MatrixSSL 3.1 OS Porting Guide

Overview

This document is the technical reference for porting the MatrixSSL C code library to a platform that isn't supported by default in the product package.

Who Is This Document For?

- Software developers that are porting the MatrixSSL library to a new platform
- Software developers who want to understand the platform-specific functionality of MatrixSSL

Documentation Style Conventions

- File names and directory paths are *italicized*.
- C code literals are distinguished with the `Monaco font`.

Commercial Version Differences

Some of the compile options and discussions in this document are specific to the commercially licensed version of MatrixSSL. Sections of this document that refer to the commercial version will be shaded.

Configuring a Development Environment

This section is a general overview on how to organize and compile the MatrixSSL source code package when configuring a development environment for a new platform.

Project Type

When beginning a port of MatrixSSL 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 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 best place to look for the list of files is in the *Makefile* that is included in the top level directory of the package. The *Makefile* separates the source files into *core*, *crypto*, and *matrixssl* lists for functional clarity but all the files should be included in a project that uses the SSL or TLS protocols. An application that only uses MatrixSSL cryptography directly can link with just *core* and *crypto* source.

Compiler Options and Preprocessor Defines

Platform Define

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 preprocessor 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 preprocessor define to set is the <OS> define that will determine which *osdep.c* file is compiled into the library. As of version 3.1 the two default <OS> options are *POSIX* and *WIN32*. 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.

Performance Defines

For commercial versions of MatrixSSL, there are preprocessor defines for common CPU architectures that will optimize some of the cryptographic algorithms. As of version 3.1, optimized assembly code for x86, ARM and MIPS platforms is included, providing significantly faster performance for public and private key operations. The preprocessor defines are: PSTM_X86, 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.

Notes on Mac OS X assembly: Due to the registers required and the build options available under OS X, it is not currently possible to build a dynamic library with assembly optimizations. If a dynamic library for MatrixSSL is required, it should be built dynamically without the assembly language objects, and just those objects should be statically linked in to each of the applications using MatrixSSL for minimal code footprint. Also, note that some of the options below must be defined on OS X.

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.

Platform-Specific Requirements

The primary effort of a MatrixSSL port is to implement a small set of specific functionality required by the inner workings of the library. This section defines all the platform-specific requirements the developer must implement.

Design Framework

PeerSec Networks 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/osdep.h	Existing header file the developer should include the platform specific headers and typedefs to
core/<OS>/osdep.c	New C source code file the developer will use to implement the specific platform functions.

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 3.1 the currently supported defines are **POSIX** and **WIN32** and can be used as references during the porting process.

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.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 USE_INT64 is defined
`uint64`	A 64-bit unsigned integer type	Only required if USE_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.

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.

osdepTimeOpen

Prototype

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

Return Values

`PS_SUCCESS`	Successful initialization.
`PS_FAILURE`	Failed time module initialization. Shouldn't continue.

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`

osdepTimeClose

Prototype

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

Return Values

None.

Servers and Clients

This function performs the one-time final cleanup 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`

psGetTime

Prototype

```
int32 psGetTime(psTime_t *currentTime);
```

Parameters

`currentTime`	output	Platform-specific time format representing the current time or ticks

Return Values

`> 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`.

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.

psDiffMsecs

Prototype

```
int32 psDiffMsecs(psTime_t then, psTime_t now);
```

Parameters

`then`	input	Time structure from a previous call to `psGetTime`
`now`	input	Time structure from a previous call to `psGetTime`

Return Values

`> 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`.

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.

psCompareTime

Prototype

```
int32 psCompare(psTime_t a, psTime_t b);
```

Parameters

`a`	input	Time structure from a previous call to `psGetTime`
`b`	input	Time structure from a previous call to `psGetTime`

Return Values

`1`	`a` is earlier in time than `b` OR `a` and `b` are the same time
`0`	`b` is earlier in time than `a`

Implementation Requirements

This routine must be able to determine which time is earlier given two time structures.

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`.

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.

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 raw random 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-1). An application developer can request these processed bytes using a second API.

Desktop and Server operating systems typically implement both the collection of data 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 (Fortuna) suitable for a small footprint application, however the first requirement of collecting random data 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 PeerSec Networks** for guidance on gathering entropy for a specific hardware platform.

osdepEntropyOpen

Prototype

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

Return Values

`PS_SUCCESS`	Successful initialization.
`PS_FAILURE`	Failed entropy module initialization. Shouldn't continue.

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`

osdepEntropyClose

Prototype

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

Return Values

None.

Servers and Clients

This function performs the one-time final cleanup 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`

psGetEntropy

Prototype

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

Parameters

`bytes`	output	Random bytes must be copied to this buffer
`size`	input	The number of random bytes the caller is requesting

Return Values

`> 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 pseudo-random data. This function is internally invoked by several areas of the MatrixSSL code base. Please contact PeerSec Networks for guidance in implementing entropy gathering and for access to the Fortuna algorithm in C.

Server and Client Usage

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

Memory Profile

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

File Access Functions

These functions can optionally be implemented in *core/<OS>/osdep.c*. They are only required if `USE_FILE_SYSTEM` is defined in coreConfig.h

psGetFileBuf

Prototype

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

Parameters

`pool`	input	The memory pool if using PeerSec Deterministic Memory (`USE_PEERSEC_MEMORY_MANAGEMENT` is enabled)
`filename`	input	The filename (with directory path) of the file to open
`buf`	output	The contents of the filename in a memory buffer
`bufLen`	output	Length, in bytes, of `buf`

Return Values

`PS_SUCESS`	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` or `matrixSslLoadDhParams` is used during initialization.

Memory Profile

The allocated `buf` will be freed internally using `psFree` by the callers of `psGetFileBuf`.

Define Dependencies

`USE_FILE_SYSTEM`	optionally enable in *coreConfig.h*

Trace and Debug Functions

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.

osdepTraceOpen

Prototype

`int osdepTraceOpen(void);`

Return Values

`PS_SUCCESS`	Successful initialization.
`PS_FAILURE`	Failed trace module initialization. Shouldn't continue.

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`

osdepTraceClose

Prototype

`void osdepTraceClose(void);`

Return Values

None.

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`

_psTrace

Prototype

```
void _psTrace(char *message);
```

Parameters

`message`	input	A fully formed string message to output as debug trace

Implementation Requirements

This routine should output the `message` to the standard debug output location.

_psTraceStr

Prototype

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

Parameters

`message`	input	A string message containing a single `%s` format character that will be output as debug trace
`value`	input	A string value that should 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.

_psTraceInt

Prototype

```
void _psTraceInt(char *message, int32 value);
```

Parameters

`message`	input	A string message containing a single `%d` format character that will be output as debug trace
`value`	input	An integer that should be substituted for the `%d` 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.

_psTracePtr

Prototype

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

Parameters

`message`	input	A string message containing a single `%p` format character that will be output as debug trace
`value`	input	A void pointer that should be substituted for the `%p` in the message parameter.

Implementation Requirements

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

osdepBreak

Prototype

```
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 add them to the source code to help narrow down run-time problems.

Define Dependencies

`HALT_ON_PS_ERROR`	optionally enable in *core/coreConfig.h*

Standard Library Dependencies

MatrixSSL also relies 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.

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

Memory Allocation

malloc

Prototype

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

Implementation Requirements

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

realloc

Prototype

```
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.

free

Prototype

```
void free(void *ptr);
```

Implementation Requirements

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

Memory Operations

memcmp

Prototype

```
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.

memcpy

Prototype

```
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`.

memset

Prototype

```
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.

memmove

Prototype

```
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`.

strstr

Prototype

```
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.

strlen

Prototype

```
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 NUL character.

Additional Topics

Client and Server Socket-based 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 is easier to rewrite the examples with the native transport API than to try to implement the apis 1:1 below.

Function	Client	Server
`accept`		✓
`bind`		✓
`close`	✓	✓
`connect`	✓	
`exit`		✓
`fcntl`	✓	✓
`inet_addr`	✓	
`listen`		✓
`puts`		✓
`recv`	✓	✓
`select`		✓
`send`	✓	✓
`setsockopt`		✓
`signal`		✓
`socket`	✓	✓
`strncpy`	✓	✓

64-Bit Integer Support

If your platform supports 64-bit integer types (`long long`) you should make sure `USE_INT64` is enabled in *core/coreConfig.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 `USE_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.

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.

Threading

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 which 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.

fork()

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.

osdepMutexOpen

Prototype

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

Return Values

`PS_SUCCESS`	Successful initialization.
`PS_FAILURE`	Failed mutex module initialization. Shouldn't continue.

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`

osdepMutexClose

Prototype

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

Return Values

None.

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()`.

psCreateMutex

Prototype

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

Parameters

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

Return Values

`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.

psLockMutex

Prototype

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

Parameters

`mutex`	input	Mutex to lock

Return Values

`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.

psUnlockMutex

Prototype

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

Parameters

`mutex`	input	Mutex to unlock

Return Values

`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.

psDestroyMutex

Prototype

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

Parameters

`mutex`	input	Mutex to destory

Server Usage

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