Bootloader for PSoC™ 6 MCUs

About this document

Scope and purpose

This application note introduces you to the bootloader architecture and describes the different types of bootloaders supported on PSoC™ 6 MCUs, while covering the MCUboot based bootloader in depth. This application note also provides information on the use cases and features of the MCUboot based bootloader along with instructions on programming, updating, and debugging the bootloader.

Infineon MCUs support multiple variants of bootloader/bootloader libraries. The purpose of this application note is to provide comprehensive information about using the MCUboot based bootloader on PSoC™ 6 devices. This document will also describe the various features supported by the bootloader.

Intended audience

This document is intended for anyone who wants to understand the MCUboot based bootloader architecture, features, and implementation on PSoC™ 6 MCUs.

Table of contents

1 Introduction

A bootloader is a software component responsible for initializing the memory hardware required to read/write flash, set protection units, and run the main application during the system startup. It plays a crucial role in the bootup sequence of an MCU-based system. Bootloaders are designed to be lightweight and efficient, often residing in a small portion of nonvolatile memory.

Key functions of a bootloader include:

- **Hardware initialization:** Configuring essential MCU peripherals like clocks, memory, and I/O interfaces.
- **Boot source selection:** Determining from where to load the main application, such as internal flash memory, external memory, or a communication interface like UART, SPI, or USB.
- **Firmware update support:** Allowing the MCU to update its firmware in the field, typically through a secure and reliable mechanism.
- **Security features:** Implementing measures like secure boot to verify the integrity and authenticity of the loaded firmware, protecting against unauthorized modifications.

Finally, a bootloader is a critical component that ensures a smooth and secure startup of MCU-based systems, facilitating firmware updates and maintaining system integrity. There are two types of bootloaders that Infineon supports namely Device Firmware Update (DFU) bootloader and MCUboot based bootloader.

DFU bootloader

The Device Firmware Update (DFU) is a process to transfer the firmware in the device. The target device (e.g., MCU) and the host machine (e.g., Computer) establishes a connection to download the firmware. This connection can either be wired (UART, I2C, SPI, USB) or wireless (Wi-Fi, Bluetooth® Low Energy). The target device receives the firmware from the host, validates, and executes it. For PSoC™ 6 MCUs, Infineon offers DFU middleware (MW) which provides an over-the-wire update solution.

DFU MW can be used to build a small footprint bootloader for Infineon MCUs. It is ideal when the selected MCU cannot support a large footprint bootloader based on the MCUboot library or can have any other use case-specific constraints. The DFU MW supports developing bootloaders both with and without transport. For example, you can build a DFU-based bootloader with transport capabilities such as UART, USB, etc. Whereas another use case can build the DFU-based bootloader without any transport.

Irrespective of the transport configuration selected, the bootloader can perform the validation and installation of the application image to the boot regions, as required. For more information, see the DFU application note.

MCUboot base bootloader

This bootloader is designed to provide a robust and secure foundation for firmware updates in microcontroller units (MCUs), along with robust security measures during the boot process, ensuring the integrity and authenticity of the firmware loaded onto the device.

This bootloader implements most of the key functionalities as follows:

- Image validation
- Image upgrade
- Reset recovery/power fail-safe mechanism
- Multi-image support

Introduction

Unlike DFU, which only handles firmware update and CRC validation, this bootloader offers many more additional features. This application note solely concentrates on the MCUboot-based bootloader and delves into its features in detail in the upcoming sections.

Boot flow

Generally, Infineon PSoC™ 6 devices are both dual and single core devices.

- PSoC™ 62 and PSoC™ 63 (dual core with Arm Cortex®-M0+ and Arm® Cortex®-M4)
- PSoC™ 61 (single core with Arm® Cortex®-M4)

The bootloader always executes on Arm® Cortex®-M0+ core for PSoC™ 62 and Arm® Cortex®-M4 for PSoC™ 61 device. When the device starts up, it begins with the bootROM, an initial startup code residing in the ROM, performing essential initialization tasks. It then hands over control to the bootloader. This bootloader validates the user application running on either CM0+ or CM4 cores before transitioning control to them.

A typical boot flow of an MCUboot based bootloader is shown in Figure 1.

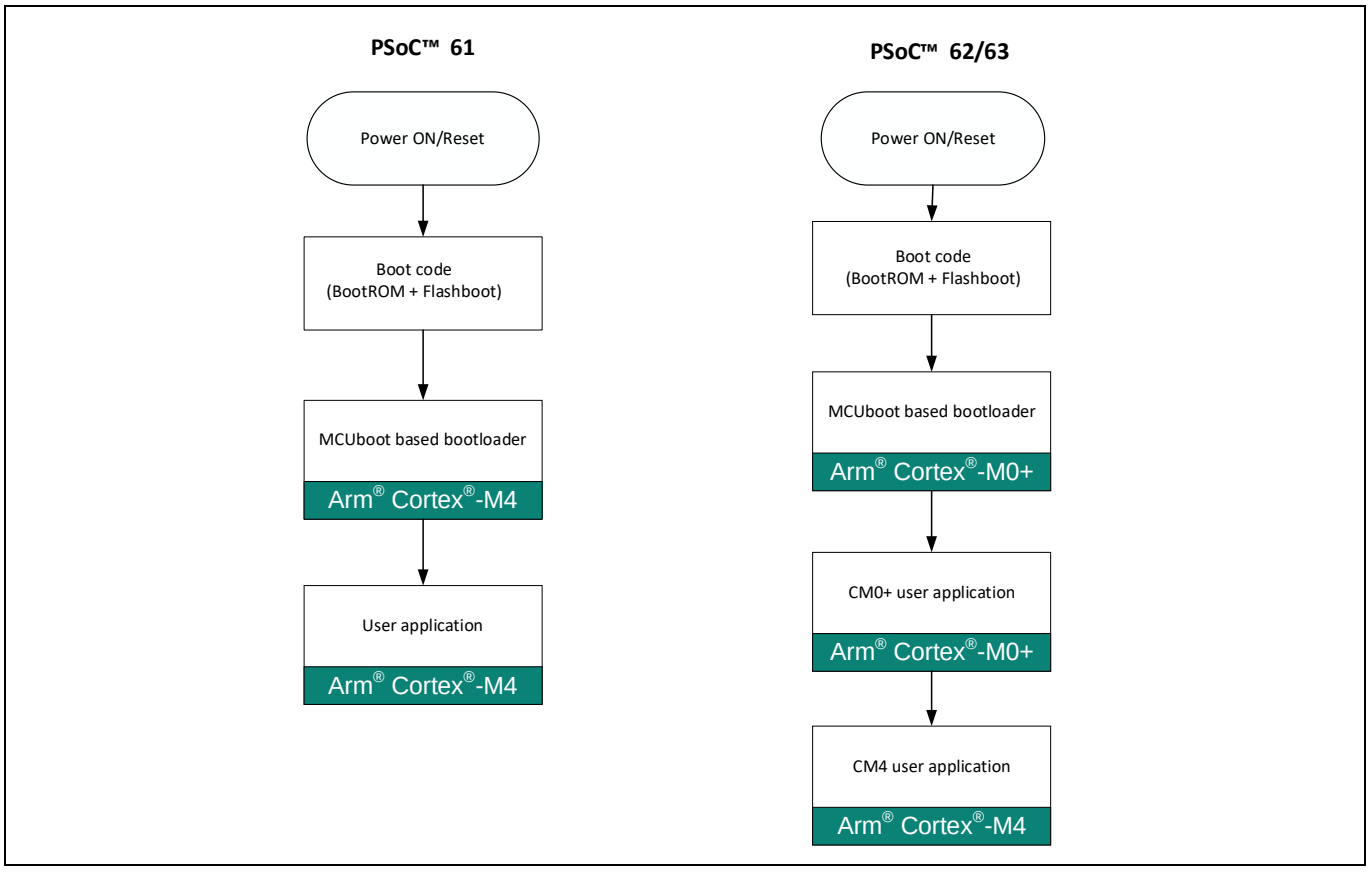


Figure 1 Boot flow with MCUboot based bootloader on PSoC™ 6 MCU

2 Bootloader architecture

The bootloader is based on a comprehensive open-source MCUboot secure bootloader library along with the dependency drivers like Mbed TLS, Crypto HW driver, retarget-io, PDL driver, etc., therefore, making it a robust bootloader solution for the Infineon chipsets.

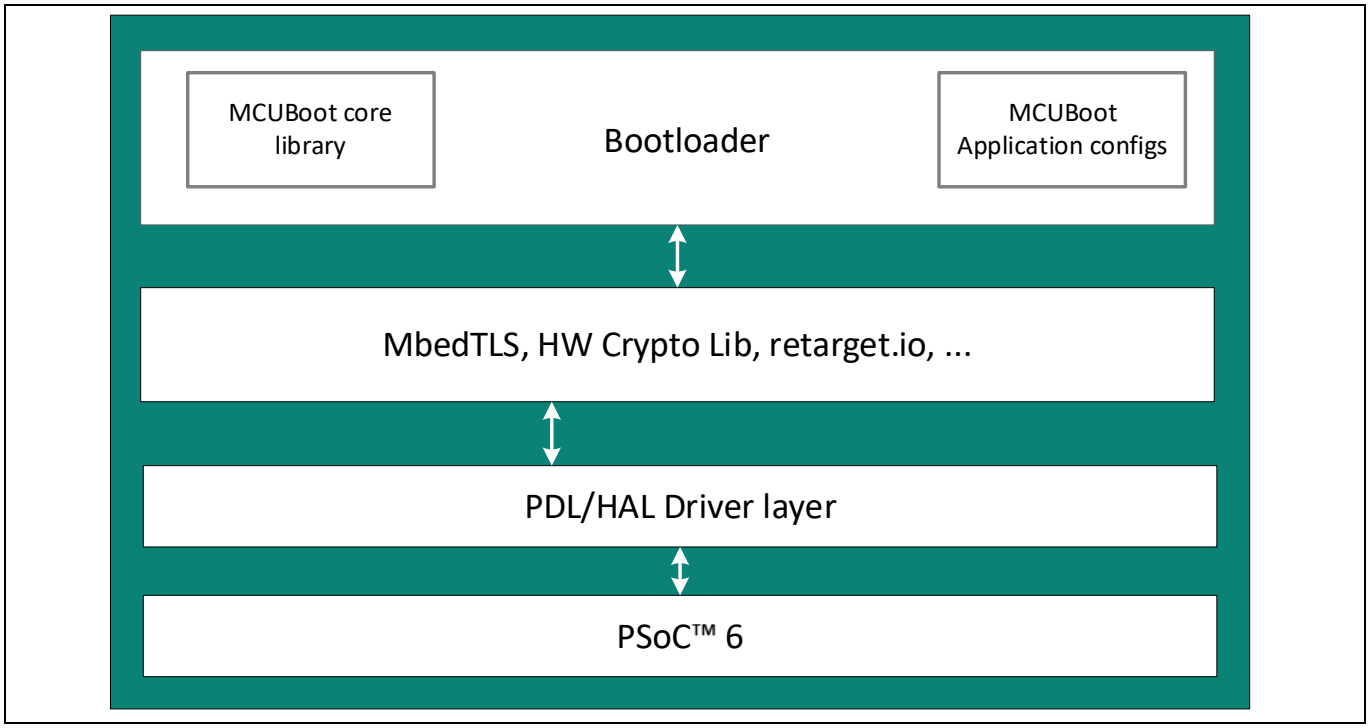


Figure 2 MCUboot based bootloader architecture

This bootloader works by dividing the flash into two slots per image - primary and secondary. The primary is where the code is executed and the secondary is for code update only, not execution. The first version of the application (factory image) is programmed into the primary slot during production. A firmware update application running in the device receives the new version image and places it in the secondary slot. This slot-based partition helps in read/write-protecting the primary slot from a less-privileged application.

Typically, a bootloader application executes in secured mode and is privileged to access the primary slot while a less-privileged application such as a firmware update application cannot access the primary slot, but it can access the secondary slot.

The bootloader always boots the application from the primary slot and overwrite/swap the image from the secondary slot into the primary slot when an upgrade is requested.

3 Bootloader use cases

- Application boot
  - In a secure device, it is critical to implement the firmware validation before it executes to protect against execution of any malicious software. This bootloader has a built-in firmware validation feature which verifies the signature of the application before transferring the control to it.
- Firmware update
  - The bootloader divides the flash memory into two sections per application: a primary and a secondary slot. The initial version of the application is stored in the primary slot. When an update is available, the device's firmware update application receives the UPGRADE image over a wired (device-firmware-upgrade or DFU) or wireless (over-the-air or OTA) communication interface and saves it to the secondary slot. This dual-slot system protects the primary slot from unauthorized changes. The bootloader always starts from the primary slot, but if an update is requested, it copies the new version from the secondary slot to the primary slot.

4 Flash map

As mentioned earlier, the bootloader always boots the application image from the **primary** slot and the **upgrade** image is placed as **secondary slot**. Some platforms support both internal and external flash and some only external flash.

Flash map describes allocated flash memory areas and defines their addresses and sizes. Also, it specifies the type of external flash memory, if applicable. It is defined at compile-time and cannot be changed dynamically. Flash map is prepared in the industry-accepted JSON format. Follow these conditions while modifying the flash map:

Bootloader contains JSON templates for flash maps with commonly used configurations. They can be found in the `<application>/flashmap` folder of the bootloader application. The slots' sizes are defined per platform to be compatible with all supported device families.

When modifying slots sizes, ensure aligning new values with the linker script files for appropriate applications.

Flash map format

Flash map must have the `boot_and_upgrade` section, that define the location of bootloader and at least one image. For instance:

Code Listing 1

```
{
    "boot_and_upgrade":
    {
        "bootloader":
        {
            "address":
            {
                "description": "Address of the bootloader",
                "value": "0x10000000"
            },
            "size":
            {
                "description": "Size of the bootloader",
                "value": "0x18000"
            }
        },
        "application_1":
        {
            "address":
            {
                "description": "Address of the application primary slot",
                "value": "0x10018000"
```

Flash map

Code Listing 1

```
            },
            "size":
            {
                "description": "Size of the application primary slot",
                "value": "0x10000"
            },
            "upgrade_address":
            {
                "description": "Address of the application secondary slot",
                "value": "0x18030200"
            },
            "upgrade_size":
            {
                "description": "Size of the application secondary slot",
                "value": "0x10000"
            }
        }
    }
}
```

Here an application identifier should follow the pattern, i.e., the second image in the multi-image case is `application_2`, the third is `application_3`, and so on. It supports up to four projects at this moment.

For each image, the location and size of its primary slot are given in the `address` and `size` parameters. The location and size of the secondary slot are specified in the `upgrade_address and upgrade_size`. All four values described earlier are mandatory.

There should also be a mandatory bootloader section, describing the location and size of the bootloader in the address and size parameters, respectively.

Code Listing 2

```
/* Independent from multiple image boot */
#define FLASH_AREA_BOOTLOADER          0
#define FLASH_AREA_IMAGE_SCRATCH       3
```

Code Listing 3

```
/* If the bootloader is working with the first image */
#define FLASH_AREA_IMG_1_PRIMARY       1
#define FLASH_AREA_IMG_1_SECONDARY     2
```

Flash map

Code Listing 4

```
/* If the bootloader is working with the second image */
#define FLASH_AREA_IMG_2_PRIMARY      5
#define FLASH_AREA_IMG_2_SECONDARY    6
```

The bootloader area contains the bootloader image itself. The other areas are described in subsequent sections. The flash could contain multiple executable images; therefore, the flash area IDs of primary and secondary areas are mapped based on the number of the active image (on which the bootloader is currently working). For more information, see the MCUboot flash map.

Multiple flash map JSON configurations are defined for PSoC™ 6 devices as mentioned in the flashmap folder of the bootloader code example.

To build the bootloader with the given flash map (e.g., *flash_map.json*), set the `FLASH_MAP` variable in the *<application>/user_config.mk* file to the desired JSON file name.

5 Bootloader features

All features of the bootloader are based on the mtb-example-mcuboot-basic code example as a reference. The files/variables referred in the following sections can be found in this code example.

5.1 Multi-image (upgradable) support

Multi-image operation considers upgrading and verification of more than one image on a device. Single or multi-image mode is stated by the `MCUBOOT_IMAGE_NUMBER` make flag. In multi-image operation, supports up to four images.

Consider a bootloader with two images support and the operation is as follows:

1. Verification of the Secondary_1 and Secondary_2 images.
2. Upgrades secondary to primary if valid images are found.
3. Verification of the Primary_1 and Primary_2 images.
4. Boots the image from the Primary_1 slot only.
5. Boots Primary_1 only if both - Primary_1 and Primary_2 are present and valid. The Primary_1 boots Primary_2 image.

This ensures that two dependent applications can be accepted by the device only if both images are valid.

The following flow chart demonstrates the boot flow in a multi-image boot with N-number of images.

Bootloader features

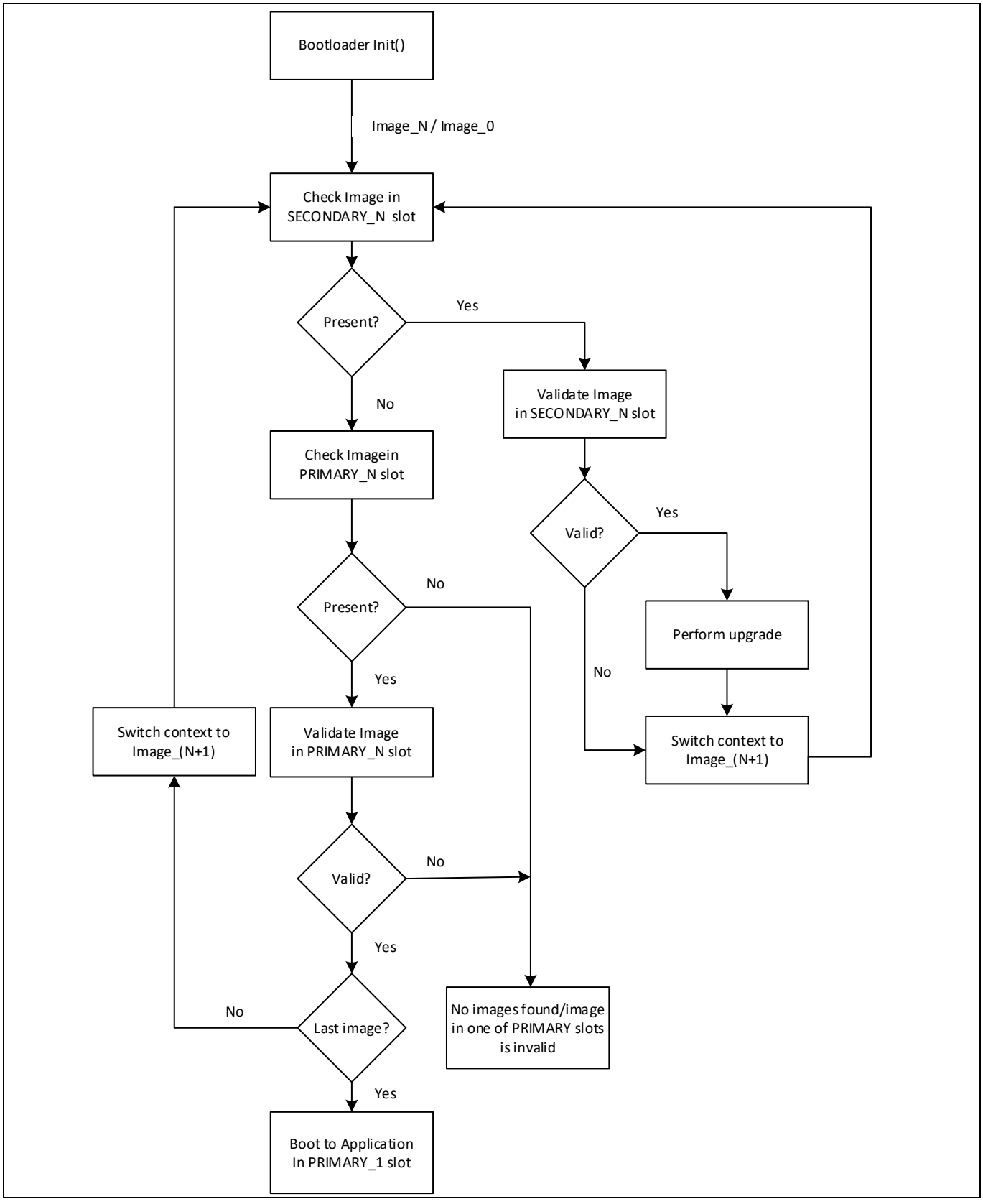


Figure 3 Multi-image boot

Bootloader features

A sample memory map JSON file with multiple images is as follows:

Code Listing 5

```
{
    "boot_and_upgrade":
    {
        "bootloader": {
            "address": {
                "description": "Address of the bootloader",
                "value": "0x10000000"
            },
            "size": {
                "description": "Size of the bootloader",
                "value": "0x18000"
            }
        },
        "application_1": {
            "address": {
                "description": "Address of the application primary_1 slot",
                "value": "0x10018000"
            },
            "size": {
                "description": "Size of the application primary_1 slot",
                "value": "0x10000"
            },
            "upgrade_address": {
                "description": "Address of the application secondary_1
slot",
                "value": "0x10028000"
            },
            "upgrade_size": {
                "description": "Size of the application secondary_1 slot",
                "value": "0x10000"
            }
        }
         "application_2": {

            ...     /* contains primary_2 and secondary_2 slot */
        }
```

Code Listing 5

```
            "application_3": {

            ...     /* contains primary_3 and secondary_3 slot */
        }

            ...
    }
}
```

Based on the number of application entries made in the JSON file, the `MCUBOOT_IMAGE_NUMBER` flag is set to a number of corresponding `application_N` memory map sections. This flag value is set in an auto-generated *memorymap.mk* file at compile time based on the memory map sections in the JSON configuration files.

5.2 Upgrade by overwrite

Bootloader always boots from the primary slot and copies the image from the secondary slot into the primary slot when an upgrade is requested. The upgrade can be either overwrite-based or swap-based mode. In an overwrite-based upgrade, the image in the primary slot is lost and there is no way to rollback if the new image has an issue.

`USE_OVERWRITE` make flag be set to '1' for overwrite and '0' for swap mode.

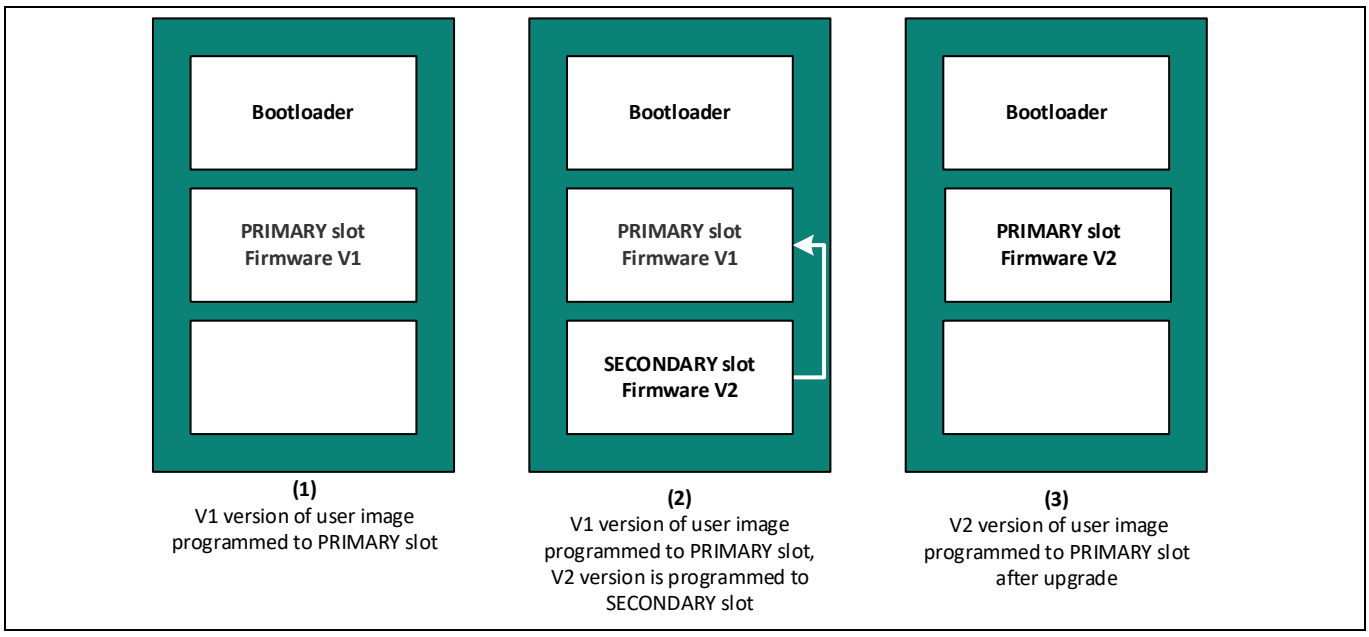


Figure 4 Upgrade by overwrite

On PSoC™ 6 devices, `PLATFORM_DEFAULT_USE_OVERWRITE` flag is set to '1' in the user_config.mk file.

This will set the `USE_OVERWRITE` flag in the auto-generated *memorymap.mk* file at the compile time. See the mtb-example-mcuboot-basic code example for more information.

5.3 Upgrade by swap

In a swap-based upgrade, the images are swapped between the two slots (primary and secondary) and rollback is possible in case of faulty upgraded image.

In SWAP mode, the content of the primary slot is copied into the secondary slot, and at the same time the content of the secondary slot is copied into the primary slot. Doing so, both versions of the application are preserved on the device until the upgraded firmware swapped to the primary slot confirms its operability. Bootloader initiates the revert procedure and rollback to the previous firmware to the primary slot if the upgraded image is faulty and does not boot.

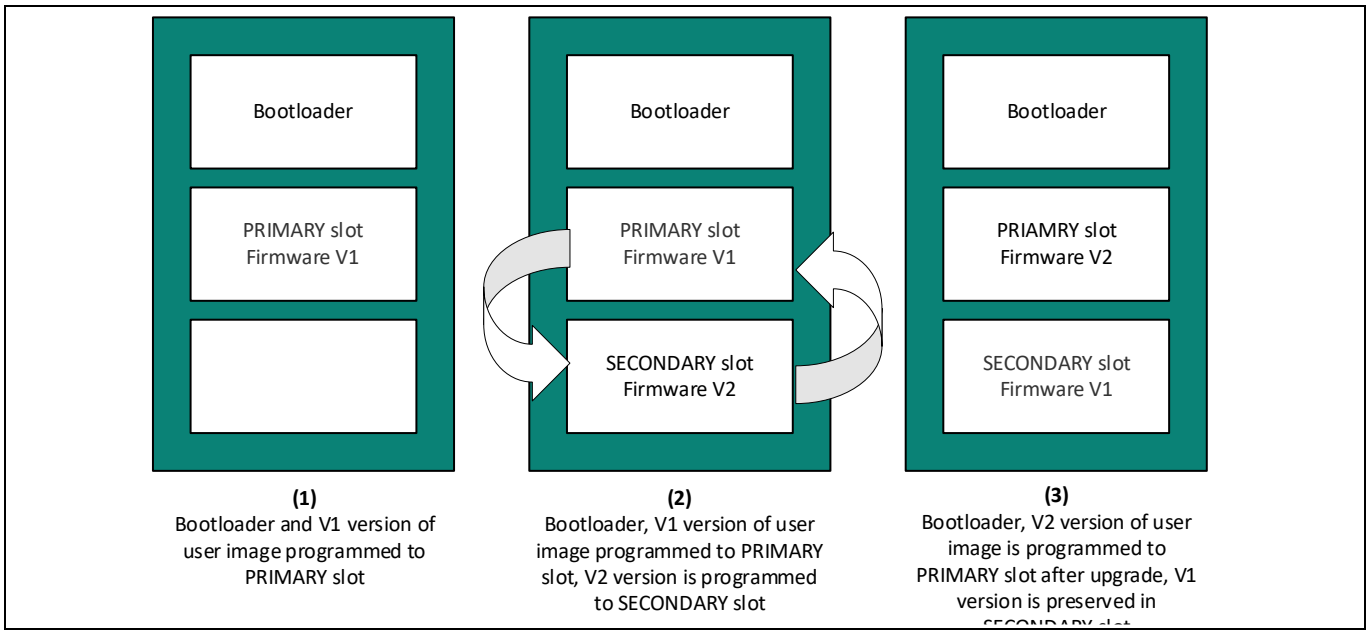


Figure 5 Upgrade by swap

On PSoC™ 6 devices, to enable the swap/upgrade mode, the `PLATFORM_DEFAULT_USE_OVERWRITE` flag is set to '0' in the Makefile.

Based on the earlier configuration, `USE_OVERWRITE` make flag which is auto-generated in *memorymap.mk* file to '1' for the swap mode.

See the user_config.mk file for more information.

There are two mechanisms in the swap-based upgrade as mentioned in the following sections.

5.3.1 Swap using scratch

The swap-using-scratch algorithm requires a dedicated scratch area in flash memory alongside the image slots. This scratch space ensures reliable image swapping during upgrades. Its size needs to accommodate the largest sector size planned for swapping. While many devices have uniform sector sizes (e.g., 4 KB), others have variable sizes with the largest reaching 128 KB or 256 KB. The scratch area must be large enough to hold this maximum sector size.

Note that the scratch area is only used during firmware upgrades, meaning swap operations. However, a larger scratch space offers a significant benefit: it promotes even wear distribution across flash memory. This is because with a single sector, data gets written twice as often compared to using two sectors. To determine the optimal scratch size for your specific needs, consider the following factors:

Bootloader features

- Ratio of image size/scratch size
- Number of erase cycles supported by the flash hardware.

The image size is used (instead of slot size) because only the slot's sectors that are used for storing the image are copied. The image/scratch ratio is the number of times the scratch will be erased on every upgrade. The number of erase cycles divided by the image/scratch ratio will give you the number of times an upgrade can be performed before the device goes out of specification.

```
num_upgrades = number_of_erase_cycles / (image_size / scratch_size)
```

For example, a device with 10000 erase cycles, an image size of 150K and a scratch of 4K (the usual minimum size of 4K sector devices). This would result in a total of:

```
10000 / (150 / 4) ~ 267
```

Increasing the scratch to 16K would provide:

```
10000 / (150 / 16) ~ 1067
```

There is no best ratio, as the right size is use case dependent. Factors to consider include the number of times a device will be upgraded both in the field and during development, as well as any desired safety margin on the manufacturer's specified number of erase cycles. In general, using a ratio that allows hundreds to thousands of field upgrades in production is recommended.

Swap-using scratch algorithm assumes that the primary and the secondary image slot areas sizes are equal. The maximum image size available for the application will be:

```
maximum-image-size = image-slot-size - image-trailer-size
```

Where, `image-slot-size` is the size of the image slot and `image-trailer-size` is the size of the image trailer.

The flash map JSON must have the `scratch_address` and `scratch_size` parameters in the bootloader subsections. For example,

Code Listing 6

```
{
    "boot_and_upgrade":
    {
        "bootloader":
        {
            . . .
            "scratch_address": {
                "description": "Address of the scratch area",
                "value": "0x18440000"
            },
            "scratch_size": {
```

Code Listing 6

```
                "description": "Size of the scratch area",
                "value": "0x10000"
            },
        },
        . . .
```

5.3.1.1 Swap with status partition

For the bootloader to determine the current state and what actions should be taken during the current boot operation, it uses metadata stored in the image flash areas. While swapping, some of this metadata is temporarily copied into and out of the scratch area.

This metadata is located at the end of the image flash areas and is called an image trailer. For more information on the image trailer, see the image trailer.

An image trailer contains multiple fields and a swap status partition is one such field which holds a series of records, which records the progress of an image swap. To swap entire images, data are swapped between the two image areas one or more sectors at a time, as follows:

- Sector data in the primary slot is copied into scratch and then erased
- Sector data in the secondary slot is copied into the primary slot and then erased
- Sector data in scratch is copied into the secondary slot

As it swaps images, the bootloader updates the swap status field in a way that allows it to compute how far this swap operation has progressed for each sector. The swap status field can therefore, use to resume a swap operation if the bootloader is halt while a swap operation is ongoing and later reset. For more information on the swap mechanism, see boot swap types and swap status partition.

If the desired upgrade mode is swap scratch with status partition, you should define the `status_address` and `status_size` parameters in the "bootloader" subsection, e.g.:

Code Listing 7

```
{
    "boot_and_upgrade":
    {
        "bootloader":
        {
            . . .
            "status_address": {
                "description": "Address of the swap status partition",
                "value": "0x10038000"
            },
            "status_size": {
                "description": "Size of the swap status partition",
```

Code Listing 7

```
                "value": "0x3800"
            }
        },
        . . .
```

The required size of the status partition relies on many factors. If an insufficient size is given in the flash map, make flag will fail with a message such as:

```
Insufficient swap status area - suggested size 0x3800
```

To calculate the minimal correct size of the status partition, one could specify the value: 0 for the `status_size`. After the intentional make failure, copy the correct size from the error message.

5.3.1.2 Boot swap types

During normal circumstances, when a device boots for the first time or after a successful upgrade, the primary slot contains an up-to-date firmware image. The bootloader verifies the image before execution. In this scenario, no image swapping is necessary.

During device upgrades, a new firmware image is placed in the secondary slot. Bootloader swaps this new image into the primary slot before booting the device. This swapping process can be a two-step procedure.

Test swap: Bootloader performs a temporary, or "test," swap of the image data in flash. The new image is then booted and executed from the primary slot. The new image can update its own status in flash at runtime to indicate successful operation.

Permanent swap (optional): If the new image confirms its successful operation, the bootloader can be instructed to make the swap "permanent" during the next boot. Conversely, if the new image fails its self-test or if a permanent swap is not desired, the bootloader performs a "revert" swap during the next boot. This action restores the original image to the primary slot for execution.

To prevent devices from bricking due to faulty firmware, the bootloader employs a rollback mechanism. If a device crashes after booting a new image, the bootloader will automatically revert it to the previously working image during the next reset. This safeguard allows device firmware to perform self-tests before permanently adopting the new image.

On startup, the bootloader inspects the contents of the flash in the trailer section to decide for each image which of these “swap types” to perform; this decision determines how it proceeds.

Table 1 Swap types and their descriptions

Swap type	Description
`BOOT_SWAP_TYPE_NONE`	The usual or no upgrade case; attempt to boot the contents of the primary slot.
`BOOT_SWAP_TYPE_TEST`	Boot the contents of the secondary slot by swapping images. Unless the swap is made permanent, revert on the next boot.
`BOOT_SWAP_TYPE_PERM`	Permanently swap images and boot the upgraded image firmware.

Bootloader features

Swap type	Description
BOOT_SWAP_TYPE_REVERT	A previous test swap is not made permanent; swap back to the old image whose data are now in the secondary slot. If the old image marks itself "OK" when it boots, the next boot will have swap type BOOT_SWAP_TYPE_NONE.
BOOT_SWAP_TYPE_FAIL	Swap failed because the image to be run is not valid.
BOOT_SWAP_TYPE_PANIC	Swapping encountered an unrecoverable error.

Each image slot contains the metadata which is used by the bootloader to determine the current state and what actions should be taken during the current boot operation. In the case of a swap-based upgrade, the `img_ok` field is updated by the application in the image trailer to make the current image (UPGRADE image) permanent in the primary slot.

For more information, see the image trailer.

5.3.2 Swap without scratch

This algorithm is an alternative to the swap-using-scratch algorithm. It uses an additional sector in the primary slot to make the swap possible. The algorithm operation as follows:

1. Moves all sectors of the primary slot up by one sector. Beginning from N = 0.
2. Copies the N-th sector from the secondary slot to the N-th sector of the primary slot.
3. Copies the (N+1)-th sector from the primary slot to the N-th sector of the secondary slot.
4. Repeats steps 2 and 3 until all the slots' sectors are swapped.

This algorithm is designed so that the higher sector of the primary slot is used only for allowing sectors to move up. Therefore, the most memory-size-effective slot layout is when the primary slot is exactly one sector larger than the secondary slot, although same-sized slots are allowed as well. The algorithm is limited to support sectors of the same sector layout. All slot's sectors should be of the same size.

When using this algorithm, the maximum image size available for the application will be:

```
maximum-image-size = (N-1) * slot-sector-size - image-trailer-sectors-size
```

Where, `N` is the number of sectors in the primary slot and the `image-trailer-sectors-size` is the size of the image trailer rounded up to the total size of the sectors it is occupied. For instance, if the image-trailer-size is equal to 1056 byte and the sector size is equal to 1024 byte, then the `image-trailer-sectors-size` will be equal to 2048 byte.

The algorithm does two erase cycles on the primary slot and one on the secondary slot during each swap. If receiving a new image by the DFU application requires one erase cycle on the secondary slot, result in levelling the flash wear between the slots.

The algorithm is enabled using the `MCUBOOT_SWAP_USING_MOVE` option in the MCUboot configuration *(.\mtb_shared\mcuboot\v1.8.1-cypress\boot\cypress\MCUBootApp\config\mcuboot_config\mcuboot_config.h)*.

The following flags in the image trailer indicate the status of the swap operation:

- **Copy done:** A single byte indicating whether the image in this slot is complete (0x01 = done; 0xff = not done).
- **Image OK:** A single byte indicating whether the image in this slot has been confirmed as good by the user (0x01 = confirmed; 0xff = not confirmed).

5.4 Boot recovery

Per design, the bootloader always boot an application from the primary slot. Whenever there is an update available for the application, it is upgraded to the secondary slot and eventually copied to the primary slot upon verification on the next boot.

During the boot loading, the boot may fail in two scenarios:

- If a bootloader resets in the middle of a swap operation.
- If a bootloader copied to the primary slot fails to boot.

The recovery mechanisms are in-place for the above-mentioned scenarios:

- Reset recovery
- Recovery using factory app

5.4.1 Reset recovery

If the bootloader resets in the middle of a swap operation, the two images may be discontiguous in flash. Bootutil recovers from this condition by using the image trailers to determine how the image parts are distributed in flash.

The first step is to determine where the relevant swap status region is located. Because this region is embedded within the image slots, its location in the flash changes during a swap operation. If the swap status region indicates that the images are not contiguous, the bootloader determines the type of swap operation that is interrupted by reading the swap info field in the active image trailer and extracting the swap type and then resumes the operation by moving an upgraded image into the primary slot and the older image into the secondary slot. If the boot status indicates that an image part is present in the scratch area, this part is copied into the correct location. For more information, see the reset recovery.

After the swap operation has been completed, the bootloader proceeds as though it had just been started.

5.4.2 Recovery using factory app

In this method, a factory app can be built to be placed in the external memory. If the bootloader fails to find a valid image in either primary or secondary slot, it initiates a recovery by copying the factory app to the primary slot to restore the device back to its functional state.

For more information, see the MCUboot-based bootloader with rollback to factory app code example.

5.5 External flash access

External flash support placing the primary/secondary/both slots into an external flash. This helps to increase the available internal flash for the primary slot or to support update operations on MCUs with lower internal flash sizes.

The internal and external memory typically have different sector (erase) sizes, making bootloader operation complicated. To make things simple, the highest sector size amongst the internal and external flash is declared as the sector size. For example, if the sector size of internal flash is 512 bytes and the external is 4 kB, the sector size of internal flash will also be virtually 4 kB (8 × 512 bytes).

Bootloader features

Per the bootloader design:

- Always use internal memory sector size for image trailer
- Locate scratch area in external memory to save internal memory
- Status partition will still reside on internal flash
- The larger sector size must be an even multiple number of smaller sizes

Bootloader accesses the external NOR flash using the Serial Memory Interface (SMIF) aka 'QSPI peripheral block' in PSoC™ 6 MCU. The SMIF block supports interfacing with QSPI devices; most of the PSoC™ 6 MCU development kits include a QSPI NOR flash. For example, the CY8CPROTO-062S2-43439 kit includes the S25FL512S, which is a 64 MB (512-Mbit) QSPI NOR flash. MCUboot for PSoC™ 6 MCU uses the Serial Flash Discoverable Parameter (SFDP) standard to auto-discover the flash read/write commands and other parameters. Ensure that the NOR flash on your board supports this standard. See the ExternalMemory.md for more information on working with the external flash.

For more information on QSPI configurations, see the AN228740 - Usage of Quad SPI (QSPI)/Serial Memory Interface (SMIF) in PSoC™ 6 MCU.

External memory is enabled when `make` flag `USE_EXTERNAL_FLASH` is set to '1'. Value of this flag is set in auto-generated `memorymap.mk` files if, `external_flash` field is present in the JSON file.

Default flash maps with a suffix *smif* are provided in the *./flashmap/*_smif.json* folder for PSoC™ 6 devices, where the presence of external memory in the system is optional.

A sample memory map JSON file (*psoc62_swap_single_smif.json*) with multiple images mapped to internal and external memories and various fields is as follows:

Code Listing 8

```
{
    "external_flash": [
        {
            "model": "S25HS256T" >> Flash model
        }
    ],
    "boot_and_upgrade":
        {
        "bootloader": {
            "address": {
                "description": "Address of the bootloader",
                "value": "0x10000000"
            },
            "size": {
                "description": "Size of the bootloader",
                "value": "0x18000"
            },
            "scratch_address": {
```

Code Listing 8

```
                "description": "Address of the scratch area",
                "value": "0x18440000" >> External Flash
            },
            "scratch_size": {
                "description": "Size of the scratch area",
                "value": "0x80000"
            },
            "status_address": {
                "description": "Address of the swap status partition",
                "value": "0x10058200" >> Internal Flash
            },
            "status_size": {
                "description": "Size of the swap status partition",
                "value": "0x2400"
            }
        },
        "application_1": {
            "address": {
                "description": "Address of the application primary slot",
                "value": "0x10018000” >> Internal Flash
            },
            "size": {
                "description": "Size of the application primary slot",
                "value": "0x40200"
            },
            "upgrade_address": {
                "description": "Address of the application secondary slot",
                "value": "0x18000000” >> External Flash
            },
            "upgrade_size": {
                "description": "Size of the application secondary slot",
                "value": "0x40200"
            }
        }
    }
}
```

Bootloader features

The following memory map combinations are possible with the external flash for upgrade.

1. **Overwrite**: Primary slot in internal and secondary slot in external flash.
2. **Swap**: Primary and swap status partition in internal; secondary and scratch partition in external flash.

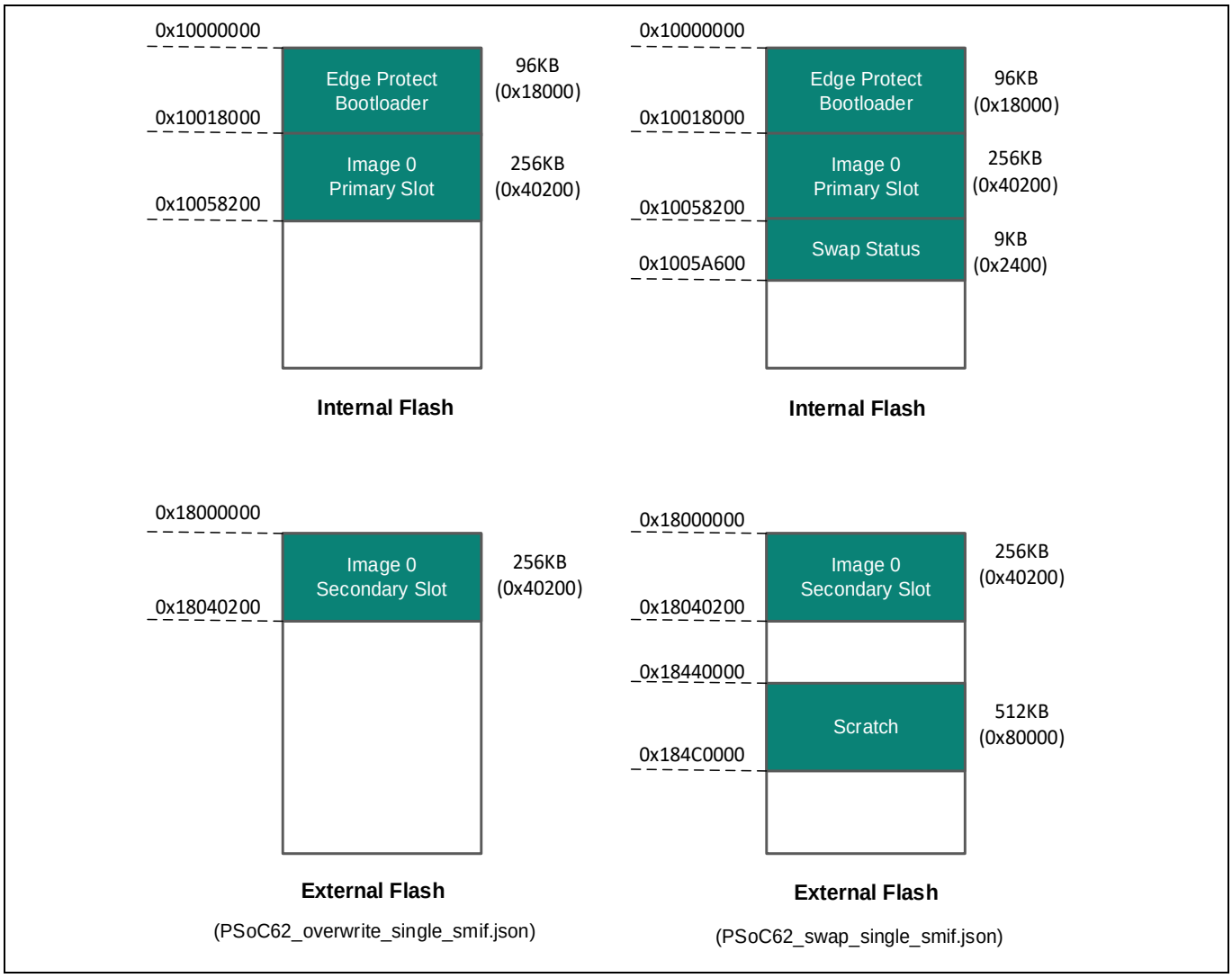


Figure 6 External flash memory map

5.6 Execute in Place (XIP)

In XIP mode, place the firmware image in the external memory and execute it directly from there. This mode is useful for devices with small internal flash or when one requires reserving internal flash for other purposes.

On PSoC™ 6 devices, this is an optional feature and can be enabled by adding `"mode": "XIP"` in the `"external_flash"` section of the JSON file if the primary slot is placed in the external flash.

During the pre-build, a `USE_XIP` flag is added to the auto-generated *flashmap.mk* file based on the XIP mode addition in the JSON.

A sample memory map JSON file (*psoc62_xip_overwrite.json*) with XIP enabled on external flash is shown in Code Listing 9.

Code Listing 9

```
{
    "external_flash": [
        {
            "model": "S25HS256T",
            "mode": "XIP"
        }
    ],
    "boot_and_upgrade":

      ...
```

The following memory map combinations are possible with the external flash for upgrade in XIP mode.

- **Overwrite**: Primary and secondary slot in external flash.
- **Swap**: Swap status partition in internal; primary, secondary, and scratch partition in external flash.

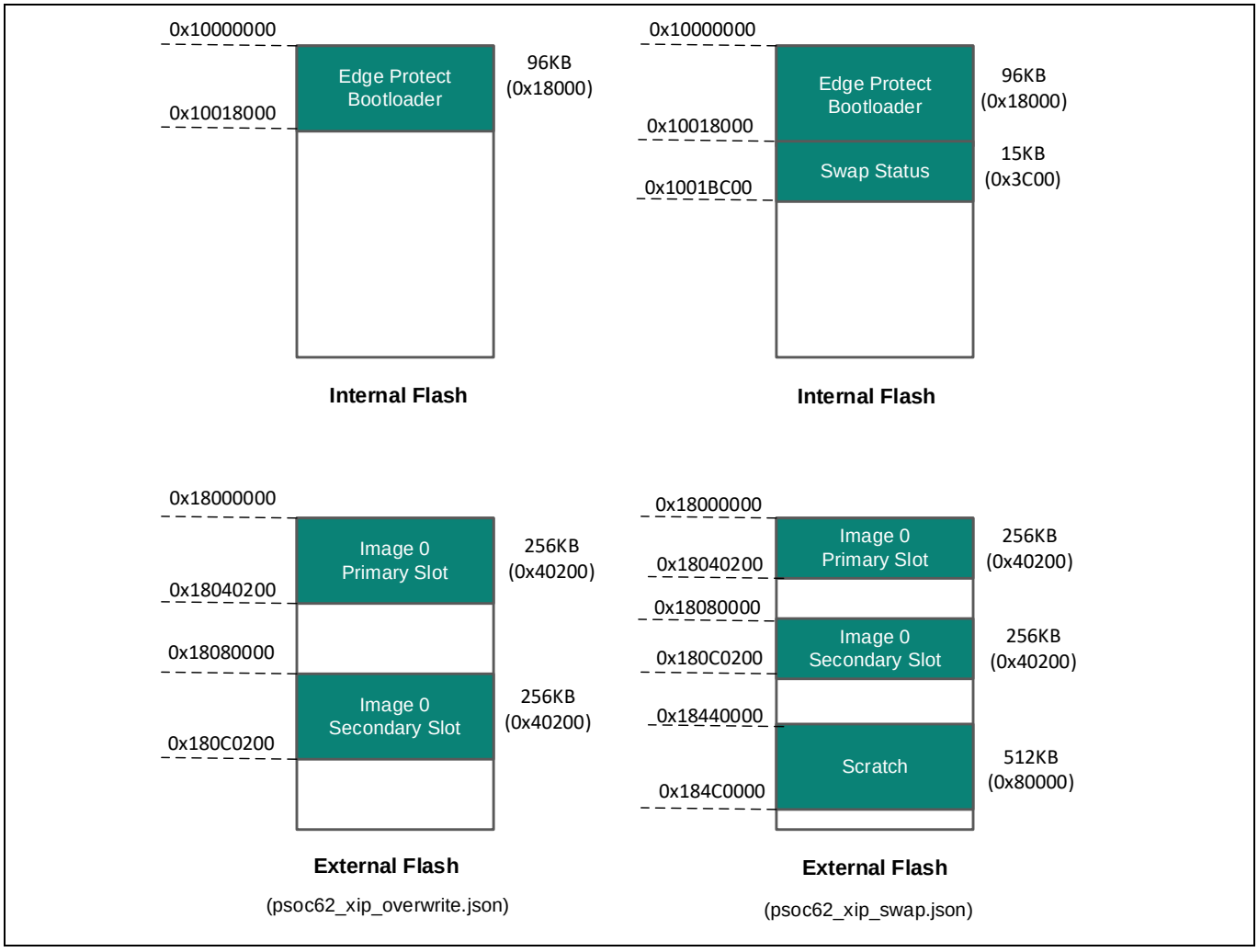


Figure 7 External flash memory with XIP mode

5.7 Secured boot

Bootloader employs a robust validation process to ensure the integrity and authenticity of applications before execution. Bootloader leverages the image format of a standard MCUboot image structure. Before diving into the validation process, it is essential to understand the basic structure of an MCUboot image. Typically, an image consists of:

- **Image header:** Contains metadata about the image, such as its size, version, load address, entry point, and security information.
- **Image payload:** The actual application code and data.
- **Image signature:** A digital signature for verifying the image's integrity.
- **Image trailer:** It consists of metadata that the bootloader determines during the boot operation.

Applications that are using the bootloader should support the MCUboot image format. Else, the bootloader fails to recognize the image. Tools like imgtool can transform raw application images into a MCUboot-compatible format.

The image begins with a small header, the executable image itself, and a set of metadata after the image and the image trailer. The format looks as shown in the following figure.

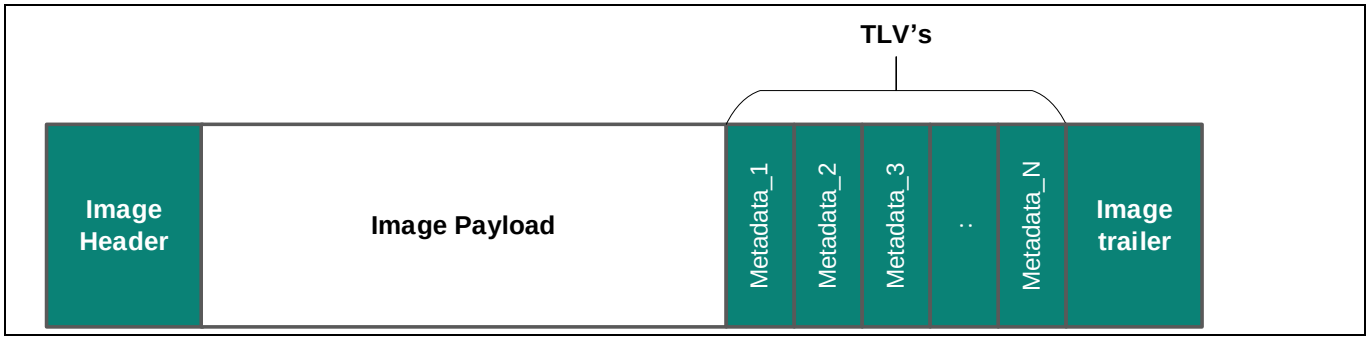


Figure 8 Bootloader image format

The header maintains the information required by the bootloader to load the application such as, the load address, header size, image size, image version, etc., as shown in Code Listing 10.

Code Listing 10 Bootloader header

```
struct image_version {
    uint8_t iv_major;
    uint8_t iv_minor;
    uint16_t iv_revision;
    uint32_t iv_build_num;
};

/** Image header.  All fields are in little endian byte order. */
struct image_header {
    uint32_t ih_magic;
    uint32_t ih_load_addr;
```

Bootloader features

Code Listing 10 Bootloader header

```
    uint16_t ih_hdr_size;           /* Size of image header (bytes). */
    uint16_t ih_protect_tlv_size;   /* Size of protected TLV area (bytes).
*/
    uint32_t ih_img_size;           /* Does not include header. */
    uint32_t ih_flags;              /* IMAGE_F_[...]. */
    struct image_version ih_ver;
    uint32_t _pad1;
};
```

Each of the metadata field is in tag-length-value (TLV) format with the following tags:

- `IMAGE_TLV_SHA256`: It has the calculated hash (SHA256) of the header and image combined.
- `IMAGE_TLV_KEYHASH`: It has the hash of public key, which is used to verify the signature of the image.
- `IMAGE_TLV_RSA2048_PSS`: Digital signature of the hash from `IMAGE_TLV_SHA256`. The signature is generated with RSA-2048 private key.
- `IMAGE_TLV_ECDSA224`: Digital signature of the hash from `IMAGE_TLV_SHA256`. The signature is generated with the ECDSA P-224 curve private key.
- `IMAGE_TLV_ECDSA256`: Digital signature of the hash from `IMAGE_TLV_SHA256`. The signature is generated with the ECDSA P-224 curve private key.

Either of RSA-2048/ECDSA P-224/ECDSA P-256 can be used for the signature generation.

For the bootloader to determine the current state and what actions should be taken during the current boot operation, it uses metadata stored in the image flash areas. While swapping, some of this metadata is temporarily copied into and out of the scratch area. This metadata is located at the end of the image flash areas and is called an image trailer. See Section 5.3.1.2 for more information.

Bootloader handles the firmware authentication after startup and the steps performed by the bootloader during the bootup are as follows:

1. During power on, the bootloader initializes the system hardware.
2. Identifies available image slots in the flash memory based on the flash map.
3. Determines the primary image slot based on the image version. The image with the greater version is validated first.
4. Read the image header from the selected slot based on the highest image version. Downgrade prevention is enabled to avoid an older firmware version for upgrade.
5. If the selected slot is primary, then the image is validated and loaded.
6. If the selected slot is secondary, then the image is first validated.
7. Once validated, the image in the secondary slot is either overwritten or swapped to the primary slot based on the boot flag set in the image header as mentioned in Section 5.2 and Section 5.3.1.2 and then loaded from the primary slot.

The image validation process by the bootloader includes the following:

- Reading the header to get the image payload size and start address
- Calculate the hash (SHA-256) of the image payload and header combined.
- Verify the generated hash with the stored hash in TLV (`IMAGE_TLV_SHA256`).

Bootloader features

- Calculate the hash of the public key required to validate the signed image.
- Verify the generated hash of the public key with the stored hash in image TLV.
- Verify the image signature using the public key (RSA-2048/ ECC-256)

Therefore, failure in any of the above-mentioned steps leads to boot failure.

At a high level, PSoC™ 64 consists of the following two cores: Cortex®-M0+ and Cortex®-M4. The bootloader runs on the Cortex®-M0+ core and is considered as a secure core and the user application runs on either Cortex®-M0+ or Cortex®-M4 core, or both. The authenticity of the user application is always validated by the Cortex®-M0+ bootloader before handing over the control to the user application on this secured MCU.

In a "secure boot" functionality, Infineon provides an immutable boot code (Flashboot) programmed in a factory that launches the bootloader. The bootloader is itself signed, so that the Flashboot can verify the integrity of the bootloader. This verified bootloader then verifies the signature of the user application before launching it. The device fails to boot in case of failed authentication. The signature verification happens in the primary slot every time before booting when `MCUBOOT_VALIDATE_PRIMARY_SLOT` is defined. Additionally, it verifies the signature of the image in the secondary slot before copying it to the primary slot.

See the Bootloader code example for more details on the bootloader implementation for PSoC™ 6 devices.

5.8 Secure firmware update

Bootloader does the job of authenticating the firmware during device startup and switching the firmware during the firmware update process. But it does not have the capability to update the firmware by itself. Downloading the new firmware can be achieved with the help of wired or wireless transport middleware.

Infineon uses Device Firmware Update (DFU) middleware for the wired (UART, I2C, SPI) update and OTA (over-the-air) middleware for the wireless (Bluetooth® LE, Wi-Fi) update. You can use any of these means (wired/wireless) to transport the code update.

This middleware can be integrated as part of the bootloader or can be part of an end application to communicate with the host and download the MCUboot-compatible hex for the new firmware image.

A DFU MW based user application downloads the new image to the secondary slot and transfers the control to the bootloader. The bootloader then checks for the image version and performs the next level validation check and depending on the configuration, it swaps or overwrites the image of the secondary slot to the primary boot slot if the secondary slot is valid. After the transfer to the primary slot, the bootloader verifies the primary image and boots to the same if verified.

The following code examples demonstrate the firmware upgrades support in the end application for the bootloader over wired and wireless on PSoC™ 6 devices.

- DFU: mtb-example-psoc6-security
- OTA: mtb-example-ota-mqtt

To learn more about incorporating the DFU middleware into either your application or bootloader, see the AN236282 - Device Firmware Update (DFU) middleware (MW) for ModusToolbox™ application note.

5.9 Key management

The bootloader supports RSA-2048, RSA-3072, ECDSA-P256. and ED25519 keys for validating applications in the primary and secondary slot. The private key is held by the user who signs the application before flashing to the device memory and is protected. The public key is embedded in the bootloader with its hash calculated and added to the `IMAGE_TLV_KEYHASH` entry.

Bootloader features

While building the bootloader, you must generate a new set of public-private key pair and embed the public key into the bootloader. The Python program `scripts/imgtool.py` that is part of the bootloader can be used to generate the keys and sign images.

The bootloader by default supports the RSA-2048 key. But this can be changed by enabling the respective flags (`MCUBOOT_SIGN_RSA / MCUBOOT_SIGN_EC256`) in the bootloader configuration under `boot\cypress\MCUBootApp\config\mcuboot_config\mcuboot_config.h`.

For more information on generating and embedding the keys and signing the images using imgtool, see the tool documentation.

Note: *The public key is embedded into the bootloader only when `MCUBOOT_VALIDATE_PRIMARY_SLOT` is defined.*

As an alternative, the bootloader can be made independent of the keys (avoiding the incorporation of the public key into the code) by enabling the `MCUBOOT_HW_KEY` option in the bootloader configuration file `mcuboot_config.h`. In this case, the public key is added to the TLV entries and the hash of the public key must be provisioned to the target device and MCUboot must retrieve the key-hash from there. See the MCUBoot HW KEY for more information.

The mtb-example-mcuboot-basic code example demonstrates the bootloader on a PSoC™ 6 device. The readme of this code example demonstrates the steps to create the keys using the imgtool. It also demonstrates how the public key is embedded in the bootloader image and how to sign the image with the private key as part of the post-build scripts in the makefile.

5.10 Downgrade prevention

Downgrade prevention is a feature which ensures that only firmware with a higher version or security counter can replace an existing image. This safeguard against malicious attempts to revert the device to a potentially vulnerable older version.

- **Software-based downgrade prevention**

Software-based downgrade prevention compares image version numbers to block installations of older firmware. This feature is activated by the `MCUBOOT_DOWNGRADE_PREVENTION` option and works exclusively with the overwrite-based image update strategy (`MCUBOOT_OVERWRITE_ONLY`).

- **Hardware-based downgrade prevention**

Each signed image can contain a security counter in its protected TLV area, which can be added to the image using the `-s` option of the imgtool script. During the hardware-based downgrade prevention (alias rollback protection) the new image's security counter will be compared with the currently active security counter value which must be stored in a nonvolatile and trusted component of the device. It is beneficial to handle this counter independently from the image version number:

- No need to increase with each software release.
- Ensures to do software downgrade to some extent: if the security counter has the same value in the older image then it is accepted.

It is an optional step of the image validation process and can be enabled with the `MCUBOOT_HW_ROLLBACK_PROT` configuration option. When enabled, the target must provide an implementation of the security counter interface defined in the *boot/bootutil/include/security_cnt.h* file.

The software-based downgrade prevention can be implemented on PSoC™ 6 device using the existing mtb-example-mcuboot-basic code example. Set `USE_SW_DOWNGRADE_PREV` to '1' to enable the `MCUBOOT_DOWNGRADE_PREVENTION` flag in the bootloader Makefile (`.\MCUboot-Based_Basic_Bootloader\bootloader_app\Makefile`). This will avoid older firmware versions from being upgraded.

5.11 Serial logging

The bootloader logs the status and reports over the serial port during the boot and upgrade process. The logs are printed over an UART port and are determined based on the log levels that are set during the compile time. This is a configurable feature that can be enabled/disabled during compile time.

There are five log levels with a unique value defined in the bootloader in *\boot\cypress\MCUBootApp\config\mcuboot_config\mcuboot_logging.h* file.

```
#define MCUBOOT_LOG_LEVEL_OFF      0
#define MCUBOOT_LOG_LEVEL_ERROR    1
#define MCUBOOT_LOG_LEVEL_WARNING  2
#define MCUBOOT_LOG_LEVEL_INFO     3
#define MCUBOOT_LOG_LEVEL_DEBUG    4
```

You can define the global logging level by defining any of the above-mentioned log level in the *<application>/bootloader_app/Makefile*.

`MCUBOOT_LOG_LEVEL` sets the minimum level that will be logged. The function priority is:

```
MCUBOOT_LOG_ERR > MCUBOOT_LOG_WRN > MCUBOOT_LOG_INF > MCUBOOT_LOG_DBG
```

5.12 Fault injection hardening (FIH)

To mitigate hardware-based fault injection attacks (e.g., power glitching, EM pulses) aimed at executing unauthorized code, MCUboot incorporates software countermeasures. These attacks exploit induced CPU behavior changes like skipped instructions, modified registers, corrupted memory reads, or altered instruction decoding to bypass security mechanisms.

While these software countermeasures enhance resistance to fault injection attacks, they do not provide absolute protection. Their activation is controlled by the `MCUBOOT_FIH_PROFILE_*` macros, offering four protection levels: none, moderate, medium, and maximum. These countermeasures are primarily integrated into the core MCUboot codebase (boot/bootutil/src), focusing on critical code paths susceptible to fault injection. Independent cryptographic libraries like mbed-crypto or tinycrypt might offer additional protections, but this needs to be assessed separately.

The main hardening code are in:

- *boot/bootutil/include/bootutil/fault_injection_hardening.h*
- *boot/bootutil/src/fault_injection_hardening.c*

The fault injection mitigation library has support for different measures, which can either be enabled/disabled separately or by defining one of the `MCUBOOT_FIH_PROFILE`s. For more information on the measures supported by each profile, see the *fault_injection_hardening.h* file.

Bootloader features

On PSoC™ 6 device, the FIH profiles are set in the bootloader application Makefile (*MCUboot-Based_Basic_Bootloader\bootloader_app\Makefile*) in the mtb-example-mcuboot-basic code example as follows:

Code Listing 11 Enabling FIH profile for PSoC™ 6 device

```
FIH_PROFILE_LEVEL_LIST:=OFF LOW MEDIUM HIGH
FIH_PROFILE_LEVEL?=MEDIUM

# Check FIH profile param
ifneq ($(filter $(FIH_PROFILE_LEVEL), $(FIH_PROFILE_LEVEL_LIST)),)
ifneq ($(FIH_PROFILE_LEVEL), OFF)
DEFINES+=MCUBOOT_FIH_PROFILE_ON
DEFINES+=MCUBOOT_FIH_PROFILE_$(FIH_PROFILE_LEVEL)
endif
else
$(error Wrong FIH_PROFILE_LEVEL param)
endif
```

6 Bootloader in ModusToolbox™

This section demonstrates creating a new bootloader application for PSoC™ 6 device using MCUboot libraries. This bootloader runs on CM0+ core. For more information on the design part, see the mtb-example-mcuboot-basic readme file.

1. Install ModusToolbox™ v3.2 or above and start with creating a new ModusToolbox™ application as shown in the following figure.

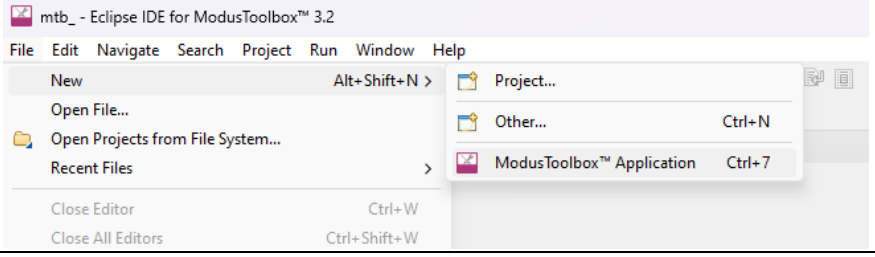


Figure 1 Create new ModusToolbox™ Application

2. As part of new application, select the board BSP/kit you are using. For demonstration purpose, selected **CY8CKIT-062s2-43012** kit as shown in the following figure and click **Next >**

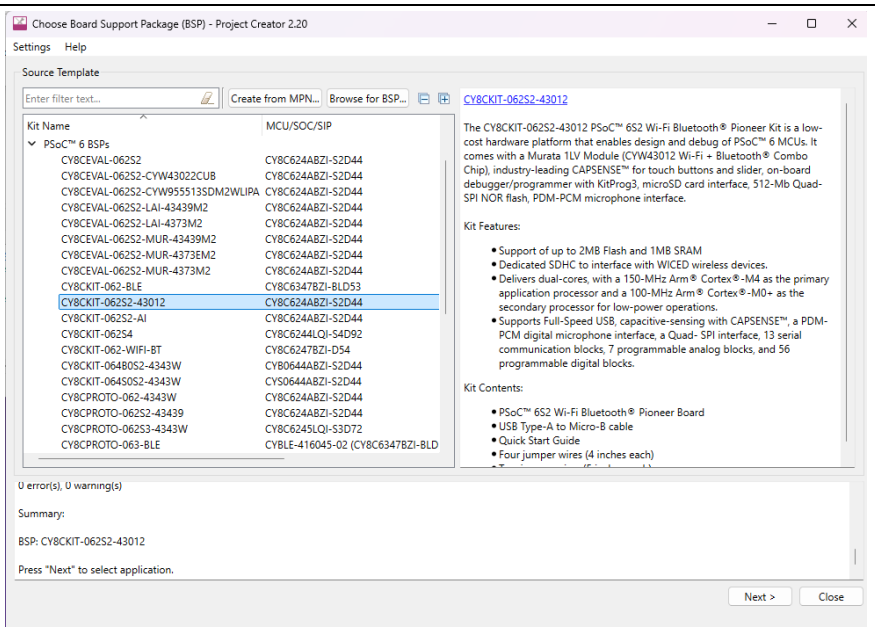


Figure 2 Selecting kit

3. After choosing the BSP, select an appropriate application as shown in the following figure. Since, demonstrating the bootloader in this document, select **MCUBoot-Based Basic Bootloader** as shown in the following figure and click **Next >**.

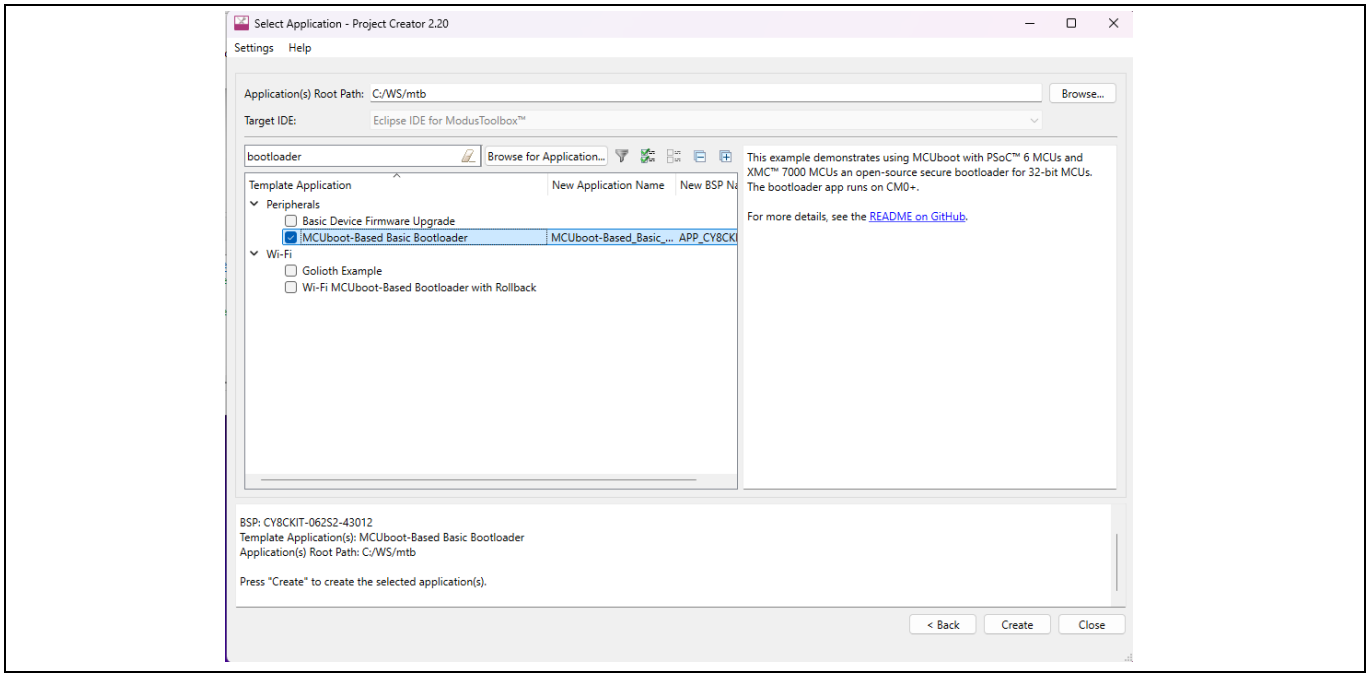


Figure 3 Select the bootloader application

Note: After selecting the bootloader application, it takes a while to load all the required libraries and the projects.

The bootloader application consists of two projects: bootloader and blinky projects. The folder structure of the bootloader application is shown in Figure 4.

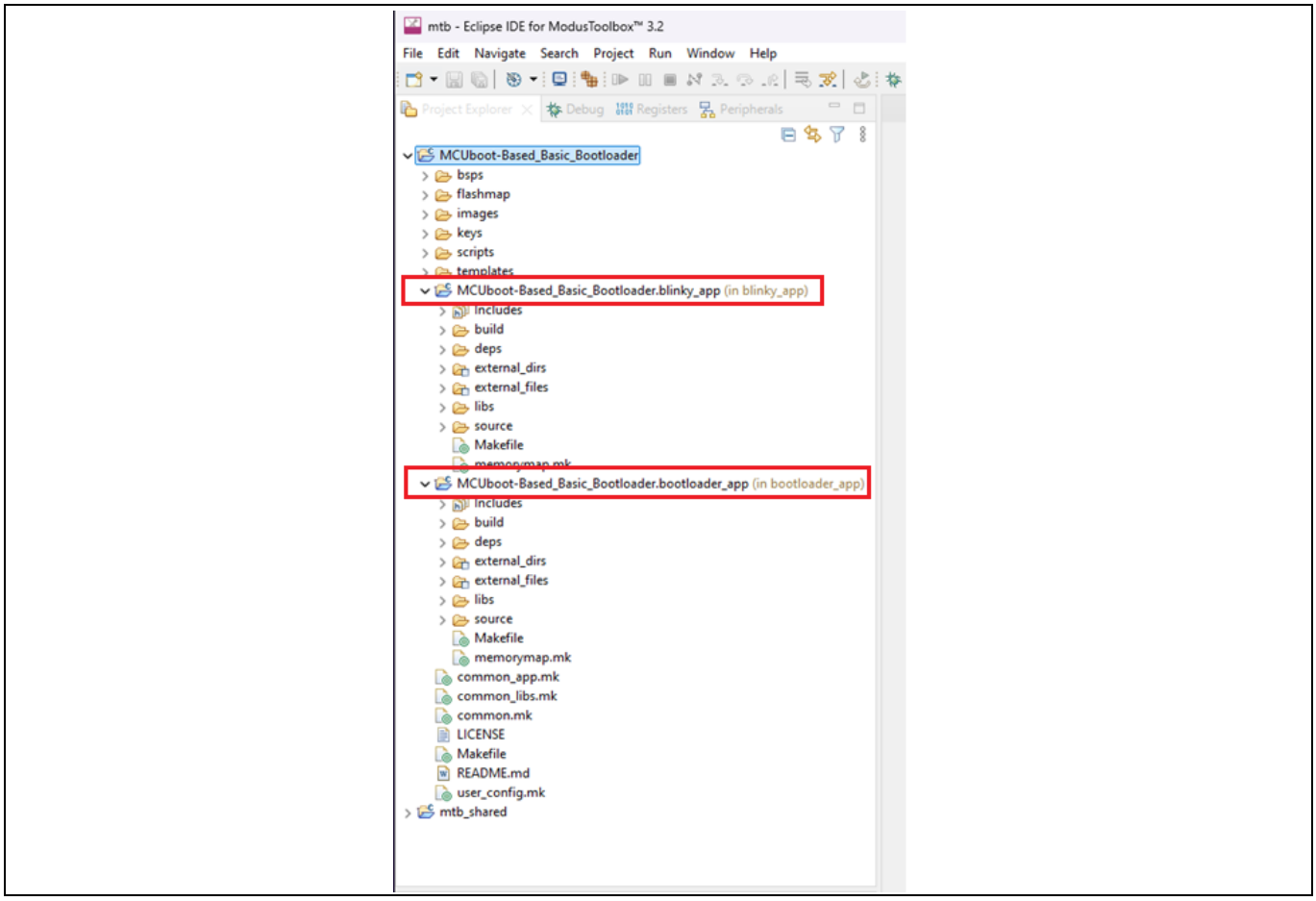


Figure 4 Bootloader directory structure

The bootloader project in the above figure is the boot application that does verification and validations of the blinky project. But the core bootloader library aka bootutil is part of \mtb_shared directory in the workspace as shown in Figure 5.

The bootutil library performs most of the functions of a bootloader. In particular, the piece that is missing in the final step of jumping to the main image. The last step is instead implemented by the boot application. Bootloader functionality is separated in this manner to enable unit testing of the bootloader. A library can be unit tested, but an application cannot. Therefore, functionality is delegated to the bootutil library when possible.

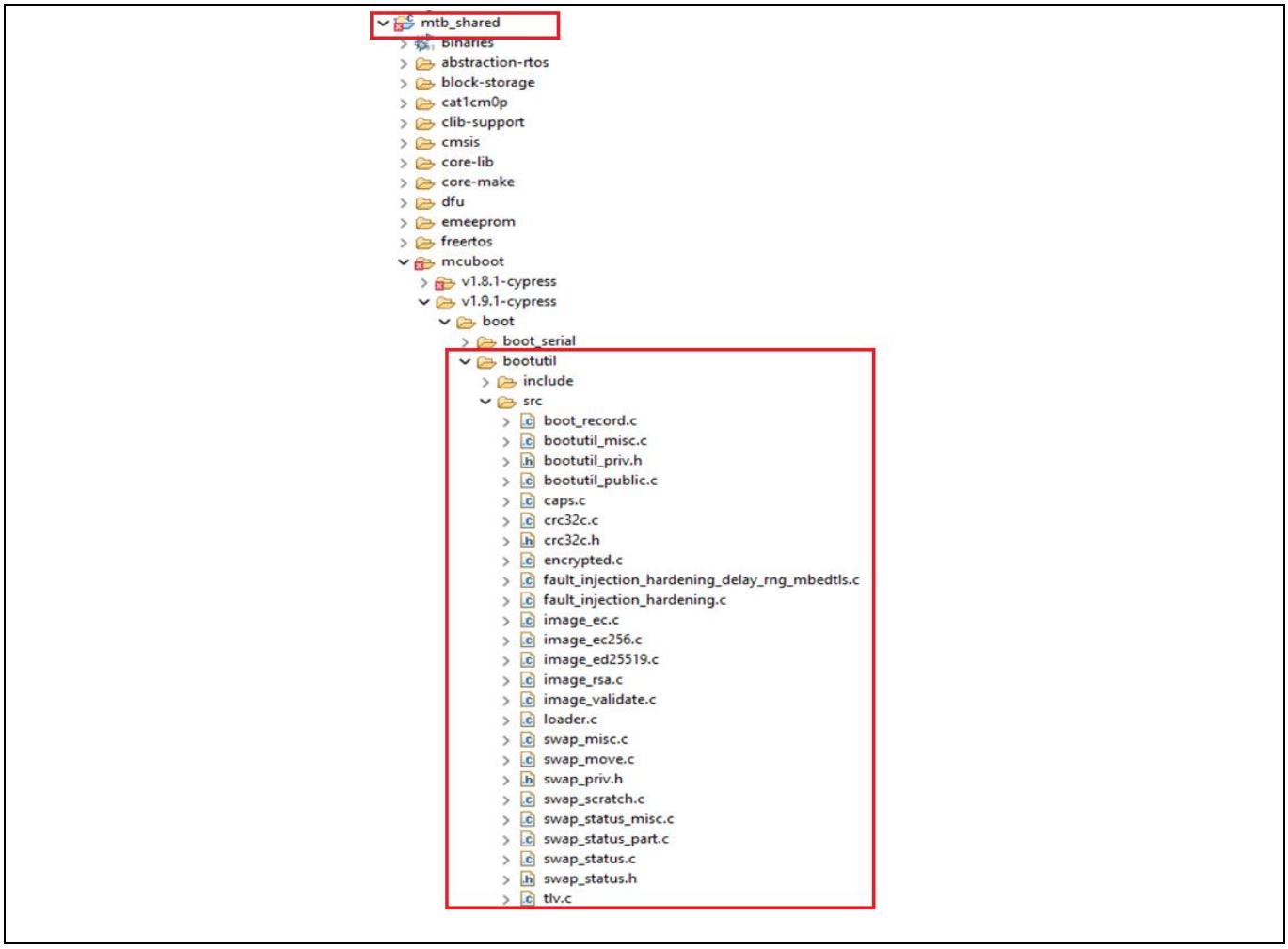


Figure 5 Bootloader core firmware

4. Build the application by clicking **Build Application** from the Quick Panel as shown in Figure 6.

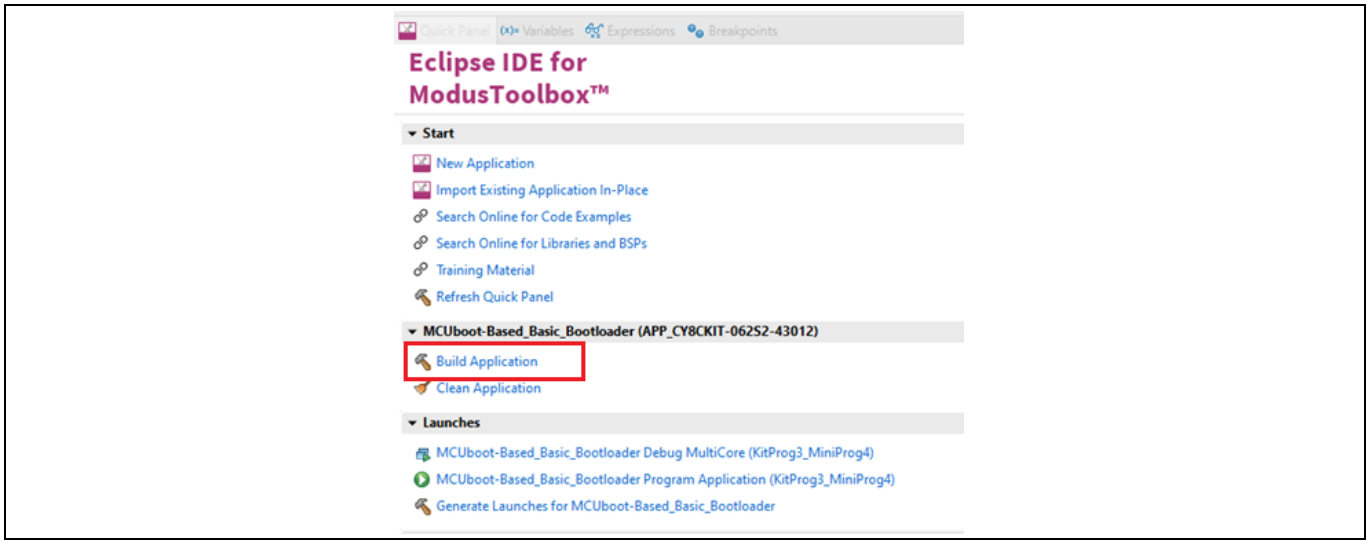
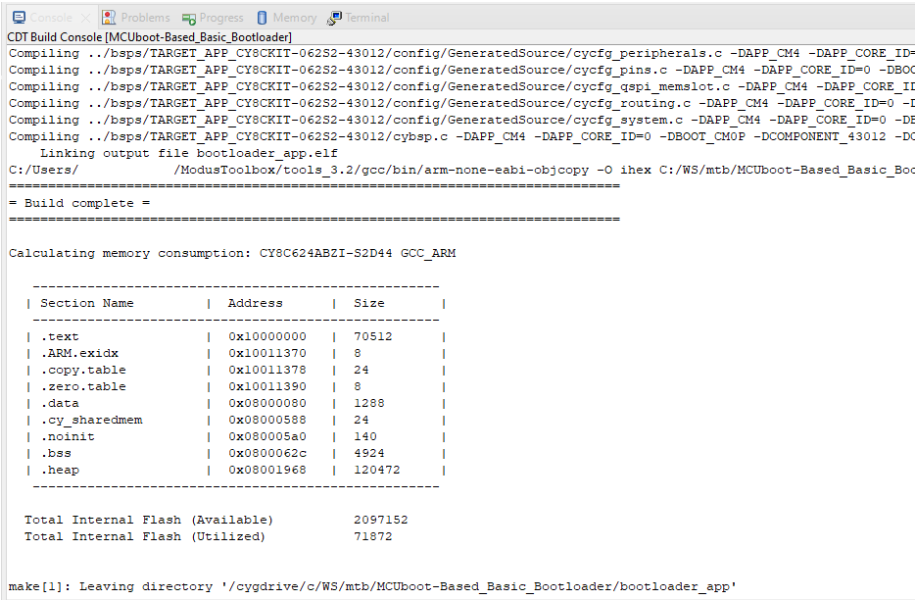


Figure 6 Building Application

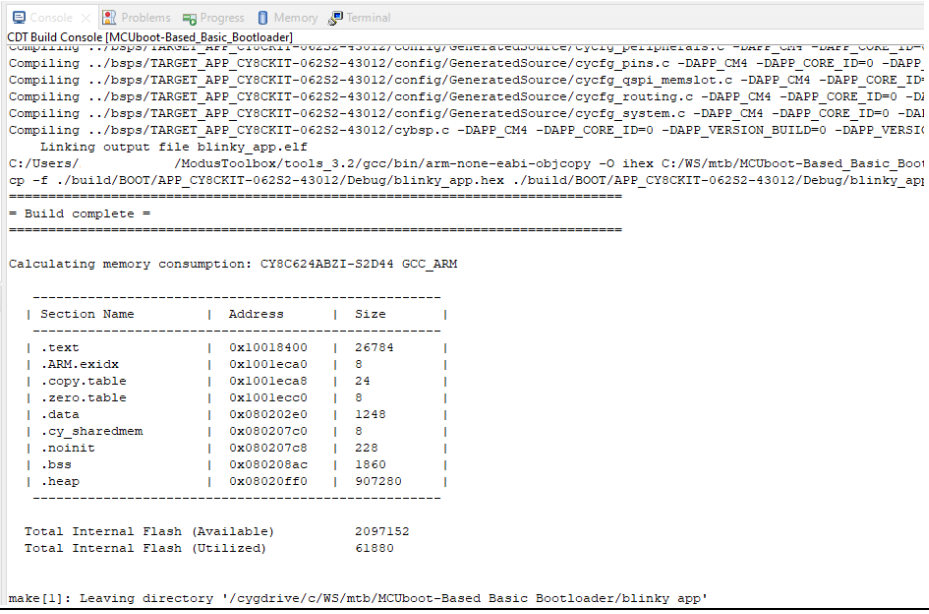
The build process first builds the bootloader project and then the blinky project. Figure 7 and Figure 8 show the successful build log for both the applications.

Bootloader in ModusToolbox™



```
Console  Problems  Progress  Memory  Terminal
CDT Build Console [MCUboot-Based_Basic_Bootloader]
Compiling ../bsps/TARGET_APP_CY8CKIT-062S2-43012/config/GeneratedSource/cycfg_peripherals.c -DAPP_CM4 -DAPP_CORE_ID=
Compiling ../bsps/TARGET_APP_CY8CKIT-062S2-43012/config/GeneratedSource/cycfg_pins.c -DAPP_CM4 -DAPP_CORE_ID=0 -DBOO
Compiling ../bsps/TARGET_APP_CY8CKIT-062S2-43012/config/GeneratedSource/cycfg_qspi_memslot.c -DAPP_CM4 -DAPP_CORE_ID
Compiling ../bsps/TARGET_APP_CY8CKIT-062S2-43012/config/GeneratedSource/cycfg_routing.c -DAPP_CM4 -DAPP_CORE_ID=0 -D
Compiling ../bsps/TARGET_APP_CY8CKIT-062S2-43012/config/GeneratedSource/cycfg_system.c -DAPP_CM4 -DAPP_CORE_ID=0 -DE
Compiling ../bsps/TARGET_APP_CY8CKIT-062S2-43012/cybsp.c -DAPP_CM4 -DAPP_CORE_ID=0 -DBOOT_CM0P -DCOMPONENT_43012 -DC
    Linking output file bootloader_app.elf
C:/Users/            /ModusToolbox/tools_3.2/gcc/bin/arm-none-eabi-objcopy -O ihex C:/WS/mtb/MCUboot-Based_Basic_Boo
================================================================================
= Build complete =
================================================================================

Calculating memory consumption: CY8C624ABZI-S2D44 GCC_ARM

  -------------------------------------------------
 | Section Name          |  Address      |  Size       |
  -------------------------------------------------
 | .text                 |  0x10000000   |  70512      |
 | .ARM.exidx            |  0x10011370   |  8          |
 | .copy.table           |  0x10011378   |  24         |
 | .zero.table           |  0x10011390   |  8          |
 | .data                 |  0x08000080   |  1288       |
 | .cy_sharedmem         |  0x08000588   |  24         |
 | .noinit               |  0x080005a0   |  140        |
 | .bss                  |  0x0800062c   |  4924       |
 | .heap                 |  0x08001968   |  120472     |
  -------------------------------------------------

 Total Internal Flash (Available)          2097152
 Total Internal Flash (Utilized)           71872


make[1]: Leaving directory '/cygdrive/c/WS/mtb/MCUboot-Based_Basic_Bootloader/bootloader_app'
```

Figure 7 Bootloader successful build



```
Console  Problems  Progress  Memory  Terminal
CDT Build Console [MCUboot-Based_Basic_Bootloader]
Compiling ../bsps/TARGET_APP_CY8CKIT-062S2-43012/config/GeneratedSource/cycfg_pins.c -DAPP_CM4 -DAPP_CORE_ID=0 -DAPP_
Compiling ../bsps/TARGET_APP_CY8CKIT-062S2-43012/config/GeneratedSource/cycfg_qspi_memslot.c -DAPP_CM4 -DAPP_CORE_ID=
Compiling ../bsps/TARGET_APP_CY8CKIT-062S2-43012/config/GeneratedSource/cycfg_routing.c -DAPP_CM4 -DAPP_CORE_ID=0 -DA
Compiling ../bsps/TARGET_APP_CY8CKIT-062S2-43012/config/GeneratedSource/cycfg_system.c -DAPP_CM4 -DAPP_CORE_ID=0 -DAP
Compiling ../bsps/TARGET_APP_CY8CKIT-062S2-43012/cybsp.c -DAPP_CM4 -DAPP_CORE_ID=0 -DAPP_VERSION_BUILD=0 -DAPP_VERSI
    Linking output file blinky_app.elf
C:/Users/            /ModusToolbox/tools_3.2/gcc/bin/arm-none-eabi-objcopy -O ihex C:/WS/mtb/MCUboot-Based_Basic_Boot
cp -f ./build/BOOT/APP_CY8CKIT-062S2-43012/Debug/blinky_app.hex ./build/BOOT/APP_CY8CKIT-062S2-43012/Debug/blinky_app
================================================================================
= Build complete =
================================================================================

Calculating memory consumption: CY8C624ABZI-S2D44 GCC_ARM

  -------------------------------------------------
 | Section Name          |  Address      |  Size       |
  -------------------------------------------------
 | .text                 |  0x10018400   |  26784      |
 | .ARM.exidx            |  0x1001eca0   |  8          |
 | .copy.table           |  0x1001eca8   |  24         |
 | .zero.table           |  0x1001ecc0   |  8          |
 | .data                 |  0x080202e0   |  1248       |
 | .cy_sharedmem         |  0x080207c0   |  8          |
 | .noinit               |  0x080207c8   |  228        |
 | .bss                  |  0x080208ac   |  1860       |
 | .heap                 |  0x08020ff0   |  907280     |
  -------------------------------------------------

 Total Internal Flash (Available)          2097152
 Total Internal Flash (Utilized)           61880


make[1]: Leaving directory '/cygdrive/c/WS/mtb/MCUboot-Based_Basic_Bootloader/blinky_app'
```

Figure 8 Blinky successful build

7 Programming and debugging the bootloader

This section mainly describes the programming and debugging steps on the PSoC™ 6 device.

See the *Program the device* section on how to program the bootloader application on a PSoC™ 6 device using ModusToolbox™ IDE in AN228571.

When you add and change code in your application, it is likely that something will not work as expected. At that point, you need to debug the application to determine what is wrong, or how to optimize the desired behavior. Similar to building an application and programming the board, you can use an IDE or command-line tool options to debug the application.

See the *Debugging the application using KitProg3/MiniProg4* section on how to debug an application on ModusToolbox™ IDE in AN228571.

References

[1] MCUboot documentation

[2] MCUboot GitHub

[3] ModusToolbox™ software

Application notes:

[4] Getting started with PSoC™ 6 MCU on ModusToolbox™ software

[5] Device Firmware Update (DFU) middleware (MW) for ModusToolbox™

Code examples:

[6] MCUboot-based basic bootloader

[7] PSoC™ 6 MCU: MCUboot-based bootloader with rollback to factory app

[8] PSoC™ 6 MCU: Basic Device Firmware Upgrade (DFU)

Reference manuals

[9] PSoC™ 6 reference manuals

Revision history

Document revision	Date	Description of changes
**	2024-09-18	Initial release

Trademarks
All referenced product or service names and trademarks are the property of their respective owners.

Edition 2024-09-18

Published by

Infineon Technologies AG

81726 Munich, Germany

Do you have a question about this document?

Email: erratum@infineon.com

Document reference

002-40585 Rev. **

Important notice

The information contained in this application note is given as a hint for the implementation of the product only and shall in no event be regarded as a description or warranty of a certain functionality, condition or quality of the product. Before implementation of the product, the recipient of this application note must verify any function and other technical information given herein in the real application. Infineon Technologies hereby disclaims any and all warranties and liabilities of any kind (including without limitation warranties of non-infringement of intellectual property rights of any third party) with respect to any and all information given in this application note.

The data contained in this document is exclusively intended for technically trained staff. It is the responsibility of customer's technical departments to evaluate the suitability of the product for the intended application and the completeness of the product information given in this document with respect to such application.

Warnings

Due to technical requirements products may contain dangerous substances. For information on the types in question please contact your nearest Infineon Technologies office.

Except as otherwise explicitly approved by Infineon Technologies in a written document signed by authorized representatives of Infineon Technologies, Infineon Technologies' products may not be used in any applications where a failure of the product or any consequences of the use thereof can reasonably be expected to result in personal injury.