CycloneBOOT

General User Manual

1 Table of Contents

2 Conventions

Whenever a limited feature is discussed, the following textbox will be present.


This feature is only available on CycloneBOOT Pro and CycloneBOOT Ultimate versions.

When a specific point is discussed in detail, the following symbol will be present:


This is a deep dive into a specific point.

A sidenote or a question is discussed within sections marked with the following symbol:


This is an explanation about some concept or an answer to a question.

3 Introduction to CycloneBOOT

CycloneBOOT is an embedded secure bootloader targeting 32-bit single and dual core microcontrollers. It can run under minimal system constraints, consuming only several kilobytes of memory.

Thanks to its architecture, it can run independent to the underlying application, either on “bare-metal” or on a RTOS.

Using CycloneBOOT, you will be able to update your firmware securely with peace of mind against malicious actors. CycloneBOOT employs several encryption algorithms to make sure that your firmware is always under lock and key. During the firmware update process, CycloneBOOT rigorously verifies the integrity of each data packet. Once the authenticity and the integrity of the received firmware image is established, the current firmware image running in the microcontroller is backed-up (either in the same internal flash memory or in an external memory) and the new firmware is image is written to the flash memory. If the latest installed firmware is not acceptable, CycloneBOOT can easily switch the firmware image to the older version, falling back to the last known working state.

CycloneBOOT is protocol agnostic, meaning that a firmware update can be performed using various physical media (Ethernet LAN, Wi-Fi, Cellular Modem, USB, UART, SD card...). However, with our experience with TCP/IP protocols we can provide you with ready-to-go demonstrations by bundling CycloneBOOT with CycloneTCP/SSL/SSH.

CycloneBOOT comes with a ready to use PC software utility (compatible with either Windows or Linux) that can generate and sign encrypted firmware images. These will then be used by the bootloader to verify the authenticity and integrity of the bootable firmware image before performing a software update.

CycloneBOOT is available either as open source (GPLv2 license) or under a royalty-free commercial license (non-GPL license). We also propose an evaluation license (90-day license in source form) with technical support for easier onboarding and effective evaluation of our software.

4 CycloneBOOT modes

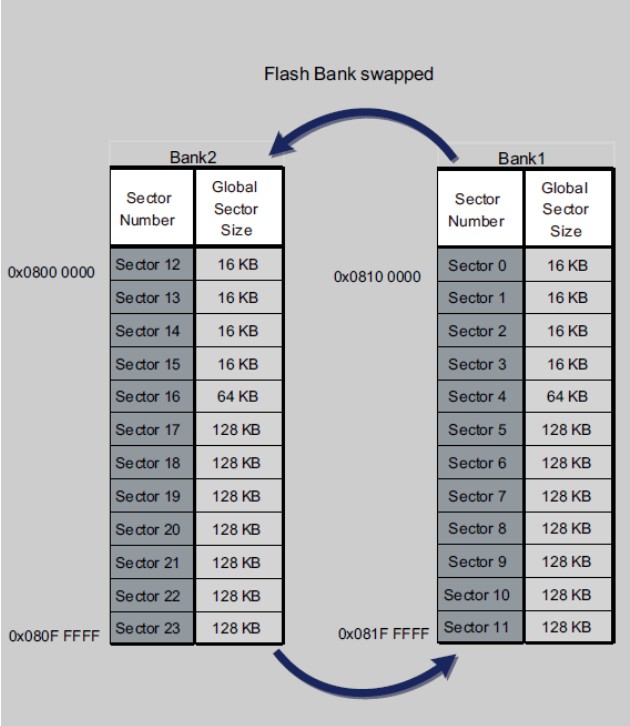
CycloneBOOT can address two different operating modes. The first one is “Dual-Bank” mode and the second is “Single-Bank” mode. These modes will be described in the following sections.

4.1 Dual-Bank mode

The “Dual-Bank” mode is a modern update approach based on device flash which is divided equally in 2 partitions or “banks”. These two distinct zones make it possible to:

- Execute a code in one bank while another bank is being erased or programmed.
- Save the CPU from being stalled during programming operations and protect the system from power failures or other errors.

Most of all, dual bank flash memory allows dual boot capability, either by booting from the first flash bank or from the second flash bank. It is done through swapping the two flash banks.



Flash bank swap on a STM32 with 2MB dual bank flash

- Finally, it is not necessary to configure the linker to change the start address or change the interrupt vector table location of the application.

In this mode CycloneBOOT does not use traditional static bootloader, it only requires a piece of code in the application that will operate the update process, allowing the application to update itself.

This piece of code located in the application is called IAP which stands for **In Application Programming**.

When a new update is received, CycloneBOOT **IAP** will:

- First, process update in data blocks, checking it and write the new firmware binary in the second bank of the flash that is not used.
- Second, swap the two device flash banks.
- Finally, boot with the new firmware application.

Above is a simplified explanation of the update process done by CycloneBOOT in Dual-Bank mode. The update data that is received is not a simple firmware binary but an update image with a specific format. This image format allows to manage integrity, authentication, or signature security verification. In addition, the firmware binary inside the image could be encrypted and then decrypted on the target making updates secure.

For more information on the update image and the update process please have a look to the sections *Application Image* and *Processing*.

4.2 Single-Bank mode

CycloneBOOT “single-bank” mode addresses devices with internal flash that doesn’t support “dual-bank” functionality. This means that the flash is seen as single block or “single-bank” flash. In this configuration, the application (with CycloneBOOT IAP) could not update itself directly as it would mean trying to write the new firmware on itself.

To solve this issue, CycloneBOOT will make use of three distinct parts.

- A piece of code located in the application: **IAP**.
- A **static bootloader** placed at the beginning of the flash.
- An **external memory**.

Inside the application, **IAP** will oversee the **first update operation**. It will process the receiving update, checking it and saving it in an external memory in a format that the bootloader can understand.

The **static bootloader** on its side will either:

- Boot directly on the current application located in internal flash after the bootloader section.
- Or complete the **second and last update operation**, by extracting the update data in external memory and writing it over the current application firmware in internal flash memory.

Above is a simplified explanation of the update process done by CyloneBOOT in Single-Bank mode. The update received is an image with a specific format for the update data that will be stored in external memory.

The image is stored in the external memory. Users can also choose to encrypt the image stored in external memory, to provide extra security against a malicious third party who could try to read the contents of the external memory.

In this mode the application will not reside in the beginning of the flash anymore. Instead, the bootloader will be placed at the beginning of the flash, followed by the application.

In this mode, it is also necessary to reconfigure the linker file to relocate the interrupt vector table (VTOR) to match the new application start address.

Refer to the sections *Application Image* and *Flash Memory Organization* for more information.

5 Introduction to Bootloaders

What is a bootloader?

Bootloader is a simple standalone application with minimal dependencies, which sits before the firmware application in the beginning of the internal flash.

What is the role of the bootloader?

The key role of the bootloader is to boot up the main firmware image contained in the bootable sector of the internal flash.

Why is it there?

Sometimes, the main firmware image can be faulty or corrupted. In such a case, the bootloader will be able to "swap" or restore the previous known working firmware version. This is especially useful in devices where no hardware assisted firmware swapping is available (I.e., dual bank devices in STM32 product lines).

What is in there?

CycloneBOOT minimal bootloader contains the following dependencies:

- *CycloneCRYPTO*: our Crypto library to manage encrypted images, calculate image authentication, signature, and verification data. Only the necessary algorithms are included so the overall bootloader size remains small.

Minimal dependencies ensure that it occupies at most 2 sectors in the internal flash.

Please refer to "Getting Started" chapter for more information about software requirements and dependencies.

5.1 How CycloneBOOT performs firmware updates

CycloneBOOT offers several safety features such as the ability to restore the last known working firmware in case of a faulty update or the ability to stop a device from being rolled-back to a known faulty firmware version. Please refer to the section "Features" for a comprehensive list of CycloneBOOT features and learn how to enable these in your application.

How is the update image prepared / fetched?

Use the provided ApplicationImageBuilder desktop utility to generate an update image for a given firmware file. This utility will generate a firmware image that is recognizable to CycloneBOOT. It contains several data fields to ensure data integrity and/or authenticity.

Please refer to the section "Application Image" and the document *CycloneBOOT Application ImageBuilder guide* for more information on this topic.

CycloneBOOT is designed to work alongside any protocol that you may employ to send the generated update image to the MCU device. It could be a method based on TCP/IP protocols (please see our demonstration projects for more information) or it could be via USB, Bluetooth, SD card, etc.

Once the application firmware image is fetched, iapUpdate() function will begin processing the image by blocks. An integrity calculation is performed at the same time and used to check the given integrity, authenticity, or signature tag contained in the footer of the firmware application image received.

At the same time, iapUpdate() function writes the received data in to either:

- Secondary bank, if the internal MCU flash is “multi-bank.”
- External memory, if the internal MCU flash contains a single bank.

At the end of this process:

- Firmware binary will be present in the secondary flash bank, ready to be “swapped” in the case of a “dual bank” internal MCU flash.
- A firmware image file will be present in the external memory, in a format that is recognizable to the static bootloader, which will perform the “update” on the binary in the internal MCU flash.

Note: Please refer to the API Guide chapter for a more complete treatment of user space API calls available with CycloneBOOT.

How is the update performed?

For this section, we distinguish the following two configurations: a) internal flash is a single continuous block OR b) internal flash is separated into two flash banks.

In addition, features enabled by the user such as fallback support or anti-rollback support will also dictate certain aspects of the update workflow.

a) MCU internal flash is a single continuous block

In this configuration, CycloneBOOT uses an external flash device to store the processed update image.

When the update image is fetched, CycloneBOOT verifies its authenticity and integrity. If the “anti-rollback” support is enabled by the user, CycloneBOOT will also verify the version of the newer firmware to make sure it is valid. Then, it applies an “index” to the firmware image, to be able to easily distinguish the “freshness” of the received update image. The index is always calculated to be an increment by one to the current running firmware’s index number.

If the fallback support is enabled, CycloneBOOT will look for a free slot in the external memory to save the fetched image. If not, the first available slot in the external memory is used to store the update image.

Please note also that all the firmware images stored in external flash mediums can also be encrypted to add an extra layer of security.

Why do I need to worry about which slot CycloneBOOT uses to save the image?

When the user activates fallback support, this means that CycloneBOOT can restore the last known working version of the firmware back in the MCU. The backup of this last known working version of the firmware is stored in the external memory. So, it is important to not to overwrite this slot when an update image is also stored in the external flash.

Once the application firmware image is stored, during the next reset routine CycloneBOOT static bootloader will perform a check of the internal and external memory. It will then detect a “fresh” update is available in the external memory, ready to be installed.

The “freshness” of the image is determined by the bootloader by comparing the “index” of the firmware currently in the main MCU internal flash with the index of the external firmware image file in each slot. If the latter’s index is higher, it means there is a newer firmware image file available.

Then the bootloader verifies the image, decrypts it and writes it in the internal flash and performs a MCU reset.

How can the static bootloader decrypt an encrypted image?

The answer to this question depends on the configuration options chosen by the user. If for example, user has enabled “fallback support” then, the bootloader also contains the same key used to encrypt the firmware file in the first place. If no such support exists, the bootloader still must be able to decipher the new firmware update image. In such instances, the encryption key is made available to the bootloader via a pre-defined zone in the RAM. At each start-up, the static bootloader will then check this zone in RAM to determine if there is a key available to decrypt any ready-to-install update images.

b) MCU internal flash is separated into banks

In MCUs where the internal flash is divided into distinct flash banks then CycloneBOOT uses hardware support available to perform a “swap” between each bank. In such cases, the primary bank contains the current firmware, and the secondary bank will be used to save the latest version of the firmware.

When the MCU internal flash is said to be “dual bank” (for example in STM32, Nordic, Atmel and EFM32 MCUs), iapUpdate() and iapFinalize() functions will write (after decrypting, if the received firmware image is encrypted) the latest firmware image to the secondary bank. Then CycloneBOOT initiates a “swap,” defining which bank will be used to boot up by the MCU (in STM32 MCUs, this is done by setting up the correct Option Bytes). During the next reset routine, the MCU will then boot from the bank containing the latest firmware version. If for some reason the user needs to fall back to the last known working firmware, it is now located in the secondary bank and another “swap” will make that bank the main bootable partition.

What happens when firmware is not valid?

It is up to the user to implement the logic necessary to classify a firmware image as non-valid. In such a case there are several possibilities to restore the last known working version of the firmware. Again, it depends on the internal flash configuration of the MCU as well as the options chosen by the user when CycloneBOOT is configured.

When the MCU internal flash has multiple banks, and the fallback support is enabled:

- A “swap” is performed, meaning the main flash bank (containing the non-valid firmware) becomes a secondary partition and the current secondary bank is set to become the primary partition (thus the bootable partition) at next reset sequence.

When the MCU internal flash does not contain any banks, and the fallback support is enabled:

- The bootloader is notified of this fact through a flag communicated to it via the RAM. Upon seeing this flag, the bootloader enters “fallback mode.” It will then scan slots in external flash memory device to find the image with an index number one less than the current firmware image. This is the last known working version of the firmware. The bootloader will fetch the firmware image from the external memory slot and write it to the internal flash, decrypting it in the process if the firmware image is encrypted.

What happens if my firmware is not valid, and I have not activated fallback support?

Depending upon the severity of the firmware issue, this could lead to a non-operative device. It is important therefore to make sure the firmware is thoroughly checked before being pushed out as an update. Of course, enabling fallback support will make sure that the application can revert to a last known working state in case of a serious error.

6 Features

CycloneBOOT is a firmware update library dedicated to embedded applications that uses modern In Application Programming (IAP) approach to manage firmware updates. It supports MCUs with “dual-bank” flash capabilities allowing to update a firmware without using a bootloader. Otherwise, a combination of a traditional bootloader and IAP will be used to support devices with a classic “single-bank” flash and an external memory. Being completely agnostic to the method of obtaining the firmware update, CycloneBOOT can be integrated alongside any protocols.

CycloneBOOT IAP supports “anti-rollback” feature, which is a guard against unintentional or malicious “downgrading” of device firmware to a known invalid or buggy firmware version. In addition, CycloneBOOT IAP supports “fallback” feature, when enabled make the MCU device revert to the last known firmware version, in case of a problem with the new firmware version.

It also supports External flash memory through an API layer, making it easy to work with a wide variety of different external flash memory devices.

6.1 Main Features

- Secure bootloader for 32-bit MCUs
- Can be integrated in client or server operation.
- Can run alongside a RTOS or in bare metal.
- Uses a dedicated firmware image format.
- Provides In-Application Programming (IAP) support.
- Single[1] and Dual-bank flash support.
- Support for encrypted firmware (AES-CBC).
- Signature (ECDSA/RSA), Authenticity (HMAC-MD5/SHA256/SHA512), or integrity verification (MD5, SHA1, SHA256, SHA512) of the firmware.
- Fallback support: Backup current firmware and restore it if required.
- Anti-rollback support: Prevent rolling-back to a known faulty firmware version.
- PC utility (Windows and Linux) to build an application image (can encrypt the firmware and compute an integrity tag, an authentication tag, or a signature).

6.2 Supported TCP/IP Protocols

CycloneBOOT is protocol agnostic and therefore can be easily bundled with CycloneTCP, CycloneSSL & CycloneSSH

- TFTP / FTP / FTPS
- HTTP / HTTPS
- MQTT / MQTTS
- SFTP / SCP

And many others.

[1] with an external memory device

6.3 Fallback feature


This feature is only available on CycloneBOOT Ultimate version.

1 - Why would I need it?

Sometime a firmware update can bring several unwanted issues or bugs introducing instabilities. Thanks to its fallback feature, the CycloneBOOT library gives a way to go back to the previous firmware that should be more stable.

2 - How does it work?

It will depend on which mode you are using the CycloneBOOT library with:

In Dual-Bank mode (directly from the application side):

We use the dual-bank flash facilities to store firmware application binary in the device internal flash (within flash memory subdivisions). This operation is done without a bootloader directly from the application.

When an update occurs, the new firmware is stored in the available bank (B2) while the current firmware is still running in the other bank (B1). Then after a swap of the flash banks, the device runs the new application firmware from the bank (B2) and the remaining bank (B1) becomes the available bank for a further update. But the previous firmware update is still present in (B1) which makes it possible to go back to it.

Following are the fallback steps in dual-bank mode:

1. Performs a swap on the flash banks to make the previous firmware bootable.
2. Trigger a device reset to boot to the previous firmware application

In Single-Bank mode (from static bootloader side):

The application cannot directly update itself as it would try to write the new firmware on itself. Therefore, a static bootloader is needed to complete the update process. That is why we make use of an external memory to store the updated firmware application as a first step. Then as a second step (after a reset), a static bootloader writes the new application from the external memory over the previous firmware located in the internal flash. When using the fallback feature two slots (distinct memory area) in external memory are needed.

When an update occurs, the new firmware (F2) is stored in the available slot (S2) while the other slot (S1) stores a backup of the current running application (F1bis). Then after a reset, the bootloader detects a new firmware (F2) in slot (S2) and writes it over the current application (F1) in internal memory. But the backup of the previous firmware (F1bis) is still present in external memory (slot S1) which makes it possible to revert to it if necessary.

Following are the fallback steps in single-bank mode:

1. Retrieve a backup image of the previous firmware from the related external memory slot and write it over into the current faulty application in the internal flash memory.
2. Delete from external memory the slot containing the bad firmware.

3. Reset the device to boot on the restored previous firmware.

3 - How to use it?

If running the CycloneBOOT library in dual-bank mode:

You will have to call a dedicated API in your application to launch the fallback process: `iapFallbackStart()`. Refer to section *iapFallbackStart()* for more information.

This function MUST be called as soon as possible in the main routine of your application when a fallback is requested. A fallback request represents a manual operation. For example, a "button press" which would change a specific device state pin.

The fallback request logic MUST be implemented by yourself.

Here is an example of a fallback procedure:

1. Press the reset button or powering OFF then ON the device
2. While pressing the reset button or before powering ON the device, press a user button (fallback button, could also be a Jumper to put in place)
3. Release the reset button or powering ON the device while keeping the user button pressed
4. Your request logic detects that the user button is pressed and waits for button release before calling the dedicated fallback API.
5. The fallback process is complete and the device restarts on the previous firmware application.

If running the CycloneBOOT library in single-bank mode:

The fallback process is done on the bootloader side. At startup, the bootloader needs to be informed that a fallback is requested to enter fallback mode. For this purpose, you will be invited to implement two user callbacks:

- `bootFallbackTriggerInit()`: initializes hardware/software related to the fallback trigger event (request).
- `bootFallbackGetTriggerState()`: returns the fallback trigger state (raised or not).

Refer to section Fallback Bootloader routines for more information.

Here is an example of fallback procedure:

1. Press the reset button or powering OFF then ON the device to start the bootloader.
2. While pressing the reset button or before powering ON the device, press a user button (fallback button, could also be a Jumper to put in place)
3. Release the reset button or powering ON the device while pressing the user button
4. The bootloader, thanks to the user callbacks, detects that the user button is pressed and waits for the button to release before entering fallback mode.
5. The fallback process is complete and the device restarts on the previous firmware application.

In any case you will have to activate the fallback support feature in CycloneBOOT:

- IAP configuration file: "iap_config.h" (valid in both modes)
- Static bootloader configuration file: "boot_config.h" (only in single-bank mode)

Refer to sections *IAP Configuration* and *Bootloader configuration* for more information.

6.4 Anti-rollback feature

1 - Why would I need it?

A firmware update can be performed for several reasons, among them: adding new functionalities, fixing bugs, or correcting security issues. The third reason could also be an attack vector used by a malicious actor. If the attackers can access firmware update infrastructure, they could try to downgrade the firmware by providing a firmware update with a previous version that contains known issues. To avoid rolling back to a previous firmware version CycloneBOOT provides an “anti-rollback” feature.

2 - How does it work?

CycloneBOOT library does not process a simple “firmware binary” file as update. Instead, it expects to process a specific update image file (refers to section *Application Image* for more information).

This image file is composed of these 3 sections: header, firmware binary and footer. In particular, the image header section will contain metadata such as version of the firmware binary inside the update image.

When the anti-rollback feature is activated, CycloneBOOT retrieves the firmware version from the update image and compares it against the version of the current running firmware application. If the retrieved version is “less than” or “equal to” the current application version then the update image will be rejected, and the update will be canceled. This way you can prevent any potentially malicious downgrading of your firmware application.

3 - How to use it?

You MUST provide the CycloneBOOT IAP code resulting in your application, the version of your current application. Refer to section *iapInit()* for more information.

In any case you will have to activate the support of the anti-rollback feature in CycloneBOOT:

- IAP configuration file: “iap_config.h” (valid in both modes)
- Static bootloader configuration file: “boot_config.h” (only in single-bank mode)

Refer to sections *IAP Configuration* and *Bootloader configuration* for more information.

6.5 Limitations

- User must make sure that the “Fallback” feature in Dual Bank mode is activated in the beginning of the firmware application. This is to make sure that for some reason the application firmware is seriously faulty and later encounters an error making it inoperable. By activating the fallback feature early on, even if the application subsequently becomes non-responsive, there will be an opportunity to make a corrective action. However, developers must make sure to verify and test the firmware for serious errors before sending the update image.

- The fallback feature cannot be activated remotely (there must be a physical intervention (e.g., pressing a button) to enable the feature.

7 Requirements

7.1 Software

- GNU GCC / Makefile
- Atollic TrueSTUDIO
- IAR Embedded Workbench
- Keil MDK-ARM
- SEGGER Embedded Studio
- AC6 System Workbench for STM32 (SW4STM32)
- ST STM32CubeIDE

7.2 Hardware

- STM32L4
- STM32F4
- STM32F7
- STM32H7

8 Getting Started

8.1 Requirements

- Provide a minimal project that compiles C/C++ project with
  - HAL (Hardware Abstraction Layer) implementation
  - Downloader or Uploader file implementation (TCP, UART, etc...)
  - Update trigger logic (event, request, action, etc.)

- **Add the following Macro to your project's preprocessor settings: CYCLONE_BOOT_IAP**

Note: Please provide the macro only if you are compiling CycloneBOOT IAP and not the static bootloader.

8.2 CycloneCRYPTO Configurations

CycloneBOOT uses a subset of CycloneCRYPTO for cryptographic primitives and algorithms. "crypto_config.h" host CycloneCRYPTO related configurations. Depending on your chosen update scenario, some cryptographic function support must be set (enabled or disabled) in a configuration file. The configuration file is named 'crypto_config.h' and must be located among your application files.

Enabling relevant features and disabling others will produce a more streamlined binary.

Fore more information about different "scenarios", please refer to the Section 10.2.

Common to all scenarios

```
//DES encryption support
#define DES_SUPPORT DISABLED
//Triple DES encryption support
#define DES3_SUPPORT DISABLED
//Camellia encryption support
#define CAMELLIA_SUPPORT DISABLED
//ARIA encryption support
#define ARIA_SUPPORT DISABLED
```

Scenario #2 – Encryption and authentication


This feature is only available on CycloneBOOT Pro and CycloneBOOT Ultimate versions.

```
//HMAC support
#define HMAC_SUPPORT ENABLED
//AES encryption support
#define AES_SUPPORT ENABLED
//CBC mode support
#define CBC_SUPPORT ENABLED
```

Scenario #3 – Encryption and signature


This feature is only available on CycloneBOOT Ultimate version.

```
//AES encryption support
#define AES_SUPPORT ENABLED
//CBC mode support
#define CBC_SUPPORT ENABLED
//RSA support
#define RSA_SUPPORT ENABLED
//Elliptic curve cryptography support
#define EC_SUPPORT ENABLED
//ECDSA support
#define ECDSA_SUPPORT ENABLED
```

8.3 Setup CycloneBOOT in Dual-Bank mode

In this section you will learn how to setup CycloneBOOT in Dual-Bank mode in your application. This will demonstrate a simple use case in which a simple update image will be managed (no encryption with just a CRC32 integrity checksum).

8.3.1 Sources files

To add CycloneBOOT in your application, you first need to add the CycloneBOOT source files. As we want to use CycloneBOOT in Dual-Bank mode we will only use the IAP-related sources files.

Add the following files into your project:

Common CycloneBOOT and CycloneCRYPTO files:

- common/error.h
- common/debug.h
- common/ cpu_endian (c/h)
- common/os_port.h
- common/os_port_<os> (c/h) (RTOS or bare metal files port)

CycloneCRYPTO files:

- cyclone_crypto/core/crypto.h (depending the usage you may need to add other files)

CycloneBOOT files:

- cyclone_boot/core/crc32.(c.h)
- cyclone_boot/core/image (c/h)
- cyclone_boot/core/slot (c/h)
- cyclone_boot/core/flash.h
- cyclone_boot/core/mcu.h

- cyclone_boot/core/cboot_error.h
- cyclone_boot/iap/iap (c/h)
- cyclone_boot/iap/iap_misc (c/h)
- cyclone_boot/iap/iap_process (c/h)
- cyclone_boot/security/verify (c/h)

  According to the device MCU, choose the right drivers:

  - cyclone_boot/drivers/mcu/<arch>/xxx_mcu_driver (c/h)
  - cyclone_boot/drivers/flash/internal/xxx_flash_driver (c/h)

Note: In case you do not find the driver adapted to your internal device flash memory in your downloaded software package, please contact us.

Configuration and user files:

- os_port_config.h (Common os port configuration file)
- iap_config.h (CycloneBOOT IAP configuration file)
- crypto_config.h (CycloneCRYPTO configuration file)
- debug.c (Trace log redirection interface)

Setup the following path into your project ("include path"):

- <path-to-lib>/common
- <path-to-lib>/cyclone_crypto
- <path-to-lib>/cyclone_boot
- <path-to-config-files>

8.3.2 Configuration User Files

The configuration files cited above need to be setup correctly to be able to use CycloneBOOT.

1. The file "iap_config.h" manages CycloneBOOT IAP related configurations. In our case we will configure CycloneBOOT to support Dual-Bank mode and update images with simple integrity check. Refer to section IAP Configuration for more information.

Please add the following code in "iap_config.h":

```
//CycloneBOOT IAP Dual-Bank mode support
#define IAP_DUAL_BANK_SUPPORT ENABLED
//CycloneBOOT IAP integrity check support
#define VERIFY_INTEGRITY_SUPPORT    ENABLED
```

CycloneBOOT uses a subset of CycloneCRYPTO for cryptographic primitives and algorithms. "crypto_config.h" host CycloneCRYPTO related configurations. However, we will use a custom CRC32 implementation as integrity check algorithm. It is not part of official CycloneCRYPTO library so you do not have to add anything in this case. You can simply use the common configuration. (Refer to 8.2 CycloneCRYPTO Configurations)

Note: You can choose to use a well-known hash algorithm as "sha256". For this purpose, you would have to:
- Add the related sources files to your project: cyclone_crypto/hash/sha256 (c/h)
- Add support for this algorithm in crypto_config.h: #define SHA256_SUPPORT ENABLED

3. CycloneCRYPTO library makes use of generic RTOS API functions, especially the ones responsible of allocating memory. The file "os_port_config.h" configures the user chosen RTOS. In any case it will link to a dedicated file "common/os_port_<os>.h". In our case we will run FreeRTOS.

Please add the following code in "os_port_config.h":

```
#define USE_FREERTOS
```

Note: You could choose to use another supported RTOS or no RTOS at all (bare metal). Have a look at the file "common/os_port.h" to see a list of supported RTOS.

4. CycloneBOOT and CycloneCRYPTO libraries employ a trace log facility which relies on custom flow redirection (stdout, stderr). This redirection could be on a dedicated USART or on the ITM, etc. This choice is made in the user file "debug.c". You can refer to one of the demos provided in the CycloneBOOT package for examples.

8.3.3 Linker file

Since we use CycloneBOOT in dual bank mode it means that the internal flash of the device is configured to be divided into 2 banks of the same size. You MUST update the linker file of your project to make sure that the code size in flash do not exceed the size of one bank (i.e., half of total flash size). Also, the flash code section MUST start at the beginning of the flash bank 1 address. Please refer to the section *Flash Memory* more information.

8.3.4 Option Bytes

Before compiling and flashing the target, please make sure that the appropriate "Option Bytes" or equivalent register values are set. Doing so will ***enable*** Dual Bank functionality in supported MCUs.

For example, in order to configure a STM32F7 MCU (2 Mbytes of flash) in Dual-Bank mode you need to set following options bytes with the right values:

- *nDBANK = 0* (Flash is set in dual bank one 1 Mbytes each)
- *nDBOOT = 0* (STM32 dual boot is enabled)

- *BOOT_ADD0 = 0x2000* (start address of the first bank 0x08000000)
- *BOOT_ADD1 = 0x2040* (start address of the second bank 0x08100000)

Images below depicts the appropriate Option Byte fields (*nDBank, nDBOOT, BOOT_ADD0, BOOT_ADD1*) for a STM32F7 Dual Bank MCU device with 1 Mbyte of flash bank each (using the STM32CubeProgrammer utility).

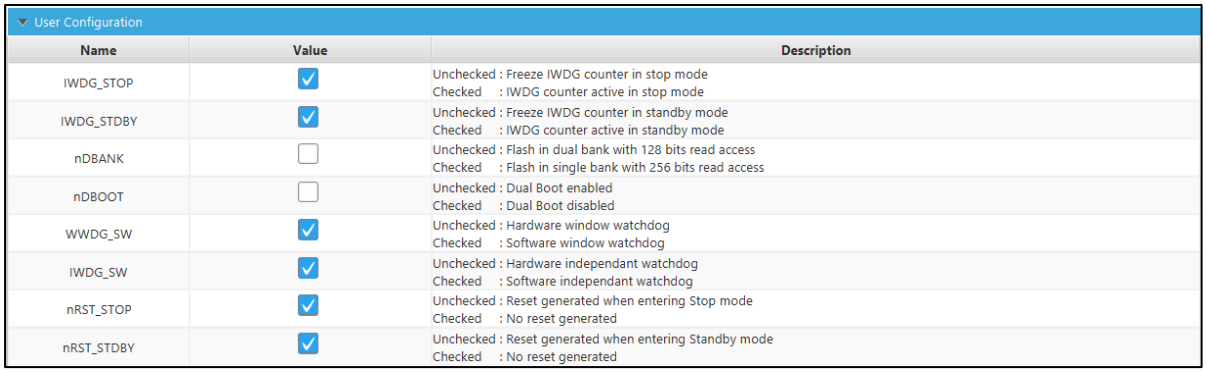


Figure 1: Enabling dual bank support

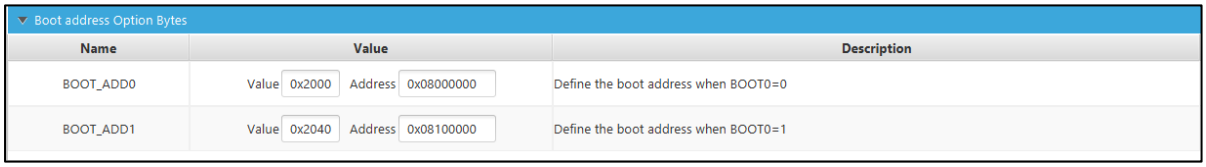


Figure 2: Selecting the appropriate boot address

Note:

- Refer to the device MCU datasheet/manual for detailed information on the option bytes or registers configuration regarding dual-bank mode.
- Please contact our support team if needed.

8.3.5 User code

For this step, it is assumed that you already have a download routine to retrieve an update image using your preferred medium. To process the update image, you will have to use the following functions:

- iapGetDefaultSettings(IapSettings *settings): Get default IAP settings
- iapInit(IapContext* context, IapSettings *settings): Initialize CycloneBOOT IAP
- iapUpdate(IapContext* context, uint8_t*data, size_t dataLen): Process update image
- iapFinalize(IapContext* context): Finalize update procedure
- iapReboot(IapContext* context): Reboot the device target

First, make sure to properly configure the VTOR of your application to match the flash bank 1 start address (default flash start address). For example, with STM32 device you can setup the VTOR inside the function *SystemInit()* in the *system_stm32xxxx.c* file.

1. Declare the CycloneBOOT IAP context and user settings structure.

```
IapContext iapContext;
IapSettings iapSettings;
```

2. Get default settings and setup the user settings to indicate:
   a. The type of image to manage, in our case a non-encrypted image with CRC32 integrity check
   b. The device internal flash memory driver

```
//Get default IAP settings
iapGetDefaultSettings(&iapSettings)
//Set CRC32 as integrity check method
iapSettings.imageInCrypto.verifySettings.integrityAlgo = CRC32_HASH_ALGO;
//Specify device internal flash memory driver (for ex: stm32f7xx device)
iapSettings.primaryMemoryDriver = &stm32f7xxFlashDriver;
```

Note:
- The device internal flash driver relies on the chosen driver file within *cyclone_boot/drivers/flash/internal* directory.
- For more information on the CycloneBOOT IAP settings structure, please refer to the section “IAP User Settings”.

3. Before starting the update image download, initialize CycloneBOOT IAP context:

```
error = iapInit(&iapContext, &iapSettings);
```

4. Process each received update image block by calling *iapUpdate(…)* as bellow:

```
//Update image download routine
while(conditions)
{
   error = userReceiveUpdateImageBlock(&block, &blockLen);
   if(error) break;

   cboot_error = iapUpdate(&iapContext, block, blockLen);
   if(error) break;
}
```

5. Once the final image block processed and if no error occurs until now, finalize the update:

```
cboot_error = iapFinalize(&iapContext);
```

6. Finally reboot the device:

```
cboot_error = iapReboot(&iapContext);
```

Note: Refer to the section *IAP User API* for more information on the CycloneBOOT IAP APIs.

8.4 Setup CycloneBOOT in Single-Bank mode

In this section you will learn how to setup CycloneBOOT in Single-Bank mode in your application. This will demonstrate a simple use case in which simple update image will be processed (plain-text update image with CRC32 integrity checksum).

8.4.1 Sources files

To add CycloneBOOT in your application, first you need to add the CycloneBOOT source files. As we want to use CycloneBOOT in Dual-Bank mode we will only use the IAP-related source files.

Add the following files into your project:

Common CycloneBOOT and CycloneCRYPTO files:

- common/error.h
- common/debug.h
- common/ cpu_endian (c/h)
- common/os_port.h
- common/os_port_<os> (c/h) (RTOS or bare metal files port)

CycloneCRYPTO files:

- cyclone_crypto/core/crypto.h (depending the usage you may need to add other files)
- cyclone_crypto/cipher/aes (c/h)
- cyclone_crypto/cipher_modes/cbc (c/h)

CycloneBOOT files:

- cyclone_boot/core/crc32.(c.h)
- cyclone_boot/core/image (c/h)
- cyclone_boot/core/slot (c/h)
- cyclone_boot/core/flash.h
- cyclone_boot/core/mcu.h
- cyclone_boot/core/cboot_error.h
- cyclone_boot/iap/iap (c/h)
- cyclone_boot/iap/iap_misc (c/h)
- cyclone_boot/iap/iap_process (c/h)
- cyclone_boot/security/verify (c/h)
- cyclone_boot/security/cipher (c/h)

  According to the device MCU, choose the right drivers:

  - cyclone_boot/driver/mcu/<arch>/xxx_mcu_driver (c/h)
  - cyclone_boot/driver/flash/internal/xxx_flash_driver (c/h) -> internal flash driver
  - cyclone_boot/driver/flash/external/yyy_flash_driver (c/h) -> external memory driver

Note: In case you do not find the driver adapted to your internal device flash or external memory in the downloaded package, please contact us.

Configuration and user files:

- os_port_config.h (Common os port configuration file)
- iap_config.h (CycloneBOOT IAP configuration file)
- crypto_config.h (CycloneCRYPTO configuration file)
- debug.c (Trace log redirection interface)

Setup the following path into your project (“include path”):

- <path-to-lib>/common
- <path-to-lib>/cyclone_crypto
- <path-to-lib>/cyclone_boot
- <path-to-config-files>

8.4.2 Configuration User Files

The configuration files cited above need to be setup correctly to be able to use CycloneBOOT.

2. The file “iap_config.h” managed CycloneBOOT IAP related configurations. In our case we will configure CycloneBOOT to support Single-Bank mode and update images with simple integrity check. Refer to section *IAP Configuration* for more information.

Please add the following code in “iap_config.h”:

```
//CycloneBOOT IAP Single-Bank mode support
#define IAP_SINGLE_BANK_SUPPORT ENABLED
//CycloneBOOT IAP Dual-Bank mode support
#define IAP_DUAL_BANK_SUPPORT DISABLED
//CycloneBOOT IAP integrity check support
#define VERIFY_INTEGRITY_SUPPORT    ENABLED
//CycloneBOOT IAP encryption support
#define IAP_ENCRYPTION_SUPPORT      ENABLED
//Bootloader size : addition of sector(s) size occupied by the bootloader
#define BOOTLOADER_SIZE 65536 //64KB -> 2 first sectors
```

3. CycloneBOOT uses a subset of CycloneCRYPTO as cryptographic library. We will use a custom CRC32 implementation as integrity check algorithm. In addition, we will need to add support for the AES-CBC algorithm to manage the encryption of the image stored in external memory.

You can specify a file as below: (Refer to section *IAP Configuration* for more information.)

```
#ifndef _CRYPTO_CONFIG_H
#define _CRYPTO_CONFIG_H

//Desired trace level (for debugging purposes)
#define CRYPTO_TRACE_LEVEL TRACE_LEVEL_INFO
//AES support
#define AES_SUPPORT ENABLED
//CBC mode support
#define CBC_SUPPORT ENABLED

#endif //!_CRYPTO_CONFIG_H
```

Note: You can choose to use a well-known hash algorithm as "sha256". For this purpose, you would have to:
- Add the related source files to your project: cyclone_crypto/hash/sha256 (c/h)
- Add support for this algorithm in crypto_config.h: #define SHA256_SUPPORT ENABLED

5. CycloneCRYPTO library makes use of generic RTOS API functions, especially the one responsible for allocating memory. The file "os_port_config.h" serves user RTOS choice. In any case it will link to a dedicated file "common/os_port_<os>.h". In our case we will run FreeRTOS.

Please add the following code in "os_port_config.h":

```
#define USE_FREERTOS
```

Notes: You could choose to use another supported RTOS or no RTOS at all (bare metal). Refer to the file "common/os_port.h" for a list of the supported RTOS.

6. CycloneBOOT and CycloneCRYPTO libraries employ a trace log facility which relies on custom flow redirection (stdout, stderr). This redirection could be on a dedicated USART or on the ITM, etc. This choice is made in the user file "debug.c". You can refer to one of the demos provided in the CycloneBOOT package as an example.

8.4.3 Linker file

As we use CycloneBOOT in single bank mode it means that the internal flash of the device is seen as a single bank. The static bootloader is placed at the beginning of the flash, followed by the internal image that will contain your application binary. You MUST update the linker file of your project to be sure that:

- The code in flash do not exceed the remaining flash size:

  flash size - bootloader sector(s) size - image header size - padding size - 4bytes CRC32

- The flash code section starts at the right address:

  bootloader sector(s) size + image header size + padding size

Refer to the section *Flash Memory* more information.

8.4.4 Option Bytes

Before compiling and flashing the target, please make sure that the appropriate "Option Bytes" or equivalent register values are set. Doing so will ***disable*** Dual Bank functionality in supported MCUs.

For example, to configure a STM32F7 MCU (2 Mbytes of flash) in Single-Bank mode you need to set following options bytes with the right values:

- *nDBANK = 1* (Flash is set in signle bank of 2 Mbytes)

- *nDBOOT = 1* (STM32 dual boot is disabled)

- *BOOT_ADD0 = 0x2000* (device will start at 0x08000000)
- *BOOT_ADD1 = xxxx* (Not needed)

Images below depicts the appropriate Option Byte fields (*nDBank*, *nDBOOT*, *BOOT_ADD0*, *BOOT_ADD1*) for a STM32F7 Dual-Bank MCU device configured with a single flash bank of 2Mbytes (using the STM32CubeProgrammer utility).

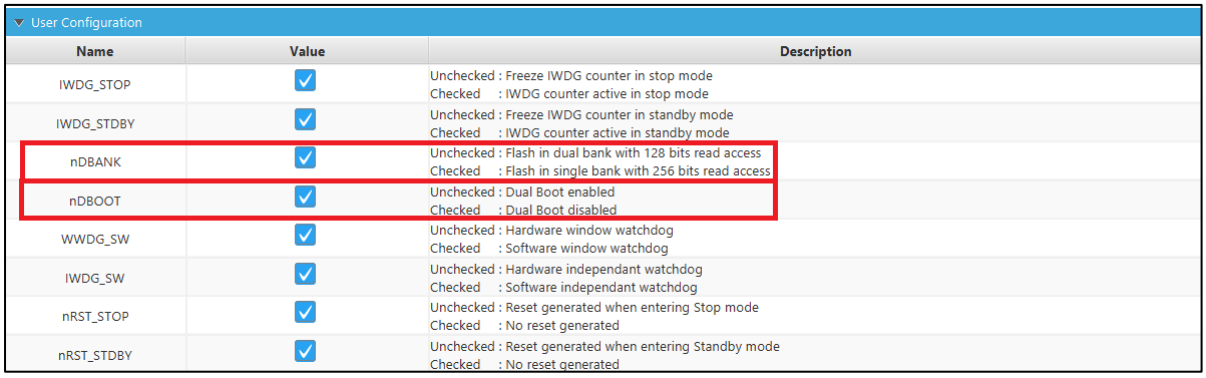


Figure 3: Disabling dual bank support

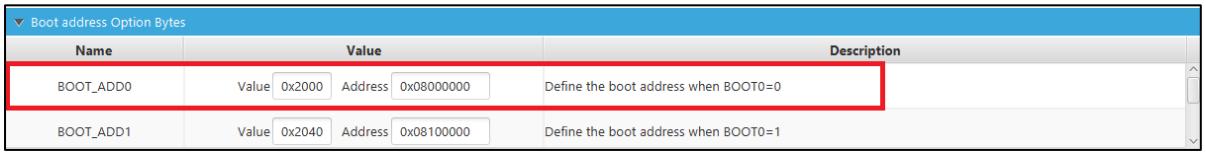


Figure 4: Selecting the appropriate boot address

Note:

- Refer to the device MCU datasheet/manual for detailed information on the option bytes or registers configuration regarding dual-bank mode.
- Please contact our support team if needed.

8.4.5 Preprocessing macros

Some drivers for internal MCU flash memory support both single and dual bank flash mode. It is the case for the following drivers: *stm32f4xx_flash_driver* & *stm32f7xx_flash_driver*.

Per default these drivers are configured for dual bank mode.

To use them in single bank mode you MUST add the “FLASH_SINGLE_BANK” macro in your project preprocessor configuration.

8.4.6 User code

For this step, it is assumed that you already have a download routine to retrieve an update image using your preferred medium. To process the update image, you will have to use the following functions:

- iapGetDefaultSettings(IapSettings *settings): Get default IAP settings
- iapInit(IapContext* context, IapSettings *settings): Initialize CycloneBOOT IAP
- iapUpdate(IapContext* context, uint8_t*data, size_t dataLen): Process update image
- iapFinalize(IapContext* context): Finalize update procedure
- iapReboot(IapContext* context): Reboot the device target

First be sure to properly configure the VTOR of your application to match the address of your application binary inside the internal image (same as flash code linker section address). For example, with STM32 devices you can setup the VTOR inside the function *SystemInit()* in the *system_stm32xxxx.c* file.

1. Declare the CycloneBOOT IAP context and user settings structure.

```
IapContext iapContext;
IapSettings iapSettings;
```

2. Get default settings and setup the user settings to indicate:
    a. The type of image to process, in this case it is a non-encrypted image with CRC32 integrity check
    b. The device internal flash memory driver and the external memory driver
    c. The slot used to store encrypted update images in external memory
    d. The PSK (Pre-shared key) used to encrypt the update image store in external memory

```
//Get default IAP settings
iapGetDefaultSettings(&iapSettings)
//Set CRC32 as integrity check method
iapSettings.imageInCrypto.verifySettings.integrityAlgo = CRC32_HASH_ALGO;

//Specify device internal flash memory driver (for ex: stm32f7xx device)
iapSettings.primaryMemoryDriver = &stm32f7xxFlashDriver;
//Specify device external memory driver (for ex: MX25L512 external flash)
iapSettings.primaryMemoryDriver = &mx25l512FlashDriver;

//Specify slot information (address + size) used to store update images
// in external memory
iapSettings.addrSlot1 = 0x00000000; // ex: address 0 of external memory
iapSettings.sizeSlot1 = 0x100000;   // ex: size 1MB

//Set PSK used for image update encryption in external memory
iapSettings.psk     = "b0e7e85f4732e76eefc3f7e5645107ca";
iapSettings.pskSize = 32;
```

Note:

- Both device internal flash driver and external memory driver relies on the driver file chosen in *cyclone_boot/drivers/flash/internal* and/or cyclone_boot/drivers/flash/external directory.
- For more information on the CycloneBOOT IAP settings structure, please refer to the section *IapSettings*

3. Before downloading the update image, initialize CycloneBOOT IAP context:

```
cboot_error = iapInit(&iapContext, &iapSettings);
```

4. Process each received update image block by calling *iapUpdate(…)* as bellow:

```
//Update image download routine
while(conditions)
{
   error = userReceiveUpdateImageBlock(&block, &blockLen);
   if(error) break;

   cboot_error = iapUpdate(&iapContext, block, blockLen);
   if(error) break;
}
```

5. Once the final image block is processed and if no error has occurred, finalize the update:

```
cboot_error = iapFinalize(&iapContext);
```

6. Finally, reboot the device:

```
cboot_error = iapReboot(&iapContext);
```

Important:

When using CycloneBOOT in single-bank mode it **implies** that:

- **CycloneBOOT IAP API** is used in the user application to manage the received (input) update image and store an output (encrypted or not) image in the external memory
- **A static bootloader** is set up at the start of the internal flash memory to finalize update procedure and / or boot the application.

Follow the next section to learn how to setup the static bootloader.

Note: Have a look at the section *IAP User API* for more information on the CycloneBOOT IAP APIs.

8.5 Setup Static Bootloader (single-bank mode)

In this section you will learn how to set up CycloneBOOT static bootloader located at the beginning of the internal flash. This will demonstrate a simple use case without anti-rollback and fallback features and without external memory image encryption. The bootloader will only manage updates based on what is available in internal flash or external memory. Eventually it will boot to the application.

8.5.1 Sources files

The static bootloader is based on several CycloneBOOT sources. We will only use the bootloader related sources files. Add the following files into your project:

Common CycloneBOOT and CycloneCRYPTO files:

- common/error.h
- common/debug.h
- common/ cpu_endian.h
- common/os_port.h
- common/os_port_<os> (c/h) (RTOS or bare metal files port)

CycloneCRYPTO files:

- cyclone_crypto/core/crypto.h
- cyclone_crypto/cipher/aes (c/h)
- cyclone_crypto/cipher_modes/cbc (c/h)

CycloneBOOT files:

- cyclone_boot/core/crc32.(c.h)
- cyclone_boot/core/image (c/h)
- cyclone_boot/core/slot (c/h)
- cyclone_boot/bootloader/boot (c/h)
- cyclone_boot/bootloader/boot_common (c/h)

  According to the device MCU, choose the right drivers:

  - cyclone_boot/driver/mcu/<arch>/xxx_mcu_driver (c/h)
  - cyclone_boot/driver/flash/internal/xxx_flash_driver (c/h) -> internal flash driver
  - cyclone_boot/driver/flash/external/yyy_flash_driver (c/h) -> external memory driver

Note: In case you do not find the driver adapted to your internal device flash or external memory in the downloaded package, please contact us.

Configuration and user files:

- os_port_config.h (Common os port configuration file)
- boot_config.h (CycloneBOOT IAP configuration file)
- crypto_config.h (CycloneCRYPTO configuration file)
- debug.c (Trace log redirection interface)

Setup the following path into your project ("include path"):

- <path-to-lib>/common
- <path-to-lib>/cyclone_crypto
- <path-to-lib>/cyclone_boot
- <path-to-config-files>

8.5.2 Configuration User Files

The configuration files cited above need to be setup correctly to be able to use CycloneBOOT bootloader.

1. The file "boot_config.h" manages CycloneBOOT bootloader related configurations. By default, the bootloader is configured without fallback, roll-back and external memory image encryption so you can specify a blank file as below. Refer to *Bootloader configuration* for more information.

```
#ifndef _BOOT_CONFIG_H
#define _BOOT_CONFIG_H

//Desired trace level (for debugging purposes)
#define BOOT_TRACE_LEVEL TRACE_LEVEL_INFO

#endif //!_BOOT_CONFIG_H
```

2. CycloneBOOT uses a subset of CycloneCRYPTO for cryptographic primitives and algorithms. "crypto_config.h" hosts CycloneCRYPTO related configurations. However, we will use a custom CRC32 implementation as integrity check algorithm. It is not part of official CycloneCRYPTO library so you can specify a blank file as bellow: (Refer to section XXX for more information.)

```
#ifndef _CRYPTO_CONFIG_H
#define _CRYPTO_CONFIG_H

//Desired trace level (for debugging purposes)
#define CRYPTO_TRACE_LEVEL TRACE_LEVEL_INFO

#endif //!_CRYPTO_CONFIG_H
```

Note: You can choose to use a well-known hash algorithm as "sha256". For this purpose, you would have to:
- Add the related sources files to your project: cyclone_crypto/hash/sha256 (c/h)
- Add support for this algorithm in crypto_config.h: #define SHA256_SUPPORT ENABLED

Note: The static bootloader manages the following crypto algorithm (as of CycloneBOOT version 2.0.0):

- Custom CRC32 integrity check algorithm: verify image header and full image integrity
- AES cipher algorithm in CBC mode: decryption of the image stored in external memory (only if the image in external memory is encrypted)

3. CycloneCRYPTO library makes use of generic RTOS API functions, especially the one responsible for allocating memory. The file “os_port_config.h” serves user RTOS choice. In any case it will link to a dedicated file “common/os_port_<os>. h.” In our case we will run “bare-metal” application.

Please add the following code in “os_port_config.h”:

```
#define USE_NO_RTOS
```

Note: You can choose to use another supported RTOS. Refer to the file “common/os_port.h” to get a list of the supported RTOS.

4. CycloneBOOT and CycloneCRYPTO libraries employ a trace log facility which relies on custom flow redirection (stdout, stderr). This redirection could be on a dedicated USART or on the ITM, etc. This choice is made in the user file “debug.c.” You can refer to one of the demos provided in the CycloneBOOT package as an example.

8.5.3 Linker file

As we use CycloneBOOT in single bank mode. It means that the internal flash of the device is seen as a single bank. The static bootloader is placed at the beginning of the flash, followed by the internal image that will contain your application binary. You MUST update the linker file of the static bootloader project to be sure that:

- The code in flash do not exceed a certain amount of flash: i.e., first 2 sections of the flash
- The flash code section starts at the beginning of the internal flash memory

Refer to the section *Flash Memory* more information.

8.5.4 Preprocessing macros

Some drivers for internal MCU flash memory support both single and dual bank flash mode. It is the case for the following drivers: *stm32f4xx_flash_driver* & *stm32f7xx_flash_driver*.

Per default these drivers are configured for dual bank mode.

To use them in single bank mode you MUST add the “FLASH_SINGLE_BANK” macro in your project preprocessor configuration.

8.5.5 User code

For this step it is assumed that you already have a bare-metal project setup with a main while loop. To make the static bootloader work, you will have to use the following functions:

- bootGetDefaultSettings(BootSettings *settings): Get default bootloader settings
- bootInit(BootContext* context, BootSettings *settings): Initialize CycloneBOOT bootloader
- bootTask(BootContext* context);

First be sure to properly configure the VTOR of your application to match the start address of the internal flash memory. For example, with STM32 device you can setup the VTOR inside the function *SystemInit()* in the *system_stm32xxxx.c* file.

1. Declare the CycloneBOOT bootloader context and user settings structure.

```
BootContext bootContext;
BootSettings bootSettings;
```

2. Get default settings and setup the user settings to indicate:

```
//Get default CycloneBOOT bootloader settings
bootGetDfaultSettings(&bootSettings);

//Specify device internal flash driver (for ex: stm32f7xx device)
bootSettings.prmFlashDrv = &stm32f7xxFlashDriver;
//Specify device external flash driver (for ex: MX25L512 external flash)
bootSettings.sndFlashDrv = &mx25l512FlashDriver;

//Specify slot information (address + size) used to store the internal
// image in internal flash (contains current application firmware)
bootSettings.prmFlashSlotAddr = 0x08010000;
bootSettings.prmFlashSlotSize = 0xF0000;

//Specify slot information (address + size) used to store output update
// images in external flash
bootSettings.sndFlashSlot1Addr = 0x00000000;
bootSettings.sndFlashSlot1Size = 0x80000;
```

Note:

- Both internal flash driver and external memory driver relies on the driver file chosen in *cyclone_boot/drivers/flash/internal* and/or *cyclone_boot/drivers/flash/external* directory.
- For more information on the CycloneBOOT Bootloader settings structure, please refer to the section *BootSettings*.

3. Before starting the bootloader, initialize CycloneBOOT Bootloader context:

```
error = bootInit(&bootContext, &bootSettings);
```

4. Finally, call the bootloader state machine routine in your bare-metal main loop or in an OS task:

```
//Bare-metal main loop or OS task loop
while(1)
{
    ...
    error = bootTask(&bootContext);
    //Is any error?
    if(error)
    {
        //Debug message
        TRACE_ERROR("Bootloader failure!\r\n");
    }
    ...
}
```

Note: Have a look to the section *CycloneBOOT API Guide* for more information on the CycloneBOOT bootloader APIs.

9 CycloneBOOT API Guide

9.1 Bootloader User API

The table below lists the available Bootloader USER related routines within their respective categories. Detailed descriptions of the routines can be found in the sections that follow.

Routine	Description
Usual IAP routines	
bootGetDefaultSettings()	Initialize CycloneBOOT Bootloader settings structure with default values.
bootInit()	Initialize CycloneBOOT IAP context.
Bootloader Task routines	
bootTask()	CycloneBOOT Bootloader main task routine
Fallback Bootloader routines	
fallbackTriggerInit()	Initialize Bootloader fallback trigger (callback)
fallbackTriggerGetStatus()	Get Bootloader fallback trigger status (callback)

9.1.1 Usual Bootloader Routines

9.1.1.1 bootGetDefaultSettings()

Syntax

```
void bootGetDefaultSettings(BootSettings *settings);
```

Parameters

Parameter	Description
settings	Pointer to the CycloneBOOT Bootloader user settings.

Description

The ***bootGetDefaultSettings*** function initializes the CycloneBOOT Bootloader settings with default values.

Example

```
BootSettings settings;

bootGetDefaultSettings(&bootSettings);
bootSettings.prmFlashDrv        = &xxxFlashDriver;
bootSettings.sndFlashDrv        = &yyyFlashDriver;
bootSettings.prmFlashSlotAddr   = 0x08010000;
bootSettings.prmFlashSlotSize   = 0xF0000;
```

```
bootSettings.sndFlashSlot1Addr  = 0x00000000;
bootSettings.sndFlashSlot1Size  = 0x50000;
```

Related topics

Refer to the section *IapSettings* for more details on CycloneBOOT Bootloader user settings.

9.1.1.2 bootInit()

Syntax

```
error_t bootInit(BootContext *context, BootSettings *settings);
```

Parameters

Parameter	Description
`context`	Pointer to the CycloneBOOT Bootloader context.
`Settings`	Pointer to the CycloneBOOT Bootloader user settings.

Return value

Returns an error status code. Can be NO_ERROR (zero) or a non-zero value in case of an error.

Description

The ***bootInit*** function initializes the CycloneBOOT Bootloader context with default values. The given Bootloader context and settings pointer cannot be NULL.

Example

```
Error_t error;
BootSettings settings;
BootContext context;

bootGetDefaultSettings(&bootSettings);
bootSettings.prmFlashDrv        = &xxxFlashDriver;
bootSettings.sndFlashDrv        = &yyyFlashDriver;
bootSettings.prmFlashSlotAddr   = 0x08010000;
bootSettings.prmFlashSlotSize   = 0xF0000;
bootSettings.sndFlashSlot1Addr  = 0x00000000;
bootSettings.sndFlashSlot1Size  = 0x80000;

error = bootInit(&context, &settings);
//Is any error?
If(error)
{
    //Debug message
    TRACE_ERROR("Failed to initialize Bootloader context!\r\n");
    return error;
}
```

9.1.2 Bootloader Task routines

9.1.2.1 bootTask()

Syntax

```
error_t bootTask(BootContext *context);
```

Parameters

Parameter	Description
context	Pointer to the CycloneBOOT Bootloader context.

Return value

Returns an error status code. Can be NO_ERROR (zero) or a non-zero value in case of an error.

Description

The ***bootTask*** function handles the CycloneBOOT Bootloader state machine. The given Bootloader context pointer cannot be NULL. It MUST be called continuously in your bare-metal main loop or in an OS task.

Example

```
Error_t error;
BootSettings settings;
BootContext context;

bootGetDefaultSettings(&bootSettings);
...
error = bootInit(&context, &settings);

...

//Bare-metal main loop or OS task loop
while(1)
{
...
    error = bootTask(&bootContext);
    //Is any error?
    if(error)
    {
        //Debug message
        TRACE_ERROR("Bootloader failure!\r\n");
    }
...
}
```

9.1.3 Fallback Bootloader routines

9.1.3.1 fallbackTriggerInit()

Syntax

```
error_t fallbackTriggerInit(void);
```

Return value

Returns an error status code. Can be NO_ERROR (zero) or a non-zero value in case of an error.

Description

The ***fallbackTriggerInit*** is a callback provided to the user to initialize the fallback trigger. It is a weak function that needs to be redefined in the user space code. The fallback trigger signal will be used to start a fallback procedure at bootloader startup.

Example

```
error_t fallbackTriggerGetStatus(TriggerStatus *status)
{
   // Initialize user button (fallback trigger)
   bspInitButttton(BUTTON_USER);

   //Successful process
   return NO_ERROR;
}
```

9.1.3.2 fallbackTriggerGetStatus()

Syntax

```
error_t fallbackTriggerGetStatus(TriggerStatus *status);
```

Parameters

Parameter	Description
status	Pointer user fallback trigger status. Value can be either: - TRIGGER_STATUS_RAISED - TRIGGER_STATUS_IDLE

Return value

Returns an error status code. Can be NO_ERROR (zero) or a non-zero value in case of an error.

Description

The ***fallbackTriggerGetStatus*** is a callback provided to the user to retrieve the fallback user trigger status. It is a weak function that needs to be redefined in the user space code. At bootloader startup if the fallback trigger is raised then a fallback procedure will be started.

Example

```
error_t fallbackTriggerGetStatus(TriggerStatus *status)
{
   uint32_t state;

   //Get button state
   state = bspGetButtonState(BUTTON_USER);

   if(state == BUTTON_PRESSED)
      *status = TRIGGER_STATUS_RAISED;
   else
      *status = TRIGGER_STATUS_IDLE;

   //Successful process
   return NO_ERROR;
}
```

9.1.4 Data Structures

9.1.4.1 BootSettings

Description

Contains all user CycloneBOOT Bootloader settings

Type definition

```
typedef struct
{
   const FlashDriver *prmFlashDrv;
   const FlashDriver *sndFlashDrv;
   uint32_t prmFlashSlotAddr;
   uint32_t prmFlashSlotSize;
   uint32_t sndFlashSlot1Addr;
   size_t sndFlashSlot1Size;
#if (BOOT_FALLBACK_SUPPORT == ENABLED)
   uint32_t sndFlashSlot2Addr;
   size_t sndFlashSlot2Size;
#if (BOOT_EXT_MEM_ENCRYPTION_SUPPORT == ENABLED)
   const char_t *psk;
   size_t pskSize;
#endif
#endif
} BootSettings;
```

Structure members

Members	Description
Mandatory settings members	
`prmFlashDrv`	Driver of the primary memory (device internal flash).
`sndFlashDrv`	Driver of the secondary memory (external flash).
`prmFlashSlotAddr`	Address of the slot that holds current application image firmware in primary memory.
`prmFlashSlotSize`	Size of the slot in primary memory.
`sndFlashSlot1Addr`	Address of slot 1 that holds an update image in external memory. The update image can be either: - The equivalent of the current application image in primary memory slot - The backup of the previous application image in primary memory slot (only if fallback is activated)
`sndFlashSlot1Size`	Size of slot 1 in external memory.

Fallback settings members only	
`sndFlashSlot2Addr`	Address of the slot 2 that holds an update image in external memory. The update image can be either: - The equivalent of the current application image in primary memory slot - The backup of the previous application image in primary memory slot
`sndFlashSlot2Size`	Size of slot 2 in external memory.
External memory encryption settings members only	
`psk`	Cipher PSK key used to decrypt update image in external slot memory.
`pskSize`	Cipher PSK key size.

Supported values for members	
`prmFlashSlotAddr` `sndFlashSlot1Addr`	32-bits address that match flash sector start address
`psk`	Constant string (16, 24 or 32 bytes long)
`prmFlashDrv` `sndFlashDrv`	Internal or external flash memory driver. From CycloneBOOT drivers list.

Additional information

Some settings are mandatory for CycloneBOOT Bootloader.

The primary memory (internal flash) slot holds a specific internal image that contains the current application firmware binary and some information needed by the bootloader to check the firmware validity.

The secondary memory (external flash) is used to store output update image in a format the bootloader can handle in case of update. These output update images are different from the received by the application when an update procedure is performed. They share some information and the same firmware binary. The output update images are in slot 1 and are overwritten when another update is received.

The firmware contained in the output update images can be encrypted if needed. In that case when fallback support is not activated, the CycloneBOOT IAP middleware in the current application uses a cipher PSK key to encrypt the firmware and then shares this key with the bootloader through a shared RAM section. This PSK key can vary with subsequent updates.

If you choose to activate the fallback feature within CycloneBOOT Bootloader you will have to provide the information for a second memory slot in external memory.

The second external flash slot allows to alternatively use one of the of the two slots to store the backup image of the previous internal application image. It will be used to perform a fallback procedure and come back to the previous internal application image.

If you chose to encrypt the update images in external memory alongside fallback feature, some additional settings will be required by the bootloader. You MUST provide a PSK key to the bootloader as the PSK used to encrypt/decrypt firmware in output update image cannot

be different. It means that the CycloneBOOT IAP middleware in the current application MUST use the same cipher PSK that the bootloader. If it is not the case the bootloader could not be able to decrypt the firmware of the backup internal application image as the cipher PSK used to encrypt it would be different that the one used by the bootloader.

See also

- Section *Fallback feature*
- Section *Anti-rollback feature*
- Section *Bootloader configuration*

9.2 IAP User API

The table below lists the available IAP USER related routines within their respective categories. Detailed descriptions of the routines can be found in the sections that follow.

Routine	Description
Usual IAP routines	
iapGetDefaultSettings()	Initialize CycloneBOOT IAP settings structure with default values.
iapInit()	Initialize CycloneBOOT IAP context.
iapUpdate()	Process update image blocks.
iapFinalize()	Finalize image verification and update process.
iapReboot()	Reboot the device.

Fallback IAP routines (Dual-Bank mode)	
iapFallbackStart()	Initiate fallback procedure in dual-bank mode.

9.2.1 Usual IAP Routines

9.2.1.1 iapGetDefaultSettings()

Syntax

```
void iapGetDefaultSettings(IapSettings *settings);
```

Parameters

Parameter	Description
settings	Pointer to the CycloneBOOT IAP user settings.

Description

The ***iapGetDefaultSettings*** function initializes the CycloneBOOT IAP settings with default values.

Example

```
IapSettings settings;

iapGetDefaultSettings(&iapSettings);
iapSettings.imageInCrypto.verifySettings.integrityAlgo       =
CRC32_HASH_ALGO;
iapSettings.primaryMemoryDriver              = &xxx_flash_driver;
```

Related topics

Refer to the section *IapSettings* for more information on CycloneBOOT IAP user settings.

9.2.1.2 iapInit()

Syntax

```
cboot_error_t iapInit(IapContext *context, const IapSettings *settings);
```

Parameters

Parameter	Description
`context`	Pointer to the CycloneBOOT IAP context to be initialized.
`settings`	Pointer to the CycloneBOOT IAP user settings.

Return value

Returns an error status code. Can be CBOOT_NO_ERROR (zero) or a non-zero value in case of an error.

Description

Initialize CycloneBOOT IAP Application context. The given IAP context and settings pointer cannot be NULL.

Example

```
cboot_error_t error;
IapSettings settings;
IapContext context;

iapGetDefaultSettings(&iapSettings);
iapSettings.imageInCrypto.verifySettings.integrityAlgo       =
CRC32_HASH_ALGO;
iapSettings.primaryMemoryDriver              = &xxx_flash_driver;

error = iapInit(&context, &settings);
//Is any error?
If(error)
{
    //Debug message
    TRACE_ERROR("Failed to initialize IAP context!\r\n");
    return error;
}
```

Related topics

Refer to the section *IapSettings* for more information on CycloneBOOT IAP user settings.

9.2.1.3 iapUpdate()

Syntax

```
cboot_error_t iapUpdate(IapContext *context, const void *data, size_t length);
```

Parameters

Parameter	Description
context	Pointer to the CycloneBOOT IAP context.
data	Pointer to the image data chunk to be processed.
length	Image data chunk length

Return value

The ***iapUpdate*** function returns CBOOT_NO_ERROR if the image data chunk has been successfully processed. Otherwise, a non-zero value status code describing the error is returned.

Description

The ***iapUpdate*** function process updates image block. It will check image header integrity and decide whether to continue image processing or not, based on the user configurations. Each image application binary data will be decrypted if needed, then written in a specified memory area. At the same time, a hash or CRC32 computation will be performed on each image application data. Finally, it will store the image check data tag for further image verification.

Example

```
error_t error;
cboot_error_t cboot_error;
IapSettings iapSettings;
IapContext iapContext;

...
cboot_error = iapInit(&iapContext, &iapSettings);
...
while(!error)
{
    //User routine to get image data block
    error = userGetImageDataBlockRoutine(data, dataSize, dataLen);
    //Is any error?
    if(error) break;

    //Process received image data block
    cboot_error = iapUpdate(&iapContext, data, dataLen);
    //Is any error?
    if(cboot_error) break;
}
```

9.2.1.4 iapFinalize()

Syntax

```
cboot_error_t iapFinalize(IapContext *context);
```

Parameters

Parameter	Description
context	Pointer to the CycloneBOOT IAP context.

Return value

The ***iapFinalize*** function returns CBOOT_NO_ERROR if the update process has been successfully processed. Otherwise, a non-zero value status code describing the error is returned.

Description

The ***iapFinalize*** function finalizes the firmware update. Depending on the user configurations:
- A firmware integrity or authentication or signature validation will be done.
- If CycloneBOOT IAP is configured in Dual-Bank mode, it will swap the flash bank memory
- If CycloneBOOT IAP is configured in Single-Bank mode without fallback feature, it will share the PSK key used to encrypt output image stored in external memory for further processing by the bootloader.

The firmware validation can only be done if all the image (header, firmware data and check has been processed successfully. If not, an error will be raised.

Example

```
cboot_error_t error;
IapSettings iapSettings;
IapContext iapContext;

...
error = iapInit(&iapContext, &iapSettings);
...
while(!error)
{
    ...
    error = iapUpdate(&iapContext, data, dataLen);
    ...
}

error = iapFinalize(&iapContext);
//Is any error?
If(error) return error;
```

Related topics

N/A

9.2.1.5 iapReboot()

Syntax

```
cboot_error_t iapReboot(IapContext *context);
```

Parameters

Parameter	Description
context	Pointer to the CycloneBOOT IAP context.

Return value

The ***iapReboot*** function should not return a value as it results in an immediate device reset. In case of an error, a non-zero value status code describing the error is returned.

Description

The ***iapReboot*** function reboots the device by triggering a system reset.

The reboot operation can only be performed if the image has been fully processed and validated. If not, an error will be raised.

Example

```
cboot_error_t error;
IapSettings iapSettings;
IapContext iapContext;

...
error = iapInit(&iapContext, &iapSettings);
...
while(!error)
{
    ...
    error = iapUpdate(&iapContext, data, dataLen);
    ...
}
error = iapFinalize(&iapContext);
//Is any error?
If(error) return error;

...

error = iapReboot(&iapContext);
//if(error) return error;
```

Related topics

N/A

9.2.2 Fallback IAP routines

9.2.2.1 iapFallbackStart()

Syntax

```
error_t iapFallbackStart(void);
```

Return value

The ***iapFallbackStart*** function will return an error code in case of an error (a non-zero value status code describing the error).

Description

The ***iapFallbackStart*** function performs a fallback procedure that is composed of a flash bank swap followed by a reboot of the device to complete the update process.

It can only be called if you are running an application using CycloneBOOT in dual-bank mode. If you are running CycloneBOOT in single-bank mode, the fallback will be managed by the static bootloader.

The function should be called at the beginning of the main routine in case of a fallback request. It is assumed that your application is functioning nominally at this stage. Otherwise, your application will not reach the main routine and you will not be able to start a fallback procedure.

The fallback request is performed manually. It could be a simple check of a device pin state.

Example

```
#include "iap/iap_fallback.h"

error_t fallbackUserRoutine(void)
{
    error_t error;

    //User fallback trigger initialization (For ex: init GPIO pin)
    fallbackUserTriggerInit();

    //Is user fallback trigger raised? (For ex: GPIO pin is HIGH)
    if(fallbackUserTriggerStatus())
    {
        //Start CycloneBOOT IAP fallback procedure
        error = iapFallbackStart();
    }

    return error;
}

int main(void)
{
    fallbackUserRoutine();
    ...
}
```

Please refer to the section Limitations for more information on using Fallback mode.

9.2.3 Data Structures

9.2.3.1 IapSettings

Description

Contains all user CycloneBOOT IAP settings

Type definition

```
/**
 * @brief IAP user settings
 **/

typedef struct
{
   uint32_t appVersion;                      ///<Version of the running app
   const FlashDriver *primaryFlashDriver;    ///<Driver of the primary \
                                                (internal) flash memory
   IapCryptoSettings imageInCrypto;          ///<Input Image cryptographic\
                                                settings
#if (IAP_SINGLE_BANK_SUPPORT == ENABLED)
   const FlashDriver *secondaryFlashDriver; ///<Driver of the secondary \
                                                (external) flash memory
   IapCryptoSettings imageOutCrypto;         ///<Cryptographic settings \
                                                used to generate the \
                                                update image
   uint16_t addrSlot1;                       ///<Address of the first \
                                                slot in secondary memory
   size_t sizeSlot1;                         ///<Size of the first slot \
                                                in secondary memory
#if (IAP_EXT_MEM_ENCRYPTION_SUPPORT == ENABLED)
   const char_t *psk;                        ///<PSK key used to encrypt \
                                                the output image
   uint32_t pskSize;                         ///<Size of the PSK key
#endif
#if IAP_FALLBACK_SUPPORT == ENABLED
    uint32_t addrSlot2;                      ///<Address of the second \
                                                slot in secondary memory
    size_t sizeSlot2;                        ///<Size of the second slot \
                                                in secondary memory
#endif
#endif
}IapSettings;
```

Structure members

Members	Description
Members of common Dual-bank / Single-Bank settings structures	
`appVersion`	Version of the current user application.
`primaryMemoryDriver`	Driver of the primary memory (device internal flash).
`imageInCrypto`	Cryptographic settings of the input update image.
Single-Bank settings members only	

`secondaryMemoryDriver`	Driver of the secondary memory (external memory)
`imageOutCrypto`	Cryptographic settings of the output update image.
`addrSlot1`	Address of the 1st slot in external memory.
`sizeSlot1`	Size of the 1st slot in external memory.
`addrSlot2`	Address of the 2nd slot in external memory.
`sizeSlot2`	Size of the 2nd slot in external memory.
`psk`	Pre-Shared-Key is used to encrypt output images stored in external memory slot.
`pskSize`	Size of the Pre-Shared-Key.

Supported values for members	
`appVersion`	32-bits value with the following format: MMmmPP - MM (8-bits): major version - mm (8-bits): minor version - PP (8-bits): patch version
`addrSlot1` `addrSlot2`	32-bits address
`psk`	Constant string (16, 24 or 32 bytes long)
`imageInCrypto` `imageOutCrypto`	IapCryptoSettings structure.

Additional information

Some settings are common to CycloneBOOT IAP dual or single bank mode.

The appVersion is needed only if the anti-rollback feature is activated. It serves to indicate the version of the current application and will be used to verify against versions of any future update images. (Comparison with the app version given in the update image).

If you choose to use CycloneBOOT IAP in single bank mode, some additional settings will be available.

The “imageOutCrypto” gives information on how to generate the output image that will be stored in external memory (single-bank mode only). As of version 2.0.0 of CycloneBOOT, only one configuration is supported (AES-CBC encryption + custom CRC32 integrity). You **do not have to,** and you **MUST not** fill this setting.

In single-bank mode you will be asked to specify a memory slot that tells where to store the generated output image in external memory. This slot is described with two components: its address and its size. The address **MUST** match the beginning of an external memory sector.

If you chose to activate the fallback feature within CycloneBOOT in single-bank mode, you will have to provide the information for a second memory slot in external memory

- **See also**
- IapCryptoSettings
- Section sur anti-rollback
- Section sur iap_config.h

9.2.3.2 IapCryptoSettings

Description

Contains all the cryptographic settings dedicated to image processing.

Type definition

```
/**
 * @brief IAP Crypto settings
 **/

typedef struct
{
#if ((IAP_ENCRYPTION_SUPPORT == ENABLED) || \
     ((IAP_SINGLE_BANK_SUPPORT == ENABLED) && \
     (IAP_EXT_MEM_ENCRYPTION_SUPPORT == ENABLED)))
   const CipherAlgo *cipherAlgo;          ///<Image cipher algorithm
   CipherMode cipherMode;                 ///<Image cipher mode
   const char_t *cipherKey;               ///<Image cipher key
   size_t cipherKeyLen;                   ///<Image cipher key size
#endif
   VerifySettings verifySettings;         ///<Various crypto settings for \
                                             image verification
} IapCryptoSettings;
```

Structure members

Members	Description
Members related to image cryptographic encryption/decryption settings	
cipherAlgo	Cipher algorithm used to encrypt/decrypt image binary data.
cipherMode	Cipher mode.
cipherKey	Cipher key used to encrypt/decrypt image binary data.
cipherKeyLen	Cipher key length
Members related to image cryptographic check settings	
verifySettings	Image Check settings

Supported values for members	
cipherAlgo	AES_CIPHER_ALGO
cipherMode	CIPHER_MODE_CBC
cipherKey	Constant string (length depends on choose cipher algo)

Additional information

These cryptographic settings are meant to give the right tools to manage the type of update image to process.

This structure can hold both cipher and check settings. Among the check settings only either integrity or authentication or signature settings will be available. In other words, you cannot specify authentication settings and signature settings at the same time.

You can activate (or deactivate) various cryptographic settings by specifying the corresponding options in the CycloneBOOT IAP configuration file: “iap_config.h”.

For example, if you need to process image with encrypted binary and authentication check you will have to define the macros below:

```
#define VERIFY_AUTHENTICATION_SUPPORT ENABLED
#define IAP_ENCRYPTION_SUPPORT ENABLED
```

9.2.3.3 VerifySettings

Description

Contains all the cryptographic settings dedicated to image verification.

Type definition

```
/**
 * @brief Verification settings
 **/

typedef struct
{
   VerifyMethod verifyMethod;            ///<Image verification method
#if (VERIFY_INTEGRITY_SUPPORT == ENABLED)
   const HashAlgo *integrityAlgo;        ///<Image integrity algorithm
#endif
#if (VERIFY_AUTHENTICATION_SUPPORT == ENABLED)
   VerifyAuthAlgo authAlgo;              ///<Image authentication
algorithm
   const HashAlgo *authHashAlgo;         ///<Image authentication hash \
                                            algorithm
   const char_t *authKey;                ///<Image authentication key
   size_t authKeyLen;                    ///<Image authentication key size
#endif
#if (VERIFY_SIGNATURE_SUPPORT == ENABLED)
   VerifySignAlgo signAlgo;              ///<Image signature algorithm
   const HashAlgo *signHashAlgo;         ///<Image signature hash
algorithm
   const char_t *signKey;                ///<Image signature public key \
                                            (PEM format)
   size_t signKeyLen;                    ///<Image signature public key \
                                            size
#endif
} VerifySettings;
```

Members	Description
Member of cryptographic image verification related structure	
`verifyMethod`	Defines the verification type (choose one): Integrity, authentication, OR signature.
Members related to image integrity check	
`integrityAlgo`	Contains algorithm used to check image integrity.
Members related to image authentication check	
`authAlgo`	Algorithm used to check image authentication.
`authHashAlgo`	Hash algorithm associated to authentication algorithm.
`authKey`	Authentication key.
`authKeyLen`	Authentication key length.
Members related to image signature check	
`signAlgo`	Contains algorithm used to check image signature.
`signHashAlgo`	Hash algorithm associated to signature algorithm.
`signKey`	Signature public key.
`signKeyLen`	Signature public key length.

Supported values for members	
`verifyMethod`	- `VERIFY_METHOD_INTEGRITY` - `VERIFY_METHOD_AUTHENTICATION` - `VERIFY_METHOD_SIGNATURE`
`integrityAlgo`	- `MD5_HASH_ALGO` - `SHA1_HASH_ALGO` - `SHA224_HASH_ALGO` - `SHA256_HASH_ALGO` - `SHA384_HASH_ALGO` - `SHA512_HASH_ALGO` - Or a custom CRC32 “hash” like algorithm: `CRC32_HASH_ALGO`
`authAlgo`	`IAP_AUTH_HMAC`
`authHashAlgo`	Same as integrity algo. (Except `CRC32_HASH_ALGO`)
`authKey`	Constant string
`signAlgo`	- `VERIFY_SIGN_RSA` - `VERIFY_SIGN_ECDSA`
`signHashAlgo`	Same as integrity algo. (Except `CRC32_HASH_ALGO`)
`signKey`	Constant string containing public key in PEM format.

See also

- IapSettings structure.

Note: Following Cryptographic algorithms apply as of CycloneBOOT version 2.0.0 with more algorithms to be added in subsequent releases. If you have a specific algorithm for your use case, please contact us.

- Only AES cipher algorithm in CBC mode is supported
- Only HMAC authentication is supported
- Only RSA and ECDSA signature are supported

9.3 IAP Configuration

The CycloneBOOT IAP can be configured to embed only the necessary features. All configuration parameters are in a dedicated header file (iap_config.h). This file is project specific and can be modified without altering the behavior of other projects. All the compilation flags are preconfigured with default values. However, this default configuration can be changed by overriding the desired compilation flags in the header file.

9.3.1 IAP modes configuration

Configuration Flags	Default Value	Description
`IAP_DUAL_BANK_SUPPORT`	ENABLED	This macro adds or removes support for IAP Dual-bank mode.
`IAP_SINGLE_BANK_SUPPORT`	DISABLED	This macro adds or removes support for IAP Single-Bank mode.

Note: IAP cannot support dual-bank and single-bank modes at the same time. The support of one of these modes MUST be disabled.

9.3.2 IAP image encryption configuration

Configuration Flags	Default Value	Description
`IAP_ENCRYPTION_SUPPORT`	DISABLED	This macro adds or removes support for IAP image encryption.

9.3.3 IAP image check configuration

Configuration Flags	Default Value	Description
Image Integrity check		
VERIFY_INTEGRITY_SUPPORT	ENABLED	This macro adds or removes support for IAP image integrity check.
Image Authentication check		
VERIFY_AUTHENTICATION_SUPPORT	DISABLED	This macro adds or removes support for IAP image authentication check.
Image Signature check		
VERIFY_SIGNATURE_SUPPORT	DISABLED	This macro adds or removes support for IAP image signature check.
VERIFY_RSA_SUPPORT	DISABLED	This macro adds or removes support for RSA algorithm.
VERIFY_ECDSA_SUPPORT	DISABLED	This macro adds or removes support for ECDSA algorithm.

Note: An image is generated with a specific check method. It can be one these methods:

- Integrity check
- Authentication check
- Signature check

You **MUST** add support for the corresponding image check method (see above) by activating the corresponding algorithm in the config file.

9.3.4 IAP feature configuration

Configuration Flags	Default Value	Description
IAP_FALLBACK_SUPPORT	DISABLED	This macro adds or removes support for fallback feature
IAP_ANTI_ROLLBACK_SUPPORT	DISABLED	This macro adds or removes support for anti-rollback feature

9.3.5 Other IAP configurations

Configuration Flags	Default Value	Description
BOOTLOADER_SIZE	65536	Specify bootloader size in bytes.

The bootloader size represents the cumulated size of sector(s) used in flash memory. For example, if the bootloader occupies one sector and half, the resulting bootloader size MUST be the addition of the total size of each sector (not only the partial occupied size of one sector).

9.4 Bootloader configuration

When using CycloneBOOT in single-bank mode a static bootloader MUST be configured. It can be configured to embed only the necessary features. All configuration parameters are in a dedicated header file (boot_config.h). This file is project specific and can be modified without altering the behavior of other projects. All the compilation flags are preconfigured with default values. However, this default configuration can be changed by overriding the desired compilation flags in the header file.

9.4.1 Bootloader feature configuration

Configuration Flags	Default Value	Description
`BOOT_FALLBACK_SUPPORT`	DISABLED	This switch adds or removes support for fallback feature
`BOOT_ANTI_ROLLBACK_SUPPORT`	DISABLED	This switch adds or removes support for anti-rollback feature
`BOOT_EXT_MEM_ENCRYPTION_SUPPORT`	DISABLED	This switch adds or removes support for external slot image encryption

You can choose to encrypt or not the external memory by activated the support of external memory encryption (`BOOT_EXT_MEM_ENCRYPTION_SUPPORT`).

If fallback feature is not activated as the encryption of the external memory, then the bootloader will be checking for a shared PSK key issued by the CycloneBOOT IAP process from the user application. The PSK key will be used to decrypt the encrypted output update image stored in external memory.

The bootloader will get the shared PSK from a dedicated shared RAM section (.boot_mailbox).

If the fallback is active as the encryption of the external memory, then you will have to provide the PSK key to the bootloader using the bootloader settings structure.

In case you activated the fallback feature you will be asked to define some specific callback in order that the bootloader can manage the fallback procedure.

This configuration MUST reflect the equivalent one of the CycloneBOOT IAP configuration in the user application. In other other words if encryption of the external memory is activated on the bootloader side it MUST also be activated in the user application side and vice-versa.

Note: All these configurations **MUST** reflect the equivalent ones of the CycloneBOOT IAP configurations in the user application. For example, if encryption of the external memory is activated on the bootloader side it **MUST** also be activated in the user application side and vice versa. The same logic applies to the fallback and roll-back features configurations.

Related topics:

- Section *Fallback feature*
- Section *Anti-rollback feature*
- Section *BootSettings*

10 Application Image

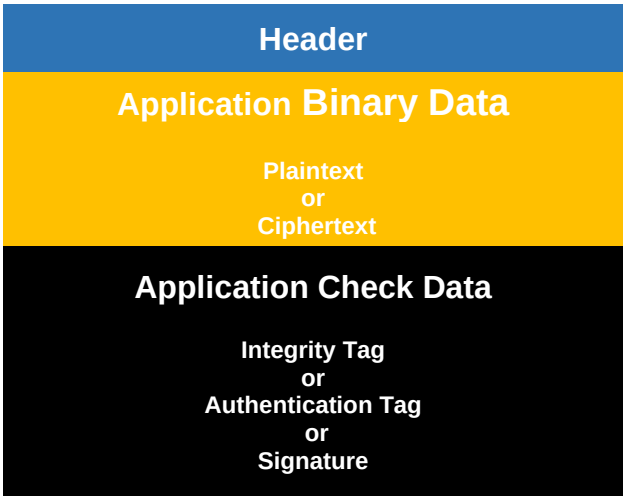
The following sections will provide a guide to the application firmware image generated by our ApplicationImageBuilder utility. These images have a specific structure (headers, data fields, etc.) that CycloneBOOT and the static bootloader (if applicable) expects.

Please refer to the “ApplicationImageBuilder Guide” for more information about generating images.

10.1 Structure

A typical image is a binary file composed of three parts:

1. **Header**: 64-byte header containing the length of the application binary. The header contains following fields:
   a. **Header Version:** Version of the header.
   b. **Image Type:** Type of the update image. As for CycloneBOOT v2, only one image type is supported.
   c. **Data Padding:** Padding between the header and the start of the binary. Only relevant in Single Bank mode. Please see section below for more information.
   d. **Data Size:** Size of the firmware application contained in the image.
   e. **Data Version:** Version of the firmware application contained in the image.
   f. **Image Time:** Time of application image generation.
   g. **Image Index:** Index used by CycloneBOOT to determine the “order” of image (used in Fallback mode).
   h. **Reserved Field:** Reserved for future improvements.
   i. **Header CRC field:** Contains the CRC32 checksum of the header, used to check the integrity of the header by CycloneBOOT.
2. **Application Binary Data**: can be encrypted or unencrypted. In case the firmware binary is encrypted, the IV (initialization vector) used to encrypt the image will be embedded at the beginning of the firmware binary potion of the image.
3. **Application Check Data**: can either be an integrity tag, an authentication tag, or a signature calculation from the application binary data.



Application Image Structure

Special header padding field for "Single Bank" images

Optionally, an image could also have a padding field in the header section. This is the case in application images generated for MCUs with internal flash without flash banks ("Single Bank" mode). In this MCU configuration, the static bootloader oversees jumping to the beginning of the firmware application section of the image after the update process.
However, this address cannot be any random value. The address must be a specific offset, depending on the ARM Cortex-M version and the number of user interrupt routines used in the application.

Adding a padding field in header to position the beginning of the firmware binary in a specific offset address allows us to keep the application image structure as is, preserving important metadata. To make CycloneBOOT compatible across all STM32 part numbers, the chosen padding size is fixed at 1024 bytes.

For more information about VTOR offset, please refer to the programming manual of your MCU.

A generated image **MUST** match CycloneBOOT configuration of the current application. In other words, if the current running application is configured to process images with encrypted application binary data and a signature verification, the images generated through ApplicationImageBuilder utility must contain encrypted binary data and a signature.

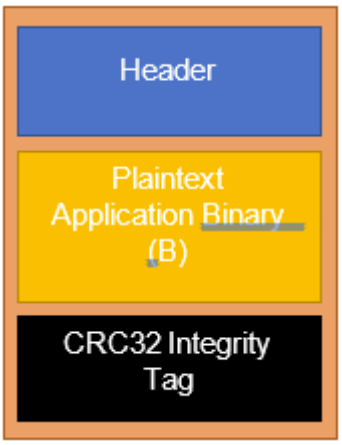
For a full treatment of ApplicationImageBuilder and the different image configurations that could be generated, please refer to "CycloneBOOT ApplicationImageBuilder Guide."

10.2 Scenarios

In this section, we will present three scenarios that cover three typical application configurations.

As mentioned previously, the generated image **MUST** match CycloneBOOT IAP configuration of the current running application. If not, the new image will be rejected by the image verification process.

10.2.1 Scenario #1: No Encryption & Integrity Check



The user does not need to encrypt the application binary but want to check its integrity:

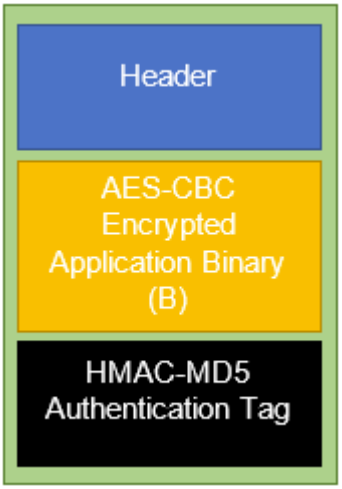
1. In the first application (A), the user configures CycloneBOOT IAP to accept images containing:
   - A plaintext application binary
   - An integrity tag generated with a specific integrity algorithm
2. The user programs this first application (A) in the microcontroller Flash through JTAG interface.
3. The user generates an image based on a new application (B) using the specific integrity algorithm defined previously.
4. The firmware update process can be initiated from application (A) to load the image containing application (B)
5. The device restarts on application (B)

Example: a user application based on **Scenario #1** could be using the following configuration:

- No Encryption
- Integrity Algorithm = CRC32

10.2.2 Scenario #2: Encryption & Authentication

This feature is only available on CycloneBOOT Pro and CycloneBOOT Ultimate versions.



The user needs to encrypt the application binary and to authenticate it.

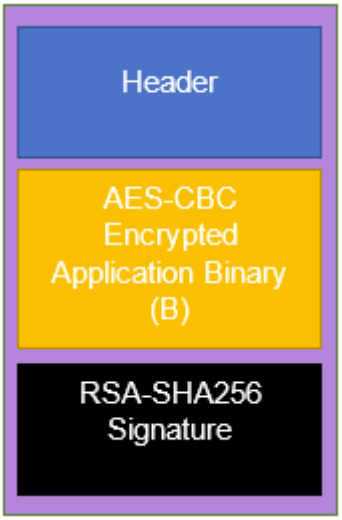
1. In the first application (A), the user configures CycloneBOOT IAP to accept images containing:
   - An application binary encrypted with a specific cipher algorithm, cipher mode and cipher key
   - An authentication tag generated with a specific algorithm, key, and integrity algorithm
2. The user programs this first application (A) in the microcontroller Flash through JTAG interface.
3. The user generates an image based on a new application (B) using the specific encryption and authentication method defined previously.
4. The firmware update process can be initiated from application (A) to load the image containing application (B).
5. The device restarts on application (B).

Example: a user application based on **Scenario #2** could be using the following configuration

- Encryption:
  - Algorithm = AES-CBC
  - 32-bit key = <32-bit key i.e., a string>
- Authentication:
  - Algorithm = HMAC-MD5
  - 32-bit key = <32-bit key i.e., a string>

10.2.3 Scenario #3: Encryption & Signature

This feature is only available on CycloneBOOT Ultimate version.



The user needs to encrypt and sign the application binary.

1. In the first application (A), the user configures CycloneBOOT IAP to accept images containing:
   - An application binary encrypted with a specific cipher algorithm, cipher mode and cipher key
   - A signature generated with a specific signature algorithm, private key and integrity algorithm
2. The user programs this first application (A) in the microcontroller Flash through JTAG interface.
3. The user generates an image based on a new application (B) using the specific encryption and signature method defined previously.
4. The firmware update process can be initiated from application (A) to load the image containing application (B).
5. The device restarts on application (B).

Example: a user application based on **Scenario #3** could be using the following configuration

- Encryption:
  - Algorithm = AES-CBC
  - 32-bit key = <32-bit key i.e., a string>
- Signature:
  - Algorithm = RSA-SHA256
  - 1024-bit public key (used for signature verification)

As a reminder, signatures use asymmetrical keys. When you generate an image with the signature method, make sure to provide the private key:

- the private key (located on your computer) is used for signature generation.
- the public key (stored in the current application binary) is used for signature verification.

10.3 Generation

The tool **app_image_builder.exe** (located in “utils/ApplicationImageBuilder”) is provided to create an application image (.img) from the binary file (.bin or binary generated by the compiler).

Depending on the selected options:

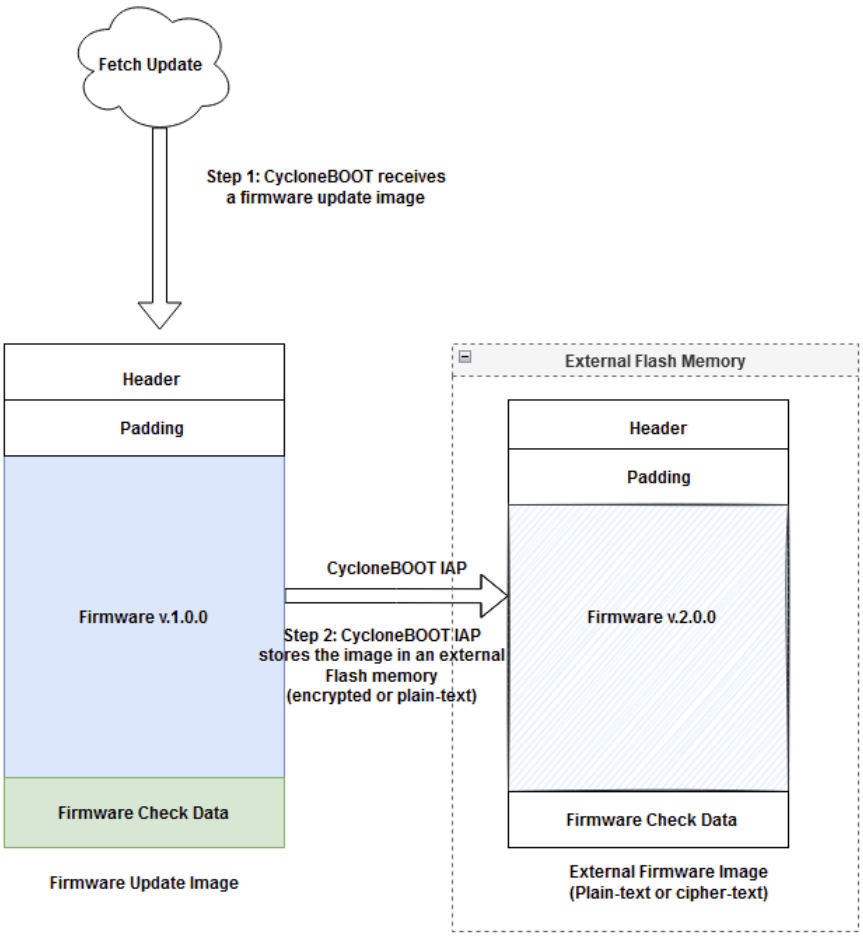
- It can encrypt the binary file or leave it unencrypted.
- It can compute an integrity tag, an authentication tag, or a signature directly from the binary file (encrypted or non-encrypted).

For a complete reference of ApplicationImageBuilder utility, please refer to “CycloneBOOT ApplicationImageBuilder User Guide.”

10.4 Processing

CycloneBOOT IAP will process an application image received through a user defined protocol. Then it will perform an update, using either the internal flash banks or an external flash device. The following sections will further detail the update flow.

10.4.1 Single Bank Processing

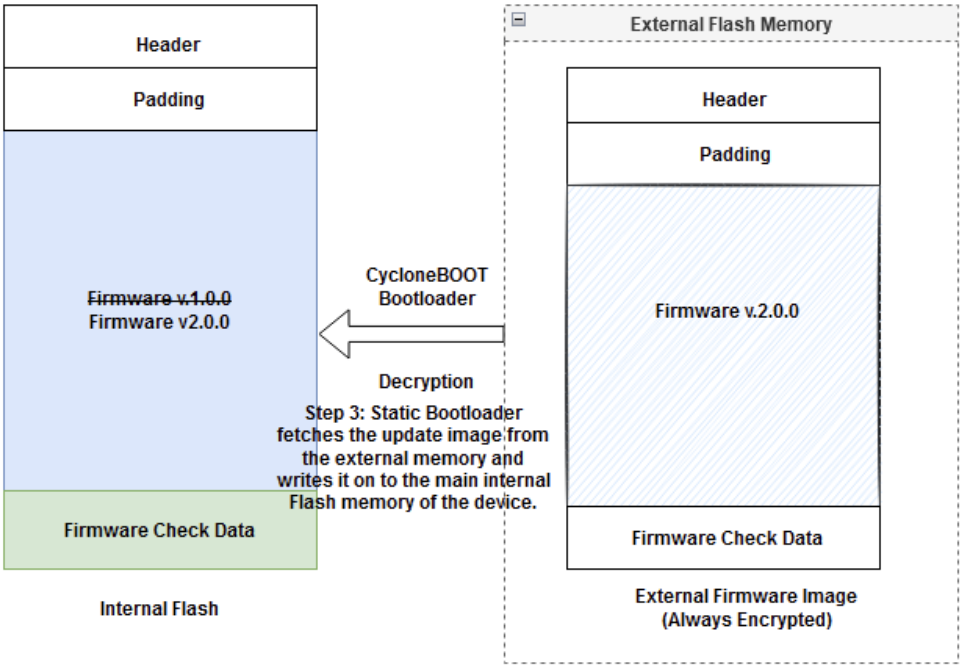


In this mode, the current firmware will fetch the update image, make sure that it is valid and then stores it inside an available slot in the external flash device.

The user can choose to encrypt the update image stored in the external memory, to provide an extra layer of protection when the device is deployed in the field.

The image stored in the external flash device is then processed by the static bootloader and written in the internal Flash memory (decrypting it in the process if it has been encrypted for storage in external memory).

10.4.2 Static Bootloader

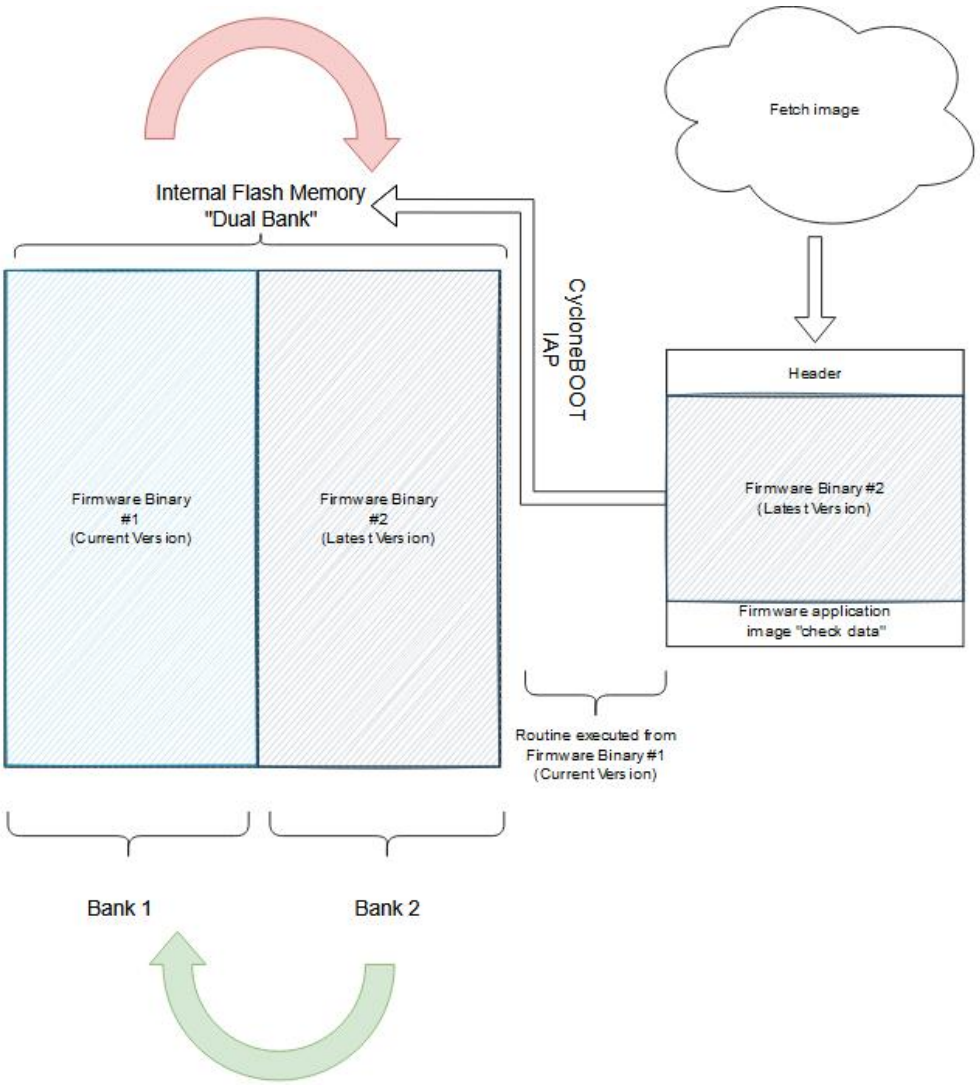


In the “Single Bank” mode when an update image is processed by CycloneBOOT IAP and stored in the external Flash device a MCU reset is performed. It also communicates any necessary decryption keys to the CycloneBOOT static bootloader via a “mailbox” that is stored in RAM.

During the MCU reset sequence, the bootloader will be able to detect that a new update image is available in external Flash device and recover (along with key required to decrypt the update image if it is encrypted for storage in external memory) and write the latest version of the firmware into the main flash memory.

The main distinction between images in Single Bank and Dual Bank mode is the fact that in Singe Bank mode, image stored in the internal flash preserves a header containing metadata, a padding field and finally its check data field. Whereas in Dual Bank mode, only the firmware potion of the update image is written in to the MCU internal flash.

10.4.3 Dual Bank Processing



Finally, in Dual Bank mode the update image will be processed by CycloneBOOT IAP, its contents verified and finally “swapped” in place of the current primary flash bank. The MCU is then reset, booting into the flash bank containing the latest firmware version.

In Dual Bank mode, contrary to Single Bank mode, when the update image is processed, only the firmware potion of the image is written into the internal flash bank. Any image header, check data potions are discarded (once information contained in these sections are verified) before the firmware is written to the internal flash.

This is a “simpler” process compared to the Singe Bank mode. The simplicity is made possible thanks to the hardware support available from the MCU to perform flash bank swaps.

Important: CycloneBOOT IAP is not reentrant and needs to be protected against concurrent access with a mutex.

11 Flash Memory Organization

This section explains general notions about MCU internal Flash organization. For more specific information for a specific MCU device, please refer to the Readme file contained with our demonstration projects.

11.1 Flash Sectors

Flash sectors correspond to the way an internal MCU Flash device is divided into blocks. These blocks or sectors each have a size and a sector number along with an address. The exact number of sectors along with their sizes and start addresses are usually found in the device reference manual.

For example, the following table is reproduced from reference manual[2] for STM32F401xB/C and STM32F401xD/E MCUs:

Table 5. Flash module organization (STM32F401xB/C and STM32F401xD/E)

Block	Name	Block base addresses	Size
Main memory	Sector 0	0x0800 0000 - 0x0800 3FFF	16 Kbytes
	Sector 1	0x0800 4000 - 0x0800 7FFF	16 Kbytes
	Sector 2	0x0800 8000 - 0x0800 BFFF	16 Kbytes
	Sector 3	0x0800 C000 - 0x0800 FFFF	16 Kbytes
	Sector 4	0x0801 0000 - 0x0801 FFFF	64 Kbytes
	Sector 5	0x0802 0000 - 0x0803 FFFF	128 Kbytes
	Sector 6	0x0804 0000 - 0x0805 FFFF	128 Kbytes
	Sector 7	0x0806 0000 - 0x0807 FFFF	128 Kbytes
System memory		0x1FFF 0000 - 0x1FFF 77FF	30 Kbytes
OTP area		0x1FFF 7800 - 0x1FFF 7A0F	528 bytes
Option bytes		0x1FFF C000 - 0x1FFF C00F	16 bytes

Here, the internal flash is divided into several blocks, including “Main memory” region, which is further subdivided into 7 sectors of varying sizes. “System memory” region contains the STMicroelectronics static bootloader, which is a factory bootloader burned-in to each MCU device. “OTP area” or One-Time-Programmable area is a special block of the Flash that is not affected by mass erase operations. Finally, “Option Bytes” area is used to configure the Flash memory. This section will be detailed in the following section.

Internal MCU Flash of the STM32F401RE device is said to be “Single Bank”. Which means that the main memory of the MCU flash is one continuous block (containing sectors of varied sizes). However, there could also be “Dual Bank” MCUs, such as STMF429 from

[2] RM0368
https://www.st.com/resource/en/reference_manual/rm0368-stm32f401xbc-and-stm32f401xde-advanced-armbased-32bit-mcus-stmicroelectronics.pdf

STMicroelectronics. The following table extracted from the STM32F429ZI reference manual[3], shows the partitioning of the internal flash in to two, also known as “Dual Bank” configuration:

Table 6. Flash module - 2 Mbyte dual bank organization (STM32F42xxx and STM32F43xxx)

Block	Bank	Name	Block base addresses	Size
Main memory	Bank 1	Sector 0	0x0800 0000 - 0x0800 3FFF	16 Kbytes
		Sector 1	0x0800 4000 - 0x0800 7FFF	16 Kbytes
		Sector 2	0x0800 8000 - 0x0800 BFFF	16 Kbytes
		Sector 3	0x0800 C000 - 0x0800 FFFF	16 Kbyte
		Sector 4	0x0801 0000 - 0x0801 FFFF	64 Kbytes
		Sector 5	0x0802 0000 - 0x0803 FFFF	128 Kbytes
		Sector 6	0x0804 0000 - 0x0805 FFFF	128 Kbytes
		-	-	-
		-	-	-
		-	-	-
		Sector 11	0x080E 0000 - 0x080F FFFF	128 Kbytes
	Bank 2	Sector 12	0x0810 0000 - 0x0810 3FFF	16 Kbytes
		Sector 13	0x0810 4000 - 0x0810 7FFF	16 Kbytes
		Sector 14	0x0810 8000 - 0x0810 BFFF	16 Kbytes
		Sector 15	0x0810 C000 - 0x0810 FFFF	16 Kbytes
		Sector 16	0x0811 0000 - 0x0811 FFFF	64 Kbytes
		Sector 17	0x0812 0000 - 0x0813 FFFF	128 Kbytes
		Sector 18	0x0814 0000 - 0x0815 FFFF	128 Kbytes
			-	-
			-	-
			-	-
		Sector 23	0x081E 0000 - 0x081F FFFF	128 Kbytes
System memory			0x1FFF 0000 - 0x1FFF 77FF	30 Kbytes
OTP			0x1FFF 7800 - 0x1FFF 7A0F	528 bytes
Option bytes	Bank 1		0x1FFF C000 - 0x1FFF C00F	16 bytes
	Bank 2		0x1FFE C000 - 0x1FFE C00F	16 bytes

Here, the internal “Main memory” block is divided into 2 “banks”. The corresponding “Option bytes” region is also divided into 2 banks. When there are multi-banks in the MCU internal flash, the configuration done within the Option bytes will dictate for example, which Bank is selected as a the main “bootable” partition. Usually, such banks will be equal in size. In this STMicroelectronics MCU, the 2Mbyte internal flash is divided along 1Mbyte bank or partition each. As a result, the size of the firmware installable in the internal Flash is reduced to 1Mbyte, or to the size of a single bank/partition.

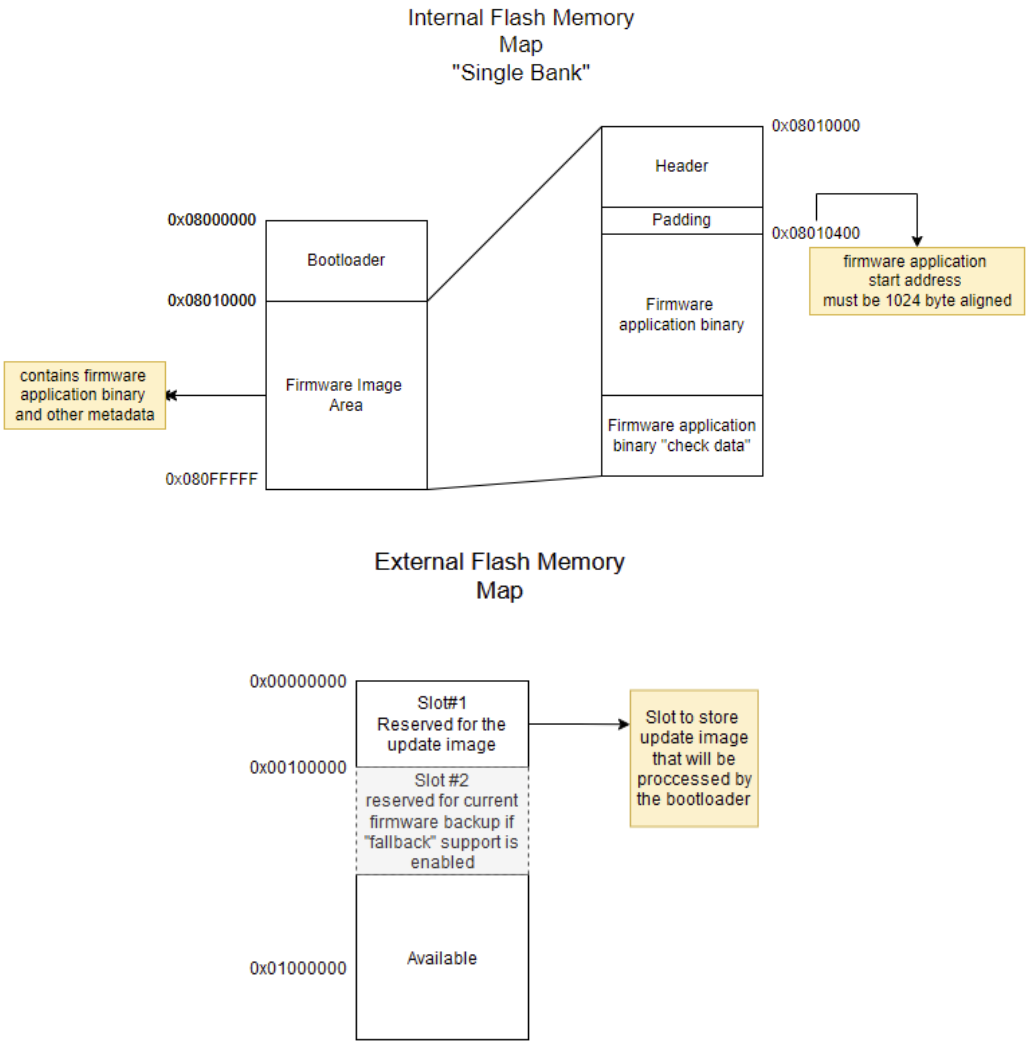
[3] RM0090
https://www.st.com/resource/en/reference_manual/rm0090-stm32f405415-stm32f407417-stm32f427437-and-stm32f429439-advanced-armbased-32bit-mcus-stmicroelectronics.pdf

11.2 Flash Partitions

CycloneBOOT employs a slightly different partition scheme depending on the configuration of the chosen MCU internal flash. For example, if the flash does not support multiple banks (“Single Bank”), then a static bootloader is there to restore the firmware update image from an external flash device. On the other hand, if the internal MCU flash consists of 2 banks (“Dual Bank”) with hardware assisted swapping, then the update image will be stored in the next available bank or partition. The user will then be able to select which firmware in either bank will be booted up.

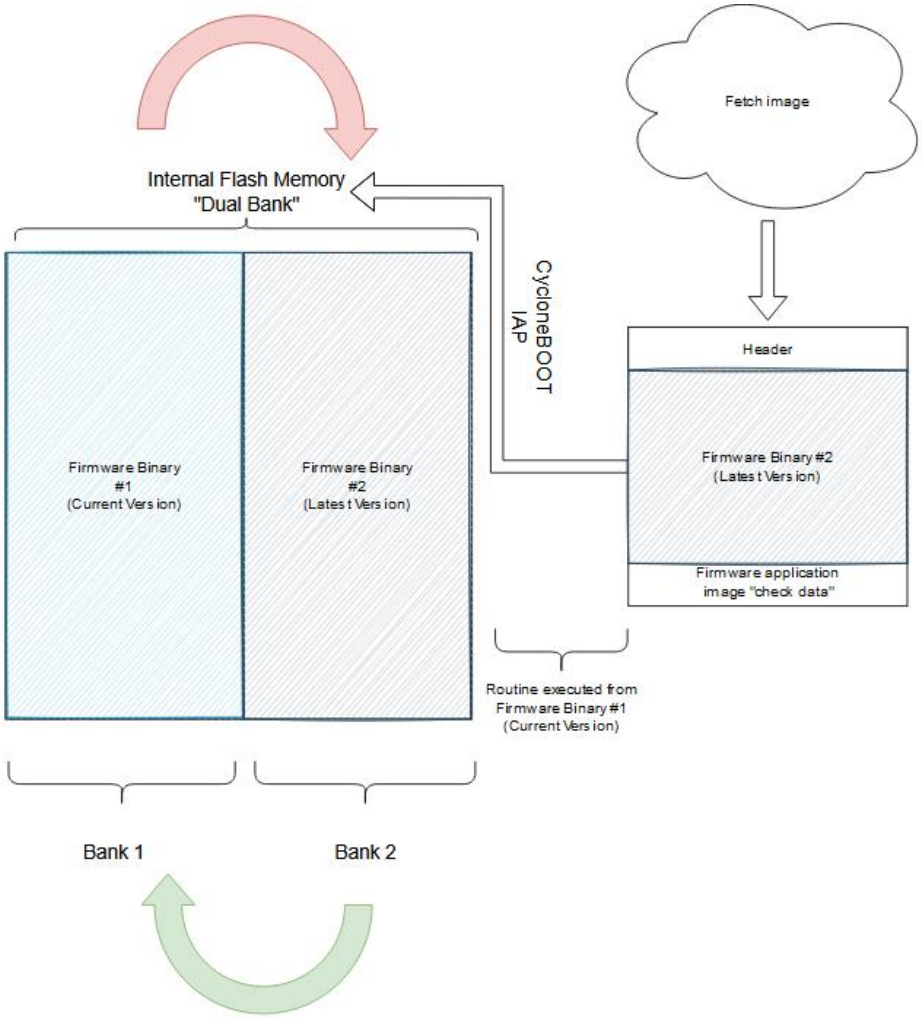
A. Single Bank

The following diagram depicts how a firmware image is stored in the internal MCU flash. The second diagram depicts the external flash device, which is used to store the incoming firmware update image. Optionally, the external flash device could contain up to 2 slots to store firmware. One slot will be used to store the update image (either encrypted or plain-text) while the second optional slot can be used to store the current firmware (or last known working firmware, encrypted or plain-text) to perform “fallback” operation if it is enabled in CycloneBOOT settings.



B. Dual Bank

When the MCU internal Flash can be partitioned in two, also known as “Dual Bank,” then each bank hosts a firmware binary each. Usually, the secondary bank is used to store the new update image and “swapped” in place of the primary partition to use the new firmware version.



For example, any application using CycloneBOOT IAP on STM32F4 or STM32F7 devices with 2MB of flash memory is organized as follows:

Start address	End Address	Size	Region
0x08000000	0x080FFFFF	1MB	Application image binary
0x08100000	0x081FFFFF	1MB	Free space for update

Each time a swap is done, the contents of the “Application image binary” region become the contents of the “Free space for update” region and vice-versa.

Note: STM32F4 and STM32F7 devices with 1MB dual bank flash have the same organization.

11.3 Option Bytes

In both, “Single Bank” and “Dual Bank” configurations, CycloneBOOT makes use of hardware mechanisms to either jump to a specific address in memory containing firmware or to perform a “swap” between primary and secondary Flash banks to finalize the firmware update.

The following sections gives a brief introduction to “Option Bytes,” the mechanism present in STM32 MCU devices to interact with the MCU internal Flash device. For more specific information, please refer to the reference manual of your chosen part number.

In brief, CycloneBOOT performs modification to the registers of Flash Option Bytes, to specify which segment of the MCU flash contains a valid bootable firmware.

Following table depicts the option byte organization for a STM32F769NI MCU:

Table 10. Option byte organization

AXI address	[63:16]	[15:0]
0x1FFF 0000	Reserved	ROP & user option bytes (**RDP** & **USER**)
0x1FFF 0008	Reserved	IWDG_STOP, IWDG_STBY and nDBANK, nDBOOT and Write protection nWRP/NWRPDB (sector 0 to 11) and user option bytes
0x1FFF 0010	Reserved	BOOT_ADD0
0x1FFF 0018	Reserved	BOOT_ADD1

This part number contains a “dual bank” internal MCU flash device. As such, a user can specify 2 different boot addresses. CycloneBOOT employs these addresses to point the MCU in the direction of the latest firmware once the update process is finalized.

Now, contrast this to a STM32 MCU without dual bank support:

Table 9. Option byte organization

Address	[63:16]	[15:0]
0x1FFF C000	Reserved	ROP & user option bytes (**RDP** & **USER**)
0x1FFF C008	Reserved	Write protection **nWRP bits for** 5 (STM32F401xB/C) and for sectors 0 to 7 (STM32F401xD/E)

There are no sperate “bootable” hardware addresses. Nevertheless, CycloneBOOT still uses Option Bytes to configure the Flash for its operations (unlock, lock, etc.).

11.4 VTOR Relocation

The vector table contains the reset value of the stack pointer, and the start addresses for all exception handlers. At startup, the processor will look for the stack pointer value and the reset vector, which is often the first routine executed at startup. This is unique to each firmware. On system reset, the vector table is systematically found at address 0x00000000.

However, in situations where multiple firmware binaries are involved, the address of a vector table is in a different memory location (usually between 0x00000000 to 0xFFFFFF80). The upper bound of this memory location is dependent on the number of interrupts implemented in the given MCU part number.

In CycloneBOOT, in Single Bank configuration mode, a padding of 1024 bytes is systematically added in order that the application vector table finds itself in a compatible address range. We have chosen 1024 bytes to be compatible across many different MCU part numbers with varying number of interrupts implemented.

By configuring the Vector table offset register (VTOR), this offset value is indicated to the MCU:

Figure 27. VTOR bit assignments

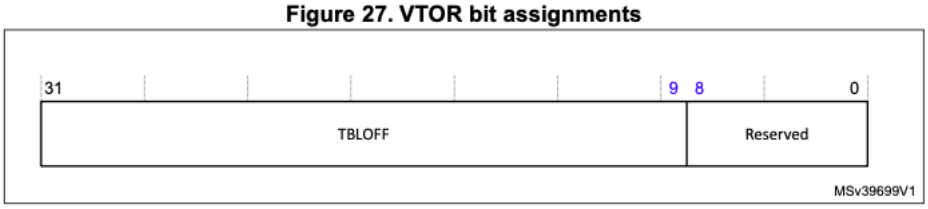


Table 54. VTOR bit assignments

Bits	Name	Function
[31:9]	TBLOFF	Vector table base offset field. It contains bits [29:7] of the offset of the table base from the bottom of the memory map.
[8:0]	-	Reserved.

The diagram above shows the offset field in the vector table offset register. Please note that this illustration comes from the reference manual of STM32F769NI MCU. Please refer to your specific MCU's reference manual for more precise information.

CycloneBOOT also modifies TBLOFF register value to then specify the specific address through which the application firmware vector table can be located.

12 Package Description

The following components are present in our “Open” and “Eval” packages, containing everything you require to evaluate CycloneBOOT before purchasing a commercial license.

These include CycloneTCP (TCP/IP stack), CycloneSSL (TLS library) and various third-party software (BSP drivers, HAL layers, RTOS, etc.) for demonstration purposes.

Open / Eval package components:

- **common:** contains features that are common to all Oryx libraries
- **cyclone_boot:** contains CycloneBOOT source code (contains a static bootloader and library source code)
- **cyclone_crypto:** contains CycloneCRYPTO library (Crypto library)
- **cyclone_ssl:** contains CycloneSSL library (TLS/DTLS library)
- **cyclone_tcp:** contains CycloneTCP library (TCP/IP stack)
- **demo:** contains several ready-to-use demo projects using various protocols (HTTP, UART) and CycloneBOOT for all major IDEs.
- **third_party:** contains third party libraries (CMSIS, freeRTOS, ST HAL)
- **utils:**
  - contains CycloneBOOT Application Image Builder utility used to generate firmware update images
  - contains ResourceCompiler utility used in some demo projects to convert static assets (HTML files, JavaScript files, CSS, etc.) to C.

Commercial package components:

- **common:** contains features that are common to all Oryx libraries
- **cyclone_boot:** contains CycloneBOOT source code (contains a static bootloader and library source code)
- **cyclone_crypto:** contains CycloneCRYPTO library (Crypto library)
- **doc:** see below
- **utils:**
  - contains CycloneBOOT Application Image Builder utility used to generate firmware update images
  - contains ResourceCompiler utility used in some demo projects to convert static assets (HTML files, JavaScript files, CSS, etc.) to C.

With our commercial packages, you will receive several documents (within **doc** directory):

- CycloneBOOT User Manual: contains everything you need to know to use CycloneBOOT alongside your application.
- CycloneBOOT AppImageBuilder Manual: containing information about the CLI utility used to generate firmware update images.
- Migration Guidelines
- Release Notes

13 Licensing Information

CycloneBOOT library and related demo applications are licensed under GPLv2 license or under a royalty-free commercial license.

Please contact us for more information: info@oryx-embedded.com

14 Version History

Revision	Date	Modifications
1.0.0	2021/03/31	Initial release
1.0.1	2021/04/09	Minor content update
2.0.0	2022/03/21	Major content update
2.1.0	2022/12/23	Minor content updates in User API section, iapSettings structure and Getting Started, source files section.