# Porting mbed TLS to a new environment or OS



**Section:**
How to

**Author:**
Manuel Pégourié-Gonnard

**Published:**
Feb 17, 2016

**Last updated:**
Apr 23, 2017



mbed TLS is designed to be portable across different architectures and runtime environments, and can execute on a variety of different operating systems or on bare metal ports. Portability of the architecture is achieved by using the C language in a generic, portable way, while environment or architecture independence is achieved by minimizing its platform dependencies, reducing the amount of code that depends on a particular environment or OS and cleanly isolating platform specific code from the highly portable core so that platform code can be easily replaced. This article is about that latter part: it explains what should be done to port mbed TLS to a new environment and how you can do it.

Please note that this article is about the mbed TLS library, and not the yotta module that is part of mbed OS; it is about porting mbed TLS to a new runtime environment, not to a new hardware platform. It is also only about the library itself, not the mbed TLS example programs nor the test suites, which have different system requirements.

## Overview

mbed TLS has a modular design and many of the modules are completely independent of any runtime or environment dependencies, with the exception of the system-independent parts of the standard C library. The only parts of the library that potentially interact with the environment are:

- The network module *net_sockets.c* that can be disabled and replaced with a separate network stack. This can mean any transport layer stack that makes use of mbed TLS.
- The timing module *timing.c* that can be disabled and replaced to suit the underlying OS or hardware drivers.
- Default sources of entropy in the entropy module, and additional sources can be registered.
- Functions that access a filesystem that can be disabled and are optional in their use.
- Functions that want to know the current time from a realtime clock can be disabled, although that does limit what validation is possible for certificates.
- Functions that print messages, which are generally used for debug and diagnosis can be disabled or replaced to output the messages to other platform specific debug output.

In short, in order to compile mbed TLS for a bare metal environment which already has a standard C library, all you have to do is to configure your build by disabling `MBEDTLS_NET_C`, `MBEDTLS_TIMING_C` and `MBEDTLS_ENTROPY_PLATFORM`, and potentially `MBEDTLS_FS_IO`, `MBEDTLS_HAVE_TIME_DATE` and `MBEDTLS_HAVE_TIME` too. This is more thoroughly documented in config.h. The following sections have more detail on how to replace the missing parts.

## Networking

The provided network module *net_sockets.c* works on Windows and Unix systems that implement the BSD sockets API. It is only optionally used by the SSL/TLS module via callback functions and can be disabled at compile-time without affecting the rest of the library.

The callbacks can be replaced by coding your own functions for (blocking or non-blocking) write and read (optionally with a timeout), based on the network or transport layer stack of your choice. Substitute functions must match the API expected by the function `mbedtls_ssl_set_bio()`.

**Related articles:**

- What external dependencies does mbed TLS rely on?
- mbed TLS tutorial
- Obtaining Code Size
- DTLS tutorial
- mbed TLS Abstraction layers
- How to compile mbed TLS to a static library in Eclipse CDT
- Alternative cryptography engines implementation
- How to add entropy sources to the entropy pool
- Compiling mbed TLS in MinGW
- Thread Safety and Multi Threading: concurrency issues

## Timing

The provided timing module *timing.c* works on Windows, Linux and BSD (including OS X). It is only optionally used by the SSL/TLS module via callback functions for DTLS and can be disabled at compile-time without affecting the rest of the library.

If you're not using DTLS, you don't need a timing function. If you are using DTLS, you'll need to write your own timer callbacks suitable to pass to the function `mbedtls_ssl_set_timer_cb()`. This is discussed in more detail in our DTLS tutorial for a full description of how to use the callbacks.

## Default entropy sources

The entropy pool, part of the RNG module, collects and securely mixes entropy from a variety of sources. On Windows and different Unix platforms that provide */dev/urandom*, a default OS-based source is registered. It can be disabled at compile-time without affecting the rest of the library.

This source can be replaced by coding one or more entropy-collection functions that implement the API expected by the function `mbedtls_entropy_add_source()` and registering it with that function at runtime, or alternatively if it's based on a hardware source, at compile-time with `MBEDTLS_ENTROPY_HARDWARE_ALT`.

Please note that, for obvious security reasons, the entropy module will refuse to output anything until a declared-strong source has been registered.

**Warning:** Evaluating the strength of the sources provided is the responsibility of those doing the platform port.

## Hardware Acceleration

Most modules that implement cryptographic primitives, can be substituted with alternative implementations of the primitives to allow platforms to take advantage of the hardware acceleration that may be present. This can be achieved by defining in the appropriate `MBEDTLS_*_ALT` pre-processor symbol for each module that needs to be replaced. For example `MBEDTLS_AES_ALT` may be defined to replace the whole AES API with a hardware accelerated AES driver, and `MBEDTLS_AES_ENCRYPT_ALT` may be defined for replacing only the AES block encrypt functionality.

## Filesystem access

Several modules include functions that access the filesystem. All of them can be disabled at compile-time without affecting the rest of the library.

Every function that accesses the filesystem is only a convenience wrapper around a function that does the same job with memory buffers, so there is nothing to replace here - just use the functions that work on buffers.

## Real time clock

A few modules optionally access the current time, either to measure time intervals, or in order to know the absolute current time and date. Those features can be disabled at compile-time without affecting the rest of the library.

Every function that measures intervals has an alternate version of the code to provide similar functionality when time is not available (for example, rotating keys based on the number of uses rather than elapsed time). Absolute time and date are only used in X.509 in order to check the validity period of certificates - if it's not available then this check is skipped.

**Warning:** depending on how you use X.509 certificates to secure your platform, this could be a serious security risk!

## Diagnostic output

In the library, the only functions that print messages (using `printf()`) are the self-test functions. These can be disabled at compile-time (`MBEDTLS_SELF_TEST`) without affecting the rest of the library.

Alternatively, `printf()` can easily be replaced with your own printing function, thanks to the

platform layer, either by enabling `MBEDTLS_PLATFORM_PRINTF_ALT` at compile-time and then using `mbedtls_platform_set_printf()` at runtime, or by using `MBEDTLS_PLATFORM_PRINTF_MACRO` at compile-time.

## Conclusion

Thanks to its modular design, mbed TLS is easy to use on a variety of different platforms and environments. If it doesn't work out of the box in your environment, it's easy to provide your own implementations of the few parts that interact with the environment, and have the rest of mbed TLS use them.

## Did this help?