MWA

MWA Software

OpenSSL Pascal API Code Generator

Issue 1.0,
29 December 2024

MWA Software

EMail: info@ mccallumwhyman.com, http://www.mccallumwhyman.com

CONTENTS **Page**

1 Introduction

This document describes a Code Generator for creating a Pascal Interface to the OpenSSL API. OpenSSL is itself an open source code library providing an implementation that includes the RFC 8446 Transport Layer Security (TLS) Protocol Version 1.3. (see https://www.openssl.org/).

The OpenSSL library is written in 'C' and the programmatic interface is defined in a series of 'C' header files. These have to be translated into Pascal units in order to declare the interface to a Pascal code library.

Utilities, such as Free Pascal's *h2pas* can help this process - but only to the extent of creating a set of Pascal constant and type definitions and Function/Procedure declarations - and even then there is usually the need for additional file edits to cope with some of the more difficult translations, especially where 'C' macros are concerned, and to convert the output function/procedure declarations into external declarations.

Additional complexity is introduced given that Pascal also requires subtly different Function/Procedure external declarations depending on the intended link model i.e.

- Compile time linkage to a shared (.so or .dll) library
- Compile time linkage to a static library
- Dynamic Library Load and the run time resolution of each API call used.

Furthermore, there is often a need to support multiple versions of the OpenSSL API, and code differences between different Pascal compilers (e.g. Delphi[1] and Free Pascal[2]).

[1]See https://www.embarcadero.com/products/delphi

[2]See https://www.freepascal.org/

With an API as rich as that provided by OpenSSL and the number of changes that have occurred in that API over the last few years, it is always going to be a major task to manually edit and maintain Pascal header files that support all of the above variations.

This document describes a code generator aimed at taking (template) header files created from OpenSSL '.h' files using a utility such as *h2pas,* and generating a set of Pascal units that supports the latest OpenSSL API that provides for:

- Different link models
- Backwards compatibility to earlier versions
- Delphi and Free Pascal support.

The initial target for these units is the Indy Project and as part of an activity to add OpenSSL 3.x support to Indy[3]. However, the intent is also that the generated units are otherwise independent of Indy and may be used by any Pascal Project that needs to access the OpenSSL API.

The Code Generator itself is written in Pascal and may be compiled using Free Pascal or Delphi. It is distributed under the GNU Public Licence (GPL). The source code distribution also includes:

- Template files for the OpenSSL Pascal API.
- Example generated Pascal units providing an implementation of the Pascal OpenSSL API and which may be used as provided in user projects.

Both template and example units are derived from the OpenSSL source code and hence are distributed using the same Apache License v2.0 as OpenSSL itself.

For a description of the use of the generated units see chapter 6.

[3]See https://github.com/IndySockets/Indy

2 Program Overview

The OpenSSL API Code Generator is a command line program written in Pascal and may be compiled with Delphi or Free Pascal. It has been tested with Delphi on Windows, and with Free Pascal on both Linux and Windows.

2.1 Process Flow

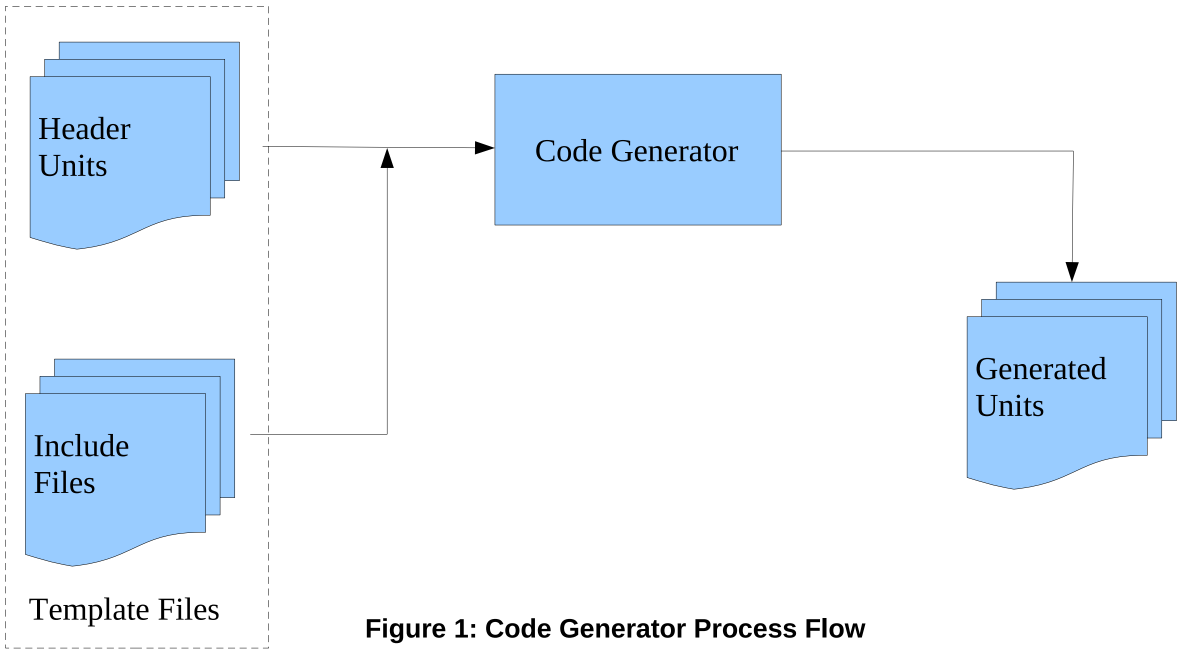

Figure 1: Code Generator Process Flow

Figure 1 above illustrates the process flow for the code generator.

- Input to the program are two sets of one or more template files:

  - The Header Unit Template files are Pascal Units that have been generated from the OpenSSL API .h files using *h2pas*. A set of suitable template header files are include with the source code and, in this set, there is a one to one correspondence between each OpenSSL .h file and the template units. The template units also include compatibility (added code) functions and procedures in their implementation sections that are intended to be used when the corresponding API call is not available in the version of the OpenSSL so/dll being used.

  - The Included Template files provide common const and type definitions, and additional functionality, such as a dynamic library loader and exception handling. Only limited processing of include files is performed.

- The code generator writes output units to a specified directory. An output unit is generated for each header unit template and for each include file.

- The code generator can add prefixes to unit names and update their use in “uses” clauses. It also supports more than one dynamic library load strategy selected at code generation time.

2.2 Command Line Syntax

```
Usage: OpenSSL_API_CodeGenerator [-h] [--smart] [--jit] [-p <output unit prefix]
[-a <include file(s)][-P <include file prefix>][-o <output directory>] <file or
directory>
Options:
-h              Help: outputs this message
-s|--smart      Generate smart load strategy (default)
-j|--jit        Generator Just In Time Load strategy
-p|--prefix     Output unit name prefix
-P|--includedFilePrefix  Include  File  (unitname)  Prefix  (overrides  unit  name
prefix
-a|--includeFiles   includefile(s) source dir
```

The command line options are defined as:

-h	Outputs the above message
-s|--smart	Generates the output units using the “smart” load strategy (see 4.2)
-j|--jit	Generates the output units using the “just in time” load strategy (See 4.2)
-p|--prefix	Prepend all output unit names with the given prefix (defaults to OpenSSL_)
-P|--includedFilePrefix	Prepend all output include file unit names with the given prefix (defaults to the empty string)
-a|--includeFiles	Directory containing include files

The final command line argument is the source header file or directory. In the latter case, all files with the .h2pas suffix in that directory are processed, otherwise it is only the specified file. Any include files are ignored unless the source is a directory.

When generating the OpenSSL for Indy, the recommended approach is that a unit prefix of “IdOpenSSLHeaders_” is used and an include file prefix of “IdSSL”. E,g,

```
OpenSSL_API_CodeGenerator -o examples/openssl -j -p IdOpenSSLHeaders_ \
        -P IdSSL -a templates/openssl/files  templates/openssl/headers
```

Assuming that the current directory is the source code root and the OpenSSL_API_CodeGenerator is on the current path.

If you are using the generated header files in your own project, you may use any prefix of your choosing. However, an empty header file prefix is not recommended and will probably result in a compilation error. This is because of the risk of name clashes between unit names and OpenSSL type names.

3 Unit Templates

As discussed above, there are two types of input files: Header File Templates and Include Files.

3.1 Header File Templates

The header file templates are Pascal Units and follow the following syntax:

1. Standard unit layout with no external definitions

2. The OpenSSL API is the subject of the interface section, and may comprise, const, type, var and procedure/function declarations. Any procedure/function declarations must not include an “external” directive nor any calling conventions.

3. Special bracketed comments after each function/procedure definition may used to identify new (introduced) and removed functions procedures together with the 3 level release number (major.minor.fix) in which they were introduced/removed.

   ```
   e.g. {introduced 1.1.1} or {introduced 1.0.0 removed 3.0.0}.
   ```

   Removed clauses may also be followed by "allow_nil" indicating that it is not an error if the function/procedure cannot be loaded when the library is dynamically loaded.

   ```
   e.g. {removed 3.0.0 allow_nil}.
   ```

4. The Implementation section may be empty or comprise one or more const, types and local procedure/function bodies for removed procedure/functions where a backwards or forwards compatible procedure/function is needed. These procedures/functions should have the same name as the API call to which they apply.

   For example, the function OpenSSL_version_num was introduced in OpenSSL 1.1.0 and is thus declared in the interface section as:

```
function OpenSSL_version_num: TOpenSSL_C_ULONG; {introduced 1.1.0}
```

It replaced the earlier SSLeay which is declared (in the template's interface section) as:

```
function SSLeay: TOpenSSL_C_ULONG; {removed 1.1.0}
```

In order to provide backwards compatibility to programs when an earlier version of the OpenSSL library is loaded, “OpenSSL_version_num” may be declared in the implementation section as:

```
function OpenSSL_version_num: TOpenSSL_C_ULONG;
begin
    Result := SSLeay;
end;
```

Note. Compatibility Functions may be provided for any removed or introduced API call and are used if the corresponding API call is not available in the loaded library. For introduced API calls they provide a means by which user code can be written to support a later version of OpenSSL (e.g. 3.0.0) but which can still work with earlier versions of OpenSSL.

Note. All Compatibility Functions are placed in conditional compilation sections such that they are ignored if the symbol OPENSSL_NO_LEGACY_SUPPORT is defined when a user program is compiled thus providing a means to avoid unnecessary bloat when not needed (see 6.1.2).

5. Both interface and implementation sections may contain one or more sections delimited by "{passthrough}" and "{/passthrough}" special comments. These sections are copied unchanged to the output unit.

By default API calls are expected to be in the library 'LibCrypto”. However, this may be overridden by including the (pseudo) directive $UnitLibName in the template file prior to any API calls .e.g.

```
{$UnitLibName LibSSL}
```

is included in ssl.h2pas in order to indicate that the API calls may be found in LibSSL.

This directive is **not** copied into the output file.

Note. “LibSSL” is a reference to a constant (prefixed by 'C')in the OpenSSLAPI unit. The constant value is the name of the library itself.

3.2 Include Files

These are Pascal units (.pas) or fragments of Pascal units (.inc), or any other file. They are otherwise unconstrained. Only .pas and .inc files are subject to limited processing (i.e. unit name prefix management). All other files are copied transparently.

4

Library Linking Strategies

In the Indy component library, the default linking strategy for OpenSSL was to load the OpenSSL libraries dynamically at runtime and to resolve each API call entry point using the 'GetProcAddress' function call. The returned address was then assigned to a function variable with an appropriate type.

The downsides of this “legacy” approach are that:

- It may take a noticeable time to load all API call addresses including many that are never used.
- An error is typically declared if an API call fails to be loaded even when the call is never used by the user program.
- There is no overall strategy for managing multiple versions of the OpenSSL library with different sets of API calls.
- Static linking is possible but only by replacing the dynamic loading units with a static linkage unit and using conditional compilation in user code to manage the differences.

The code generator is intended to avoid these problems, and uses conditional compilation to support each linking strategy in the same unit transparently to the user program. Both static and dynamic linking are thus supported.

4.1 Static Linking

Two flavours of static linking are supported:

- Linking to a static library (typically with a *.a* suffix).
- Linking to a shared library (*.so* or *.dll*) with call API resolution by the Operating System's program loader at program load time.

With static linking, a specific OpenSSL version has to be supported. Currently this is OpenSSL 3.0.0 or later. Only the API calls present in 3.0.0 can be declared as external function/procedure calls.

Note. This restriction is a consequence of the linking/loading of the program failing if not all declared API calls are present in the external library.

4.1.1 Linking to a static library

In order to achieve this:

- each API call must be formatted as an external declaration with an empty library name.
- FPC: a {$LINKLIB …} compiler directive must be provided for each external library to which the compiled unit is to be linked (e.g. {$LINKLIB ssl.a} {$LINKLIB crypto.a}).
- Delphi: a {$LINK …} compiler directive must be provided for each external object file to which the compiled unit is to be linked. You can use an object file produced by Borland C++ and Borland TASM (Turbo assembler), but not by C++ Builder (tbc).

4.1.2 Linking to a shared library

In order to achieve this:

- each API call must be formatted as an external declaration with the name of the shared library in which the API call is located.

4.1.3 Implementation

The Code Generator generates a conditional compilation section in each output unit's interface section providing:

- Each 3.0.0 API call as an external declaration with a named constant used for the library name.
- Each compatibility function for an API call removed prior to or by 3.0.0 and declared as an internal function (this allows a user program to use any API calls removed by 3.0.0 or earlier and implemented as compatibility functions).

The implementation section includes a conditional compilation section with the function/procedure bodies for each compatibility function declared in the implementation section.

In the example templates, the OpenSSLAPI unit (include file) defines common types used by the API calls. It also includes:

- a {$LINKLIB …} directive for standard OpenSSL libraries (e.g. those generated by gcc) when FPC is used. Delphi is not currently supported for static linking.
- The definitions of the named constants used for library names. These constants are empty strings when linking to a static library, while their value is set to the library names when linking to a shared library.

4.2 Dynamic Linking

Two flavours of dynamic linking are supported:

- “Smart Linking”: API calls are loaded in turn by a loader function. If the API call is not present in the library, the call is replaced with a call to a compatibility function, if one exists, or to an error function otherwise. The error function raises an EOpenSSLAPIFunctionNotPresent exception if a user later executes the API call. Template markup may alternatively allow an API call to remain nil on failure to load (see 3.1).

- “Just in Time Linking (JIT)”: At compile time, each API call is set to a generated loader function specific to the API call. The first time a user calls the API call, the API call is resolved from the library and replaces the customised loader function address in the API Call function variable. The API call is then invoked. If the API call is not present in the library, the call is replaced with a call to a compatibility function, if one exists, or an EOpenSSLAPIFunctionNotPresent exception is raised.

  If the template markup allows an API call to remain nil on failure to load then “smart linking” applies to that call instead i.e. it is loaded when the library is loaded.

4.2.1 Implementation

A conditional compilation section is included in each output unit's interface section providing:

- Var declarations for each 3.0.0 API call, followed by

- Var declarations for each API call removed prior to or by 3.0.0 and for which a compatibility function is available.

  Note. In both cases the variable name is the same as the API call name and the variable type is the call signature.

- “Smart”: at compile time each variable is initialised to nil.

- “JIT”: at compile time each variable is initialised to the address of the API Call's generated loader function.

A conditional compilation section is included in each output unit's implementation section providing:

- Var declarations for each API call removed prior to or by 3.0.0 and for which a compatibility function is not available. They are thus made available for use by compatibility functions if present. They are initialised as above.

- Each compatibility function defined in the template is copied to the implementation section and with its name prefixed by “COMPAT_”.

- JIT: a loader function body for each API call is generated.

- A “load” function is defined which attempts to use “GetProcAddress” to load each API call in turn.

  - Smart: every API call is loaded.

    - JIT: only API calls which are allowed to remain nil on a failure to load are loaded.

- Load Function Registration:

  At unit initialisation time, the Load function is registered with the library loader (in the OpenSSLAPI unit) by a call to the registration function “Register_SSLLoader”.

  Note. Under JIT if there are no (allow nil) API calls to be loaded by the load function in this unit then no load function is generated and hence the registration function is not called.

4.3 The OpenSSLAPI Unit

In the example templates the include file “OpenSSLAPI.pas” includes the library loader. This is also subject to conditional compilation and only included when dynamic linking is used.

The library loader loads both OpenSSL libraries (libssl and libcrypto) and then calls each registered load function in turn.

The library loader must be explicitly called for “smart linking” but may be implicitly called for “Just in Time linking”. The loader also exposes several information functions about the loaded libraries and has properties that can be used to specify the library location (see section 5).

5

Output Unit Layout

5.1 Include Files

These are copied unchanged to the output directory except for:

- A unit name prefix is added if required and the output file name changed appropriately (extension always .pas).
- Any uses clause in either the interface or implementation section or both is updated such that unit names referring to include or header file units, are replaced with the unit name updated to include the required prefix, if any.

5.2 The “All Headers” Unit

An “All Headers” unit is added (unit name <prefix>AllHeaders.pas). This is empty apart from a “uses” clause that lists all generated header units.

It is intended to provide a convenient means to compile all output units if necessary.

5.3 Header Units

These are syntactically correct pascal units with a .pas extension. There is one such unit for each input template header. The unit name is the same as the corresponding template header with any required prefix added. The unit comprises the following in order:

5.3.1 Interface section:

1. All const, type and var declarations up to the first function/procedure declaration are copied unchanged to the output unit, including any embedded comments and compiler directives. This is followed by:

2. All passthrough sections found in the template unit's interface section.
3. An “externalsym” special comment identifying each API call name found in the template.
4. Alternative static library linking and dynamic library linking sections follow.
5. The static library linking section comprises every API function/procedure declaration identified as being supported by OpenSSL 3.0.0 or later (including all those without any version information). Each is formatted as an external declaration using the “cdecl” calling method. A Pascal constant name follows the external directive. This is intended to give the name of the appropriate OpenSSL library if linking to a shared library at load time is required and to be empty if linking to a static library at link time is required.

   Note: in the provided templates, the include file “OpenSSLAPI” defines this constant with conditional compilation used to determine whether or not the constant is empty.
6. The static linking section is completed by providing declarations for each API call removed prior to OpenSSL 3.0.0 but for which a backwards compatibility function is present in the implementation section. These are subject to additional conditional compilation and are not present if the defined symbol OPENSSL_NO_LEGACY_SUPPORT has been defined at compile time (see 6.1.2).
7. The dynamic library linking section comprises every OpenSSL 3.0.0 or later API function/procedure declaration formatted as a “var” declaration that defines a function/procedure variable with the same name as the API call and a type compatible with the function signature.
8. Each such “var” declaration is initialised at compile time to:

   a) “Smart” Loading: to nil

   b) “JIT Loading” to a dynamic load function. These functions are declared before all such “var” declarations as forward references.
9. A var declaration then follows for each API call removed prior to OpenSSL 3.0.0 but for which a backwards compatibility function is present in the implementation section.
10. External function version information is then provided as a series of constants giving an encoded integer representation of the OpenSSL library version that the API Call was introduced and/or removed, as applicable. The encode version numbers are compatible and may be compared with the result of the GetOpenSSLVersion function provided by the IOpenSSL interface (see 6.2.1).

A template unit header file may include compilation directives in the interface section, including within the API function/procedure declarations. These are copied into the corresponding output unit. When such directives are located within the API function/procedure declarations they will be repeated in both the static and dynamic linking sections.

5.3.2 Implementation Section

1. In a conditional compilation section for dynamic library linking, a var declaration follows for each API call removed prior to OpenSSL 3.0.0 but for which a backwards compatibility function is not present in the implementation section.

2. All const, type and var declarations up to the first function/procedure declaration are copied unchanged to the output unit, including any embedded comments and compiler directives.

3. All passthrough sections found in the template unit's implementation section.

4. In a conditional compilation section for static linking, the function/procedure bodies for each compatibility function declared in the template implementation section.

5. In a conditional compilation section for dynamic library linking, the function/procedure bodies for each compatibility function declared in the template implementation section, with the function/procedure name prefixed by “COMPAT_”.

6. Smart: An Error Function for each API call loaded by the loader function and for which a compatibility function is not available.

7. A Load Function:

    ◦ Smart: All API calls are loaded in turn.

    ◦ JIT: only API calls marked as “allow_nil” are loaded in turn.

8. An Unload Function:

    ◦ Smart: All API calls are set to nil.

    ◦ JIT: All API calls not marked as allow_nil are set to their loader function, otherwise, the call is set to nil.

5.3.3 Processing of “Uses” clauses

Unit “uses” clauses may be found at the start of both the interface and implementation sections.

1. Any uses clauses found in the template or include file units are copied to the output unit with an unit names that refer to template or include files adjust to include the required prefix, if any.

2. Additional unit names are added to an implementation uses clause in support of generated exceptions and error handling. These are “Classes, OpenSSLExceptionHandlers and OpenSSLResourceStrings.

6

Using the OpenSSL API Units

The output header files (JIT or Smart) should be copied into your project/package. It is recommended that they are all placed in a dedicated subdirectory and, the project options are set such that the unit and include file search paths include this subdirectory.

Note. In order to avoid linking in OpenSSL header units unnecessarily, these units should not be explicitly added to your project or package.

A set of “Just in Time” (JIT) headers is provided with the source code along with the original template files. In most cases, all you will need to do is to copy the provided JIT headers. Only if you want to make modifications to the templates or use the alternative “smart” headers will you need to compile and run the header generator in order to generate a new set of header file units.

In order to use an API call it is sufficient to include the appropriate header file unit name in your unit's uses clause and call the API call as a normal function or procedure. This is regardless of whether static or dynamic linking is used. Each API call has the same name as the original OpenSSL 'C' API call and the header file unit names follow the OpenSSL header naming conventions. The OpenSSL Documentation should be consulted for the semantics of each API call.

With dynamic linking, there is no need to explicitly load the library prior to use as it is implicitly loaded and initialised when the first API call is called. With static linking, the library must be explicitly initialised prior to use (see 6.2.1).

With “smart” linking, the library should always be explicitly loaded prior to use. With JIT, if your code tests API Calls tagged with “allow_nil” to see if they are present then the library should also be explicitly loaded prior to use, otherwise, implicit loading is acceptable.

6.1 Compile Time Code Selection

The following header file options may be selected at compile time.

6.1.1 Link Strategy Selection

If static linking is required then your program (or package) must be compiled with the defined symbol:

- OPENSSL_USE_STATIC_LIBRARY, or
- OPENSSL_USE_SHARED_LIBRARY.

The former selects linking to static OpenSSL libraries (e.g. libssl.a and libcrypto.a), while the latter selects linking to shared OpenSSL libraries (.so/.dll). In both cases, the libraries must be 3.0.0 or later.

Dynamic linking is the default (neither symbol is defined). In this case, the dynamic library loader (in the OpenSSLAPI unit) searches for the shared OpenSSL libraries.

Note: The OpenSSLAPI unit is always included by default if an OpenSSL Header unit is included in your project.

6.1.2 Legacy Support

Primarily in support of Indy, the header files include compatibility functions to enable the use of OpenSSL from 1.0.2 or later (with dynamic library load). For example, these compatibility functions:

- Replace OpenSSL 3.0 API calls with functions that call equivalent functions in earlier versions of OpenSSL.
- Replace 1.0.2 API calls with calls to equivalent 3.0.0 functions. This reduces the transition effort for update to 3.0.0 support.

If your application only uses OpenSSL 3.0.0 and 3.0.0 API calls then the compatibility functions can be ignored at compile time by compiling with the defined symbol:

OPENSSL_NO_LEGACY_SUPPORT

This should avoid unnecessary program bloat.

6.1.3 Memory Management

OpenSSL's internal memory management functions can be replaced by the Pascal Memory Manager by compiling with the defined symbol:

OPENSSL_SET_MEMORY_FUNCS

This may be useful for debugging e.g. reporting memory leaks on completion.

6.1.4 Minimum Library Version Checking

By default, the OpenSSL API dynamic library loader checks the library version and raises an exception if the version number is less that 1.0.0. This can be turned off by compiling with the defined symbol

OPENSSL_NO_MIN_VERSION

6.2 Run Time Interface

The OpenSSLAPI unit provides two Pascal COM interfaces which provide information and option selection for the OpenSSL Library.

6.2.1 The IOpenSSL Interface

This interface is available for all link strategies and the interface is accessed using the function:

```
  function GetIOpenSSL: IOpenSSL;
```

The interface is declared as:

```
  IOpenSSL = interface
  ['{aed66223-1700-4199-b1c5-8222648e8cd5}']
    function GetOpenSSLPath: string;
    function GetOpenSSLVersionStr: string;
    function GetOpenSSLVersion: TOpenSSL_C_ULONG;
    function Init: boolean;
  end;
```

`function GetOpenSSLPath: string`	Static Linking: Returns the OpenSSL installation path as compiled into the OpenSSL library. Dynamic Loading: Returns • The same as above when the library is loaded • the specified OpenSSL installation path, otherwise.
`function GetOpenSSLVersionStr: string`	Returns the OpenSSL library version string. e.g “OpenSSL 3.2.0 23 Nov 2023”
`function GetOpenSSLVersion: TOpenSSL_C_ULONG`	Returns the OpenSSL library version as an integer as defined by the OpenSSL documentation.
`function Init: boolean`	Called to initialise the OpenSSL library prior to use. This must be called when using static linking. It is optional for dynamic linking (implicitly called on library load).

6.2.2 The IOpenSSLDLL Interface

This interface is only available when the API is configured at compile time for dynamic library loading. The interface is accessed using the function:

```
function GetIOpenSSLDDL: IOpenSSLDLL;
```

This function returns “nil” if the interface is not available. This may be used as a runtime test to determine if dynamic library loading if been configured.

Note The OpenSSLAPI unit only declares the constant

```
OpenSSL_Using_Dynamic_Library_Load = true;
```

when configured at compile time for dynamic library loading. The constant is not declared otherwise. This feature may be used as a compile time test for the dynamic library loading strategy e.g.

```
{$if declared(OpenSSL_Using_Dynamic_Library_Load)}
…
{$ifend}
```

The interface includes the IOpenSSL interface and is declared as:

```
  IOpenSSLDLL = interface('-3;-1;'IOpenSSL)
  ['{1d6cd9e7-e656-4981-80d2-288b12a69306}']
    procedure SetOpenSSLPath(const Value: string);
    function GetSSLLibVersions: string;
    procedure SetSSLLibVersions(AValue: string);
    function GetLibCryptoHandle: TLibHandle;
    function GetLibSSLHandle: TLibHandle;
    function GetLibCryptoFilePath: string;
    function GetLibSSLFilePath: string;
    function GetFailedToLoadList: TStrings;
    function Load: Boolean;
    procedure Unload;
    function IsLoaded: boolean;
    property SSLLibVersions: string read GetSSLLibVersions write SetSSLLibVersions;
end;
```

`procedure SetOpenSSLPath(const Value: string);`	This sets the OpenSSLPath used to locate the OpenSSL library modules (e.g. libcrypto.so). It needs to be set when OpenSSL has not been installed in a default location and/or multiple versions of OpenSSL have been installed on the same system.

`function GetSSLLibVersions: string;`
 	
`procedure SetSSLLibVersions(AValue: string);`
	This is a colon separated (Unixes) or semi-colon separated (Windows) ordered list of OpenSSL version numbers that can be used as suffices for the OpenSSL library modules (e.g. libcrypto-3-x64.dll.
	
	When searching for the OpenSSL library, the loader searches the default (or specified) OpenSSLPath for OpenSSL libraries using these suffices in turn.
	
	Unix: defaults to
	
	'.3:.1.1:.1.0.2:.1.0.0:.0.9.9:.0.9.8:.0.9.7:.0.9.6'
	
	Note includes legacy versions.
	
	Windows defaults to
	
	'-3-x64;-1-x64;' (64 bit)
	
	'-3;-1;' (32 bit)

`function GetLibCryptoHandle: TLibHandle`	After a successful library load, this returns the value of the internal handle to libcrypto (internal use only recommended).
`function GetLibSSLHandle: TLibHandle;`	After a successful library load, this returns the value of the internal handle to libssl (internal use only recommended).
`function GetLibCryptoFilePath: string`	After a successful library load, this returns the path to the loaded libcrypto library. (note: may be empty if default library loaded).
`function GetLibSSLFilePath: string`	After a successful library load, this returns the path to the loaded libssl library. (note: may be empty if default library loaded).

`function GetFailedToLoadList: TStrings;`	After a successful library load, this returns a list of API call names that failed to load at library load time. JIT: only API calls tagged as “allow_nil” are loaded at library load time and hence this list is restricted to API calls tagged as “allow_nil” only. Smart: Any API Call that fails to load. Note: API calls that fail to load and are not tagged as “allow_nil” will cause an EOpenSSLAPIFunctionNotPresent exception if called later.
`function Load: Boolean;`	Explicitly loads the library if not already loaded and returns “true” on successful load.
`procedure Unload;`	Explicitly unloads the library if current loaded.
`function IsLoaded: boolean`	Returns true if the OpenSSL library has been successfully loaded.