Rambus

MatrixSSL Core Porting Guide

Rambus

TABLE OF CONTENTS

1 GENERAL CONFIGURATION

This document is the technical reference for porting the Rambus libraries including MatrixSSL and CL libraries to a software platform that isn't supported by default in the product package.

This document is primarily intended for the software developer performing MatrixSSL and CL integration into their custom application.

1.1 Configuring a Development Environment

When beginning a port of MatrixSSL or CL the first thing to decide is what type of binary is being built. The typical options on many platforms are to create a **static library**, a **shared library**, an **executable** or a **binary image**. It is most common on full-featured operating systems to create a MatrixSSL library and use that to later link with the custom executable being created. If multiple applications use MatrixSSL functionality, then a dynamic library is more efficient for disk space; otherwise a static library can actually be smaller when directly linked with an application. On many embedded and real time operating systems, a single binary is compiled with MatrixSSL and all other objects (operating system and application code) into a binary image.

Once the project type is chosen, the next step is to include the MatrixSSL source files. The MatrixSSL library is created from the C source code files that exist in the *core*, *crypto*, and *matrixssl* directories at the top of the package directory structure. The best place to look for the list of files is in the *core/Makefile*, *crypto/Makefile*, and *matrixssl/Makefile*. An application that only uses CL cryptography directly can link with just *core* and *CL*. Similarly, an application that only uses MatrixSSL cryptography directly can link with just *core* and *crypto*.

1.2 Compiler Options and Pre-Processor Defines

1.2.1 Platform Defines

The MatrixSSL source is written in ANSI C and the compiler options for your platform should be set to reflect that if necessary.

The majority of pre-processor defines for MatrixSSL are contained within the configuration headers and are used to enable and disable functionality within the library. These functionality-based defines are discussed in the API and Developer's Guide documentation. By comparison, the defines that are used to specify the hardware platform and operating system should be set using the *-D* compiler flag inside the development environment.

If you choose to follow the platform framework that is implemented in the default package the most important pre-processor define to set is the <*OS*> define that will determine which *osdep.c* file is compiled into the library. As of version 3.9 the default <*OS*> option is *POSIX*. However, if you are reading this document it is likely that your platform is not supported by either of these defaults. For more information on the <*OS*> define and implementing MatrixSSL on an unsupported platform, see the **Implementing core/<OS>** section below.

Processor Architecture

You should confirm the **PSTM_#BIT** define being set in *osdep.h* does match the bit architecture of the platform. For example, set PSTM_32BIT for 32-bit processors or PSTM_64BIT for 64-bit processors.

1.2.2 Performance Defines

There are pre-processor defines for common CPU architectures that will optimize some of the cryptographic algorithms. As of version 3.7, optimized assembly code for x86, ARM and MIPS platforms are included, providing significantly faster performance for public and private key operations. The pre-processor defines are: *PSTM_X86*, *PSTM_X86_64, PSTM_ARM* or *PSTM_MIPS*.

Notes on PSTM_X86 assembly option: Some of the defines below also must be specified in order to compile the code under GCC. Also, the assembly code is written in AT&T UNIX syntax, not Intel syntax. This means it will not compile under Microsoft Visual Studio or the Intel Compiler. Currently, to use the

optimizations on Windows, the files containing assembly code must be compiled with GCC under windows and then linked with the remaining code as compiled by the non-GCC tools. Due to large number of registers required, it the *PSTM_X86* option is compatible with older compiler toolchains. Please, use GCC 4.8 or later compiler if you encounter problems with the *PSTM_X86* option (or disable the option). The *PSTM_X86* option may conflict with other compilation options that consume additional registers like *–fPIC* or *–fno-omit-frame-pointer*.

There are several compiler options that also affect size and speed on various platforms. The options below are given for the GCC compiler. Other compilers should support similar types of optimizations.

GCC Flag	Notes
-Os	By default, MatrixSSL uses this optimization, which is a good balance between performance and speed. Because embedded devices are often constrained by RAM, FLASH and CPU cache sizes, often optimizing for size produces faster code than optimizing for speed with -O3
-g	Because MatrixSSL is provided as source code, it can be compiled with debug flags (-g) rather than optimization flags.
-fomit-frame-pointer	This option allows one additional register to be used by application code, allowing the compiler to generate faster code. It is required on X86 platforms when using MatrixSSL assembly optimizations because the assembly code is written to take advantage of this register.
-mdynamic-no-pic	This option can also allow an additional register to be useable by application code. On the Mac OS X platform, it is required to compile with assembly language optimizations.
-ffunction-sections	This option effectively allows the linker to treat each function as its own object when statically linking. This means that only functions that are actually called are linked in. MatrixSSL is already divided into objects optimally, but this option may help overall when producing a binary of many different objects.
-fdata-sections	This option effectively allows the linker to treat each data symbol as its own object when statically linking. This means that only the data that is actually referenced will be linked in. MatrixSSL is already divided into objects optimally, but this option may help overall when producing a binary of many different objects.
-fpic or –fPIC	Build position independent code. One of these flag is recommended or needed on most targets when intending to use the components as a *shared library*. However, typically *static libraries* do not require position independent code.

2 SOURCE CODE FRAMEWORK

The primary effort of a MatrixSSL or CL port is to map headers files and/or implement a small set of specific functionalities required by the inner workings of the library. This section defines all the platform-specific requirements the developer must implement.

The location of core directory varies depending on package. When MatrixSSL is packaged as a standalone product, the core resides alongside crypto and matrixssl.

Rambus has attempted to make the porting process as straightforward as possible. There is any number of ways the developer can organize the source files to include these specific requirements, but it is strongly encouraged that the default framework be used for purposes of support and modularization.

The default design framework will only require the user to work with two source code files:

File	Description
core/config	Header files describing configuration and platform dependent data types.
core/include	Header files for core module. These files are used by CL and MatrixSSL applications to interact with the core module, but the files typically need no porting.
core/osdep/include	Mapping between platform specific headers and headers required by MatrixSSL and/or CL.
core/<OS>/	Platform dependent part of port of core for a target system.
core/test core/apps	Test applications using subset of core's services. These programs can be used to try port of core on a target system (if the target provides support for C applications launched from command prompt).

The <OS> designation will be defined by the developer and should be a brief capitalized description of the operating system being ported to. This value should then be set as a preprocessor define so that the correct data types are pulled from *core/osdep.h* and that the *core/<OS>/osdep.c* file is compiled into the library. As of version 3.9 the currently supported defines are **POSIX** and **WIN32** and can be used as references during the porting process.

3 STANDARD LIBRARY DEPENDENCIES

MatrixSSL and CL rely on a small set of standard library calls that the platform must provide. The functions in this list should typically be provided in the standard C libraries, such as libc, newlib and uClibc, but if not, you will need to implement them.

These dependencies are used via a header wrapping mechanism.

The **Implementation Requirements** descriptions for these routines are taken directly from the BSD Library Functions Manual.

3.1 Data Types and Structures

The following table describes the set of data types the user must define for the new platform. These should be added to the existing *core/osdep/include/osdep-types.h* file and wrapped in #ifdef <OS> blocks.

Data Type	Definition	Comments
int32	A 32-bit integer type	Always required
uint32	A 32-bit unsigned integer type	Always required
int16	A 16-bit integer type	Always required
uint16	A 16-bit unsigned integer type	Always required
int64	A 64-bit integer type	Only required if HAVE_NATIVE_INT64 is defined
uint64	A 64-bit unsigned integer type	Only required if HAVE_NATIVE_INT64 is defined
psTime_t	A data type to store a time counter for the platform	Always required. See the Time Functions section below for details. Allows for a higher resolution and length than a single 32 bit value, if desired.
psMutex_t	A data type to support a lockable mutex.	Only required if USE_MULTITHREADING is defined. See the Multithreading section below for details.

3.2 Platform header file wrapping

ISO C99 and POSIX standards define operating environments to provide various header files, such as *stdio.h*, *stdint.h*, *string.h* and *unistd.h*. These header files are commonly available on target systems conforming with these specifications. It is common that these header files exist also on many targets that are not fully conformant with the standard.

To deal with platform specific differences core includes wrapper headers.

When MatrixSSL or CL component needs one of these headers, instead of include the header directly, it will include *core/osdep/include/osdep_<header_base_name>.h*, for instance *core/osdep/include/osdep_stdlib.h*. These wrapper includes allow one layer of indirection. In case the target system does not have *stdlib.h* header or includes it in a non-standard place, *core/osdep/include/osdep_stdlib.h* can be adjusted (for instance if a system provides only memory allocation centered *malloc.h* instead of *stdlib.h*).

The wrappers also wrap function names used by MatrixSSL or CL. When intended function call is memory allocation, MatrixSSL or CL will call macro `Malloc`, which will invoke `malloc` standard library function. This is to allow dealing with small differences in symbol name of `malloc`.

3.2.1 Porting osdep_stdlib.h

As an example of header file porting, I present example: porting of *stdlib.h*. *stdlib.h* provides dynamic memory allocation, and access to environment variables and fatal failure.

If target system provides *malloc.h* instead of *stdlib.h*, please alter *core/osdep/include/osdep_stdlib.h* to **#include <malloc.h>** instead of **#include <stdlib.h>**. In case neither of these files are found, replace with **#include** statements that provide target specific dynamic memory allocation library.

Let us say target provided functions `xxMalloc`, `xxFree`, `xxCalloc` and `xxRealloc`, with API equivalent to ISO C89 or ISO C99 `malloc` API family. Then just adjust macros to map these to memory allocation and free.

#define Malloc xxMalloc

#define Free xxFree

#define Calloc xxCalloc

#define Realloc xxRealloc

In addition, `Getenv` and `Abort` functions need to be dealt with. In case the operating environment does not provide environment variables, it is recommendable to return always **NULL** from `Getenv`, for instance: **#define Getenv(string) (NULL)**

Some parts of MatrixSSL or CL may call `Abort()` upon critical error to step execution of the program. If the operating system does not such facility or do not wish to handle critical failure, define the macro as follows:

#define Abort() do { } while(0)

3.3 Memory Allocation

The memory allocation operations are wrapped by *core/osdep/include/osdep_stdlib.h*.

3.3.1 Malloc

```
void *Malloc(size_t size);
```

Implementation Requirements

The `Malloc` function allocates size bytes of memory and returns a pointer to the allocated memory.

3.3.2 Realloc

```
void *Realloc(void *ptr, size_t size);
```

Implementation Requirements

The `Realloc` function tries to change the size of the allocation pointed to by `ptr` to `size`, and returns `ptr`. If there is not enough room to enlarge the memory allocation pointed to by `ptr`, `Realloc` creates a new allocation, copies as much of the old data pointed to by `ptr` as will fit to the new allocation, frees the old allocation, and returns a pointer to the allocated memory. If `ptr` is NULL, `Realloc` is identical to a call to `Malloc` for `size` bytes. If `size` is zero and `ptr` is not NULL, a new, minimum sized object is allocated and the original object is freed.

3.3.3 Free

```
void Free(void *ptr);
```

Implementation Requirements

The `Free` function deallocates the memory allocation pointed to by ptr.

3.4 Memory Operations

The memory operations are wrapped by *core/osdep/include/osdep_string.h*.

3.4.1 Memcmp

```
int Memcmp(const void *s1, const void *s2, size_t n);
```

Implementation Requirements

The `Memcmp` function compares byte string `s1` against byte string `s2`. Both strings are assumed to be `n` bytes long. The `Memcmp` function returns 0 if the two strings are identical, otherwise returns the difference between the first two differing bytes (treated as unsigned char values, so that `\200' is greater than `\0', for example). Zero-length strings are always identical.

3.4.2 Memcpy

```
void *Memcpy(void *s1, void *s2, size_t n);
```

Implementation Requirements

The `Memcpy` function copies `n` bytes from memory area `s2` to memory area `s1`. If `s1` and `s2` overlap, behavior is undefined. Applications in which `s1` and `s2` might overlap should use `Memmove` instead. The `Memcpy` function returns the original value of `s1`.

3.4.3 Memset

```
void *Memset(void *s, int c, size_t n);
```

Implementation Requirements

The `Memset` function writes `n` bytes of value `c` (converted to an unsigned char) to the string `s`. The `Memset` function returns its first argument.

3.4.4 Memmove

```
void *Memmove(void *s1, void *s2, size_t n);
```

Implementation Requirements

The `Memmove` function copies `n` bytes from string `s2` to string `s1`. The two strings may overlap; the copy is always done in a non-destructive manner. The `Memmove` function returns the original value of `s1`.

3.4.5 Strstr

```
char *Strstr(const char *s1, const char *s2);
```

Implementation Requirements

The `Strstr` function locates the first occurrence of the null-terminated string `s2` in the null-terminated string `s1`. If `s2` is an empty string, `s1` is returned; if `s2` occurs nowhere in `s1`, NULL is returned; otherwise a pointer to the first character of the first occurrence of `s2` is returned.

3.4.6 Strlen

```
size_t Strlen(const char *s);
```

Implementation Requirements

The `Strlen` function computes the length of the string `s`. The `Strlen` function returns the number of characters that precede the terminating NULL character.

3.5 Multithreading and fork()

In most cases, a single threaded, non-blocking event loop is more efficient for handling multiple socket connections. This is evidenced by the architecture of high performance web servers such as nginx and lightHttpd. As such, MatrixSSL does not contain any locking for individual SSL connections.

If threading is present in an application using MatrixSSL, a few guidelines should be followed:

- It is highly recommended that any given SSL session be associated only with a single thread. This means multiple threads should never share access to a single `ssl_t` structure, or its associated socket connection. Theoretically, the connection may be handled by a thread and then passed on to another without simultaneous access, but this complexity isn't recommended. It is also possible to add a mutex lock around each access of the `ssl_t` structure, associated socket, etc., however ensuring that parsing and writing of records is properly interleaved between threads is difficult.
- The only SSL protocol resource shared between sessions in MatrixSSL is the session cache on server side SSL connections. This is a performance optimization that allows clients that reconnect to bypass CPU intensive public key operations for a period of time. MatrixSSL does internally define and lock a mutex to keep this cache consistent if multiple threads access it at the same time. This code is enabled with the `USE_MULTITHREADING` define.
- User implementations of entropy gathering, filesystem and time access may internally require mutex locks for consistency, which is beyond the scope of this document.

Applications using `fork()` to handle new connections are common on Unix based platforms. Because the MatrixSSL session cache is located in the process data space, a forked process will not be able to update the master session cache, thereby preventing future sessions from being able to take advantage of this speed improvement. In order to support session resumption in forked servers, a shared memory or file based session cache must be implemented.

The mutex implementation is wrapped within the `USE_MULTITHREADING` define in *core/coreConfig.h* and the platform-specific implementation should be included in the *core/<OS>/osdep.c* file.

3.5.1 osdepMutexOpen

```
int osdepMutexOpen(void);
```

Return Value	Description
PS_SUCCESS	Successful initialization
PS_FAILURE	Failed mutex module initialization

Servers

This is the one-time initialization function for the platform specific mutex support. This function must always exist even if there is no operation to perform. This function is internally invoked by `matrixSslOpen`.

Memory Profile

This function may internally allocate memory that can be freed during `osdepMutexClose`

3.5.2 osdepMutexClose

```
int osdepMutexClose(void);
```

Servers

This function performs the one-time final cleanup for the platform specific mutex support. This function must always exist even if there is no operation to perform. This function is internally invoked by matrixSslClose.

Return Value	Description
PS_SUCCESS	Success
PS_FAILURE	Failure

3.5.3 psCreateMutex

```
int32 psCreateMutex(psMutex_t *mutex);
```

Parameter	Input/Output	Description
mutex	input/output	An allocated `psMutex_t` structure to initialize for future calls to `psLockMutex`, `psUnlockMutex`, and `psDestroyMutex`

Return Value	Description
PS_SUCCESS	Success. The mutex has been created
PS_FAILURE	Failure

Server Usage

The server uses this function to create the `sessionTableLock` during application initialization.

3.5.4 psLockMutex

```
int32 psLockMutex(psMutex_t *mutex);
```

Parameter	Input/Output	Description
mutex	input	Mutex to lock

Return Value	Description
PS_SUCCESS	Success. The mutex has been locked
PS_FAILURE	Failure

Server Usage

The server uses this function to lock the `sessionTableLock` each time the session cache table is being modified.

3.5.5 psUnlockMutex

```
int32 psUnlockMutex(psMutex_t *mutex);
```

Parameter	Input/Output	Description
mutex	input	Mutex to unlock

Return Value	Description
PS_SUCCESS	Success. The mutex has been locked
PS_FAILURE	Failure

Server Usage

The server uses this function to unlock the `sessionTableLock` each time the session cache table is done being modified.

3.5.6 psDestroyMutex

```
void psDestroyMutex(psMutex_t *mutex);
```

Parameter	Input/Output	Description
mutex	input	Mutex to destroy

Server Usage

The server uses this function to destroy the `sessionTableLock` mutex during application shutdown.

3.6 Client and Server Socket-Based Applications

The application examples shipped with MatrixSSL require additional headers and functions. These dependencies are handled via the same header and function call wrapping mechanism as MatrixSSL library dependencies. This is for user convenience: the porting effort done for the base libraries will expedite also porting of the simple applications.

If directly porting the BSD sockets-based client and server applications that are provided in the apps directory, there is an additional set of functions that must be available on the platform. Below is an alphabetical list of the functions with an ✓ indicating that is it needed by that application. If porting to non-

BSD sockets applications, it may be easier to partially rewrite the examples with the native transport API than to try to implement the APIs 1:1 below.

Macro (mapping to BSD function)	Client	Server
Accept		✓
Bind		✓
Close	✓	✓
Connect	✓	
Exit		✓
Fcntl	✓	✓
Inet_addr	✓	
Listen		✓
Puts		✓
Recv	✓	✓
Select		✓
Send	✓	✓
Setsockopt		✓
Signal		✓
Socket	✓	✓
Strncpy	✓	✓

4 PLATFORM ADAPTION LIBRARY

The core provides a platform adaption library, which is located at *core/osdep/<OS>* directory.

When porting to operating system that is partially POSIX compliant, it is typically sufficient to slightly adjust header file abstraction as explained in section 3.2 Platform header file wrapping. However, when porting for environments further away from POSIX, platform adaption library will also require changes.

Platform adaption library includes various functionality that requires access to time or file system.

4.1 Time Functions

These functions should be implemented in *core/<OS>/osdep.c*

The implementation allows for an arbitrary internal time structure to be used (for example a 64 bit counter), while providing the ability to get a delta between times and a time-based monotonically increasing value, both as 32 bit signed integers.

4.1.1 osdepTimeOpen

```
int osdepTimeOpen(void);
```

Return Value	Description
PS_SUCCESS	Successful initialization
PS_FAILURE	Failure.

Servers and Clients

This is the one-time initialization function for the platform specific time support. For example, it may initialize a high-resolution timer, calibrate the system time, etc. This function must always exist even if there is no operation to perform. This function is internally invoked from `matrixSslOpen`.

Memory Profile

This function may internally allocate memory that can be freed during `osdepTimeClose`

4.1.2 osdepTimeClose

```
void osdepTimeClose(void);
```

Servers and Clients

This function performs the one-time final clean-up for the platform specific time support. This function must always exist even if there is no operation to perform. This function is internally invoked from `matrixSslClose`.

Memory Profile

This function should free any memory that was allocated during `osdepTimeOpen`

4.1.3 psGetTime

```
int32 psGetTime(psTime_t *currentTime, void *userPtr);
```

Parameter	Input/Output	Description
currentTime	output	Platform-specific time format representing the current time (or ticks)
userPtr	input	An opaque pointer that may be used to pass context information to the routine. When called from MatrixSSL the `userPtr` will be the `ssl->userPtr` member of the `ssl_t` session context structure.

Return Value	Description
> 0	A platform-specific time measurement (each subsequent call to psGetTime must return an ever-increasing value.)
<= 0	Error retrieving time

Implementation Requirements

This routine must be able to perform two tasks:

1. This function MUST return a platform-specific time measurement in the return parameter. This value SHOULD be the GMT UNIX Time, which is the number of elapsed seconds since January 1st, 1970 GMT. If it is not possible to return the GMT UNIX Time the function MAY return a platform-specific counter value such as CPU ticks or seconds since platform boot. Ideally, if using CPU time, the current count will be stored in non-volatile memory each power down, so that it may be loaded again at startup, and the value returned by this function will continue to increase between any number of power cycles. The SSL and TLS protocols use this 32 bit signed value as part of the prevention of replay message attacks.

2. This function must populate a platform specific static time structure if it is provided in the `currentTime` parameter. The contents of the `psTime_t` structure must contain the information necessary to compute the difference in milliseconds between two `psTime_t` values. If the platform cannot provide a millisecond resolution time, it is fine to scale up from the most accurate source available. For example, if the clock can only return values in 1 second granularity, that value can simply be multiplied by 1000 and returned, when requested. The currently supported `psTime_t` structures for POSIX and WIN32 are defined in *./core/osdep.h* and it is recommended additional <OS> versions are included there as well.

Servers and Clients Usage

Clients and Servers both use this function as described in **Implementation Requirement 1** above. This routine is called when the CLIENT_HELLO and SERVER_HELLO handshake messages are being created. The SSL/TLS specifications require that the first 4 bytes of the Random value for these messages be the GMT UNIX Time, which is the number of elapsed seconds since January 1st, 1970 GMT. Many embedded platforms do not maintain the true calendar date and time so it is acceptable for these platforms to simply return a counter value such as 'ticks' since power on, or 'CPU lifetime ticks'. Also it is acceptable that UNIX Time will overflow 32 bits in 2038. Ideally, this value is designed to provide a "forever increasing" value for each SSL message across multiple SSL sessions and CPU power cycles.

Server Usage

Servers also use this function as described in **Implementation Requirement 2** above. Servers must manage an internal session table in which entries can expire or be compared for staleness against other

entries. The psGetTime function is used to store the current time for these table entries for later comparison using psDiffMsecs and psCompareTime.

The userPtr

When this function is called from within the MatrixSSL library, the userPtr input parameter will be set to the ssl->userPtr from the ssl_t session structure. It is possible the psGetTime implementation does not require any context and the parameter may be safely ignored.

Memory Profile

This implementation requires that the psTime_t structure only contain static members. Implementations of psGetTime must not allocate memory that is not freed before the function returns.

4.1.4 psDiffMsecs

```
int32 psDiffMsecs(psTime_t then, psTime_t now, void *userPtr);
```

Parameter	Input/Output	Description
then	input	Time structure from a previous call to psGetTime
now	input	Time structure from a previous call to psGetTime
userPtr	input	An opaque pointer that may be used to pass context information to the routine. When called from MatrixSSL the userPtr will be the ssl->userPtr member of the ssl_t session context structure.

Return Value	Description
> 0	Success. The difference in milliseconds between then and now
<= 0	Error computing the difference in time

Implementation Requirements

This routine must be able to return the difference, in milliseconds, between two given time structures as a signed 32-bit integer. The value will overflow if then differs from now by more than 24 days.

Server Usage

Servers are the only users of this function. Servers manage an internal session cache table for protocol optimization in which entries can expire or be compared for staleness against other entries. The psGetTime function is used to store the current time for these table entries for later comparison using psDiffMsecs and psCompareTime.

The userPtr

When this function is called from within the MatrixSSL library, the userPtr input parameter will be set to the ssl->userPtr from the ssl_t session structure. It is possible the psGetTime implementation does not require any context and the parameter may be safely ignored.

Memory Profile

This implementation requires that the psTime_t structure can only contain only static members. Implementations of psGetTime must not allocate memory that isn't freed before the function returns.

4.2 Random Number Generation Functions

A source of pseudo-random bytes is an important component in the SSL security protocol. These functions must be implemented in *core/<OS>/osdep.c*

The generation of truly random bytes of data is critical to the security of SSL, TLS and any of the underlying algorithms. There are two components to providing truly random data for cryptography.

1. Collecting Entropy (random data from external events):
   - User interaction such as the low bit of the time between key presses and clicks, mouse movement direction, etc.
   - Operating system state, such as network packet timing, USB timing, memory layout, etc.
   - Hardware state, such as the variation of pixels on a webcam, the static on a radio tuner card, etc.

2. **Pseudo Random Number Generation (PRNG)** is a step that combines (scrambles) the entropy input into bytes of data suitable for use in crypto applications. For example, the Yarrow PRNG is an algorithm that takes random data as input and processes it using a symmetric cipher (AES) and a one-way hash (SHA-256). An application developer can request these processed bytes using a second API.

Desktop and Server operating systems typically implement both the collection of entropy and PRNG, and provide a method for reading random bytes from the OS. For example, LINUX and BSD based operating system provide */dev/random* and/or */dev/urandom*, and Windows has `CryptGenRandom` and related APIs.

On embedded platforms, MatrixSSL can provide a PRNG algorithm (Yarrow) suitable for a small footprint application, however the first requirement of collecting entropy is more difficult and very platform specific. Looking at the points above, embedded hardware often has very limited user interaction, very limited time variation on operating system events (close to zero on an RTOS) and very limited hardware peripherals from which to draw. Entropy can be gathered from some timing measurements, and high quality entropy can be gathered if the processor can sample from ADC or a free-running oscillator on the hardware platform. Please contact Rambus for guidance on gathering entropy for a specific hardware platform.

4.2.1 osdepEntropyOpen

```
int osdepEntropyOpen(void);
```

Return Value	Description
PS_SUCCESS	Successful initialization
PS_FAILURE	Failure.

Servers and Clients

This is the one-time initialization function for the platform specific PRNG support. This function must always exist even if there is no operation to perform. This function is internally invoked from `matrixSslOpen`.

Memory Profile

This function may internally allocate memory that can be freed during `osdepEntropyClose`

4.2.2 osdepEntropyClose

```
void osdepEntropyClose(void);
```

Servers and Clients

This function performs the one-time final clean-up for the platform specific entropy and PRNG support. This function is internally invoked from `matrixSslClose`.

Memory Profile

This function should free any memory that was allocated during `osdepEntropyOpen`

4.2.3 psGetEntropy

```
int32 psGetEntropy(unsigned char *bytes, uint32 size, void *userPtr);
```

Parameter	Input/Output	Description
Bytes	input/output	Random bytes must be copied to this buffer
Size	input	The number of random bytes the caller is requesting
userPtr	input	An opaque pointer that may be used to pass context information to the routine. When called from MatrixSSL the `userPtr` will be the `ssl->userPtr` member of the `ssl_t` session context structure. There is one exception to this and is discussed below.

Return Value	Description
> 0	Success. The number of random bytes copied to `bytes`. Should be the same value as `size`
PS_FAILURE	Failure. Error generating random bytes

Implementation Requirements

This routine must be able to provide an indefinite quantity of random data. This function is internally invoked in several areas of the MatrixSSL code base. Please contact Rambus for guidance in implementing entropy gathering.

Servers and Clients

There are various places in which random data is needed within the SSL protocol for both clients and servers.

The Relationship Between PRNG and Entropy in MatrixSSL

MatrixSSL does not call `psGetEntropy` directly. Random data is gathered through the crypto `psGetPrng` function, which optionally wraps a PRNG algorithm around the entropy gathering. A PRNG uses hash and encryption algorithms to output a stream of random bytes rather than going directly to the entropy source for each required random byte. Occasionally, the PRNG will gather additional entropy using `psGetEntropy` to scramble the algorithm. In the case of no PRNG wrapper, the `psGetEntropy` function will be called each time `psGetPrng` is called.

A single global PRNG instance is created during the call to `matrixSslOpen` with the function `psInitPrng`. To support this single instance in the case of `USE_MULTITHREADING` a lockable mutex is used to wrap all calls to `psGetPrng`.

The userPtr and MatrixSSL

In all but one case, when `psGetEntropy` is invoked from within MatrixSSL the `ssl->userPtr` member will be passed as the `userPtr` to this routine. This is the one exception to this rule.

- Servers and Clients - If a PRNG algorithm is enabled within the `psInitPrng` mechanism the algorithm will require an entropy seed value that is fetched using `psGetEntropy`. However, when `psInitPrng` is invoked during `matrixSslOpen` there is no SSL session context so the `userPtr` value will be `NULL` in this case.

NOTE: For clients, if the `userPtr` context is important for `psGetEntropy` the `sslSessOpts_t` member `userPtr` must be populated when passing `options` to `matrixSslNewClientSession` because the creation of CLIENT_HELLO will require entropy gathering.

Memory Profile

This API may adjust its internal state or storage size of collected entropy data.

4.3 File Access Functions

These functions can optionally be implemented in *core/<OS>/osdep.c*. They are only required if `MATRIX_USE_FILE_SYSTEM` is defined in the platform build environment.

4.3.1 psGetFileBuf

```
int32 psGetFileBuf(psPool_t *pool, const char *filename,
      unsigned char **buf, int32 *bufLen);
```

Parameter	Input/Output	Description
pool	input	The memory pool if using Matrix Deterministic Memory (`USE_MATRIX_MEMORY_MANGEMENT` is enabled)
filename	input	The filename (with directory path) of the file to open
buf	output	The contents of the filename in an internally allocated memory buffer
bufLen	output	Length, in bytes, of `buf`

Return Value	Description
PS_SUCCESS	Success. The file contents are in the memory buffer at `buf`
< 0	Failure.

Implementation Requirements

This routine must be able to open a given file and copy the contents into a memory location that is returned to the caller. The memory location must be allocated from within the function and if a `pool` parameter is passed in, the function must use `psMalloc` for the memory allocation.

Server and Client Usage

Reading files from disk will only be necessary if `matrixSslLoadRsaKeys`, `matrixSslLoadEcKeys`, or `matrixSslLoadDhParams` is used during initialization.

Memory Profile

Internal library usage of `psGetFileBuf` will free the allocated `buf` using `psFree` after the useful life. If using `psGetFileBuf` in a custom application `psFree` will need to be called manually.

4.4 Trace and Debug Functions

The logging facilities are invoked in MatrixSSL and SL with various macros including debugf, tracef, psTrace, psTraceInt, and psTraceStr. These macros map to a logging facilities. This section explains how to control what logging features are enabled during build and how to write backend functions that direct the log messages to their intended target.

4.4.1 Trace and Debug Functions Source Code

The *psLog.h* is implemented by *core/osdep/POSIX/psLog.c*. The default implementation requires formatting facilities available within ISO C99 standard *stdio.h*, and in case of multithreading, it will use a locks to prevent multiple threads from writing the same log file at once.

4.4.2 Trace and Debug Functions Build Time Configuration

The *psLog.h* provides the logging API in MatrixSSL and CL. It is intended to be generic and extensible and to allow hooking. The API depends on formatted printing capabilities (*stdio.h*).

Build time options for *psLog.h* can be chosen within *coreConfig.h*.

Define	Purpose	Comments
PS_NO_LOGF	Substitute psLogf functionality with embedded API (see 4.4.4).	Not recommended unless mandatory for embedded footprint.
PS_NO_LOGF_FATAL	Omit code for fatal messages.	Omitting message may decrease binary size, but may cause important diagnostics to be missed.
PS_NO_LOGF_ERROR	Omit code for error messages.	"
PS_NO_LOGF_WARNING	Omit code for warning messages.	"
PS_NO_LOGF_INFO	Omit code for informational messages.	"
PS_NO_LOGF_VERBOSE	Omit code for verbose informational messages.	"
PS_NO_LOGF_DEBUG	Omit code for debug messages.	"
PS_NO_LOGF_TRACE	Omit code for debug trace messages.	These are omitted by default when building non-debug builds.
PS_NO_LOGF_CALL_TRACE	Omit code for call trace messages.	These are omitted by default when building non-debug builds.
PS_NO_LOGF_PRINT_FILELINE	Omit file and line information.	These are omitted by default when building non-debug builds.
PS_NO_LOGF_PRINT_UNIT	Omit unit from log messages.	Not omitted by default.

4.4.3 Trace and Debug Functions Dynamic Configuration

The tracing facilities support functions that control which of the compiled in messages will be actually shown to the user (or developer) and how to display them.

4.4.3.1 user_psLogfSetHookEnabledCheckFunction

```
int user_psLogfSetHookEnabledCheckFunction(const char *level, const char
*unit);
```

Parameter	Input/Output	Description
level	input	A level of message. Currently pointer to one of `psLogf_Log_Fatal` (`"Log_Fatal"`), `psLogf_Log_Error` (`"Log_Error"`), `psLogf_Log_Warning` (`"Log_Warning"`), `psLogf_Log_Info` (`"Log_Info"`), `psLogf_Log_Debug` (`"Log_Debug"`), `psLogf_Log_Verbose` (`"Log_Verbose"`), `psLogf_Log_Trace` (`"Log_Trace"`) or `psLogf_Log_CallTrace` (`"Log_CallTrace"`).
unit	input	A string variable representing code unit where log message was generated. This is e.g. `"PS_MATRIXSSL_HS"` for handshake trace.

Return Value	Description
0	Inhibit log messages with this level and unit.
1	Allow log messages with this level and unit.
-1	Standard environment variables based processing. (See below.)

Implementation Requirements

This routine should consider based on level and/or unit if specific log message will be inhibited or allowed to be processed. Depending on configuration there can be very large amount of log messages, and for this reason it is recommended for the function to be fast. For comparison of log level, the function may use pointers to the strings `psLogf_Log_Fatal` etc. to determine log level quickly without string comparison.

The logging hook function needs to be installed with `psLogfSetHookEnabledCheck` function. There can be multiple hooks implemented, but just one can be active at one specific time.

4.4.3.1.1 Standard processing

When message filtering hooks are not available, all the logging messages are omitted by default.

However, environment variable *PS_ENABLE_LOG* and *PS_DISABLE_LOG* can be used to set the global default logging status. Each logging level and unit may be controlled via their individual toggle, such as *PS_ENABLE_LOG_DEBUG* or *PS_ENABLE_UNIT*. Messages omitted by build configuration cannot be enabled without rebuilding the software.

4.4.3.2 User_psLogfSetHookPrintfFunction

```
int user_psLogfSetHookPrintfFunction(const char *level, const char *unit,
                                     const char *format_string, va_list va);
```

Parameter	Input/Output	Description
Level	input	A level of message. Currently pointer to one of `psLogf_Log_Fatal` (`"Log_Fatal"`), `psLogf_Log_Error` (`"Log_Error"`), `psLogf_Log_Warning` (`"Log_Warning"`), `psLogf_Log_Info` (`"Log_Info"`), `psLogf_Log_Debug` (`"Log_Debug"`), `psLogf_Log_Verbose` (`"Log_Verbose"`), `psLogf_Log_Trace` (`"Log_Trace"`) or `psLogf_Log_CallTrace` (`"Log_CallTrace"`).
Unit	input	A string variable representing code unit where log message was generated. This is e.g. `"PS_MATRIXSSL_HS"` for handshake trace.
format_string	format string	Formatting string using C99 *printf* format string syntax.
Va	arguments	Variable arguments list, for processing with vprintf function family, including vprintf and vsprintf.

Return Value	Description
1	Message successfully printed.
0	Printing omitted (e.g. due to an error).
-1	Standard environment variables based processing. (See below.)

Implementation Requirements

This routine should format the string using formatting function that accepts variable arguments as a va_list, such as `vprintf` (print to standard output), `vfprintf` (print to file) or `vsnprintf` (print to string buffer). Some of logging messages can be very long, so when outputting to buffer, the buffer either needs to be dynamically allocated to fit the message or messages need to be truncated to buffer size.

The function may use logging level and unit to determine where to log the message (if not all).

Typical logging targets include console, logging service such as android log or logd, serial port.

In case `vprintf` formatting functions are not available on the target platform, `sfzcl_vsnprintf` function from the core library can be used as a substitute.

The log filtering function set by `psLogfSetHookEnabledCheck` function is executed before this function.

The logging hook function needs to be installed with `psLogfSetHookPrintf` function. There can be multiple hooks implemented, but just one can be active at one specific time.

4.4.3.2.1 Standard processing

When printing hooks are not available, the logging messages are printed to standard output. However, environment variables *PS_LOG_FILE* or *PS_LOG_FILE_APPEND* can be used to override the output file. (*PS_LOG_FILE* writes to the specified file; *PS_LOG_FILE_APPEND* will append to the specified file.)

4.4.4 Trace and Debug Functions Embedded API

WARNING: These functions are not used unless `PS_NO_LOGF` is defined.

The `_psTrace` set of APIs are the low level platform-specific trace routines that are used by `psTraceCore` (`USE_CORE_TRACE`), `psTraceCrypto` (`USE_CRYPTO_TRACE`), `psTraceInfo` (`USE_SSL_INFORMATIONAL_TRACE`), and `psTraceHs` (`USE_SSL_HANDSHAKE_MSG_TRACE`). These functions should be implemented in *core/<OS>/osdep.c*, and can be stubbed out if trace is not required.

Note: This streamlined, but limited API is only used of *PS_NO_LOGF* is defined in *coreConfig.h*.

The most significant limitations compared to *psLog.h* are that this API is only configured statically.

4.4.4.1 osdepTraceOpen

```
int osdepTraceOpen(void);
```

Return Value	Description
PS_SUCCESS	Successful initialization
PS_FAILURE	Failed trace module initialization

Servers and Clients

This is the one-time initialization function for the platform specific trace support. This function must always exist even if there is no operation to perform. This function is internally invoked by `matrixSslOpen`.

Memory Profile

This function may internally allocate memory that can be freed during `osdepTraceClose`

4.4.4.2 osdepTraceClose

```
void osdepTraceClose(void);
```

Servers and Clients

This function performs the one-time final cleanup for the platform specific trace support. This function must always exist even if there is no operation to perform. This function is internally invoked by matrixSslClose.

Memory Profile

This function must free any memory that was allocated during osdepTraceOpen

4.4.4.3 _psTraceStr

```
void _psTraceStr(char *message, char *value);
```

Parameter	Input/Output	Description
message	input	A string message containing a single `%s` format character that will be output as debug trace
Value	input	A string variable value that will be substituted for the `%s` in the `message` parameter

Implementation Requirements

This routine should substitute the `value` string for `%s` in the `message` parameter and output the result to the standard debug output location.

4.4.4.4 _psTraceInt

```
void _psTraceInt(char *message, char *value);
```

Parameter	Input/Output	Description
message	input	A string message containing a single `%d` format character that will be output as debug trace
value	input	A integer variable value that will be substituted for the `%d` in the `message` parameter

Implementation Requirements

This routine should substitute the `value` integer for `%d` in the `message` parameter and output the result to the standard debug output location.

4.4.4.5 _psTracePtr

```
void _psTraceInt(char *message, char *value);
```

Parameter	Input/Output	Description
message	input	A string message containing a single `%p` format character that will be output as debug trace
value	input	A memory pointer variable value that will be substituted for the `%p` in the `message` parameter

Implementation Requirements

This routine should substitute the `value` integer for `%p` in the `message` parameter and output the result to the standard debug output location.

4.4.4.6 osdepBreak

```
void osdepBreak(void);
```

Implementation Requirements

This routine should be a platform-specific call to halt program execution in a debug environment. This function is invoked as part of the `_psError` set of APIs if `HALT_ON_PS_ERROR` is enabled to aid in source code debugging. There is a small set of `_psError` calls inside the library but the intention is that the user adds them to the source code to help narrow down run-time problems.

5 ADDITIONAL TOPICS

5.1 64-Bit Integer Support

If your platform supports 64-bit integer types (`long long`) you should make sure `HAVE_NATIVE_INT64` is enabled in *core/osdep.h* so that native 64-bit math operations can be used. If used, your platform may also require the `udivdi3` function. If this symbol (or other 64-bit related functions) is not available, you can optionally disable `HAVE_NATIVE_INT64` to produce a slower performance library.

64-bit integer math support is not the same as running in '64 bit addressing mode' for the operating system. Many 32 bit processors do support multiplying two 32-bit numbers for a 64-bit result, and can enable this define.