

# i.MX Windows 10 IoT User's Guide

for NXP i.MX Platform

Document Number: IMXWGU

Rev. W1.0.0-ear, 10/2019



# Contents

<b>1</b>	<b>Overview</b>	<b>6</b>
1.1	Audience . . . . .	6
1.2	Conventions . . . . .	6
1.3	Directories . . . . .	6
1.4	References . . . . .	6
<b>2</b>	<b>Introduction</b>	<b>8</b>
<b>3</b>	<b>Feature List Per Board</b>	<b>9</b>
<b>4</b>	<b>Flash a Windows IoT Core image</b>	<b>11</b>
<b>5</b>	<b>Basic Terminal Setup</b>	<b>13</b>
<b>6</b>	<b>Basic board setup</b>	<b>14</b>
<b>7</b>	<b>Bootting WinPE and Flashing eMMC</b>	<b>16</b>
7.1	Identifying boot loader location . . . . .	16
7.2	Preparing an FFU to be flashed to eMMC . . . . .	16
7.3	Creating the WinPE Image . . . . .	17
<b>8</b>	<b>Windows 10 IoT Boot Sequence on i.MX Platform</b>	<b>19</b>
8.1	On-chip ROM code . . . . .	19
8.2	SPL . . . . .	20
8.3	OP-TEE . . . . .	22
8.4	U-Boot Proper . . . . .	24
8.5	UEFI . . . . .	25
8.6	SD/eMMC Layout . . . . .	27
<b>9</b>	<b>Securing Peripherals on i.MX using OP-TEE</b>	<b>28</b>
9.1	OP-TEE . . . . .	28
9.2	Windows . . . . .	29
<b>10</b>	<b>Building Windows 10 IoT Core for NXP i.MX Processors</b>	<b>30</b>
10.1	Building the BSP . . . . .	30
10.1.1	Required Tools . . . . .	30
10.1.1.1	Visual Studio 2017 . . . . .	30

10.1.1.2	Windows Kits from Windows 10, version 1809 . . . . .	31
10.1.1.3	IoT Core OS Packages . . . . .	31
10.1.2	One-Time Environment Setup . . . . .	31
10.1.3	FFU Generation . . . . .	31
10.1.3.1	Building the FFU for other boards . . . . .	32
10.1.4	Building the FFU with the IoT ADK AddonKit . . . . .	32
<b>11</b>	<b>Building and Updating ARM32 Firmware</b>	<b>33</b>
11.1	Setting up your build environment . . . . .	33
11.2	Adding updated firmware to your FFU . . . . .	34
11.3	Deploying firmware to an SD card manually . . . . .	34
11.3.1	Bootable Firmware without installing an FFU . . . . .	34
11.3.2	Deploying U-Boot and OP-TEE (firmware_fit.merged) for development . . . .	35
11.3.3	Deploying UEFI (uefi.fit) for development . . . . .	35
11.3.4	Updating the TAs in UEFI . . . . .	35
<b>12</b>	<b>Building and Updating ARM64 Firmware</b>	<b>37</b>
12.1	Setting up your build environment . . . . .	37
12.2	Adding updated firmware to your ARM64 FFU . . . . .	41
12.3	Deploying firmware to an SD card manually . . . . .	41
12.3.1	Deploying U-Boot, ATF, OP-TEE (flash.bin) and UEFI (uefi.fit) for development .	41
<b>13</b>	<b>Adding New Boards and Drivers to the BSP</b>	<b>42</b>
13.1	Adding a New Board . . . . .	42
13.1.1	Initialize a new board configuration . . . . .	42
13.1.2	Setup the solution in Visual Studio . . . . .	42
13.1.3	Update the firmware for your board . . . . .	42
13.1.4	Build the FFU in Visual Studio . . . . .	43
13.1.5	Board Package Project Meanings . . . . .	44
13.2	Adding a New Driver . . . . .	44
13.2.1	Adding a New Driver to the Solution . . . . .	44
13.2.2	Adding a Driver to the FFU . . . . .	45
<b>14</b>	<b>i.MX Porting Guide</b>	<b>46</b>
14.1	U-Boot . . . . .	46
14.1.1	U-Boot Configuration Options . . . . .	47
14.1.2	Adding a new board to U-Boot . . . . .	49
14.2	OP-TEE . . . . .	50

14.3	Setting up your build enviroment to build firmware_fit.merged . . . . .	51
14.3.1	Testing SPL . . . . .	52
14.3.2	Testing OP-TEE . . . . .	52
14.3.3	Testing U-Boot . . . . .	53
14.4	UEFI . . . . .	54
14.4.1	DSC and FDF file . . . . .	54
14.4.2	Board-specific Initialization . . . . .	54
14.4.3	SMBIOS . . . . .	55
14.4.4	ACPI Tables . . . . .	55
14.4.4.1	SDHC . . . . .	55
14.4.4.2	PWM . . . . .	57
14.4.5	Security TAs . . . . .	57
14.4.6	Building UEFI . . . . .	58
14.4.7	Testing UEFI . . . . .	58
14.5	Bootting Windows . . . . .	58
<b>15</b>	<b>Updating the BSP port</b>	<b>59</b>
15.1	Reworked firmware build system . . . . .	59
15.2	FIT load for OP-TEE and U-Boot Proper inside of SPL . . . . .	59
15.3	FIT loading UEFI inside of U-Boot Proper . . . . .	59
15.4	Miscelaneous U-Boot defconfig settings . . . . .	60
<b>16</b>	<b>Revision History</b>	<b>61</b>

# 1 Overview

This chapter describes the process of building and installing the Windows 10 IoT OS BSP (Board Support Package) on the i.MX platform. It also covers special i.MX features and how to use them.

The chapter also lists the steps to run the i.MX platform, including board DIP switch settings, and instructions on configuring and using the U-Boot bootloader.

Features covered in this guide may be specific to particular boards or SOCs. For the capabilities of a particular board or SOC, see the *i.MX Windows 10 IoT Release Notes* (IMXWIN10RN).

## 1.1 Audience

This chapter is intended for software, hardware, and system engineers who are planning to use the product, and for anyone who wants to know more about the product.

## 1.2 Conventions

This chapter uses the following conventions:

- Courier New font: This font is used to identify commands, explicit command parameters, code examples, expressions, data types, and directives.

## 1.3 Directories

BSP - Generated at build time. Contains Board Support Packages for the [IoT ADK AddonKit](#).

build - Contains Board Packages, build scripts, and the VS2017 solution file.

driver - Contains driver sources.

documentation - Contains usage documentation.

hal - Contains hal extension sources.

## 1.4 References

For more information about Windows 10 IoT Core, see Microsoft online documentation.

- <http://windowsondevices.com>

The quick start guides contain basic information on the board and setting it up. They are on the NXP website.

- [SABRE Platform Quick Start Guide \(IMX6QSDPQSG\)](#)
- [SABRE Board Quick Start Guide \(IMX6QSDBQSG\)](#)
- [i.MX 6UltraLite EVK Quick Start Guide \(IMX6ULTRALITEQSG\)](#)
- [i.MX 6ULL EVK Quick Start Guide \(IMX6ULLQSG\)](#)
- [i.MX 6SoloX Quick Start Guide \(IMX6SOLOXQSG\)](#)
- [i.MX 7Dual SABRE-SD Quick Start Guide \(SABRESDBIMX7DUALQSG\)](#)
- [i.MX 8M Quad Evaluation Kit Quick Start Guide \(IMX8MQUADEVKQSG\)](#)
- [i.MX 8M Mini Evaluation Kit Quick Start Guide \(8MMINIEVKQSG\)](#)

Documentation is available online at [nxp.com](http://nxp.com)

- i.MX 6 information is at <http://nxp.com/imx6series>
- i.MX SABRE information is at <http://www.nxp.com/imxSABRE>
- i.MX 6UltraLite information is at <http://www.nxp.com/imx6ul>
- i.MX 6ULL information is at <http://www.nxp.com/imx6ull>
- i.MX 6SoloX information is at <http://www.nxp.com/imx6sx>
- i.MX 7Dual information is at <http://www.nxp.com/imx7d>
- i.MX 8 information is at <http://www.nxp.com/imx8>

## 2 Introduction

The i.MX Windows 10 IoT BSP is a collection of binary files, source code, and support files you can use to create a bootable Windows 10 IoT image for i.MX development systems.

Before you start, see the [Feature List Per Board chapter](#). This section lists all the i.MX boards covered by this BSP and also contains a list of possible features.

If you have downloaded a standalone Windows 10 IoT core image, please go to [Flash a Windows IoT Core image](#) to create a bootable SD card.

If you have downloaded an archive with BSP sources, please go to [Building Windows 10 IoT Core for NXP i.MX Processors](#) and check the process of the building the BSP and Boot firmware. After that you can prepare bootable SD card according to [Flash a Windows IoT Core image](#) chapter.



### 3 Feature List Per Board

Table 3.1: Overview of the currently supported features for every board.

Feature	MCIMX6Q-SDB/SDP	MCIMX6QP-SDB	MCIMX6DL-SDP	MCIMX6SX-SDB
BSP name	Sabre_iMX6Q_1GB	Sabre_iMX6QP_1GB	Sabre_iMX6DL_1GB	Sabre_iMX6SX_1GB
SD Card boot	Y	Y	Y	Y
eMMC boot	Y	Y	Y	N*
Audio	Y	Y	Y	Y
GPIO	Y	Y	Y	Y
I2C	Y	Y	Y	Y
Ethernet	Y	Y	Y	Y
PWM	Y	Y	Y	Y
SD Card	Y	Y	Y	Y
eMMC	Y	Y	Y	N*
SPI (master)	Y	Y	Y	Y
Display	Y	Y	Y	Y
UART	Y	Y	Y	Y
USB (host)	Y	Y	Y	Y
PCIe	Y	Y	Y	Y
TrEE	Y	Y	Y	Y
M4	N/A	N/A	N/A	N**
Processor PM	Y	Y	Y	Y
Device PM	Y	Y	N**	N**
LP standby	N**	N**	N**	N**
Display PM	Y	Y	Y	Y
DMA	Y	Y	Y	Y

	MCIMX6UL- EVK	MCIMX6ULL- EVK	MCIMX7SABRE	MCIMX8M- EVK	8MMINILPD4- EVK
BSP name	EVK_iMX6UL_51	EVK_iMX6ULL_51	Sabre_iMX7D_	NXPEVK_IMX8M	NXPEVK_IMX8M_Mini_2GB
SD Card boot	Y	Y	Y	Y	Y
eMMC boot	N*	N*	Y	Y	Y
Audio	Y	Y	Y	Y	Y
GPIO	Y	Y	Y	Y	Y

Feature	MCIMX6UL- EVK	MCIMX6ULL- EVK	MCIMX7SABRE	MCIMX8M- EVK	8MMINILPD4- EVK
I2C	Y	Y	Y	Y	Y
Ethernet	Y	Y	Y	Y	Y
PWM	Y	Y	Y	Y	Y
SD Card	Y	Y	Y	Y	Y
eMMC	N*	N*	Y	Y	Y
SPI (master)	N*	N*	Y	N/A	Y
Display	Y	Y	Y	Y	Y
UART	Y	Y	Y	Y	Y
USB (host)	Y	Y	Y	Y	Y
PCIe	N/A	N/A	Y	Y	Y
TrEE	Y	N	Y	Y	Y
M4	N/A	N/A	N**	N**	N**
Processor	Y	Y	Y	Y	Y
PM					
Device PM	N**	N**	N**	N**	N**
LP standby	N**	N**	N**	N**	N**
Display PM	Y	Y	Y	Y	Y
DMA	Y	Y	Y	Y	Y

Legend	Meaning
Y	Enabled
N/A	Feature not applicable
N*	Feature not enabled - feature is not available in default board configuration
N**	Feature not enabled - feature is not currently supported
PM	Power management
LP	Low power

It's possible that not every feature of a given subsystem is fully enabled and/or optimized. If you encounter issues with supported features, open an issue.

## 4 Flash a Windows IoT Core image

This chapter describes the process of creating a bootable SD card from a downloaded FFU file containing an image of Windows 10 IoT Core system.

- 1) Download and Start the [Windows IoT Core Dashboard](#) utility.
- 2) Navigate to “Set up a new device” tab.
- 3) Select **NXP [i.MX6/iMX7/i.MX8]** under “Device Type” list box.
- 4) Select **Custom** under “OS Build” list box.
- 5) Click **Browse** and navigate and select the FFU file you have downloaded or created by building the BSP.
- 6) Plug the SD card into the PC, and choose this SD card in “Drive” list box.
- 7) Optionally, you can set the **Device Name** and **Administrator Password** selections for your device.
- 8) Check the **I accept the software license terms** checkbox (lower right) and click **Install**.

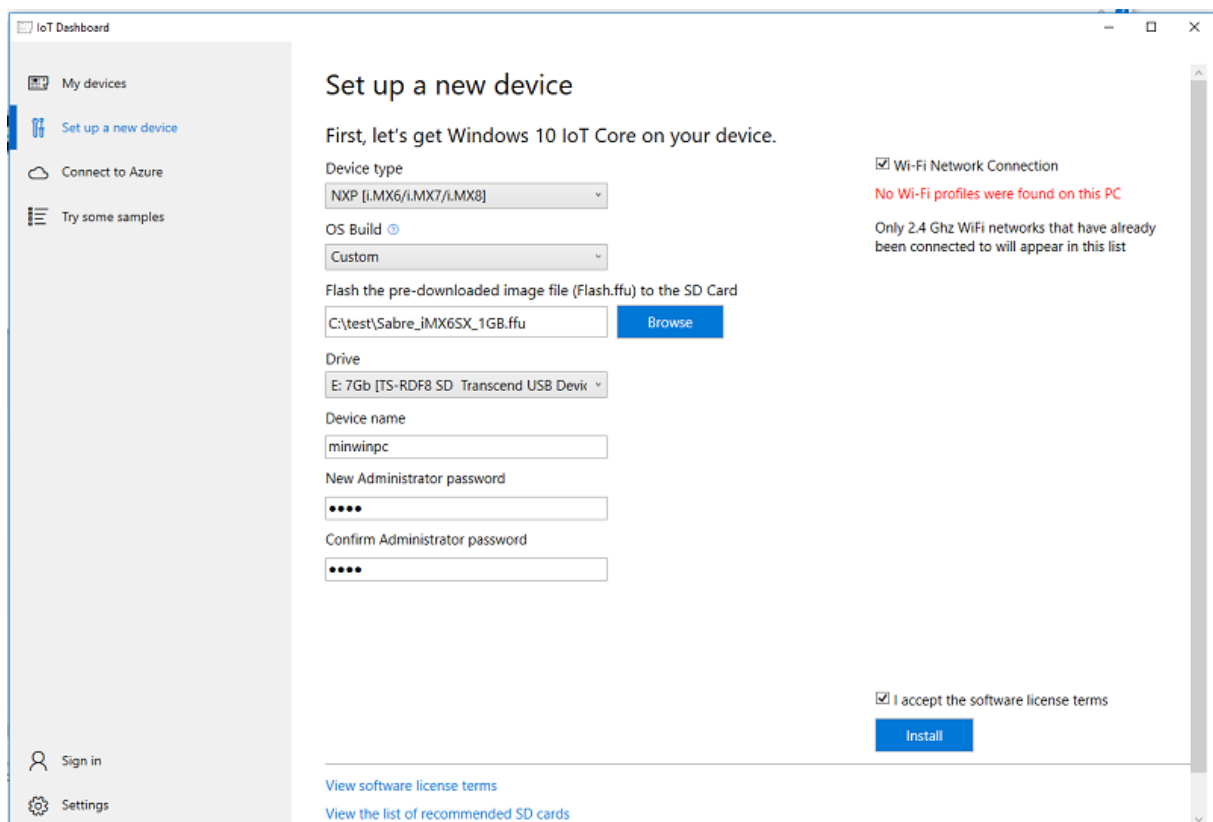


Figure 4.1: IoT Dashboard

Windows IoT Core Dashboard will now open a command window and use DISM (Deployment Image Servicing and Management Tool) to flash the FFU file to your microSD card.

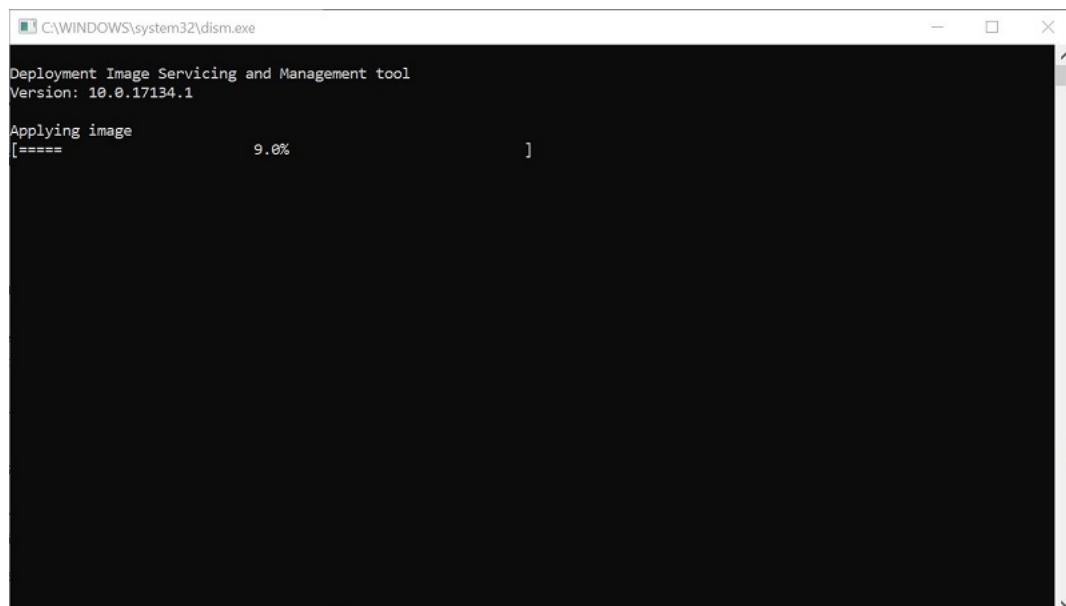


Figure 4.2: Flash

**NOTE:** Alternatively, you can also use the DISM command to manually flash the image:

```
1 dism.exe /Apply-Image /ImageFile:"D:\flash.ffu" /ApplyDrive:.\PhysicalDriveX /SkipPlatformCheck
```

Where “PhysicalDriveX” is a name of your SDCARD physical drive. You can use wmic command to see your physical drives:

```
1 wmic diskdrive get Name, Manufacturer, Model, InterfaceType, MediaType, SerialNumber
```

For more information about flashing the FFU onto an SD Card using the Windows IoT Core Dashboard, follow the [IoT Core Manufacturing Guide](#).

Once the SD card will be created, plug the card into the board and power up the board. The board should successfully boot up. If not, check the configuration of the boot switches - chapter [Basic board setup](#). Optionally you can follow the steps in [Basic Terminal Setup](#) to establish serial connection between the host PC and the target IoT device to check output from U-Boot and UEFI.

## 5 Basic Terminal Setup

During the boot, you can check the U-Boot and UEFI firmware output on the host PC by using the serial interface console. In the case you don't see any output on the connected display, for example, this might be helpful to confirm that the board is booting. Common serial communication applications such as HyperTerminal, Tera Term, or PuTTY can be used on the host PC side. The example below describes the serial terminal setup using Tera Term on a host running Windows OS. The i.MX boards connect the host driver using the micro-B USB connector.

1. Connect the target and the PC running Windows OS using a cable mentioned above.
2. Open Tera Term on the PC running Windows OS and select the settings as shown in the following figure.

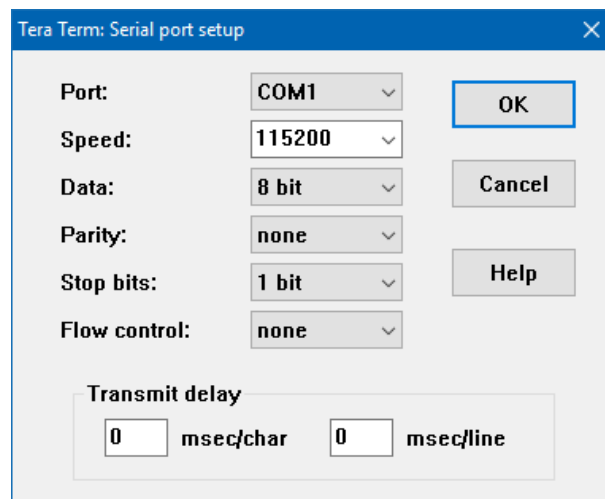


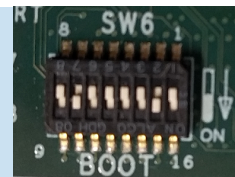
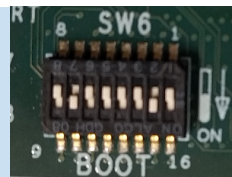
Figure 5.1: Tera Term settings for terminal setup

The USB to serial driver for FTDI chip can be found under <http://www.ftdichip.com/Drivers/VCP.htm>. The FTDI USB to serial convertors provide up to four serial ports. Users need to select the first port (COM) in the terminal setup. The USB to serial driver for CP210x chip can be found under <https://www.silabs.com/products/development-tools/software/usb-to-uart-bridge-vcp-drivers>. The CP210x USB to serial convertors provide up to two serial ports.

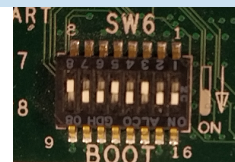
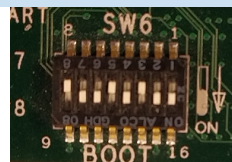
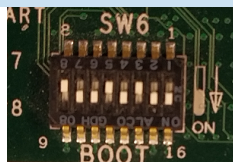
## 6 Basic board setup

Feature	MCIMX6Q-SDB/SDP	MCIMX6QP-SDB	MCIMX6DL-SDP
BSP name	Sabre_iMX6Q_1GB	Sabre_iMX6QP_1GB	Sabre_iMX6DL_1GB
Debug UART*	J509	J509	J509
Default display	J8 (HDMI)***	J8 (HDMI)***	J8 (HDMI)***
SD card boot slot	J507 (SD3)	J507 (SD3)	J507 (SD3)

SD card boot DIP cfg

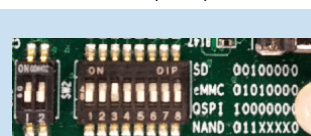


eMMC boot DIP cfg



Feature	MCIMX6SX-SDB	MCIMX7SABRE
BSP name	Sabre_iMX6SX_1GB	Sabre_iMX7D_1GB
Debug UART*	J16	J11
Default display	J12 (LVDS)	J9 (HDMI)
SD card boot slot	J4 (SD4)	J6 (SD1)

SD card boot DIP cfg

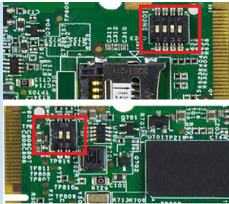
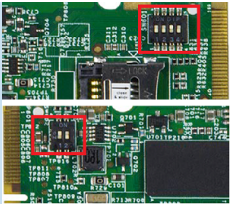


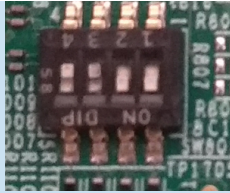
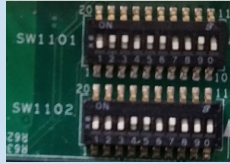
eMMC boot DIP cfg

N/A

N/A

Feature	MCIMX6UL-EVK	MCIMX6ULL-EVK
BSP name	EVK_iMX6UL_512MB	EVK_iMX6ULL_512MB
Debug UART*	J1901	J1901
Default display	J901 (LCDIF)	J901 (LCDIF)
SD card boot slot	J301**	J301**

Feature	MCIMX6UL-EVK	MCIMX6ULL-EVK
SD card boot DIP cfg		
eMMC boot DIP cfg	N/A	N/A

Feature	MCIMX8M-EVK	8MMINILPD4-EVK
BSP name	NXPEVK_IMX8M_4GB	NXPEVK_IMX8M_Mini_2GB
Debug UART*	J1701	J901
Default display	J1001 (HDMI)	J801 (MIPI DSI)
SD card boot slot	J1601**	J701**
SD card boot DIP cfg		
eMMC boot DIP cfg	N/A	N/A

Legend	Meaning
*	Serial port configuration: 115200 baud, 8 data bits, 1 stop bit, no parity.
**	MicroSD card slot
***	Updated UEFI image needed to have LVDS as default display output. Please set TRUE for “PcdLvdsEnable” in *.dsc file. Section [PcdsFeatureFlag.common].

## 7 Booting WinPE and Flashing eMMC

This chapter describes the process of booting Windows from eMMC. To boot a device from eMMC, you first need to flash a Windows image to eMMC. Since eMMC is soldered to the board, the only way to write to it is to boot a manufacturing OS on the device, then write the image from this manufacturing OS. The manufacturing OS is booted from a removable storage such as USB or SD. The tools and techniques can be adapted to other hardware designs.

For the manufacturing OS, we will use [Windows PE \(WinPE\)](#), which stands for Windows Preinstallation Environment. Windows PE is a small Windows image that can boot without persistent storage, and contains tools to help install Windows such as `diskpart` and `dism`.

The high-level process we will follow is:

1. Specify the location of the bootloader.
2. Prepare an FFU to be flashed to eMMC.
3. Prepare a WinPE image, which will flash the FFU to eMMC.

### 7.1 Identifying boot loader location

First, specify the location of the bootloader. i.MX6/7/8 can boot from a number of sources including eMMC/SD, NOR flash, SPI, I2C, and USB. For more information about i.MX6/7/8 early boot, see [Firmware Boot Documentation](#) and the “System Boot” chapter of the processor reference manual. In this chapter we will assume the initial boot device is SD.

To avoid bricking your device, we recommend putting the first stage bootloader on media that can be reprogrammed via external means if necessary, such as an SD card, SPI flash with external programming circuitry, or I2C flash with external programming circuitry. By external programming circuitry, we mean that you can write new contents to the storage device without booting the i.MX processor.

Another strategy is to place the bootloader on eMMC and have a second, read-only eMMC part containing a recovery boot image which can be selected via hardware muxing. This way, if the primary eMMC part becomes corrupted, you can press a button or connect a jumper and boot the device from the backup eMMC part, which then allows you to recover the main eMMC part.

### 7.2 Preparing an FFU to be flashed to eMMC

The FFU that gets flashed to MMC does not have any special requirements. If the firmware is going to be located on a different storage device, it does not need to contain the firmware packages. Use the



same FFU that gets flashed to the SD card.

## 7.3 Creating the WinPE Image

Create an image that can boot from removable media, does not require persistent storage, and contains `dism.exe`, which is the tool that writes an FFU to storage. WinPE is designed for this purpose. To create a bootable WinPE image, we need to:

1. Inject i.MX drivers into the image.
2. Copy the WinPE image to an SD card.
3. Copy firmware to the SD card.

The script `build/tools/make-winpe.cmd` (or `build/tools/make-winpe-i.MX8.cmd` for ARM64 platform) does all of the above, and can be used to set the WinPE image to flash an FFU to a storage device at boot.

Install the following software:

1. [ADK for Windows 10](#)
2. [Windows PE add-on for the ADK](#)
3. `dd for windows`, which must be placed on your PATH or in the current directory

First, create a WinPE image on our machine. In this example, we specify the `"/ffu"` option because we want to deploy an FFU to eMMC. You must have already built the FFU. The script must be run as administrator, and it will create files in the directory in which it runs.

```
1 mkdir winpe
2 cd winpe
3 :: For ARM platform run:
4 make-winpe.cmd /bulddir ..\imx-iotcore\build\solution\iMXPlatform\Build\FFU\bspcabs\ARM\Debug\ /
    firmware path\to\firmware_fit.merged /uefi path\to\uefi.fit /ffu path\to\bsp.ffu
5 :: For ARM64 platform run:
6 make-winpe-i.MX8.cmd /bulddir ..\imx-iotcore\build\solution\iMXPlatform\Build\FFU\bspcabs\ARM64\
    Debug\ /firmware path\to\flash.bin /uefi /uefi path\to\uefi.fit /ffu path\to\bsp.ffu
```

If `/ffu` switch command is omitted the board will boot just into WinPE.

Then, apply the image to an SD card. Insert an SD card into your machine, then determine the physical disk number by running:

```
1 diskpart
2 > list disk
3 > exit
```

The output will look something like this:

```
1 DISKPART> list disk
2
3   Disk ###  Status              Size       Free      Dyn  Gpt
4   -----  -
5   Disk 0      Online                931 GB       0 B
6   Disk 1      Online                931 GB       0 B
7   Disk 2      Online                953 GB       0 B      *
8   Disk 3      No Media                 0 B         0 B
9   Disk 4      No Media                 0 B         0 B
10  Disk 5      No Media                 0 B         0 B
11  Disk 6      No Media                 0 B         0 B
12  * Disk 7      Online                 14 GB       0 B
```

In this example, the physical disk number is 7.

Apply the WinPE image to the SD card:

```
1 :: For ARM platfrom run:
2 make-winpe.cmd /apply 7
3 :: For ARM64 platform run:
4 make-winpe-i.MX8.cmd /apply 7
```

It will format the SD card, copy the WinPE image contents, and write the firmware to the reserved sector at the beginning of the card.

You can now insert the SD card in your board and boot. It will boot into WinPE, then flash the FFU to eMMC, then reboot. Before rebooting, it renames the `EFI` folder at the root of the SD card to `_efi`, which causes UEFI to skip the SD card when it's looking for a filesystem to boot from. It will find the `EFI` directory on eMMC instead, and boot from there. If you wish to boot into WinPE again, you can rename `_efi` back to `EFI`.

If you wish to boot directly from eMMC configure the board switched accordingly and restart the board.

## 8 Windows 10 IoT Boot Sequence on i.MX Platform

This chapter describes the boot sequence on i.MX6 from power-on to the first Windows component (bootmgr). Several components are involved: on-chip ROM code, U-Boot SPL, U-Boot proper, OP-TEE, and UEFI.

1. The on-chip ROM code
  1. Loads SPL into OCRM.
  2. If High Assurance Boot is enabled boot ROM halts if SPL signature is invalid.
  3. Jumps into SPL.
2. SPL
  1. Captures and hides the secret device identity when High Assurance Boot is enabled.
  2. Verifies Flat Image Tree containing U-Boot proper and OP-TEE.
  3. Loads OP-TEE and U-Boot proper.
  4. Jumps into OP-TEE.
3. OP-TEE
  1. OP-TEE runtime initialization.
  2. Switches to normal world then jumps into U-Boot proper.
4. U-Boot proper loads UEFI then jumps to UEFI.
5. UEFI loads and starts bootmgr.

SPL and U-Boot are not retained in memory after boot, while parts of OP-TEE and UEFI remain in memory while the OS runs. The OS calls into OP-TEE and UEFI at runtime to perform certain functions, such as real-time clock operations and processor power management.

### 8.1 On-chip ROM code

Execution begins in on-chip ROM code which is burned into the chip. The ROM code reads its configuration from on-chip fuses. Fuses control options such as boot source, JTAG settings, high-assurance boot (HAB), and TrustZone configuration. Boot source is the media from which the next boot stage will be loaded. This can be EIM, SATA, serial ROM, SD/eSD, MMC/eMMC, or NAND flash. Only SD and eMMC are supported by the reference firmware, and the rest of this chapter describes the boot flow from SD/eMMC. The on-chip ROM code reads the boot binary from the boot source into memory, performs high-assurance boot verification, and jumps to it.

The on-chip ROM code

1. Reads fuses to determine boot source
2. Reads fuses to determine HAB state
3. Loads boot header from MMC sector 2 to OCRAM
4. Parses boot header and loads rest of boot binary (SPL) into OCRAM
  - Load address defined by `CONFIG_SPL_TEXT_BASE` in `include/configs/imx6_spl.h`
5. Parses CSF and does HAB verification
6. Runs DCD commands from boot header
7. Jumps to SPL entry point

Memory layout just before jump to SPL:

```
1  DRAM not yet initialized
2
3  SRAM
4  +-----+ 0x00940000 (end of SRAM)
5  | reserved by boot ROM |
6  +-----+ 0x00938000
7  |
8  |
9  |
10 |
11 | SPL
12 +-----+ 0x00908000 (CONFIG_SPL_TEXT_BASE)
13 |
14 +-----+ 0x00907000
15 | reserved by boot ROM |
16 +-----+ 0x00900000 (start of SRAM)
```

## 8.2 SPL

SPL is a binary produced by the U-Boot build whose purpose is to prepare the system for execution of full U-Boot (U-Boot proper) from DRAM. SPL is the first piece of code that can be changed, as opposed to on-chip ROM code which is burned into the chip and cannot be changed. SPL builds from the same sources as full U-Boot, but is designed to be as small as possible to fit in OCRAM. The included U-Boot has modifications to load OP-TEE.

The reference implementation of SPL

1. Begins execution at `arch/arm/cpu/armv7/start.S : reset`
2. Does low-level CPU init
  1. Errata
  2. CP15 and system control registers

3. Initializes DDR
4. Initializes critical hardware
  1. Pin muxing
  2. Clocks
  3. Timer
  4. Console UART
5. Enables L1 cache
  1. Sets up page tables above stack. There must be 16k of available memory above the stack to hold the page tables.
  2. Stack top defined as `CONFIG_SPL_STACK` in `include/configs/imx6_spl.h`
6. Initializes CAAM security hardware
  1. Initializes RNG capabilities of the CAAM
7. Attempt to read and hide a unique secret device identity from the SoC.
  1. This will only succeed if the system has been fused for High Assurance Boot
8. Load U-Boot proper and OP-TEE binaries using a Flattened Image Tree (FIT)
  1. Loads the FIT header from MMC sector `CONFIG_SYS_MMCSD_RAW_MODE_U_BOOT_SECTOR`. The default for ARCH\_MX6 is defined in `common/spl/Kconfig`, but a defconfig can override the value.
    - A FIT is a single binary which both stores the U-Boot and OP-TEE images, and encodes their load addresses and entry points
    - The FIT source file (`image_source.its`) describes the structure of the FIT to the U-Boot `mkimage` tool which is responsible for assembling the image. The load and entry addresses for U-Boot (`CONFIG_SYS_TEXT_BASE`) and OP-TEE (`CFG_TEE_LOAD_ADDR`) are updated automatically when the firmware is built
    1. Verifies that the signature of the config block in the FIT matches the public key baked in SPL's Device Tree Blob (DTB) at image creation time.
    2. Verifies that the hashes for U-Boot and OP-TEE match the values stored in the signed FIT config.
    3. Loads U-Boot and OP-TEE to memory based on offsets stored in the FIT
    4. Identifies that the OP-TEE image is bootable using the FIT, disables caches and interrupts, then jumps into OP-TEE.

Memory layout just before jump to OP-TEE:

```

1  DRAM
2  +-----+
3  |
4  |
5  +-----+
6  |
7  | U-Boot proper
8  +-----+ 0x17800000 (CONFIG_SYS_TEXT_BASE)
9  |
10 |
11 +-----+
12 |
13 | OP-TEE
14 +-----+ 0x10A00000 (CFG_TEE_LOAD_ADDR)
15 |
16 |
17 +-----+ 0x10000000 (start of DRAM)
18
19 SRAM
20 +-----+ 0x00940000
21 | heap (grows down) | 0x0093C000 (CONFIG_SPL_STACK)
22 |
23 | stack (grows down)
24 |
25 |
26 | bss
27 | data
28 | text
29 | SPL
30 |
31 +-----+ 0x00908000 (CONFIG_SPL_TEXT_BASE)
32 |
33 |
34 | page tables (16k)
35 +-----+ 0x00900000

```

### 8.3 OP-TEE

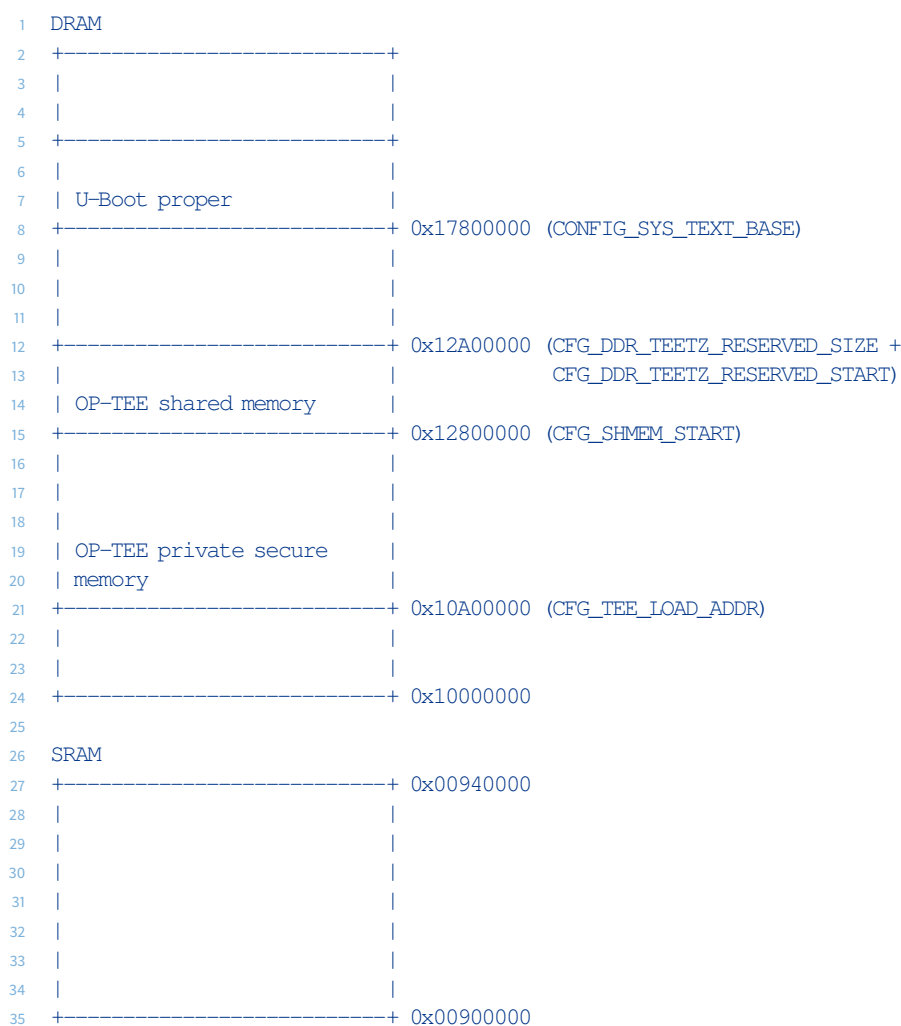
OP-TEE is a trusted operating system that runs in ARM TrustZone. It implements a trusted execution environment that can host trusted applications, and it implements the ARM Power State Coordination Interface (PSCI).

The reference implementation of OP-TEE

1. Begins execution at `core/arch/arm/kernel/generic_entry_a32.S : _start`
  1. OP-TEE's load address is `CFG_TEE_LOAD_ADDR` defined in `optee_os/core/arch/arm/plat-imx/platform_config.h`
2. Does low-level CPU initialization

1. Disallows unaligned access
2. Configures non-secure world (NS) access to CPU features
3. Configures and enables SCU
4. Configures i.MX6 CSU
5. Enables debug console UART
6. Enables MMU and L1 cache
7. Sets up secure monitor code
8. Initializes GIC
9. Initializes TEE core
10. Initializes drivers and services
11. Jumps to nonsecure world at the address passed in LR at entry, which should be the U-Boot proper entry point in DRAM

Memory layout just before jump to U-Boot proper:



## 8.4 U-Boot Proper

U-Boot proper is the full U-Boot binary with scripting support, filesystem support, command support, and hardware support. U-Boot proper executes in normal world and initializes hardware, then loads UEFI and jumps to UEFI.

The reference implementation of U-Boot proper

1. Begins execution at `arch/arm/cpu/armv7/start.S : reset`
  1. Load address is `CONFIG_SYS_TEXT_BASE` defined in `configs/mx6sabresd_nt_defconfig`
2. Does low-level CPU initialization
3. Executes `arch/arm/lib/crt0.S : _main`
4. Executes `common/board_f.c : board_init_f`
  1. Hardware initialization
  2. Muxing
  3. Clocks
  4. Console UART
5. Relocates to top of DRAM
6. Enables L1 cache
7. Does hardware initialization (USB, ENET, PCI, SPI, I2C, PMIC, Thermal, etc.)
8. Runs the boot command. The boot command is a script defined by `CONFIG_BOOTCOMMAND` which is defined in `configs/mx6sabresd_nt_defconfig`. This script
  1. Initializes a global page.
  2. Loads `uefi.fit` from a specified MMC device / FAT partition to a specified address in DRAM.
  3. Calls `bootm` on `uefi.fit` in memory, which will load UEFI to its `BaseAddress`.
  4. `Bootm` then disables caches and interrupts and jumps to UEFI's entry point, which is specified as `load` and `entry` in `imx-iotcore/build/firmware/boardname/uefi.its`. This address must match the values defined by `BaseAddress` in `imx-edk2-platforms/Silicon/NXP/iMX6Pkg/iMX6CommonFdf.inc`

Memory map just before jumping to UEFI:

```
1  DRAM
2  +-----+
3  |                                     |
4  | relocated U-Boot and               |
5  | stack, heap                       |
6  +-----+
7  |                                     |
8  |                                     |
```



9		
10	+-----+	
11		
12	U-Boot proper	
13	+-----+	0x17800000 (CONFIG_SYS_TEXT_BASE)
14		
15		
16		
17	+-----+	0x12A00000 (CFG_DDR_TEETZ_RESERVED_SIZE +
18		CFG_DDR_TEETZ_RESERVED_START)
19	OP-TEE shared memory	
20	+-----+	0x12800000 (CFG_SHMEM_START)
21		
22		
23		
24	OP-TEE private secure	
25	memory	
26	+-----+	0x10A00000 (CFG_TEE_LOAD_ADDR)
27		
28	UEFI	
29	+-----+	0x10820000 (BaseAddress, load/entry in uefi.its)
30		
31		
32	+-----+	0x10000000
33		
34	SRAM	
35	+-----+	0x00940000
36		
37		
38		
39		
40		
41		
42		
43	+-----+	0x00900000

## 8.5 UEFI

UEFI prepares for Windows and starts the Windows boot manager. The Windows boot manager (bootmgr) and bootloader (winload) are written as UEFI applications, and must run within the UEFI environment.

The reference implementation of UEFI

1. Begins execution at `ArmPlatformPkg/PrePi/Arm/ModuleEntryPoint.S : _ModuleEntryPoint`
  1. Load address is `BaseAddress` defined in each platform's fdf file such as `edk2-platforms/Platform/NXP/Sabre_iMX6Q_1GB/Sabre_iMX6Q_1GB.fdf`
2. Does hardware initialization

3. Unless `CONFIG_NOT_SECURE_UEFI=1` is set, the authenticated variable store Trusted Application (TA) is loaded by `imx-edk2-platforms/Platform/Microsoft/OpteeClientPkg/Drivers/AuthVarsDxe.c`. This TA is responsible for storing non-volatile variables in eMMC RPMB.
4. The Authvar TA may also contain Secure Boot keys. If the keys are present UEFI will enable Secure Boot and verify the signatures on all subsequent components as they are loaded.
5. Unless `CONFIG_NOT_SECURE_UEFI=1` is set a firmware TPM TA is also loaded by `imx-edk2-platforms/Platform/Microsoft/OpteeClientPkg/Library/Tpm2DeviceLibOptee/Tpm2DeviceLibOptee.c`. The TPM also uses RPMB for non-volatile secure storage. UEFI measures each subsequent component as it is loaded and saves these values in Platform Configuration Registers (PCRs) in the TPM. Windows will use these measurements to verify the system is secure and unlock BitLocker encrypted drives.
6. Sets up structures for handoff to Windows
7. Loads and runs bootmgr

Bootmgr then orchestrates the process of loading Windows.

Memory map just before jumping to bootmgr:

1	DRAM	
2	+-----+ +	
3		
4	UEFI stack	
5	+-----+ +	
6		
7		
8	+-----+ +	0x12A00000 (CFG_DDR_TEETZ_RESERVED_SIZE +
9		CFG_DDR_TEETZ_RESERVED_START)
10	OP-TEE shared memory	
11	+-----+ +	0x12800000 (CFG_SHMEM_START)
12		
13		
14		
15	OP-TEE private secure	
16	memory	
17	+-----+ +	0x10A00000 (CFG_TEE_LOAD_ADDR)
18		
19	UEFI	
20	+-----+ +	0x10820000 (BaseAddress, uefi_addr)
21	Global data	
22	+-----+ +	0x10817000 (PcdGlobalDataBaseAddress)
23	TPM2 control area	
24	+-----+ +	0x10814000 (PcdTpm2AcpiBufferBase)
25		
26	+-----+ +	0x10800000
27	Frame Buffer	
28	+-----+ +	0x10000000 (PcdFrameBufferBase)
29		
30	SRAM	
31	+-----+ +	0x00940000
32		

```

33 |                                     |
34 |                                     |
35 |                                     |
36 |                                     |
37 |                                     |
38 |                                     |
39 +-----+ 0x00900000

```

## 8.6 SD/eMMC Layout

SD/eMMC is laid out as follows:

```

1  +-----+ Sector 0
2  | partition table |
3  |                 |
4  +-----+ Sector 2
5  | SPL_signed.imx |
6  |   IMX bootloader header |
7  |   SPL binary      |
8  |   DTB with image.fit key |
9  |   CSF data (for HAB) |
10 |                   |
11 +-----+
12 |                   |
13 |                   |
14 +-----+ Sector 136 (CONFIG_SYS_MMCSD_RAW_MODE_U_BOOT_SECTOR)
15 | image.fit |
16 |   OPTEE   |
17 |   U-Boot Proper |
18 |   DTB with uefi.fit key |
19 +-----+
20 |                   |
21 |                   |
22 +-----+
23         .
24         .
25         .
26 +-----+ Partition 2 (FAT)
27 | uefi.fit |
28 |         |
29 +-----+

```

## 9 Securing Peripherals on i.MX using OP-TEE

This chapter describes the process of configuring an i.MX peripheral for secure access using OP-TEE only. It also describes the Windows behavior toward i.MX peripherals.

Note: In the text to follow, we assume you are familiar with the required build tools, general [boot flow](#), and process to [build ARM32 firmware](#) or [build ARM64 firmware](#).

### 9.1 OP-TEE

Locking down specific peripherals for secure access occurs during the OP-TEE portion of boot, when OP-TEE configures the CSU.

By default, the CSU registers are initialized to allow access from both normal and secure world for all peripherals.

To override this default configuration, add an override entry to the `access_control` global array. You can find this array in `optee_os/core/arch/arm/plat-imx/imx6.c`

```
1 static struct csu_csl_access_control access_control[] = {/  
2     * TZASC1   - CSL16 [7:0] *//  
3     * TZASC2   - CSL16 [23:16] *//  
4     {16, ((CSU_TZ_SUPERVISOR << 0) | (CSU_TZ_SUPERVISOR << 16))},  
5 }
```

The first field is the CSU CSL register index to secure the required device. The second field is the required CSU CSL register value. This value will override the default CSU initialization value. OP-TEE provides some useful defines to create this value:

```
1 /  
2 *  
3 * Grant R+W access:  
4 * - Just to TZ Supervisor execution mode, and  
5 * - Just to a single device  
6 */  
7 #define CSU_TZ_SUPERVISOR      0x22/  
8  
9 *  
10 * Grant R+W access:  
11 * - To all execution modes, and  
12 * - To a single device  
13 */  
14 #define CSU_ALL_MODES         0xFF
```

Note: Each CSU CSL register is responsible for two peripheral devices. You must set the override value carefully to ensure you are securing the intended peripheral device.

## 9.2 Windows

Any access to a secure peripheral from a non-secure environment will cause system failure. To avoid this scenario, we have added code into the PEP driver to automatically read the CSU registers and determine if a Windows-enabled peripheral can be interacted with from a non-secure environment. If it can't, Windows will automatically hide the secured peripheral to avoid the potential system failure.

# 10 Building Windows 10 IoT Core for NXP i.MX Processors

## 10.1 Building the BSP

Before you start building the BSP, you need to have an archive with BSP sources downloaded and extracted. After that, you should get the following directory structure:

```
1  \ \ \
2  - %WORKSPACE%
3    |- cst
4    |- edk2
5    |- imx-edk2-platforms
6    |- imx-iotcore
7    |- optee_os
8    |- RIOT
9    |- u-boots
10 \ \ \
```

### 10.1.1 Required Tools

The following tools are required to build the driver packages and IoT Core FFU: Visual Studio 2017, Windows Kits (ADK/SDK/WDK), and the IoT Core OS Packages.

#### 10.1.1.1 Visual Studio 2017

- Make sure that you **install Visual Studio 2017 before the WDK** so that the WDK can install a required plugin.
- Download [Visual Studio 2017](#).
- During install select **Desktop development with C++**.
- During install select the following in the Individual components tab. If these options are not available try updating VS2017 to the latest release:
  - **VC++ 2017 version 15.9 v14.16 Libs for Spectre (ARM)**
  - **VC++ 2017 version 15.9 v14.16 Libs for Spectre (ARM64)**
  - **VC++ 2017 version 15.9 v14.16 Libs for Spectre (X86 and x64)**
  - **Visual C++ compilers and libraries for ARM**
  - **Visual C++ compilers and libraries for ARM64**

### 10.1.1.2 Windows Kits from Windows 10, version 1809

- **IMPORTANT: Make sure that any previous versions of the ADK and WDK have been uninstalled!**
- Install [ADK 1809](#)
- Install [WDK 1809](#)
  - Make sure that you allow the Visual Studio Extension to install after the WDK install completes.
- If the WDK installer says it could not find the correct SDK version, install [SDK 1809](#)

### 10.1.1.3 IoT Core OS Packages

- Visit the [Windows IoT Core Downloads](#) page and download “Windows 10 IoT Core Packages – Windows 10 IoT Core, version 1809 (LTSC)”.
- Open the iso and install [Windows\\_10\\_IoT\\_Core\\_ARM\\_Packages.msi](#)
- Install [Windows\\_10\\_IoT\\_Core\\_ARM64\\_Packages.msi](#) for ARM64 builds.

## 10.1.2 One-Time Environment Setup

Test certificates must be installed to generate driver packages on a development machine.

1. Open an Administrator Command Prompt.
2. Navigate to your BSP and into the folder `imx-iotcore\build\tools`.
3. Launch `StartBuildEnv.bat`.
4. Run `SetupCertificate.bat` to install the test certificates.

## 10.1.3 FFU Generation

1. Launch Visual Studio 2017 as Administrator.
2. Open the solution `imx-iotcore\build\solution\iMXPlatform\iMXPlatform.sln` located in the path where you have extracted BSP archive.
3. Change the build type from Debug to Release. Change the build flavor from ARM to ARM64 if building for iMX8.
4. To build press Ctrl-Shift-B or choose Build -> Build Solution from menu. This will compile all driver packages then generate the FFU.

5. Depending on the speed of the build machine FFU generation may take around 10-20 minutes.

Note: During the process of FFU creation you may error `Error: CreateUsnJournal: Unable to create USN journal, as one already exists on volume`. In this case it is needed to set the USN journal registry size to 1 Mb on your development PC by following command and restart PC:

```
PC: reg add HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\FileSystem /v NtfsAllowUsnMinSize1Mb /t REG_DWORD /d 1
```

6. After a successful build the new FFU will be located in `imx-iotcore\build\solution\iMXPlatform\Build\FFU\Sabre_iMX6Q_1GB\` for ARM builds and `imx-iotcore\build\solution\iMXPlatform\Build\FFU\NXPEVK_iMX8M_4GB` for ARM64 builds.
7. The FFU contains firmware components for the NXP IMX8M EVK with i.MX8M Quad Core SOM depending on build flavor. This firmware is automatically applied to the SD Card during the FFU imaging process.

### 10.1.3.1 Building the FFU for other boards

In order to build an FFU for another board you'll need to modify `GenerateFFU.bat` in the Build Scripts folder of the Solution Explorer. Comment out the default `Sabre_iMX6Q_1GB` or `NXPEVK_iMX8M_4GB` builds with `REM` and uncomment any other boards you want to build.

```
1 REM cd /d %BATCH_HOME%
2 REM echo "Building EVK_iMX6ULL_512MB FFU"
3 REM call BuildImage EVK_iMX6ULL_512MB EVK_iMX6ULL_512MB_TestOEMInput.xml
4
5 cd /d %BATCH_HOME%
6 echo "Building Sabre_iMX6Q_1GB FFU"
7 call BuildImage Sabre_iMX6Q_1GB Sabre_iMX6Q_1GB_TestOEMInput.xml
```

### 10.1.4 Building the FFU with the IoT ADK AddonKit

1. Build the `GenerateBSP` project to create a BSP folder in the root of the repository.
2. Clone the [IoT ADK AddonKit](#).
3. Follow the [Create a basic image instructions](#) from the IoT Core Manufacturing guide with the following changes.
  - When importing a BSP use one of the board names from the newly generated BSP folder in the `imx-iotcore` repo. `Import-IoTBSP Sabre_iMX6Q_1GB <Path to imx-iotcore\BSP>`
  - When creating a product use the same board name from the BSP import. `Add-IoTProduct ProductA Sabre_iMX6Q_1GB`



# 11 Building and Updating ARM32 Firmware

This chapter describes how to set up a build environment to build the latest ARM32 firmware, update the firmware on the SD Card for testing and development, and include the new firmware in the FFU builds.

## 11.1 Setting up your build environment

1) Set up a Linux environment.

- Dedicated Linux system
- Linux Virtual Machine
- Windows Subsystem for Linux ([WSL setup instructions](#))
- Note: We validate with both Ubuntu in WSL and standalone Ubuntu machines.

2) Update and install build tools

```
1 $ sudo apt-get update
2 $ sudo apt-get upgrade
3 $ sudo apt-get install build-essential python python-dev python-crypto python-wand
   device-tree-compiler bison flex swig iasl uuid-dev wget git bc libssl-dev python3-setuptools
   python3 python3-pyelftools
4 $ pushd ~
5 $ wget https://releases.linaro.org/components/toolchain/binaries/6.4-2017.11/arm-linux-gnueabi/f/
   gcc-linaro-6.4.1-2017.11-x86_64_arm-linux-gnueabi/f.tar.xz
6 $ tar xf gcc-linaro-6.4.1-2017.11-x86_64_arm-linux-gnueabi/f.tar.xz
7 $ rm gcc-linaro-6.4.1-2017.11-x86_64_arm-linux-gnueabi/f.tar.xz
8 $ popd
```

3) Download and extract the BSP archive. At this point your directory structure should look like the following

```
1 - %WORKSPACE%
2   |- cst
3   |- edk2
4   |- firmware-imx-8.1
5   |- imx-atf
6   |- imx-edk2-platforms
7   |- imx-iotcore
8   |- imx-mkimage
9   |- mu_platform_nxp
10  |- optee_os
11  |- RIOT
12  |- u-boots
```

4) Build firmware to test the setup. Adding “-j 20” to make will parallelize the build and speed it up

significantly on WSL, but since the firmwares build in parallel it will be more difficult to diagnose any build failures. You can customize the number to work best with your system.

```
1 $ cd imx-iotcore/build/firmware/boardname
2 $ make
```

5) After a successful build you should have several output files:

```
1 firmware_fit.merged - Contains SPL, OP-TEE, and U-Boot proper
2 uefi.fit - Contains the UEFI firmware
```

## 11.2 Adding updated firmware to your FFU

1) To make the updated firmware a part of your FFU build, you must copy the firmwares to your board's Package folder in imx-iotcore.

- Copy uefi.fit into /board/boardname/Package/BootFirmware
- Copy firmware\_fit.merged into /board/boardname/Package/BootLoader
- You can also use the following make command to copy uefi.fit and firmware\_fit.merged to the correct package directories.

```
1 $ make update-ffu
```

2) When preparing to commit your changes, you should use the following make command to save your OP-TEE SDK and the commit versions of your firmware automatically in your board folder.

```
1 $ make commit-firmware
```

## 11.3 Deploying firmware to an SD card manually

### 11.3.1 Bootable Firmware without installing an FFU

If you want to rapidly deploy and test firmware configurations without needing the full Windows boot, you can prepare an SD card manually to boot only the firmware stages.

The SD card must have two partitions in the following order:

- 4MB partition at the start of the disk, no filesystem. This is where U-Boot and OP-TEE get deployed.
- 50MB partition formatted fat32, optionally labeled efi. This is where UEFI gets deployed.

Here are the steps to run in an admin CMD to prepare an SD card in Windows:

```
1 powershell Get-WmiObject Win32_DiskDrive
2 REM Find the SD card in that list and use the number after PhysicalDrive as your disk number.
3 diskpart
4 list disk
5 sel disk <#>
6 list part
7 REM Check the partitions to make sure this disk is actually your SD card.
8 clean
9 create partition primary size=4
10 create partition primary size=50
11 format quick fs=fat32 label=EFI
12 assign
13 list vol
14 exit
```

### 11.3.2 Deploying U-Boot and OP-TEE (firmware\_fit.merged) for development

On Windows you can use [DD for Windows](#) from an administrator command prompt to deploy firmware\_fit.merged. Be careful that you use the correct `of` and `seek`, DD will not ask for confirmation.

```
1 powershell Get-WmiObject Win32_DiskDrive
2 dd if=firmware_fit.merged of=\\.\PhysicalDrive<X> bs=512 seek=2
```

Where `PhysicalDrive<X>` is the DeviceID of your SD card as shown by `Get-WmiObject`.

You might get the output: `Error reading file: 87 The parameter is incorrect`. This error can be ignored.

If you are working on a dedicated Linux machine (not WSL or VM) use:

```
1 dd if=firmware_fit.merged of=/dev/sdX bs=512 seek=2
```

### 11.3.3 Deploying UEFI (uefi.fit) for development

Copy `uefi.fit` over to the EFI partition on your SD card.

### 11.3.4 Updating the TAs in UEFI

A firmware TPM TA, and UEFI authenticated variable TA, are included with EDK2. Generally, these TAs should work on any ARM32 system where OP-TEE is running, and eMMC RPMB is available.

These binaries are built using OpenSSL by default but can also be built using WolfSSL (See `FTPM_FLAGS` and `AUTHVARS_FLAGS` in `common.mk`).

They are omitted from the firmware if the `CONFIG_NOT_SECURE_UEFI=1` flag is set. This is useful for early development work if RPMB storage is not functioning yet, or if eMMC is not present on the device.

They can be rebuilt using:

```
1 make update_tas
```

This updates the binaries included in the imx-edk2-platforms repo. ##### Clearing RPMB If the TAs are changed significantly, or the storage becomes corrupted, it may be necessary to clear the OP-TEE secure filesystem in RPMB. This can be done by building OP-TEE with the `CFG_RPMB_RESET_FAT=y` flag set. This flag will cause OP-TEE to erase its FAT metadata when it first accesses RPMB during every boot. This effectively clears all the data stored by the TAs. After clearing the RPMB OP-TEE should be switched back to `CFG_RPMB_RESET_FAT=n` to allow variables to persist again.

## 12 Building and Updating ARM64 Firmware

This chapter describes the process of setting up a build-environment to build the latest firmware, update the firmware on the SD Card for testing and development, and include the new firmware in the FFU builds.

Note: The UEFI build environment has changed for 1903 and any existing build environment must be updated.

### 12.1 Setting up your build environment

1) Set up a Linux environment.

- Dedicated Linux system
- Linux Virtual Machine
- Windows Subsystem for Linux ([WSL setup instructions](#))
- Note: We validate with both Ubuntu in WSL and standalone Ubuntu machines.

2) Update and install build tools

```
1 sudo apt-get update
2 sudo apt-get upgrade
3 sudo apt-get install attr build-essential python python-dev python-crypto python-wand
   device-tree-compiler bison flex swig iasl uuid-dev wget git bc libssl-dev zlib1g-dev python3-pip
4 *** new for 1903 UEFI
5 sudo apt-get install gcc g++ make python3 mono-devel
6 ***
7 pushd ~
8 wget https://releases.linaro.org/components/toolchain/binaries/7.2-2017.11/aarch64-linux-gnu/
   gcc-linaro-7.2.1-2017.11-x86_64_aarch64-linux-gnu.tar.xz
9 tar xf gcc-linaro-7.2.1-2017.11-x86_64_aarch64-linux-gnu.tar.xz
10 rm gcc-linaro-7.2.1-2017.11-x86_64_aarch64-linux-gnu.tar.xz
11 popd
```

3) At this point your directory structure should look like the following

```
1 - %WORKSPACE%
2   |- cst
3   |- edk2
4   |- firmware-imx-8.1
5   |- imx-atf
6   |- imx-edk2-platforms
7   |- imx-iotcore
8   |- imx-mkimage
9   |- mu_platform_nxp
10  |- optee_os
11  |- RIoT
```

12       |- u-boots

- 4) Build firmware to test the setup. Adding “-j 20” to make will parallelize the build and speed it up significantly on WSL, but since the firmwares build in parallel it will be more difficult to diagnose any build failures. You can customize the number to work best with your system.

```
1  # U-Boot
2  export CROSS_COMPILE=~/.gcc-linaro-7.2.1-2017.11-x86_64-aarch64-linux-gnu/bin/aarch64-linux-gnu-
3  export ARCH=arm64
4
5  pushd u-boots/u-boot-imx_arm64
6
7  make imx8mq_evk_nt_defconfig
8  ---or---
9  make imx8mm_evk_nt_defconfig
10
11 make
12 popd
13
14
15 # Arm Trusted Firmware
16
17 export CROSS_COMPILE=~/.gcc-linaro-7.2.1-2017.11-x86_64-aarch64-linux-gnu/bin/aarch64-linux-gnu-
18 export ARCH=arm64
19
20 pushd imx-atf
21 make PLAT=imx8mq SPD=opteed bl31
22 ---or---
23 make PLAT=imx8mm SPD=opteed bl31
24 popd
25
26 # OP-TEE OS
27
28 export -n CROSS_COMPILE
29 export -n ARCH
30 export CROSS_COMPILE64=~/.gcc-linaro-7.2.1-2017.11-x86_64-aarch64-linux-gnu/bin/aarch64-linux-gnu-
31 pushd optee_os/optee_os_arm64
32
33 make PLATFORM=imx PLATFORM_FLAVOR=mx8mqevk \
34   CFG_TEE_CORE_DEBUG=n CFG_TEE_CORE_LOG_LEVEL=2 \
35   CFG_RPMB_FS=y CFG_RPMB_TESTKEY=y CFG_RPMB_WRITE_KEY=y CFG_REE_FS=n \
36   CFG_IMXCRIPT=y CFG_CORE_HEAP_SIZE=98304
37 ---or---
38 make PLATFORM=imx PLATFORM_FLAVOR=mx8mmevk \
39   CFG_TEE_CORE_DEBUG=n CFG_TEE_CORE_LOG_LEVEL=2 \
40   CFG_RPMB_FS=y CFG_RPMB_TESTKEY=y CFG_RPMB_WRITE_KEY=y CFG_REE_FS=n \
41   CFG_IMXCRIPT=y CFG_CORE_HEAP_SIZE=98304
42
43 # debug
44 # make PLATFORM=imx PLATFORM_FLAVOR=mx8mqevk \
45 #   CFG_TEE_CORE_DEBUG=y CFG_TEE_CORE_LOG_LEVEL=3 \
46 #   CFG_RPMB_FS=y CFG_RPMB_TESTKEY=y CFG_RPMB_WRITE_KEY=y CFG_REE_FS=n \
47 #   CFG_TA_DEBUG=y CFG_TEE_CORE_TA_TRACE=1 CFG_TEE_TA_LOG_LEVEL=2 \
48 #   CFG_IMXCRIPT=y CFG_CORE_HEAP_SIZE=98304
49 # ---or---
```

```
50 # make PLATFORM=imx PLATFORM_FLAVOR=mx8mmevk \  
51 #   CFG_TEE_CORE_DEBUG=y CFG_TEE_CORE_LOG_LEVEL=3 \  
52 #   CFG_RPMB_FS=y CFG_RPMB_TESTKEY=y CFG_RPMB_WRITE_KEY=y CFG_REE_FS=n \  
53 #   CFG_TA_DEBUG=y CFG_TEE_CORE_TA_TRACE=1 CFG_TEE_TA_LOG_LEVEL=2 \  
54 #   CFG_IMXCRIPT=y CFG_CORE_HEAP_SIZE=98304  
55  
56 ${CROSS_COMPILE64}objcopy -O binary ./out/arm-plat-imx/core/tee.elf ./out/arm-plat-imx/tee.bin  
57 popd  
58  
59 # OP-TEE Trusted Applications  
60  
61 export TA_DEV_KIT_DIR=../../../../optee_os/out/arm-plat-imx/export-ta_arm64  
62 export TA_CROSS_COMPILE=~/.gcc-linaro-7.2.1-2017.11-x86_64_aarch64-linux-gnu/bin/aarch64-linux-gnu-  
63 export TA_CPU=cortex-a53  
64  
65 # imx-mkimage  
66  
67 export CROSS_COMPILE=~/.gcc-linaro-7.2.1-2017.11-x86_64_aarch64-linux-gnu/bin/aarch64-linux-gnu-  
68 export ARCH=arm64  
69  
70 pushd imx-mkimage/imx8M  
71  
72 cp ../../firmware-imx-8.1/firmware/ddr/synopsys/lpddr4_pmu_train*.bin .  
73 cp ../../firmware-imx-8.1/firmware/hdmi/cadence/signed_hdmi_imx8m.bin .  
74 cp ../../optee_os/optee_os_arm64/out/arm-plat-imx/tee.bin .  
75  
76 cp ../../imx-atf/build/imx8mq/release/bl31.bin .  
77 ---or---  
78 cp ../../imx-atf/build/imx8mm/release/bl31.bin .  
79  
80 cp ../../u-boots/u-boot-imx_arm64/u-boot-nodtb.bin .  
81 cp ../../u-boots/u-boot-imx_arm64/spl/u-boot-spl.bin .  
82  
83 cp ../../u-boots/u-boot-imx_arm64/arch/arm/dts/fsl-imx8mq-evk.dtb .  
84 ---or---  
85 cp ../../u-boots/u-boot-imx_arm64/arch/arm/dts/fsl-imx8mm-evk.dtb .  
86  
87 cp ../../u-boots/u-boot-imx_arm64/tools/mkimage .  
88  
89 mv mkimage mkimage_uboot  
90  
91 cd ..  
92  
93 make SOC=IMX8M flash_hdmi_spl_uboot  
94 ---or---  
95 make SOC=IMX8MM flash_hdmi_spl_uboot  
96  
97 popd  
98  
99 # UEFI  
100 # note: On Windows Ubuntu, ignore Python errors during build specifically like  
101 # "ERROR - Please upgrade Python! Current version is 3.6.7. Recommended minimum is 3.7."  
102  
103 # setup  
104 pushd mu_platform_nxp  
105 export GCC5_AARCH64_PREFIX=~/.gcc-linaro-7.2.1-2017.11-x86_64_aarch64-linux-gnu/bin/
```

```
aarch64-linux-gnu-
106 pip3 install -r requirements.txt --upgrade
107
108 python3 NXP/MCIMX8M_EVK_4GB/PlatformBuild.py --setup
109 # if error here about NugetDependency.global_cache_path, then make sure mono-devel package is
    installed
110 # using apt-get as listed in "Update and install build tools" above.
111
112 cd MU_BASECORE
113 make -C BaseTools
114 cd ..
115
116 popd
117
118 # clean
119 pushd mu_platform_nxp
120 rm -r Build
121 rm -r Config
122 popd
123
124 # build
125 pushd mu_platform_nxp
126
127
128 export GCC5_AARCH64_PREFIX=~/.gcc-linaro-7.2.1-2017.11-x86_64_aarch64-linux-gnu/bin/
    aarch64-linux-gnu-
129
130 python3 NXP/MCIMX8M_EVK_4GB/PlatformBuild.py -V TARGET=RELEASE \
131     PROFILE=DEV MAX_CONCURRENT_THREAD_NUMBER=20
132 ---or---
133 python3 NXP/MCIMX8M_MINI_EVK_2GB/PlatformBuild.py -V TARGET=RELEASE \
134     PROFILE=DEV MAX_CONCURRENT_THREAD_NUMBER=20
135
136 # debug
137 # python3 NXP/MCIMX8M_EVK_4GB/PlatformBuild.py -V TARGET=DEBUG \
138 #     PROFILE=DEV MAX_CONCURRENT_THREAD_NUMBER=20
139 # ---or---
140 # python3 NXP/MCIMX8M_MINI_EVK_2GB/PlatformBuild.py -V TARGET=DEBUG \
141 #     PROFILE=DEV MAX_CONCURRENT_THREAD_NUMBER=20
142
143 cd Build/MCIMX8M_EVK_4GB/RELEASE_GCC5/FV
144 ---or---
145 cd Build/MCIMX8M_MINI_EVK_2GB/RELEASE_GCC5/FV
146
147 cp ../../../../../../imx-iotcore/build/firmware/its/uefi_imx8_unsigned.its .
148 ../../../../../../u-boots/u-boot-imx_arm64/tools/mkimage -f uefi_imx8_unsigned.its -r uefi.fit
149
150 popd
```

- 5) After a successful build you should have several output files: “bash imx-mkimage/iMX8M/flash.bin  
- Contains SPL, ATF, OP-TEE, and U-Boot proper

mu\_platform\_nxp/Build/MCIMX8M\_EVK\_4GB/RELEASE\_GCC5/FV/uefi.fit - Contains the UEFI  
firmware —or— mu\_platform\_nxp/Build/MCIMX8M\_MINI\_EVK\_2GB/RELEASE\_GCC5/FV/uefi.fit -



Contains the UEFI firmware ““

## 12.2 Adding updated firmware to your ARM64 FFU

1) To make the updated firmware a part of your FFU build, you must copy the firmwares to your board's Package folder in imx-iotcore.

- Copy uefi.fit into /board/boardname/Package/BootFirmware
- Copy flash.bin into /board/boardname/Package/BootLoader

```
1 cp imx-mkimage/iMX8M/flash.bin imx-iotcore/build/board/NXPEVK_iMX8M_4GB/Package/BootLoader/flash.bin
2 cp mu_platform_nxp/Build/MCIMX8M_EVK_4GB/RELEASE_GCC5/FV/uefi.fit imx-iotcore/build/board/NXPEVK_iMX8M_4GB/Package/BootFirmware/uefi.fit
3 ---or---
4 cp imx-mkimage/iMX8M/flash.bin imx-iotcore/build/board/NXPEVK_iMX8M_Mini_2GB/Package/BootLoader/flash.bin
5 cp mu_platform_nxp/Build/MCIMX8M_MINI_EVK_2GB/RELEASE_GCC5/FV/uefi.fit imx-iotcore/build/board/NXPEVK_iMX8M_Mini_2GB/Package/BootFirmware/uefi.fit
```

## 12.3 Deploying firmware to an SD card manually

### 12.3.1 Deploying U-Boot, ATF, OP-TEE (flash.bin) and UEFI (uefi.fit) for development

On Windows you can use [DD for Windows](#) from an administrator command prompt to deploy flash.bin and uefi.fit. Be careful that you use the correct `of` and `seek`, DD will not ask for confirmation.

```
1 powershell Get-WmiObject Win32_DiskDrive
2 dd if=flash.bin of=\\.\PhysicalDrive<X> bs=512 seek=66
3
4 dd if=uefi.fit of=\\.\PhysicalDrive<X> bs=1024 seek=2176
```

Where `PhysicalDrive<X>` is the DeviceID of your SD card as shown by `Get-WmiObject`.

You might get the output: `Error reading file: 87 The parameter is incorrect`. This error can be ignored.

If you are working on a dedicated Linux machine (not WSL or VM) use:

```
1 dd if=flash.bin of=/dev/sdX bs=512 seek=66
2
3 dd if=uefi.fit of=/dev/sdX bs=1024 seek=2176
```

## 13 Adding New Boards and Drivers to the BSP

### 13.1 Adding a New Board

This chapter describes the process of setting up a new board configuration for FFU image builds.

#### 13.1.1 Initialize a new board configuration

1. Open PowerShell and run `imx-iotcore\build\tools\NewiMX6Board.ps1 <NewBoardName>`.
  - Note: `<NewBoardName>` must follow the schema of `BoardName_SoCType_MemoryCapacity`. See `imx-iotcore\build\board` for examples.
  - The following instructions assume an example board named **MyBoard\_iMX6Q\_1GB**.
  - If the script is blocked by execution policy, invoke a powershell from an administrator command prompt to bypass the powershell script execution policy: `powershell.exe -executionpolicy bypass .\NewiMX6Board.ps1 <NewBoardName>`
2. This step will create a new board configuration in `imx-iotcore\build\board\` and a new firmware folder in `imx-iotcore\build\firmware`.

#### 13.1.2 Setup the solution in Visual Studio

1. Open up the Solution Explorer view (Ctrl + Alt + L).
2. Right-click the Board Packages folder and select Add Existing Project.
3. Select `imx-iotcore\build\board\MyBoard_iMX6Q_1GB\Package\MyBoardPackage.vcxproj`.
4. Right-click your new project => Build Dependencies => Project Dependencies then select **HalExtiMX6Timers**, **imxusdhc**, and **mx6pep**.
  - For an i.MX7 project select **HalExtiMX7Timers** and **imxusdhc**.
5. Right-click the GenerateTestFFU project => Build Dependencies => Project Dependencies then select your new project from the list.

#### 13.1.3 Update the firmware for your board

1. Port the firmware to your board following the [i.MX Porting Guide](#).
2. Modify `imx-iotcore\build\firmware\ContosoBoard_iMX6Q_2GB\Makefile` with the appropriate values for all CONFIG options. This is used by the makefile to configure each firmware build.

```
1  # Select the defconfig file to use in U-Boot
2  UBOOT_CONFIG=mx6sabresd_nt_defconfig
3
4  # Select the DSC file name to use in EDK2
5  EDK2_DSC=Sabre_iMX6Q_1GB
6  # Select the subdirectory of the Platform folder for this board
7  EDK2_PLATFORM=NXP/Sabre_iMX6Q_1GB
8  # Select DEBUG or RELEASE build of EDK2
9  EDK2_DEBUG_RELEASE=RELEASE
10
11 # Select the FIT Image Tree Source file used to bundle and sign U-Boot and OP-TEE
12 UBOOT_OPTEE_ITS=uboot_optee_unsigned.its
13 # Select the FIT Image Tree Source file used to bundle and sign UEFI
14 UEFI_ITS=uefi_unsigned.its
15
16 all: firmware_fit.merged firmwareversions.log
17
18 include ../Common.mk
19
20 .PHONY: $(OPTEE)
21 # Select the PLATFORM for OP-TEE to build
22 $(OPTEE):
23     $(MAKE) -C $(OPTEE_ROOT) O=$(OPTEE_OUT) PLATFORM=imx-mx6qsabresd \
24     $(OPTEE_FLAGS_IMX6)
```

3. This new firmware folder and updated makefile will allow you to use the common firmware makefile to build your firmwares. The makefile can be invoked from `imx-iotcore\build\firmware`. This can be run directly from WSL, on a Linux host, or in CMD by prepending make with “wsl”

WSL and Linux:

```
1  cd imx-iotcore/build/firmware
2  make MyBoard_iMX6Q_1GB
```

CMD and PowerShell:

```
1  cd imx-iotcore\build\firmware
2  wsl make MyBoard_iMX6Q_1GB
```

### 13.1.4 Build the FFU in Visual Studio

1. Edit **GenerateFFU.bat** in Build Scripts and comment out the board build target using REM. This will speed up FFU generation time since it will only build the FFU for your board.
2. Select the Release or Debug build target, then right click and build GenerateTestFFU.
3. After FFU generation completes, your FFU will be available in `imx-iotcore\build\solution\iMXPlatform\Build\FFU\MyBoard_iMX6Q_1GB` and can be flashed to an SD card following the instructions in the [IoT Core Manufacturing Guide](#).

### 13.1.5 Board Package Project Meanings

The board package projects are used to build the following packages:

- **Platform Firmware:** BootFirmware, BootLoader
- **Platform Identity:** SystemInformation
- **Filesystem Layout:** DeviceLayoutProd, OEMDevicePlatform
- **UpdateOS Drivers:** SVPlatExtensions

The board packages have a dependency on HalExtiMX6Timers, mx6pep, and imxusdhc because those are the minimum set of boot critical drivers for i.MX6, so the UpdateOS Drivers package SVPlatExtensions requires them.

## 13.2 Adding a New Driver

This chapter describes the process of adding a new driver to FFU image builds.

### 13.2.1 Adding a New Driver to the Solution

1. Right-click the Drivers folder in Solution Explorer and add a New Project.
2. Select Windows Drivers then Kernel Mode Driver or User Mode Driver. Set the name of your driver and set the location to `imx-iotcore\driver`. The rest of the instructions assume the name `MyTestDriver`. After the project has been created, select it in Solution Explorer and save your changes with Ctrl+S.
3. Copy the reference TestDriver.wm.xml from `imx-iotcore\build\tools\TestDriver.wm.xml` into your project directory and rename it after your project. The rest of the instructions assume the name `MyTestDriver.wm.xml`.
4. Edit `MyTestDriver.wm.xml` to set the name, namespace, owner, legacyName, and INF source. The legacyName field determines the name of your driver cab package.
5. Open `MyTestDriver.vcxproj` in a text editor and add the following XML as the first entry under the `<Project>` tag at the top of the file. Change the wm.xml names to match your new one, then save and close the file.

```
1 <Import Project="$(SolutionDir)..\..\common.props"/>
2 <ItemGroup>
3   <PkgGen Include="MyTestDriver.wm.xml">
4     <AdditionalOptions>/universalbsp</AdditionalOptions>
5   </PkgGen>
6 </ItemGroup>
```

6. Navigate back to Visual Studio and select Reload Solution if it prompts.
7. Modify your driver's inf to store driver in the Driver Store. Change the DIRID number in DefaultDestDir and ServiceBinary from 12 to 13. This will cause your driver binary to be stored in `C:\Windows\System32\DriverStore` instead of `C:\Windows\System32\Drivers`.

```
1 [DestinationDirs]
2 DefaultDestDir = 13
3 ...
4 ServiceBinary = %13%\MyTestDriver.sys
```

8. Right-click the GenerateTestFFU project, select Project Dependencies, then check the box next to your new project.

### 13.2.2 Adding a Driver to the FFU

1. After adding your driver to the project and building it, confirm that your driver has built and placed its binaries and .cab file inside of `imx-iotcore\build\solution\iMXPlatform\Build\Release\ARM`.
2. Open the Device Feature Manifest of your board (for example, `imx-iotcore\build\board\Sabre_iMX6DL_1GB\InputFMs\Sabre_iMX6DL_1GB_DeviceFM.xml`).
3. Add a new PackageFile section to the XML and modify it with the package name set by legacyName in your wm.xml. Change FeatureID to match the other drivers in your board file, or create a new FeatureID for your feature.

```
1 <PackageFile Path="%BSPPKG_DIR%" Name="MyOEM.MyNamespace.MyTestDriver.cab">
2   <FeatureIDs>
3     <FeatureID>MYNEWFEATURE_DRIVERS</FeatureID>
4   </FeatureIDs>
5 </PackageFile>
```

4. **If you did not create a new FeatureID, skip this step.** If you created a new FeatureID in your DeviceFM.xml then you must select it in your ProductionOEMInput.xml and TestOEMInput.xml files to include the driver in the respective image (for example `imx-iotcore\build\board\Sabre_iMX6DL_1GB\Sabre_iMX6DL_1GB_TestOEMInput.xml` and `imx-iotcore\build\board\Sabre_iMX6DL_1GB\Sabre_iMX6DL_1GB_ProductionOEMInput.xml`).

```
1 <OEM>
2   <Feature>MYNEWFEATURE_DRIVERS</Feature>
3   <Feature>IMX_DRIVERS</Feature>
4 </OEM>
```

5. Clean the solution then rebuild the GenerateTestFFU project and your driver will be included in the FFU.

## 14 i.MX Porting Guide

This chapter describes the process of initializing Windows on new i.MX6 and i.MX7 boards. The firmware and drivers can be ported to new boards by changing settings that differ from board to board. These settings include:

- SoC type (i.MX6 Quad/QuadPlus/Dual/DualLite/Solo/SoloX, i.MX7 Dual)
- MMDC initialization
- DDR size
- Console UART selection
- Boot device selection
- Pin muxing
- Exposed devices
- Off-SoC peripherals

The general procedure is to bring up each of the firmware components in sequence, then create packages, and finally create an FFU configuration. By the end, a new board configuration will be located in the repository that will build an FFU for your board. It is important to create new configurations for your board instead of modifying existing ones, so that you can easily integrate code changes from our repositories. We encourage you to submit your changes by using a pull request to our repositories so that we can make code changes without breaking your build.

This guide is structured in two parts:

1. Create Windows Compatible U-Boot and OPTEE configurations for your own board.
2. Add your board to the firmware build system.
3. Resolve compilation errors to build your firmwares into a `firmware_fit.merged`.
4. Iterate on builds of SPL, U-BOOT, OP-TEE, UEFI until Windows boots with minimum support.
5. Bring up devices one at a time.

Note: Before starting, read the [boot flow](#) document to get an idea of the boot process.

In the following sections, replace `yourboard` with a concise name of your board.

### 14.1 U-Boot

The first step is to bring up U-Boot SPL. We use U-Boot in a specific way to implement certain security features, so even if a U-Boot configuration already exists for your board, you will need to create a new configuration for booting Windows. The operation of SPL is described [here](#). Your board must follow the same general flow, with only board-specific changes.

1. Copy `configs/mx6sabresd_nt_defconfig` to `configs/yourboard_nt_defconfig` (For iMX7 start with `mx7sabresd_nt_defconfig`)
2. Edit `configs/yourboard_nt_defconfig`
3. Change `CONFIG_TARGET_MX6SABRESD=y` to `CONFIG_TARGET_YOURBOARD=y`. If your board is not already supported by U-Boot, you will need to add it to U-Boot. We walk through this process below.
4. Change `MX6QDL` in the line `CONFIG_SYS_EXTRA_OPTIONS="IMX_CONFIG=arch/arm/mach-imx/spl_sd.cfg,MX6QDL"` to the appropriate value for your board. Possible values are listed in `arch/arm/mach-imx/mx6/Kconfig` and `arch/arm/mach-imx/mx7/Kconfig`.

If your board is already supported by U-Boot, you'll still need to make sure that the correct configuration options are set.

### 14.1.1 U-Boot Configuration Options

Here are some important configuration options for booting Windows.

- `CONFIG_BAUDRATE=115200` - Sets the UART baud rate to 115200.
- `CONFIG_BOOTCOMMAND="globalpage init 0x10817000; globalpage add ethaddr; fatload mmc 0:2 0x80A20000 /uefi.fit; bootm 0x80A20000"` - This is the command that automatically runs on startup. It will store the MAC address into the global page for UEFI, then load `uefi.fit` from mmc and boot the fit to start UEFI. The globalpage commands should be omitted if `CONFIG_CMD_GLOBAL_PAGE` has not been selected.
- `CONFIG_DISTRO_DEFAULTS=y` - Enables default boot scripting which is used by `CONFIG_BOOTCOMMAND`.
- `CONFIG_BOOTDELAY=-2` - Disables the delay before U-Boot runs the bootcommand. The value -2 means that it will not check the serial port for interrupts unlike a delay of 0. This is important because WinDBG continuously sends characters through the UART which will U-Boot to stop in the console if it checks.
- `CONFIG_FIT=y` - Allow booting of Flattened Image Trees (FIT) which store both binaries and their metadata.
- `CONFIG_OF_LIBFDT=y` - Adds to U-Boot the library responsible for working with Flattened Device Trees (FDT), of which FITs are a subset.
- `CONFIG_IMX_PERSIST_INIT=y` - Prevents U-Boot proper from disabling the display and PCIe when booting into UEFI.
- `CONFIG_CMD_FAT=y` - Enables FAT filesystem commands and is used by the boot script to load UEFI.
- `CONFIG_CMD_MMC=y` - Enables MMC commands and is used by the boot script to load UEFI.
- `CONFIG_CMD_PART=y` - Enables part command which is used by UEFI boot script.

- `CONFIG_HUSH_PARSER=y` - Necessary to enable script parsing.
- `CONFIG_SECURE_BOOT=y` - Enables the i.MX6 crypto driver.
- `CONFIG_DEFAULT_DEVICE_TREE="devicetreename"` - Selects a device tree for the platform (eg `imx6qdl-sdb`). Important for the FIT build path that it exists, but the tree can be empty.
- `CONFIG_SPL=y` - Enables the Secondary Program Loader framework which is required to load and run OP-TEE as soon as possible.
- `CONFIG_SPL_FIT=y` - Allows SPL to read FITs (U-Boot and OPTEE binaries).
- `CONFIG_SPL_FIT_SIGNATURE_STRICT=y` - Halt if loadables or firmware don't pass FIT signature verification.
- `CONFIG_SPL_LOAD_FIT=y` - SPL will attempt to load a FIT to memory.
- `CONFIG_SPL_OF_LIBFDT=y` - Adds to SPL the library responsible for working with Flattened Device Trees (FDT), of which FITs are a subset.
- `CONFIG_SPL_LEGACY_IMAGE_SUPPORT=n` - Prevents SPL from loading legacy images (which cannot support future security features).
- `CONFIG_SPL_BOARD_INIT=y` - Enables board specific implementation of `void spl_board_init(void)`.
- `CONFIG_SPL_CRYPT_SUPPORT=y` - Enables the crypto driver in SPL.
- `CONFIG_SPL_FSL_CAAM=y` - Enables the CAAM driver in SPL.
- `CONFIG_SPL_HASH_SUPPORT=y` - Enable hashing drivers in SPL.
- `CONFIG_SPL_ENABLE_CACHES=y` - Enables caches in SPL, required for the RIoT Tiny SHA256 implementation.
- `CONFIG_SPL_ECC=y` - Enable support for Elliptic-curve cryptography in SPL using code from the RIoT submodule.
- `CONFIG_USE_TINY_SHA256=y` - Select the smaller SHA256 implementation from the RIoT submodule instead of U-Boot's default implementation.
- `CONFIG_SPL_OF_CONTROL=y` - Enable run-time configuration via Device Tree in SPL.
- `CONFIG_SPL_OPTEE_BOOT=y` - Instructs SPL to load and jump to OP-TEE. Required to boot Windows.
- `CONFIG_SPL_MMC_SUPPORT=y` - Enables MMC support in SPL. Required to load OP-TEE and U-Boot proper.
- `CONFIG_SPL_USE_ARCH_MEMCPY=n` - Disables use of optimized memcpy routine. Saves space in SPL.
- `CONFIG_SYS_L2CACHE_OFF=y` - Saves space in SPL by not including L2 initialization and maintenance routines. L2 is not necessary for performance. L2 is enabled by Windows later on.
- `CONFIG_USE_TINY_PRINTF=y` - Saves space in SPL by selecting minimal printf implementation.

Here are some more configuration options that aren't boot critical for Windows.

- `CONFIG_CMD_GLOBAL_PAGE=y` - Enables the `globalpage` command seen in `CONFIG_BOOTCOMMAND`. This is used to pass the MAC addresses to UEFI.



- `CONFIG_FIT_VERBOSE=y` - Enables high verbosity when loading and parsing a Flattened Image Tree. Helpful for debugging boot.

### 14.1.2 Adding a new board to U-Boot

Numerous resources to guide you through porting U-Boot to new boards are available. We recommend that you familiarize yourself with them, as this section may not be exhaustive.

1. Edit `arch/arm/mach-imx/<mx6 or mx7>/Kconfig` and add a config option for your board:

```
1 config TARGET_YOURBOARD
2     bool "Your i.MX board"
3     select MX6QDL # (This should change to match CONFIG_SYS_EXTRA_OPTIONS from your defconfig)
4     select BOARD_LATE_INIT
5     select SUPPORT_SPL
```

2. Create and initialize a board directory:

```
1 mkdir -p board/yourcompany/yourboard
2 cp board/freescale/mx6sabresd/* board/yourcompany/yourboard/
3 mv board/yourcompany/yourboard/mx6sabresd.c board/yourcompany/yourboard/yourboard.c
```

3. Edit `board/yourcompany/yourboard/Makefile` and replace `mx6cuboxi.c` with `yourboard.c`
4. Edit `board/yourcompany/yourboard/Kconfig` and set appropriate values for your board. Note that the build system expects `SYS_CONFIG_NAME` to correspond to the name of a header file in `include/configs`:

```
1 if TARGET_YOURBOARD
2
3 config SYS_BOARD
4     default "yourboard"
5
6 config SYS_VENDOR
7     default "yourcompany"
8
9 config SYS_CONFIG_NAME
10     default "yourboard"
11
12 endif
```

5. Create a config header for your board:

```
1 cp include/configs/mx6cuboxi.h include/configs/yourboard.h
```

6. Edit `include/configs/yourboard.h` as necessary for your board. You may need to add, remove, or change options depending on what's available on your board. Some notable settings are:
- `CONFIG_MXC_UART_BASE` - set this to the base address of the UART instance that should be used for debug and console output.

- `CONFIG_SYS_FSL_ESDHC_ADDR` - set this to the base address of the SDHC instance that U-Boot resides on.
- `CONFIG_EXTRA_ENV_SETTINGS` - This should be set as follows to enable booting UEFI. If `CONFIG_UEFI_BOOT` is defined, you should include `config_uefi_bootcmd.h` and set `CONFIG_EXTRA_ENV_SETTINGS` to `BOOTENV`. You must replace the 0 in `mmcdev=0` with the mmc number your device boots from.

```
1  #ifdef CONFIG_UEFI_BOOT
2  #include <config_uefi_bootcmd.h>
3
4  #define CONFIG_EXTRA_ENV_SETTINGS \
5      "mmcdev=0\0" \
6      BOOTENV
7  #else
```

7. Edit `board/yourcompany/yourboard/yourboard.c` and add, change, and remove code as appropriate for your board. Some configurations that will need to change are pin muxing, MMC initialization, DDR size, and DRAM timing parameters.
8. Build your board. Be prepared to spend some time fixing compilation errors as you get your board into buildable shape.

```
1  make yourboard_nt_defconfig
2  make
```

Note: SPL must be less than 44k to fit into the allocated space.

## 14.2 OP-TEE

OP-TEE is an operating system that runs in ARM TrustZone and provides a Trusted Execution Environment (TEE). OP-TEE is required to boot Windows. OP-TEE does the following:

- Provides a trusted execution environment for trusted applications
- Switches to normal world
- Configures and enables L2 cache when requested by Windows
- Enables secondary cores when requested by Windows
- Implements shutdown and reboot
- Implements power management functionality through PSCI

OP-TEE is mostly board-independent. Right now, the only configuration that needs to be changed is the console UART. In the future, there may be other board-specific configurations that need to change as trusted I/O is implemented.

1. Source code of OP-TEE are stored in 'firmware' ZIP file: `optee_os/optee_os_arm/`
2. Add a platform flavor for your board. Edit `core/arch/arm/plat-imx/conf.mk` and add your board to the appropriate flavorlist, for example:

```
1  mx6q-flavorlist = mx6qsabresd mx6qyourboard
2  ---or---
3  mx6dl-flavorlist = mx6dlsabresd mx6dlyourboard
4  ---or---
5  mx7-flavorlist = mx7dsabresd mx7yourboard
```

3. Edit `core/arch/arm/plat-imx/platform_config.h` and set `CONSOLE_UART_BASE` to the appropriate value for your platform.
4. Follow the next section to set up a firmare build folder for your system. This will select the correct flags and make OP-TEE for you. If you need more debug output, you can customize `OPTEE_FLAGS` in the Common Makefile `build/firmware/Common.mk` set `CFG_TEE_CORE_DEBUG=y` and `CFG_TEE_CORE_LOG_LEVEL=4`.

### 14.3 Setting up your build enviroment to build `firmware_fit.merged`

In order to build and load both OPTEE and U-Boot, create a Flattened Image Tree (FIT) binary to flash onto your device. The build enviroment for FIT images is integrated into the build infrastructure. This will sign SPL for high assurance boot, and combine SPL, U-Boot, and OP-TEE into a single `firmware_fit.merged` file that can be tested manually, or built into an FFU image as part of a BSP.

1. Copy `imx-iotcore/build/firmware/existingboard` to `imx-iotcore/build/firmware/yourboard`.
2. Edit `imx-iotcore/build/firmware/yourboard/Makefile` and change the `UBOOT_CONFIG` and the OP-TEE `PLATFORM` for your board.
3. Run `make` in `imx-iotcore/build/firmware/yourboard` and verify that `firmware_fit.merged` is generated.

Flash `firmware_fit.merged` to your SD card. If you are using Linux, run:

```
1  dd if=firmware_fit.merged of=/dev/sdX bs=512 seek=2
```

If you are using Windows, use [dd for Windows](#):

```
1  dd if=firmware_fit.merged of=\\.\PhysicalDriveX bs=512 seek=2
```

### 14.3.1 Testing SPL

When you have `firmware_fit.merged` building, you should run SPL. If successful, SPL will initialize DRAM, initialize MMC, load OP-TEE and U-Boot proper from MMC, then jump to OP-TEE.

Open a serial terminal to your board at 8N1 115200. Insert the SD card into your board and boot. You should see output similar to the following:

```
1      U-Boot SPL 2018.05-rc1-00004-g5a771d5 (May 25 2018 - 13:16:09 -0700)
2      Booting from SD card
3      Trying to boot from MMC1
```

If SPL was able to load and start OPTEE, the next few lines will be

```
1      I/TC:
2      I/TC:  OP-TEE version: v0.4.0 #1 Fri May 25 20:22:16 UTC 2018 arm
3      I/TC:  Initialized
```

### 14.3.2 Testing OP-TEE

When you have built OP-TEE successfully, run it and see that it gets as far as normal world. This will also test SPL. SPL will not jump to OP-TEE unless it also successfully loads U-Boot proper so your FIT image will need to contain both OP-TEE and U-Boot proper.

```
1      dd if=firmware_fit.merged of=/dev/sdX bs=512 seek=2
2      ---or---
3      dd if=firmware_fit.merged of=\\.\PhysicalDriveX bs=512 seek=2
```

Boot your device. You should see output similar to the following:

```
1  U-Boot SPL 2018.05-rc1-00004-g5a771d5 (May 25 2018 - 13:16:09 -0700)
2  Booting from SD card
3  Trying to boot from MMC1
4  DEBUG:  [0x0] TEE-CORE:add_phys_mem:524: CFG_SHMEM_START type NSEC_SHM 0x12800000 size 0x00200000
5  DEBUG:  [0x0] TEE-CORE:add_phys_mem:524: CFG_TA_RAM_START type TA_RAM 0x10c00000 size 0x01c00000
6  .
7  .
8  .
9  INFO:    TEE-CORE: OP-TEE version: 2.3.0-480-gf68edcc #4 Thu Feb  1 00:41:33 UTC 2018 arm
10 DEBUG:  [0x0] TEE-CORE:mobj_mapped_shm_init:592: Shared memory address range: 10b00000, 11500000
11 DEBUG:  [0x0] TEE-CORE:protect_tz_memory:201: pa 0x10a00000 size 0x01e00000 needs TZC protection
12 FLOW:   [0x0] TEE-CORE:protect_tz_memory:221: Unaligned pa 0x10a00000 size 0x01000000
13 FLOW:   [0x0] TEE-CORE:protect_tz_memory:221: Unaligned pa 0x10a00000 size 0x00800000
14 FLOW:   [0x0] TEE-CORE:protect_tz_memory:221: Unaligned pa 0x10a00000 size:protect_tz_memory:240:
      Protecting pa 0x12000000 size 0x00800000
15 INFO:    TEE-CORE: Initialized
16 DEBUG:  [0x0] TEE-CORE:init_primary_helper:680: Primary CPU switching to normal world boot
```

You may also see output from U-Boot.

### 14.3.3 Testing U-Boot

U-Boot should already be building from earlier and included in your `firmware_fit.merged` file. U-Boot will run after OP-TEE and attempt to load UEFI. Since UEFI is not present yet, it should fail the script and go to the U-Boot prompt. U-Boot initializes devices then executes the commands in `CONFIG_BOOTCOMMAND`. If it does not attempt to load UEFI, then `CONFIG_BOOTCOMMAND` is probably not set correctly. To see the actual value of `CONFIG_BOOTCOMMAND`, you can inspect `u-boot.cfg` or run `printenv` at the U-Boot prompt and look at the `bootcmd` variable.

During initial bringup it may be helpful to disable all devices in U-Boot except UART and eMMC.

You can create a debug build of U-Boot with the following command:

```
1 make KCFLAGS=-DDEBUG
```

This is very helpful for debugging, but will cause the size of the binaries to increase. SPL may grow too big, so you may have to use a release build of SPL and a debug build of `u-boot-ivt.img`. It is OK to mix a release build of SPL with a debug build of full U-Boot.

A successful run of U-Boot should have a similar output to the following:

```
1 U-Boot 2018.01-00125-gfb1579e (Jan 31 2018 - 16:54:39 -0800)
2
3 CPU:   Freescale i.MX6Q rev1.5 996 MHz (running at 792 MHz)
4 CPU:   Extended Commercial temperature grade (-20C to 105C) at 41C
5 Reset cause: WDOG
6 Board: MX6 Board (som rev 1.5)
7 DRAM:  2 GiB
8 MMC:   FSL_SDHC: 0, FSL_SDHC: 1
9 Using default environment
10
11 auto-detected panel HDMI
12 Display: HDMI (1024x768)
13 In:     serial
14 Out:    serial
15 Err:    serial
16 Net:    FEC
17 starting USB...
18 USB0:   Port not available.
19 USB1:   USB EHCI 1.00
20 scanning bus 1 for devices... 2 USB Device(s) found
21     scanning usb for storage devices... 0 Storage Device(s) found
22 switch to partitions #0, OK
23 mmc0 is current device
24 ** Unable to read file imx6board_efi.fd **
25 ** Unrecognized filesystem type **
26 Error: failed to load UEFI
27 =>
```

## 14.4 UEFI

UEFI is required to boot Windows. UEFI provides a runtime environment for the Windows bootloader, access to storage, hardware initialization, ACPI tables, and a description of the memory map. First, construct a minimal UEFI with only eMMC and debugger support. Then, add devices one-by-one to the system.

1. Clone our reference implementation of EDK2. It is split between `edk2` and `edk2-platforms`. See the Readme here: <https://github.com/ms-iot/imx-edk2-platforms>
2. Copy `Platform\NXP\EXISTING_BOARD` to `Platform\<Your Company Name>\YOURBOARD_IMX6_XGB`.
3. Rename the `.dsc` and `.fdf` files to match the folder name.

### 14.4.1 DSC and FDF file

Edit the `.dsc` file and change the following settings as appropriate for your board:

- `DRAM_SIZE` - set to `DRAM_512MB`, `DRAM_1GB`, or `DRAM_2GB`
- `IMX_FAMILY` - set to `IMX6DQ`, `IMX6SX`, or `IMX6SDL`
- `IMX_CHIP_TYPE` - set to `QUAD`, `DUAL`, or `SOLO`
- `gIMXPlatformTokenSpaceGuid.PcdSdhc[1, 2, 3, 4]Enable` - enable the right SDHC instance for your platform. For example,  

```
1 gIMXPlatformTokenSpaceGuid.PcdSdhc2Enable|TRUE
```
- `gIMXPlatformTokenSpaceGuid.PcdSerialRegisterBase` - set to the base address of the UART instance that you want UEFI output to go to.
- `gIMXPlatformTokenSpaceGuid.PcdKdUartInstance` - set this to 1, 2, 3, 4, or 5 (6 and 7 are also available on i.MX7). This is the UART instance that Windows will use for kernel debugging. You will also need to reference `gIMXPlatformTokenSpaceGuid.PcdKdUartInstance` in your board's `AcpiTables.inf` file. U-Boot must initialize the UART, including baud rate and pin muxing. Windows will not reinitialize the UART.

### 14.4.2 Board-specific Initialization

The file `Platform\<Your Company Name>\YOURBOARD_IMX6_XGB\Library\iMX6BoardLib\iMX6BoardInit.c` contains board-specific initialization code, which includes:

- Pin Muxing

- Clock initialization
- PHY initialization

Much of the same functionality exists in U-Boot. The content in this file should be minimized and board-specific initialization should be done in U-Boot. The goal is to eventually eliminate this file.

Start with an empty `ArmPlatformInitialize()` function, and add code as necessary when you bring up each device. **Prefer to add code to U-Boot instead.** This will keep pin muxing, clock initialization, and PHY initialization all in one place.

### 14.4.3 SMBIOS

Edit `Platform\<Your Company Name>\YOURBOARD_IMX6_XGB\Drivers\PlatformSmbiosDxe\PlatformSmbiosDxe.c` and set values appropriate for your board. Settings to change are:

```
1     mBIOSInfoType0Strings
2     mSysInfoType1Strings
3     mBoardInfoType2Strings
4     mEnclosureInfoType3Strings
5     mProcessorInfoType4Strings
6     mMemDevInfoType17.Size
```

### 14.4.4 ACPI Tables

For initial bringup, start with a minimal DSDT that contains only the devices required to boot. Then add devices one-by-one, and test each device as you bring it up.

Edit `Platform\<Your Company Name>\YOURBOARD_IMX6_XGB\AcpiTables\DSDT.asl` and remove all but the following entries:

```
1     include("Dsd-Common.inc")
2     include("Dsd-Platform.inc")
3     include("Dsd-Gpio.inc")
4     include("Dsd-Sdhc.inc")
```

#### 14.4.4.1 SDHC

Edit `Dsd-Sdhc.inc` and ensure that the SDHC instance on which your boot media resides is present and enabled. To simplify bringup you should disable the other SDHC instances. A minimal SDHC device node looks like:

```
1     ///
2
3     uSDHC2: SDIO Slot//
```

```
4
5 Device (SDH2)
6 {
7     Name (_HID, "FSCL0008")
8     Name (_UID, 0x2)
9
10    Method (_STA) // Status
11    {
12        Return(0xf) // Enabled
13    }
14
15    Name (_S1D, 0x1)
16    Name (_S2D, 0x1)
17    Name (_S3D, 0x1)
18    Name (_S4D, 0x1)
19
20    Method (_CRS, 0x0, NotSerialized) {
21        Name (RBUF, ResourceTemplate () {
22            MEMORY32FIXED(ReadWrite, 0x02194000, 0x4000, )
23            Interrupt(ResourceConsumer, Level, ActiveHigh, Exclusive) { 55 }
24        })
25        Return(RBUF)
26    }
27
28    Name (_DSD, Package())
29    {
30        ToUUID ("DAFFD814-6EBA-4D8C-8A91-BC9BBF4AA301"),
31        Package ()
32        {
33            Package (2) { "BaseClockFrequencyHz", 198000000 }, // SDHC Base/Input Clock: 198MHz
34            Package (2) { "Regulator1V8Exist", 0 }, // 1.8V Switching External
35            Circuitry: Not-Implemented
36            Package (2) { "SlotCount", 1 }, // Number of SD/MMC slots
37            connected on the bus: 1
38            Package (2) { "RegisterBasePA", 0x02194000 } // Register base physical address
39        }
40    }
41    Child node to represent the only SD/MMC slot on this SD/MMC bus//
42    In theory an SDHC can be connected to multiple SD/MMC slots at//
43    the same time, but only 1 device will be selected and active at//
44    a time//
45
46    Device (SD0)
47    {
48        Method (_ADR) // Address
49        {
50            Return (0) // SD Slot 0
51        }
52
53        Method (_RMV) // Remove
54        {
55            Return (0) // Removable
56        }
57    }
```



58        }

[\\_RMV](#) is an ACPI method that returns whether the slot is removable or not where 1 indicates removable while 0 means non-removable. eMMC slots should be marked as non-removable, while SD slots should be also marked as non-removable if it can be used as a boot media not as a secondary storage.

[\\_DSM](#) is an ACPI method that is used by the SD bus to perform very specialized and platform dependent tasks. It is currently used by Windows to perform SD bus power control On/Off which is required during 3V3/1V8 SD voltage switching sequence. For bring-up, the [\\_DSM](#) is not required and in that case the [Regulator1V8Exist](#) field should be set to 0 to indicate that SD voltage switching is not implemented/supported.

```
1 Package (2) { "Regulator1V8Exist", 0 }, // 1.8V Switching External Circuitry: Not implemented
```

#### 14.4.4.2 PWM

For the best experience using the PWM WinRT APIs from UWP apps some additional device properties need to be set. Documentation on these device interface properties can be found here in the [Setting device interface properties](#) section of the PWM DDI MSDN article.

For an example of setting the PWM device interface properties statically from an inf file, see the [Virtual PWM driver sample](#).

For an example on how to read the ACPI [\\_DSD](#) from within a kernel driver, see the i.MX SDHC driver driver/sd/imxusdhc.

#### PWM References:

- [PWM DDI](#)
- [PWM Driver Reference](#)
- [PWM WinRT APIs](#)
- [Virtual PWM Driver Sample](#)

#### 14.4.5 Security TAs

UEFI includes a pair of OP-TEE Trusted Applications (TAs) which implement a firmware TPM, and a UEFI authenticated variable store. These binaries should generally not require re-compiling; however, if your OP-TEE has been changed (including build flags) it may introduce incompatibilities. See [Updating the TAs](#) for instructions on adding new TAs to your firmware binaries.

They are included in UEFI by default but can be omitted with the `CONFIG_NOT_SECURE_UEFI=1` flag. The TAs require OP-TEE to have access to secure storage (eMMC's RPMB). Windows will not support Bitlocker, Secure Boot, or persistent firmware variable storage without these TAs enabled.

### 14.4.6 Building UEFI

For a detailed guide on how to build the i.MX UEFI firmware image, please refer to [Building and Updating ARM32 Firmware](#) or [Building and Updating ARM64 Firmware](#).

### 14.4.7 Testing UEFI

To test UEFI, you will need an SD card with a FAT partition. The easiest way to get an SD card with the right partition layout is to flash the FFU of the existing board, then replace the firmware components.

1. Build FFU
2. Flash the FFU to your SD card

```
1  dism /apply-image /imagefile:BoardTestOEMInput.xml.Release.ffu /applydrive:\\.PhysicalDriveX /
    skipPlatformCheck
```

3. Use the `dd` command to flash `firmware_fit.merged` to the SD card.
4. Replace `uefi.fit` on the EFIESP partition of the SD card with your `uefi.fit`.

Power on the system. You should see UEFI run after U-Boot, and UEFI should attempt to load Windows.

## 14.5 Booting Windows

As long as the serial console and SDHC device node are configured correctly in UEFI, the Windows kernel should get loaded. Once you see the kernel looking for a debugger connection, you can close the serial terminal and start WinDBG.

```
1  windbg.exe -k com:port=COM3,baud=115200
```

If you hit an `INACCESSIBLE_BOOT_DEVICE` bugcheck, it means there's a problem with the storage driver. Run `!devnode 0 1` to inspect the device tree, and see what the status of the SD driver is. You can dump the log from the SD driver by running:

```
1  !rcdrkd.rcdrlogdump imxusdhc.sys
```

After you have a minimal booting Windows image, the next step is to bring up and test each device.

## 15 Updating the BSP port

Below is a list of changes that may have occurred since any initial enablement of Windows 10 IoT Core on your i.MX platform.

### 15.1 Reworked firmware build system

The firmware build system now builds entirely in WSL and Linux and uses a makefile as the only front-end. For more information on firmware build system-setup and usage, see the [Building and Updating ARM32 Firmware](#) or [Building and Updating ARM64 Firmware](#) guide.

In order to use the Makefile, you will need to have a folder for your board in the firmware folder. To set up this firmware folder, run `NewiMX6Board.ps1`, which is [documented here](#). Please note that this firmware folder name must match the EDK2-Platforms name for your board.

### 15.2 FIT load for OP-TEE and U-Boot Proper inside of SPL

In order to use existing loading infrastructure, we have updated the way U-Boot proper and OP-TEE are packaged so that SPL can load and run them as a Flattened Image Tree. When built through the firmware Makefile the required `firmware_fit.merged` file will be generated if all of the features required for FIT in SPL are enabled in your U-Boot defconfig.

### 15.3 FIT loading UEFI inside of U-Boot Proper

We have updated the way we run the UEFI firmware from U-Boot proper to use U-Boot's built in FIT boot path.

- The defconfig settings `CONFIG_UEFI_BOOT`, `CONFIG_UEFI_LOAD_ADDR`, and `CONFIG_UEFI_IMAGE_NAME` are no longer required and should be removed.
- The UEFI binary is no longer called `IMX6BOARD_EFI.fd` or `imxboard_efi.fd` on the EFI partition. Instead the UEFI is built into a Flattened Image Tree called `uefi.fit` and is stored on the EFI partition. The `uefi.fit` packaging is done by the `mkimage` tool and is done automatically when EDK2 is built through the firmware Makefile.
- The new boot path no longer uses a hardcoded UEFI BOOTCOMMAND, instead `CONFIG_BOOTCOMMAND` is customized directly in the defconfig. `CONFIG_USE_BOOTCOMMAND` and `CONFIG_CMD_BOOTM` must

be enabled to make sure that the bootcommand is enabled, and that the `bootm` command is available.

- `CONFIG_IMX_PERSIST_INIT` has been added so that U-Boot proper does not disable the IPU before booting into UEFI, keeping the display on. UEFI assumes that the IPU is enabled and configured when the GOP driver loads.

```
1 CONFIG_USE_BOOTCOMMAND=y
2 CONFIG_BOOTCOMMAND="fatload mmc 0:2 0x80A20000 /uefi.fit; bootm 0x80A20000"
3 CONFIG_CMD_BOOTM=y
4 CONFIG_IMX_PERSIST_INIT=y
```

- Some board header files statically define a `CONFIG_BOOTCOMMAND`, which will conflict with the one set in the defconfig. If your board header has a `#define CONFIG_BOOTCOMMAND`, wrap it in an `#if !defined` block like below:

```
1 #if !defined(CONFIG_BOOTCOMMAND)
2 #define CONFIG_BOOTCOMMAND \
3     "run findfdt;" \
4     ...
5     "else run netboot; fi"
6 #endif /* !defined(CONFIG_BOOTCOMMAND) */
```

- If you previously disabled `CONFIG_DISTRO_DEFAULTS`, you may need to reenale it to pull in dependencies for bootm's FIT boot: `CONFIG_DISTRO_DEFAULTS=y`

## 15.4 Miscellaneous U-Boot defconfig settings

- `CONFIG_BOOTDELAY=-2` boots the `CONFIG_BOOTCOMMAND` without delay or checking serial input to interrupt. This is important because WinDBG will interrupt boot if U-Boot checks serial input.
- The list of additional U-Boot options used when booting Windows is available here: [U-Boot Configuration Options](#)

## 16 Revision History

Table 16.1: Revision history

Revision number	Date	Substantive changes
W1.0.0_ear	10/2019	Initial engineering release for i.MX6, i.MX7 and i.MX8M platforms