# Marvell Wireless Microcontroller

Internals

**User Guide**

Not Approved by Document Control.

August 21, 2015

**Marvell.Moving Forward Faster**

## Document Conventions

| | |
|---|---|
| | **Note:** Provides related information or information of special importance. |
| Caution | **Caution:** Indicates potential damage to hardware or software, or loss of data. |
| | **Warning:** Indicates a risk of personal injury. |

For more information, visit our website at: www.marvell.com

| Date | Revision | Changelog |
|---|---|---|
| Apr 20 2015 | V1.0 | First Version |

# Table of Contents

**1 Overview ........ 6**

**2 Flash Layout ........ 7**

2.1 Changing the Flash Layout ........ 8

2.1.1 Upgrade Buddies ........ 10

**3 Firmware Image Layout ........ 11**

3.1 88MW30X ........ 11

3.2 88MC200 ........ 11

3.3 Firmware Regions ........ 12

**4 Boot up ........ 14**

**5 XIP ........ 15**

5.1.1 Architecture ........ 15

5.1.2 Performance Impact ........ 15

**6 Secure Boot ........ 17**

6.1 88MC200 ........ 17

6.1.1 Recovering a JTAG-disabled module ........ 17

6.2 88MW300 ........ 17

6.3 Building Secure Boot Images ........ 19

6.4 Programming the OTP ........ 19

**7 Wi-Fi Features ........ 21**

7.1 Wi-Fi Region Configuration ........ 21

7.1.1 Specifying Country Code ........ 21

7.1.2 Using 802.11D ........ 21

**8 Application Framework ........ 22**

8.1 Micro-AP Interface ........ 22

8.2 Station Interface ........ 22

8.2.1 Starting Services ........ 22

**9 Network Provisioning Options ........ 24**

9.1 Marvell Provisioning ........ 24

9.2 EZConnect Provisioning ........ 24

9.3 WAC (Wireless Accessory Configuration) ........ 24

9.4 WPS (Wi-Fi Protected Setup) ........ 25

**10 Power Management ........ 26**

10.1 Microcontroller Power Save ........ 26

10.1.1 Controlling Power Save ........ 26

10.2 Wi-Fi Power Save ........ 27

10.2.1 Controlling Power-Save ........ 27

**11 Cloud Communication ........ 28**

11.1 Supported Cloud Agents ........ 28
11.1.1 Arrayent ........ 28
11.1.2 Xively ........ 28
11.1.3 Evrythng ........ 28

11.2 Communicating with other clouds ........ 28
11.2.1 HTTP Client and TLS ........ 29
11.2.2 BSD Socket API ........ 29
11.2.3 JSON/XML ........ 30

**12 Reference Phone Apps ........ 31**

**13 Tools ........ 32**

13.1 Flashprog ........ 32
13.1.1 A Word about OpenOCD ........ 32
13.1.2 Using Flashprog ........ 32

13.2 UART boot ........ 35
13.2.1 Building ........ 35
13.2.2 Using ........ 35
13.2.3 Fixing the linker script for 88MW30X ........ 36

13.3 footprint.pl ........ 36
13.3.1 Looking at Free Space ........ 36
13.3.2 Looking at Footprint Contributions ........ 37
13.3.3 Disabling LTO Option ........ 39

**14 Glossary ........ 41**

# 1 Overview

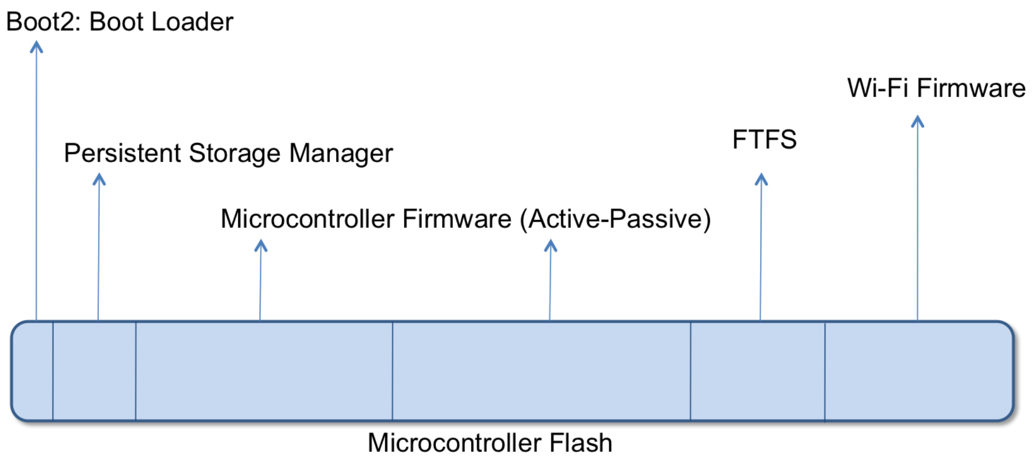
Marvell’s Wireless Microcontroller Software Development Kit (WMSDK) enables the development of custom firmware images that can be run on the processor built-into Marvell’s 88MW300 Single Chip WLAN Microcontroller as well as the 88MC200 System-on-Chip (SoC).

This software/hardware platform can be used to realize complex applications that can run on low-cost and low-power consuming hardware.

The SDK and the corresponding documentation addresses all the products phases from hardware design, software development to all the manufacturing and certification related requirements.

# 2 Flash Layout

Typical flash contents of most devices are shown below:


Boot2: Boot Loader
Wi-Fi Firmware
FTFS
Persistent Storage Manager
Microcontroller Firmware (Active-Passive)
Microcontroller Flash

1. **Boot2**: This is the boot-loader. The SDK generates a *boot2.bin* image that acts as the boot-loader. The primary responsibility of the boot-loader is to identify the active firmware partition, and load it into the RAM. Once the image is loaded, the boot-loader jumps to the entry point of the firmware image.

   If Execute-in-Place (XIP) is being used, then the boot-loader directly jumps to the entry point of the firmware image within flash. Only the data (read/write) and bss sections are loaded to RAM.

2. **Manufacturing Configuration**: This is an optional partition. This partition could hold any manufacturing time data that should NOT be erased across a *Reset to Factory* action.
3. **Persistent Configuration**: This partition holds any configuration data that should be maintained persistently. Developers are free to store their own data in this partition.
4. **Microcontroller Firmware**: This is the firmware image that primarily runs on microcontroller. When the developer builds any accessory, this is the firmware image that is generated. For example, on building the wm_demo, the SDK generates a *wm_demo.bin* file that should be stored in here.

   As can be noticed, two partitions are present for the microcontroller Firmware. This is to facilitate firmware upgrades functionality. Only one of these partitions is active at a time.

   During boot-up the microcontroller Firmware downloads the Wi-Fi firmware into the Wi-Fi chipset in order to initialize the Wi-Fi chipset.

5. **FTFS Filesystem**: Some application developers may also choose to include a FTFS filesystem partition within the flash. This partition will contain any web-apps, or files that the Web Server running on the microcontroller can serve to browsers.

   The SDK also includes support for upgrading the FTFS filesystem. If desired, developers can also create two partitions, active-passive for the FTFS upgrades.

6. **Wi-Fi Firmware**: This is the Wi-Fi firmware that is loaded into the Wi-Fi chip on boot-up. This firmware cannot be customized. It is available in a binary format along with the SDK.

   The SDK also includes support for upgrading the Wi-Fi firmware. If such a feature is desired, developers can create two partitions, active-passive for the Wi-Fi Firmware as well, as has been done for the microcontroller firmware.

**Default flash layout for MW300:**

Layout to be written to flash is stored in the file *wmsdk/tools/OpenOCD/mw300/layout.txt.*. The default flash layout is: (Note that there is no in-package flash in MW300)

| Component | Address | Size | Flash |
|---|---|---|---|
| Boot2 | 0x0 | 0x6000 | External QSPI |
| PSM | 0x6000 | 0x4000 | External QSPI |
| MCU Firmware | 0xa000 | 0x58000 | External QSPI |
| MCU Firmware | 0x62000 | 0x58000 | External QSPI |
| Filesystem | 0xba000 | 0x30000 | External QSPI |
| Filesystem | 0xea000 | 0x30000 | External QSPI |
| WLAN Firmware | 0x11a000 | 0x49000 | External QSPI |
| WLAN Firmware | 0x163000 | 0x49000 | External QSPI |

If you wish to create a different flash layout, please refer to section **Error! Reference source not found.** for instructions on creating the specifications of the new layout.

The layout can be written to flash using the '-l' option of the flashprog utility. This can be done as:

```
# cd wmsdk/tools/OpenOCD
# ./flashprog.sh -l new_layout.txt
```

**Default flash layout for MC200 :**

Layout to be written to flash is stored in the file *wmsdk/tools/OpenOCD/mc200/layout.txt.*. The default flash layout is:

| Component | Address | Size | Flash |
|---|---|---|---|
| Boot2 | 0x0 | 0x3000 | Internal |
| PSM | 0x3000 | 0x4000 | Internal |
| MCU Firmware | 0x7000 | 0x58000 | Internal |
| MCU Firmware | 0x5f000 | 0x58000 | Internal |
| WLAN Firmware | 0xb7000 | 0x49000 | Internal |
| Filesystem | 0x0 | 0x20000 | External |
| Filesystem | 0x20000 | 0x20000 | External |
| WLAN Firmware | 0x40000 | 0x49000 | External |

The default layout assumes the presence of the external flash.

## 2.1 Changing the Flash Layout

In a system built from WMSDK, the following components reside on the flash:

1. Boot2: boot-loader
2. PSM: Persistent Storage Manager
3. Firmware Images (potentially two, for active-passive upgrades)
4. FTFS Images (potentially two, for active-passive upgrades)

5. WLAN Firmware

Applications have a flexibility to rearrange some of these components in flash. The flash layout can be changed using a layout configuration file. A sample layout configuration file is as shown below:

```
# Component         Address     Size     device  Name[8]
FC_COMP_BOOT2       0x0         0x3000    0        boot2
FC_COMP_PSM         0x3000      0x4000    0        psm
FC_COMP_FW          0x7000      0x58000   0        mcufw
FC_COMP_FW          0x5f000     0x58000   0        mcufw
FC_COMP_WLAN_FW     0xb7000     0x49000   0        wififw
FC_COMP_FTFS        0x0         0x20000   1        ftfs
FC_COMP_FTFS        0x20000     0x20000   1        ftfs
FC_COMP_WLAN_FW     0x40000     0x49000   1        wififw
```

The file consists of 5 columns.

1. **Component Names**: The following components are available
   a. **FC_COMP_FW**: This component identifies the firmware image. There could potentially be two firmware images for active-passive upgrades. Note that if only one of these components is present, then the firmware upgrades feature will not be available.
   b. **FC_COMP_FTFS**: This component identifies the filesystem image. The filesystem image can be active-passive upgradeable.
   c. **FC_COMP_WLAN_FW**: This component identifies the WLAN firmware. This firmware is loaded into the Wi-Fi card on bootup.
   d. **FC_COMP_BOOT2**: This component identifies the boot2 boot-loader image that is booted up by the bootROM. The location and size of this component is fixed and should not be changed.
   e. **FC_COMP_PSM**: This component identifies the PSM partition where persistent configuration information is stored.
2. **Address**: The start address in flash for this component. This should always be aligned to a 4K boundary.
3. **Size**: The size of the component from above mentioned start address. This should always be a multiple of 4K (since 4K is the erase size of the flash).
4. **Device**: The flash device on which this component is available. As of now, 0 stands for the internal flash and 1 stands for the external flash. On boards that do not have the external flash interfaced with MW300/MC200, the id 1 should not be used.

   The Hi-Flying module HF-LPB200 uses an AT25 external flash. Please use flash id 2 to refer to this flash.
5. **Name**: A friendly name that identifies the component. This should be of a maximum 8 character long. Another important role that the name plays is to identify *upgrade-buddies*. Please refer to the following subsection for more details.

Once the flash layout configuration file is written, the new layout can be written to flash using the '-l' option of the flashprog utility. This can be done as:

```
# cd wmsdk/tools/OpenOCD
# ./flashprog.sh –l new_layout.txt
```

**Note** Writing a new flash layout to flash erases the entire contents of the flash.

### 2.1.1 Upgrade Buddies

Some components in the layout like firmware and the filesystem can have active-passive upgradeable components. Using the same name for these components makes these components upgrade buddies. Note that only two components can share the same friendly name. Also, by definition, the components that share the same name should have the same component type.

# 3 Firmware Image Layout

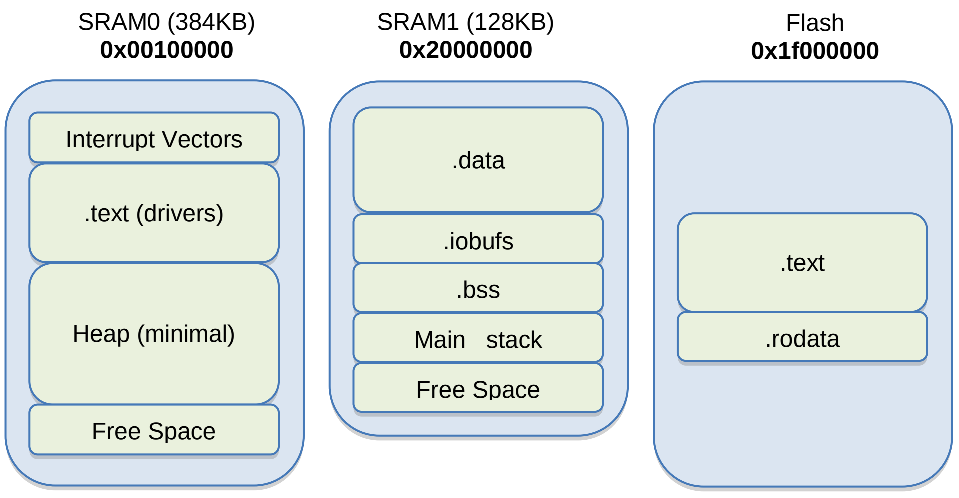
The SDK packs a large bunch of middleware and cryptographic components in the microcontroller's 512KB of RAM.

## 3.1 88MW30X

The 88MW30X's SRAM/memory layout is as shown below:

| Location | Address | Size |
|---|---|---|
| SRAM0 | 0x00100000 | 384KB |
| SRAM1 | 0x20000000 | 128KB |
| ROM | 0x0 (Following the BootROM) | ~80KB |
| FLASH | 0x1f000000 | Depends on Board Design |

Every application contains a linker script, mw300.ld, which instructs the linker how to place the different sections of the generated code into the SRAM regions. By default, the layout looks as follows:

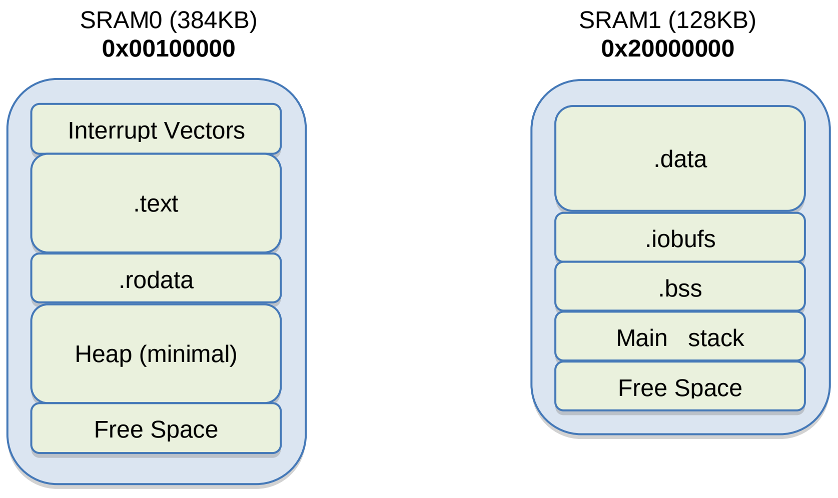


## 3.2 88MC200

The 88MC200's SRAM layout is as shown below:

| Location | Address | Size |
|---|---|---|
| SRAM0 | 0x00100000 | 384KB |
| SRAM1 | 0x20000000 | 128KB |

Every application contains a linker script, mc200.ld, which instructs the linker how to place the different sections of the generated code into the SRAM regions. By default, the layout looks as follows:



## 3.3 Firmware Regions

An explanation of the various regions is as follows:

1. **Interrupt Vectors**: This is the interrupt vector table that instructs the microcontroller where to find the corresponding interrupt service routines.
2. **.text**: This is the text section of the accessory. This includes all the executable code that is a part of the microcontroller firmware.
3. **.rodata**: This is the read-only data of the accessory. Any read-only strings, initializations or other read-only data goes in this section.
4. **Heap (minimal)**: This is the heap region. All dynamic allocations happen from this heap. The stack space of the various threads in the system is also allocated from this heap region.

   The only exception to this is the network stack. The LWIP network stack uses its own heap allocator and its own heap space to dynamically allocate memory. This network stack’s heap space is accounted for in the ‘bss’ section for accounting purposes.

   The heap automatically expands to absorb any leftover free space in SRAM0.
5. **Free Space (SRAM0)**: Any leftover space is the free space that is left in SRAM0. This indicates that the accessory firmware has still as much room to fit in any additional .text or .rodata sections.

   As mentioned above, the free space from SRAM0 is automatically absorbed into the heap of the system.
6. **.data**: This is the data section of the microcontroller firmware. Any initialized read/write data goes in this section.
7. **.iobufs**: These are I/O buffers that are used by the SDIO driver to transmit/receive data from the Wi-Fi chipset. Since these buffers are DMA-able, it is mandatory that these stay in the SRAM1 region.
8. **.bss**: This is the bss, or the uninitialized data section. This section is all set to zero at the start of bootup.
9. **main_stack**: This is a stack space that is used by all the interrupt handlers when they are executing.
10. **Free Space (SRAM1)**: This is any leftover space that is left in SRAM1.

If you wish to look at the footprint contributions of your firmware, please refer to Section 13.3. This talks about the usage of the footprint.pl tool.

# 4 Boot up

The typical boot-up steps are as follows:

1. Execution begins in the **BootROM** of the microcontroller, which hands over control to the boot2 boot-loader, after loading it into the RAM.
    a. If secure-boot is enabled:
        i. if boot2 is signed, it will verify that the boot2 has a valid signature
        ii. if boot2 is encrypted, it will decrypt the boot2 and then load it to RAM
2. The **boot2** boot-loader looks at both the microcontroller firmware partitions and determines which one is the Active partition. Once detected, the boot-loader loads this image into the RAM and hands over control to this firmware.
    a. If Execute-in-Place (XIP) is enabled, the boot2 will directly jump to entry point of the firmware in the flash. Only the data (read-write) and bss sections are loaded into the RAM.
    b. If secure-boot is being used:
        i. if microcontroller firmware is signed, it will verify that the firmware has a valid signature
        ii. if microcontroller firmware is encrypted, it will first decrypt the firmware before loading it to RAM
3. The **Microcontroller Firmware** is the accessory firmware that executes on the microcontroller.
4. The microcontroller firmware reads the **Wi-Fi Firmware** from the flash and loads it into the Wi-Fi chip.
5. The microcontroller firmware looks up the **Persistent Configuration** to determine whether the home network configuration is present or not. If home network is configured the accessory initiates the connection with the home network. If no home network configuration is found, the accessory starts the Wireless Accessory Configuration module.

# 5 XIP

The MW300 supports eXecute-In-Place (XIP). With this feature, the micro-controller can directly execute code from flash, without having to load it entirely into the RAM first. With this feature enabled, applications get much larger room for:

- code (text section): since it can be stored in as large a flash as possible
- as well as data (bss, data sections): larger free RAM is available, since the code isn't getting loaded into the RAM

By default, the WMSDK builds images with XIP disabled. You can enable XIP by simply passing the XIP=1 option to your build command.

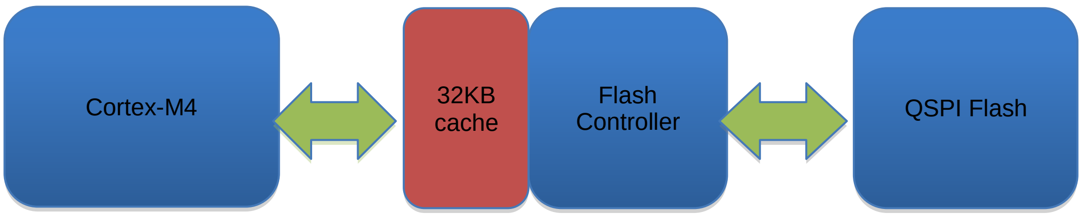
```
# make XIP=1
```

**Note**

Firmware images generated with XIP cannot be loaded to RAM using *ramload.sh*. This is because XIP implies that the parts of the firmware will be directly read from the flash.

### 5.1.1 Architecture

The following is the architectural block diagram that facilitates XIP:



The addresses referenced by the microcontroller can be directly read from the QSPI Flash by using a flash controller. A 32-KB cache ensures that the penalty for flash read is offset because of cached contents. The cache is an 8-way set associative cache with a 32-byte cache line.

This allows the microcontroller to directly execute instructions from the flash, or read read-only data from the flash.

### 5.1.2 Performance Impact

The following experiment was performed to quickly assess the usefulness of the XIP feature and the cache controller.

1. An application is written that creates a HTTP WSGI handler and is executed on the 88MW300.
2. The cache is wiped out to get cold cache readings.
3. From a host that is on the same network an HTTP request is executed for this WSGI handler. Such a request will go through multiple modules, for example: SDIO driver, Wi-Fi driver, TCP/IP network stack, HTTP Web Server and finally to the Application's WSGI handler.

The following measurements were taken:

1. Average cache miss penalty: 1.7 us
2. Total cache miss count: 1435

Firstly, this indicates that the path traversed by this request loads references a sizeable chunk of code/data (1435 misses x 32-bytes = ~45KB). This indicates that the working set for our example here is larger than the cache size.

Also note that the 32-KB cache belongs to the same class as the RAM. So the access latencies to the cache are equivalent to that of the RAM. This implies that the XIP execution in this experiment results in about a delay of 2300us (1435 misses x 1.7us), as compared to a fully in-RAM application.

# 6 Secure Boot

## 6.1 88MC200

When secure boot is enabled on the 88MC200:

- JTAG access to the microcontroller is disabled. This ensures that nobody can execute any other code on the microcontroller, or read the contents of the internal flash
- Password protects the UART boot option. Grounding the GPIO 27 pin on reset can enable the UART boot option. This can be used for debug and diagnostic purposes. The secure boot reinforces the UART boot with password protection. Only the correct password will enable someone to boot via UART. Given that the UART boot process runs at a 9600-baud rate, this will take an attacker a long time to brute force the password.

### 6.1.1 Recovering a JTAG-disabled module

This section talks about how to recover a board/module that has security enabled on it. The primary way is to use the UART boot feature. This is done as follows:

1. Start with a host that is configured to act as a development host (Please refer *Development Host Setup Guide*).
2. Ensure the pin GPIO27 is pulled low on power-on
3. Connect your board to your development host using the USB cable.
4. Note the virtual COM port that is created for the board.
5. Build the SDK, and then run the following commands:

```
$ cd wmsdk_bundle-x.y.z/
$ make tools_install
$ ./wmsdk/tools/src/uartboot/uartboot –p /path/to/ttyUSB -e
```

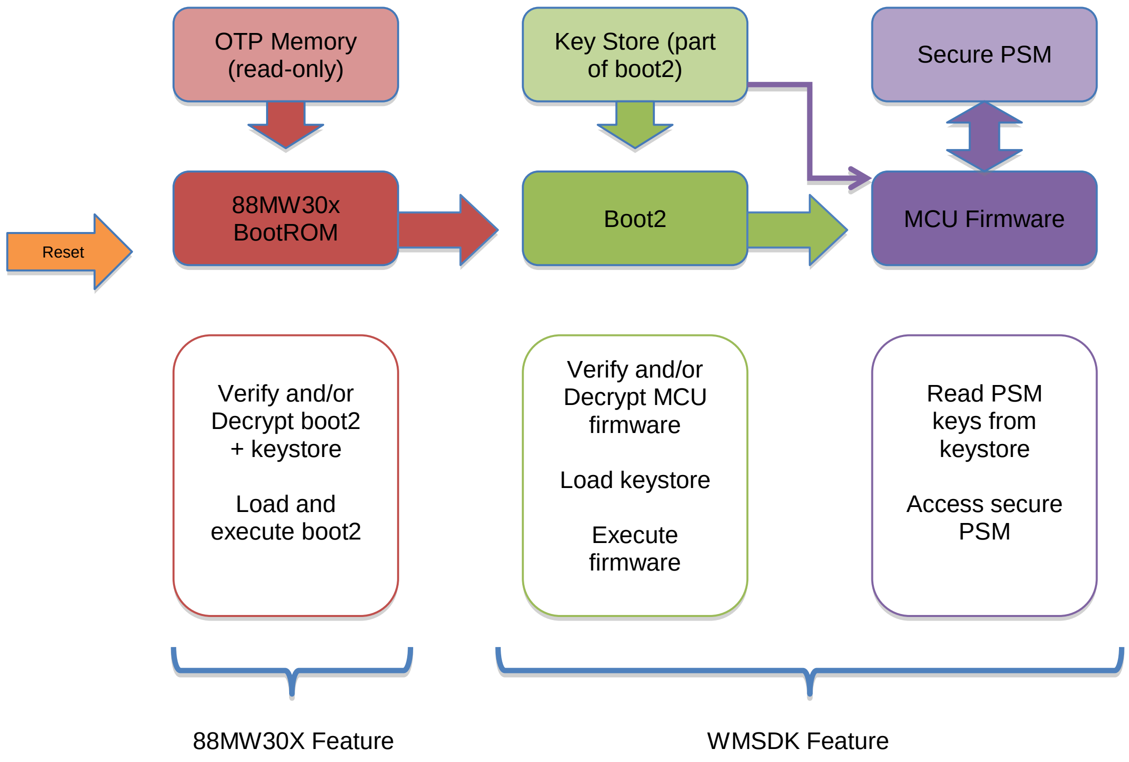
6. This will erase the entire flash of the board. Reset the board and you can now use JTAG to reprogram the board.

## 6.2 88MW300

The secure boot feature allows you any or all of the following:

- Ensure that only trusted code is executed on the microcontroller by:
  - Disabling JTAG access to the microcontroller
  - Disabling non-flash boot options
  - Ensuring that only signed and/or encrypted images are booted from the flash
- Securely store data in the PSM (Persistent Storage Manager) that resides on the external flash

A quick snapshot of the secure boot process supported by the 88MW30x and the SDK is shown below:



As can be seen a chain of trust is established. A few quick words about the secure boot process:

1. OTP: The OTP is a one-time programmable memory. This can be programmed at the time of manufacturing with secure boot configuration settings.
2. BootROM: It looks for the secure boot configuration in the OTP. It looks for the following entities:
    a. Disable JTAG
    b. Disable non-flash Boot
    c. Is the boot2 image signed, if so what is the verification key? Currently **RSA-2048** is supported,
    d. Is the boot2 image encrypted, if so what is the decryption key? Currently **AES-CCM-256** is supported
3. KeyStore: The key store is a secure storage area that contains all the other keys that are used in the system. The keystore is appended to the boot2 image, so that it can be verified and decrypted by the BootROM. The entire keystore is also made available to the microcontroller firmware. Application developers can store any keys that should be securely accessed by the microcontroller in the keystore.
4. Boot2: The boot2 boot-loader looks at the keystore to determine the security configuration options used for the microcontroller firmware. It looks for the following entities:
    a. Is the MCU firmware signed, if so what is the verification key? Currently **RSA-2048** is supported.
    b. Is the MCU firmware encrypted, if so what is the decryption key? Currently **AES-CTR-128** is supported.
5. MCU Firmware: The MCU firmware currently reads the encryption key, to be used for secure PSM, from the keystore. Currently AES-CTR-128 is supported.

## 6.3 Building Secure Boot Images

Secure boot images can be generated by passing the SECURE_BOOT option to the build command line. The easiest way to build secure boot images is to run the following command:

```
# make SECURE_BOOT=1
```

This will generate the following:

1. An encrypted boot2 image, boot2.ke.bin
2. A signed MCU firmware image, <app_name>.s.bin
3. An OTP programming application that can program the OTP with keys and configuration option that match the generated images.

The secure boot images of boot2 and the MCU firmware can now be programmed to the flash.

**Note**

There are multiple different secure boot configurations that are possible. For example, should bootloader / mcu firmware be encrypted or signed, should JTAG be disabled, should non-flash boot options be enabled etc. Please refer to the *Application Note: Secure Boot* for more details about using these combinations.

## 6.4 Programming the OTP

**Note**

Note that once OTP is programmed it cannot be changed ever again. Please make this change with caution.

If your secure boot configuration is such that it disables JTAG, it will no longer be possible to flash or load images using JTAG.

The OTP programming application that is generated above can be used to program the OTP. This can be done as follows:

```
# cd wmsdk_bundle-x.y.z/bin/wmsdk/tools/OpenOCD
# sudo ./ramload.sh /path/to/otp_prog.axf
```

Once the OTP Programming application is loaded, go to the serial console of the device (UART0/115200/8N1). Once the application is running it will print the following message:

```
[otp_app] OTP Programming Application v0.x Started
```

Once this is seen please run the following command on the console:

```
$ otp_write
```

This will program the OTP memory of the 88MW30x.

**Note**

Once they keys are written to the OTP memory they will remain there forever. Please ensure to backup the security configuration directory, *wmsdk_bundle-x.y.z/sboot_conf/*. The security configuration file will be used to build the

| |
|---|
| encrypted/signed images corresponding to the keys stored in the OTP. |

# 7 Wi-Fi Features

## 7.1 Wi-Fi Region Configuration

By default, the SDK builds application firmware that is compliant with the US region configuration. This implies that the module obeys the US regulations for Wi-Fi transmissions on certified frequency bands. The SDK provides a mechanism for configuring various region codes in the application firmware. This can be performed in one of the following two ways:

### 7.1.1 Specifying Country Code

In this method of configuration, the application firmware defines up-front  the country code that the device is going to be deployed in. Once configured the device’s Wi-Fi firmware obeys the configured countries regulations. This configuration can be set by making a call to the *wlan_set_country()* API. For example:

```
wlan_set_country(COUNTRY_CN);
```

### 7.1.2 Using 802.11D

**Note**: The FCC does not allow the use of 802.11D in the US from Jan 1, 2015.

In this method of configuration, the Wi-Fi driver of the application firmware will scan for Access Points in the vicinity and accordingly configure itself to operate in the available frequency bands. This configuration can be set by making a call to the *wlan_11d_enable()* API. Note that this API should be called after *wlan_init()* but before starting any station or micro-AP interface. For example:

```
wlan_11d_enable();
```

# 8 Application Framework

The application framework provides convenience wrappers to manage the common use cases. The application framework is so designed that it offers the application developers maximum flexibility at various steps, yet keeping the footprint within limits. Applications that wish to use the application framework can invoke the application framework with the app_framework_start() function and register an event callback with the same. The application framework keeps the application informed of its various states through this handler.

## 8.1 Micro-AP Interface

Applications may start the micro-AP interface using the app_uap_start_with_dhcp() call. It is up to the applications to determine what services should be started on this interface.

## 8.2 Station Interface

The station interface in the application framework (AF) can be in one of the two modes: un-configured or configured. In the configured mode the settings of the wireless network (AP) that the station connects to are available in the device, while in the un-configured mode they are not.

In the *un-configured* mode, the applications can choose one of three alternatives based on their needs. These options are:

1. Use one of WMSDK's provisioning services:
    a. Use Marvell Provisioning by calling app_provisioning_start(). This should be started as a service once the micro-AP interface has been started. The provisioning service starts some web services that can be used by clients.Web clients can use the */sys/scan* service to scan for the available networks and later use */sys/network* service to configure the desired network.
    b. Use EZConnect Provisioning by calling app_ezconnect_provisioning_start(). This provisioning method does not require a micro-AP interface to be up. It acts as a Wi-Fi sniffer instead to sniff packets over the network.
2. If the applications wish to configure the station interface over the micro-AP interface with a non-Marvell provisioning protocol they may first start the micro-AP and then decide what services should be started on this network and how should network configuration information be provided to the device. Once the application receives the configuration information it can inform the AF of the same using app_configure_network() call.
3. Use a network-independent provisioning mechanism. In this mechanism the application can use any of the associated peripherals like a keyboard or touch-screen UI to accept network configuration information from the user. Once this information is received applications can inform the AF of the same using app_configure_network()

All the 3 options mentioned above finally take the station interface into the configured state. Once the network configuration information is available, the AF will remember this setting. On subsequent bootup, if the application calls app_sta_start(), the AF will try to make connection attempts to this configured network.

Applications can decide what services should be running in station mode as desired.

### 8.2.1 Starting Services

The AF provides convenience wrapper functions for the following services:

1. app_httpd_: Provides common httpd services
2. app_mdns_: Provides common mDNS services
3. app_sys_register_: Provides common HTTP services like upgrades and diagnostics

For more information about application framework please refer to documentation of app_framework.h from WMSDK reference manual.

# 9 Network Provisioning Options

When the WMSDK device acts as a Wi-Fi station interface, it connects to a Wi-Fi network in the vicinity. In case of devices targeting the home users, this could be the user's home Wi-Fi network. Provisioning is the process performed by the end user. At the end of this process, the WMSDK based devices are configured with the end-customer's Wi-Fi network settings. The WMSDK supports all the following network provisioning methods. Any of these methods can co-exist with the other provisioning mechanisms.

## 9.1 Marvell Provisioning

In this provisioning mechanism the WMSDK devices initially boot up in micro AP mode.. Customers then connect to this micro-AP and access the web UI served by the on device HTTP server. This webUI guides the customers through the simple process of configuring WMSDK device for the target Wi-Fi network. As an alternative to the webUI, the provisioning can also be done using native Android/iPhone apps. Fully functional reference applications for Android and iOS are shipped along with the WMSDK bundle.

Security of the provisioning process can be guaranteed by starting the micro-AP network with WPA2 security. This will ensure that only the user with the correct passphrase for the micro-AP network (probably printed on the device's box or label) can provision the device. And also it ensures over-the-air security of the network credentials being exchanged.

The application framework APIs app_provisioning_start() and app_provisioning_stop() can be used for starting and stopping the provisioning. Similarly, the API, app_configure_network() can be used for configuring the provisioned network and instructing the application framework to start the station. Please refer to the sample application, wlan/wlan_prov for a quick demonstration of this provisioning in action.

The advantage of this provisioning method is that it is simple to implement, and consistently works across all phones and desktops.

The one drawback of this provisioning method is that iOS users manually have to switch the phone's Wi-Fi network to the one that is hosted by the device. Android users face no such problems. Android allows applications to themselves change the Wi-Fi network, thus making the provisioning process completely application driven.

## 9.2 EZConnect Provisioning

In this provisioning mechanism, a native smart phone app transparently interacts with the WMSDK device. The app broadcasts the credentials of the target Wi-Fi network in a secure and encrypted manner. The WMSDK device sniffs these packets from the air and extracts the network configuration information. Fully functional reference applications for Android and iOS are shipped along with the WMSDK bundle.

The advantage of this provisioning method is that the entire user experience is completely driven from the app, without the user having to manually change Wi-Fi networks, on iOS as well as Android phones.

The drawback of this method is that some small percentage of APs buffer multicast traffic. This interferes with the provisioning protocol that is used by EZConnect provisioning. These users may face problems for the provisioning process.

## 9.3 WAC (Wireless Accessory Configuration)

This is Apple's standard for performing the network provisioning of the devices. Note that this requires Apple's MFi authentication chip to be interfaced with the microcontroller. Support for WAC is in-built in OS-X and iOS. Thus, as against the other provisioning mechanisms, no other phone/desktop application need be installed for performing the WAC provisioning. Note that this support is currently only available in the HAP SDK.

The application framework API app_wac_start() and app_wac_stop() can be used for controlling the WAC provisioning. The HAP SDK includes a sample accessory that shows how to perform the WAC provisioning.

The advantage of this mechanism is that, since it is completely integrated in OS-X and iOS the user experience is seamless.

The drawback with this method is that it requires a MFi license, and it requires the MFi chip to be interfaced with the microcontroller.

## 9.4 WPS (Wi-Fi Protected Setup)

This is the easiest provisioning mechanism and published as a standard by the Wi-Fi Alliance. In its simplest form, this includes the customer pressing a button on their home AP, and another button on the WMSDK device. The device and the AP then perform the negotiation, and follow a defined process. The end result of the process is that the device gets the configuration of the target Wi-Fi network. While this method is the easiest, not all APs may support this mechanism because of some known security issues. This method can be used along with the *Marvell Provisioning* method defined below to address this problem.

The drawback with this method is that it consumes quite a lot of footprint on the microcontroller.

# 10 Power Management

## 10.1 Microcontroller Power Save

A quick summary of the power modes of the 88MC200 and 88MW300 is provided below:

**Microcontroller Speed**: 200MHz

**Wi-Fi**: Deep Sleep

| Power State | Name | Comments | Power Consumption | |
|---|---|---|---|---|
| | | | 88MC200 | 88MW300[#] |
| PM0 | Active | This is the mode where all execution happens. | 52.8mA | 43.08mA |
| PM1 | Idle | Switches off core clocks. Wakes up on any interrupt to the Cortex M3 | 38.57mA | 28.2mA |
| PM2 | Standby | M3 Core, most CPU peripherals and SRAM arrays in low power mode with retention<br>Wakes up on timeout or by asserting WAKE_UPx pin | 1.45mA | 488uA |
| PM3 | Sleep | On top of PM2, only partial SRAM (160KB SRAM0, 32KB SRAM1) is maintained in retention state, rest is switched off<br>Wakes up on timeout or by asserting WAKE_UPx pin | 212uA | 120uA |
| PM4 | Shutdown | Only VDD_AON is active.<br>Wakes up on timeout or by asserting WAKE_UPx pin.<br>On wakeup execution begins from the BootROM | 178uA | 92uA |

# - Numbers taken on RD-88MW302-B0-V3 board with USB_AVDD33 trace cut on the board. The readings include flash and board leakage power numbers.

Please refer to the document *AppNote: Power Management* for additional details.

### 10.1.1 Controlling Power Save

If the power manager is initialized with a call to pm_init(), the power manager automatically manages the transitions between PM0 and PM1. This ensures that the system enters a relatively lower power mode, whenever it is idle, without any side effects.

The API pm_mc2u_state() can be used to put the microcontroller into any of the desired power states. This API can be used to enter any of the other idle state, PM2/3/4.

#### 10.1.1.1 Tickless Idle

The WMSDK also supports a more sophisticated mechanism, called the power manager framework that is integrated with the system scheduler to put the microcontroller into power save modes. This implies that the kernel can automatically detect idle periods and transparently enter low power modes. Since the kernel is already aware of the next scheduled activity, it can wake up from low power mode, just in time to service it. The API pm_mcu_cfg() can be used for enabling this mode.

When this mode is used, applications can use pm_register_cb() to register entry and exit callbacks for the various power save states. Thus while the system automatically transitions between the power states, applications are notified and can take any desired action within these callbacks.

Wakelocks have also been provided. Applications can take a wakelock to prevent the microcontroller from entering the power-save mode until the wakelock is being released.

When used in conjunction with the pm_ieeeps_hs_cfg(), the power manager framework also configures the Wi-Fi chipset to wake up the microcontroller on the desired network activity. This, of course, requires that the wakeup from the Wi-Fi chipset is connected to the appropriate wakeup-pin of the microcontroller.

## 10.2 Wi-Fi Power Save

The Wi-Fi power-save states are mentioned below:

| **Wi-Fi Power State** | **Wi-Fi State** | **Comment** |
|---|---|---|
| WLAN Active State | Connected | This is the mode when TX/RX can be performed. |
| Deep Sleep state | Disconnected | This mode can be enabled ONLY when Wi-Fi is disconnected. |
| Power Down (PDN) state | Disconnected | This mode can be enabled ONLY when Wi-Fi is disconnected. |
| IEEE Power Save state | Connected | This mode is enabled when Wi-Fi is connected to an AP. This is the most common power save mode. |

Please refer to the document *AppNote: Power Manager* for additional details about the power consumption in these states.

### 10.2.1 Controlling Power-Save

The following APIs can be used to put the Wi-Fi chipset into the various power-save state:

- Deep Sleep – wlan_deepsleepps_on() / wlan_deepsleepps_off()
- Power Down – wlan_pdnps_on() / wlan_pdnps_off()
- IEEE Power Save – pm_ieeeps_hs_cfg()

When in IEEE-power save, the Wi-Fi chipset in conjunction with the micro-controller ensures that the Wi-Fi chipset is in low power mode whenever no traffic is being exchanged.

# 11 Cloud Communication

The SDK provides APIs that allow applications to communicate with a remote cloud service. Cloud communication allows an application firmware to be controlled remotely, to be upgraded to latest firmware releases or to post data allowing data analysis opportunities.

## 11.1 Supported Cloud Agents

Marvell has partnered with multiple third-party cloud providers two of whom are *Arrayent*, *Xively* and *Evrythng*. The respective cloud communication agents for the cloud service providers are already integrated and part of the SDK. Please contact your Marvell sales representative for further details.

### 11.1.1 Arrayent

The following steps quickly discuss the various APIs that will typically be required to communicate with the Arrayent cloud.

1. A cloud communication thread handles all the communication with the cloud
2. The thread calls ArrayentConfigure() followed by ArrayentInit() to establish communication with the cloud
3. It can read the value of any property using the ArrayentRecvProperty() API
4. It can update the value of any property using the ArrayentSetProperty() API

Please refer to the *Arrayent Reference Manual* for detailed documentation of the available API.

### 11.1.2 Xively

The following steps quickly discuss the various APIs that will typically be required to communicate with the Xively cloud.

1. A cloud communication thread handles all the communication with the cloud
2. The thread calls xi_create_context() to create a communication context
3. It can read the value of any data stream using the xi_feed_get() API
4. It can update the value of any data stream using the xi_feed_update() API

Please refer to the *Xively Reference Manual* for detailed documentation of the available API.

### 11.1.3 Evrythng

The following steps quickly discuss the various APIs that will typically be required to communicate with the Evrythng cloud.

1. A cloud communication thread handles all the communication with the cloud
2. The thread calls EvrythngConfigure() to configure the agent
3. It can subscribe to updates to properties using the EvrythngSubThngProperty() API
4. It can publish changes to properties using the EvrythngPubThngProperty() API

## 11.2 Communicating with other clouds

Communicating with cloud services apart from the supported cloud agents above is also very easy with the SDK. The SDK provides APIs at various levels to address all the typical use cases for communicating with any other cloud service.

### 11.2.1 HTTP Client and TLS

The most common cloud communication model is to use HTTP for accessing RESTful web services. The SDK provides a HTTP Client implementation that can be used for this purpose. The SDK has integrated TLS (Transport Layer Security) within the HTTP Client. This allows accessories to communicate with the cloud service over the secure *https* protocol.

The following steps quickly discuss the various APIs that will typically be required to communicate with and https cloud service.

1. A cloud communication thread handles all the communication with the cloud
2. The thread calls http_open_session() for opening a HTTP connection with the cloud. Any TLS configuration, like certificates etc, can be passed to this API.
3. It can prepared and send an HTTP GET/POST/PUT request using the http_prepare_req() API followed by the http_send_req() API
4. It can then read the response header using the http_get_response_hdr() API. Once the response header is processed, subsequent data in the HTTP response can be read using the http_read_content() API.

Please refer to the *Reference Manual* for detailed documentation of the available API.

### 11.2.2 BSD Socket API

The SDK also provides BSD socket level APIs. These APIs allow you to communicate with the cloud service over any other protocol. The following BSD Socket APIs are available, net_socket(), net_bind(), net_listen(), net_accept(), net_read(), net_write() and net_close(). TLS is also available to ensure that a secure communication is facilitated.

#### 11.2.2.1 Creating a sample TCP Server

This subsection showcases how a typical TCP server can be implemented using the provided API. Most of this is standard BSD Socket API.

1. Firstly, create a socket that will listen for connections on a given port

   ```
   int socket;

   socket = net_socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
   ```

2. Bind this socket to the appropriate IP Address of the micro-AP or the station interface

   ```
   struct sockaddr_in addr_listen;

   addr_listen.sin_family = AF_INET;

   addr_listen.sin_port = htons(4321);

   addr_listen.sin_addr.s_addr = ip;

   addr_len = sizeof(struct sockaddr_in);

   error = net_bind(socket, (struct sockaddr *)&addr_listen,
                                    addr_len);
   ```

3. Specify the number of connections that can stay in the backlog queue

   ```
   error = net_listen(socket, 3);
   ```

4. Wait for connections on this socket

   ```
   int new_socket;

   new_socket = net_accept(socket,

                     (struct sockaddr *)&addr_from, &addr_from_len);
   ```

5. Once you receive a new connection, perform data transfer on the new socket for this connection

   ```
   error = send(new_socket, buffer, sizeof_data, 0);
   ```

Similarly a simple TCP client can be implement by directly using the net_connect() call after creating the socket in step 1. Once the connection is successful, data transfer happens in a similar manner.

### 11.2.3 JSON/XML

Once secure data communication has been established, by means of the mechanisms outlined above, the data can be exchanged in JSON or XML formats. The SDK provides JSON and XML parsers and generators to exchange data with the cloud.

Please refer to the *Reference Manual* for detailed documentation of the available API.

# 12 Reference Phone Apps

The WMSDK provides reference phone applications for showcasing some of the provisioning mechanisms. The phone apps are available for the iOS and Android operating systems.

All phone apps contain a README file that describes that steps that should be taken to build the respective phone app.

The following is showcased:

1. **Marvell uAP Provisioning**: This phone app implements the provisioning mechanism as described in Section 9.1. The steps that the end-user performs are as follows:
   a. Android:
      i. The user launches the app
      ii. The app connects to a micro-AP network with a pre-defined prefix
      iii. The app asks the user to select the home network from those available in the vicinity
      iv. The app prompts the user to enter the passphrase of the home network
      v. The app provisions the device with the credentials. The result of the provisioning is displayed to the user.
   b. iOS:
      i. The user launches the app
      ii. The app instructs the user to connect to the micro-AP network with a pre-defined prefix (As against Android, iOS does not allow apps to connect to Wi-Fi network, the user has to perform this action by herself)
      iii. The user launches the app again
      iv. The app asks the user to select the home network from those available in the vicinity
      v. The app prompts the user to enter the passphrase of the home network
      vi. The app provisions the device with the credentials. The result of the provisioning is displayed to the user
2. **EZConnect Provisioning**: This app implements the provisioning method that is defined in Section 9.2. The steps that the end-user performs are as follows:
   a. Android/iOS:
      i. The app prompts the user to enter the passphrase of the home network (the phone is expected to be already connected to the home network)
      ii. The app starts performing provisioning
      iii. The app listens for announcements of the device on the home network. When an announcement is seen, the app confirms provisioning success to the end-user
      iv. In case of timeout, the entire procedure is repeated again

# 13 Tools

## 13.1 Flashprog

### 13.1.1 A Word about OpenOCD

The Development board has the FTDI 2232D chip, which enables JTAG access via the mini-USB port (the same USB port that is used for serial console and power). Using the supplied USB cable and the OpenOCD software that is bundled with the SDK, you can download and run firmware images as described below. You can also use OpenOCD to burn firmware images in flash such that the firmware will run automatically when the system boots up.

**Note**

For more information on OpenOCD, including detailed documentation on how to use it, please visit: http://openocd.sourceforget.net/. The site includes a detailed user-manual showing the various ways in which you can use OpenOCD.

Pre-built versions of OpenOCD are included for Linux and Windows platforms with the SDK. Various configuration files that are needed for OpenOCD to run with the development kit are also included under *wmsdk/tools/OpenOCD* directory.

**Note**

**For Windows Users:** The OpenOCD binary bundled with the sdk doesn't work cleanly with the usb-serial console. You may experience that the console becomes inaccessible after using the OpenOCD to perform any operations on the board. After flashing a new image, unplugging and plugging the USB cable into your development host will get the console back.

We cannot provide a binary which does work because of licensing issues. You will need to build OpenOCD yourself with the proprietary FTD2xx drivers to make the usb-serial console work. Refer to the README (Section "Building OpenOCD with FTD2xx drivers") in the OpenOCD directory in the SDK for instructions on how to do it.

**Note**

OpenOCD does not work if the device is in any of the power save states: PM2, PM3 or PM4, This is because power to the JTAG pins is turned off. For OpenOCD to work, please ensure that the device comes out of these PS states, either by pressing the wakeup key or the reset key.

If your application immediately puts the device into PS modes, you may have to press-and-hold the wakeup key while you execute the OpenOCD utilities like *ramload* and *flashprog*. (**NOTE:** ramload will not work with XiP code)

### 13.1.2 Using Flashprog

A flash layout (partition table) is available on flash that identifies the location of other components (boot2, psm, ftfs, wlanfw, and mcufw) in flash. Once written, all entities (application, flashprog) reference this layout to understand where each of the flash components is stored.

The flashprog utility allows you to program the internal and external flash of the device as per your requirements. This utility can be used to

1. write the partition table
2. write the other components as per the partition table

While writing the components to flash, the utility can operate in two modes, batch mode and interactive mode. The batch mode is typically used to write all the components in flash at once. The interactive mode can be used when individual components have to be written to the flash. This utility internally uses the OpenOCD tool to perform all the modifications.

#### 13.1.2.1 Burning All Components in Flash

There are multiple images that are written to the flash. These include the boot2 boot-loader, the firmware image, the filesystem (FTFS) image and the wireless firmware binary. The interactive mode for flashprog can be used to burn these components individually. All of these components can be burnt in a single-go by using the batch mode of the flashprog utility.

##### 13.1.2.1.1 Batch Configuration File

The batch mode requires a configuration file to be passed to the flashprog utility that will then be used toflash everything. A sample configuration file is as shown below:

```
$ cat /cygdrive/c/config
boot2           c:/cygwin/images/boot2.bin
mcufw         c:/cygwin/images/wm_demo.bin
ftfs      c:/cygwin/images/wm_demo.ftfs
wififw          c:/cygwin/images/sd8845_uapsta.bin.xz
```

This file indicates to the flashprog utility where to pick the individual components to be written to flash.  The file contains two columns separated by tabs.

The first column is the name of the component as mentioned in the layout file. The batch mode matches the name to the corresponding section in your flash layout and writes the image at that location.

**Note**

The names of the flash component in the batch configuration file should match those specified in your flash layout. If a corresponding component is not found in the flash layout, flashprog will notify you of the error.

It is not necessary to write all the components in the batch configuration file. If only two of your components change frequently, you can only include those two components in the batch configuration file. The flashprog utility will keep the rest of the components untouched.

The second column indicates the location of the component that should be written to flash. Note that this should be an absolute path name. For Cygwin users, this should be a Windows-aware path (starting with C: or D: ) instead of Cygwin path starting with (/cygdrive/c/ or /cygdrived/).

Caution

Boot2 and MCU firmware images are specific to the board that they are built for. Please ensure that the images that you write are built for your board.

### 13.1.2.1.2 Batch Mode Execution

Once this configuration file is created as per your configuration, you may execute the batch mode as follows:

```
$ ./flashprog.sh -b c:/config.txt
Open On-Chip Debugger 0.7.0 (2013-05-05-10:41)
Licensed under GNU GPL v2
For bug reports, read
        http://openocd.sourceforge.net/doc/doxygen/bugs.html
Info : only one transport option; autoselect 'jtag'
adapter speed: 5000 kHz
adapter_nsrst_delay: 100
jtag_ntrst_delay: 100
cortex_m3 reset_config sysresetreq
sh_load
Info : clock speed 3000 kHz
Info : JTAG tap: mw300.cpu tap/device found: 0x4ba00477 (mfg: 0x23b, 
part: 0xba00, ver: 0x4)
Info : mw300.cpu: hardware has 6 breakpoints, 4 watchpoints
Info : JTAG tap: mw300.cpu tap/device found: 0x4ba00477 (mfg: 0x23b, 
part: 0xba00, ver: 0x4)
target state: halted
target halted due to debug-request, current mode: Thread
xPSR: 0x01000000 pc: 0x00005028 msp: 0x20001000
requesting target halt and executing a soft reset
target state: halted
target halted due to debug-request, current mode: Thread
xPSR: 0x01000000 pc: 0x00005028 msp: 0x20001000
26704 bytes written at address 0x00100000
downloaded 26704 bytes in 0.220013s (118.530 KiB/s)
verified 26704 bytes in 0.345019s (75.585 KiB/s)
semihosting is enabled

Writing "boot2" @0x0 (primary)...done
Writing "mcufw" @0x7000 (primary)......done
Writing "ftfs" @0xa7000 (primary)....done
Writing "wififw" @0xcb000 (primary)....done
Please press CTRL+C to exit.
Exiting.

Terminated
```

Note

OpenOCD does not work if the device is in any of the power save states: PM2, PM3 or PM4, This is because power to the JTAG pins is turned off. For OpenOCD to work, please ensure that the device comes out of these PS states, either by pressing the wakeup key or the reset key.

If your application immediately puts the device into PS modes, you may have to press-and-hold the wakeup key while you execute the OpenOCD utilities like *ramload* and *flashprog*.

#### 13.1.2.2 Burning File System Images in Flash

Filesystem images are created with a *.ftfs* extension in the project's source directory. The file system image can be burnt into the flash using the flashprog utility. Execute the flashprog program with the option '--ftfs', followed by the name of the file system image.

```
# cd wmsdk/tools/OpenOCD
```

```
# ./flashprog.sh –ftfs /path/to/my_www.ftfs
```

#### 13.1.2.3 Burning the Wi-Fi Firmware

Note that by-default the Wi-Fi firmware is already available in the Development Kits that you received. This procedure is required only if you wish to change the Wi-Fi firmware to a new version, or you accidentally lost the firmware written to flash, or if you changed the flash layout for your application.

For burning the Wi-Fi firmware, execute the flashprog utility with the '--wififw' option.

#### 13.1.2.4 Burning the boot2 image

Note that by-default the boot2 image is already available in the Development Kits that you received. This procedure is required only if you wish to change the boot2 to a new version, or you accidentally lost the boot2 written to flash.

For burning the boot2 image, execute the flashprog utility with the '--boot2' option.

## 13.2 UART boot

Both the 88MC200 and the 88MW30x provide an option of booting from the UART0. Their respective datasheet mentions the protocol that should be followed for such a UART based boot up. The *uartboot* utility can be used to load images using the UART-BOOT protocol on the microcontroller.

### 13.2.1 Building

The uartboot utility can be built as follows:

```
$ cd wmsdk_bundle-x.y.z/
```

```
$ make
```

```
$ make tools_install
```

This will build and install the uartboot at *wmsdk_bundle-x.y.z/bin/wmsdk/tools/bin/Linux/uartboot*.

### 13.2.2 Using

Before an application firmware can be loaded on the microcontroller, it should be converted to the ihex format. This can be done using the following command:

```
$ arm-none-eabi-objcopy –O ihex /path/to/firmware.axf firmware.ihex
```

Now that the ihex file is created, it can be loaded on the microcontroller as follows:

```
$ ./uartboot –p <COM_PORT> –f firmware.ihex
```

where, *COM_PORT* is the port that is enumerated on your development host (for example, /dev/ttyUSB0 etc.)

The microcontrollers also support the protection of UART boot using a password. If this feature is enabled, the uartboot utility can also be used as follows:

```
$ ./uartboot -p <COM_PORT> -f firmware.ihex -s <password>
```

where, *password* is the password that should be used for UART boot.

## 13.2.3 Fixing the linker script for 88MW30X

For the 88MW30x, the BootROM uses part of the SRAM1 for image loading purposes. Firmware images that are booted using the UART-BOOT protocol should ensure that nothing should be placed / loaded into this section, which is used by the BootROM. Please modify the linker script, mw300.ld, or your firmware to reserve this space. The line, which describes the SRAM1 size, should be changed such that it has the following contents. Note that the length is changed from 128K to 105K.

```
SRAM1 (rwx)  : ORIGIN = 0x20005658, LENGTH = 105K
```

## 13.3 footprint.pl

The utility *footprint.pl* can be used to analyze the footprint of any application firmware. Let us look at the various modes of execution of footprint.pl. This utility looks at the map file that is generated for all firmware images. This map file is parsed by footprint.pl to interpret the various sections and the contributions to these sections from the various archive (*.a) and object (*.o) files.

If you have used LTO (Link-Time Optimization) configuration option then the map files that are generated by gcc are garbled. Please refer to Section 13.3.3 for disabling LTO.

## 13.3.1 Looking at Free Space

The “-a” option to footprint.pl shows the placement of the various firmware sections in the available memory space. Please see below for a sample run (User input is in bold):

```
$ cd wmsdk_bundle-x.y.z/
$ ./wmsdk/tools/bin/footprint.pl -m /path/to/my_smart_switch-0.1.map -a mc200
section                              start         end        size    comment
---------------------------------------------------------------------------
SRAM0:
text                            0x00100000 0x00144f38     282424
--Free--                        ---------- ----------        (8)
heap (minimum)                  0x00144f40 0x00155f40      69632       note1
--Free--                        ---------- ----------    (41152)
---------------------------------------------------------------------------
SRAM1:
data                            0x20000000 0x20000bf6       3062
--Free--                        ---------- ----------        (2)
iobufs                          0x20000bf8 0x20002438       6208
bss                             0x20002438 0x200180f8      89280
_main_stack                     0x200180f8 0x200188f8       2048
--Free--                        ---------- ----------    (30472)
---------------------------------------------------------------------------
```

```
NVRAM:

nvram                         0x480c0000 0x480c0078        120

--Free--                      ---------- ----------     (3976)

---------------------------------------------------------------------------

note1: Actual Heap Size = Minimum-Heap + Leftover-Free-Space

       Actual Heap Size = 69632 + 41152 = 110784
```

This output shows how the two RAM banks are occupied with the different firmware sections. As can be seen, SRAM0 contains 41152 bytes of free space, while SRAM1 contains 30472 bytes of free space. The 41152 bytes of free space from SRAM0 is automatically absorbed in the heap section, thus making the effective heap of 110784 bytes in size. This is summarized at the bottom of the output.

## 13.3.2 Looking at Footprint Contributions

The contributions of the various libraries and C files to the firmware’s static footprint can be seen with the following command:

```
$ cd wmsdk_bundle-x.y.z/

$ ./wmsdk/tools/bin/footprint.pl -m /path/to/my_smart_switch-0.1.map

Library                              text     rodata       data        bss
common      Total

##########################

liblwip.a                           52208       1066         16      51315
0     104605

libwifidriver.a                     27812        457       1385      17049
0      46703

libcustom_framework.a               22924       7877        444       3355
2794      37394

libdrv.a                            27694       1227        180       3421
0      32522

libed25519.a                        19690       4045          0        781
0      24516

libctaocrypt.a                      18078        279          0         12
0      18369

libmdns.a                            8860         89        304       6764
0      16017

libwlcmgr.a                          8098        208         61       1007
0       9374

libfw_supplicant.a                   7712        343         34       1144
0       9233

libapp_framework.a                   6780        972        388        828
0       8968

libapi.a                             7628        124          0         17
0       7769

libhttpd.a                           5302        800          4       1188
0       7294
```

```
libpsm.a                            5536        817         60          9
0       6422
libfreertos.a                       5868         74         12        400
0       6354
libaescrypto.a                      2510       2314          0          0
0       4824
libhkdf-sha512.a                    2444       1409         64          8
0       3925
libgcc.a                            3788          0          0          0
0       3788
libdhcpd.a                          2192        237         16       1204
0       3649
libcli.a                            1850        456         28       1014
0       3348
libc.a                              2748        298          0         10
0       3056
libnet.a                            2210         45          6        365
0       2626
libutil.a                           1390        167          0       1056
0       2613
libsrp6a.a                          1664        817          0          4
0       2485
libwm_supplicant.a                  1076         22          8        363
0       1469
libpoly1305.a                       1348          0          0          0
0       1348
libhealthmon.a                       828         31          4        419
0       1282
libchacha-20.a                      1176         32          0          0
0       1208
libos.a                              892        264          0         36
0       1192
lib_services.a                       930        157          0          8
16       1111
libdiagnostics.a                     848        149          0          0
0        997
libcyassl.a                          702          0          0        288
0        990
libwmcrypto.a                        828          0          0          0
0        828
libwmstdio.a                         464          0         20        279
0        763
libpart.a                            366          0          0        261
0        627
libg.a                               588          0          0          0
4        592
```

```
libwmtime.a                      492         9          1         45
0        547

libxflash.a                      432         7          0         40
0        479

libpwrmgr.a                      298         0          0        125
0        423

obj/mc200_8801.o                 406         0          0          0
0        406

obj/smart_plug.o                 296        75          4          5
0        380

obj/main.o                       176        52          0          0
44        272

obj/event_handler.o              190         0          0          0
0        190

libmdev.a                        180         0          0          4
0        184

libarch.a                        152         0          0          1
0        153

libtls.a                          92         0          0          2
0         94

obj/smart_plug_pwrmgr.o            6         0          0          0
0          6

##########################

                                text     rodata       data        bss
common

Totals:                       257752      24919       3039      92827
2858

   +

linker_script                 113401

Final:                    494796 (483 KB )

##########################

As per 'size' utility:

.text = text + rodata = 282671

.data = data = 3039

.bss = bss + common = 95685
```

As can be seen from the output, for every library, this lists the contributions from the various sections of that library. The final column indicates the total contribution from that library/file.

Please refer to the usage documentation of footprint.pl to further try additional options of the same.

### 13.3.3 Disabling LTO Option

When the LTO (Link-Time Optimization) option is enabled, the map file generated by GCC is fully garbled. This makes the footprint.pl utility pretty unusable. You could perform footprint analysis by disabling the LTO option. Note that disabling the LTO option increases the firmware footprint by about 20-30KB. Please keep this in mind while performing the footprint optimizations. Please use the following steps for disabling the LTO option:

1. Edit the build configuration file for the SDK. Comment out the line *CONFIG_ENABLE_LTO=y*, by placing a hash (#) character at the start of the line.

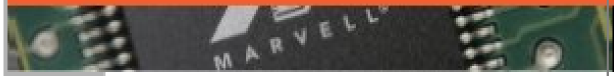
   ```
   # CONFIG_ENABLE_LTO=y
   ```

2. Perform *make clean* and then rebuild the SDK

Note that some libraries that are shipped with the SDK are pre-compiled. These libraries will continue to show up as 'ltrans' files in the footprint output.

# 14 Glossary

ACOMP: Analog comparator
ADC: Analog to Digital convertor
AES: Advanced encryption standard
AP: Access Point
AMPDU: Aggregated Mac Protocol Data Unit
AMSDU: Aggregated Mac Service Data Unit
CRC: Cyclic redundancy check
DAC: Digital to Analog convertor
DHCP: Dynamic host configuration protocol
FTFS: File Table file system
GPT: General purpose timer
HTTP: Hypertext transfer protocol
IEEEPS: IEEE power save
JSON: JavaScript Object Notation
MCU: Microcontroller
mDNS: Multicast DNS
uAP: micro access point
P2P: Wi-Fi Peer-to-Peer
PSM: Persistent storage manager
QSPI: Quad serial peripheral interface
REST: Representational state transfer
RTC: Real Time Clock
SDIO: Secure Digital IO
SoC: System on chip
SPI: Serial Peripheral interface
SSL: Secure Sockets Layer
TLS: Transport Layer Security
URL: Uniform resource locator
WMSDK: Wireless microcontroller software development kit.
WPS: Wi-Fi Protected Setup
WSGI: Web Server Gateway Interface
XIP: Execute in place



Marvell Semiconductor, Inc.
5488 Marvell Lane
Santa Clara, CA 95054, USA

Tel: 1.408.222.2500
Fax: 1.408.752.9028

www.marvell.com

**Marvell.Moving Forward Faster**