3];
#endif
```

word32 はコンパイラの 32 ビット型にマッピングされ、word16 はコンパイラの 16 ビット型にマッピングされます。これらのデフォルトのマッピングがプラットフォームに適していない場合は、settings.h または user_settings.h で WOLFSSL_TYPES を定義し、word32 および word16 に独自のカスタム typedef を割り当てる必要があります。

wolfSSL の fastmath ライブラリは、fp_digit および fp_word 型を使用します。デフォルトでは、これらはビルド構成に応じて <wolfssl/wolfcrypt/tfm.h> にマッピングされます。

fp_word は fp_digit の 2 倍のサイズである必要があります。デフォルトのケースがプラットフォームに当てはまらない場合は、settings.h または user_settings.h で WOLFSSL_BIGINT_TYPES を定義し、fp_word および fp_digit に独自のカスタム typedef を割り当てる必要があります。

wolfSSL は、一部の操作において使用可能な場合は 64 ビット型を使用します。ビルド時には、SIZEOF_LONG および SIZEOF_LONG_LONG の設定に基づいて、word64 の正しいデータ型を検出して設定しようとします。真の 64 ビット型を持たない一部のプラットフォームでは、2 つの 32 ビット型を合わせて使用するため、パフォーマンスが低下する可能性があります。コンパイル時に NO_64BIT を定義することで，64 ビット型を使用しないこともできます。

H.4.2 エンディアン

Q: どんな場合にこの章の内容が役立ちますか?

A: あなたのプラットフォームがビッグエンディアンの場合です。

お使いのプラットフォームはビッグエンディアンとリトルエンディアン，どちらでしょうか．wolfSSL はデフォルトでリトルエンディアンを使用しています。システムがビッグエンディアンの場合は、wolfSSL をビルドする際に BIG_ENDIAN_ORDER を定義します。例えば，settings.h で次のように設定できます．

```
#ifdef MY_NEW_PLATFORM
    ...
    #define BIG_ENDIAN_ORDER
    ...
#endif
```

H.4.3 writev

Q: どんな場合にこの章の内容が役立ちますか?

A: <sys/uio.h> を利用できない場合です。

デフォルトでは、wolfSSL API は、writev() セマンティクスをシミュレートする wolfSSL_writev() をアプリケーションに提供します。<sys/uio.h> ヘッダーが利用できないシステムでは、NO_WRITEV を定義してこの機能を除外します。

H.4.4 ネットワーク I/O

Q: どんな場合にこの章の内容が役立ちますか?

A: BSD スタイルのソケット API を利用できない、あるいはカスタムトランスポート層または TCP/IP スタックを使用している、または静的バッファーのみを使用したい場合です。

wolfSSL はデフォルトで BSD スタイルのソケットインターフェイスを使用します。トランスポート層が BSD ソケットインターフェイスを提供する場合、カスタムヘッダーが必要でない限り、wolfSSL はそのまま統合する必要があります。

wolfSSL はカスタム I/O 抽象化レイヤーを提供し、ユーザーは wolfSSL の I/O 機能をシステムに合わせて調整できます。詳細については、5.1.2 節 をご参照ください。

具体的には WOLFSSL_USER_IO を定義し、wolfSSL デフォルトの EmbedSend() と EmbedReceive() をテンプレートとして使用し，独自の I/O コールバック関数を記述します。これら 2 つの関数は ./src/io.c にあります。

wolfSSL は、入力と出力に動的バッファを使用します。このバッファのデフォルトは 0 バイトです。バッファよりも大きいサイズの入力レコードを受信した場合、動的バッファが一時的にリクエストの処理に使用され、その後解放されます。

動的メモリを必要としない 16kB の大きな静的バッファを使用する場合は、LARGE_STATIC_BUFFERS を定義することでこのオプションを使用できます。

動的バッファが使用され、ユーザーがバッファサイズよりも大きい wolfSSL_write() を要求した場合、最大 MAX_RECORD_SIZE までの動的ブロックを使用してデータを送信します。RECORD_SIZE で定義されている現在のバッファサイズの最大のチャンクでのみデータを送信したいユーザーは、STATIC_CHUNKS_ONLY を定義することでこれを実現できます。この定義を使用する場合、RECORD_SIZE はデフォルトで 128 バイトになります。

H.4.5 ファイルシステム

Q: どんな場合にこの章の内容が役立ちますか?

A: ファイルシステムを利用できない、あるいは標準のファイルシステム機能を利用できない、または独自のファイルシステムを使用している場合です。

wolfSSL は、TLS セッションまたはコンテキストに鍵と証明書をロードするためにファイルシステムを使用します。wolfSSL では、これらをメモリバッファからロードすることもできます。メモリバッファのみを使用する場合、ファイルシステムは必要ありません。

ライブラリをビルドするときに NO_FILESYSTEM を定義することで、wolfSSL によるファイルシステムの使用を無効にできます。つまり、証明書と鍵はファイルではなくメモリバッファからロードする必要があります。settings.h で，以下のように設定できます.

```
#ifdef MY_NEW_PLATFORM
    ...
    #define NO_FILESYSTEM
```

```
    ...
#endif
```

テスト用の鍵と証明書バッファーは、./wolfssl/certs_test.h ヘッダーファイルにあります。これらは、./certs ディレクトリにある対応する証明書や鍵と一致します。

certs_test.h ヘッダーファイルは、必要に応じて ./gencertbuf.pl スクリプトを使用して更新できます。gencertbuf.pl 内には、fileList_1024 と fileList_2048 の 2 つの配列があります。鍵サイズに応じて、追加の証明書またはキーをそれぞれの配列に追加できます。なお，DER 形式である必要があります。上記の配列は、証明書・鍵ファイルの場所を目的のバッファー名にマップします。gencertbuf.pl を変更した後、wolfSSL ルートディレクトリから実行すると、./wolfssl/certs_test.h の証明書・鍵のバッファーが更新されます。

```
./gencertbuf.pl
```

デフォルト以外のファイルシステムを使用する場合、ファイルシステム抽象化レイヤーは ./wolfssl/wolfcrypt/wc_port.h にあります。ここには、EBSNET、FREESCALE_MQX、MICRIUM などのさまざまなプラットフォームのファイルシステムポートがあります。必要に応じて、プラットフォームのカスタム定義を追加できます。これにより、XFILE、XFOPEN、XFSEEK などを使用してファイルシステム関数を定義できます。たとえば、Micrium の µC/OS (MICRIUM) の wc_port.h のファイルシステムレイヤーは次のとおりです。

```
#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 スレッド化

Q: どんな場合にこの章の内容が役立ちますか?

A: マルチスレッド環境で wolfSSL を使用したい、またはシングルスレッドモードでコンパイルしたい場合です。

wolfSSL をシングルスレッド環境でのみ使用する場合、wolfSSL をコンパイルするときに SINGLE_THREADED を定義することで wolfSSL ミューテックスレイヤーを無効にできます。これにより、wolfSSL ミューテックスレイヤーを移植する必要がなくなります。

wolfSSL をマルチスレッド環境で使用する場合、wolfSSL ミューテックスレイヤーを新しい環境に移植する必要があります。ミューテックスレイヤーは ./wolfssl/wolfcrypt/wc_port.h および ./wolfcrypt/src/wc_port.c にあります。新しいシステムでは、wc_port.h に wolfSSL_Mutex を定義し、wc_port.c にミューテックス関数 (wc_InitMutex、wc_FreeMutex、wc_LockMutex、wc_UnLockMutex) を定義する必要があります。wc_port.h と wc_port.c を検索すると、既存のプラットフォームポートレイヤー (EBSNET、FREESCALE_MQX など) の例を参照できます。

H.4.7 ランダムシード

Q: どんな場合にこの章の内容が役立ちますか?

A: /dev/random や /dev/urandom を利用できない、あるいはハードウェア RNG に統合する必要がある場合です。

デフォルトでは、wolfSSL は /dev/urandom または /dev/random を使用して RNG シードを生成します。wolfSSL をビルドするときに NO_DEV_RANDOM 定義を使用して、デフォルトの GenerateSeed() 関数を無効にすることができます。これが定義されている場合は、ターゲットプラットフォームに固有の ./wolfcrypt/src/random.c にカスタム GenerateSeed() 関数を記述する必要があります。これにより、ハードウェアベースのランダムエントロピーソースが利用可能な場合は、wolfSSL の PRNG にシードを設定できます。

GenerateSeed() の記述方法の例については、./wolfcrypt/src/random.c にある wolfSSL の既存の GenerateSeed() 実装を参照してください。

H.4.8 メモリー

Q: どんな場合にこの章の内容が役立ちますか?

A: 標準のメモリ関数が利用できない、あるいはオプションの数学ライブラリ間のメモリ使用量の違いに関心がある場合です。

wolfSSL 本体はデフォルトで malloc() と free() の両方を使用します。旧来使用されてきた第 1 世代の整数演算ライブラリ (Normal Math) を使用する場合、wolfCrypt は realloc() も使用します。

現在、整数演算ライブラリとして最初期に開発された第 1 世代の Normal Math ライブラリ、パブリックドメインの TFM(Tom's Fast Math) をベースに開発した第 2 世代の Fast Math ライブラリ、SP(Single Precision) 最適化を適用した第 3 世代の SP Math ライブラリが存在します。このうち、第 2 世代以降の Fast Math, SP Math ライブラリでは、暗号操作において動的メモリを使用しません。

メモリ使用量とスピードの観点から、私たちは SP Math ライブラリの使用を推奨しています。wolfSSL 5.4.0 以降ではデフォルトで採用しており、第 1 世代の Normal Math ライブラリは廃止予定です。詳細は以下のページをご覧ください。

- wolfSSL の新たな Multi-Precision 演算ライブラリ
- wolfSSL Math Library Comparison Matrix
- レガシー数学ライブラリを廃止します

wolfSSL の TLS レイヤーは依然としていくらかの動的メモリを使用するため、malloc() と free() は依然として必要です。

通常の malloc()、free()、および場合によっては realloc() 関数が使用できない場合は、XMALLOC_USER を定義し、ターゲット環境に固有の ./wolfssl/wolfcrypt/types.h でカスタムメモリ関数フックを提供します。

XMALLOC_USER の使用の詳細については、5.1.1.1 節 をご参照ください。

H.4.9 時間

Q: どんな場合にこの章の内容が役立ちますか?

A: 標準の時間関数 (time()、gmtime()) を利用できない、あるいはカスタムのクロックティック関数を指定する必要がある場合です。

デフォルトでは、wolfSSL は ./wolfcrypt/src/asn.c で指定されているように、time()、gmtime()、および ValidateDate() を使用します。これらは、XTIME、XGMTIME、および XVALIDATE_DATE に抽象化されています。標準の時間関数と time.h を利用できない場合は、ユーザーは USER_TIME を定義できます。USER_TIME を定義した後、ユーザーは独自の XTIME、XGMTIME、および XVALIDATE_DATE 関数を定義できます。

wolfSSL は、クロックティック関数にデフォルトで time(0) を使用します。これは、LowResTimer() 関数内の ./src/internal.c にあります。

USER_TICKS を定義すると、time(0) が必要ない場合にユーザーが独自のクロックティック関数を定義できるようになります。カスタム関数には秒単位の精度が必要ですが、エポックと相関させる必要はありません。参考として、./src/internal.c の LowResTimer() 関数を参照してください。

H.4.10 C 標準ライブラリ

Q: どんな場合にこの章の内容が役立ちますか?

A: C 標準ライブラリを使用できない、あるいは独自のライブラリがある場合です.

wolfSSL は、開発者に高いレベルの移植性と柔軟性を提供するために、C 標準ライブラリなしで構築できるようにしています。その場合、ユーザーは C 標準の関数の代わりに使用したい関数をマップする必要があります。

第 7 章では、メモリ関数について説明しました。メモリ関数の抽象化に加えて、wolfSSL は文字列関数と数学関数も抽象化します。特定の関数は通常、X<FUNC> 形式の定義に抽象化されます。<FUNC> は抽象化される関数の名前です。

詳細については、5.1 章 をご覧ください。

H.4.11 ロギング

Q: どんな場合にこの章の内容が役立ちますか?

A: デバッグメッセージを有効にしたいが、stderr を使用できない場合です。

デフォルトでは、wolfSSL は stderr を介してデバッグ出力を提供します。デバッグメッセージを有効にするには、wolfSSL を DEBUG_WOLFSSL を定義してコンパイルし、アプリケーションコードから wolfSSL_Debugging_ON() を呼び出す必要があります。同様に，アプリケーションから wolfSSL_Debugging_OFF() を呼び出すことで、wolfSSL デバッグメッセージをオフにすることもできます。

stderr を使用できない，あるいは別の出力ストリームや別の形式でデバッグメッセージを出力したい環境には、独自のコールバック関数を使用できるようにしています.

詳細については、8.1 章 をご覧ください。

H.4.12 公開鍵操作

Q: どんな場合にこの章の内容が役立ちますか?

A: wolfSSL で独自の公開鍵実装を使用したい場合です。

wolfSSL では、SSL/TLS レイヤーが公開鍵操作を行う必要があるときに呼び出される，独自の公開鍵コールバックをユーザーが作成できます。ユーザーはオプションで 6 つの機能を定義できます。

1. ECC 署名 コールバック
2. ECC 検証 コールバック
3. RSA 署名 コールバック
4. RSA 検証 コールバック
5. RSA 暗号化コールバック
6. RSA 復号 コールバック

詳細については、6.4 章 をご覧ください。

H.4.13 アトミックレコードレイヤーの処理

Q: どんな場合にこの章の内容が役立ちますか?

A: レコードレイヤーの処理、特に MAC/暗号化および復号/検証操作を独自に実行したい場合です。

デフォルトでは、wolfSSL は暗号ライブラリ wolfCrypt を使用し、ユーザーに代わってレコードレイヤーの処理を行います。wolfSSL は、SSL/TLS 接続中に MAC/暗号化および復号/検証機能をより細かく制御したいユーザーのために、アトミックレコード処理コールバックを提供します。

ユーザーは 2 つの関数を定義できます.

1. MAC/暗号化コールバック関数
2. 復号/検証コールバック関数

詳細については、6.3 章 をご覧ください.

H.4.14 機能

Q: どんな場合にこの章の内容が役立ちますか?

A: 特定の機能を無効にしたい場合です。

wolfSSL をビルドする際，マクロ定義を使用して特定の機能を無効化できます。使用可能なマクロ定義については、2 章 をご覧ください。

H.5 次のステップ

H.5.1 wolfCrypt テストアプリケーション

wolfSSL をターゲットプラットフォームで適切にビルドできるようになったら、次のステップとして wolfCrypt テストアプリケーションを移植するのがよいでしょう。このアプリケーションをターゲットシステムで実行すると、NIST テストベクトルを使用して、すべての暗号アルゴリズムが正しく動作しているかどうか検証されます。

このステップを省略して SSL/TLS 接続に進むと、基盤となる暗号操作の失敗によって発生する問題のデバッグが難しくなる可能性があります。

wolfCrypt テストアプリケーションは `./wolfcrypt/test/test.c` にあります。組み込みアプリケーションに独自の `main()` 関数がある場合は、`./wolfcrypt/test/test.c` をコンパイルするときに `NO_MAIN_DRIVER` を定義する必要があります。これにより、アプリケーションの `main()` が各暗号/アルゴリズム テストを個別に呼び出すことができます。

組み込みデバイスに wolfCrypt テストアプリケーション全体を実行するのに十分なリソースがない場合は、個々のテストを `test.c` から切り離して個別にコンパイルできます。`test.c` から分離された暗号テストを抽出するときは、特定のテストケースに必要な正しいヘッダーファイルがビルドに含まれていることを確認してください。

H.6 サポート

一般的なサポートの質問は、メール、サポートフォーラム (英語)、または wolfSSL の Zendesk チケットトラッキングシステムを介して wolfSSL に直接送信できます。

Web サイト: https://www.wolfssl.com

サポート用メールアドレス: support@wolfssl.com

Zendesk: https://wolfssl.zendesk.com

フォーラム: https://www.wolfssl.com/forums

wolfSSL は、ユーザーと顧客が wolfSSL を新しい環境に移植するのを支援するために、いくつかのサポートパッケージとコンサルティングサービスを提供しています。

サポートパッケージ: https://wolfssl.jp/license/support-packages/

お問い合わせ: info@wolfssl.jp

I wolfSM (ShangMi)

この章では、wolfSSL における中国国家標準の暗号化アルゴリズムである ShangMi（SM）に関する情報を提供します。

wolfSM は以下の要素をサポートしています。

- SM3 - ハッシュ関数
- SM4 - 暗号
- SM2 - ECDH 鍵共有と指定された 256 ビット楕円曲線を使用した署名スキーム

使用するためには、コードを wolfSSL にインストールする必要があります。ただし、テストとビルド設定コードはすでに wolfSSL に含まれています。

I.1 wolfSM の取得とインストール

I.1.1 GitHub から wolfSM を取得する

GitHub から wolfSM リポジトリをクローンします。

```
git clone https://github.com/wolfssl/wolfsm.git
```

I.1.2 GitHub から wolfSSL を取得する

SM アルゴリズム実装をビルドしてテストするには wolfSSL が必要です。wolfSM の隣に GitHub から wolfSSL リポジトリをチェックアウトします。

```
# ディレクトリ構造は次のようになります。
# <install-dir>
# ├── wolfsm
# └── wolfssl

cd .. # wolfsm を含むディレクトリに移動
git clone https://github.com/wolfssl/wolfssl.git
```

I.1.3 SM コードを wolfSSL にインストールする

wolfSSL に SM コードをインストールするには、インストールスクリプトを使用します。

```
cd wolfsm
./install.sh
```

I.2 wolfSM のビルド

wolfSM ファイルを wolfSSL にインストールした後に、SM アルゴリズムを wolfSSL にビルドできます。

configure 時に必要なアルゴリズムをご選択ください。

- `--enable-sm3`
- `--enable-sm4-ecb`
- `--enable-sm4-cbc`
- `--enable-sm4-ctr`
- `--enable-sm4-gcm`
- `--enable-sm4-ccm`
- `--enable-sm2`

例えば SM3、SM4-GCM、SM2 を含める場合、次のようにします。

```
./autogen.sh
./configure --enable-sm3 --enable-sm4-gcm --enable-sm2
make
sudo make install
```

I.2.1 SM2 の最適化実装

SM2 の最適化された実装を使用するには、C のみのコードを使用するか、より高速なアセンブリコードを備えた C コードを使用できます。

- C コードのみの場合：`--enable-sp`
- C とアセンブリコードの場合：`--enable-sp --enable-sp-asm`

最適化された C コードは、32 ビットおよび 64 ビット CPU で利用可能です。

アセンブリコードは、以下のプラットフォームで利用可能です。

- Intel x64
- Aarch64
- ARM 32-bit
- ARM Thumb2
- ARM Thumb

I.3 wolfSM のテスト

SM 暗号が動作することをテストするには、次のコマンドを実行します。

```
make test
```

次のコマンドで、有効化されたアルゴリズムのベンチマークも実行できます。

```
./wolfcrypt/benchmark/benchmark
```

特定のアルゴリズムのベンチマークを行うには、アルゴリズムに対応するオプションをコマンドラインに追加してください。

- SM2: `-sm2`
- SM3: `-sm3`
- SM4: `-sm4` もしくは、
  - SM4-CBC: `-sm4-cbc`
  - SM4-GCM: `-sm4-gcm`
  - SM4-CCM: `-sm4-ccm`

I.3.1 TLS のテスト

SM 暗号は TLS 1.2 および TLS 1.3 で使用できます。

注意: SM 暗号スイートを動作させるには、SM2、SM3、および少なくとも 1 つの SM4 暗号をビルドする必要があります。すべてのアルゴリズムは SM である必要があります。

追加される暗号スイートは以下の通りです。

- 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 TLS 1.2 で SM 暗号スイートを使用する

「ECDHE-ECDSA-SM4-CBC-SM3」暗号スイートを使用して TLS 1.2 をテストする例を以下に示します。

```
./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
```

実行結果は以下のようになります。

```
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 TLS 1.3 で SM 暗号スイートを使用する

「TLS13-SM4-GCM-SM3」暗号スイートを使用して TLS 1.3 をテストする例を以下に示します。

```
./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
```

実行結果は以下のようになります。

```
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!
```