MatrixSSL Design Documentation

TABLE OF CONTENTS

1 INTRODUCTION

SSL (Secure Socket Layer) is an application layer protocol that enables two peers to authenticate each other and exchange data in an encrypted envelope. The purpose of the protocol is to prevent eavesdroppers or a man-in-the-middle from seeing the data being exchanged. The MatrixSSL product is a toolkit that enables any application running on any platform to incorporate the SSL protocol into its communications.

TLS (Transport Layer Security) is a more recent and descriptive name for the security protocol and is synonymous with SSL for the purposes of this document.

SSL is a ***client/server*** protocol in which the client initiates the ***handshake***, which allows the two entities to authenticate one another and to agree upon cryptographic keys that will be used to encrypt and decrypt application data. The handshake is essentially the SSL protocol itself and is a back-and-forth exchange of ***handshake messages***. Collections of handshake messages that are sent to the peer in one pass are referred to as a ***flight***.

At the successful completion of the handshake, the two peers enter the ***application data exchange*** portion of the communication using the agreed upon keys from the handshake negotiation stage.

Every data fragment regardless of whether it is a handshake message or an application data message is formatted as an SSL ***record***. Each record contains basic header information about the data to follow.

1.1 How to use this document

This document provides source code details that fall outside the scope of the standard documentation set. It is intended for those wishing to understand more about the software architecture and interactions of the library. This document does not detail the SSL protocol itself. The MatrixSSL API, Developer's Guide and Porting Guide should be read before this document in order to better understand the division between the public interfaces and internal workings described here.

1.2 Background

MatrixSSL 1.0 was written in 2003 to fulfil the need of transport security for applications running on embedded platforms that did not have sufficient memory resources or operating system support to run the ubiquitous OpenSSL software project.

MatrixSSL has been under active development since that initial release, adding protocol updates, cryptographic algorithms, and keeping up on security updates and feature additions.

1.3 Design Goals

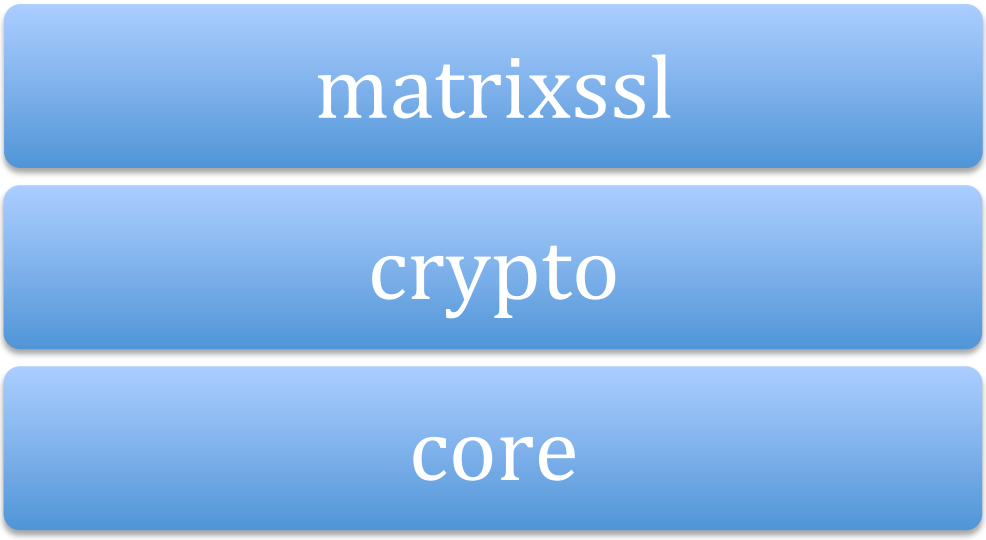
The primary design goal of MatrixSSL is to remain platform agnostic in terms of operating system, hardware architecture, and data transport mechanism. MatrixSSL does have many platform specific optimizations available to customers but those do not generally affect the design of the software and are outside the scope of this document.

The secondary design goal of MatrixSSL is to be an SSL solution for embedded environments by maintaining a small binary code size and small RAM footprint at runtime.

2 ARCHITECTURE DESIGN

2.1 Overview

MatrixSSL is layered as three modules. Each module is a top-level directory in the package structure.



Core is the foundation module and implements the lowest level functionality and platform dependencies.

Crypto contains all the raw cryptographic algorithms that SSL relies upon. It also contains the format parsers for X.509, ASN.1, and private keys.

Matrixssl is the implementation of the SSL protocol for servers and clients.

These are logical modules that exist within a single MatrixSSL binary object. The modules interact only through internal C code function calls. There are no TCP or out-of-band communications between the three modules.

2.2 Code Files

This section summarizes which features/functionality each source code file is responsible for.

2.2.1 Matrixssl

File	Functionality
sslEncode.c	Creates handshake response flights. Encrypts application data.
sslDecode.c	Parses and processes incoming handshake flights.
tls.c	TLS master secret generation. TLS HMAC generation. Activates the read and write cipher states for a given SSL session. Location of client hello extension APIs.
sslv3.c	SSLv3 finished hash calculation. SSLv3 master secret generation. SSLv3 HMAC generation.

hsHash.c	TLS finished hash calculation. Maintains the on-going handshake hash (and snapshot hash for client authentication)
prf.c	Digest-based pseudo random functions.
cipherSuite.c	Associates cryptographic parameters with the supported cipher suites. Cipher suite specific tests to validate that the loaded key material supports the cipher suite being negotiated.
matrixssl.c	Library open and close APIs. Top-level key material loading APIs. SSL session open, configure, and close APIs. Server side session resumption cache. SSL memory pool open and close APIs.
matrixsslApi.c	Most of the public APIs.

2.2.2 Crypto

The crypto module is very straightforward in terms of the file naming conventions. The subdirectory name is the class of cryptographic operation and the file name is the implementation of the algorithm or an identification of the specification being implemented such as *x509.c* and *asn1.c*.

2.2.3 Core

Core is the foundation module and implements the lowest level functionality.

There is a set of platform-specific functionality that must be implemented in the core module and is contained within the *osdep.c* file. The MatrixSSL Porting Guide is the technical reference guide for this portion of the product.

File	Functionality	Consumers (module)
osdep.c	Return the current platform time (real or ever-increasing ticks)	servers (matrixssl) clients (matrixssl)
osdep.c	Return entropy bytes	PRNG (crypto)
osdep.c	File access if reading certificate/keys from PEM formats on disk	servers (matrixssl) clients (matrixssl)
osdep.c	Implement where the psTrace set of APIs will output their messages	many places
list.h	A fast, circular, doubly-linked list implementation	psMalloc (core)
list.h	A basic single-linked list	X.509 (crypto)
psmalloc.c	The matrix deterministic memory feature, psMalloc	many places

Table 1 -

2.3 Data

2.3.1 Matrixssl structures

The primary MatrixSSL data structures are defined in *matrixssllib.h*

2.3.1.1 ssl_t

The primary data structure in MatrixSSL is the `ssl_t` session structure. This type is instantiated at session creation and exists until session deletion. It is the context data parameter that is passed to the majority of public APIs. Some of the more important members stored in the data type are the input/output buffers and members that control the SSL state. There is no need for an application to inspect the contents of the `ssl_t` type.

When using the Deterministic Memory feature of MatrixSSL, the `sizeof(ssl_t)` structure represents the entirety of the SESSION_POOL and requires about 2-3KB of storage depending on which features are enabled.

An `ssl_t` structure holds pointers for four important sub-structures: `sslKeys_t`, `sslRec_t`, `sslCipherSpec_t`, and `sslSec_t`. These structures are discussed below.

2.3.1.2 sslKeys_t

This data type holds the certificate and optional private key information for the SSL entity that is nominated by the application. This data exists longer than the session lifecycle and is allocated and freed by `matrixSslNewKeys` and `matrixSslDeleteKeys`.

Key data is essential to the SSL handshake process. It enables the ability to perform public key authentication and key exchange.

- Used during cipher suite negotiation to determine if the key material will support the requested cipher suite (`haveKeyMaterial` function)
- During the CERTIFICATE and CERTIFICATE_VERIFY messages to authenticate peers
- During the SERVER_KEY_EXCHANGE and CLIENT_KEY_EXCHANGE messages to perform key exchange
- During the CERTIFICATE_REQUEST message of client-authentication when the server tells the client which Certificate Authorities it supports

When referenced from within an `ssl_t` context, the `sslKeys_t` member is not duplicated. So it is important that `matrixSslDeleteKeys` is not called until all sessions that use the key material have been shutdown.

2.3.1.3 sslRec_t

The `sslRec_t` data type contains information about the current (most recent) SSL record being parsed. In addition to the length of the current record, the most important field in an SSL record header is the ***content type***. This field indicates whether the record is a handshake message, application data, or an alert. The state machine makes sure all records that arrive are expected.

2.3.1.4 sslCipherSpec_t

The `sslCipherSpec_t` data type holds the encryption and MAC information for the SSL cipher suite that has been negotiated. In addition to the key lengths, IV lengths and MAC lengths for message size pre-calculations; the data type holds the function pointers that perform the raw encryption/decryption and MAC generate/verify functionality from the crypto module.

The `type` member is also a very important member of this structure and is used to classify the required key exchange method that determines which path the handshake will take (Diffie-Hellman or RSA, for

example). The type member is transitioned to the overall `ssl_t flags` member in the internal `matrixSslSetKexFlags` function in *cipherSuite.c*.

The full set of supported cipher suites are collected in an array of `sslCipherSpec_t` types in *cipherSuite.c*. The array name is `supportedCiphers`.

2.3.1.5 sslSec_t

The `sslSec_t` data type holds the specific per-session cryptographic data such as the symmetric key material and the public key information from the peer. These data members are managed to have the shortest possible lifecycle necessary.

2.3.1.6 sslSessionId_t

The session ID data type is used on the client side for SSL resumption. It is allocated and freed on the client side with the APIs `matrixSslNewSessionId` and `matrixSslDeleteSessionId`. It is passed as a parameter to the `matrixSslNewClientSession` API. If empty, a full handshake will be performed and will be internally populated with the resulting session ID. On subsequent `matrixSslNewClientSession` API calls, the library will find the session ID and attempt a resumed handshake.

2.3.1.7 sslSessionEntry_t

The `sslSessionEntry_t` data type is the server side of supporting session resumption. This data type is stored in a simple list of size `SSL_SESSION_TABLE_SIZE` and is searched when a client sends a session ID in the CLIENT_HELLO message.

2.3.2 Application-level Buffers

The data buffers are the byte streams that are sent and received between the two peers during an SSL connection. They are used to hold incoming and outgoing handshake data during the handshake phase of the connection and also the application data during the application data exchange phase. MatrixSSL internally manages these data buffers and that can be most easily understood by looking at the primary public APIs such as `matrixSslGetOutdata`, `matrixSslGetReadbuf`, and `matrixSslGetWritebuf`.

Internally, there are two buffers: the **input buffer** and the **output buffer** and are identified within the `ssl_t` structure by the `inbuf` and `outbuf` members and are managed with the `inlen`, `insize`, `outlen`, and `outsize` indexes.

During the application data exchange phase these two buffers are used as the naming suggests. The empty input buffer is retrieved with a call to `matrixSslGetReadbuf`. After the buffer is populated with incoming data it is decoded in-situ and the decrypted data is returned to the caller with `matrixSslRecievedData`. Likewise, the empty output buffer is retrieved during `matrixSslGetWritebuf` and the in-situ encrypted result is returned to the caller with `matrixSslGetOutdata`.

During the handshake phase, the input and output buffers are not as clearly delineated. The same buffer location is used for both receiving the incoming flight and writing the outgoing flight. So it is the `inbuf` that is used for the outgoing flight creation. In this case, the library will swap the `inbuf` and `outbuf` pointers after the flight creation so the standard public API calls will pull off the outgoing data when called upon. You can see this swap in the `matrixSslReceivedData` function in the `SSL_SEND_RESPONSE` action case.

The two data buffers are assigned default sizes that are set by the user with the `SSL_DEFAULT_IN_BUF_SIZE` and `SSL_DEFAULT_OUT_BUF_SIZE` defines in *matrixsslConfig.h*. The buffers will grow as needed up to the specification record maximum of 16KB (plus header overhead). If the buffers do have to grow, they will be shrunk down to the default size after the data has been processed. The resize is performed by the `revertToDefaultBufsize` and is called on the `inbuf` as the last step in `matrixSslReceivedData` and as part of the processing of `matrixSslProcessedData`. The `outbuf` is resized by `matrixSslSetData`.

2.4 Error Handling

2.4.1 Protocol Errors

A primary responsibility of the library is to enforce the SSL/TLS specifications in terms of message formats and expected results from cryptographic operations. The `matrixSslDecode` function is the enforcer of the message formats and if an error is found the `err` member of the `ssl_t` session is used to set the SSL alert and the processing immediately stops and a fatal alert message is created and queued for sending.

Upon creating the fatal alert the `flags` member of `ssl_t` is set to `SSL_FLAGS_ERROR`, which will prevent the public APIs from creating or receiving any future records.

When receiving a fatal alert, the `flags` member of `ss_t` is set to `SSL_FLAGS_ERROR` which will prevent the public APIs from sending or receiving any more data on the connection. The user should always be closing a connection at the application upon receiving a fatal alert.

2.4.2 Library Errors

The vast majority of internal functions in MatrixSSL use an integer return code to signify the success or failure of the function. A value of <0 generally indicates a failure and a value of 0 indicates success. In some cases, a >0 return code will be used to return a useful data value and imply that the function was a success.

There is no generic error management framework in the library. Each error must be bubbled up to the calling public API where the user must inspect the status of the call to determine what to do next.

Being a standard C code library, there is no inherent garbage collector and so every function has to be fully aware of the context it can be called from and memory freed where appropriate.

2.4.3 User Errors

Effort has been made to ensure that bad function parameter values passed to the public APIs are handled gracefully.

Tests are not performed to ensure that gross misuse of the public API is error handled gracefully.

3 OPERATION

3.1 Overview

MatrixSSL is toolkit that exposes a public API for applications to engage in SSL communications. The library itself does not run any processes or dictate a run-time model for the application that is using the interface. The only exception to this is that if it is known a MatrixSSL-enabled server will be used in a multi-threaded environment that will serve simultaneous client connections, the `USE_MULTITHREADING` define in *coreConfig.h* should be enabled to support mutex locking on the global session ID cache.

The matrixssl module can be separated into two major components: the handshake and the application data exchange. The handshake is the SSL protocol itself in which messages are exchanged to perform authentication and to agree on the shared encryption and decryption keys for use by the application data exchange component. Collections of handshake messages that are sent to the peer are referred to as a **flight**.

3.2 Handshakes

A primary duty of the handshake component is to track the state of the protocol so that the proper, expected flight is being created or parsed for the given entity. The state machine at this level of handshake control consists of the `hsState` and `flags` member of the `ssl_t` session structure.

3.2.1 Receiving and Parsing Flights

Data received from a peer is passed to the `matrixSslReceivedData` public API function. This function is the buffer management wrapper around the actual parsing of the handshake flight. After some initial sanity tests, the `matrixSslDecode` workhorse function is called (see details in Important Functions below). The result of `matrixSslDecode` is the action that `matrixSslRecievedData` must perform on the data buffers. During the handshake, the final expected action is `SSL_SEND_RESPONSE`. This is the indication that `matrixSslDecode` was able to parse the expected flight and internally construct the response flight (or alert) that is ready to be sent to the peer. Before reaching this desired `SSL_SEND_RESPONSE` action it is possible some of the other actions will be encountered:

The `matrixSslDecode` function will only process a single record at a time. So, the action for `SSL_SUCCESS` is to test if more data is available in `inbuf` and re-invoke `matrixSslDecode` if so. During that test, `inbuf` will be 'packed' so that the start of the next available record will be moved to the front the buffer to aid in ease of buffer management and to keep the available 'free' space as large as possible for cases where more incoming data will need to be appended.

The action for `SSL_FULL` is an indication that the response flight is larger than the current buffer size. The action is to grow the buffer with a reallocation call and invoke `matrixSslDecode` again. Note that the `SSL_FULL` handler is growing the `inbuf` member rather than the `outbuf` member. Recall in the discussion of Application-Level Buffers above that the `inbuf` is used for both incoming and outgoing handshake flights and the swap with `outbuf` does not occur until the `SSL_SEND_RESPONSE` action is hit.

The action for `SSL_PARTIAL` is an indication that a full record is not available in `inbuf` for `matrixSslDecode` to parse. The response is to ensure that the `inbuf` is large enough to hold the incoming record and then return `MATRIXSSL_REQUEST_RECV` to the caller. The reason the action handler will know if `inbuf` is large enough is because the SSL record header will contain the record size and `matrixSslDecode` will pass back that hint in the `reqLen` output parameter.

If there is an error while parsing the incoming flight or if there is an error performing a cryptographic operation related to the data sent in the incoming flight the `MATRIXSSL_ERROR` action will be hit. In this case, the negative return value will be immediately passed to the caller of `matrixSslReceivedData` and the user should shutdown the connection.

3.2.1.1 End-of-handshake

The obvious exception to the expected `SSL_SEND_RESPONSE` action during a handshake is when the peer is receiving the final message of the handshake and does not need to reply. In this case, the `MATRIXSSL_SUCCESS` action is hit and a manual test of the SSL state is performed to see if this is truly the completion of the handshake. In this case, the special return code of `MATRIXSSL_HANDSHAKE_COMPLETE` is returned to the caller to let them know the application data exchange phase is active.

The `MATRIXSSL_SUCCESS` case will only test for handshake completion if there is no more incoming data queued in the `inbuf`. It is possible a peer might be sending the final FINISHED handshake message and immediately sending application data in the same flight. In this scenario, `matrixSslDecode` will be invoked immediately and the `SSL_PROCESS_DATA` action will be hit. There you will see the same end-of-handshake scenario being tested for so that the proper `BFLAG_HS_COMPLETE` flag may be set. The caller will never receive the `MATRIXSSL_HANDSHAKE_COMPLETE` return code in this scenario but it will be implicit to them because they are receiving the `MATRIXSSL_APP_DATA` code.

3.2.2 Creating Response Flights

If you read the section on Receiving and Parsing Flights you will have gleaned that flight response creation is 'automatically' done as part of the `matrixSslReceivedData`/`matrixSslDecode` responsibilities. Details on the state machine and creation actions are detailed in the Important Functions section for `sslEncodeResponse` below.

3.2.2.1 Postponed Flight Encryption

The response handshake flight is not encrypted (when applicable) until all the plaintext flight data has been written to the buffer. This design choice was introduced to better support platforms on which non-blocking hardware acceleration is used for public key operations. The state machine is not well suited for re-entering the single flight buffer at specific locations to continue message creation after a public key operation has finished. So the postponement mechanism allows the entire flight to be written in one pass and the buffer locations for public key operations and message boundaries to be saved in simple lists.

The message boundaries are held in the `flightEncode` member of the `ssl_t` structure. The public key operation data is held in the `pkaAfter` member. The encryption of the flight is done in `encryptFlight`.

Although more complex for the standard use-case this postponement method does have an offshoot benefit that, if in the encrypted state of a re-handshake, an error occurs in writing a handshake message the alert that is triggered will be encrypted using the current encryption state so the peer will have no trouble decoding the alert.

3.2.3 Transition to encryption states

Another primary responsibility of the handshake state machine is to transition from plaintext data exchange to encrypted data exchange. This happens during the handshake when the CHANGE_CIPHER_SPEC message is sent/received. Therefore, the FINISHED message of the handshake will always be encrypted.

It is necessary to keep the encrypted read distinct from the encrypted write because the first side to send the encrypted FINISHED will still have to process the plaintext CHANGE_CIHPER_SPEC message. The functions that perform the transition are aptly named `sslActivateWriteCipher` and `sslActivateReadCipher`.

`sslActivateWriteCipher` is called immediately after generating the CHANGE_CIPHER_SPEC message in `writeChangeCipherSpec`. The function performs the transition by setting `ssl_t` encryption members based on the `sslCipherSpec_t` and `sslSec_t` definition for the agreed upon suite and symmetric keys. Specifically, the `sslCipherSpec_t` data is used to set the `ssl_t` members `encrypt`, `generateMac`, `nativeEnMacSize`, `enMacSize`, `enBlockSize`, and `enIvSize` are set. The `sslSec_t` data is used to set the `writeMAC`, `writeKey`, and `writeIV` members. The final important setting is the `flags` parameter to enable `SSL_FLAGS_WRITE_SECURE` which is the primary test the state machine will use to check for secure write status.

sslActivateReadCipher is called immediately after parsing the CHANGE_CIPHER_SPEC message in matrixSslDecode. The function performs the transition by setting ssl_t decryption members based on the sslCipherSpec_t and sslSec_t definition for the agreed upon suite and symmetric keys. Specifically, the sslCipherSpec_t data is used to set the ssl_t members decrypt, verifyMac, nativeDeMacSize, deMacSize, deBlockSize, and deIvSize are set. The sslSec_t data is used to set the readMAC, readKey, and readIV members. The final important setting is the flags parameter to enable SSL_FLAGS_READ_SECURE which is the primary test the state machine will use to check for secure read status.

In re-handshake scenarios, the entire second handshake will be encrypted but the CHANGE_CIPHER_SPEC message will still transition to the newly agreed symmetric encryption state in the same way described above.

3.3 Application Data Exchange

The MatrixSSL API document covers the user interface for this phase of SSL communications. This section describes the internal design.

3.3.1 Encrypting Data

Application data encryption is handled by the internal function matrixSslEncode. The function is quite easy to follow and consists of creating the record header using writeRecordHeader and the symmetric encryption of the data itself using encryptRecord.

The only complicating aspect to matrixSslEncode is the check for whether the USE_BEAST_WORKAROUND define is enabled to thwart a known CBC mode attack for TLS 1.0 and lower. The solution to the exploit is to simply encrypt the first byte of each plaintext message into its own record and encrypt the remainder in a second record.

3.3.2 Decrypting Data

The initial path for decrypting application data is identical to that of a handshake flight parse. matrixSslRecievedData is called for any incoming data into the library and matrixSslDecode is invoked to handle the decryption. Details on matrixSslDecode are below.

3.4 Corner Cases

Re-handshaking scenarios can be a particularly complex area due to the fact that handshake messages may be arriving at any point during an existing connection.

3.4.1 Application Data Queued and Re-handshake Message Received

The scenario here is that a peer has encoded application data for sending but then calls matrixSslReceivedData to discover the peer is requesting a re-handshake. This case is handled in the SSL_SEND_RESPONSE case of matrixSslReceivedData by making sure to insert the outgoing application data before the outgoing response flight in the buffer. Well-designed applications can avoid this corner cases by handling all pending outgoing data prior to accepting incoming data.

3.4.2 Server Sends HELLO_REQUEST with Application Data Appended

The scenario here is that a server can send a HELLO_REQUEST handshake message to request a re-handshake but also append an application data record right behind it in the same flight. This is a complicated scenario for the state machine and public API because the HELLO_REQUEST parse will trigger the process of a re-handshake and prepare the state machine for sending a CLIENT_HELLO response handshake flight to the caller. Then the application data record will be immediately encountered and the client will be in the wrong state to be accepting those. MatrixSSL handles this situation by ignoring the HELLO_REQUEST and giving precedence to the application data. Ignoring a HELLO_REQUEST is

allowed by the SSL specification. The handling of this corner case can be found in `matrixSslDecode` by searching for the `flagsBk` member of the `ssl_t` structure. This member is one in a set of flags that are used to rewind the state machine from the HELLO_REQUEST processing and put it back into an application data exchange state.

3.5 Workhorse Functions

The majority of the processing in the MatrixSSL library happens within a relatively small number of functions. This section highlights the important functions of the product.

3.5.1 sslEncodeResponse

The function `sslEncodeResponse` manages the creation of a response flight during handshaking. It is invoked from `matrixSslDecode` when the state machine determines a response flight is required (or an alert needs to be created).

The function is invoked without any explicit indication of the session state. The first test is to check if the session has an alert status active. If so, `writeAlert` is invoked to create the alert message and the function returns immediately with a `MATRIXSSL_SUCCESS` status, which will lead to an `SSL_SEND_RESPONSE` action to the public API, `matrixSslReceivedData`. If an alert status is not active, the `hsState` is used to determine where the entity is in the handshake protocol to begin the proper flight creation. The `hsState` is always tracking the next expected incoming handshake message so the case statements are implemented so that the flight being created will produce that expected response from the peer.

The code logic becomes very manual at this point. Because there are several different handshake variants the creation of the response flight might include tests for client or server, whether the handshake is in a client authentication mode, and what key exchange mode the active cipher suite uses. There may also be tests for the protocol version to account for their differences as defined in the specifications. All of these individual tests and `#ifdef` functionality blocks make the function difficult to read in a casual code examination.

An important design note for this function is the calculation of the overall flight size. Once the correct case statement for the state is hit the next thing that happens is the estimated size calculation for the flight. This is the manual process that involves many of the tests mentioned above and each test is done to determine the individual handshake messages that comprise the flight. The message sizes are used to increment the `messageSize` variable. At the end of the additions, if the buffer is not large enough to hold the entire flight the function will return `SSL_FULL` to `matrixSslDecode` so it can grow the buffer and re-invoke. If the connection is in an encrypted write state, the `secureWriteAdditions` function is used to include the encryption overhead to each message. In this case, it is possible to get a false `SSL_FULL` return because the padding for the messages is always estimated as the largest it could be. There is no danger in this, however, as the buffer will simply have to grow and the calculation is performed again.

The individual handshake messages themselves are given a dedicated creation function. For example, `writeServerKeyExchange` and `writeCertificate` are responsible for generating the specific message for the connection. For these calls, the buffer has been converted to an `sslBuf_t` type to easily track the position.

It would be very unusual for the creation of a response flight to fail. If it did, the likely reason would be a memory allocation error. In this error case, the very final test in the function will find this condition and set the alert status to INTERNAL_ERROR and encode that fatal alert.

On success, the function will simply return `MATRIXSSL_SUCCESS` and that is the indication to `matrixSslDecode` that a flight has been queued and to return `SSL_SEND_RESPONSE` to `matrixSslReceivedData`, which will then return `MATRIXSSL_REQUEST_SEND` to the application.

3.5.2 matrixSslDecode

The function matrixSslDecode parses any data coming into the peer. It is called from the public API matrixSslReceivedData. The data may be a handshake record, an alert, or an application data record.

The function will decode a single SSL record at a time. If a handshake flight is being parsed and more than one handshake message exists in the buffer, it is the role of matrixSslReceivedData to advance the buffer and re-invoke matrixSslDecode after each message is parsed. When the final message in a handshake flight is parsed and a response is required, matrixSslDecode will detect this case itself and prepare the response with sslEncodeResponse.

As the function begins an initial check is made to see if the session is in an error state, in which case the function is not eligible to be called. The next test is for whether SSL_FLAGS_NEED_ENCODE is set. This is a goto mechanism to support the SSL_FULL case from sslEncodeResponse. The call stack will have returned to matrixSslReceivedData to grow the buffer and matrixSslDecode will be invoked again. This time the incoming handshake flight decodes will be bypassed.

Next, the SSL record header will be parsed to find the length and type of data being processed. If the length is greater than the number of available bytes in the buffer, the SSL_PARTIAL code will be returned and the user will be sent the MATRIXSSL_REQUEST_RECV code to read more data from the peer. A full record is always required to be available before parsing can begin.

The one exception to the full record rule is the special MatrixSSL USE_CERT_CHAIN_PARSING feature that allows the CERTIFICATE message to be divided on single certificate boundaries. This feature is only meaningful, then, to implementations that are expecting certificate chains.

Now, the record is decrypted and the MAC is checked if the session is in the “secure read” state. If any errors occur while decrypting or verifying the MAC, an alert creation will be triggered.

Next, the record type is tested and passed to the correct handler case.

3.5.2.1 ChangeCipherSpec record

The single byte message size is confirmed and the current handshake state is tested to ensure this message is expected. If so, sslActivateReadCipher will be called to activate the symmetric keys to decrypt further incoming data, the input buffer will be updated, and MATRIXSSL_SUCCESS will be returned.

3.5.2.2 Handshake Record

A handshake record is passed to parseSSLHandshake for processing and that function is detailed in the section below. The return code from parseSSLHandake informs matrixSslDecode on the status of that parse. MATRIXSSL_SUCCESS means the handshake message was a mid-flight message and another handshake message is still expected. SSL_PROCESS_DATA is the indication that the handshake message was the final expected message in the flight and a response is required. The response is driven by a call to sslEncodeResponse as discussed above.

3.5.2.3 Application Data Record

An application data record is essentially just ready to be passed directly to the application. The record will have already been decrypted. The state machine does first test to see if the handshake is complete and the connection is in a valid state to be receiving application data.

3.5.2.4 Alert Record

An alert record is ready to be passed directly to the application. The record will have already been decrypted.

3.5.3 parseSSLHandshake

The `parseSSLHandshake` function performs the specific handshake message parsing and calls the cryptographic operations necessary to perform the SSL key exchange and authentication.

The first task of the function is to read the handshake header to find which message is being parsed. The state machine tests to see if the message is allowed for the current state. This is time during which the variation in handshake types is determined. For example, a client will first be made aware that the server is attempting a client authentication handshake if the CERTIFICATE_REQUEST message is encountered after the CERTIFICATE message. The state flags will be set appropriately as each variation is encountered.

The `hsState` is always tracking the next expected handshake message to be received so the switch statement for that value will match exactly with the message being parsed. Each handshake message is parsed as intelligently as possible, confirming embedded lengths are never larger than available data. If a cryptographic operation is required during the parsing, it is handled in-line immediately in a blocking manner (hardware integration relaxes this blocking restriction but that is beyond the scope of this document).

After each message parse, the `hsState` is updated as required and either `MATRIXSSL_SUCCESS` or `SSL_PROCESS_DATA` is returned as described in the Handshake Record section of the `matrixSslDecode` discussion above.