zilog
Embedded in Life
An IXYS Company

eZ80® CPU

ZTP Network Security SSL Plug-In

Reference Manual

RM004707-1211

www.zilog.com

Warning: DO NOT USE THIS PRODUCT IN LIFE SUPPORT SYSTEMS.

LIFE SUPPORT POLICY

ZILOG'S PRODUCTS ARE NOT AUTHORIZED FOR USE AS CRITICAL COMPONENTS IN LIFE SUPPORT DEVICES OR SYSTEMS WITHOUT THE EXPRESS PRIOR WRITTEN APPROVAL OF THE PRESIDENT AND GENERAL COUNSEL OF ZILOG CORPORATION.

As used herein

Life support devices or systems are devices which (a) are intended for surgical implant into the body, or (b) support or sustain life and whose failure to perform when properly used in accordance with instructions for use provided in the labeling can be reasonably expected to result in a significant injury to the user. A critical component is any component in a life support device or system whose failure to perform can be reasonably expected to cause the failure of the life support device or system or to affect its safety or effectiveness.

Document Disclaimer

 Information in this publication concerning the devices, applications, or technology described is intended to suggest possible uses and may be superseded. ZILOG, INC. DOES NOT ASSUME LIABILITY FOR OR PROVIDE A REPRESENTATION OF ACCURACY OF THE INFORMATION, DEVICES, OR TECHNOLOGY DESCRIBED IN THIS DOCUMENT. ZILOG ALSO DOES NOT ASSUME LIABILITY FOR INTELLECTUAL PROPERTY INFRINGEMENT RELATED IN ANY MANNER TO USE OF INFORMATION, DEVICES, OR TECHNOLOGY DESCRIBED HEREIN OR OTHERWISE. The information contained within this document has been verified according to the general principles of electrical and mechanical engineering.

Revision History

Each instance in the following revision history table reflects a change to this document from its previous version. For more details, refer to the corresponding pages or appropriate links provided in the table.

Date	Revision Level	Description	Page
Dec 2011	07	Updated for the SSL v2.4.0 release.	All
Feb 2009	06	Updated for the SSL v2.3.0 release.	
Nov 2008	05	Updated for the SSL v2.2.0 release.	All
Aug 2007	04	Updated for the SSL v2.1.0 release.	All
Jul 2006	03	Updated for the SSL v2.0.0 release.	All
Apr 2006	02	Updated for the SSL v1.2.0 release.	All
Jan 2006	01	Formatted to current publication standards.	All

Table of Contents

Introduction

This Reference Manual describes the APIs associated with the Zilog Network Security (SSL) Plug-In for eZ80® CPU-based microprocessors and microcontrollers. SSL supports the eZ80Acclaim!™ family of devices, which includes the eZ80F91, eZ80F92 and eZ80F93 microcontrollers; the eZ80L92 microprocessor is also supported.

About this Manual

Zilog recommends that you read and understand all chapters and instructions in this manual before using the ZTP SSL product. This manual serves as a reference guide for SSL APIs.

Intended Audience

This document is written for Zilog customers who are familiar with real-time operating systems, microprocessors, assembly language and high-level languages such as C.

Manual Organization

This Reference Manual contains the following chapters and appendices.

ZTP Network Security Plug-In

This chapter provides an architectural overview of Zilog's SSL implementation.

Data Structures

This chapter provides a detailed description of the SSL Data Structures in terms of syntax, argument descriptions, return values and example codes.

API Definitions

This chapter describes the SSL APIs in terms of syntax, argument descriptions, return values and example codes.

SSL Configuration

This chapter describes the global variables used to configure the SSL handshake protocols.

Related Documents

In addition to this manual, you must be familiar with the documents listed in Table 1.

Table 1. Related Documentation

Zilog Network Security Plug-In Quick Start Guide	QS0059
Zilog Network Security Plug-In User Manual	UM0201
Zilog Real-Time Kernel Quick Start Guide	QS0048
Zilog TCP/IP Software Suite Quick Start Guide	QS0049
Zilog TCP/IP Software Suite Reference Manual	RM0041

Manual Conventions

The following conventions are adopted to provide clarity and ease of use:

Use of X.Y.Z

Throughout this document, `X.Y.Z` refers to the SSL version number in `Major.Minor.Revision` format.

Courier New Typeface

Code lines and fragments, functions and various executable items are distinguished from general text by displaying in the Courier New typeface.

For example: `#include "ZSysgen.h"`.

Safeguards

When you use SSL with one of Zilog's development platforms, follow the precautions listed on this page to avoid permanent damage to the development platform.

Power-Up Precautions

When powering up, observe the following sequence.

1. Apply power to the PC and ensure that it is running properly.
2. Start the terminal emulator program on the PC.
3. Apply power through connector P3 on the eZ80 Development Platform.

Power-Down Precautions

When powering down, observe the following sequence.

1. Exit the monitor program.
2. Remove power from the eZ80 Development Platform.

Note: Always use a grounding strap to prevent damage resulting from electrostatic discharge (ESD).

ZTP Network Security Plug-In

This reference manual discusses Secure Sockets Layer (SSL) support to any TCP-based application that uses the Zilog TCP/IP (ZTP) Software Suite. This document also provides information about SSL APIs and data structures included in the ZTP Network Security plug-in package.

This package is only compatible with the ZTP Software Suite of the same version. For example, ZTP Network Security Plug-In package `X.Y.Z` is only compatible with ZTP Software Suite `X.Y.Z`; it is not compatible with ZTP Software Suite `X.Y+1.Z` or `X.Y.Z–1`.

Although the ZTP Network Security SSL Plug-In runs on the same processors as the ZTP Software Suite, the best performance is obtained when using faster platforms. For example, the ZTP Network Security SSL Plug-In will run much faster on a RAM-based eZ80F91 system than on a Flash-based eZ80F92 system.

About SSL

SSL is a suite of protocols that provides authentication, privacy and data integrity over an insecure channel; e.g., a TCP connection. Authentication ensures that communication occurs only with the intended target and not an attacker masquerading as the target. Privacy prevents eavesdroppers from understanding the communication, and the data integrity provides a mechanism for participants to detect a message that may have been tampered with by an attacker.

This package provides support for the SSL version 2, SSL version 3 and TLS version 1 protocols. Existing TCP server or client applications use one or more of these protocols to secure communication with a peer device. As a rule of thumb, later versions of the SSL protocol provide bet-

ter security than earlier versions of the protocol at the expense of increased code size and slightly slower operation.

Data Structures

This chapter describes the basic data types and data structures used by the SSL handshake protocols and cryptographic support routines.

Basic SSL Data Types

Table 2 lists the basic SSL data types (see `ssl_types.h`).

Table 2. Basic SSL Data Types

Data Type	Definition
BYTE	unsigned char (8-bit)
BOOL	unsigned char (8-bit)
WORD	unsigned short int (16-bit)
DWORD	unsigned long int (32-bit)
SSL_BYTE	BYTE
SSL_BOOL	BOOL
SSL_STATUS	SSL_BYTE
SSL_WORD	WORD
SSL_DWORD	DWORD
SSL_TRIO	unsigned int (24-bit)

Constructed SSL Data Types

The remainder of this chapter describes the data structures used by the SSL handshake protocols plus the cryptographic support routines that are constructed from the basic SSL data types listed in Table 2.

SSL_Data_Block_S

```
typedef struct SSL_DATA_BLOCK_S
{
  SSL_BYTE          *pData;
  SSL_WORD          Length;
} SSL_DATA_BLOCK_S;
```

SSL_DATA_BLOCK_S describes an opaque data structure. A data structure is said to be opaque if its owner does not have any knowledge of the information in the structure. In this instance, support routines are used to interpret and act upon the data. As an example, base64 ASN.1 DER-encoded X.509 certificates are used to initialize various SSL handshake protocol servers. The application programmer uses SSL_DATA_BLOCK_S structure to pass the opaque certificate data to the corresponding initialization routine which can decode and interpret information within the certificate. Table 3 lists the SSL_DATA_BLOCK_S structure members.

Table 3. SSL_DATA_BLOCK_S Structure Members

Member Name	Description
pData	References a block of opaque data bytes.
Length	Number of bytes of data in the opaque data block.

SSL_BN

```
typedef struct EZ80_BN
{
  BYTE *pNum;       // Pointer to the number
  WORD NumBytes;    // Number of Bytes in the number
  WORD MaxBytes;    // Maximum number of bytes the
                    // number buffer can contain
} EZ80_BN;
```

Big numbers are used by the public key infrastructure (PKI) algorithms that SSL uses for authentication. All eZ80 big numbers are stored in little-endian format; i.e., the least-significant byte of the big number is stored in the lowest memory location, and the most-significant byte of the big number is stored in the highest memory location. All eZ80 numbers are interpreted as unsigned values.

Big numbers are an extension of the standard C data types implemented by the ZDSII compiler. For example, the value `0x11223344` can be represented as an unsigned long or as an `SSL_BN` of length 4 (`NumBytes=4`). The maximum length of an SSL big number is determined by the value of the `MAX_EZ80_BN_BYTES` macro (currently defined as 4096).

The SSL layer allocates big numbers from the run-time heap. Arithmetic operations performed on big numbers increase the size of the number. For example, if a 64-byte value is multiplied by a 64-byte value, the result can be up to 128 bytes long. If the size of the big number is not large enough to contain the result, then a much bigger number must be allocated. Therefore, the SSL layer will allocate more space than required to store the big number. In this instance, the `MaxBytes` field will be larger than the `NumBytes` field.

Table 4. EZ80_BN Structure Members

Member Name	Description
`pNum`	References the little-endian array of bytes representing the value of the big number.
`NumBytes`	Indicates the number of bytes in the big number.
`MaxBytes`	Indicates the maximum size (in bytes) of the big number. A big number cannot grow beyond `MaxBytes` without being reallocated.

ASN1_Enc_Data

```
typedef struct ASN1_ENC_DATA_S
{
  SSL_BYTE           Encoding;
  SSL_WORD           Length;
  SSL_BYTE           *pData;
} ASN1_ENC_DATA;
```

An ASN1_ENC_DATA block describes a block of ASN.1-encoded data. The data must be encoded using ASN.1 distinguished encoding rules (DER) and can be Base64-encoded. ASN.1 DER-encoded data is indicated when the encoding member has the value DER_ENCODED_DATA. Base64 ASN.1 DER-encoded data is indicated when the encoding member has the value BASE64_DER_ENCODED_DATA (or equivalently PEM_ENCODED_DATA).

This data structure contains the Ephemeral Diffie-Hellman parameters used by the SSLv3 and TLSv1 servers.

Table 5. ASN1_ENC_DATA Structure Members

Member Name	Description
Encoding	Specifies the encoding method of the data block. The only permissible values are: DER_ENCODED_DATA, BASE64_DER_ENCODED_DATA and PEM_ENCODED_DATA.
pData	References a block of opaque data bytes.
Length	Number of bytes of data in the opaque data block.

Cert_Chain

```
typedef struct CERT_CHAIN
{
  SSL_BYTE          Depth;
  SSL_BYTE          Encoding;       // DER or PEM
  SSL_X509_S *pX509;                // System defined
(use
                                    // NULLPTR)
  SSL_DATA_BLOCK_S PrivKey;         // Must be in
same// encoding scheme
  SSL_DATA_BLOCK_S Cert[MAX_CERTIFICATE_CHAIN_LENGTH];
} CERT_CHAIN;
```

With the ZTP Network Security SSL Plug-In, all SSL servers are required to have a X.509 certificate and a corresponding private key (anonymous cipher suites are not supported). Additionally, the SSLv3 and TLSv1 protocols allow the server to present prospective clients with a hierarchical chain of certificates which can be used to authenticate the server. In SSLv2, the certificate chain is permitted to have only one entry.

The `CERT_CHAIN` data structure is used to communicate the server's certificate chain to the underlying SSL protocol during initialization.

Table 6. CERT_CHAIN Structure Members

Member Name	Description
`Depth`	Identifies the number of certificates in the `Cert` array. The value of `MAX_CERTIFICATE_CHAIN_LENGTH` macro (currently defined to 4) defines the maximum number of entries permitted in the `Cert` array. For SSLv2 servers, this field must have a value of 1.
`Encoding`	Specifies the encoding of all certificates in the `Cert` array and the `PrivateKey`. The only permissible values are: `DER_ENCODED_DATA`, `BASE64_DER_ENCODED_DATA` and `PEM_ENCODED_DATA`.

Table 6. CERT_CHAIN Structure Members (Continued)

Member Name	Description
pX509	This member is initialized by the SSL layer during initialization. When defining a CERT_CHAIN data structure, set the value of this member to NULLPTR.
PrivateKey	This SSL_DATA_BLOCK_S structure contains the private key corresponding to the first certificate in the Cert array (i.e., the server's certificate). The data encoding of the private key is specified by the Encoding structure member.
Cert[]	This is an array of certificate ordered such that the first certificate in the chain, Cert[0], contains the server's X.509 certificate. Subsequent entries in the chain contain the X.509 certificate of the entity that signed the previous certificate. The last entry in the chain is typically the self-signed certificate of a certificate authority. The data encoding of all certificates is specified by the Encoding structure member. The number of entries in the Cert array will correspond to the value of the Depth parameter.

SSL_X509_S

```
typedef struct SSL_X509_S
{
  SSL_BYTE          Flags;
  SSL_BYTE          Version;
  SSL_BN            *pSerialNumber;
  SSL_BYTE          SignatureID;
  SSL_DATA_BLOCK_S  Issuer;
  SSL_DATA_BLOCK_S  Subject;
  SSL_DWORD         NotValidBefore;
  SSL_DWORD         NotValidAfter;
  SSL_BYTE          PKAlgorithmID;
  SSL_BN            *pSignature;
  SSL_DATA_BLOCK_S  DerCert;
  SSL_BN            *pPublicKeyInfo;
  SSL_DATA_BLOCK_S  KeyExchangeParams;
  SSL_DATA_BLOCK_S  TBSCert;
  SSL_PKI           PKI;
} SSL_X509_S;
```

Internally, the ZTP Network Security SSL Plug-In uses an `SSL_X509_S` data structure to contain information about ASN.1 DER-encoded X.509 certificates.

Application programmers must not initialize `SSL_X509_S` data structures. These data structures will be created and destroyed by the SSL layer during system initialization and while processing a certificate chain received from a remote SSL server. When a certificate chain is processed, the last entry in the chain is typically a self-signed root certificate.

The ZTP Network Security SSL Plug-In creates an `SSL_X509_S` structure and pass it as a parameter on the call to the `VerifyCertificate` callback routine. The data in the `SSL_X509_S` certificate must be treated as read-only when examined by an application program in the `VerifyCertificate` callback routine. If any data in the `SSL_X509_S` certificate is modified, the

SSL layer might fail to operate and can even crash the entire system. Table 7 lists the SSL_X509_S structure members.

Table 7. SSL_X509_S Structure Members

Member Name	Description
`Flags`	Bits within this field are used to identify the results of processing on this certificate, see Table 8 for more information.
`Version`	Indicates the X.509 certificate version number.
`pSerialNumber`	The `pNum` member of the `SSL_BN` structure references the serial number of the X.509 certificate.

`SignatureID`	This value indicates the algorithm used to generate the signature. The ZTP Network Security SSL Plug-In understands only the following signature algorithms:
	`OID_MD5_WITH_RSA_ENCRYPTION`
	`OID_SHA1_WITH_RSA_ENCRYPTION`
	`OID_ID_DSA_WITH_SHA1`
	If the certificate uses any other value for the `SignatureID`, it is set to `OID_UNKNOWN`.

`NotValidBefore`	32-bit time stamp (UTC time) indicating the date and time at which or after which, the certificate is deemed valid. This field will be 0 if the certificate uses `GeneralizedTime` or the SSL layer is unable to parse the certificate time stamp.
`NotValidAfter`	32-bit time stamp (UTC time) indicating the date and time after which, the certificate is no longer valid. This field will be 0 if the certificate uses `GeneralizedTime` or the SSL layer is unable to parse the certificate time stamp.
`Issuer`	This field contains the ASN.1 DER- encoded data that identifies the entity that issued this certificate.
`Subject`	This field contains the ASN.1 DER- encoded data which identifies the subject of the certificate. In the context of SSL, the subject is the server that presented this certificate.

Table 7. SSL_X509_S Structure Members (Continued)

Member Name	Description
PKAlgorithmID	Identifies the PKI algorithm which can be used to authenticate the certificate owner (Subject). The ZTP Network Security SSL Plug-In understands only the following PKI algorithms: OID_RSA_ENCRYPTION OID_ID_DSA OID_DH_PUBLIC_NUMBER
pSignature	Contains a digitally signed string that contains the digest of the TBSCert data. The digest algorithm (hash function) and signature algorithm are indicated by the SignatureID field.
DerCert	References an opaque data block of the entire ASN.1 DER-encoded X.509 certificate.
pPublicKeyInfo	Contains the ASN.1 DER- encoded public key of the entity presenting this certificate (Subject). The public key algorithm is indicated by the PKIAlgorithmID structure member.
KeyExchangeParams	Contains the ASN.1 DER- encoded parameters associated with the PKIAlgorithmID.
TBSCert	References the ASN.1 DER data which is digested and then signed by the certificate Issuer to produce the pSignature certificate. The TBSCert contains the raw encoding of the certificate version, serial number, issuer, subject and public key as well as other parameters.
PKI	See the SSL_PKI structure definition on page 15.

Table 8. SSL_X509_S Flags

Flag	Description
SSL_X509_PARSED_OK	If this flag is set, it indicates that the ASN.1 certificate encoding is acceptable and contains the following fields: • Version • Serial Number • Signature Algorithm • Issuer • Valid From • Valid Until • Subject • Public Key Algorithm (and any associated parameters) • Public Key • Signature
SSL_X509_DATE_VALID	If this flag is set, it indicates that the current system time stamp is between the Not Valid Before and Not Valid After fields of the X.509 certificate.
SSL_X509_SIGNATURE_VERIFIED	If this flag is set, it indicates that the Signature field matches the digest of the TBS Cert data
SSL_X509_SELF_SIGNED	If this flag is set, it indicates that the certificate is self-signed (i.e., the Issuer and Subject are identical).
SSL_X509_PERMANENT	If this flag is set, the certificate will not be deleted by the SSL layer after it is parsed. Application programs are permitted to set this flag in the VerifyCertificate callback.

Table 8. SSL_X509_S Flags (Continued)

Flag	Description
SSL_X509_UNKNOWN_SIG_ALG	If this flag is set, it indicates that the SSL layer does not support the SignatureID algorithm. In this instance, the SignatureID will be OID_UNKNOWN.
SSL_X509_TRUSTED	If this flag is set, it indicates that the SSL layer is uses this certificate. The SSL layer will trust certificates that have the following flags set: SSL_X509_PARSED_OK, SSL_X509_DATE_VALID and X509_SIGNATURE_VERIFIED unless the VerifyCertificate callback routine indicates that the certificate should not be trusted. Conversely any certificate that the VerifyCertificate callback routine indicates to be trusted will always be trusted regardless of the certificate's flags.

SSL_PKI

```
typedef struct SSL_PKI
{
  SSL_BYTE          Algorithm;
  PKI_Encrypt       Encrypt;        // RSA only
  PKI_Decrypt       Decrypt;
  PKI_Sign          Sign;           // RSA & DSA
  PKI_Verify        Verify;
  PKI_Agree         Agree;          // DH & DHE
  PKI_Free          Free;
  union
  {
   RSA_PARAMS Rsa;
   DSA_PARAMS Dsa;
   DH_PARAMS Dh;
  } Params;
} SSL_PKI;
```

Within each X.509 certificate structure created by the ZTP Network Security Plug-In, there is an `SSL_PKI` structure. This `SSL_PKI` structure is used in conjunction with the associated certificate to authenticate the owner of the certificate and create a secret shared key used by the SSL client and server to encrypt all application data sent during the SSL session. This structure must not be modified by application programs.

The set of valid function pointers depend on the PKI algorithm extracted from the associated certificate. This algorithm determines which cryptographic operations are performed with the certificate.

Note: The Sign and Decrypt operations can be performed only if private key corresponding to the public key in the certificate is known.

Table 9. SSL_PKI Structure Members

Member Name	Description
Algorithm	Identifies the PKI algorithm which will be used to authenticate the owner of the certificate and establish a secret shared key used to encrypt all SSL session data. In this implementation, the only permissible values are: SSL_PKI_RSA, SSL_PKI_DHE_DSS and SSL_PKI_DH.
Encrypt	Used to encrypt a block of data with the RSA public key found in the certificate. This function pointer is only valid when the Algorithm member is SSL_PKI_RSA.
Decrypt	Used to decrypt a block of data with the RSA private key. This function pointer is valid only when the Algorithm member is SSL_PKI_RSA.
Sign	Used to sign a block of data using either an RSA or a DSA private key. This function pointer is valid when the Algorithm member is SSL_PKI_RSA or SSL_PKI_DHE_DSS.
Verify	Used to verify the signature of a block of data using either an RSA or a DSA public key. This function pointer is valid when the Algorithm member is SSL_PKI_RSA or SSL_PKI_DHE_DSS.
Agree	Used to establish a shared secret via the Diffie-Hellman key agreement algorithm. This function pointer is only valid when the Diffie-Hellman key agreement algorithm is used.
Free	Used to release the resource held in the Params member.
Params	The contents of this field will be either an RSA_PARAMS, DSA_PARAMS, or DH_PARAMS structure depending on the Algorithm structure member.

RSA_Params

```
typedef struct RSA_PARAMS
{
  SSL_BN             *pModulus;
  SSL_BN             *pPublicExp;
  SSL_BN             *pPrivateExp;
  SSL_BN             *pPrime1;
  SSL_BN             *pPrime2;
  SSL_BN             *pExp1;
  SSL_BN            *pExp2;
  SSL_BN            *pCoefficient;
} RSA_PARAMS;
```

This structure is used to contain the RSA public and private key corresponding to an X.509 certificate in which the `PKIAlgorithmID` is `OID_RSA_ENCRYPTION`. The owner of the certificate possesses both the public and private key. The entity that receives the certificate will only be able to extract the public key parameters.

DSA_Params

```
typedef struct      DSA_PARAMS
{
  SSL_BN *p;        // Prime Modulus
  SSL_BN *g;        // Generator (base used for
                    // exponentiation)
  SSL_BN *q;        // prime factor of p
                    // (such that p = j*q + 1)
  SSL_BN *x;        // DSA Private Key
  SSL_BN *Y;        // DSA Public Value (Y=g^x mod p)
} DSA_PARAMS;
```

This structure contains the DSA public value and private key corresponding to an X.509 certificate in which the `PKIAlgorithmID` is `OID_ID_DSA`. The owner of the certificate will possess both the public and private key. The entity that receives the certificate will only be able to extract the public key parameters.

DH_Params

```
typedef struct DH_PARAMS
{
  SSL_BN *p;            // Prime Modulus
  SSL_BN *g;            // Generator (base used for
                        // exponentiation)
  SSL_BN *q;            // prime factor of p (such
                        // that p = j*q + 1)
  SSL_BN *x;            // DH Private Key
  SSL_BN *Y;            // DH Public value(Y = g^x mod
                        // p)
  SSL_BOOL IsEphemeral; // If TRUE, x (and Y) can be
                        // changed
} DH_PARAMS;
```

This structure is used to contain the DH public value and private key corresponding to an X.509 certificate in which the `PKIAlgorithmID` is `OID_DH_PUBLIC_NUMBER`. The owner of the certificate possess both the public and private key. The entity that receives the certificate will only be able to extract the public key parameters.

PKI_Init

```
typedef SSL_STATUS (* PKI_Init)  (struct SSL_X509_S *
pX509, struct ASN1_ENC_DATA_S * pPrivateKey);
```

The PKI_Init structure is a function pointer used in the construction of the PKIGen table. Each of the PKI algorithms has one PKI_init function pointer supported by the ZTP Network Security SSL Plug-In. These function pointers are:

- rsa_init
- dsa_init
- dh_init

There is also a NullPki_init function pointer which can be used in the PKIGen table to disable support for the corresponding PKI algorithm.

The default PKIGen table found in the pki_conf.c file is shown below.

```
PKI_Init              PkiGen[SSL_MAX_PKI] =
{
  NullPki_init,
  rsa_init,           // To disable RSA, replace this
                      // entry with NullPki_init
  dsa_init,           // To disable DSA, replace this
                      // entry with NullPki_init
  dh_init             // To disable DH, replace this
                      // entry with NullPki_init
};
```

To disable support for the corresponding PKI algorithm, replace the algorithm entry in the PKIGen table with NullPki_init.

PKI_DheInit

```
typedef SSL_STATUS (* PKI_DheInit) (struct SSL_X509_S
*pX509, struct ASN1_ENC_DATA_S *pDheParams, DH_PARAMS
*pDh );
```

To enable Ephemeral Diffie-Hellman support, the global function pointer variable must reference the `dhe_init` routine. To disable support for Ephemeral Diffie-Hellman cipher suites, the global function pointer is set to `NULLPTR`.

SSL_Hash

```
typedef struct SSL_HASH_S
{
  SSL_BYTE          HashType;
  SSL_WORD          DigestSize;
  SSL_WORD          BlockSize;
  HASH_UPDATE       Update;
  HASH_FINAL        Final;
  HASH_FREE         Delete;
} SSL_HASH;
```

The SSL_HASH structure is used to access digest functions supported by the ZTP Network Security SSL Plug-In. Application programmers also use these digest functions. The set of digest functions included in a project is determined by the entries in the HashGen array. For more information, see the HASH_New structure definition on page 24.

An SSL_HASH structure is dynamically created by calling a function pointer of type HASH_NEW.

Table 10. SSL_HASH Structure Members

Member Name	Description
HashType	Identifies the digest algorithm implemented by this structure. The HashType is set when the structure is created. The only permissible values are: SSL_HASH_NULL SSL_HASH_MD5 SSL_HASH_HMAC_MD5 SSL_HASH_SHA1 SSL_HASH_HMAC_SHA1
DigestSize	Identifies the number of bytes of output produced by the digest algorithm.
BlockSize	Identifies the block size of the digest algorithm.

Table 10. SSL_HASH Structure Members

Member Name	Description
Update	Function pointer to supply more input data to the digest algorithm.
Final	Function pointer that returns the output of the digest algorithm.
Free	Function pointer to release the resources associated with this structure.

The prototypes for the function pointers are:

```
typedef SSL_STATUS (*HASH_UPDATE)(struct SSL_HASH_S
*pHash, SSL_BYTE *pData, SSL_WORD Length);

typedef SSL_STATUS (*HASH_FINAL) (struct SSL_HASH_S
*pHash, SSL_BYTE *pData, SSL_WORD Length);

typedef void (*HASH_FREE) (struct SSL_HASH_S *pHash);
```

By using the SSL_HASH structure, a single orthogonal API is used to generate the digests using any of the supported digest algorithms. For more information, see the HASH_New structure definition on page 24.

HASH_New

```
typedef struct SSL_HASH_S * (* HASH_NEW) ( SSL_BYTE *
pKey, SSL_WORD KeyLen );
```

The HASH_NEW function pointer is used in the construction of the HashGen table, see hash_conf.c. There is one entry in the HashGen table for each digest algorithms supported by ZTP Network Security SSL Plug-In. The SSL library uses entries in the HashGen table to dynamically create SSL_HASH structures based on the appropriate digest algorithm. Application programs also use the HashGen table to allocate SSL_HASH structures for their own purposes.

The default HashGen table is shown below (see hash_conf.c):

```
HASH_NEW HashGen[ SSL_MAX_HASH ] =
{
  NullHash_New,
  MD5_New,
  HMAC_MD5_New,
  SHA1_New,
  HMAC_SHA1_New
};
```

Support for a digest algorithm is removed by replacing the corresponding table entry with NullHash_New. However, some versions of the SSL protocol do not function unless the corresponding digest algorithm is available. Table 11 shows which digest algorithms must be supported to allow proper operation of the corresponding SSL protocol.

Table 11. Digest Routines by SSL Protocol Version

SSL Protocol	Mandatory Digest Algorithms
SSLv2	NullHashNew, MD5
SSLv3	NullHashNew, MD5, SHA1
TLSv1	NullHashNew, MD5, HMAC_MD5, SHA1, HMAC_SHA1

Note: The HTTP server in ZTP v2.3.0 and later releases also uses the MD5 algorithm to authenticate HTTP clients. This authentication does not use SSL, but uses a special version of the `HashGen` array which contains only an entry for the MD5 algorithm (all other entries are specified as NULL). Therefore, the `hash_conf.c` file contains two versions of the `HashGen` array. The first version of the `HashGen` array is used only in ZTP projects which do not include SSL but require HTTP authentication. These projects must define a symbol named `MD5_HTTP` to use the non-SSL version of the `HashGen` array. The second version of the `HashGen` array is used when SSL support is required. This applies to ZTP systems regardless of whether HTTP authentication is required or not. To use the SSL version of the `HashGen` array, the `MD5_HTTP` symbol must not be defined.

The `HashGen` table provides an orthogonal API that can be used to digest data using any of the supported algorithms.

For example, the code fragment below will compute the MD5 digest of "`abc`".

```
SSL_BYTE Buffer[ 100 ];
SSL_HASH *pHash;
pHash = HashGen[ SSL_HASH_MD5 ]( NULLPTR, 0 );
pHash->Update( pHash, "abc", 3 );
pHash->Final( pHash, Buffer, pHash->DigestSize );
pHash->Free( pHash );
```

The same code fragment is used to compute the `SHA1` digest of "`abc`" by changing the index into the `HashGen` table from `SSL_HASH_MD5` to `SSL_HASH_SHA1`. The `HashGen` table is indexed using one of the following macros:

- `SSL_HASH_NULL`
- `SSL_HASH_MD5`
- `SSL_HASH_HMAC_MD5`

- SSL_HASH_SHA1
- SSL_HASH_HMAC_SHA1

The pKey and KeyLen parameters are used only for the *keyed* digest routines SSL_HASH_HMAC_MD5 and SSL_HASH_HMAC_SHA1. For MD5 and SHA1, the pKey and KeyLen parameters must be set to NULLPTR and 0, respectively.

SSL_Cipher

```
typedef struct      SSL_CIPHER
{
  SSL_BYTE          Algorithm;
  SSL_WORD          KeyLength;
  SSL_WORD          BlockSize;
  SSL_BYTE          CipherMode;
  CIPHER_TRANSFORM  Transform;
  CIPHER_FREE       Delete;
} SSL_CIPHER;
```

A cipher is used to encrypt and decrypt data using a symmetric shared secret. The encryption process transforms a plain-text data block into cipher-text using a secret key. The decryption process transforms the cipher-text back into plain-text using the same secret key. Because both entities possess the same key, the cipher algorithm is said to be symmetric.

The ZTP Network Security SSL Plug-In uses an `SSL_CIPHER` structure to encrypt and decrypt data using any of the supported symmetric cipher algorithms. Application programs can also use the symmetric cipher algorithms for their own purpose.

The set of cipher algorithms included in a project is determined by the entries in the `CipherGen` array; see the CIPHER_New structure definition on page 30 for more information. An `SSL_CIPHER` structure is created by calling a function pointer of type `CIPHER_NEW`.

Table 12. SSL_CIPHER Structure Members

Member Name	Description
Algorithm	Identifies the symmetric cipher algorithm implemented by this structure. The Algorithm type is set when the structure is created. The only permissible values are: SSL_CIPHER_NULL, SSL_CIPHER_RC4, SSL_CIPHER_DES, SSL_CIPHER_3DES and SSL_CIPHER_AES.
KeyLength	Identifies the number of bytes in the symmetric key. This is independent of the value of the key (for example, if the key is 0x00,0x00,0x00,0x00, then the KeyLength will be 4).
BlockSize	For stream ciphers (for example, RC4), this field will be 1. For BlockCiphers (DES, 3DES and AES) identifies the number of input bytes transformed by one invocation of the algorithm.
CipherMode	Indicates if the SSL_CIPHER is used to encrypt or decrypt data. The only permissible values are: SSL_CIPHER_MODE_BULK_DECRYPT and SSL_CIPHER_MODE_BULK_ENCRYPT.
Transform	Function pointer used to either encrypt or decrypt a given block of data.
Delete	Function pointer to release resources associated with this structure.

The prototypes for the function pointers are:

```
typedef SSL_STATUS  ( *CIPHER_TRANSFORM )
(
  struct SSL_CIPHER *pCipher,
  SSL_BYTE          *pDecoded,
  SSL_BYTE          *pEncoded,
  SSL_WORD          Length
);
typedef void        (*CIPHER_FREE)
```

```
(
  struct SSL_CIPHER *pCipher
);
```

By using the `SSL_CIPHER` structure, a single orthogonal API is used to encrypt or decrypt data using any of the supported symmetric cipher algorithms. For more information, see the CIPHER_New structure definition on page 30.

CIPHER_New

```
typedef struct SSL_CIPHER *(* CIPHER_NEW )
(
  SSL_BYTE           CipherMode,
  SSL_BYTE          *pKey,
  SSL_WORD           KeyLen,
  SSL_BYTE           *InitVector,
  SSL_WORD           IV_Len
);
```

The `CIPHER_NEW` function pointer is used in the construction of `CipherGen` table, see `cipher_conf.c`. There is one entry in the `CipherGen` table for each symmetric `CipherGen` algorithms supported by ZTP Network Security SSL Plug-In. The SSL library uses entries in the `CipherGen` table to create `SSL_CIPHER` structures based on the appropriate cipher algorithm. You can also use the `CipherGen` table to allocate `SSL_CIPHER` structures for their own purposes.

The default `CipherGen` table is shown below, see `cipher_conf.c`.

```
CIPHER_NEW          CipherGen[ SSL_MAX_CIPHERS ] =
{
  NullCipher_New,
  RC4_New,
  DES_New,
  DES3_New,
  AES_New
};
```

Support for a cipher algorithm is removed by replacing the corresponding table entry with `NullCipher_New`. However, the SSL protocol requires both the client and server to support at least one common cipher algorithm. If a given client and server do not share a common cipher algorithm, it will not be possible to establish an SSL session.

The `CipherGen` table provides an orthogonal API that is used to encrypt or decrypt data using any of the supported algorithms.

For example, the code fragment below will encrypt the plain text message "Hello" using an RC4 symmetric key of length 16 bytes (128 bits).

```
#define KEY_SIZE 16
extern SSL_BYTE *pKey;
extern SSL_BYTE *pIV;
SSL_BYTE Buffer[ 100 ];
SSL_Cipher *pCipher;

pCipher = CipherGen[ SSL_CIPHER_RC4 ]
(
  SSL_CIPHER_MODE_BULK_ENCRYPT,
  pKey, KEY_SIZE,
  pIV, RC4_IV_SIZE_BYTES );

pCipher->Transform( pCipher, Buffer, "Hello", 5 );
pCipher->Delete( pCipher );
```

The same code fragment is used to encrypt data using AES by changing the index into the `CipherGen` table from `SSL_CIPHER_RC4` to `SSL_CIPHER_AES`. In addition, the `AES_IV_SIZE_BYTES` macro is used to specify the size of the AES initialization vector. The `CipherGen` table is indexed using one of the following macros:

- `SSL_CIPHER_NULL`
- `SSL_CIPHER_RC4`
- `SSL_CIPHER_DES`
- `SSL_CIPHER_3DES`
- `SSL_CIPHER_AES`

Note: The `InitVector` and `IV_Len` parameters are not required by the RC4 cipher algorithm and will be ignored. Table 13 describes the limitations and restrictions on the input parameters based on the cipher algorithm.

Table 13. Key and Initialization Vector Sizes by Cipher

Cipher	Key Size	Initialization Vector Size
RC4	Arbitrary SSL (between 1 and 256 bytes) uses 16-byte RC4 keys (i.e., RC4_128)	Must be 0; i.e., RC4_IV_SIZE_BYTES
DES	Must be DES_KEY_SIZE_BYTES	Must be DES_IV_SIZE_BYTES
3DES	Must be DES3_KEY_SIZE_BYTES	Must be DES3_IV_SIZE_BYTES
AES	Must be one of these values: AES_128_KEY_SIZE_BYTES AES_192_KEY_SIZE_BYTES AES_256_KEY_SIZE_BYTES SSL does not use AES_192	AES_IV_SIZE_BYTES

SSL_CS_Info

```
typedef struct SSL_CS_INFO
{
  SSL_WORD          CipherSuite;
  SSL_BYTE          KeyAlg;
  SSL_BYTE          CipherAlg;
  SSL_BYTE          HashAlg;
  SSL_BOOL          IsExport;
  SSL_BYTE          KeySize;
  SSL_BYTE          IVSize;
  SSL_BYTE          MacSize;
  SSL_BOOL          IsValid;
} SSL_CS_INFO;
```

The SSL handshake protocols are used to determine a set of compatible cryptographic algorithms which are used for authentication, privacy (i.e., encryption) and message integrity. These goals are accomplished with a key exchange algorithm, a cipher and a digest (hash) algorithm. This tuple is called a Cipher Suite (SSLv2 used the term Cipher Kind to describe a cipher and a digest function as RSA was the only key exchange algorithm supported.)

The ZTP Network Security SSL Plug-In uses a `SSL_CS_INFO` structure to identify a particular cipher suite regardless the version of the SSL protocol used. `SSL_CS_INFO` structures are also used to build tables of cipher suites that identify the combination of key exchange algorithm, cipher and digest supported by each of the SSL protocols.

Table 14. SSL_CS_INFO Structure Members

Member Name	Description
CipherSuite	Code used to identify a particular Cipher Suite (see Cipher-Suite.h).
KeyAlg	Identifies the PKI key exchange algorithm for this Cipher Suite. Valid values are: SSL_PKI_RSA SSL_PKI_DHE_RSA SSL_PKI_DHE_DSS SSL_PKI_DH
CipherAlg	Identifies the symmetric cipher used to encrypt data. Valid values are: SSL_CIPHER_RC4 SSL_CIPHER_DES SSL_CIPHER_3DES SSL_CIPHER_AES
HashAlg	Identifies the digest algorithm for this Cipher Suite. Valid values are: SSL_HASH_MD5 SSL_HASH_SHA1
IsExport[1]	Set to TRUE if the size of the private key used in the key exchange algorithm is less than or equal to 512 bits.
KeySize[2]	Indicates the number of bits in the (expanded) symmetric key used by the CipherAlg.
IVSize	Size of the Initialization Vector for the Cipher Algorithm. This initialization vector is internally generated by the SSL protocol.

Notes:

1. The SSLv3 and TLSv1 servers implemented in the ZTP Network Security SSL Plug-In do not support the generation of temporary RSA or Ephemeral Diffie-Hellman keys. Therefore, these cipher suites will be selected only in the Server Hello message, if the size of the modulus in the corresponding public key is less than or equal to 512 bits. However, the SSLv3 and TLSv1 clients implemented in the ZTP Network Security SSL Plug-In will establish connections with servers using temporary RSA keys.
2. All export ciphers initially generate a 40-bit symmetric key expanded to the default key size for the given cipher algorithm.

Table 14. SSL_CS_INFO Structure Members (Continued)

Member Name	Description
`MacSize`	Size of the message authentication code used to verify the integrity application data.
`IsValid`	If TRUE, indicates that this Cipher Suite is used to establish an SSL session. If FALSE, this Cipher Suite cannot be used to establish an SSL session.

Notes:

1. The SSLv3 and TLSv1 servers implemented in the ZTP Network Security SSL Plug-In do not support the generation of temporary RSA or Ephemeral Diffie-Hellman keys. Therefore, these cipher suites will be selected only in the Server Hello message, if the size of the modulus in the corresponding public key is less than or equal to 512 bits. However, the SSLv3 and TLSv1 clients implemented in the ZTP Network Security SSL Plug-In will establish connections with servers using temporary RSA keys.
2. All export ciphers initially generate a 40-bit symmetric key expanded to the default key size for the given cipher algorithm.

SSL Constants

This section presents tables which list significant constants used by the SSL handshake protocols and cryptographic libraries. Table 15 lists the types of SSL constants.

Table 15. Constants by Category

Type of Constant
Status Codes
Digest Algorithm Identifiers
Cipher Algorithm Identifiers
PKI Algorithm Identifiers in PKIGen Array
PKI Algorithm Identifiers in SSL_CS_INFO Structure
X.509 Public Algorithm Identifiers

Table 16 through Table 21 lists the SSL constants and their values.

Table 16. Status Codes

Constant	Value
SSL_SUCCESS	0
SSL_FAILURE	–1

Table 17. Digest Algorithm Identifiers

Constant	Value
SSL_HASH_NULL	0
SSL_HASH_MD5	1
SSL_HASH_HMAC_MD5	2

Table 17. Digest Algorithm Identifiers (Continued)

Constant	Value
SSL_HASH_SHA1	3
SSL_HASH_HMAC_SHA1	4

Table 18. Cipher Algorithm Identifiers

Constant	Value
SSL_CIPHER_NULL	0
SSL_CIPHER_RC4	1
SSL_CIPHER_DES	2
SSL_CIPHER_3DES	3
SSL_CIPHER_AES	4

Table 19. PKI Algorithm Identifiers in PKIGen Array

Constant	Value
SSL_PKI_NULL	0
SSL_PKI_ID_RSA	1
SSL_PKI_ID_DSA	2
SSL_PKI_ID_DH	3

Table 20. PKI Algorithm Identifiers in SSL_CS_INFO Structure

Constant	Value
SSL_PKI_NULL	0
SSL_PKI_RSA	1
SSL_PKI_DH	2

Table 20. PKI Algorithm Identifiers in SSL_CS_INFO Structure

Constant	Value
SSL_PKI_DHE_RSA	3
SSL_PKI_DHE_DSS	4

Table 21. X.509 Public Algorithm Identifiers

Constant	Value
OID_RSA_ENCRYPTION	1
OID_ID_DSA	2
OID_DH_PUBLIC_NUMBER	3

API Definitions

This chapter describes the APIs that are provided by ZTP Network Security SSL Plug-In. Click the links below to learn more about each of these APIs.

Initialize_SSL

Include

```
#include “SSL.h”
```

Library

`ssl_rzk.lib` for use with ZTP 1.6.0 and later

Prototype

```
SSL_STATUS Initialize_SSL(void);
```

Description

This `Initialize_SSL()` API initializes the SSL interface layer exposed to upper layer applications. This API is called before using any other SSL-related APIs. The SSL interface layer is similar to the TCP interface of the underlying ZTP system but provides seamless integration of the SSL handshake protocols. Therefore, application programs which are written to use the TCP interface can be easily migrated to SSL. For more information, refer to ZTP Network Security SSL Plug-In User Manual (UM0201).

Argument(s)

None

Return Value(s)

`SSL_SUCCESS`	Indicates that the API is executed successfully.
`SSL_FAILURE`	Indicates that the API is not executed successfully. Possible reasons for the failure are insufficient memory or the unavailability of kernel resources to represent all SSL devices/sockets.

Usage Scenario

After `Initialize_SSL()` is called and before any upper layer applications attempt to use the SSL interface, it is necessary to initialize one or more of the SSL handshake protocols by calling any of the following APIs:

- SSL2_ClientInit
- SSL2_ServerInit
- SSL3_ClientInit
- SSL3_ServerInit
- TLS1_ClientInit
- TLS1_ServerInit

Example

The following example initializes the SSL interface layer.

```
Initialize_SSL();
```

See Also

SSL2_ClientInit SSL2_ServerInit
SSL3_ClientInit SSL3_ServerInit
TLS1_ClientInit TLS1_ServerInit

SSL2_ClientInit

Include

```
#include “SSL.h”
```

Library

`ssl_rzk.lib` for use with ZTP 1.6.0 and later.

Prototype

```
SSL_STATUS SSL2_ClientInit( void );
```

Description

This API initializes the client-side of the SSL version 2 handshake protocol. If this API is not called then it is not possible for the ZTP Network Security SSL Plug-In to establish any SSLv2 sessions in client mode. In client-only mode, applications will only be able to establish SSLv2 sessions with remote SSLv2 servers.

The SSLv2 protocol implemented in the ZTP Network Security SSL Plug-In allows multiple client and server sessions to operate at the same time, but this causes an extra processing burden on the system.

Argument(s)

None.

Return Value(s)

`SSL_SUCCESS`	Indicates that the API is executed successfully.
`SSL_FAILURE`	Indicates that the API did not complete successfully.

Usage Scenario

This API must only be called after calling the Initialize_SSL API (see page 40). This API must be called before establishing any SSL sessions with remote SSLv2 servers. Do not call this API multiple times.

Example

The following example initializes the client-side of the SSLv2 handshake protocol.

```
Initialize_SSL();
SSL2_ClientInit();
```

See Also

Initialize_SSL	SSL2_ServerInit
SSL3_ClientInit	SSL3_ServerInit
TLS1_ClientInit	TLS1_ServerInit

SSL2_ServerInit

Include

```
#include “SSL.h”
```

Library

`ssl_rzk.lib` for use with ZTP 1.6.0 and later.

Prototype

```
SSL_STATUS SSL2_ServerInit
(
  struct CERT_CHAIN *pCertChain,
  ASN1_ENC_DATA     *pDheParams
);
```

Description

The `SSL2_ServerInit()` API initializes the server-side of the SSL version 2 handshake protocol. If this API is not called then it is not possible for the ZTP Network Security SSL Plug-In to establish any SSLv2 sessions in server mode. In server mode, applications will only be able to establish SSLv2 sessions initiated by remote clients.

The SSLv2 protocol implemented in the ZTP Network Security SSL Plug-In allows multiple client and server sessions to operate simultaneously, but this causes an extra processing burden on the system.

The `pCertChain` parameter references a certificate chain which contains the server’s X.509 certificate. For SSLv2, the certificate chain must only contain a single RSA certificate and private key. This certificate will be shared by all SSLv2 servers. After processing the certificate in the certificate chain, the SSL library calls the `VerifyCertificate` callback routine to determine if the certificate should be trusted. If the `VerifyCertificate` callback returns `SSL_SUCCESS`, the SSL library accepts the certificate and marks it as a permanent certificate.

Since SSLv2 does not support the Diffie-Hellman key exchange algorithm, the pDheParams parameter is specified as NULLPTR.

Argument(s)

pCertChain	Contains the SSLv2 server's RSA certificate and private key. The chain must include only one certificate.
pDheParams	For SSLv2 this parameter is initialized to NULLPTR.

Return Value(s)

SSL_SUCCESS	Indicates that the API is executed successfully.
SSL_FAILURE	Indicates that the API is not completed successfully. Possible reasons for the failure are invalid certificate, or more than one certificate in the certificate chain.

Usage Scenario

This API must be called only after calling the Initialize_SSL API (see page 40). This API must be called before starting any SSLv2 servers (including the supplied HTTPS server). Do not call this API multiple times.

This routine requires significant stack space. Zilog recommends that this API has to be called only from a task which has a 1 KB run-time stack.

Example

The following example initializes the server side of the SSLv2 handshake protocol.

```
extern CERT_CHAIN SSL2_CertChain;
Initialize_SSL();
SSL2_ServerInit(&SSL2_CertChain, NULLPTR);
```

See Also

Initialize_SSL
SSL2_ClientInit
TLS1_ServerInit
VerifyCertificate

SSL3_ClientInit

Include
```
#include “SSL.h”
```

Library
`ssl_rzk.lib` for use with ZTP 1.6.0 and later.

Prototype
```
SSL_STATUS SSL3_ClientInit( void );
```

Description
The `SSL3_ClientInit()` API initializes the client-side of the SSL version 3 handshake protocol. If this API is not called then it is not possible for the ZTP Network Security SSL Plug-In to establish any SSLv3 sessions in client mode. In client-only mode, applications will only be able to establish SSLv3 sessions with remote SSLv3 servers.

The SSLv3 protocol implemented in the ZTP Network Security SSL Plug-In allows multiple client and server sessions to operate at the same time, but this causes an extra processing burden on the system.

Argument(s)
None.

Return Value(s)

`SSL_SUCCESS`	Indicates that the API is executed successfully.
`SSL_FAILURE`	Indicates that the API did not complete successfully.

Usage Scenario

This API must be called only after calling the Initialize_SSL API (see page 40) and before establishing any SSL sessions with remote SSLv3 servers. Do not call this API multiple times.

Example

The following example initializes the client side of the SSLv3 handshake protocol.

```
Initialize_SSL();
SSL3_ClientInit();
```

See Also

Initialize_SSL	SSL3_ServerInit
TLS1_ClientInit	TLS1_ServerInit

SSL3_ServerInit

Include

```
#include “SSL.h”
```

Library

`ssl_rzk.lib` for use with ZTP 1.6.0 and later.

Prototype

```
SSL_STATUS SSL3_ServerInit
(
  struct CERT_CHAIN *pCertChain,
  ASN1_ENC_DATA     *pDheParams
);
```

Description

The `SSL3_ServerInit()` API initializes the server-side of the SSL version 3 handshake protocol. If this API is not called then it is not possible for the ZTP Network Security SSL Plug-In to establish any SSLv3 sessions in server mode. In server mode, applications will only be able to establish SSLv3 sessions initiated by remote clients.

The SSLv3 protocol implemented in the ZTP Network Security SSL Plug-In allows multiple client and server sessions to operate simultaneously, but this causes extra processing burden on the system.

The `pCertChain` parameter references a certificate chain which contains the server’s X.509 certificate and private key. Any intermediate certificate authority certifies and terminates in a self-signed root certificate. A certificate in the chain is followed by the issuer’s certificate. A self-signed certificate is not followed by any other certificates as it is issued for and issued by the same entity. Currently, the maximum number of certificates allowed in the certificate chain is four. The server’s certificate is shared by all SSLv3 servers in the system.

As certificates in the certificate chain are processed, the SSL library calls the `VerifyCertificate` callback for any certificate which cannot be implicitly trusted (or if the `SSL_PresentAllCertificates` configuration variable is set to TRUE). If the `VerifyCertificate` callback routine returns `SSL_SUCCESS` for all certificates, the SSL library accepts the certificate chain and mark the first entry in the chain (the server's certificate) as a permanent certificate.

The `pDheParams` pointer references the Diffie-Hellman parameters which are used for all Ephemeral Diffie-Hellman (DHE) Cipher suites. If this parameter is set to `NULLPTR` then SSLv3 servers will not be able to establish sessions using any of the DHE cipher suites.

Note: If the `pDheParams` pointer is set to `NULLPTR`, SSLv3 clients will be able to establish sessions using Ephemeral Diffie-Hellman parameters, provided that the global `pDheInit` function pointer is not set to `NULLPTR`.

The Diffie-Hellman parameters must be an ASN.1 DER-encoded sequence of two integers:

1. The prime modulus (p)
2. The generator (g)

These parameters may optionally be Base64-encoded.

Argument(s)

`pCertChain`	Contains the SSLv3 server's certificate and private key followed by the certificate(s) of any intermediate certificate authorities and terminating in the self-signed root certificate.
`pDheParams`	SSLv3 servers reference an ASN.1 DER- encoded (and optionally Base64-encoded) sequence of 2 integers (p and g). If Ephemeral Diffie-Hellman cipher suites are not required, set this parameter to NULLPTR.

Return Value(s)

`SSL_SUCCESS`	Indicates that the API is executed successfully.
`SSL_FAILURE`	Indicates that the API is not completed successfully. Possible reasons for the failure are invalid certificate, or more than four certificates in the certificate chain.

Usage Scenario

This API must be called only after calling the Initialize SSL API (see page 40) and before starting any SSLv3 servers (including the supplied HTTPS server). Do not call this API multiple times.

This routine requires significant stack space. Zilog recommends that this API has to be called only from a task that has a 1 KB run-time stack.

Example

The following example initializes the server-side of the SSLv3 handshake protocol.

```
extern CERT_CHAIN SSL3_CertChain;
extern ASN1_ENC_DATA SSL3_DhParams;
SSL3_ServerInit(&SSL3_CertChain, &SSL3_DhParams);
```

See Also

Initialize_SSL
SSL3_ClientInit
TLS1_ServerInit
VerifyCertificate

TLS1_ClientInit

Include

```
#include “SSL.h”
```

Library

`ssl_rzk.lib` for use with ZTP 1.6.0 and later.

Prototype

```
SSL_STATUS TLS1_ClientInit( void );
```

Description

The `TLS1_ClientInit()` API initializes the client-side of the TLSv1 handshake protocol. If this API is not called then it is not possible for the ZTP Network Security SSL Plug-In to establish any TLSv1 sessions in client mode. In client-only mode, applications will establish only TLSv1 sessions with remote TLSv1 servers.

The TLSv1 protocol implemented in the ZTP Network Security SSL Plug-In allows multiple client and server sessions to operate simultaneously, but this causes an extra processing burden on the system.

Argument(s)

None.

Return Value(s)

`SSL_SUCCESS`	Indicates that the API is executed successfully.
`SSL_FAILURE`	Indicates that the API did not complete successfully.

Usage Scenario

This API must only be called after calling the Initialize_SSL API (see page 40) and before attempting to establish any SSL sessions with remote TLSv1 servers. Do not call this API multiple times.

Example

The following example initializes the client-side of the TLSv1 handshake protocol.

```
Initialize_SSL();
TLSv1_ClientInit();
```

See Also

Initialize_SSL	TLS1_ServerInit
SSL3_ClientInit	SSL3_ServerInit

TLS1_ServerInit

Include

```
#include “SSL.h”
```

Library

`ssl_rzk.lib` for use with ZTP 1.6.0 and later.

Prototype

```
SSL_STATUS TLS1_ServerInit
(
  struct CERT_CHAIN *pCertChain,
  ASN1_ENC_DATA *pDheParams
);
```

Description

The `TLS1_ServerInitAPI()` initializes the server-side of the TLS version 1 handshake protocol. If this API is not called then it is not possible for the ZTP Network Security SSL Plug-In to establish any TLSv1 sessions in either client or server mode. In server mode, applications will only be able to establish TLSv1 sessions initiated by remote clients.

The TLSv1 protocol implemented in the ZTP Network Security SSL Plug-In allows multiple client and server sessions to operate at the same time, but this will place extra processing burden on the system.

The `pCertChain` parameter references a certificate chain which contains the server’s X.509 certificate and private key. Any intermediate certificate authority certifies and terminates in a self-signed root certificate. A certificate in the chain is followed by the issuer’s certificate. A self-signed certificate is not followed by any other certificates since it was issued for and issued by the same entity. Currently, the maximum number of certificates allowed in the certificate chain is four. The server certificate will be shared by all TLSv1 servers in the system.

As certificates in the certificate chain are processed, the SSL library will call the `VerifyCertificate` callback for any certificate that cannot be implicitly trusted (or if the `SSL_PresentAllCertificates` configuration variable is set to TRUE). If the `VerifyCertificate` callback returns `SSL_SUCCESS` for all certificates, the SSL library will accept the certificate chain and mark the first entry in the chain (the servers certificate) as a permanent certificate.

The `pDheParams` pointer references the Diffie-Hellman parameters that are used for all DHE Cipher suites. If this parameter is set to `NULLPTR` then TLSv1 servers will not establish sessions using any of the DHE Cipher suites.

Note: Even if the `pDheParams` pointer is set to `NULLPTR`, TLSv1 clients will still be able to establish sessions using Ephemeral Diffie-Hellman parameters provided that the global `pDheInit` function pointer is not `NULLPTR`.

The Diffie-Hellman parameters must be an ASN.1 DER-encoded sequence of two integers: the prime modulus (p) and the generator (g). The parameters are optionally Base64-encoded.

Argument(s)

`pCertChain`	Contains the TLSv1 server's certificate and private key followed by the certificate(s) of any intermediate certificate authorities and terminating in the self-signed root certificate.
`pDheParams`	TLSv1 servers, references an ASN.1 DER- encoded (and optionally Base64-encoded) sequence of two integers (p and g). If Ephemeral Diffie-Hellman cipher suites are not required, set this parameter to NULLPTR.

Return Value(s)

SSL_SUCCESS	Indicates that the API is executed successfully.
SSL_FAILURE	Indicates that the API is not completed successfully. Possible reasons for the failure are invalid certificate, or more than four certificates in the certificate chain.

Usage Scenario

This API must be called only after calling the Initialize_SSL API (see page 40) and before starting any TLSv1 servers (including the supplied HTTPS server). Do not call this API multiple times.

This routine requires significant stack space. Zilog recommends that this API has to be called only from a task which has a 1 KB run-time stack.

Example

The following example initializes the client-side of the TLSv1 handshake protocol.

```
extern CERT_CHAIN TLS1_CertChain;
extern ASN1_ENC_DATA TLS1_DhParams;
TLS1_ServerInit( &TLS1_CertChain, &TLS1_DhParams );
```

See Also

Initialize_SSL
TLS1_ClientInit
VerifyCertificate

VerifyCertificate

Include

```
#include “ez80_x509.h”
```

Library

None.

Prototype

```
SSL_STATUS VerifyCertificate
(
  SSL_X509_S *pX509
);
```

Description

When the SSL library processes certificate chains, it performs basic integrity checks on the certificate and updates the `Flags` field within the `SSL_X509_S` structure as appropriate. See Table 8 on page 13 for a listing of valid flags. If the `SSL_X509_TRUSTED` flag is not set or if the `SSL_PresentAllCertificates` configuration variable is set to TRUE (see `ssl_conf.c`), then the SSL library calls this API with a reference to the certificate being processed.

The SSL library will implicitly trust certificates. When processing an untrusted certificate (i.e., one that does not have the `SSL_X509_TRUSTED` flag set), the SSL layer will set the `SSL_x509_TRUSTED` flag if it is also determines that the following flags are set:

- `SSL_X509_PARSED_OK`
- `SSL_X509_DATE_VALID`
- `SSL_X509_SIGNATURE_VERIFIED`

If any of these flags are not set, or if the `SSL_PresentAllCertificates` configuration variable is set to

TRUE, then the `VerifyCertificate` callback routine is called. Once a local permanent server certificate (one that has the `SSL_X509_PERMANENT` flag set) is verified, the callback routine will not be called for that certificate.

Application programs are allowed to modify the source code which implements this routines (see `Certificate.c`). This is done to perform additional validation of the certificate or to contact a certificate revocation server to determine if the certificate is retired. However, the certificate referenced by `pCertificate` must not be modified. The only exception is that the `SSL_X509_PERMANENT` flag can be set (but not cleared) by the `VerifyCertificate` callback to indicate that the certificate is not to be deleted after it has been verified. In this instance, the application calls the `free_x509_certificate` API to release resources associated with the certificate when it is no longer required. Applications must be aware that the SSL library will allocate a new `SSL_X509_S` structure each time a remote server sends its certificate chain. Therefore, even if the application already accepted the remote server's certificate from a previous invocation of the `VerifyCertificate` callback, the SSL library will continue to call the `VerifyCertificate` callback routine each time a new `SSL_X509_S` structure is created that references the same x.509 certificate data from the same server. This will happen even if the application sets the `SSL_X509_PERMANENT` flag the last time the `VerifyCertificate` callback routine was called.

Note: A local SSL server's certificate that is the first certificate in the chain passed to any of SSL2_ServerInit (see page 44), SSL3_ServerInit (see page 49), or TLS1_ServerInit (see page 55) routines must never be released by an application. Otherwise, the SSL server owning that certificate will not operate properly, and could crash the system. Therefore, an application must not release a certificate with a call to `free_x509_certificate` unless the application sets the certificate's `SSL_X509_PERMANENT` flag in the `VerifyCertificate` callback.

If this routine returns SSL_SUCCESS, the SSL library sets the SSL_X509_TRUSTED flag. In this instance, the SSL library continues to use the certificate to establish an SSL session. If this routine returns SSL_FAILURE, the SSL library deletes the certificate (unless the SSL_X509_PERMANENT flag is set). In this instance, if the certificate is used to establish an SSL connection, the handshake protocol fails and an SSL session is not established.

Argument(s)

pCertificate	References an SSL_X509_S structure containing information regarding the certificate to be verified. This structure must not be modified by the VerifyCertificate callback routine.

Return Value(s)

SSL_SUCCESS	Instructs the SSL library to accept the certificate and continue to establish the SSL session.
SSL_FAILURE	Instructs the SSL library to destroy the certificate and does not allow an SSL session to be established that uses this certificate.

Example

```
SSL_STATUS VerifyCertificate(SSL_X509_S *pCertificate
)
{
  // Accept the certificate
  return(SSL_SUCCESS);
}
```

See Also

free_x509_certificate	SSL2_ServerInit
SSL3_ServerInit	TLS1_ServerInit

free_x509_certificate

Include

```
#include “SSL.h”
#include “ez80_x509.h”
```

Library

`ssl_rzk.lib` for use with ZTP 1.6.0 and later.

Prototype

```
void free_x509_certificate (
  SSL_X509_S *pCertificate );
```

Description

The `free_x509_certificate()` API releases resources associated with an `SSL_X509_S` structure. The only certificates that an application can release are those for which the application explicitly set the `SSL_X509_PERMANENT` flag in the `VerifyCertificate` callback routine. Once an application calls this API, it must not attempt to access any fields within the `SSL_X509_S` structure referenced by `pCertificate`.

Argument(s)

`pCertificate`	References the `SSL_X509_S` structure to be released.

Return Value(s)

None.

Example

```
extern SSL_X509_S *pCert;
free_x509_certificate(pCert);
```

See Also

VerifyCertificate

SSL Configuration

This section describes the global variables used to configure SSL handshake protocols. Table 22 provides a brief usage description for each of these configuration variables.

Table 22. SSL Configuration Variables

Configuration Variable	Usage
SSL_MAX_SESSION_CACHE_ENTRIES	Maximum size of SSL session cache.
SSL_CACHE_TIMEOUT	Maximum lifetime of an inactive SSL session cache entry measured in 10ms ticks.
SSL_Debug_level	Controls the amount of diagnostic information printed to the console.
SSL_VerifySignatures	Controls whether the SSL library will verify signatures on certificates and temporary session parameters.
SSL_PresentAllCertificates	Determines if the `VerifyCertificate` callback function is called for all certificates or only for suspect certificates.
NumSSL2_CipherSuites	Specifies the number of cipher suites configured for the SSLv2 protocol.
pSSL2_CipherSuites	References an array of `NumSSL2_CipherSuites` `SSL_CS_INFO` structures.
NumSSL3_CipherSuites	Specifies the number of cipher suites configured for the SSLv3 protocol.
pSSL3_CipherSuites	References an array of `NumSSL3_CipherSuites` `SSL_CS_INFO` structures.
NumTLS1_CipherSuites	Specifies the number of cipher suites configured for the TLSv1 protocol.
pTLS1_CipherSuites	References an array of `NumTLS1_CipherSuites` `SSL_CS_INFO` structures.

Table 22. SSL Configuration Variables (Continued)

Configuration Variable	Usage
HashGen	Determines the set of available digest algorithms.
CipherGen	Determines the set of available symmetric cipher algorithms.
PKIGen	Determines the set of available public key algorithms.
pDheInit	Determines if Ephemeral Diffie-Hellman cipher suites can be supported.
DheParams	Contains a server's Ephemeral Diffie-Hellman parameters (p and g).
CertChain	Contains a server's certificate chain.

SSL_MAX_Session_Cache_Entries

Default Configuration

```
SSL_BYTE SSL_MAX_SESSION_CACHE_ENTRIES = 8;
```

Source File

`ssl_conf.c`

Description

Establishing an SSL session on an 8-bit processor is a time consuming process because of the complexity of the asymmetric key exchange algorithm used to arrive at a shared secret between the client and server. Therefore, after the SSL layer establishes a session it stores information regarding the session in a temporary data store called an SSL session cache. The next time the same client and server attempt to establish a session, information can be extracted from the cache, thereby avoiding the need to use the asymmetric key exchange algorithm. This can dramatically decrease the time required to establish subsequent SSL sessions.

The `SSL_MAX_SESSION_CACHE_ENTRIES` configuration variable determines the maximum size of the session cache. If this variable is set to 0, the session cache will be disabled and all SSL sessions will have to execute an asymmetric key exchange algorithm.

During the SSL handshake, the client checks if the local session cache contains an entry for the target server socket. If it does it will attempt to reuse the session identifier. Similarly the server will check for an entry in the session cache from the client with a matching session ID. If the session has not expired, then the server allows the session to be established using information contained within the cache entry and avoiding the asymmetric computation. See the SSL_Cache_Timeout API on page 68.

If the SSL session cache is full and a new entry has to be added for the session which was just established, the SSL layer will destroy the oldest

entry in the cache and reuse that entry to store information about the new session.

In the current release, each entry in the session cache requires 110 bytes of storage.

See Also

SSL_Cache_Timeout

SSL_Cache_Timeout

Default Configuration

```
SSL_DWORD SSL_CACHE_TIMEOUT = 30000; /* measured in
10ms ticks */
```

Source File

`ssl_conf.c`

Description

When an entry is added to the SSL session cache, it is time-stamped with the current system time. Each time the SSL layer searches and locates this entry, its time-stamp is reset and the entry is moved to the beginning of the cache. This ensures that the most recently accessed session cache entry is always located at the beginning of the cache.

When searching for a session cache entry, the SSL library examines the time-stamp of all entries in the cache and delete any entries which are older than `SSL_CACHE_TIMEOUT`.

If the end of the session cache is reached and the appropriate entry cannot be located and the cache is not yet full, a new entry will be created and added to the beginning of the cache.

If end of the cache is reached and the appropriate entry is not located and the cache is full, the SSL library will destroy the entry at the end of the cache and reuse the entry to store information about the new SSL session. The reused entry is then moved to the beginning of the cache.

The default value of the `SSL_CACHE_TIMEOUT` ensures that a cache entry will remain valid for five minutes from the last time that it was accessed.

See Also

SSL_MAX_Session_Cache_Entries

SSL_Debug_level

Default Configuration

```
SSL_BYTE SSL_Debug_level = SSL_DEBUG_ERROR;
```

Source File

`ssl_conf.c`

Description

The SSL library includes a significant number of debug messages which can (optionally) be displayed on the console during operation. The amount of information displayed is controlled by the value of `SSL_Debug_level` configuration variable as shown in Table 23.

Table 23. SSL_Debug_level Settings

SSL_Debug_level Setting	Description
`SSL_DEBUG_NONE`	All debug messages are suppressed.
`SSL_DEBUG_ERROR`	Only Error messages are displayed.
`SSL_DEBUG_WARNING`	Only Error and Warning messages are displayed.
`SSL_DEBUG_INFO`	Display all debug messages.

SSL_VerifySignatures

Default Configuration

```
SSL_BOOL SSL_VerifySignatures = TRUE;
```

Source File

`ssl_conf.c`

Description

Each time the SSL library encounters a certificate chain, one of the integrity checks performed is to verify the certificate's signature. This occurs during local SSL server initialization and upon receipt of a remote server's certificate chain. See SSL2_ServerInit, SSL3_ServerInit, or TLS1_ServerInit for more details. Similarly when a remote SSL server sends a Key Exchange message containing PKI algorithm parameters, the SSL handshake protocol verifies the signature on the parameters.

Each signature verification involves the use of an asymmetric operation using the server's public key. These operations are computationally expensive requiring significant CPU processing time. This can add a noticeable delay in establishing SSL sessions.

To speed up SSL session establishment, the verification of these signatures are disabled by setting the `SSL_VerifySignatures` configuration variable to FALSE. This will instruct the SSL layer to assume that all signatures are valid but potentially reduces system security.

Note: If the `SSL_VerifySignatures` variable is set to FALSE, then it will not be possible for the routine that processes X.509 certificates to set the `SSL_X509_SIGNATURE_VERIFIED` flag. As a result, for each X.509 certificate encountered, the `VerifyCertificate` callback function is called.

See Also

VerifyCertificate

SSL_PresentAllCertificates

Default Configuration

```
SSL_BOOL SSL_PresentAllCertificates = FALSE;
```

Source File

`ssl_conf.c`

Description

The ZTP Network Security SSL Plug-In will only establish sessions with trusted certificates. In this implementation, a certificate becomes trusted when the `SSL_X509_TRUSTED` flag is set in the certificate's `SSL_X509_S` data structure.

When the SSL layer processes certificates within a certificate chain, it will attempt to parse the certificate, validate the starting and ending dates on the certificate and verify the certificate's signature. If all these basic integrity checks are passed, the SSL layer will set the `SL_X509_PARSED_OK`, `SSL_X509_DATE_VALID` and `SSL_X509_SIGNATURE_VERIFIED` status flags. Additionally, if the certificate was not issued to and by the same entity (i.e., the certificate was not self-signed), then the `SSL_X509_SELF_SIGNED` flag will not be set. Under these conditions, the SSL layer will set the certificate's `SSL_X509_TRUSTED` flag.

Normally, the SSL layer only calls the `VerifyCertificate` callback function if a certificate's `SSL_X509_TRUSTED` flag is not set. However, if the `SSL_PresentAllCertificates` configuration variable is set to TRUE, the SSL layer will always call the `VerifyCertificate` callback, even for trusted certificates.

If the `VerifyCertificate` callback returns `SSL_SUCCESS`, then the SSL layer will set the certificate's `SSL_X509_TRUSTED` flag. This will allow the SSL layer to establish a session using the certificate. If the `VerifyCertificate`

callback does not return SSL_SUCCESS, the certificate is rejected and it will not be possible to establish an SSL session with that certificate.

Note: The ZTP Network Security SSL Plug-In does not cache certificates from remote servers. Therefore, if a remote server's certificate becomes trusted, it is used to establish a session but later destroyed. Therefore, the next time an attempt is made to start a session with the same remote server, the server's certificate will once again be not trusted.

See Also

VerifyCertificate SSL_X509_S

NumSSL2_CipherSuites

Default Configuration

```
SSL_BYTE NumSSL2_CipherSuites =
sizeof(SSL2_CipherSuites) / sizeof(SSL_CS_INFO);
```

Source File

`ssl_conf.c`

Description

The SSL version 2 specification defines the default set of cipher kinds or cipher suites which may be used by the SSLv2 protocol. In the ZTP Network Security SSL Plug-In, this information is contained in an array of `SSL_CS_INFO` structures. Each entry in the array contains information regarding one of the cipher suites that can be supported. The `NumSSL2_CipherSuites` configuration variable is used to identify the number of entries in this array.

If the SSLv2 protocol is not used, the `NumSSL2_CipherSuites` configuration variable should be set to 0.

See Also

SSL_CS_Info pSSL3_CipherSuites

pSSL2_CipherSuites

Default Configuration

```
SSL_CS_INFO *pSSL2_CipherSuites   = SSL2_CipherSuites;
```

Source File

`ssl_conf.c`

Description

The SSL version 2 specification defines the default set of cipher kind (i.e., cipher suites) which is used by the `SSLv2` protocol. In the ZTP Network Security SSL Plug-In, this information is contained in an array of `SSL_CS_INFO` structures. Each entry in the array contains information regarding one of the cipher suites that can be supported. The `pSSL2_CipherSuites` configuration variable is a pointer to the first `SSL_CS_INFO` structure in the array.

If the SSLv2 protocol is not used, the `pSSL2_CipherSuites` configuration variable should be set to `NULLPTR`.

See Also

SSL_CS_Info NumSSL2_CipherSuites

NumSSL3_CipherSuites

Default Configuration

```
SSL_BYTE NumSSL3_CipherSuites =
sizeof(SSL3_CipherSuites)/sizeof(SSL_CS_INFO);
```

Source File

`ssl_conf.c`

Description

The SSL version 3 specification defines the default set of cipher suites that will be used by the SSLv3 protocol. In the ZTP Network Security SSL Plug-In, this information is contained in an array of `SSL_CS_INFO` structures. Each entry in the array contains information regarding one of the cipher suites that can be supported. The `NumSSL3_CipherSuites` configuration variable identifies the number of entries in this array.

If the SSLv3 protocol is not being used, the `NumSSL3_CipherSuites` configuration variable must be set to 0.

See Also

SSL_CS_Info pSSL3_CipherSuites

pSSL3_CipherSuites

Default Configuration

```
SSL_CS_INFO *pSSL3_CipherSuites   = SSL3_CipherSuites;
```

Source File

`ssl_conf.c`

Description

The SSL version 3 specification defines the default set of cipher suites which is used by the SSLv3 protocol. In the ZTP Network Security SSL Plug-In, this information is contained in an array of `SSL_CS_INFO` structures. Each entry in the array contains information regarding one of the cipher suites which can be supported. The `pSSL3_CipherSuites` configuration variable is a pointer to the first `SSL_CS_INFO` structure in the array.

If the SSLv3 protocol is not being used, the `pSSL3_CipherSuites` configuration variable must be set to `NULLPTR`.

In the default system configuration, the SSLv3 and TLSv1 protocols share the same array of `SSL_CS_INFO` structures; i.e., they are configured to support the same set of cipher suites, but this is not mandatory.

See Also

SSL_CS_Info NumSSL3_CipherSuites

NumTLS1_CipherSuites

Default Configuration

```
SSL_BYTE NumTLS1_CipherSuites =
sizeof(TLS1_CipherSuites)/sizeof(SSL_CS_INFO);
```

Source File

`ssl_conf.c`

Description

The TLS version 1 specification defines the default set of cipher suites that may be used by the TLSv1 protocol. In the ZTP Network Security SSL Plug-In, this information is contained in an array of `SSL_CS_INFO` structures. Each entry in the array contains information regarding one of the cipher suites that can be supported. The `NumTLS1_CipherSuites` configuration variable identifies the number of entries in this array.

If the TLSv1 protocol is not being used, the `NumTLSv1_CipherSuites` configuration variable should be set to 0.

See Also

SSL_CS_Info pTLS1_CipherSuites

pTLS1_CipherSuites

Default Configuration

```
SSL_CS_INFO *pTLS1_CipherSuites = TLS1_CipherSuites;
```

Source File

`ssl_conf.c`

Description

The TLS version 1 specification defines the default set of cipher suites that may be used by the TLSv1 protocol. In the ZTP Network Security SSL Plug-In, this information is contained in an array of `SSL_CS_INFO` structures. Each entry in the array contains information regarding one of the cipher suites that can be supported. The `pTLS1_CipherSuites` configuration variable is a pointer to the first `SSL_CS_INFO` structure in the array.

If the TLSv1 protocol is not used, the `pTLS1_CipherSuites` configuration variable must be set to `NULLPTR`.

In the default system configuration, the SSLv3 and TLSv1 protocols share the same array of `SSL_CS_INFO` structures; i.e., they are configured to support the same set of cipher suites, but this is not mandatory.

See Also

SSL_CS_Info NumSSL3_CipherSuites

HashGen

Default Configuration

```
HASH_NEWHashGen[ SSL_MAX_HASH ] =
{
  NullHash_New,
  MD5_New,
  HMAC_MD5_New,
  SHA1_New,
  HMAC_SHA1_New
};
```

Source File

`hash_conf.c`

Description

The `HashGen` array contains `SSL_MAX_HASH` (currently defined as 5) `HASH_NEW` function pointers. There is one function pointer for each of the digest (or hash) algorithms supported by the ZTP Network Security SSL Plug-In. To create an `SSL_HASH` structure the SSL library indexes the HashGen array using one of the Digest Algorithm Identifiers (see page 36). Then the SSL library uses the returned `SSL_HASH` pointer to perform digest operations. The first entry in the table must be the `NullHash_New` function pointer. The `NULL hash` does not perform any useful digest function. It removes unnecessary digest routines from the final project. For example, when only `SSLv2` operation is required, it is not necessary to include the `SHA1`, `HMAC_MD5`, or `HMAC_SHA1` digest algorithms in the system. Therefore, these entries in the `HashGen` array can be replaced with the `NullHash_New` function pointer. See Table 11 on page 24 for a list of digest routines that must be included for a particular version of the SSL handshake protocol.

See Also

SSL_Hash
HASH_New
Digest Routines by SSL Protocol Version
Digest Algorithm Identifiers

CipherGen

Default Configuration

```
CIPHER_NEW          CipherGen[ SSL_MAX_CIPHERS ] =
{
  NullCipher_New,
  RC4_New,
  DES_New,
  DES3_New,
  AES_New
};
```

Source File

`cipher_conf.c`

Description

The `CipherGen` array contains `SSL_MAX_CIPHERS` (currently defined as 5) `CIPHER_NEW` function pointers. There is one function pointer for each of the symmetric cipher algorithms supported by the ZTP Network Security SSL Plug-In. To create an `SSL_CIPHER` structure the SSL library indexes the `CipherGen` array using one of the Cipher Algorithm Identifiers (see page 37). Then the SSL library uses the returned `SSL_CIPHER` pointer to perform encryption/decryption operations.

The first entry in the table must be the `NullCipher_New` function pointer. The `NULL` cipher does not perform any useful encryption operation. It removes unnecessary cipher algorithms from the final project. For example, if none of the `SSL_CS_INFO` entries in the SSLv2, SSLv3, or TLSv1 cipher suite tables specify `SSL_CIPHER_AES` as the symmetric cipher algorithm, then the `AES_New` entry in the `CipherGen` array can be replaced with the `NullCipher_New` function pointer. This prevents AES code from being linked into the project.

See Also

SSL_Cipher
CIPHER_New
Cipher Algorithm Identifiers
SSL_CS_INFO Structure Members

PKIGen

Default Configuration

```
PKI_Init PkiGen[SSL_MAX_PKI] =
{
  NullPki_init,
  rsa_init,
  dsa_init,
  dh_init
};
```

Source File

`pki_conf.c`

Description

The PKIGen array contains `SSL_MAX_PKI` (currently defined as 4) `PKI_Init` function pointers. There is one function pointer for each of the PKI algorithms supported by the ZTP Network Security SSL Plug-In. To initialize the `SSL_PKI` structure within an `SSL_X509_S` structure the SSL library indexes the PKIGen array using one of the PKI Algorithm Identifiers in PKIGen Array (see page 37). Then the SSL library uses the function pointers in the `SSL_PKI` structure to perform asymmetric key exchange operations.

The first entry in the table must be the `NullPki_init` function pointer. The `NullPki_init` routine does not perform any useful operation. It removes unnecessary cipher algorithms from the final project. For example, if only SSLv2 operation is required then only RSA support is required and the `dsa_init` routine can be replaced with `NullPki_init`. This prevents the DSA code from being linked into the project.

See Also

SSL_PKI

PKI_Init

PKI Algorithm Identifiers in SSL_CS_INFO Structure

PKI Algorithm Identifiers in PKIGen Array

pDheInit

Default Configuration

```
PKI_DheInitpDheInit = dhe_init;
```

Source File

```
pki_conf.c
```

Description

The `pDheInit` function pointer determines if Ephemeral Diffie-Hellman (DHE) cipher suites will be supported by the `SSLv3` and `TLSv1` handshake protocols. These cipher suites are prefixed with either `TLS_DHE_DSS` or `TLS_DHE_RSA` (the equivalent SSLv3 cipher suites are prefixed with `SSL_DHE_DSS` or `SSL_DHE_RSA`). By default, this function pointer references the `dhe_init` routine which indicates that Ephemeral Diffie-Hellman Cipher Suites will not be supported.

Note: For `SSLv3` servers and `TLSv1` servers that contain `SSL_CS_INFO` structures in which the `KeyAlg` structure member is set to either `SSL_PKI_DHE_RSA` or `SSL_PKI_DHE_DSS`. The `pDheInit` function pointer must refer the `dhe_init` routine or else the Ephemeral Diffie-Hellman cipher suite will not be supported.

If support for DHE cipher suites is not required, then the `pDheInit` function pointer must be set to `NULLPTR`.

See Also

PKI_DheInit
PKI_Init
PKI Algorithm Identifiers in SSL_CS_INFO Structure
PKI Algorithm Identifiers in PKIGen Array

DheParams

Default Configuration

```
SSL_BYTE DH_Params_Pem[] = {"\
MIGKAkEA3uxiDPwIuoU6r22inWehs84FBTvrD8bQufdCltw6RAoV+D
M5PHkyMLoH\
KEThy65yDANqA0s4tukYX+jEg98IFQJBAKK+9mbWv9G6WqQExbjrjx
KUJG863bYR\
QlwmO9kd6hs6rQDa1g1E5UQ9SOrUcs6cLGzuSQYE+0K8G7UEknvAKT
YCAgCg"};

ASN1_ENC_DATA DheParams =
{
  PEM_ENCODED_DATA,
  sizeof(DH_Params_Pem)-1,
  DH_Params_Pem
};
```

Source File

`dh_params.c`

Description

`TLSv1` or `SSLv3` servers which are configured to use Ephemeral Diffie-Hellman cipher suites require that a variable of type `ASN1_ENC_DATA` be created that contains the server's Ephemeral Diffie-Hellman parameters (generator g and prime modulus p). By default the SSLv3 and TLSv1 servers use the same parameters, but it is possible to use different DHE parameters for each server.

If client-only operation is required for either the SSLv3 or TLSv1 handshake protocols, then it is not necessary to create a variable of type `ASN1_ENC_DATA` containing DHE parameters. When operating in client mode, the remote server's DHE parameters is used to establish the session.

See Also

PKI_DheInit
DH_Params
ASN1_Enc_Data
TLS1_ServerInit

CertChain

Default Configuration

```
/*
 * RSA self-signed Certificate (512-bit)
 */
SSL_BYTE cert_data[] = {"\
MIIB5jCCAZCgAwIBAgIQytdHl3HrbqhC6ugpKKHldjANBgkqhkiG9w0B
AQQFADA7\
MTkwNwYDVQQDHjAAZQBaADgAMABBAGMAYwBsAGEAaQBtACEAKABUAE0A
KQAgAFIA\
bwBvAHQAIABDAEEwHhcNMDUwOTA4MDIwNDI1WhcNMDYwOTA4MDIwNDI0
WjA7MTkw\
NwYDVQQDHjAAZQBaADgAMABBAGMAYwBsAGEAaQBtACEAKABUAE0AKQAg
AFIAbwBv\
AHQAIABDAEEwXDANBgkqhkiG9w0BAQEFAANLADBIAkEA5YWRoD1Upa8g
ZY2pQ+9F\
6SanLwNmXab2GEH1KossK3V+flrMCBTEoAB+eIeA+vKmuBeX6tDbcAOD
s3llgz0P\
4QIDAQABo3AwbjBsBgNVHQEEZTBjgBAWVXuaCu63gahlvbuVpWEzoT0w
OzE5MDcG\
A1UEAx4wAGUAWgA4ADAAQQBjAGMAbABhAGkAbQAhACgAVABNACkAIABS
AG8AbwB0\
ACAAQwBBghDK10eXcetuqELq6CkooeV2MA0GCSqGSIb3DQEBBAUAA0EA
WZtSSD6A\
wUWylmF9e3z2egpyWBdSMGoyUC+thoMCKCnii19qS6HiyVunSrseY7WF
X9CMDVcv\
g3CKGZg3rwWAsg==\"};

SSL_BYTE key_data[] = {"\
MIIBPQIBAAJBAOWFkaA9VKWvIGWNqUPvRekmpy8DZl2m9hhB9SqLLCt1
fn5azAgU\
xKAAfniHgPryprgXl+rQ23ADg7N5ZYM9D+ECBAABAAECQQC1UWBqwyik
vwWL1G58\
gYCsIGIAjOIIaAaPwUNpuYpKRUcqvThgSS+oPhnpap6D1GGwvsulVJUS
OWjd6MVt\
jwLxAiEA/XmOYqdd5yTIft/A6SGpVU2C/
PQJ1fX+q7+x51D0VqUCIQDnzuyqQ5t1\
```

```
Cb1sp1hVHQdeVgr7yiul8lwXeEX1/
ntLjQIhALJzMK39vJtthwXji1HWE/vtLQne\
2Unb/
OZ3d80tbkfNAiEAmG7iAjTjDVuSTNjepVmpdsduAZU4jq+JD4Xvu4vU2
CEC\
ICqRZp3qGT2JjzyxyY48XnYTVof7ep0fvMw9yHHfh8Mx"};

CERT_CHAIN CertChain =
{
 1,                                    // 1 certificate
                                       // in this chain
 BASE64_DER_ENCODED_DATA,              // All certs and
                                       // keys are in PEM
                                       // format
 NULLPTR,                              // Created by SSL
                                       // layer when the
                                       // chain is parsed
 {key_data, sizeof(key_data)-1},       // Private Key
 { {cert_data, sizeof(cert_data)-1},   // Root
                                       // Certificate
 {NULLPTR, 0},
 {NULLPTR, 0},
 {NULLPTR, 0} }
};
```

Source File

`Certificate.c`

Description

When one of the SSL servers in the ZTP Network Security SSL Plug-In is initialized, a pointer to a `CERT_CHAIN` data structure containing the server's certificate chain must be passed as a parameter. Servers for each of the SSL handshake protocols have a unique certificate chain, but the default configuration uses the same certificate for all servers.

If client-only operation is required for one of the SSL handshake protocols, it is not necessary to define a server certificate chain.

When the server's certificate chain is defined, the `CERT_CHAIN` variable must contain a private key. When the server's certificate chain is sent to a remote client through the handshake protocol, the private key will not be transmitted.

See Also

Cert_Chain
SSL_X509_S
TLS1_ServerInit

Customer Support

To share comments, get your technical questions answered, or report issues you may be experiencing with our products, please visit Zilog's Technical Support page at http://support.zilog.com.

To learn more about this product, find additional documentation, or to discover other facets about Zilog product offerings, please visit the Zilog Knowledge Base at http://zilog.com/kb or consider participating in the Zilog Forum at http://zilog.com/forum.

This publication is subject to replacement by a later edition. To determine whether a later edition exists, please visit the Zilog website at http://www.zilog.com.