



MM-IoT-SDK Porting Guide

APPNOTE-45



Table of Contents

1	Introduction	4
1.1	Who this guide is for	4
1.2	What this guide covers (and what it doesn't)	4
1.3	Terminology	5
1.4	What “porting” means	5
1.5	How to use this guide	6
2	Requirements	7
2.1	Hardware components	7
2.1.1	Transport bus	7
2.1.2	GPIO pins	7
2.1.3	Morse Micro transceiver	9
2.1.4	Required memory	10
2.2	Recommended host peripherals	10
3	Porting context	11
3.1	Validating the port with example applications	12
4	The MM-IoT-SDK	14
4.1	Overview of the MM-IoT-SDK	14
4.1.1	Morse Micro UMAC and driver library – morselib	14
4.1.2	Morse Micro transceiver firmware binaries – firmware	15
4.1.3	Morse Micro packet memory – mmpkmem	15
4.1.4	Morse Micro regulatory database – mmregdb	15
4.1.5	Morse Micro utilities – mmutils	15
4.1.6	MM-IoT-SDK contents and structure	15
5	Porting process	17
5.1	Build integration	17
5.1.1	Integrate the MM-IoT-SDK	17
5.1.2	Create a platform stub	19
5.1.3	Build your platform stub	20
5.2	The Porting Assistant	20
5.2.1	Compiling on your platform	21
5.2.2	Operating system abstraction layer	22
5.2.3	Communications bus hardware abstraction layer	23
5.2.4	BCF and firmware	25
5.2.5	Porting Assistant completed	26
5.2.6	Porting Assistant benchmarks on supported platforms	27
5.3	Scan	27

5.3.1	Build integration	27
5.3.2	Run example	28
5.4	iPerf	29
5.4.1	Build integration	29
5.4.2	Run example	30
5.4.3	IP traffic benchmarks on supported platforms (EKH05)	31
6	Troubleshooting	33
6.1	Multiple definition linker errors - HostAP already defined	33
6.2	Transceiver fails to boot - SPI IRQ not implemented	33
6.3	Lower than expected data throughput	33
6.4	How can I debug SPI?	34
6.5	How to optimize flash and RAM usage?	34
	Glossary	36
A	Appendix	37
A.1	Building MM-IoT-SDK examples and morselib	37
A.1.1	Examples	38
A.1.2	morselib	39
A.2	Bus transport details	39
A.2.1	SPI	41
A.2.2	SDIO	42
A.3	Implementing a network stack driver	43
A.3.1	Initialization	43
A.3.2	Link State	44
A.3.3	Receive path	44
A.3.4	Transmit path	45
A.4	Converting .bin to .mbin	45
A.5	Flash and RAM requirements explained	46
A.5.1	Flash requirements explored	46
A.5.2	RAM requirements explored	47
A.6	Name mangling HostAP and morselib	48
A.7	Porting Stubs Repository	49
	Revision History	50

1 Introduction

1.1 Who this guide is for

This guide is for embedded firmware engineers porting the `mm-iot-sdk` to a microcontroller that Morse Micro does not officially support.

To follow this guide, you should be comfortable with:

- Embedded C and your target's toolchain and debugger;
- Driver-level programming of serial peripheral interface (SPI) or secure digital input output (SDIO), general-purpose input/output (GPIO) interrupts, and direct memory access (DMA) on your host;
- Basic real-time operating system (RTOS) concepts (for example, FreeRTOS tasks, queues, and semaphores);
- A C build system such as CMake or GNU Make; and
- The software environment that comes with your host.

You can complete a port without prior knowledge of HaLow, Wi-Fi, or Morse Micro transceivers.

1.2 What this guide covers (and what it doesn't)

This guide helps you do the following using `mm-iot-sdk-2.11.2`:

- Check you have a suitable hardware platform, including central processing unit (CPU) speed, memory capacities, and peripherals;
- Structure your porting code to prepare for future changes and upgrades to the `mm-iot-sdk`;
- Implement the necessary hardware abstraction layers (HALs); and
- Verify behavior and performance at each step.

If you have not yet chosen a host chip, a communication protocol, or a Morse Micro transceiver, start with a [supported platform](#), and iterate on it. For example, Morse Micro offers two variants of the [EKH05 IoT Development Board](#), one with each of our chips: [MM6108](#) and [MM8108](#). These boards come with an STM32U585 microcontroller unit (MCU) host connected via SPI to the Morse Micro transceiver, but can be modified to use SDIO (see the user guides in our [customer portal](#)).

Morse Micro also provides platform ports for Zephyr, ESP32, and CMSIS. If you are using one of these platforms, see [MCU Platform Ports](#) for more information.

Porting the `mm-iot-sdk` to a new platform is only one part of creating your application. This guide provides advice where it can, but it does not cover:

- The design and behavior of your application;
- Application-specific tradeoffs, such as whether a more capable platform justifies its higher cost; or

- Acceptable performance targets (for example, a battery-powered device may tolerate lower throughput if it improves power consumption).

Additionally, this guide will not cover porting for versions of the `mm-iot-sdk` earlier than 2.11.2.

1.3 Terminology

This guide uses the following terms to avoid confusion:

- **Host:** your MCU. The host runs your application and boots the transceiver.
- **Transceiver:** the Morse Micro HaLow transceiver that the host communicates with over SPI or SDIO.
- **Platform:** either the hardware your application runs on (for example, the EKH05 IoT Development Board), or the broader ecosystem of build system, operating system, drivers, and hardware (for example, a Zephyr platform port). Where the distinction matters, this guide uses **hardware platform** or **platform port**.

This guide avoids the term **chip** on its own. The host, the transceiver, and other peripherals such as QSPI flash are all chips, so the term is ambiguous. This guide uses **host**, **transceiver**, or the specific peripheral name instead.

1.4 What “porting” means

Porting the `mm-iot-sdk` means making it build, run, and communicate with a Morse Micro transceiver on a host platform that Morse Micro does not provide as a reference.

In practice, this involves work across the following layers:

- **Build integration:** Create the build scripts, such as Makefiles, and project files for building your application and platform.
- **Operating system:** Create the porting layer for your operating system’s primitives, such as tasks, semaphores, and timers.
- **Peripherals and communications:** Implement the GPIO, interrupt handling, and communications bus (SPI or SDIO) necessary to interact with the Morse Micro transceiver.

The process of porting moves through the tiers of completeness:

1. **Communicates with the Morse Micro transceiver:** the port passes the basic bring-up tests and exchanges messages with the transceiver. Verified by the `porting_assistant` example application.
2. **Reads incoming networking traffic:** the port can read beacons. Verified by the `scan` example application.
3. **Passes network traffic:** the port can transmit and receive traffic from other 802.11ah transceivers. Verified by the `iperf` example.
4. **Optimized for target application:** the port meets the performance requirements of the target

application, such as data throughput or power consumption. Verified by your application running successfully; this guide uses `iperf` and other examples as a proxy for your application.

This guide helps you work through each tier.

1.5 How to use this guide

Read the guide from start to finish. Each section builds on the previous sections:

1. [Section 2 – Requirements](#) helps you check that your hardware and software platform is sufficient for a port.
2. [Section 3 – Porting context](#) helps you form a high-level view of the parts of a port.
3. [Section 4 – The MM-IoT-SDK](#) shows you how to bring the `mm-iot-sdk` into your project.
4. [Section 5 – Porting process](#) steps through completing the port: the operating system abstraction layer (OSAL), the bus driver, build integration, and bring-up verification.

Use [Section 6 – Troubleshooting](#) as a reference when something does not work, rather than reading it front to back.

Before you start creating the software port, you should have:

- Your host development board, debugger, and toolchain set up and building a known-good “hello world” for the target.
- Your host platform connected to your Morse Micro transceiver (an evaluation board or module) via SPI or SDIO and the necessary GPIOs.
- A copy of `mm-iot-sdk` at the version this guide references (see [Section 2 – Requirements](#)).

This guide uses the following typographic conventions:

- monospace for file paths, directory names, function names, and shell commands.
- monospace for `mm-iot-sdk` example applications such as `iperf` (as opposed to `iPerf`, the application `iperf` originates from).

-

```
Code blocks for multi-line code
```

Listing 1: Code block

If you get stuck on something this guide does not cover, ask a question in the [Community Forum](#).

2 Requirements

This part of the guide describes the hardware required and software obligations you must meet to port the `mm-iot-sdk` successfully.

2.1 Hardware components

2.1.1 Transport bus

There are two types of transport bus available for communication with Morse Micro transceivers:

- [SPI](#)
- [SDIO](#)

Implement the HAL for one of these bus types, not both.

Morse Micro has found that porting to SDIO tends to be more straightforward because although the SDIO is more complex, SPI host controllers vary widely in behavior. For more information, see [Appendix A.2 – Bus transport details](#).

Note, the `mm-iot-sdk` does not support sharing the transport bus with other peripherals.

2.1.2 GPIO pins

The host microcontroller requires several GPIO pins to interface with the Morse Micro transceiver. See [Table 1](#) and [Figure 1](#).

MM Transceiver Pin Name	Data Direction	Function
BUSY	Transceiver → Host	Prevents host from sleeping. The host microcontroller must configure this pin as an input that fires an interrupt on a rising edge.
WAKE	Host → Transceiver	Prevents transceiver from sleeping. This can be permanently set HIGH [†] until you need to optimize power consumption for your application.
RESET_N	Host → Transceiver	LOW signal resets the transceiver. Equivalent to a cold boot.

MM Transceiver Pin Name	Data Direction	Function
SPI_IRQ	Transceiver → Host	SPI only. Indicates the host should read data from the transceiver. The host must configure this pin as an input that fires an interrupt on a falling edge.
CSS	Host → Transceiver	SPI only. Software-controlled chip select. Lets the host send a clock training sequence to the transceiver.
* See the important note below.		

Table 1: GPIO pin for host microcontroller

Important

Your port must call `mmwlan_set_sleep_mode(MMWLAN_SLEEP_DISABLE)` after booting the transceiver, even when WAKE is held permanently HIGH. Otherwise the driver fails to boot. See the SPI host requirement or SDIO host requirement section in your transceiver's datasheet for detailed schematics, electrical, and timing requirements.

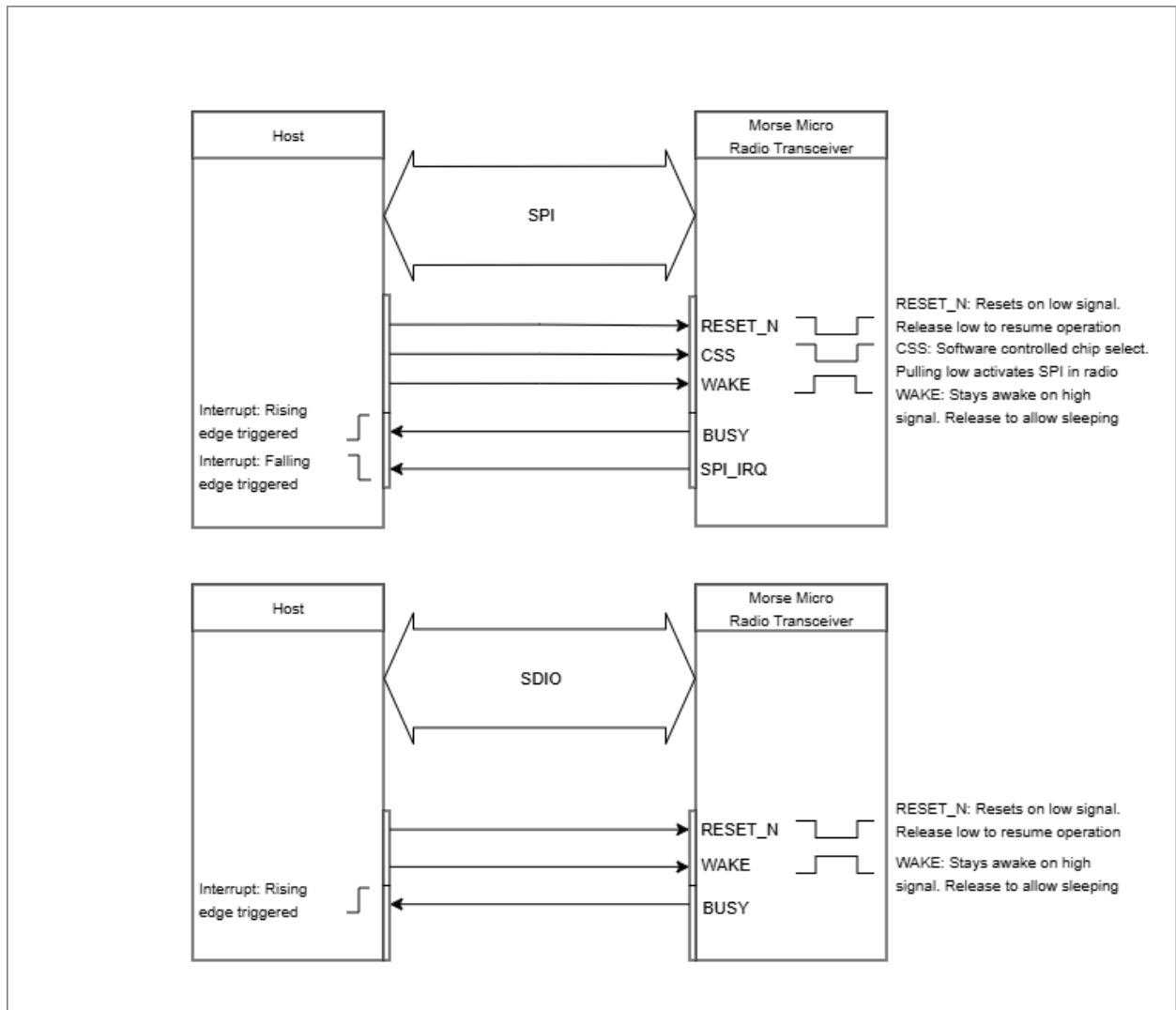


Figure 1: GPIO lines required for SPI vs SDIO

2.1.3 Morse Micro transceiver

There are two Morse Micro transceivers. Review each datasheet to choose the best transceiver for your application.

Morse Micro transceiver	Datasheet
MM6108	MM6108 datasheet
MM8108	MM8108 datasheet

Table 2: Available transceivers and associated datasheets

2.1.4 Required memory

The `mm-iot-sdk` operates as a SoftMAC, so your host platform is responsible for loading firmware and a board configuration file (BCF) onto the Morse Micro transceiver, and for hosting `morselib` — both its code and its runtime random access memory (RAM).

We recommend your porting platform has at least:

1. ≥ 2 megabytes of flash memory.
2. ≥ 256 kilobytes of SRAM, to support the needs of `morselib` at runtime.

These figures are not necessarily the upper or lower limits for your final product, but starting with a platform that comfortably supports debugging builds will make the porting effort more straightforward. See Appendix A.5 for the reasoning behind these numbers.

2.2 Recommended host peripherals

1. A public key accelerator (PKA) cryptographic peripheral to accelerate elliptic-curve cryptography (ECC) operations. This shortens association time and reduces RAM usage during cryptographic operations.
2. A random number generator on the chosen platform.

3 Porting context

The `mm-iot-sdk` operates as a SoftMAC: Morse Micro-supplied firmware running on the transceiver handles the lower media access control (LMAC) and PHY, while the upper media access control (UMAC) runs in the firmware you build for your host.

Your application communicates with the transceiver via the `morselib` driver, provided by Morse Micro. To run `morselib` on your host, you must implement the functions it depends on.

Before you write any port code, identify which pieces of the stack you implement and which Morse Micro provides. [Figure 2](#) maps a typical `mm-iot-sdk` application onto the host and the Morse Micro transceiver.

The purple boxes — the OSAL, GPIO layer, communications bus driver, and possibly an IP stack driver — are the components **you implement** during the port.

`morselib`, the transceiver firmware, and the example applications are provided by Morse Micro.

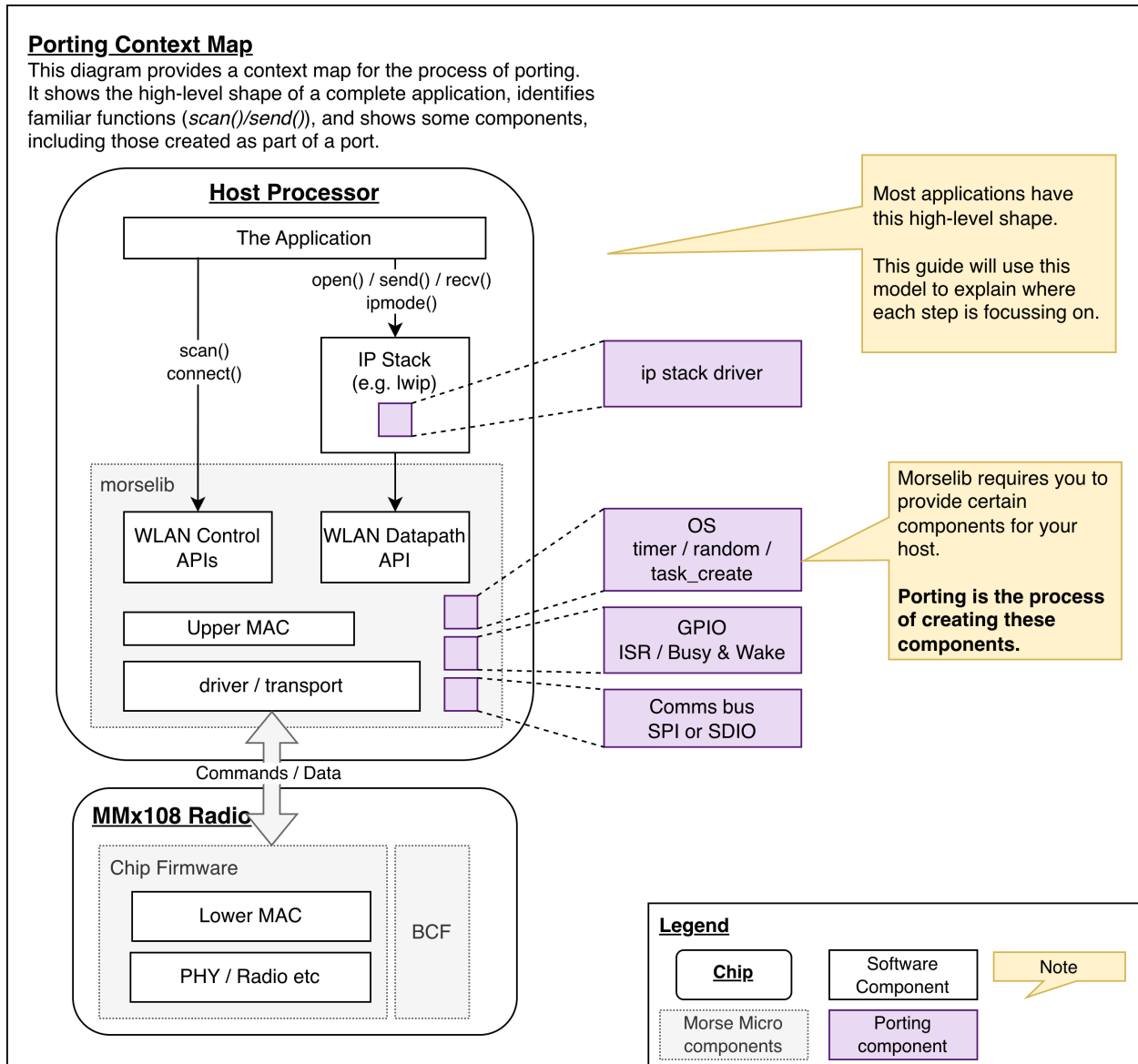


Figure 2: Porting context map. Purple boxes are the components you implement; everything else is provided by Morse Micro or by your existing software stack.

3.1 Validating the port with example applications

The `mm-iot-sdk` ships with example applications that you run on the host to verify successively larger slices of the stack. Each one exercises a different range of components, as shown in Figure 3:

- `porting_assistant` validates the host-side components you write during the port: the OSAL, GPIOs, and communications bus. It does not require a working transceiver link.
- `scan` validates basic WLAN integration – that the host can boot and use the transceiver.
- `iperf` validates the full end-to-end data path.

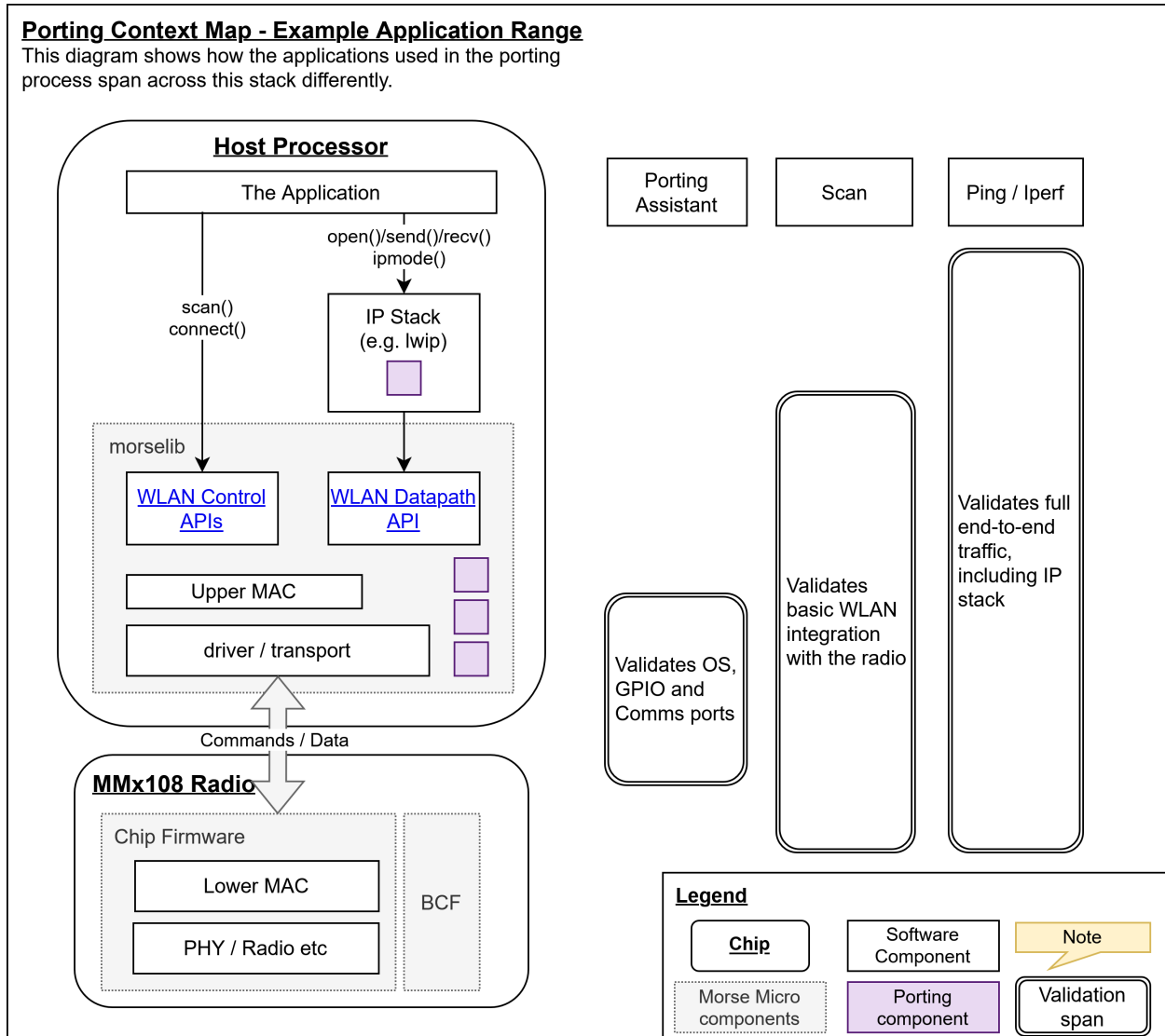


Figure 3: Validation spans of the example applications. Each application exercises progressively more of the stack, from the host-side porting components alone (porting_assistant) up to full end-to-end traffic (iperf).

4 The MM-IoT-SDK

4.1 Overview of the MM-IoT-SDK

The `mm-iot-sdk` includes example applications and libraries, the Morse Micro driver, and the supported platforms. It also includes firmware for each Morse Micro transceiver, BCF files for some common modules, and the UMAC code. The Morse Micro driver and essential HaLow libraries are called `morselib`. This component is what manages communication with the transceiver.

When porting the `mm-iot-sdk` to your own platform, the following components are required:

- `morselib` - the core Morse Micro UMAC and driver library;
- `firmware` - the Morse Micro transceiver firmware and BCF;
- `mmpktmem` - a library for allocating and managing packet memory used by `morselib`;
- `mmregdb` - the Wi-Fi HaLow regulatory database;
- `mmutils` - a library of helper utilities.

Most applications interact with the transceiver via a TCP/IP-capable network stack running on the host. You may provide your own network stack, or use the `lwip` component provided in the `mm-iot-sdk`.

This guide uses four example applications as you progress through the porting process; porting assistant, scan, and iPerf. These example applications rely on components which aren't strictly required for your application. They include:

- `mmipal` - the `mm-iot-sdk`'s abstraction over networking and TCP/IP, as well as an `lwip` driver;
- `mmconfig` - the `mm-iot-sdk`'s configuration management library;
- `mmlog` - the `mm-iot-sdk`'s logging helper library;
- `mmiperf` - the `mm-iot-sdk`'s iPerf implementation, with an abstraction layer for extension to other network stacks;

4.1.1 Morse Micro UMAC and driver library – `morselib`

The Upper MAC in this library implements the Wi-Fi HaLow protocol. It provides functions for converting 802.3 frames into 802.11 data packets, building 802.11 management frames, and managing the state of the Wi-Fi stack.

The driver included in this library is a platform-agnostic implementation of the protocol used to communicate with the Morse Micro transceiver. To remain platform-agnostic, you must implement a small number of hardware abstraction layers. These include SPI communication functions, GPIO handlers, operating system primitives and more.

`morselib` has dependencies on `hostap` and `mbedtls`, included in the `mm-iot-sdk`.

4.1.2 Morse Micro transceiver firmware binaries – firmware

To boot the Morse Micro Wi-Fi HaLow transceiver, the host-side driver code loads two binaries onto the transceiver: the transceiver firmware itself and the BCF. The `mm-iot-sdk` provides both of these files in a `.mbin` format (see `mm-iot-sdk/framework/morsefirmware`).

Module vendors provide BCFs in a `.bin` format. Both formats carry the same information, but the `.mbin` is a compressed format designed to reduce flash memory requirements on MCU hosts. You must convert these to `.mbin` before your application can load them onto the transceiver. See Appendix A.4 for instructions on how to perform a conversion.

During porting, you provide the functionality that lets `morselib` read these binaries.

4.1.3 Morse Micro packet memory – mmpktmem

`mmpktmem` provides platform-agnostic memory handling to support the transmitting and receiving of packets. This library provides two implementations that can be used by `morselib` - one using heap memory, and the other using statically allocated memory. You must choose one for your platform.

4.1.4 Morse Micro regulatory database – mmregdb

Regulatory restrictions vary globally for Wi-Fi HaLow. The `mmregdb` defines the operating channels, bandwidths, and maximum power limits for each supported region.

The BCF provided to you by your module vendor enforces these restrictions. The regulatory database may include more information than provided by your BCF. Contact your module manufacturer for specific details.

No implementation is required from you to include this component in your port.

4.1.5 Morse Micro utilities – mmutils

The helper utilities in `mmutils` parse common elements in beacons and other Wi-Fi management frames. These elements include robust security network (RSN) information, vendor elements, and the S1G operation fields.

No implementation is required from you to include this component in your port.

4.1.6 MM-IoT-SDK contents and structure

HTML documentation for the `mm-iot-sdk` is included with each release. The documentation for the latest release can be viewed at [Morse Micro IoT SDK Documentation](#).

Path	Description
<code>applications/ examples/</code>	Example applications.
<code>framework/doc/</code>	HTML documentation, including Getting Started Guide and API documentation.
<code>framework/mk/</code>	Makefile fragments.
<code>framework/morsefirmware/</code>	Firmware and BCF files for supported Morse Micro transceiver modules.
<code>framework/morserlib/</code>	The core part of the SDK. Contains the UMAC and driver.
<code>framework/morserlib/include/</code>	Contains the definitions of the core APIs that <code>morserlib</code> relies on. Part of the porting work is implementing the functions defined in these APIs.
<code>framework/src/</code>	Contains the supporting software components and middleware.
<code>framework/src/hostap/</code>	HostAP project with modifications to support HaLow.
<code>framework/src/lwip/</code>	LWIP project with modifications to improve HaLow performance.
<code>framework/src/platforms/</code> <code>framework/src/bsps/</code>	Platform-specific code. Includes microcontroller BSP code, HAL shims, and the OSAL shim.
<code>framework/tools/</code>	Tools that run on your computer to support development and debugging.

Table 3: mm-iot-sdk package structure

5 Porting process

To complete a port, we recommend you follow this process:

1. Integrate the `mm-iot-sdk` components into your build system
2. Compile and run Porting Assistant (PA)
 - (a) Implement the OSAL
 - (b) Implement the communication HAL
 - (c) Implement the BCF and firmware collectors
3. Compile and run Scan to test more of the UMAC functionality.
4. Compile and run iPerf to test an application with a full network stack and get a throughput benchmark

This moves you through the tiers of completeness outlined in [Section 1.4 – What “porting” means](#), and gives you confidence that your port is working correctly at each stage of the process.

5.1 Build integration

Goal: Integrate the `mm-iot-sdk`, the example applications and your platform with your build system.

Success: You can use your build system to compile and link example applications against the `mm-iot-sdk` using a stub implementation of your platform, and can flash the binary image produced to your host.

Command examples in this guide assume the `mm-iot-sdk` Makefile-based system. Adapt them if you use a different build system.

5.1.1 Integrate the MM-IoT-SDK

We recommend you:

1. Implement your application and platform build system so it uses the `MMIOT_ROOT` environment variable to refer to `mm-iot-sdk` components. This lets you move the `mm-iot-sdk` to wherever is most convenient on your filesystem.
2. Create a version-controlled copy of the `mm-iot-sdk` source code in your project directory.

This approach helps you:

1. Switch between versions of the `mm-iot-sdk` to upgrade or compare features.
2. Apply and track patches to your copy of the `mm-iot-sdk`.
3. Keep your application logic, your platform SDK, and the `mm-iot-sdk` separate, which can help with support.

A recommended project structure is shown in [Figure 4](#).

```
YOUR_REPOSITORY/  
├── third_party/  
│   └── mm-iot-sdk-<version>/  
├── your_application/  
│   ├── Makefile  
│   ├── include/  
│   └── src/  
├── targets/  
│   └── your_port/  
│       ├── mm_shims/  
│       ├── bsp/  
│       └── platform.mk
```

Figure 4: Recommended project structure: mm-iot-sdk outside your project directory

We strongly recommend that you do **not** place your project inside the `mm-iot-sdk` directory, as shown in [Figure 5](#). Making direct edits to the `mm-iot-sdk` prevents you from switching versions easily and blurs the line between platform and application.

```
mm-iot-sdk-<version>/
├── applications/
├── framework/
├── LICENSES/
├── examples/
│   ├── iperf/
│   ├── scan/
│   ├── ...
│   └── your_project/
│       ├── Makefile
│       ├── src/
│       └── include/
```

Figure 5: Not recommended project structure: your project inside the `mm-iot-sdk` directory

5.1.2 Create a platform stub

First, create a stub of your platform port for your host.

This guide assumes you have:

- Copied an example platform stub from the `mm-iot-sdk` into your project, or
- Based your platform on the porting-stubs repository provided as a companion resource to this guide (see [Appendix A.7 – Porting Stubs Repository](#) for more information), or
- Created your platform in a way that you are familiar with.

This guide also assumes you have created a directory structure as shown in [Figure 6](#), and that your platform specific files are part of the BSP defined in the `./bsp` directory.

```
your_port/
├── mm_shims/
├── bsp/
└── platform.mk
```

Figure 6: Assumed platform with `mm_shims` and `bsp` structure.

Creating the `bsp` folder content depends on your platform; consult your platform's documentation. It will involve configuring host clocks, choosing and configuring peripherals. This guide **cannot** provide guidance on these steps. The result should be a set of C files that provide platform drivers and configuration, plus a `main` function that initializes the platform and then calls `mmosal_main(app_init)`. See the `mm-iot-sdk` example platforms for reference.

[Section 5.2 – The Porting Assistant](#) walks you through implementing the functions in the `mm_shims` directory (where most of your port will be). You will likely also modify the `bsp` files to

provide the required functionality.

5.1.3 Build your platform stub

Once you have created your platform stub and integrated the `mm-iot-sdk` with your build system, you should be able to build an example application for your platform.

Use this command:

```
make -C applications/porting_assistant \  
MMIOT_ROOT=./third_party/mm-iot-sdk-<version>/framework \  
TARGET=targets/your_port
```

Listing 2: Compile

This command will:

- Compile all of the necessary object files from their source, and
- Link the object files to produce a binary image (an ELF file).

You should be able to program and run that stub application on your host.

```
openocd -c "set MM_PROGRAM 1" \  
-f targets/your_port \  
-c "program applications/porting_assistant/porting_assistant.elf  
verify reset exit"
```

Listing 3: Program to your host

If you can compile the ELF and program it to your platform you have completed this step. The application is not expected to run at this stage.

5.2 The Porting Assistant

The Porting Assistant is an example application in the `mm-iot-sdk` that performs a series of tests to validate that the basic functionality of your port is working. This guide uses it to step through the process of doing the port. The steps are:

1. The Porting Assistant compiles and runs on your platform.
2. The OSAL tests pass.
3. The communications HAL tests pass.
4. The firmware and BCF tests pass.

5.2.1 Compiling on your platform

Goal: Your port has the implementations allowing the `porting_assistant` to execute tests and to output the results to a serial port.

Success: The `porting_assistant` runs on your platform, and outputs the failure for all tests.

At the end of this step, your serial output should be similar to below.

```
MM-IoT-SDK Porting Assistant
-----
HW Version:                MM-MM8108-EKH05
morselib version:          2.11.2
-----

Memory allocation           [ FAIL ]
Memory reallocation         [ FAIL ]
Passage of time             [ FAIL ]
Task creation and preemption [ FAIL ]
WLAN HAL initialization
Hard reset device
SDIO/SPI Startup            [ FAIL ]
Read chip id from the MM chip [ FAIL ]
Verify BUSY pin             [ FAIL ]
Bulk write/read into the MM chip [ FAIL ]
Raw throughput test         [ FAIL ]
Validate MM firmware         [ FAIL ]
Validate BCF                 [ FAIL ]
```

Listing 4: Captured Serial output of Porting Assistant application - Compile

The Porting Assistant application consists of 6 source files and 5 header files.

1. `chip_cfg.c`
2. `chip_cfg.h`
3. `config.h`
4. `log.h`
5. `porting_assistant.c`
6. `porting_assistant.h`
7. `sdio_spi.c`
8. `sdio_spi.h`
9. `test_os.c`
10. `test_wlan_fw_bcf.c`
11. `test_wlan_io.c`

Follow these steps:

1. Implement `mmhal_log_write()` so that the Porting Assistant can output results to a serial terminal.
2. Provide a function stub for all functions that fail to link. We advise putting `ASSERT(false)` calls inside the stubs to understand the call hierarchy at runtime.

Your Porting Assistant example should now compile. Flash the compiled output to your platform and verify you see the output via a serial port.

5.2.2 Operating system abstraction layer

Goal: Your port implements the OSAL functions for memory management and basic operating system dependencies.

Success: The Porting Assistant passes the OSAL tests. All other tests fail.

At the end of this step, your serial output should be similar to below.

```
MM-IoT-SDK Porting Assistant
-----
HW Version:                MM-MM8108-EKH05
morselib version:          2.11.2
-----

Memory allocation           [ PASS ]
Memory reallocation         [ PASS ]
Passage of time             [ PASS ]
Task creation and preemption [ PASS ]
WLAN HAL initialization
Hard reset device
SDIO/SPI Startup           [ FAIL ]
Read chip id from the MM chip [ FAIL ]
Verify BUSY pin            [ FAIL ]
Bulk write/read into the MM chip [ FAIL ]
Raw throughput test        [ FAIL ]
Validate MM firmware       [ FAIL ]
Validate BCF               [ FAIL ]
```

Listing 5: Captured Serial output of Porting Assistant application - OSAL

The porting assistant tests the operating system (OS) functionality that `morselib` requires: threads, timers, and memory allocation.

Test	Associated function
Memory allocation	<code>mmosal_malloc_()</code>

Test	Associated function
Memory reallocation	<code>mmosal_realloc_()</code>
Passage of time	<code>mmosal_get_time_ms()</code>
Task creation and preemption	<code>mmosal_task_create()</code>

Table 4: OSAL functions tested

If you are using FreeRTOS for your OS, you can use the `mm-iot-sdk` FreeRTOS OSAL implementation, `mmosal_shim_freertos.c`. Confirm your FreeRTOS version matches the one provided in the `mm-iot-sdk` (v11.2). Otherwise, refer to the FreeRTOS OSAL to understand how to implement the OSAL for your chosen OS. All of the functions in `mmosal.h` and `mmhal_os.h` must be implemented, however note that not all of these functions are tested in the Porting Assistant.

5.2.3 Communications bus hardware abstraction layer

Goal: Your port implements the configuration for your communications bus peripheral and implements the HAL functions for communications.

Success: The Porting Assistant passes the OSAL tests and the communications HAL tests. The firmware and BCF validation tests fail.

You must have a host that is physically connected to a Morse Micro transceiver for this step.

At the end of this step, your serial output should be similar to below.

```
MM-IoT-SDK Porting Assistant
-----
HW Version:                MM-MM8108-EKH05
morselib version:          2.11.2
-----

Memory allocation           [ PASS ]
Memory reallocation         [ PASS ]
Passage of time             [ PASS ]
Task creation and preemption [ PASS ]
WLAN HAL initialization
Hard reset device
SDIO/SPI Startup            [ PASS ]
Read chip id from the MM chip [ PASS ]
Verify BUSY pin             [ PASS ]
Bulk write/read into the MM chip [ PASS ]
Raw throughput test         [ PASS ]
Validate MM firmware         [ FAIL ]
Validate BCF                 [ FAIL ]
```

Listing 6: Captured Serial output of Porting Assistant application - communications HAL

The communications HAL allows `morselib` to control the selected Morse Micro transceiver. This includes the GPIO lines, RESET, WAKE, BUSY, SPI_IRQ, as well as the communications protocol (SPI or SDIO). The test and its associated functions are listed in the table below.

Test	Associated functions
WLAN HAL Initialization	<code>mmhal_wlan_init()</code>
Hard reset device	<code>mmhal_wlan_hard_reset()</code> <code>mmhal_wlan_assert_reset()</code>
Verify BUSY pin	<code>mmhal_wlan_busy_is_asserted()</code>
SDIO/SPI Startup	Implement SPI or SDIO, not both SDIO
Read chip id from the MM chip	<code>mmhal_wlan_sdio_startup()</code> <code>mmhal_wlan_sdio_cmd53_write()</code> <code>mmhal_wlan_sdio_cmd53_read()</code> <code>mmhal_wlan_sdio_cmd</code>
Bulk write/read into the MM chip	SPI <code>mmhal_wlan_send_training_sequence()</code> <code>mmhal_wlan_spi_cs_assert()</code> <code>mmhal_wlan_spi_cs_deassert()</code>
Raw throughput test	<code>mmhal_wlan_spi_write_buf()</code> <code>mmhal_wlan_spi_read_buf()</code> <code>mmhal_wlan_spi_rw()</code>

Table 5: communications HAL Tests

We recommend implementing the individual GPIO manipulation functions first. Test each one by calling its `assert`/`deassert` function from your application code while observing the pin on a logic analyzer. For the purposes of an initial port, the WAKE GPIO can be kept in a permanently high state. It will ensure that the chip can never sleep, or make use of powersave features.

Morse Micro has implemented communications HALs for use with a selection of STM32 products. If you're using an STM32 product, use the reference shims indicated in [Table 6](#) to create your communications HAL. Otherwise, refer to these implementations while writing your own.

Please be aware that the reference shims include additional components (such as `mmconfig` and `littlefs`), which are not required to successfully complete your port.

MCU	mm-iot-sdk platform name
STM32H753xx	mm-ekh08-h753
STM32U575xx	mm-ekh08-u575
STM32U585xx	mm-mm6108-ekh05 / mm-mm8108-ekh05
STM32WB55xx	mm-ekh08-wb55

Table 6: Supported STM32 platforms

The "Raw throughput test" sends bytes to the transceiver, which echoes them back to the host. On receiving the echoed bytes, the PA verifies that they match the bytes it transmitted. After completing successfully, the throughput test will provide a measure of how fast the transport bus is. Refer to [Section 5.2.5](#) for what the captured serial looks like for a complete Porting Assistant. Refer to [Section 5.2.6](#) to compare your platform's performance against the officially supported platforms.

5.2.4 BCF and firmware

Goal: Your port provides the firmware and BCF to the transceiver.

Success: The Porting Assistant passes all tests.

At the end of this step, your serial output should be similar to below.

```
MM-IoT-SDK Porting Assistant
-----
HW Version:                MM-MM8108-EKH05
morselib version:          2.11.2
-----

Memory allocation           [ PASS ]
Memory reallocation         [ PASS ]
Passage of time             [ PASS ]
Task creation and preemption [ PASS ]
WLAN HAL initialization
Hard reset device
SDIO/SPI Startup           [ PASS ]
Read chip id from the MM chip [ PASS ]
Verify BUSY pin            [ PASS ]
Bulk write/read into the MM chip [ PASS ]
Raw throughput test         [ PASS ]
Validate MM firmware        [ PASS ]
Validate BCF                [ PASS ]
```

Listing 7: Captured Serial output of Porting Assistant application - BCF and Firmware

Implement the functions in the table below to handle firmware and BCF files.

Test	Associated functions
Validate MM firmware	<code>mmhal_wlan_read_fw_file()</code>
Validate BCF	<code>mmhal_wlan_read_bcf_file()</code>

Table 7: BCF and Firmware tests

See `mmhal_wlan_binaries.c` for a reference implementation of these functions.

The transceiver firmware and BCF files are provided in `mm-iot-sdk/framework/morsefirmware/` in an `mbin` format. The most direct way to provide these files is to convert the `mbin` files into C arrays and compile them into your application.

```
xxd -i <file.mbin> | sed 's/unsigned char/const unsigned char/' >
<file.c>
```

Listing 8: Convert `mbin` to C array

Another method to include the BCF in your application is to create an object file from the `.mbin` file and compile it with the other source files. Refer to `framework/mk/morsefirmware.mk` and the build define `INCLUDE_BCF_FILE_IN_APPLICATION` in the `mm-iot-sdk` to see how this works.

5.2.5 Porting Assistant completed

```
MM-IoT-SDK Porting Assistant
-----
HW Version:                MM-MM8108-EKH05
morselib version:          2.11.2
-----

Memory allocation           [ PASS ]
Memory reallocation         [ PASS ]
Passage of time             [ PASS ]
Task creation and preemption [ PASS ]
WLAN HAL initialisation
Hard reset device
SDIO/SPI Startup           [ PASS ]
Read chip id from the MM chip [ PASS ]
Chip ID: 0x0306
Verify BUSY pin            [ PASS ]
Bulk write/read into the MM chip [ PASS ]
Raw throughput test        [ PASS ]

Note: This will not be the final WLAN TPUT. See test step for more
```

```

info.
Time spent (ms): 2501
Transaction count: 2572
Bytes Transferred: 7695424
Raw TPUT (kbit/s): 24615

Validate MM firmware [ PASS ]
Validate BCF [ PASS ]

```

Listing 9: Captured Serial output of Porting Assistant application - Fully completed

5.2.6 Porting Assistant benchmarks on supported platforms

Official Platform	Time Spent (ms)	Transaction Count	Bytes Transferred	Raw TPUT (kbit/s)
mm-mm6108-ekh05	2501	2254	6743968	21572
mm-mm8108-ekh05	2501	2564	7671488	24538
mm-mm8108-ekh05-sdio	2501	11670	34916640	111688

Table 8: Porting Assistant benchmarks on supported platforms

5.3 Scan

Scan builds upon the previous success with the Porting Assistant by validating that the transceiver and UMAC control paths are functional. Scan works by sending a probe request to nearby access points, then captures the probe response and prints it to the serial terminal.

For this example, you need access to another HaLow device that can act as an access point (AP). Good examples include:

- HaLowLink1 - [DigiKey](#)
- HaLowLink2 - [DigiKey - AU Version](#)
- EKH01 - [DigiKey](#)

5.3.1 Build integration

Goal: Compile an example application that uses Wi-Fi HaLow functionality.

Success: The scan example builds without errors and is ready to flash to your platform.

To compile the scan application, run this command:

```
make -C applications/scan MMIOT_ROOT=./third_party/mm-iot-sdk-<
version>/framework TARGET=targets/your_port
```

Listing 10: Compile

Follow these steps:

1. Set COUNTRY_CODE to your region in mm_app_loadconfig.c. It must be a region that matches the list of domains in mmregdb.c

```
struct mmwlan_slg_channel_list regulatory_db_domains[].
```
2. Remove the following components from the example (they are optional and can be removed without affecting its main functionality): mmconfig, mbedtls, and littlefs.
3. Build the example. If there are build errors or missing files, Appendix A.1 has instructions to identify files used in the build.

5.3.2 Run example

Goal: Run the Scan example **Success:** The scan example executes, outputting details of Wi-Fi HaLow networks it can hear.

Follow these steps:

1. Set up an AP. Refer to the user guide for your particular platform.
2. (Optional) Set up a third HaLow device as a sniffer to see the packets being broadcast on the air.
3. Run the microcontroller platform.

Expected output should be similar to below

```
-----
HW Version:           MM-MM8108-EKH05
BCF API version:      12.1.0
BCF build version:    be169c87 ef639c34fd
BCF board description: {"module": "mm8108-mf15457"}
morselib version:     2.11.2
Morse firmware version: 1.17.8
Morse chip ID:        0x0809
Morse chip name:      MM8108B2
-----

Scan started on US channels, Waiting for results...
MorseMicro-Demonstration
Operating BW: 8 MHz
BSSID: 0c:bf:74:01:23:45
RSSI: -31 dBm
Noise: -107 dBm
```

```

Beacon Interval(TUs): 100
Capability Info: 0x0031
Security: SAE
SlG Operation:
Operating class: 4
Primary channel: 26
Primary channel width: 2 MHz
Operating channel: 28
Operating channel width: 8 MHz

```

Listing 11: Scan serial capture

You have now successfully scanned the area for HaLow networks.

5.4 iPerf

Goal: Validate your platform can pass network traffic.

Success: iPerf successfully runs, outputting a successful throughput result.

5.4.1 Build integration

Follow the same steps outlined in [Section 5.3.1](#).

You must configure the iPerf to connect to your AP to run a throughput test. Refer to `iperf/src/mm_app_loadconfig.c` to configure the country code, service set identifier (SSID), passphrase and ensure `SECURITY_TYPE` matches the security type on your AP.

```

/* Default Country Code */
#ifndef COUNTRY_CODE
/** Country code. (Do not quote; it will be stringified, as it can
    be passed at build time.) */
/* clang-format off */
#define COUNTRY_CODE US
/* clang-format on */
#endif

/* Default SSID */
#ifndef SSID
/** SSID of the AP to connect to. (Do not quote; it will be
    stringified.) */
#define SSID MorseMicro
#endif

/* Default passphrase */
#ifndef SAE_PASSPHRASE
/** Passphrase of the AP (ignored if security type is not SAE).

```

```

* (Do not quote; it will be stringified.) */
#define SAE_PASSPHRASE 12345678
#endif

/* Default security type */
#ifndef SECURITY_TYPE
/** Security type (@see mmwlan_security_type). */
#define SECURITY_TYPE MMWLAN_SAE
#endif

```

Listing 12: Configuration snippet from `iperf/src/mm_app_loadconfig.c`

Additionally, you can configure the `iperf` application. Refer to `iperf/src/iperf.c` for configuration options and see the snippet below

```

#ifndef IPERF_TYPE
/** Type of iperf instance to start. */
#define IPERF_TYPE IPERF_UDP_SERVER
#endif

#ifndef IPERF_TIME_AMOUNT
/**
 * Duration for client transfers specified either in seconds or
 * bytes.
 * If this is negative, it specifies a time in seconds; if positive
 * , it
 * specifies the number of bytes to transmit.
 */
#define IPERF_TIME_AMOUNT -10
#endif
#ifndef IPERF_SERVER_PORT
/** Specifies the port to listen on in server mode. */
#define IPERF_SERVER_PORT 5001
#endif

```

Listing 13: Configuration snippet from `iperf/src/iperf.c`

5.4.2 Run example

1. Set up an AP. Refer to the user guide for your particular platform. Ensure that `iPerf` (v2.x) is available on it.
2. (Optional) Set up a third HaLow device as a sniffer to see the packets being broadcast on the air.
3. Run the microcontroller platform.

```

Morse iPerf Demo (Built Jun  2 2026 12:18:02)

-----
HW Version:                MM-MM8108-EKH05
BCF API version:           12.1.0
BCF build version:         be169c87 ef639c34fd
BCF board description:     {"module": "mm8108-mf15457", "s"}
morselib version:          2.11.2
Morse firmware version:    1.17.8
Morse chip ID:             0x0809
Morse chip name:           MM8108B2
-----

Initialize IPv4 using DHCP...
Morse LwIP interface initialised.
Attempting to connect to MorseMicro-Demonstration with passphrase
12345678
This may take some time (~10 seconds)
WLAN STA connecting
WLAN STA connected
Morse STA interface up with MAC address 0c:bf:74:01:23:45
4865 NetIf Link status DOWN
9961 NetIf Link status UP
DNS server 0: 10.0.0.1
Link is up. Time: 9961 ms, IP: 10.0.0.232, Net-mask:
255.255.255.0, Gateway: 10.0.0.1, DNS server 01

Iperf UDP server started, waiting for client to connect...
Execute cmd on AP 'iperf -c 10.0.0.232 -p 5001 -i 1 -u -b 20M' for
IPv4

# iperf -c 10.0.0.232 -p 5001 -i 1 -u -b 20M is executed on the AP
# Replace IP with the one provided by your platform

Iperf Report
Remote Address: 10.0.0.1:48240
Local Address:  0.0.0.0:5001
Transferred: 25 MBytes, duration: 9950 ms, bandwidth: 20867 kbps

```

Listing 14: iPerf serial capture

5.4.3 IP traffic benchmarks on supported platforms (EKH05)

The following table shows benchmark values for the supported platforms. They are provided for comparison, though your platform may differ.

Transceiver used in AP	STA Device	Protocol	UDP Downlink	UDP Uplink	TCP Downlink	TCP Uplink
MM6108 AP	MM6108-EKH05	SPI	18.0 Mbps	18.0 Mbps	8.1 Mbps	6.4 Mbps
MM6108 AP	MM6108-EKH05	SDIO	24.0 Mbps	24.0 Mbps	9.5 Mbps	7.0 Mbps
MM8108 AP	MM8108-EKH05	SPI	19.0 Mbps	19.9 Mbps	9.8 Mbps	7.8 Mbps
MM8108 AP	MM8108-EKH05	SDIO	30.0 Mbps	24.9 Mbps	10.5 Mbps	8.5 Mbps

Table 9: Benchmark TPUT (Mbps) by transceiver and communication protocol

6 Troubleshooting

6.1 Multiple definition linker errors - HostAP already defined

Symptoms: Multiple definition linker errors on HostAP symbols.

If HostAP is already defined in your project (for example, for a 2.4 GHz wireless interface) but you want to include `morselib` for a 900 MHz interface, you encounter multiple definition linker errors. `morselib` requires the modified HostAP supplied by `mm-iot-sdk`, so you must integrate it into your project.

Resolution: Name-mangle the Morse Micro HostAP and all references to it in `morselib`. If you require guidance, see Appendix A.6.

6.2 Transceiver fails to boot - SPI IRQ not implemented

Symptoms: `mmwlan_boot()` returns a `mmwlan_status:MMWLAN_ERROR`.

In systems using an SPI bus to connect the host and the transceiver, the transceiver indicates data is ready to be retrieved using the `SPI_IRQ` line. If you don't implement this interrupt correctly, the transceiver fails to boot.

Resolution: Ensure the `SPI_IRQ` line has been implemented correctly.

6.3 Lower than expected data throughput

Symptoms: Data throughput is significantly lower than expected, or appears capped.

There are many factors that affect throughput, including but not limited to the performance of the CPU and memory systems, the speed of the communications bus, and the configured region and bandwidth settings of the Wi-Fi HaLow network.

To debug throughput issues we recommend you:

- compare your results to the reference platforms,
- review the output of the `porting_assistant`, particularly the `Raw TPUT (kbits/s)` result,
- use the `iPerf` example application to baseline your throughput (instead of your application),
- enable any optional tests that are present in the `porting_assistant`,
- capture a trace of the communications bus with a logic analyzer, and review for inconsistent clock behavior and large delays.

Common areas to check:

- Ensure you have allocated sufficient packet memory (`mmpktmem`) and buffers for your network stack (LwIP uses `pbufs`).
- Ensure the configuration of your communication bus is correct, including that you are using a DMA controller.
- Ensure that the settings for your DMA controller are correct and that you are controlling the GPIOs correctly.
- Ensure that your interrupt-handling routines return quickly.
- Review the “when to use the DMA” optimizations included in Morse Micro’s reference shim implementations (on some platforms it can be more efficient to send small transfers without using the DMA controller).

The following are examples of throughput issues we have debugged and resolved:

- The default linker script for a platform placed and executed code from XIP flash memory via a QSPI interface. Executing code from this flash memory was a bottleneck for system performance. Using an alternate linker definition provided by the platform vendor to move code into RAM at boot time removed the bottleneck and resulted in satisfactory throughput.
- During the port for a platform the default RAM was used at the location for the `mmpktmem` buffers. As the product evolved, these buffers were moved to a different, larger region of chip RAM but these showed worse performance. The issue was eventually diagnosed that the DMA controller was configured to expect 8-bit instead of 32-bit access, which only worked for the default RAM type. Updating the DMA configuration resulted in satisfactory throughput.

6.4 How can I debug SPI?

You can implement the R/W functions, and then hard-wire MISO to MOSI, and test that bytes transmitted over MOSI using the `write` function, can be read back over MISO via `read`.

6.5 How to optimize flash and RAM usage?

Flash and RAM usage depends heavily on the application. [Appendix A.5 – Flash and RAM requirements explained](#) explores the requirements and provides guidance on how to trim their use.

We recommend trimming flash and RAM requirements **after** you have a working port that meets your performance targets.

Glossary

AP access point.

API application programming interface.

ARM Advanced RISC machine.

AWS Amazon Web Services.

BCF board configuration file.

BSP board support package.

CMake CMake build tool.

CPU central processing unit.

DHCP dynamic host configuration protocol.

DMA direct memory access.

ECC elliptic-curve cryptography.

ELF [Executable and Linkable Format](#).

function stub a trivial function that does nothing, sometimes containing an `ASSERT(false)` call to mark it as unimplemented.

GPIO general-purpose input/output.

HAL hardware abstraction layer.

host the microcontroller unit running your application.

LMAC lower media access control.

Makefile file used by the make build tool to describe how to compile and link a program, specifying targets, dependencies, and the commands required to build them.

MCU microcontroller unit.

module a ready-to-use package containing a Morse Micro transceiver, supporting RF circuitry that has been precertified.

MQTT Message Queuing Telemetry Transport.

OS operating system.

OSAL operating system abstraction layer.

PA Porting Assistant.

PHY physical layer.

PKA public key accelerator.

platform either the hardware of a product, or that hardware together with its build system, operating system, and driver software — this guide uses both senses; context indicates which.

QoS quality of service.

RAM random access memory.

RSN robust security network.

RTOS real-time operating system.

SDIO secure digital input output.

SDK software development kit.

SoftMAC See [Linux Wireless Documentation](#).

SPI serial peripheral interface.

SSID service set identifier.

STA station.

TID traffic identifier.

TLS transport layer security.

TPUT throughput.

transceiver a wireless communication device — a Morse Micro HaLow transceiver in this guide.

UMAC upper media access control.

VIF virtual network interface.

XIP [Execute In Place](#), a method of executing programs from long-term storage without copying them to RAM.

A Appendix

When including different software components use [Table 10](#) to identify dependencies between components.

Component	Dependencies
morselib	HostAP
src/mmutils	morselib
src/mmpktmem	morselib, mmutils
src/mmconfig	morselib
src/mmregdb	morselib
src/mmipal	lwip, morselib, mmutils. Requires IP_STACK=lwip.
src/mmiperf	lwip, morselib, mmutils. Requires IP_STACK=lwip.
src/mmping	lwip, morselib, mmutils. Requires IP_STACK.
src/lwip	mbedtls (only when altcp_tls_mbedtls.c is compiled)
src/mbedtls	lwip (via library/net_lwip.c, only when IP_STACK set), morselib
src/hostap	morselib (os_mmosal.c, morse_stubs.c). MbedTLS (crypto_mbedtls_mm.c) Drivers/shim in morselib/src/umac/supPLICANT_shim bridge it to morselib.

Table 10: A table highlighting dependencies between components that are involved in porting

A.1 Building MM-IoT-SDK examples and morselib

Build the examples found in the mm-iot-sdk to identify compile information such as source files, include paths, and build flags.

The mm-iot-sdk uses a Makefile based build system and expects an Advanced RISC machine (ARM) C compiler for its STM32 based targets. Ensure Make is installed on your development machine and see [arm-none-eabi-gcc-xpack](#) for a compiler which supports the platforms in the mm-iot-sdk

A.1.1 Examples

To build an example on a specific platform, use the make command shown below.

```
# To identify source files, build defines and include paths for an
  example in the mm-iot-sdk,
# run the following
make -C mm-iot-sdk/examples/<chosen_example>/targets/ \
  <chosen_platform> VERBOSE=1
```

Listing 15: VERBOSE Build command for mm-iot-sdk example

To build the Porting Assistant, Scan, and iPerf for the mm-mm6108-ekh05 platform, see the table below.

Example	Working directory	Command
porting assistant	examples/porting_assistant	make -C targets/mm-mm6108-ekh05 -j12 VERBOSE=1
scan	examples/scan	make -C targets/mm-mm6108-ekh05 -j12 VERBOSE=1
iPerf	examples/iperf	make -C targets/mm-mm6108-ekh05 -j12 VERBOSE=1

Table 11: Build commands for example applications on mm-mm6108-ekh05

Parse the output of make VERBOSE=1 commands to extract build flags, build defines, include paths, and source files.

A.1.2 morselib

You can add the `morselib` source files to your project and compile them as part of your application. However, you may find it easier to link against the prebuilt `libmorse.a` provided in `mm-iot-sdk/framework/morselib/lib/*`. Note that the prebuilt `libmorse.a` is only available for a limited set of CPU architectures and toolchain configurations. You must use the `NewLib-Nano` implementation of the C Library. You must ensure your CPU architecture and toolchain flags match against one of the options listed below.

- `arm-cortex-m33f`
- `arm-cortex-m4f`
- `arm-cortex-m7f`

A.2 Bus transport details

The following functions are defined in `mmhal_wlan.h`. You must implement these functions and configure your communication peripheral (see Appendix A.2.1 and Appendix A.2.2) to pass the communications HAL tests.

Implement only one of SDIO or SPI.

Function	Description
<i>Common</i>	
<code>mmhal_wlan_wake_assert()</code>	Drives the WAKE GPIO high to signal the transceiver to exit sleep mode.
<code>mmhal_wlan_wake_deassert()</code>	Drives the WAKE GPIO low, allowing the transceiver to enter sleep modes.
<code>mmhal_wlan_busy_is_asserted()</code>	Returns <code>true</code> if the BUSY GPIO is currently asserted (transceiver is performing work).
<code>mmhal_wlan_register_busy_irq_handler()</code>	Registers the callback to invoke when the BUSY line is driven high.
<code>mmhal_wlan_set_busy_irq_enabled()</code>	Enables or disables the BUSY line interrupt.
<code>mmhal_wlan_assert_reset()</code>	Asserts or deasserts the RESET GPIO line.
<code>mmhal_wlan_hard_reset()</code>	Toggles the RESET line to perform a hard reset of the transceiver.
<i>SPI (implement SPI or SDIO, not both)</i>	

Function	Description
<code>mmhal_wlan_spi_cs_assert()</code>	Asserts the SPI chip-select line (drives NSS low).
<code>mmhal_wlan_spi_cs_deassert()</code>	Deasserts the SPI chip-select line (drives NSS high).
<code>mmhal_wlan_spi_rw()</code>	Transfers a single byte over SPI; returns the received byte.
<code>mmhal_wlan_spi_read_buf()</code>	Reads a buffer of bytes from the SPI bus. DMA is recommended for transfers of 16 bytes or more.
<code>mmhal_wlan_spi_write_buf()</code>	Writes a buffer of bytes to the SPI bus. DMA is recommended for transfers of 16 bytes or more.
<code>mmhal_wlan_register_spi_irq_handler()</code>	Registers the callback to invoke on the SPI IRQ line interrupt.
<code>mmhal_wlan_set_spi_irq_enabled()</code>	Enables or disables the SPI IRQ line interrupt.
<code>mmhal_wlan_spi_irq_is_asserted()</code>	Returns <code>true</code> if the SPI IRQ line is currently asserted.
<code>mmhal_wlan_clear_spi_irq()</code>	Clears the pending SPI IRQ interrupt.
<code>mmhal_wlan_send_training_seq()</code>	Send >74 clock pulses with MOSI high while CS is deasserted.
<i>SDIO (implement SPI or SDIO, not both)</i>	
<code>mmhal_wlan_sdio_startup()</code>	Initializes and enumerates the SDIO card; called once at startup.
<code>mmhal_wlan_sdio_cmd()</code>	Issues an SDIO CMD52 single-register read or write command.
<code>mmhal_wlan_sdio_cmd53_write()</code>	Performs a CMD53 block/byte-mode write to the transceiver.
<code>mmhal_wlan_sdio_cmd53_read()</code>	Performs a CMD53 block/byte-mode read from the transceiver.
<i>Firmware / BCF</i>	
<code>mmhal_wlan_read_bcf_file()</code>	Reads a region of the Board Configuration File (BCF) from non-volatile storage.

Function	Description
<code>mmhal_wlan_read_fw_file()</code>	Reads a region of the Morse Micro firmware binary from non-volatile storage.

Table 12: Mandatory `mmhal_wlan.h` functions

A.2.1 SPI

In SPI mode, the host microcontroller operates as a SDIO device over SPI as per Part 1 SD Physical Layer Specification Section 7.3 and the Morse Micro transceiver (MM6108 or MM8108) is the slave. Morse Micro operates the SPI bus in half-duplex mode. Only SPI mode 0 (CPOL=0, CPHA=0) is supported. The reference shims for support platforms using the SPI bus can be found in

`mm-iot-sdk/framework/src/platforms/*/mm_shims/mmhal_wlan.c`

Parameter	Requirement
Role	Host is SPI master; the Morse Micro transceiver is the slave.
SPI mode	Mode 0 only (CPOL=0, CPHA=0).
Bit order	MSB first.
Frame size	8-bit data frames.
Duplex	Half-duplex protocol over a full-duplex pin configuration (both MISO and MOSI must be wired).
SCK frequency	Refer to the transceiver datasheet for the supported maximum. The reference platforms run between 20 MHz (STM32U575/H753, mm-mm6108-ekh05, mm-mm8108-ekh05) and 40 MHz (STM32WB55); 20 MHz is a safe starting point for bring-up.
Chip select	Software-controlled NSS driven from a GPIO. The reference shims do not use the SPI peripheral's hardware NSS.
Power-on training	The host must be able to clock the bus with CS deasserted to issue the 74-clock training sequence required before the first transaction (see <code>mmhal_wlan.h</code>).

Parameter	Requirement
DMA	Strongly recommended. The reference shims use DMA for any transfer of 16 bytes or more (<code>DMA_TRANSFER_MIN_LENGTH</code>) and fall back to polled transfers below that. Without DMA the host will struggle to meet throughput targets and may stall the <code>morselib</code> worker thread.
Buffer alignment	Bulk read/write buffers passed through <code>mmhal_wlan_spi_read_buf</code> and <code>mmhal_wlan_spi_write_buf</code> must be 32-bit aligned.
Transfer completion signaling	The driver must be able to block efficiently on transfer completion (DMA-complete interrupt or equivalent) with a per-transfer timeout on the order of 10 ms.

Table 13: SPI peripheral requirements on the host microcontroller

A.2.2 SDIO

The host microcontroller is always the SDIO host (controller); the Morse Micro transceiver (MM6108 or MM8108) is the SDIO card. The transceiver presents two SDIO functions: function 1 is used for control with an 8-byte block size, and function 2 carries data with a 512-byte block size. Communication uses standard SDIO CMD52 (single-register access) and CMD53 (block / byte multi-access) commands. The reference shims for supported platforms using the SDIO bus can be found at `mm-iot-sdk/framework/src/platforms/*-sdio/mm_shims/mmhal_wlan.c`

Parameter	Requirement
Role	Host is SDIO controller; the Morse Micro transceiver is the SDIO card.
Bus width	4-bit wide bus is the default and recommended configuration (enabled by the <code>ENABLE_SDIO_4BIT</code> build option). 1-bit operation is supported as a fallback.
Initialization clock	Bus must be driven at 400 kHz during card enumeration (<code>SD_INIT_FREQ_HZ</code>).

Parameter	Requirement
DMA	Internal DMA (IDMA, single-buffer mode) is required for data transfers. Transfer buffers must be 32-bit word aligned and the transfer length must be a multiple of 4 bytes.
Transfer timeouts	The driver must be able to block on completion with timeouts of 750 ms for writes and 1000 ms for reads.
Card interrupts	The SDIO in-band interrupt (DAT1) must be wired and the SDMMC peripheral configured to deliver <code>SDIO_IT_SDIOIT</code> . A separate GPIO is still required for the BUSY signal – it is not carried in-band.

Table 14: SDIO peripheral requirements on the host microcontroller

As with SPI, the host must also provide the GPIO lines described above (Reset, Wake, Busy). The SDIO IRQ is delivered in-band on DAT1 by the SDMMC controller, so no additional GPIO interrupt line is needed for that purpose.

A.3 Implementing a network stack driver

When porting `mm-iot-sdk` to a platform or project which does not use `lwip` and `mmipal`, you must provide an interface driver. This driver is the bridge between your network stack and the Wi-Fi HaLow UMAC and driver provided by `morselib`. At a minimum, the network stack driver will be responsible for:

- Initializing the Morse Micro network interface;
- Passing received frames from `morselib` up to your network stack;
- Passing outbound frames from your network stack to `morselib` for transmission; and
- Reporting link state changes (link up / link down) to your network stack.

A.3.1 Initialization

Your driver's initialization function should boot the transceiver, configure the network interface, and register the callbacks described in the following subsections

1. Call `mmwlan_boot()` to boot the transceiver.
2. Call `mmwlan_get_mac_addr()` and apply it into your network stack's interface.
3. Register the receive (`mmwlan_register_rx_cb()`) and link state (`mmwlan_register_vif_state_cb()`) callbacks.

A.3.2 Link State

`morselib` calls your link state callback with `MMWLAN_LINK_UP` or `MMWLAN_LINK_DOWN` whenever the connection to the AP changes. The `mmwlan_register_vif_state_cb()` must be registered to communicate these link state changes to your network stack.

The callback signature used by `mmwlan_register_vif_state_cb()` is:

```
void link_state_cb(const struct mmwlan_vif_state *state,
                  void *arg);
```

Listing 16: Link state callback signature

These events should be translated to equivalent notifications used by your network stack. On link up, dynamic host configuration protocol (DHCP) should be restarted and the carrier marked active.

A.3.3 Receive path

When `morselib` receives a frame, it delivers it to your registered receive callback. Your callback must pass the frame to your network stack and then release the `morselib` buffer.

The callback signature used by `mmwlan_register_rx_cb()` is:

```
void rx_cb(uint8_t *header, unsigned header_len,
           uint8_t *payload, unsigned payload_len,
           void *arg);
```

Listing 17: Receive callback signature

The received frame is split into an 802.3 header (the Ethernet header) and a payload (the packet). Your interface driver is required to reassemble and deliver both to the network stack. Your callback must:

1. Allocate a buffer from your network stack large enough to hold `header_len + payload_len` bytes.
2. Copy the header and payload into the buffer.
3. Pass the buffer to your network stack's receive function.
4. Release the buffer on any error path.

If your application requires multiple virtual network interface (VIF) or a zero-copy receive via the `mmpkt` API use `mmwlan_register_rx_pkt_ext_cb` instead.

A.3.4 Transmit path

The network stack will call a transmit function registered by your interface driver. This function must copy the frame into a buffer that `morselib` can transmit.

There are two `morselib` transmit APIs. Use the one that suits your application

Function	Use when...
<code>mmwlan_tx(buf, len)</code>	Your network stack gives you a flat contiguous buffer, and a single VIF or QoS.
<code>mmwlan_tx_pkt(pkt, metadata)</code>	You need fine-grained control over QoS TID, per-VIF targeting, or zero-copy via the <code>mmpkt</code> API.

Table 15: `morselib` transmit APIs

When using `mmwlan_tx()`:

1. Read the outgoing packet from your network stack into a flat buffer.
2. Call `mmwlan_tx(buf, len)`.
3. Check the returned `mmwlan_status` and translate any error to your network stack's error type.

When using `mmwlan_tx_pkt()` with the extended `mmpkt` API:

1. Call `mmwlan_tx_wait_until_ready()` with a suitable timeout to apply back-pressure when the transceiver is busy.
2. Allocate an `mmpkt` with `mmwlan_alloc_mmpkt_for_tx()`, sizing it to the total frame length.
3. Open a view with `mmpkt_open()`, append each segment with `mmpkt_append_data()`, then close the view with `mmpkt_close()`.
4. Populate a `mmwlan_tx_metadata` structure with the TID and VIF
5. Call `mmwlan_tx_pkt()`.

A.4 Converting .bin to .mbin

The `mm-iot-sdk` supplies a script, `convert-bin-to-mbin.py`, to perform this conversion. For example, the Quectel FGH100MH high power module has a BCF distributed by Morse Micro at [Morse Micro - morse-firmware](#). Convert it to an `.mbin` with `./convert-bin-to-mbin.py bcf_fgh100mhaamd.bin`.

A.5 Flash and RAM requirements explained

The `mm-iot-sdk` recommends 2 megabyte of flash and 256 kilobytes of RAM. These figures are very general, and the space required will vary depending on your application, CPU architecture and platform. Appendix A.5.1 and Appendix A.5.2 were written using an ARM Cortex-M33 CPU core for getting flash and RAM estimates.

This section provides more detail on the flash and RAM requirements, using the `aws_iot` example application as a basis for comparison.

A.5.1 Flash requirements explored

The main requirements for flash storage in `mm-iot-sdk` based applications are:

- Storing the transceiver's firmware and BCF,
- Storing any persistent configuration (for example the `mmconfig` component), and
- Storing your application's code,

Firmware and BCF

As of SDK version 2.11.2, the transceiver firmware and BCF together consume approximately 400 kilobytes of space.

Persistent configuration

The examples in `mm-iot-sdk` use 16 kilobytes of space for persistent configuration. Your application may need more or less than this.

Application code size

The code size of your application depends on many factors including the amount of functionality included, the compile-time optimization settings, and the choice of debug settings.

The `aws_iot` example is a large application that uses Message Queuing Telemetry Transport (MQTT) to communicate with an Amazon Web Services (AWS) service over transport layer security (TLS).

The following table shows the flash size of building `aws_iot` for different supported platforms, with different options:

#	Command	Flash	Comments
1.	<code>make -C targets/mm-mm6108-ekh05 \</code> <code>BUILDTYPE=production</code>	908 KB	390 KB for transceiver firmware, 518 KB for application.

#	Command	Flash	Comments
2.	<code>make \</code> <code>-C targets/mm-mm6108-ekh05-sdio \</code> <code>BUILDTYPE=production</code>	911 KB	SDIO 3 KB larger than SPI build in #1.
3.	<code>make -C targets/mm-mm8108-ekh05 \</code> <code>BUILDTYPE=production</code>	894 KB	377 KB for MM8108 firmware, 517 KB for application. MM8108 firmware smaller than MM6108.
4.	<code>make -C targets/mm-mm6108-ekh05 \</code> <code>BUILDTYPE=debug</code>	973 KB	390 KB for transceiver firmware, 583 KB for application. 65 KB larger than #1 build, but still using optimized <code>morselib.a</code> .
5.	<code>make -C targets/mm-mm6108-ekh05 \</code> <code>BUILDTYPE=debug \</code> <code>BUILD_MORSELIB_FROM_SOURCE=y</code>	1018 KB	390 KB for transceiver firmware, 628 KB for application. 45 KB larger than #4 build using <code>morselib.a</code> .
6.	<code>make -C targets/mm-mm6108-ekh05 \</code> <code>BUILDTYPE=debug \</code> <code>BUILD_MORSELIB_FROM_SOURCE=y</code> and (specify in Makefile) <code>BUILD_DEFINES+=MMLOG_LEVEL_OVRD=7</code> <code>BUILD_DEFINES+=AWS_LOG_LEVEL=3</code>	1104 KB	390 KB for transceiver firmware, 714 KB for application. 86 KB larger than #5 due to log levels.

Table 16: Flash usage based on platform and build options

If your host processor cannot be changed, there are two common workarounds for coping with the flash storage requirement of the transceiver firmware and BCF:

- **Use external flash** — Store the transceiver firmware and BCF on an external flash device (for example, a QSPI flash), reducing the on-chip flash required on the host processor.
- **Compress the firmware and BCF** — Store a compressed copy in flash and decompress on boot. This reduces flash usage at the cost of added port complexity and a longer boot time, since the host must decompress the images before delivering them to the transceiver.

A.5.2 RAM requirements explored

The main requirements for RAM storage in `mm-iot-sdk` based applications are:

- Stack and heap space for your running program, and
- Data memory for networking functions, such as `mmpktmem`.

The `mm-iot-sdk` recommends greater than 256 kilobytes of RAM for your porting platform so

that most `mm-iot-sdk` examples can be run on your platform. Table 17 provides details about RAM usage for a small group of applications.

#	App	RAM
1.	porting_assistant	102 KB
2.	scan	218 KB
3.	iperf	254 KB
4.	aws_iot	280 KB

Table 17: RAM usage per App with default build options

For a fully featured application like `aws_iot`, the largest memory sections are the following:

- **Operating system heap** - approximately 120 kilobytes,
- **mmpkmem packet buffers** - approximately 111 kilobytes, and
- **lwip packet buffers** - approximately 36 kilobytes.

In total, this is 267 kilobytes used in the default build configuration. Expect to see similar sizes in your own application.

The operating system and `mmpkmem` APIs provide flexibility to change total RAM usage. Options to change RAM usage include:

1. Packet memory pool - tune `MMPKMEM_{TX,RX}_POOL_N_BLOCKS` to reduce memory usage. Most `mm-iot-sdk` examples default to 32 blocks. Switch between the static and heap-based implementations to change RAM usage. **Note:** The amount of RAM used for transmit and receive packet buffers (`mmpkmem`) is configurable, but directly affects throughput performance.
2. Trim operating system heap.
3. Cryptographic accelerators - use hardware accelerators for cryptographic operations to reduce RAM usage.

A.6 Name mangling HostAP and morselib

This is a general overview of the process needed to mangle `hostap` and `morselib` symbols. It will vary depending on how your build system works, and which features of `hostap` and `morselib` you are using. Follow these steps:

1. Generate a list of all conflicting symbols and collate them into a single text file. One symbol per line, a single newline character separating each line.
2. Put an appropriate rename for each symbol on the same line, separated by a space character, to form a symbol pair. See the symbol file snippet below, we recommend prefixing all conflicting symbols with `"mm_"`

3. After compiling, but before linking, call `obj-copy` on `hostap` and `morselib` object files, as outlined below.
4. Link against the new redefined object files.

```
hexstr2bin mm_hexstr2bin
hmac_sha256_kdf mm_hmac_sha256_kdf
hw_get_freq mm_hw_get_freq
hostapd_acl_comp mm_hostapd_acl_comp
```

Listing 18: Symbol pairs snippet from a text file

```
arm-none-eabi-objcopy \
  --redefine-syms=<SYMBOL_RENAME_FILE> \
  <OBJ_FILES>
```

Listing 19: Command to redefine symbols

A.7 Porting Stubs Repository

We have created a porting-stubs repository which contains the following:

- A similar folder structure to what is recommended in [Figure 4](#).
- The three applications you will use for testing your port; porting assistant, scan and iperf.
- A `makefile` based build system which integrates the `mm-iot-sdk` using the `MMIOT_ROOT` variable.
- A platform directory with stubbed `mmhal_wlan*` and `mmosal` files for you to complete.

The files in this repository have empty implementations for all of the OSAL and HAL functions, so they can be compiled, linked and loaded onto your platform. However, since they are empty, the application will not run until you provide your port. The purpose of the porting-stubs repository is to supplement the information in this guide. We recommend you either download the stub repository and begin your own project inside it, or use it for guidance on how to create your own application project.

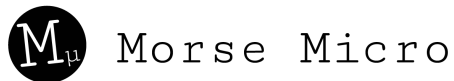
The repository is available on our Github page: [Morse Micro: porting-stubs](#)

Revision History

Release Number	Release Date	Release Notes
01	30-Jun-2026	Initial release

Table 18: Revision history

Morse Micro provides this information “as is” without warranties of any kind, express or implied. No guarantee is made as to the accuracy, completeness, or suitability of this information or Morse Micro's products for any specific purpose. Use of this information and products is at the user's sole risk. Morse Micro products are not designed or tested for use in mission-critical systems, and should not be used in such applications. Performance specifications are based on internal testing and are believed to be reliable; however, they are not guaranteed. It is the Buyer's responsibility to test and validate all product performance, compatibility, and compliance, both in isolation and within end applications. Morse Micro assumes no liability for the use or application of any product, circuit, or information described herein. No license or other rights—express or implied—are granted under Morse Micro's intellectual property. This document contains proprietary information of Morse Micro and is subject to change without notice. Wi-Fi®, Wi-Fi HaLow™, and the Wi-Fi logo are trademarks of Wi-Fi Alliance. ZigBee™ and Z-Wave™ are trademarks of their respective owners. All other trademarks are the property of their respective owners.



Morse Micro Pty. Ltd. Corporate Headquarters

Level 8, 10-14 Waterloo Street, Surry Hills, NSW 2010, AUSTRALIA

Email: sales@morsemicro.com

Copyright © Morse Micro Pty. Ltd.