MatrixSSL 3.4 APIs

TABLE OF CONTENTS

1 OVERVIEW

This document is the technical reference for the MatrixSSL and MatrixDTLS C code library APIs. The functions documented here can be used to add server or client SSL/TLS security to any new or existing application on any hardware platform using any data transport mechanism.

This document is primarily intended for the software developer performing MatrixSSL integration into their custom application but is also a useful reference for anybody wishing to learn more about MatrixSSL or the SSL/TLS protocol in general.

For additional information on how to implement these APIs in an application, see the MatrixSSL Developer's Guide included in this package.

1.1 Commercial Version Differences

Some of the compile options, functions, and structures in this document provide additional features only available in the commercially licensed version of MatrixSSL. Sections of this document that refer to the commercial version will be shaded as this paragraph has.

Functionality and features that are available exclusively in the commercial version are not be included in this documentation package if obtained through an open source product. Below is a list of topics available in the full MatrixSSL documentation library.

- Elliptic Curve Cipher Suites
- Diffe-Hellman Cipher Suites
- Pre-Shared Key Cipher Suites
- Matrix Deterministic Memory
- Matrix RSA Key and X.509 Certificate Generation

1.2 Source Code Package

MatrixSSL is distributed as a C source code package with compile environments for the most popular development platforms.

1.2.1 Package Structure

MatrixSSL's public interface function prototypes are defined in the *matrixsslApi.h* file. Applications compiling with MatrixSSL APIs only have to include this single header file.

```
#include "matrixsslApi.h"
```

The *matrixsslApi.h* file includes other package-specific header files using relative paths based on the default directory structure. Optional product features are enabled and disabled by toggling documented #defines. There is no need to restructure the include logic within the header files or to move the header files from the default directory locations when configuring features.

The C data types used by functions in *matrixsslApi.h* come from a variety of module headers in the package directories. MatrixSSL API custom data types with publicly accessible members are documented where applicable.

1.2.2 Integer Size

MatrixSSL was designed without dependency on platform specific integer sizes. MatrixSSL uses the `int32` and `uint32` type definitions throughout the code to ensure compatibility. These typedefs are contained in the *core/osdep.h* header file. This layer enables global redefinitions for platforms that do not support 32-bit integer types as the native `int` type.

1.2.3 Configurable Features

MatrixSSL contains a set of optional features that are configurable at compile time. This allows the user to remove unneeded functionality to reduce code size footprint. Each of these options are pre-processor defines that can be disabled by simply commenting out the `#define` in the header files or by using the `-D` compile flag during build. APIs with dependencies on optional features are highlighted in the **Define Dependencies** sub-section in the documentation for that function.

MATRIX_USE_FILE_SYSTEM	Define in the build environment. Enables file access for parsing X.509 certificates and private keys.
USE_CLIENT_SIDE_SSL	*matrixsslConfig.h* - Enables client side SSL support
USE_SERVER_SIDE_SSL	*matrixsslConfig.h* - Enables server side SSL support
USE_TLS	*matrixsslConfig.h* - Enables TLS 1.0 protocol support (SSL version 3.1)
USE_TLS_1_1	*matrixsslConfig.h* - Enables TLS 1.1 (SSL version 3.2) protocol support. USE_TLS must be enabled
USE_TLS_1_2	*matrixsslConfig.h* - Enables TLS 1.2 (SSL version 3.3) protocol support. USE_TLS_1_1 must be enabled
DISABLE_SSLV3	*matrixsslConfig.h* - Disables SSL version 3.0
DISABLE_TLS_1_0	*matrixsslConfig.h* – Disables TLS 1.0 if USE_TLS is enabled but only later versions of the protocol are desired
DISABLE_TLS_1_1	*matrixsslConfig.h* – Disables TLS 1.1 if USE_TLS_1_1 is enabled but only later versions of the protocol are desired
SSL_SESSION_TABLE_SIZE	*matrixsslConfig.h* – Applicable to servers only. The size of the session resumption table for caching session identifiers. Old entries will be overwritten when size is reached
SSL_SESSION_ENTRY_LIFE	*matrixssConfig.h* – Applicable to servers only. The time in seconds that a session identifier will be valid in the session table. A value of 0 will disable SSL resumption
ENABLE_SECURE_REHANDSHAKES	*matrixsslConfig.h* - Enable secure rehandshaking as defined in RFC 5746
REQUIRE_SECURE_REHANDSHAKES	*matrixsslConfig.h* - Halt communications with any SSL peer that has not implemented RFC 5746
ENABLE_INSECURE_REHANDSHAKES	*matrixsslConfig.h* - Enable legacy renegotiations. NOT RECOMMENDED
REQUESTED_MAX_PLAINTEXT_RECORD_LEN	*matrixsslConfig.h* – Enable the “max_fragment_length” TLS extension defined in RFC 4366. Value of #define determines fragment length (server may reject)
ENABLE_FALSE_START	*matrixsslConfig.h* – See code comments in file
USE_BEAST_WORKAROUND	*matrixsslConfig.h* – See code comments in file.
USE_CLIENT_AUTH	*matrixsslConfig.h* - Enables two-way(mutual) authentication
SERVER_CAN_SEND_EMPTY_CERT_REQUEST	*matrixsslConfig*.h – A client authentication feature. Allows the server to send an empty CertificateRequest message if no CA files have been loaded

SERVER_WILL_ACCEPT_EMPTY_CLIENT_CERT_MSG	*matrixsslConfig*.h – A client authentication feature. Allows the server to 'downgrade' a client authentication handshake to a standard handshake if client does not provide a certificate
USE_PRIVATE_KEY_PARSING	*cryptoConfig*.h - Enables X.509 private key parsing
USE_PKCS5	*cryptoConfig.h* - Enables the parsing of password protected private keys
USE_PKCS8	*cryptoConfig.h* - Enables the parsing of PKCS#8 formatted private keys
USE_PKCS12	*cryptoConfig.h* - Enables the parsing of PKCS#12 formatted certificate and key material
USE_1024_KEY_SPEED_OPTIMIZATIONS	*cryptoConfig.h* - Enables fast math for 1024-bit public key operations
PS_PUBKEY_OPTIMIZE_FOR_SMALLER_RAM PS_PUBKEY_OPTIMIZE_FOR_FASTER_SPEED	*cryptoConfig.h* - RSA and Diffie-Hellman speed vs. runtime memory tradeoff. Default is to optimize for smaller RAM.
PS_AES_IMPROVE_PERF_INCREASE_CODESIZE PS_3DES_IMPROVE_PERF_INCREASE_CODESIZE PS_MD5_IMPROVE_PERF_INCREASE_CODESIZE PS_SHA1_IMPROVE_PERF_INCREASE_CODESIZE	*cryptoConfig.h* - Optionally enable for selected algorithms to improve performance at the cost of increased binary code size.

Table 1 - Compile-time Configuration Options

1.2.4 Debug Configuration

MatrixSSL contains a set of optional debug features that are configurable at compile time. Each of these options are pre-processor defines that can be disabled by simply commenting out the #define in the specified header files.

HALT_ON_PS_ERROR	*coreConfig.h* - Enables the `osdepBreak` platform function whenever a `psError` trace function is called. Helpful in debug environments.
USE_CORE_TRACE	*coreConfig.h* - Enables the `psTraceCore` family of APIs that display function-level messages in the core module
USE_CRYPTO_TRACE	*cryptoConfig.h* - Enables the `psTraceCrypto` family of APIs that display function-level messages in the crypto module
USE_SSL_HANDSHAKE_MSG_TRACE	*matrixsslConfig.h* - Enables SSL handshake level debug trace for troubleshooting connection problems
USE_SSL_INFORMATIONAL_TRACE	*matrixsslConfig.h* - Enables SSL function level debug trace for troubleshooting connection problems

Table 2 - Compile-time Debug Options

1.2.5 Cipher Suites

The user can enable or disable any of the supported cipher suites at compile-time from the *matrixsslConfig.h* header file. Simply comment out the cipher suites that are not needed. If run-time disabling of cipher suites is required, `matrixSslSetCipherSuiteEnabledStatus` can be used to disable (and re-enable) ciphers that have been compiled into the library.

The individual cryptographic algorithms may be enabled and disabled through the *cryptoConfig.h* header file for fine-tuning of library size. Below is a representative list of cipher suites along with their cryptographic requirements. The comprehensive list of which cipher suites are supported in the specific MatrixSSL package can be found in the *matrixsslConfig.h* file.

Sample Cipher Suites in matrixsslConfig.h	**cryptoConfig.h Dependencies**
USE_TLS_RSA_WITH_AES_256_CBC_SHA	USE_RSA USE_AES
USE_SSL_RSA_WITH_3DES_EDE_CBC_SHA	USE_RSA USE_3DES
USE_SSL_RSA_WITH_RC4_128_SHA	USE_RSA USE_ARC4
USE_TLS_DHE_RSA_WITH_AES_256_CBC_SHA	USE_DH USE_RSA USE_AES
USE_TLS_DH_anon_WITH_AES_256_CBC_SHA	USE_DH USE_AES
USE_TLS_DHE_RSA_WITH_AES_256_CBC_SHA256	USE_DH USE_RSA USE_AES USE_SHA256
USE_TLS_RSA_WITH_AES_256_CBC_SHA256	USE_RSA USE_AES USE_SHA256
USE_TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA	USE_ECC USE_AES
USE_TLS_DHE_PSK_WITH_AES_256_CBC_SHA	USE_DH USE_AES
USE_TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA	USE_ECC USE_RSA USE_AES
USE_TLS_PSK_WITH_AES_256_CBC_SHA	USE_AES
USE_TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384	USE_ECC USE_AES_GCM USE_SHA384

2 MATRIXSSL API

2.1 matrixSslOpen

```
int32 matrixSslOpen(void);
```

Return Value	Description
PS_SUCCESS	Successful initialization
PS_FAILURE	Failed core module initialization. Can't continue

Servers and Clients

This is the initialization function for the MatrixSSL library. Applications must call this function as part of their own initialization process before any other MatrixSSL functions are called.

Memory Profile

This function internally allocates memory that is freed during `matrixSslClose`

2.2 matrixSslNewKeys

```
int32 matrixSslNewKeys(sslKeys_t **keys);
```

Parameter	Input/Output	Description
keys	input/output	Internally allocated structure to use when loading key material

Return Value	Description
PS_SUCCESS	Successful key storage initialization
PS_MEM_FAIL	Failure. Unable to allocate memory for the structure

Servers and Clients

This is a necessary function that all implementations must call before loading in the specific key material that will be used in the SSL handshake.

After allocating the key structure, the user will load custom key material from files (or memory) using `matrixSslLoadRsaKeys`, `matrixSslLoadEcKeys`, `matrixSslLoadPkcs12`, `matrixSslLoadDhParams`, and/or `matrixSslLoadPsk`. Loading RSA/ECC keys or DH parameters may be done once for each `keys` context. Multiple calls can be made to load pre-shared keys for a single `keys` context.

Once loaded with the key material, the `keys` structure will be passed to `matrixSslNewClientSession` or `matrixSslNewServerSession` to associate those keys with the SSL session.

Memory Profile

This function internally allocates memory that is freed during `matrixSslDeleteKeys`. The caller does not need to free the keys parameter if this function does not return `PS_SUCCESS`.

2.3 matrixSslLoadRsaKeys

```
int32 matrixSslLoadRsaKeys(sslKeys_t *keys, const char *certFile,
      const char *privFile, const char *privPass, const char *trustedCAFiles);
```

Parameter	Input/Output	Description
keys	input/output	Allocated key structure returned from a previous call to `matrixSslNewKeys`. Will become input to `matrixSslNewClientSession` or `matrixSslNewServerSession` to associate key material with a SSL session.
certFile	input	The fully qualified filename(s) of the PEM formatted X.509 RSA identity certificate for this SSL peer. For in-memory support, see `matrixSslLoadRsaKeysMem` This parameter is always relevant to servers. Clients will want to supply an identity certificate and private key if supporting client authentication. `NULL` otherwise.
privFile	input	The fully qualified filename of the PEM formatted PKCS#1 or PKCS#8 private RSA key that was used to sign the `certFile`. This parameter is always relevant to servers. Clients will want to supply an identity certificate and private key if supporting client authentication. `NULL` otherwise.
privPass	input	The plaintext password used to encrypt the private key file. `NULL` if private key file is not password protected or unused. MatrixSSL supports the MD5 PKCS#5 2.0 PBKDF1 password standard.
trustedCAFiles	input	The fully qualified filename(s) of the trusted root certificates (Certificate Authorities) for this SSL peer. This parameter is always relevant to clients. Servers will want to supply a CA if requesting client authentication. `NULL` otherwise.

Return Value	Test	Description
PS_SUCCESS	0	Success. All input files parsed and the keys parameter is available for use in session creation
PS_CERT_AUTH_FAIL	< 0	Failure. Certificate or chain did not self-authenticate or private key could not authenticate certificate
PS_PLATFORM_FAIL	< 0	Failure. Error locating or opening an input file
PS_ARG_FAIL	< 0	Failure. Bad input function parameter
PS_MEM_FAIL	< 0	Failure. Internal memory allocation failure
PS_PARSE_FAIL	< 0	Failure. Error parsing certificate or private key buffer
PS_FAILURE	< 0	Failure. Password protected decoding failed. Likey incorrect password provided
PS_UNSUPPORTED_FAIL	< 0	Failure. Unsupported key algorithm in certificate material

Servers and Clients

This function is called to load the RSA certificates and private key files from disk that are needed for SSL client-server authentication. The key material is loaded into the `keys` parameter for input into the subsequent session creation APIs `matrixSslNewClientSession` or `matrixSslNewServerSession`. This API can be called at most once for a given `sslKeys_t` parameter.

A standard SSL connection performs one-way authentication (client authenticates server) so the parameters to this function are specific to the client/server role of the application. The `certFile`, `privFile`, and `privPass` parameters are server specific and should identify the certificate and private key file for that server. The `certFile` and `privFile` parameters represent the two halves of the public key so they must both be non-NULL values if either is used.

The `trustedCAFiles` parameter is client specific and should identify the trusted root certificates that will be used to validate the certificates received from a server.

Calling this function is a resource intensive operation because of the file access, parsing, and internal public key authentications required. For this reason, it is advised that this function be called once per set of key files for a given application. All new sessions associated with the certificate material can reuse the existing key pointer. At application shutdown the user must free the key structure using `matrixSslDeleteKeys`.

Client Authentication

If client authentication functionality is desired, all parameters to this function become relevant to both clients and servers. The `certFile` and `privFile` parameters are used to specify the identity certificate of the local peer. Likewise, each entity will need to supply a `trustedCAcertFile` parameter that lists the trusted CAs so that the connecting certificates may be authenticated. It is easiest to think of client authentication as a mirror image of the normal server authentication when considering how certificate and CA files are deployed.

It is possible to configure a server to engage in a client authentication handshake without loading CA files. Enable the `SERVER_CAN_SEND_EMPTY_CERT_REQUEST` define in *matrixsslConfig.h* to allow the server to send an empty CertificateRequest message. The server can then use the certificate callback function to perform a custom authentication on the certificate returned from the client.

The MatrixSSL library must be compiled with `USE_CLIENT_AUTH` defined in *matrixsslConfig.h* to enable client authentication support.

Multiple CA Certificates and Certificate Chaining

It is not uncommon for a server to work from a certificate chain in which a series of certificates form a child-to-parent hierarchy. It is even more common for a client to load multiple trusted CA certificates if numerous servers are being supported.

There are two ways to pass multiple certificates to the `matrixSslLoadRsaKeys` API. The first is to pass a semi-colon delimited list of files to the `certFile` or `trustedCAcertFiles` parameters. The second way is to append several PEM certificates into a single file and pass that file to either of the two parameters. Regardless of which way is chosen, the `certFile` parameter MUST be passed in a child-to-parent order. The first certificate parsed in the chain MUST be the child-most certificate and each subsequent certificate must be the parent (issuer) of the former. There must only ever be one private key file passed to this routine and it must correspond with the child-most certificate.

Encrypted Private Keys

It is strongly recommended that private keys be password protected when stored in files. The `privPass` parameter of this API is the plaintext password that will be used if the private key is encrypted. MatrixSSL supports an MD5 based PKCS#5 2.0 PBKDF1 standard for password encryption. The most common way a password is retrieved is through user input during the initialization of an application.

Memory Profile

The keys parameter must be freed with `matrixSslDeleteKeys` after its useful life.

Define Dependencies

MATRIX_USE_FILE_SYSTEM	Must be enabled in platform compile options
USE_SERVER_SIDE_SSL	Optionally enable in *matrixsslConfig.h* for SSL server support
USE_CLIENT_SIDE_SSL	Optionally enable in *matrixsslConfig.h* for SSL client support
USE_PKCS5	Optionally enable in *cryptoConfig.h* to support password encrypted private keys
USE_PKCS8	Optionally enable in *cryptoConfig.h* to support PKCS#8 formatted private keys
USE_CLIENT_AUTH	Optionally enable in *matrixsslConfig.h* to support client authentication

2.4 matrixSslLoadRsaKeysMem

```
int32 matrixSslLoadRsaKeysMem(sslKeys_t *keys, unsigned char *certBuf,
      int32 certLen, unsigned char *privBuf, int32 privLen,
      unsigned char *trustedCABuf, int32 trustedCALen);
```

Parameter	Input/Output	Description
keys	input/output	Allocated key structure returned from a previous call to `matrixSslNewKeys`. Will become input to `matrixSslNewClientSession` or `matrixSslNewServerSession` to associate key material with a SSL session.
certBuf	input	The X.509 ASN.1 identity certificate for this SSL peer. For file-based support, see `matrixSslLoadRsaKeys` This parameter is always relevant to servers. Clients will want to supply an identity certificate and private key if supporting mutual authentication. `NULL` otherwise.
certLen	input	Byte length of `certBuf`
privBuf	input	The PKCS#1 or PKCS#8 private RSA key that was used to sign the certBuf. This parameter is always relevant to servers. Clients will want to supply an identity certificate and private key if supporting mutual authentication. `NULL` otherwise.
privLen	input	Byte length of `privBuf`
trustedCABuf	input	The X.509 ASN.1 stream of the trusted root certificates (Certificate Authorities) for this SSL peer. This parameter is always relevant to clients. Servers will want to supply a CA if requesting mutual authentication. `NULL` otherwise.
trustedCALen	input	Byte length of `trustedCABuf`

Return Value	Test	Description
PS_SUCCESS	0	Success. All input buffers parsed successfully and the keys parameter is available for use in session creation
PS_CERT_AUTH_FAIL	< 0	Failure. Certificate or chain did not self-authenticate or private key could not authenticate certificate
PS_PLATFORM_FAIL	< 0	Failure. Error locating or opening an input file
PS_ARG_FAIL	< 0	Failure. Bad input function parameter
PS_MEM_FAIL	< 0	Failure. Internal memory allocation failure
PS_PARSE_FAIL	< 0	Failure. Error parsing certificate or private key buffer
PS_UNSUPPORTED_FAIL	< 0	Failure. Unsupported key algorithm in certificate material

Servers and Clients

This function is the in-memory equivalent of the `matrixSslLoadRsaKeys` API to support environments where the certificate material is not stored as files on disk. Please consult the information above about `matrixSslLoadRsaKeys` for detailed information on how clients and servers should manage the certificate and private key parameters. This API can be called at most once for a given `sslKeys_t` parameter.

The buffers for the certificates and private key must be in the native ASN.1 format of the X.509 v3 and PKCS#1/PKCS#8 standards, respectively. Typically, the ".der" file extension is used for certificate material in this binary format.

There is no password protection support for private key buffers. It is recommended that the user implement secure storage for the private key material.

Multiple CA Certificates and Certificate Chaining

This in-memory version of the key parser also supports multiple CAs and/or certificate chains. Simply append the ASN.1 certificate streams together for either the `certBuf` or `trustedCAbuf` parameters. If using a certificate chain in the `certBuf` parameter the order of the certificates still MUST be in child-to-parent order with the `privBuf` being the key associated with the child-most certificate.

Memory Profile

The keys parameter must be freed with `matrixSslDeleteKeys` after its useful life.

Define Dependencies

USE_SERVER_SIDE_SSL	Optionally enable in *matrixsslConfig.h* for SSL server support
USE_CLIENT_SIDE_SSL	Optionally enable in *matrixsslConfig.h* for SSL client support
USE_PKCS8	Optionally enable in *cryptoConfig.h* to support PKCS#8 formatted private keys
USE_CLIENT_AUTH	Optionally enable in *matrixsslConfig.h* to support client authentication

2.5 matrixSslLoadPkcs12

```
int32 matrixSslLoadPkcs12(sslKeys_t *keys, unsigned char *p12File,
      unsigned char *importPass, int32 ipasslen,
      unsigned char *macPass, int32 mpasslen, int32 flags);
```

Parameter	Input/Output	Description
keys	input/output	Allocated key structure returned from a previous call to `matrixSslNewKeys`. Will become input to `matrixSslNewClientSession` or `matrixSslNewServerSession` to associate key material with a SSL session.
p12File	input	The fully qualified filename(s) of the PKCS#12 file.
importPass	input	The plaintext import password used to decrypt `p12File`
ipassLen	input	Byte length of the `importPass` parameter
macPass	input	Optional plaintext password used to verify the MAC of the PKCS#12 file. In most cases, the MAC password is identical to the import password and if set to `NULL` the import password will be used by default.
mpassLen	input	The byte length of the `macPass` parameter
flags	input	Reserved. Pass a 0

Return Value	Test	Description
PS_SUCCESS	0	Success. File parsed and the `keys` parameter is available for use
PS_CERT_AUTH_FAIL	< 0	Failure. Certificate or chain did not self-authenticate or private key could not authenticate certificate
PS_PLATFORM_FAIL	< 0	Failure. Error locating or opening input file
PS_ARG_FAIL	< 0	Failure. Bad input function parameter
PS_MEM_FAIL	< 0	Failure. Internal memory allocation failure
PS_PARSE_FAIL	< 0	Failure. Error parsing certificate or private key buffer
PS_UNSUPPORTED_FAIL	< 0	Failure. Unsupported algorithm in file material

Servers

This function is called to load certificate and key material from a PKCS#12 file. The PKCS#12 standard enables certificates and private keys to be stored together in a single file. This function requires that only a single private key be present in the PKCS#12 file and includes the accompanying certificate (or certificate chain).

The `sslKeys_t` output is loaded into the keys parameter for input into the subsequent session creation API `matrixSslNewServerSession`. This API can be called at most once for a given `sslKeys_t` parameter.

Calling this function is a resource intensive operation because of the file access, parsing, and internal public key authentications required. For this reason, it is advised that this function be called once per set of key files for a given application. All new sessions associated with the certificate material can reuse the

existing key pointer. At application shutdown the user must free the key structure using `matrixSslDeleteKeys`.

Client Authentication

Clients may use this function to load certificates and the private key if engaging in a client authentication handshake.

However, for both server and client cases the counterpart Certificate Authority files must be loaded separately using the `matrixSslLoadRsaKeys` function because this PKCS#12 API does not support CA files. In this case, the same `sslKeys_t` parameter should be used in both APIs.

The MatrixSSL library must be compiled with `USE_CLIENT_AUTH` defined in *matrixsslConfig.h* to enable client authentication support.

Certificate Chaining

It is not uncommon for a server to work from a certificate chain in which a series of certificates form a child-to-parent hierarchy. The PKCS#12 file must have the certificate chain in a child-to-parent order and the private key must be for the child-most certificate.

Supported Integrity and Encryption Algorithms

The parser supports PKCS#12 files that are encoded in the standard “password integrity” and “password privacy” modes. If you require public-key modes please contact Inside Secure.

Each certificate and private key will be wrapped within a “password privacy” algorithm. The supported algorithms are:

- pbeWithSHAAnd3-KeyTripleDES-CBC
- pbewithSHAAnd40BitRC2-CBC

The use of these algorithms is historical and certificates are generally encrypted with RC2 and private keys are generally encrypted with 3DES. Please contact INSIDE if you require additional “password privacy” algorithms.

Memory Profile

The keys parameter must be freed with `matrixSslDeleteKeys` after its useful life.

Define Dependencies

USE_SERVER_SIDE_SSL	Optionally enable in *matrixsslConfig.h* for SSL server support
USE_CLIENT_SIDE_SSL	Optionally enable in *matrixsslConfig.h* for SSL client support
USE_PKCS12	Must enable in *cryptoConfig.h* to support PKCS#12
USE_CLIENT_AUTH	Optionally enable in *matrixsslConfig.h* to support client authentication
MATRIX_USE_FILE_SYSTEM	Must define in platform build environment for file access
USE_RC2	Optionally enable in *cryptoConfig.h* if RC2 encryption is needed

2.6 matrixSslNewSessionId

```
int32 matrixSslNewSessionId(sslSessionId_t **sid);
```

Parameter	Input/Output	Description
sid	input/output	Storage for an SSL session ID used for future session resumption

Return Value	Test	Description
PS_SUCCESS	0	Success. Session ID storage ready to be passed to `matrixSslNewClientSession`
PS_MEM_FAIL	< 0	Failure. Internal memory allocation failed

Clients

This function is only meaningful to a client wishing to perform future SSL session resumptions with a particular server. After allocating a session ID with this call, the structure is passed to the `sid` parameter of `matrixSslNewClientSession` where it will be populated with valid resumption credentials during the handshake process. Subsequent calls to `matrixSslNewClientSession` to reconnect with the same server should pass this same session ID to initiate the much faster session resumption handshake.

See the **Session Resumption** chapter in the MatrixSSL Developer's Guide document accompanying this release for more information.

Memory Profile

The `sid` parameter must be freed with `matrixSslDeleteSessionId` after its useful life.

Define Dependencies

USE_CLIENT_SIDE_SSL	Must be defined in *matrixsslConfig.h*

2.7 matrixSslClearSessionId

```
void matrixSslClearSessionId(sslSessionId_t *sid);
```

Parameter	Input/Output	Description
sid	input/output	Previously allocated SSL session ID to be cleared

Clients

This function is only meaningful to clients using the SSL session resumption feature. This function will empty the session ID contents of the `sid` parameter that were previously stored during an earlier handshake. The `sid` parameter will have been allocated by a previous call to `matrixSslNewSessionId`. This function is simply for convenience if wishing to initiate a new session with a full handshake without having to call `matrixSslDeleteSessionId` and `matrixSslNewSessionId`.

Define Dependencies

USE_CLIENT_SIDE_SSL	Must be defined in *matrixsslConfig.h*

2.8 matrixSslDeleteSessionId

```
void matrixSslDeleteSessionId(sslSessionId_t *sid);
```

Parameter	Input/Output	Description
sid	input	Previously allocated SSL session ID to be cleared and freed

Clients

This function is only meaningful to clients using the SSL session resumption feature. This function will free the session ID that was previously allocated by `matrixSslNewSessionId`.

Define Dependencies

USE_CLIENT_SIDE_SSL	Must be defined in *matrixsslConfig.h*

2.9 matrixSslNewClientSession

```
int32 matrixSslNewClientSession(ssl_t **ssl, sslKeys_t *keys,
      sslSessionId_t *sessionId, uint32 cipherSuite,
      int32 (*certValidator)(ssl_t *, psX509Cert_t *, int32),
      tlsExtension_t *extensions, int32 (*extensionCback)(ssl_t *ssl,
      unsigned short type, unsigned short len, void *data), int32 flags);
```

Parameter	Input/Output	Description
ssl	input/output	New context for this SSL session
keys	input	Key pointer that has been populated with the necessary certificate and key material (see `matrixSslNewKeys`)
sessionId	input/output	SSL session id storage previously allocated by `matrixSslNewSessionId`
cipherSuite	input	Pass a value of 0 to allow the client and server to negotiate the cipher suite automatically OR pass the integer identifier of the specific cipher suite the client wants to use. See the full cipher suite list in the source code file *matrixssllib.h* for possible values.
certValidator	input	The function that will be invoked during the SSL handshake to see the internal authentication status of the server certificate chain. This callback is also the opportunity for the application to perform custom validation tests as needed
extensions	input	Custom CLIENT_HELLO extensions. See `matrixSslNewHelloExtension` for details.
extensionCback	input	The function that will be invoked as a callback during the SSL handshake to see any SERVER_HELLO extensions that have been received
flags	input	`SSL_FLAGS_DTLS` for MatrixDTLS product. 0 otherwise.

Return Value	Test	Description
MATRIXSSL_REQUEST_SEND	> 0	Success. The `ssl_t` context is initialized and the CLIENT_HELLO message has been encoded and is ready to be sent to the server to being the SSL handshake
PS_ARG_FAIL	< 0	Failure. Bad input function parameter
PS_MEM_FAIL	< 0	Failure. Memory allocation failure
PS_PROTOCOL_FAIL	< 0	Failure. SSL context is not in the correct state for creating a CLIENT_HELLO message or there was an error encrypting the message
PS_UNSUPPORTED_FAIL	< 0	Failure. The requested cipher suite was not found or library was not compiled with client support
PS_PLATFORM_FAIL	< 0	Failure. Internal call to `psGetEntropy` failed while encoding CLIENT_HELLO message

Clients

Clients call this function to start a new SSL session or to resume a previous one. The session context is returned in the output parameter `ssl`. The CLIENT_HELLO handshake message is internally generated when this function is called and the typical action to take after this function returns is to retrieve that message with `matrixSslGetOutdata` and send that data to the server.

This function requires a pointer to an `sslKeys_t` structure that was returned from a previous call to `matrixSslNewKeys` and loaded with the relevant certificate and key material using `matrixSslLoadRsaKey` or equivalent.

If the client wishes to resume a session with a server the `sessionId` parameter can be used. For the initial handshake with a new server this parameter should point to a `matrixSslNewSessionId` allocated `sslSessionId_t` location in which the library will store the session ID information during the handshake process. For this reason, it is essential that the `sessionId` location be scoped for the lifetime of the SSL session it is passed into. On subsequent handshakes with the same server, the client can simply pass through this same `sessionId` memory location and `matrixSslNewClientSession` will extract the session ID and encode a CLIENT_HELLO message that will initiate a resumed handshake with the server. The `sessionId` parameter may be `NULL` if session resumption is not desired.

If the user wants to ensure the `sessionId` parameter is initialized or cleared of any previous session ID information, `matrixSslClearSessionId` should be used to guarantee a full handshake.

The `cipherSuite` parameter can be used to force the client to send a single cipher to the server rather than the entire set of supported ciphers. Set this value to 0 to send the entire cipher suite list that is enabled in *matrixsslConfig.h*. Otherwise the value is the decimal integer value of the cipher suite specified in the standards. The supported values can be found in *matrixssllib.h*.

An explicit cipher suite will take precedence over the cipher suite in `sessionId` if they do not match. So if both `sessionId` and `cipherSuite` are passed in and the `cipherSuite` does not match the cipher that is contained in the `sessionId` parameter, the `sessionId` will be cleared and the client will encode a new CLIENT_HELLO with the `cipherSuite` value. If the `cipherSuite` value is 0 or if it identically matches the cipher suite in the `sessionId` parameter, session resumption will be attempted.

The `certValidator` parameter is used to register a callback routine that will be invoked during the certificate validation portion of the SSL handshake. This optional (but highly recommended) registration will enable the application to see the internal authentication results of the server certificate, perform custom validation checks, and pass certificate information on to end users wishing to manually validate certificates. Additional tests a callback may want to perform on the certificate information might include date validation and host name (common name) verification. If a certificate callback is not registered the internal public-key authentication against the nominated Certificate Authorities will determine whether or not to continue the handshake.

Detailed information on the certificate callback routine is found in the section **The Certificate Validation Callback Function** towards the end of this document.

The `extensions` parameter enables the user to pass custom CLIENT_HELLO extensions to the server. See `matrixSslNewHelloExtension` for more information.

The `extensionCback` parameter enables the user to register a function callback that will be invoked during the parsing of SERVER_HELLO if the server has provided extensions. The callback should return < 0 if the handshake should be terminated.

Memory Profile

The user must free the `ssl_t` structure using `matrixSslDeleteSession` after the useful life of the session. The caller does not need to free the `ssl` parameter if this function does not return `MATRIXSSL_REQUEST_SEND`.

The `keys` pointer is referenced in the `ssl_t` context without duplication so it is essential the user does not call `matrixSslDeleteKeys` until all associated sessions have been deleted.

Define Dependencies

USE_CLIENT_SIDE_SSL	Must be enabled in *matrixsslConfig.h*
ENABLE_SECURE_REHANDSHAKES	Optionally disable support for RFC 5746

2.10 matrixSslNewServerSession

```
int32 matrixSslNewServerSession(ssl_t **ssl, sslKeys_t *keys,
      int32 (*certCB)(ssl_t *, psX509Cert_t *, int32), int32 flags);
```

Parameter	Input/Output	Description
ssl	input/output	New context for this SSL session
keys	input	Key pointer that has been populated with the necessary certificate and key material (see `matrixSslNewKeys`)
certCb	input	Only relevant if using client authentication. `NULL` if not using client authentication, otherwise the function that will be invoked during the SSL handshake to see the internal authentication status of the client certificate chain. This callback is also the opportunity for the application to perform custom validation tests as needed.
flags	input	`SSL_FLAGS_DTLS` for MatrixDTLS product. 0 otherwise.

Return Value	Test	Description
PS_SUCCESS	0	Success. The `ssl_t` context is initialized and ready for use
PS_ARG_FAIL	< 0	Failure. Bad input function parameter
PS_FAILURE	< 0	Failure. Internal memory allocation failure

Servers

When a server application has received notice that a client is requesting a secure socket connection (a socket accept on a secure port), this function should be called to initialize the new SSL session context. This function will prepare the server for the SSL handshake and the typical action to take after returning from this function is to call `matrixSslGetReadbuf` to retrieve an allocated buffer in which to copy the incoming handshake message from the client.

This function requires a pointer to an `sslKeys_t` structure that was returned from a previous call to `matrixSslNewKeys` and populated with key material from `matrixSslLoadRsaKeys` (or equivalent)

In client authentication scenarios the `certValidator` parameter must be used to register a callback on the server side to perform application specific checks on the client certificate. Setting a certificate callback is an explicit indication that client authentication will be used for this session.

If a server wants to be able to optionally enable client authentication but not require it for the initial handshake the certificate callback should be included in `matrixSslNewServerSession` but then `matrixSslSetSessionOption` with the `SSL_OPTION_DISABLE_CLIENT_AUTH` should be called immediately after. When the server later determines client authentication should be used, it can call `matrixSslSetSessionOption` with `SSL_OPTION_ENABLE_CLIENT_AUTH`.

Detailed information on the callback routine can be found below in the section entitled **The Certificate Validation Callback Function**.

Memory Profile

The user must free the `ssl_t` structure using `matrixSslDeleteSession` after the useful life of the session. The caller does not need to free the `ssl` parameter if this function does not return `PS_SUCCESS`.

The `keys` pointer is referenced in the `ssl_t` context without duplication so it is essential the user does not call `matrixSslDeleteKeys` until all associated sessions have been deleted.

Define Dependencies

USE_SERVER_SIDE_SSL	Must be enabled in *matrixsslConfig.h*

2.11 matrixSslGetReadbuf

```
int32 matrixSslGetReadbuf(ssl_t *ssl, unsigned char **buf);
```

Parameter	Input/Output	Description
ssl	input	The SSL session context
buf	output	Pointer to the memory location where incoming peer data should be read into

Return Value	Description
>= 0	Success. Indicates how many bytes are available in buf for incoming data
PS_ARG_FAIL	Failure. Bad function parameters

Servers and Clients

Any time the application is expecting to receive data from a peer this function must be called to retrieve the memory location where the incoming data should be read into. By providing a buffer to read network data into, the MatrixSSL API avoids an internal buffer copy.

The length of available bytes in `buf` is indicated in the return code. This is a maximum length and it is the user's responsibility to adhere to this size and not read data bytes beyond the given length. The mechanism for handling incoming data beyond the returned size is discussed below.

Once the user has read data into this buffer, `matrixSslReceivedData` must be called to process the data in-situ. If the return code from `matrixSslReceivedData` is `MATRIXSSL_REQUEST_RECV` this indicates that additional data needs to be read. In this case, `matrixSslGetReadbuf` must be called again for an updated pointer and buffer size to copy the additional data into.

2.12 matrixSslReceivedData

```
int32 matrixSslReceivedData(ssl_t *ssl, uint32 bytes, unsigned char **ptbuf,
      uint32 *ptLen);
```

Parameter	Input/Output	Description
ssl	input	The SSL session context
bytes	input	The number of bytes received
ptbuf	output	If the data being received is an application-level record (or an alert) the unencrypted plaintext will be delivered to the user through this parameter. This will be a read-only pointer into the buffer that the user can process directly or copy locally for parsing at a later time.
ptLen	output	If `ptbuf` is non-NULL this is the byte length of the data

Return Value	Test	Description
MATRIXSSL_REQUEST_SEND	> 0	Success. The processing of the received data resulted in an SSL response message that needs to be sent to the peer. If this return code is hit the user should call `matrixSslGetOutdata` to retrieve the encoded outgoing data.
MATRIXSSL_REQUEST_RECV	> 0	Success. More data must be received and this function must be called again. User must first call `matrixSslGetReadbuf` again to receive the updated buffer pointer and length to where the remaining data should be read into.

MATRIXSSL_HANDSHAKE_COMPLETE	> 0	Success. The SSL handshake is complete. This return code is returned to client side implementation during a full handshake after parsing the FINISHED message from the server. It is possible for a server to receive this value if a resumed handshake is being performed where the client sends the final FINISHED message.
MATRIXSSL_RECEIVED_ALERT	> 0	Success. The data that was processed was an SSL alert message. In this case, the `ptbuf` pointer will be two bytes (`ptLen` will be 2) in which the first byte will be the alert level and the second byte will be the alert description. After examining the alert, the user must call `matrixSslProcessedData` to indicate the alert was processed and the data may be internally discarded.
MATRIXSSL_APP_DATA	> 0	Success. The data that was processed was application data that the user should process. In this return code case the `ptbuf` and `ptLen` output parameters will be valid. The user may process the data directly from `ptbuf` or copy it aside for later processing. After handling the data the user must call `matrixSslProcessedData` to indicate the plain text data may be internally discarded
PS_SUCCESS	0	Success. This return code will be returned if the bytes parameter is 0 and there is no remaining internal data to process. This could be useful as a polling mechanism to confirm the internal buffer is empty. One real life use-case for this method of invocation is when dealing with a Google Chrome browser that uses False Start.
PS_MEM_FAIL	< 0	Failure. Internal memory allocation error
PS_ARG_FAIL	< 0	Failure. Bad input parameters
PS_PROTOCOL_FAIL	< 0	Failure. Internal protocol error

Servers and Clients

This function must be called each time data is received from the peer. The sequence of events surrounding this function is to call `matrixSslGetReadbuf` to retrieve empty buffer space, read or copy the received data from the peer into that buffer, and then call this function to allow MatrixSSL to decode the peer data. Notice the actual received buffer that is being processed is not passed as an input to this function, since it is internal to the SSL session structure. However, it is important that the `bytes` parameter correctly identifies how many bytes have been received, and thus be processed.

The return value from this function indicates how the user should respond next:

MATRIXSSL_REQUEST_RECV - The user must call `matrixSslGetReadbuf` again, copy additional peer data into the buffer, and call this function again. Typically this indicates that a partial record has been received, and more data must be read to complete the record. Also it can mean that a internal SSL record was processed internally and another record is expected to follow.

MATRIXSSL_REQUEST_SEND - The library has internally generated an SSL handshake response message to be sent to the peer. The user must call `matrixSslGetOutdata`, send the data to the peer, and then call `matrixSslSentData`.

MATRIXSSL_HANDSHAKE_COMPLETE - This is an indication that there are no remaining SSL handshake messages to be sent or received and the first application message can be sent. This is generally an important return code for a client application to handle because in most protocols it is the client that will be sending the initial application data request (such as an HTTPS GET or POST request). In this typical usage scenario, the user will then encrypt application data using the following steps: Call `matrixSslGetWritebuf` to retrieve an allocated buffer for outgoing application data, write the plaintext data to this buffer, call `matrixSslEncodeWritebuf` to encrypt the data, call `matrixSslGetOutdata` to retrieve the encrypted data, send that encrypted data to the peer, and finally call `matrixSslSentData` to notify the library the data has been sent.

> *NOTE*: If this code is returned, there are not any additional full SSL records in the buffer available to parse, although there may be a partial record remaining. If there were a full SSL record available, for example an application data record, it would be parsed and `MATRIXSSL_APP_DATA` would be returned instead.

MATRIXSSL_APP_DATA - This means the received data was an application record and the plain text data is available in the `ptbuf` output parameter for user processing. The length of the plain text application data is indicated by the `ptLen` parameter. The user can either directly parse the read only data out of this buffer at this time or copy it aside to be parsed later. In either case it is essential the user call `matrixSslProcessedData` when finished working with it, so the buffer may be internally re-used and tested for the existence of an additional record. The user MUST parse or copy aside all unparsed data in the buffer, as it will be overwritten after the `matrixSslProcessedData` call.

> *NOTE*: If application data has been appended to a handshake FINISHED message it is possible the `MATRIXSSL_APP_DATA` return code can be received without ever having received the `MATRIXSSL_HANDSHAKE_COMPLETE` return code. In this case, it is implied the handshake completed successfully because application data is being received.

MATRIXSSL_RECEIVED_ALERT - This means an alert has been decoded that the user should examine. The alert material will always be a two-byte plain text message available in the `ptbuf` parameter of the function (`ptLen` will be 2). The first byte will be the alert level. It will either be `SSL_ALERT_LEVEL_WARNING` or `SSL_ALERT_LEVEL_FATAL`. The second byte will be the alert identification as specified in the SSL and TLS RFC documents. It is sometimes possible to continue after receiving a WARNING level alert, but FATAL alerts should always result in the connection being closed. In either case the user should always call `matrixSslProcessedData` to update the library that the plain text data can be discarded.

2.13 matrixSslGetOutdata

```
int32 matrixSslGetOutdata(ssl_t *ssl, unsigned char **buf);
```

Parameter	Input/Output	Description
ssl	input	The SSL session context
buf	output	Pointer to beginning of data buffer that needs to be sent to the peer

Return Value	Description
> 0	The number of bytes in `buf` that need to be sent
0	No pending data to send
PS_ARG_FAIL	Failure. Bad input parameters

Servers and Clients

Any time the application is expecting to send data to a peer this function must be called to retrieve the memory location and length of the encoded SSL buffer. This API can also be polled to determine if there is encoded data pending that should be sent out the network.

The length of available bytes in `buf` is indicated in the return code.

There are several ways data can be encoded in outdata and ready to send:

1. After a client calls `matrixSslNewClientSession` this function must be called to retrieve the encoded CLIENT_HELLO message that will initiate the handshake
2. After a client or server calls `matrixSslEncodeRehandshake` this function must be called to retrieve the encoded SSL message that will initiate the rehandshake
3. If the `matrixSslReceivedData` function returns `MATRIXSSL_REQUEST_SEND` this function must be called to retrieve the encoded SSL handshake reply.
4. After the user calls `matrixSslEncodeWritebuf` this function must be called to retrieve the encrypted buffer for sending.

5. After the user calls `matrixSslEncodeToOutdata` this function must be called to retrieve the encrypted buffer for sending.
6. After the user calls `matrixSslEncodeClosureAlert` to encode the CLOSE_NOTIFY alert this function must be called to retrieve the encoded alert for sending.

After sending the returned bytes to the peer, the user must always follow with a call to `matrixSslSentData` to update the number of bytes that have been sent from the returned `buf`. Depending on how much data was sent, there may still be data to send within the internal outdata, and the function should be called again to ensure 0 bytes remain.

2.14 matrixSslProcessedData

```
int32 matrixSslProcessedData(ssl_t *ssl, unsigned char **ptbuf,
      uint32 *ptlen);
```

Parameter	Input/Output	Description
ssl	input	The SSL session context
ptbuf	output	If another full application record was present in the buffer that was returned from `matrixSslReceivedData`, this will be an updated pointer to this next decrypted record. Thus, this parameter is only meaningful if the return value of this function is `MATRIXSSL_APP_DATA` or `MATRIXSSL_RECEIVED_ALERT`.
ptlen	output	The length of the `ptbuf` parameter

Return Value	Test	Description
PS_SUCCESS	0	Success. This indicates that there are no additional records in the data buffer that require processing. The application protocol is responsible for deciding the next course of action.
MATRIXSSL_APP_DATA	> 0	Success. There is a second application data record in the buffer that has been decoded. In this return code case the `ptbuf` and `ptlen` output parameters will be valid. The user may process the data directly from `ptbuf` or copy it aside for later processing. After handling the data the user must call `matrixSslProcessedData` again to indicate the plain text data may be internally discarded.
MATRIXSSL_REQUEST_SEND	> 0	Success. This return code is possible if the buffer contained an application record followed by a SSL handshake message to initiate a re-handshake (CLIENT_HELLO or HELLO_REQUEST). In this case the SSL re-handshake response has been encoded and is waiting to be sent.
MATRIXSSL_REQUEST_RECV	> 0	Success. This return code is possible if there is a partial second record that follows in the buffer. Data storage must be retrieved via `matrixSslGetReadbuf` and passed through the `matrixSslReceivedData` call again.
MATRIXSSL_RECEIVED_ALERT	> 0	Success. There is a second record in the data buffer that is an SSL alert message. In this case, the `ptbuf` pointer will be two bytes (`ptlen` will be 2) in which the first byte will be the alert level and the second byte will be the alert description. After examining the alert, the user must call `matrixSslProcessedData` again to indicate the alert was processed and the data may be internally discarded.
PS_MEM_FAIL	< 0	Failure. Internal memory allocation failure
PS_ARG_FAIL	< 0	Failure. Bad input parameters
PS_PROTOCOL_FAIL	< 0	Failure. Internal protocol error

Servers and Clients

This essential function is called after the user has finished processing plaintext application data that was returned from `matrixSslReceivedData`. Specifically, this function must be called if the return code from `matrixSslReceivedData` was `MATRIXSSL_APP_DATA` or `MATRIXSSL_RECEIVED_ALERT`.

It is also possible that this function be called multiple times in succession if multiple SSL records have been received in a single `matrixSslReceivedData` call. See the very important section **Multi-Record Buffers** immediately below.

Plaintext application data is returned to the user through `matrixSslReceivedData` on a per-record basis whose length is stored internal to the library as part of the buffer management. This is why there are no input parameters regarding the length of the processed data. This function will destroy the plaintext record that was retrieved through the previous `matrixSslReceivedData` call (or the previous `matrixSslProcessedData` call) so if the user requires the data to persist it must be copied aside before calling this function.

Multi-Record Buffers

The `matrixSslReceivedData` function will only process a single application data record at a time. However, it is possible there will be more than one record in the buffer. In this case the return code from `matrixSslProcessedData` will indicate the status of the next record in the buffer. Any return code other than `PS_SUCCESS` (0) or a failure code (< 0) is an explicit indication that an additional record is present in the buffer and will inform the caller how it should be handled.

The multi-record return codes are a subset of the `matrixSslReceivedData` function and should be handled identically so it should be a straightforward code implementation to examine the return codes from this function in the standard processing loop. The *client.c* and *server.c* sample application files are a good reference for how to handle multi-record buffers.

2.15 matrixSslSentData

```
int32 matrixSslSentData(ssl_t *ssl, uint32 bytes);
```

Parameter	Input/Output	Description
ssl	input	The SSL session context
bytes	input	Length, in bytes, of how much data has been written out to the peer

Return Value	Test	Description
PS_SUCCESS	0	Success. No pending data remaining
MATRIXSSL_REQUEST_SEND	> 0	Success. Call `matrixSslGetOutdata` again and send more data to the peer. Indicates the number of `bytes` sent was not the full amount of pending data.
MATRIXSSL_REQUEST_CLOSE	> 0	Success. This indicates the message that was sent to the peer was an alert and the caller should close the session.
MATRIXSSL_HANDSHAKE_COMPLETE	> 0	Success. Will be returned to the peer if this is the final FINISHED message that is being sent to complete the handshake.
PS_ARG_FAIL	< 0	Failure. Bad input parameters.

Servers and Clients

This function must be called each time data has been sent to the peer. The flow of this function is that the user first calls `matrixSslGetOutdata` to retrieve the outgoing data buffer, the user sends part or all of this data, and then calls `matrixSslSentData` with how many bytes were actually sent.

The return value from this function indicates how the user should respond next:

MATRIXSSL_REQUEST_SEND - There is still pending data that needs to be sent to the peer. The user must call `matrixSslGetOutdata`, send the data to the peer, and then call `matrixSslSentData` again.

MATRIXSSL_SUCCESS - All of the data has been sent and the application will likely move to a state of awaiting incoming data.

MATRIXSSL_REQUEST_CLOSE - All of the data has been sent and the application should close the connection. This will be the case if the data being sent is a closure alert (or fatal alert).

MATRIXSSL_HANDSHAKE_COMPLETE - This is an indication that this peer is sending the final FINISHED message of the SSL handshake. In general this will be an important return code for client applications to handle because most protocols will rely on the client sending an initial request to the server once the SSL handshake is complete. If a client receives this return code, a resumed handshake has just completed.

2.16 matrixSslGetWritebuf

```
int32 matrixSslGetWritebuf(ssl_t *ssl, unsigned char **buf,
      uint32 requestedLen);
```

Parameter	Input/Output	Description
ssl	input	The SSL session context
buf	output	Pointer to allocated storage that the user will copy plaintext application data into
requestedLen	input	The amount of buffer space, in bytes, the caller would like to use

Return Value	Test	Description
> 0		Success. The number of bytes available in `buf`. Might not be the same as `requestedLen`
PS_MEM_FAIL	< 0	Failure. Internal memory allocation error
PS_ARG_FAIL	< 0	Failure. Bad input parameters
PS_FAILURE	< 0	Failure. Internal error managing data buffers

Servers and Clients

This function is used in conjunction with `matrixSslEncodeWritebuf` when the user has application data that needs to be sent to the peer. This function will return an allocated buffer in which the user will copy the plaintext data that needs to be encoded and sent to the peer.

The event sequence for sending plaintext application data is as follows:

1. The user first determines the length of the plaintext that needs to be sent
2. The user calls `matrixSslGetWritebuf` with that length to retrieve an allocated buffer.
3. The user writes the plaintext into the buffer and then calls `matrixSslEncodeWritebuf` to encrypt the plaintext
4. The user calls `matrixSslGetOutdata` to retrieve the encoded data and length to be sent
5. The user sends the out data buffer contents to the peer
6. The user calls `matrixSslSentData` with the number of bytes that were sent

The internal buffer will grow to accommodate the `requestedLen` bytes and this function may be called multiple times (in conjunction with `matrixSslEncodeWritebuf`) before sending the data out via `matrixSslGetOutdata`. However, if the requested length is larger than the maximum allowed SSL plaintext length the return code will be smaller than the `requestedLen` value. In this fragmentation case, the caller must adhere to the returned length and only copy in as much plaintext as allowed. These two functions can then be called again immediately to retrieve a new buffer to encode the remainder of the

plaintext data. It is also possible to receive a value that is smaller than `requestedLen` if using this function in MatrixDTLS when the encoded size will exceed the maximum datagram size (PMTU).

This function is most appropriate when sending a file or application data that is generated on-the-fly into the returned buffer. If the user wishes to encode an existing plaintext buffer the function, `matrixSslEncodeToOutdata` may be used as an alternative to this function to avoid having to copy the plaintext data into the returned buffer.

This function is specific to application level data. This function is not necessary during the SSL handshake portion of the connection because the MatrixSSL library internally generates all SSL handshake records.

2.17 matrixSslEncodeWritebuf

```
int32 matrixSslEncodeWritebuf(ssl_t *ssl, uint32 len);
```

Parameter	Input/Output	Description
ssl	input	The SSL session context
len	input	Length of plaintext data

Return Value	Test	Description
> 0		Success. The number of bytes in the encoded buffer to send to the peer. Will be a larger value than the input `len` parameter.
PS_ARG_FAIL	< 0	Failure. Bad input parameters
PS_PROTOCOL_FAIL	< 0	Failure. This session is flagged for closure at the time of this call
PS_FAILURE	< 0	Failure. Internal error managing buffers

Servers and Clients

This function is used in conjunction with `matrixSslGetWritebuf` when the user has application data that needs to be sent to the peer. This function will encrypt the plaintext data that has been copied into the buffer that was previously returned from a call to `matrixSslGetWritebuf`.

The event sequence for sending plaintext application data is as follows:

1. The user first determines the length of the plaintext that needs to be sent
2. The user calls `matrixSslGetWritebuf` with that length to retrieve an allocated buffer.
3. The user writes the plaintext into the buffer and then calls `matrixSslEncodeWritebuf` to encrypt the plaintext
4. The user calls `matrixSslGetOutdata` to retrieve the encoded data to be sent
5. The user sends the out data buffer contents to the peer
6. The user calls `matrixSslSentData` with the number of bytes that were sent

If the user wishes to encode an existing plaintext buffer the function `matrixSslEncodeToOutdata` may be used as an alternative to this function. This function is specific to application level data. This function is not necessary during the SSL handshake portion of the connection because the MatrixSSL library internally generates all SSL handshake records.

2.18 matrixSslEncodeToOutdata

```
int32 matrixSslEncodeToOutdata(ssl_t *ssl, unsigned char *ptBuf, uint32 len);
```

Parameter	Input/Output	Description
ssl	input	The SSL session context
ptBuf	input	Pointer to plaintext application data that will be encrypted into the internal outdata buffer for sending to the peer
len	input	Length, in bytes, of `ptBuf`

Return Value	Test	Description
> 0		Success. The number of bytes in the encoded buffer to send to the peer. Will be a larger value than the input `len` parameter.
PS_LIMIT_FAIL	< 0	Failure. The plaintext length must be smaller than the SSL specified value of 16KB. In MatrixDTLS this return code indicates the encoded size will exceed the maximum datagram size.
PS_MEM_FAIL	< 0	Failure. The internal allocation of the destination buffer failed.
PS_ARG_FAIL	< 0	Failure. Bad input parameters
PS_PROTOCOL_FAIL	< 0	Failure. This session is flagged for closure.
PS_FAILURE	< 0	Failure. Internal error managing buffers.

Servers and Clients

This function offers an alternative method to `matrixSslEncodeWritebuf` when the user has application data that needs to be sent to the peer. This function will encrypt the plaintext data to the internal output buffer while leaving the plaintext data untouched. This function does not require that `matrixSslGetWritebuf` be called first.

This function is specific to application level data. This function is not necessary during the SSL handshake portion of the connection because the MatrixSSL library internally generates any SSL handshake records.

The event sequence for sending plaintext application data is as follows:

1. The user calls `matrixSslEncodeToOutdata` with the plaintext buffer location and length.
2. The user calls `matrixSslGetOutdata` to retrieve the encoded data to be sent
3. The user sends the out data buffer contents to the peer
4. The user calls `matrixSslSentData` with the number of bytes that were sent

2.19 matrixSslEncodeClosureAlert

```
int32 matrixSslEncodeClosureAlert(ssl_t *ssl);
```

Parameter	Input/Output	Description
ssl	input	The SSL session context

Return Value	Test	Description
PS_SUCCESS	0	Success. The alert is ready to be retrieved and sent.
PS_PROTOCOL_FAIL	< 0	Failure. SSL context not in correct state to create the alert or there was an error encrypting the alert message.
PS_ARG_FAIL	< 0	Failure. Bad input parameter
PS_MEM_FAIL	< 0	Failure. Internal memory allocation error

Servers and Clients

The SSL specification highlights an optional alert message that SHOULD be sent prior to closing the communication channel with a peer. This function generates this CLOSE_NOTIFY alert that the peer may send to the other side to notify that the connection is about to be closed. Many implementations simply close the connection without an alert, but per spec, this message should be sent first. Our recommendation is to make an attempt to send the closure alert as a non-blocking message and ignore the return value of the attempt. This way, best efforts are made to send the alert before closing, but application code does not block or fail on a connection that is about to be closed.

After calling this function the user must call `matrixSslGetOutdata` to retrieve the buffer for the encoded alert to send.

2.20 matrixSslGetAnonStatus

```
void matrixSslGetAnonStatus(ssl_t *ssl, int32 *anon);
```

Parameter	Input/Output	Description
ssl	input	The SSL session context
anon	output	1 – Anonymous 0 - Authenticated

Clients

This function returns whether or not the server session is anonymous in the `anon` output parameter. A value of 1 indicates the peer is anonymous and a value of 0 indicates the connection has been fully authenticated. An anonymous connection in this case means the application explicitly allowed the SSL handshake to continue despite not being able to authenticate the certificate supplied by the other side with an available Certificate Authority. The mechanism to allow an anonymous connection is for the certificate validation callback function to return `SSL_ALLOW_ANON_CONNECTION`. Detailed information on the callback routine can be found below in the section entitled **The Certificate Validation Callback Function**.

`matrixSslGetAnonStatus` is only meaningful to call after the successful completion of the SSL handshake. Anonymous connections are not normally recommended but can be useful in a scenario in which encryption is the only security concern. Other reasons the caller may choose to use anonymous connections might be to allow a subset of the normal functionality to anonymous connectors or to temporarily accept a connection while a certificate upgrade is being performed.

Servers

Calling this routine from the server side is meaningless for an implementation that has not performed client authentication. In other words, it is not possible for one side of the connection to know if the peer believes the connection to be anonymous or not. This is an easy rule to remember if you recall the mechanism to allow anonymous connections is controlled through the certificate validation callback routine when the `SSL_ALLOW_ANON_CONNECTION` define is returned.

2.21 matrixSslEncodeRehandshake

```
int32 matrixSslEncodeRehandshake(ssl_t *ssl, sslKeys_t *keys,
      int32 (*certCb)(ssl_t *, psX509Cert_t *, int32),
      uint32 sessionOption, uint32 cipherSpec);
```

Parameter	Input/Output	Description
ssl	input	The SSL session context
keys	input	Populated key structure if changing key material for this re-handshake. `NULL` if not changing key material
certCb	input	Certificate callback function for the re-handshake if a change is being made to it. `NULL` to keep existing callback
sessionOption	input	`SSL_OPTION_FULL_HANDSHAKE` or 0
cipherSpec	input	Client specific. Cipher suite for the re-handshake. Only meaningful if the `sessionOption` parameter is set to `SSL_OPTION_FULL_HANDSHAKE`

Return Value	Test	Description
PS_SUCCESS	0	Success. Handshake message is encoded and ready for retrieval.
PS_UNSUPPORTED_FAIL	< 0	Failure. Client specific. Cipher spec could not be found.
PS_PROTOCOL_FAIL	< 0	Failure. SSL context not in correct state for a re-handshake or buffer management error.
PS_ARG_FAIL	< 0	Failure. Bad input parameter
PS_MEM_FAIL	< 0	Failure. Internal memory allocation error
PS_PLATFORM_FAIL	< 0	Failure. Client specific. Error in `psGetEntropy` when encoding CLIENT_HELLO

Clients and Server

Clients or servers call this function on an already secure connection to initiate a re-handshake. A re-handshake is an encrypted SSL handshake performed over an existing connection in order to derive new symmetric key material and/or to change the public keys or cipher suite of the secured communications.

A re-handshake can either be a full handshake or a resumed handshake and the determination is made by the input parameters to this function.

A resumed re-handshake will be used if the `keys`, `certCb`, `sessionOption`, and `cipherSpec` parameters are all set to 0 (or `NULL` for pointers). This is an indication that there is no underlying algorithm or handshake type change that is being made to the connection and the intention is simply to re-key the encrypted communications.

If the `keys`, `certCb`, or `cipherSpec` parameters are set, this is an indication that an “upgraded” connection is desired and a full handshake will be performed with the new parameters.

A full re-handshake can always be guaranteed if `SSL_OPTION_FULL_HANDSHAKE` is passed as the `sessionOption` parameter to this function.

After calling this function the user must call `matrixSslGetOutdata` to retrieve the buffer for the encoded HELLO message to send.

Servers

This function is called on the server side to build a HELLO_REQUEST message to be passed to a client to initiate a re-handshake. This is the only mechanism in the SSL protocol that allows the server to initiate a handshake.

As with `matrixSslNewServerSession` the nomination of a `certCb` is in explicit indication that a client authentication handshake should be performed.

Note that the SSL specification allows clients to ignore a HELLO_REQUEST message. The MatrixSSL client does not ignore this message and will send a CLIENT_HELLO message with the current session ID to initiate a resumed handshake.

Clients

If a client invokes this function a new CLIENT_HELLO handshake message will be internally generated.

For more information about re-handshaking and related security issues, see the Re-handshake section of the MatrixSSL Developers Guide.

2.22 matrixSslSetCipherSuiteEnabledStatus

```
int32 matrixSslSetCipherSuiteEnabledStatus(ssl_t *ssl,
      uint16 cipherId, uint32 status);
```

Parameter	Input/Output	Description
ssl	input	The SSL session context or `NULL` for a global setting
cipherId	input	A single SSL/TLS specification cipher suite ID. Values may be found in *matrixssslIib.h*
status	input	`PS_FALSE` to disabled the cipher suite or `PS_TRUE` to re-enable a previously disabled cipher suite.

Return Value	Test	Description
PS_SUCCESS	0	Success. Cipher suite has been successfully enabled or disabled
PS_FAILURE	< 0	Failure. The cipher suite specified in `cipherId` was not found
PS_LIMIT_FAIL	< 0	Failure. No additional room to store disabled cipher. Increase the `SSL_MAX_DISABLED_CIPHERS` define.
PS_ARG_FAIL	< 0	Failure. Bad input parameter
PS_ARG_FAIL	< 0	Failure. Bad input parameter

Servers

This function may be called on the server side to programmatically disable (`PS_FALSE`) and re-enable (`PS_TRUE`) cipher suites that have been compiled into the library. By default, all cipher suites compiled into the library (as defined in *matrixsslConfig.h*) will be enabled and available for clients to connect with.

The disabling of a cipher suite may be done at a global level or a per-session level. If the `ssl` parameter to this routine is `NULL`, the setting will be global. If the server wishes to disable ciphers on a per-session basis this function must be called immediately after `matrixSslNewServerSession` using the new `ssl_t` structure that was returned from that session creation function. If a cipher suite has been globally disabled the per-session setting will be ignored.

The maximum number of cipher suites that may be disabled on a per-session basis is determined by the value of `SSL_MAX_DISABLED_CIPHERS`. The default is 8. There is no limit to the number of cipher suites that may be globally disabled.

2.23 matrixSslDeleteSession

```
void matrixSslDeleteSession(ssl_t *ssl);
```

Parameter	Input/Output	Description
ssl	input	The SSL session context

Servers and Clients

This function is called at the conclusion of an SSL session that was created using `matrixSslNewServerSession` or `matrixSslNewClientSession`. This function will free the internally

allocated state and buffers associated with the session. It should be called after the corresponding socket or network transport has been closed.

2.24 matrixSslDeleteKeys

```
void matrixSslDeleteKeys(sslKeys_t *keys);
```

Parameter	Input/Output	Description
keys	input	A pointer to an `sslKeys_t` value returned from a previous call to matrixSslNewKeys

Servers and Clients

This function is called to free the key structure and elements allocated from a previous call to `matrixSslNewKeys`. Any key material that was loaded into the key structure using `matrixSslLoadRsaKeys`, `matrixSslLoadEcKeys`, `matrixSslLoadDhParams`, or `matrixSslLoadPsk` will also be freed.

2.25 matrixSslClose

```
void matrixSslClose(void);
```

Servers and Clients

This function performs the one-time final cleanup for the MatrixSSL library. Applications should call this function as part of their own de-initialization.

2.26 matrixSslNewHelloExtension

```
int32 matrixSslNewHelloExtension(tlsExtension_t **extension);
```

Parameter	Input/Output	Description
extension	output	Newly allocated tlsExtension_t structure to be used as input to matrixSslLoadHelloExtension

Return Value	Test	Description
PS_SUCCESS	0	Success. The `extension` parameter is ready for use
PS_MEM_FAIL	< 0	Failure. Internal memory allocation failure

Clients

Enables basic support for the client side hello extension mechanism, as defined in RFC 3546. This mechanism does not support the CERTIFICATE_URL and CERTIFICATE_STATUS handshake message additions.

This function allocates a new `tlsExtension_t` that `matrixSslLoadHelloExtension` will use to populate with extension data. This populated extension parameter will eventually be passed to `matrixSslNewClientSession` in the extensions input parameter so that CLIENT_HELLO will be encoded with the desired hello extensions.

If the client is expecting the server to reply with extension data in the SERVER_HELLO message, that data may be accessed in the certificate callback routine in the `helloExtIn` member of the `ssl_t` data structure.

Memory Profile

The user must free `tlsExtension_t` with `matrixSslDeleteHelloExtension` after the useful life. The extension data is internally copied into the CLIENT_HELLO message during the call to `matrixSslNewClientSession` so `matrixSslDeleteHelloExtension` may be called immediately after returning from this function if the user does not require further use.

Define Dependencies

USE_CLIENT_SIDE_SSL	Must be enabled in matrixsslConfig.h

2.27 matrixSslLoadHelloExtension

```
int32 matrixSslLoadHelloExtension(tlsExtension_t *extension,
      unsigned char *extData, uint32 extLen, uint32 extType);
```

Parameter	Input/Output	Description
extension	input	Previously allocated `tlsExtension_t` structure from a call to `matrixSslNewExtension`
extData	input	A single, fully encoded hello extension to be included in the CLIENT_HELLO message. Formats for extensions can be found in RFC 3546
extLen	input	Length, in bytes, of `extData`
extType	input	The standardized extension type.

Return Value	Test	Description
PS_SUCCESS	0	Success. The data has been added to the extension
PS_MEM_FAIL	< 0	Failure. Memory allocation failure
PS_ARG_FAIL	< 0	Failure. Bad input parameters

Clients

Enables basic support for the client side hello extension mechanism, as defined in RFC 3546.

Extension data to the `extData` must be formatted per specification. For example, the ServerNameList extension must be encoded in the format per RFC 3546:

```
struct {
      NameType name_type;
      select (name_type) { case host_name: HostName; } name;
} ServerName;

enum { host_name(0), (255) } NameType;

opaque HostName<1..2^16-1>;

struct { ServerName server_name_list<1..2^16-1> } ServerNameList;
```

The `extType` parameter will also be a value as specified by a standards body. The extensions defined in RFC 3546, for example:

```
enum {
      server_name(0), max_fragment_length(1),
      client_certificate_url(2), trusted_ca_keys(3),
      truncated_hmac(4), status_request(5), (65535)
} ExtensionType;
```

It is possible to call this function multiple times for each extension that needs to be added. On success, this populated extension parameter will be passed to `matrixSslNewClientSession` in the extensions input parameter so that CLIENT_HELLO will be encoded with the desired hello extensions.

Note the current level of support in MatrixSSL does not include the additional handshake messages of CERTIFICATE_URL and CERTIFICATE_STATUS that accompany some of these extension types. For information on how to fully support these features, please contact Inside Secure.

Memory Profile

The user must free `tlsExtension_t` with `matrixSslDeleteHelloExtension` after the useful life. The extension data is internally copied into the CLIENT_HELLO message during the call to `matrixSslNewClientSession` so `matrixSslDeleteHelloExtension` may be called immediately after returning from this function if the user does not require further use.

Define Dependencies

USE_CLIENT_SIDE_SSL	Must be enabled in matrixsslConfig.h

2.28 matrixSslDeleteHelloExtension

```
void matrixSslDeleteHelloExtension(tlsExtension_t *extension);
```

Parameter	Input/Output	Description
extension	input	A pointer to an `tlsExtension_t` value returned from a previous call to `matrixSslNewHelloExtension`

Clients

This function is called to free the structure allocated from a previous call to `matrixSslNewHelloExtension`. Any extension material that was loaded into the key structure using `matrixSslLoadHelloExtension` will also be freed.

It is possible to call this function immediately after `matrixSslNewClientSession` returns because the extension data will have been internally copied into the CLIENT_HELLO message.

Define Dependencies

USE_CLIENT_SIDE_SSL	Must be enabled in matrixsslConfig.h

2.29 matrixSslGetCRL

```
int32 matrixSslGetCRL(sslKeys_t *keys, int32 (*crlCb)(psPool_t *pool,
      psX509Cert_t *CA, int append, char *url, uint32 urlLen),
      int *numLoaded);
```

Parameter	Input/Output	Description
keys	input	A `sslKeys_t` structure that has been populated with CA files from a previous call to `matrixSslLoadRsaKeys` (or equivalent).
crlCb	input	A user callback that will be invoked to allow the user to locate the CRL from a given URI
numLoaded	input/output	The number of CRLs that were successfully loaded during this process

Return Value	Test	Description
PS_SUCCESS	0	Success. There were no errors loading CRLs. The number loaded will be indicated by `numLoaded`, which could be 0
PS_ARG_FAIL	< 0	Failure. Callback not provided or no CA certificates present in `keys`
-1	< 0	CRL load error. At least one CRL failed to load. The number of successfully loaded will be indicated by `numLoaded`

Clients

The fetching and loading of Certificate Revocation Lists (CRL) is relevant to clients using the standard one-way SSL authentication handshake. This function should be called after Certificate Authority (CA) files have been loaded via `matrixSslLoadRsaKeys` to check if any CRL Distribution Point extensions are present in the X.509 CAs. If so, the `crlCb` will be invoked as an opportunity for the user to fetch the CRL and load it into the Matrix library using `matrixSslLoadCRL`.

The CA files will be parsed looking at the X.509 CRL Distribution Point extension. The parser supports the URI `fullName` member of the `distributionPoint` field of the ASN.1 specification (RFC 5280 Section 4.2.1.13). This is the most common field for CRL identification and will likely be a URL that specifies an HTTP or LDAP location. The URL will be passed to the user callback (detailed below).

Servers

This function is relevant to servers that are using client-authentication and have CA files that might contain CRL information.

The crlCb Callback

As a buffer-based library, MatrixSSL does not internally fetch the CRL via HTTP or LDAP. That is the responsibility of the user and the `crlCb` callback is the opportunity for that to occur.

The callback has two responsibilities:

1. Use the incoming `url` and `urlLen` parameters to fetch the CRL. If unable to fetch the CRL a negative return code should be returned to indicate an error.
2. Once the CA is successfully fetched, use the incoming `pool`, `CA`, and `append` parameters untouched to the `matrixSslLoadCRL` function to load the CRL into the CA. The callback should return the return value of `matrixSslLoadCRL`.

The callback will be invoked for each CRL distribution point that is encountered during the parse. The CRL information is copied internally so the callback should not allocate any memory that isn't freed before returning.

A callback example of an HTTP fetch and load is available in the *client.c* source code of the package.

Define Dependencies

USE_CRL	Must be enabled in *cryptoConfig.h*

2.30 matrixSslLoadCRL

```
int32 matrixSslLoadCRL(psPool_t *pool, psX509Cert_t *CA, int append,
      char *CRLbin, int32 CRLbinLen);
```

Parameter	Input/Output	Description
pool	input	The memory pool if using Matrix Deterministic Memory. NULL otherwise.
CA	input	The Certificate Authority associated with the `CRLbin` parameter
append	input	0 if loading a new CRL or if refreshing an existing list. 1 if appending a CRL to an existing list
CRLbin	input	The ASN.1 DER encoding of a X.509 v2 CRL
CRLbinLen	input	Byte length of `CRLbin`

Return Value	Test	Description
PS_SUCCESS	0	Success. CRL has been authenticated against the CA and the list loaded
PS_CERT_AUTH_FAIL_SIG	< 0	Failure. Given `CA` did not sign (issue) the given CRL
PS_ARG_FAIL	< 0	Failure. `CA` is NULL
PS_PARSE_FAIL	< 0	Failure. CRL could not be parsed
PS_MEM_FAIL	< 0	Failure. Internal memory allocation failure
PS_LIMIT_FAIL	< 0	Failure. Malformed ASN.1
PS_CERT_AUTH_FAIL_DN	< 0	Failure. CA distinguished name does not match the CRL distinguished name
PS_UNSUPPORTED_FAIL	< 0	Failure. CRL uses an unsupported signature algorithm

Clients

The fetching and loading of Certificate Revocation Lists (CRL) is relevant to clients using the standard one-way SSL authentication handshake. This function is used to load a given CRL into a given Certificate Authority. This function can be used in two scenarios.

1. This function should be used during the `crlCb` callback of the `matrixSslGetCRL` API. Once the CRL has been fetched in the callback, this `matrixSslLoadCRL` function should be called with the untouched `pool`, `CA`, and `append` parameters that were passed into the callback. The return value from this call should be used as the return value from the callback.
2. This function can be manually invoked at any time a CRL needs to be loaded to a CA.

The function authenticates that the CA was indeed the issuer of the CRL through a distinguished name test and the public key signature validation. If valid, the serial numbers of the revoked certificates are associated with the CA and checked against during each SSL connection.

During the handshake, if a revoked certificate is found being used the certificate callback will be invoked with an alert status of `SSL_ALERT_CERTIFICATE_REVOKED`. It is recommended the user terminate the handshake at this time.

Servers

This function is relevant to servers that are using client-authentication and want to load CRLs into their CAs.

Memory Profile

The internal CRL information will be freed during the call to `matrixSslDeleteKeys` of which the CA is a member.

Define Dependencies

USE_CRL	Must be enabled in *cryptoConfig.h*

3 MATRIXDTLS API

DTLS is an extension of the TLS protocol that enables the same strong level of security to be implemented over non-reliable transport mechanisms such as UDP. In addition to this API documentation, the MatrixDTLS Developer's Guide discusses all the differences that a developer needs to know when implementing MatrixDTLS.

3.1 Debug Configuration

The *matrixsslConfig.h* file contains the full set of compile-time configurable options for the protocol. Most of the features are documented in the **Configurable Features** section of the **Source Code Notes** chapter in this document. Below is the table of DTLS specific debug definitions that the user may set in the library.

Define	Description
DTLS_SEND_RECORDS_INDIVIDUALLY	If enabled, each handshake message will be returned individually when `matrixDtlsGetOutdata` is called. When left disabled, the default behaviour of `matrixDtlsGetOutdata` is to return as much data as possible that fits within the maximum PMTU.
USE_DTLS_DEBUG_TRACE	Enables DTLS-specific trace messages for debug

3.2 Integration Notes

With the exception of two functions, the entire MatrixSSL public API set is available for use in MatrixDTLS and this MatrixSSL API document is the primary technical reference for the interface for both products.

In MatrixDTLS the function `matrixDtlsGetOutdata` is used instead of `matrixSslGetOutdata` and the function `matrixDtlsSentData` is used instead of `matrixSslSentData`. The prototypes for these functions are identical to their MatrixSSL counterparts and are documented below.

The only other change that is required for DTLS use is to pass `SSL_FLAGS_DTLS` as the final `flags` parameter to `matrixSslNewClientSession` and `matrixSslNewServerSession`.

3.3 matrixDtlsGetOutdata

```
int32 matrixDtlsGetOutdata(ssl_t *ssl, unsigned char **buf);
```

Parameter	Input/Output	Description
ssl	input	The SSL session context
buf	output	Pointer to beginning of data buffer that needs to be sent to the peer

Return Value	Description
0	No pending data to send
> 0	The number of bytes in `buf` that need to be sent
PS_ARG_FAIL	Failure. Bad input parameters

This function must be used instead of `matrixSslGetOutdata`

Servers and Clients

Any time the application is expecting to send data to a peer this function must be called to retrieve the memory location and length of the encoded DTLS buffer. This API is used in conjunction with `matrixDtlsSentData` and MUST be called in a loop until it returns 0.

The length of encoded bytes in `buf` that needs to be sent is passed through the return code and that value will always be within the Maximum Transmission Unit that was set by default with the DTLS_PMTU define or the updated value set by `matrixDtlsSetPmtu`.

The unique DTLS functionality included in this version of `GetOutdata` is that it will return an encoded flight of handshake messages that has previously been sent. This resend case must be determined by the application itself if a timeout from the peer has occurred. This case is highlighted as number 7 in the following list.

There are several ways data can be encoded into outdata and ready to send:

1. After a client calls `matrixSslNewClientSession` this function must be called to retrieve the encoded CLIENT_HELLO message that will initiate the handshake
2. After a client or server calls `matrixSslEncodeRehandshake` this function must be called to retrieve the encoded SSL message that will initiate the re-handshake
3. If the `matrixSslReceivedData` function returns MATRIXSSL_REQUEST_SEND this function must be called to retrieve the encoded SSL handshake reply.
4. After the user calls `matrixSslEncodeWritebuf` this function must be called to retrieve the encrypted buffer for sending.
5. After the user calls `matrixSslEncodeClosureAlert` to encode the CLOSE_NOTIFY alert this function must be called to retrieve the encoded alert for sending.
6. After the user calls `matrixSslEncodeToOutdata` this function must be called to retrieve the encrypted buffer for sending.
7. If the application logic has determined a DTLS timeout has occurred during the handshake phase this function must be called to rebuild the previous flight of handshake message to be resent to the peer.

After sending the returned bytes to the peer, the user must always follow with a call to `matrixDtlsSentData` to update the number of bytes that have been sent from the returned `buf`. After each call to `matrixDtlsSentData` this function must be called again to set the resend state machine to the proper state.

3.4 matrixDtlsSentData

```
int32 matrixDtlsSentData(ssl_t *ssl, uint32 bytes);
```

Parameter	Input/Output	Description
ssl	input	The SSL session context
bytes	input	Length, in bytes, of how much data has been written out to the peer

Return Value	Test	Description
MATRIXSSL_REQUEST_SEND	> 0	Success. Call `matrixDtlsGetOutdata` again and send more data to the peer. The number of `bytes` sent was not the full amount of pending data.
MATRIXSSL_SUCCESS	0	Success. No pending data remaining.
MATRIXSSL_REQUEST_CLOSE	> 0	Success. If this was an alert message that was being sent, the caller should close the session.
MATRIXSSL_HANDSHAKE_COMPLETE	> 0	Success. Will be returned to the peer if this is the final FINISHED message that is being sent to complete the handshake.
PS_ARG_FAIL	< 0	Failure. Bad input parameters.

This function must be used instead of `matrixSslSentData`

Servers and Clients

This function must be called each time data has been sent to the peer. The flow of this function is that the user first calls `matrixDtlsGetOutdata` to retrieve the outgoing data buffer, the user sends part or all of this data, and then calls `matrixDtlsSentData` with how many bytes were actually sent.

The return value from this function indicates how the user should respond next:

MATRIXSSL_REQUEST_SEND - There is still pending data that needs to be sent to the peer. The user must call `matrixDtlsGetOutdata`, send the data to the peer, and then call `matrixDtlsSentData` again.

MATRIXSSL_SUCCESS - All of the data has been sent and the application will likely move to a state of awaiting incoming data. The application must call `matrixDtlsGetOutdata` next.

MATRIXSSL_REQUEST_CLOSE - All of the data has been sent and the application should close the connection. This will be the case if the data being sent is a closure alert (or fatal alert).

MATRIXSSL_HANDSHAKE_COMPLETE - This is an indication that this peer is sending the final FINISHED message of the SSL handshake. In general this will be an important return code for client applications to handle because most protocols will rely on the client sending an initial request to the server once the SSL handshake is complete. If a client receives this return code, a resumed handshake has just completed. For details on how to handle handshake completion see the MatrixDTLS Developer's Guide. The application must call `matrixDtlsGetOutdata` next.

3.5 matrixDtlsSetPmtu

```
int32 matrixDtlsSetPmtu(int32 pmtu);
```

Parameter	Input/Output	Description
pmtu	input	The new Path Maximum Transmission Unit size for a datagram. <0 to reset the default value defined by `DTLS_PMTU`

Return Value	Description
> 0	The new PMTU value

Servers and Clients

This function is used to modify the global PMTU setting for the library. It is essential that the server and client in a DTLS connection agree on the maximum datagram size they can send and receive. Unlike standard SSL/TLS protocols, fragmentation is not supported at the transport layer. In DTLS, a fragment must be encoded into a single datagram. The library handles this transparently.

3.6 matrixDtlsGetPmtu

```
int32 matrixDtlsGetPmtu(void);
```

Return Value	Description
> 0	The current PMTU value

Servers and Clients

Retrieve the current PMTU value.

4 THE CERTIFICATE VALIDATION CALLBACK FUNCTION

This section describes the `certValidator` parameter of the `matrixSslNewClientSession` and `matrixSslNewServerSession` functions.

4.1 Application Layer Certificate Acceptance

This callback offers a mid-handshake opportunity for a user to intervene in the authentication process. After receiving the CERTIFICATE handshake message the callback will be invoked and the user can determine whether the handshake should continue or whether a fatal alert should be sent and the handshake terminated. The callback will be invoked with the certificate material sent by the peer as well as the status of the public-key (RSA or ECC) authentication performed internally by the MatrixSSL library.

The registered callback function must have the following prototype:

```
int32 certValidator(ssl_t *ssl, psX509Cert_t *certInfo, int32 alert);
```

The `ssl` parameter is the session context and must be treated as read-only.

The `certInfo` parameter is the incoming `psX509Cert_t` structure containing information about the peer certificate or certificate chain. It is the certificate information in this structure that an application will generally wish to examine. If it is a certificate chain, the `next` member of the structure will link to the next certificate. The `next` member should be the parent (or issuer) of the current certificate. This certificate information is read-only from the perspective of the validating callback function. The structure members are specified in the **psX509Cert_t Structure** section of this document.

The incoming `alert` parameter will indicate whether or not the certificate chain passed the internal X.509 and RSA (or other public-key authentication) authentication checks. The alert member will be `MATRIXSSL_SUCCESS` (0) if the certificate chain was valid and the issuing CA was found. If alert is > 0 the authentication did not succeed and the value is the SSL alert ID and will be set to one of the following.

Value for incoming alert parameter	Description
0	Authentication success. The certificate chain received from the peer was valid and the issuing CA file was found.
SSL_ALERT_BAD_CERTIFICATE	Authentication failure. This alert is an indication that the certificate chain from the peer did not self-validate. No attempt to locate the issuing CA for the chain has been made if this alert is present.
SSL_ALERT_UNKNOWN_CA	Authentication failure. This alert is an indication that the certificate chain from the peer is valid but the issuing CA could not be found.
SSL_ALERT_CERTIFICATE_REVOKED	Authentication failure. The certificate has been checked against a user provided Certificate Revocation List and determined to be untrusted.

Table 3 - Certificate Callback Incoming "alert" Values

In addition to the bottom line `alert` value, the individual certificates in the `certInfo` parameter will indicate their own authentication status through the `authStatus` member of the `psX509Cert_t` structure. This is particularly useful if certificate chains are being used and the user would like to identify the specific certificate that did not internally authenticate. The callback can walk the subject certificate chain using the next member of the structure to find the first `authStatus` that is not set to `PS_CERT_AUTH_PASS`.

Regardless of the internal authentication tests and `alert` value, the callback function will ultimately determine whether or not to continue the SSL handshake through the return value it chooses.

Return Value from the Certificate Callback Function	Description
0	Continue handshake. The user callback is indicating that it accepts the certificate material. If an authentication alert was internally set, it will be ignored and cleared.
> 0	Fail the handshake; **return a fatal alert**, and close connection with peer. The positive value is the SSL alert ID as defined in *matrixssllib.h.* The incoming alert parameter may be one of `SSL_ALERT_BAD_CERTIFICATE` or `SSL_ALERT_UNKNOWN_CA` and it is recommended those be passed through in the return code. Other alert codes can be found in the table below.
< 0	Fail the handshake; issue a fatal `INTERNAL_ERROR` alert, and close connection with peer. This return code should be used if the user callback code itself encounters an unrecoverable error.
SSL_ALLOW_ANON_CONNECTION	Continue handshake. The user callback is acknowledging that the certificate has not been authenticated but it is being allowed to continue. See the section immediately below for more information.

Table 4 - Certificate Callback Return Value Ranges

SSL Alerts for Failed Authentication

The internal library will test only the X.509 values of DistinguishedName and BasicConstraints, the revocation status (if feature is enabled), and perform the mathematical public key operation to determine authentication status. The certificate callback can be used to perform additional authentication tests and return the alert status based on those tests. The following table shows the possible options that may be returned.

Fatal Alert Return Values for Certification Callback	Description
SSL_ALERT_BAD_CERTIFICATE	A certificate was corrupt, contained signatures that did not verify correctly, etc. This value could already be the incoming `alert` value.
SSL_ALERT_UNKNOWN_CA	A valid certificate chain or partial chain was received, but the certificate was not accepted because the CA certificate could not be located or couldn`t be matched with a known, trusted CA. This value could already be the incoming `alert` value.
SSL_ALERT_CERTIFICATE_REVOKED	The certificate was revoked by its signer. This value could already be the incoming `alert` value.
SSL_ALERT_CERTIFICATE_EXPIRED	A certificate has expired or is not currently valid based on the `notBefore` and `notAfter` values.
SSL_ALERT_CERTIFICATE_UNKNOWN	Some other (unspecified) issue arose in processing the certificate, rendering it unacceptable.
SSL_ALERT_ACCESS_DENIED	A valid certificate was received, but when access control was applied, the sender decided not to proceed with negotiation.
SSL_ALERT_UNSUPPORTED_CERTIFICATE	A certificate was of an unsupported type.

Table 5 - Certificate Callback SSL_ALERT Return Values

Anonymous Connections

The callback may also choose to return `SSL_ALLOW_ANON_CONNECTION` if the user wishes to continue a connection despite a `PS_CERT_AUTH_FAIL` indication on any of the certificates. If this return value is used, the handshake will continue and will result in a secure (data encryption) but unauthenticated SSL connection. If this return value is used, the `matrixSslGetAnonStatus` function may be used during the lifetime of the connection to verify the status.

It is important to note that this anonymous connection mechanism is not related to anonymous cipher suites. The certificate validation callback is only invoked for cipher suites that utilize public key authentication. Therefore, it is not advised to allow anonymous connections using this mechanism. If anonymous connections are desired, it is recommended that an anonymous cipher suite be used instead.

Server (Client-Authentication)

In client authentication handshakes the server will need to implement the callback function as well.

By default, the MatrixSSL server will immediately terminate the handshake if the client replies to the server CERTIFICATE_REQUEST message with an empty CERTIFICATE message. If the server wishes to potentially continue the connection, the compile time define `SERVER_WILL_ACCEPT_EMPTY_CLIENT_CERT_MSG` in *matrixsslConfig.h* must be enabled. If enabled, the certificate callback function will be invoked with a NULL `certInfo` parameter and an alert status of `SSL_ALERT_BAD_CERTIFICATE`. If the user callback determines the handshake can continue without client-authentication the handshake is effectively "downgraded" on-the-fly to a standard handshake.

4.2 psX509_t Structure

```
typedef struct psCert {
      int32                   version;
      unsigned char           *serialNumber;
      int32                   serialNumberLen;
      x509DNattributes_t      issuer;
      x509DNattributes_t      subject;
      int32                   notBeforeTimeType;
      int32                   notAfterTimeType;
      char                    *notBefore;
      char                    *notAfter;
      psPubKey_t              publicKey;
      int32                   pubKeyAlgorithm;
      int32                   certAlgorithm;
      int32                   sigAlgorithm;
      unsigned char           *signature;
      int32                   signatureLen;
      unsigned char           sigHash[MAX_HASH_SIZE];
      unsigned char           *uniqueIssuerId;
      int32                   uniqueIssuerIdLen;
      unsigned char           *uniqueSubjectId;
      int32                   uniqueSubjectIdLen;
      x509v3extensions_t      extensions;
      int32                   authStatus;
      unsigned char           *unparsedBin;
      int32                   binLen;
      struct psCert           *next;
} psX509Cert_t;

typedef struct {
      char        *country;
      char        *state;
      char        *locality;
      char        *organization;
      char        *orgUnit;
      char        *commonName;
      char        hash[SHA1_HASH_SIZE];
      char        *dnenc;
      int32       dnencLen;
      short       countryType;
      short       countryLen;
      short       stateType;
      short       stateLen;
      short       localityType;
      short       localityLen;
```

```
      short       organizationType;
      short       organizationLen;
      short       orgUnitType;
      short       orgUnitLen;
      short       commonNameType;
      short       commonNameLen;
} x509DNattributes_t;

typedef struct {
      x509extBasicConstraints_t     bc;
      x509GeneralName_t       *san;
} x509v3extensions_t;

typedef struct {
      int32       cA;
      int32       pathLenConstraint;
} x509extBasicConstraints_t;


typedef struct psGeneralNameEntry {
      int32                         id;
      unsigned char                 name[16];
      unsigned char                 *data;
      int32                         dataLen;
      struct psGeneralNameEntry     *next;
} x509GeneralName_t;
```

This is the data type that stores the parsed information from an X.509 certificate file. The X.509 format is somewhat complex, so we document the most important fields here.

This data type is most important in the context of the session creation APIs in which the application registers a custom function to be invoked during the SSL handshake to validate the peer certificate. This registered callback function may wish to perform custom checks on the individual members of the `psX509Cert_t` structures that are passed in.

version	X.509 version. MatrixSSL supports v3 certificates. `0` = v1, `1` = v2, `2` = v3
serialNumber	Serial number issued to this certificate. Some certificates insert non-integer values for this member
serialNumberLen	Byte length of `serialNumber`
issuer	Distinguished Name of the CA that issued this certificate. See `x509DNattributes_t`
subject	Distinguished Name of this certificate. See `x509DNattributes_t`
notBeforeTimeType notAfterTimeType	Format specification for the `notBefore` and `notAfter` members of this structure. Either `ASN_UTCTIME` or `ASN_GENERALIZEDTIME`
notBefore	NULL terminated UTCTime or GeneralizedTime indicating the valid start date for the certificate
notAfter	NULL terminated UTCTime or GeneralizedTime indicating the valid end date for the certificate
publicKey	The public key of this certificate. See `psPubKey_t`
pubKeyAlgorithm	The algorithm identifier for the public key encryption mechanism this certificate is using. Either `OID_RSA_KEY_ALG` or `OID_ECDSA_KEY_ALG`

certAlgorithm	The algorithm identifier the issuing CA used to sign this certificate. Supported values are found in the `/* Signature algorithms */` section of the *cryptolib.h* file. This value must match `sigAlgorithm` and that is tested internally during certificate parsing.
sigAlgorithm	The verification of the signature algorithm the issuing CA used for this certificate. The `/* Signature algorithms */` section of the *cryptolib.h* file defines the possible values. This value must match `certAlgorithm` and that is tested during certificate parsing.
signature	The full CA-generated digital signature for this certificate that binds the subject to the CA private key
signatureLen	The byte length of signature
sigHash	The digest hash portion of the signature used internally for public key authentication
uniqueIssuerId	Optional certificate field to handle possible reuse of the issuer name. See section 4.1.2.8 of RFC 3280 for more information.
uniqueIssuerIdLen	Byte length of `uniqueIssuerId`
uniqueSubjectId	Optional certificate field to handle possible reuse of the subject name. See section 4.1.2.8 of RFC 3280 for more information.
uniquesSubjectIdLen	Byte length of `uniqueSubjectId`
extensions	The X.509 certificate extensions for this certificate. See `x509v3extensions_t`
authStatus	This flag is set on subject certificates when `psX509AuthenticateCert` is called. The value indicates the public key authentication status of whether the issuer certificate is the CA of the subject certificate. MatrixSSL calls this internally before the user's custom certificate verification callback is invoked so the user can examine it. The value may be; `PS_FALSE` = untested (chain validation stops on first certificate to fail so this should only be set on certificates beyond the one that did not pass) `PS_CERT_AUTH_PASS` = successfully authenticated `PS_CERT_AUTH_FAIL_BC` = failed authentication because the issuing certificate did not have CA permissions `PS_CERT_AUTH_FAIL_DN` = failed authentication because the Distinguished Name of the issuer did not match the DN of the issuer `PS_CERT_AUTH_FAIL_SIG` = failed authentication because the public key signature did not validate
unparsedBin	The raw ASN.1 binary stream of this certificate (if applicable).
binLen	Byte length of `unparsedBin`
next	Pointer to the next `psX509Cert_t` if this is a chain of certificates

Table 6 - Important psX509_t Structure Members

The DistinguishedName X.509 attribute is the plaintext description of the certificate owner and issuer.

country state locality organization orgUnit commonName	The self-identifying collection of supported string attributes that comprise the Distinguished Name. Distinguished Names are used to identify the subject and issuer of an X.509 certificate.

countryType stateType localityType organizationType orgUnitType commonNameType	These members specify the ASN.1 string type for their corresponding char* string members (ie. countryType for country). Types can be found in the *crypto/keyformat/asn1.h* header file `ASN_UTF8STRTING` (8-bit chars) == 12 `ASN_PRINTABLESTRING` (8-bit chars) == 19 `ASN_IA5STRING` (8-bit chars) == 22 `ASN_BMPSTRING` (16-bit chars) == 30
countryLen stateLen localityLen organizationLen orgUnitLen commonNameLen	These members specify the byte length for their corresponding char* string members. **The length includes two terminating NULL bytes.**
hash	A digest representation of the above attributes used for easy comparisons of DN
dnenc	The unparsed ASN.1 stream of the DN (if applicable)
dnencLen	Byte length of `dnenc`

Table 7 - x509DNattributes_t Structure Members

X.509 extensions are held in the `extensions` member.

bc	The critical Basic Constraints extension. See `x509extBasicConstraints_t`
san	The Subject Alternative Name extension. This extension is used to associate additional identities with the certificate subject. Common alternate identities include email addresses and IP addresses. See `x509GeneralName_t`

Table 8 - x509v3extensions_t Structure Members

`x509extBasicConstraints_t` Members

cA	Boolean to indicate whether this certificate is a Certificate Authority.
pathLenConstraint	If `cA` is true, this member indicates the maximum length that a certificate chain may extend beyond this CA.

`x509GeneralName_t` Members

id	Integer identifier of the name type. `id` to `name` mappings 0 = “other”, 1 = “email”, 2 = “DNS”, 3 = “x400Address”, 4 = “directoryName”, 5 = “ediPartyName”, 6 = “URI”, 7 = “iPAddress”, 8 = “registeredID”, x = “unknown”
name	String identifier for the name type. Possible values are the quoted names from the list above.
data	The data value for the alternate name
dataLen	Byte length of `data`
next	The next `x509GeneralName_t` alternate name in this extension.

5 QUICK REFERENCE

API	Description	API Dependencies
matrixSslOpen matrixSslClose	One time initialization and clean up for MatrixSSL	
matrixSslNewKeys matrixSslDeleteKeys matrixSslLoadRsaKeys	Key management functions	matrixSslNewKeys must be called prior to calling matrixSslLoadRsaKeys
matrixSslNewClientSession matrixSslNewServerSession matrixSslDeleteSession	Respective session initialization and common session deletion	
matrixSslGetOutdata	Retrieve encoded data that is ready to be sent out over the wire to the peer	Must be followed by a call to matrixSslSentData
matrixSslReceivedData	Any data received from the peer must be passed to this function	An empty data buffer must have been retrieved by a prior call to matrixSslGetReadbuf
matrixSslProcessedData	Must be called each time the application is done processing plaintext data	Plaintext data will only be given to the application when the return code from matrixSslReceivedData or matrixSslProcessedData is MATRIXSSL_APP_DATA or MATRIXSSL_RECEIVED_ALERT
matrixSslGetWriteBuf matrixSslEncodeWriteBuf - OR - matrixSslEncodeToOutdata	Used for encoding plaintext application data after SSL handshake that will be sent to the peer	matrixSslGetWriteBuf must be called to get an empty buffer in which to copy plaintext. matrixSslEncodeWriteBuf must be called to do the actual encryption. Encrypted data must be retrieved with matrixSslGetOutdata

APPENDIX A - LIST OF TABLES