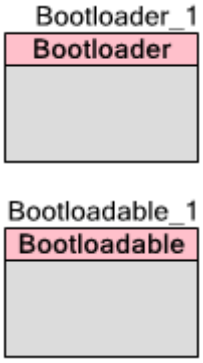
Bootloader and Bootloadable

1.40



Features

- Separate Bootloader and Bootloadable components
- Configurable set of supported commands
- Flexible component configuration

General Description

The bootloader system manages the process of updating the device flash memory with new application code and/or data. To make the process work, PSoC Creator uses the following:

- Bootloader project – Project with a Bootloader component and communication component.
- Bootloadable project – Project with a Bootloadable component, which creates the code.

Related Material

PSoC Creator provides example code; see Sample Firmware Source Code in this datasheet for more information. In addition, Cypress provides application notes located on the Cypress website or within the Cypress Document Manager. The following are a few application notes available:

- AN73854 – PSoC® 3, PSoC 4, and PSoC 5LP Introduction to Bootloaders
- AN60317 – PSoC® 3 and PSoC 5LP I2C Bootloader
- AN68272 – PSoC® 3, PSoC 4 and PSoC 5LP UART Bootloader
- AN73503 – USB HID Bootloader for PSoC® 3 and PSoC 5LP
- AN86526 – PSoC® 4 I2C Bootloader
- AN84401 – PSoC® 3 and PSoC 5LP SPI Bootloader

Export a Design to a 3rd Party Integrated Development Environment (IDE)

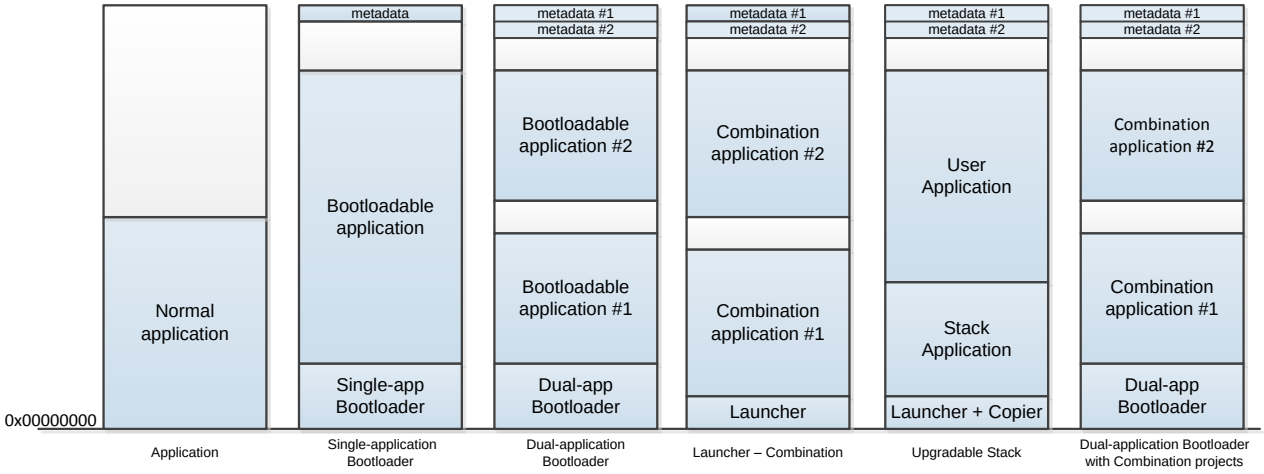
See the "Exporting a Design to a 3rd Party IDE" topic in the PSoC Creator Help for details on exporting a bootloader and bootloadable application to a 3rd party IDE.

Definitions

- **Bootloading**, **bootload operation**, or just **bootload** for short – The act of transferring a bootloadable from the host to the target flash.
- **Host** – The system that provides data to update the flash.
- **Target** – The device being updated.
- **Bootloader Component** – A PSoC Creator component that is placed onto the schematic of a project. It defines the project type as bootloader, dual-application bootloader, or launcher; and adds bootloader functionality support.
- **Bootloader Project** – A PSoC Creator project type that is defined through incorporation of a Bootloader component and a communication component. This definition may collectively include a dual-application bootloader project.
- **Bootloader** – General term for bootloader project, dual-application bootloader project, or launcher project.
- **Bootloadable Component** – A PSoC Creator component that is placed onto the schematic of a **project**. It defines the project type as bootloadable or combination, and adds bootloadable functionality support.
- **Bootloadable Project** – A PSoC Creator project type that implements a user application. It is defined through incorporation of a Bootloadable component. It is loaded into target flash by a bootloader or another application.
- **Communication Component** – Any PSoC Creator component that:
  - Advertises itself as a communication component.
  - Implements a standard set of bootloader interface functions.
- **Dual-Application Bootloader Project** – A bootloader project that supports two applications in the flash.
- **In-Application Bootloader** – A dual-application bootloader project that also perform other application-specific tasks.
- **Launcher Project** – A bootloader project that is defined through incorporation of a Bootloader component without a communication component.
- **Classic Bootloader** – Bootloader functionality of the 1.30 version. This bootloader does not support launcher/combination project types.
- **Application** – General term for bootloadable project or combination project.

- **Combination Project** – A PSoC Creator project type that implements a user application that can bootload another application. It is defined through incorporation of a Bootloader component, a Bootloadable component, and a communication component. It is loaded into target flash by another combination project.
- **Internal Memory** – Flash memory that is in the target device. With PSoC, it can be further defined as:
  - Main flash, used to store code.
  - ECC flash, which is used to check errors and correct main flash accesses, or to store additional data. PSoC 3 and PSoC 5LP have ECC flash, PSoC 4 does not.
- **Row** – A portion of internal memory accessed in a single operation. It includes main flash and ECC flash for PSoC 3/PSoC 5LP devices. PSoC 3/PSoC 5LP devices have 256 main flash bytes + 32 ECC flash bytes = 288 total bytes per row. PSoC 4100/PSoC 4200 devices have 128 bytes in a flash row. PSoC 4000 devices have 64 bytes in a flash row.

Use Cases



The diagram above represents existing bootloader and use cases. Each blue box in the diagram is a separate PSoC Creator project. See project type section to understand what components contain in each project type.

- Application – Bootloading is not supported. The application is updated through the JTAG or SWD pins, for example by using the PSoC Programmer tool.
- Single-application Bootloader – The application is updated by the Bootloader through the communication channel.
- Dual-application Bootloader – There are two applications. Either one is updated by the Bootloader. The Bootloader passes control to, or launches, an application according to the switching logic or by receiving a corresponding set of commands. The applications are allocated an equal amount of flash space.

- Launcher-Combination – The Bootloader only performs the switching function. The applications can update each other (see Definitions section). The applications are allocated an equal amount of flash space.
- Upgradable stack – This use case is intended for communication components like Bluetooth Low Energy (BLE). One application (stack) contains the communication component and "shares" it with the other application (user application). See the description of a sharing mechanism in a corresponding example project's datasheet. The Copier function makes it possible for the stack to be updated. The applications are not allocated an equal amount of flash space; the user application is located just after the stack application.
- Dual-application Bootloader with Combination projects – This use case is a more secure version of the Launcher-Combination case. The Bootloader has a communication component, so if both applications are corrupted, new applications can be bootloaded. The applications are allocated an equal amount of flash space.

Bootloader Component

The Bootloader component allows you to update the device flash memory with new code. The bootloader accepts and executes commands, then passes command responses back to the communication component. The bootloader collects and arranges the received data and manages the actual writing of flash through a simple command/status register interface.

The bootloader manages the communication protocol to receive commands from an external system and pass those commands to the bootloader. It also passes command responses from the bootloader back to the off-chip system.

Architecture	Supported Interfaces					
	Custom Interface	USB	UART	I²C	SPI	Bluetooth
PSoC 3 / PSoC 5LP	✓	✓	✓	✓	✓	-
PSOC 4000	✓	PSoC devices with USB only	Software Transmit UART only	✓	-	-
PSOC 4100 and PSoC 4200	✓	PSoC devices with USB only	✓	✓	✓	-
PSOC4100 BLE and PSoC4200 BLE	✓	PSoC devices with USB only	✓	✓	✓	✓
PRoC BLE	✓	-	✓	✓	✓	✓
PSoC 4200M	✓	✓	✓	✓	✓	-

Notes:

- The I^2C interface on PSoC 4 is implemented with the SCB component.
- The Custom Interface option allows adding bootloader support to any existing communication component. Refer to the appropriate communication component datasheet for more details about the appropriate communication method.
- For PSoC 4000 devices, each update to a flash row will automatically modify the device clock settings. Writing to flash requires changing the IMO and HFCLK settings. The configuration is restored after each row is written.
  - The HFCLK's frequency changes several times during each write to a flash row between the minimum frequency of the current IMO frequency divided by 8 and the maximum frequency of 12 MHz.
  - These clock changes impact the operation of the communication component and any other hardware in the bootloader project.
  - The I^2C slave component is tolerant of clock changes, but the clock changes can result in a NAK response when transactions occur during a row write. The bootloader host should be designed to retry in this case.

Bootloadable Component

When you use the Bootloadable component, you can specify additional parameters for the bootloadable project.

Project Types

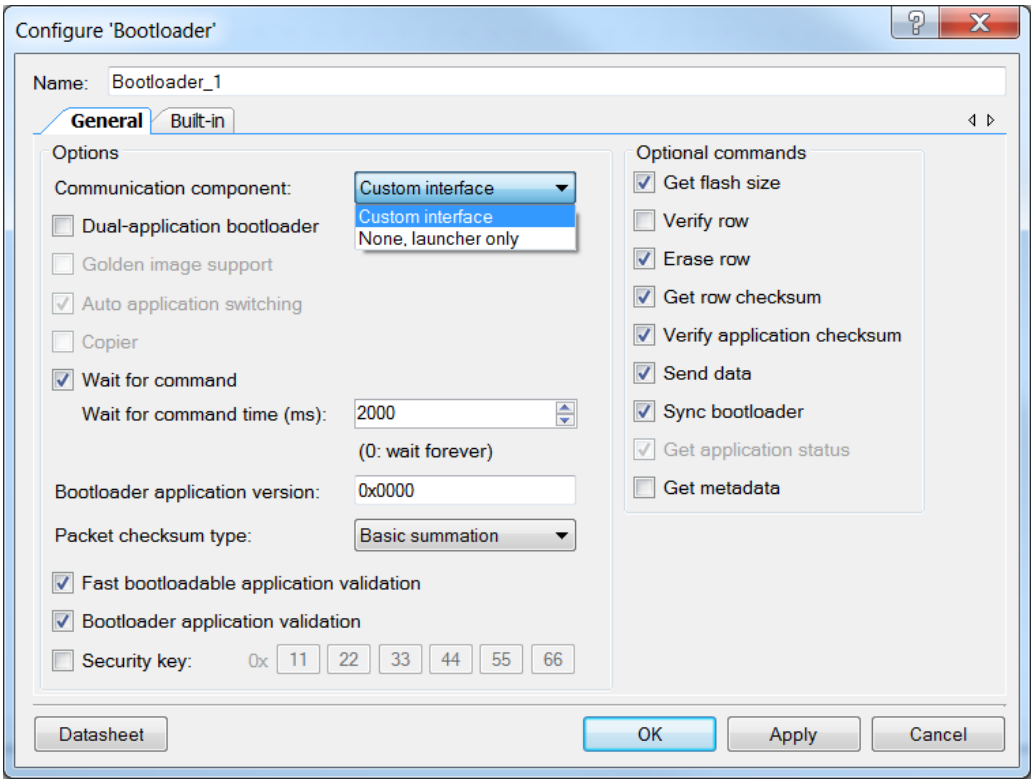
Five different project types and two Components are available. The project type is defined at build time, according to the Components in the project schematic.

Components Present in Project	Resultant Project Type
None	Normal. This project type is not used for bootloading and is not a bootloadable application.
Bootloader and communication	Bootloader
	Dual-application bootloader
Bootloader	Launcher
Bootloadable	Bootloadable
Bootloader, communication, and Bootloadable	Combination

Note Launcher and Combination project types are supported for PSoC 4 and PSoC 5LP device families and are not supported for the PSoC 3 device family.

Bootloader Component Parameters

Drag a Bootloader component onto your design and double-click it to open the **Configure** dialog.



The details of the Bootloader component's parameters are described next.

Communication component

This is the communication component that the bootloader uses to receive commands and send responses. Select one, and only one, communication component. This property is a list of the available communication protocols on the schematic that have bootloader support.

In all cases, independent of what is on the schematic, there is also a method for implementing the bootloader functions directly. For information and instructions on how to do this, see the *Component Author Guide*.

If available, selecting the **None, Launcher only** option specifies a Launcher project. This kind of bootloader project supports a limited set of functionality, which is needed for verification and launching of the bootloadable application. This kind of bootloader doesn't need a communication component.

Note This type of bootloader project can be linked only to a combination bootloadable application.

If there is no communication component on the schematic, then the **Custom interface** option is selected. This allows for implementing the communication in any way. See the corresponding component datasheet for more details about the appropriate communication method.

Dual-application bootloader

This option allows two bootloadable applications to reside in flash. It is useful for designs that require a guarantee that there is always a valid application that can be run. This guarantee comes with the limitation that each application has one half of the flash available from what would have been available for a "standard" bootloader project.

Note For a Bootload-only Bootloader in a Combination project, this option should be unchecked.

Golden image support

This option prohibits overwriting of application #1. This option is valid for a Classic Bootloader only if the **Dual-application bootloader** option is enabled; otherwise, it is ignored and grayed out.

Auto application switching

This option enables switching to the valid, but non-active application, in case the active application is not valid. This option is available for a Classic Bootloader only if the **Dual-application bootloader** is enabled; otherwise, it is ignored and grayed out.

Copier

This option is specific for Launcher-Combination architecture only and to the case when there is a large communication component like BLE. To save flash space for the user application, such a large communication component is present only in the first application (Stack application) and is "shared" with the second application (the user's application actually) by "Code sharing feature" (See code sharing and BLE Upgradable Stack example project).

If this option is enabled and the "need-to-copy" flag is set in the second application's metadata, Copier performs copying of a previously saved Stack application image from a "temporary location" to the application#1 flash space (over the current application#1).

Before performing the copying operation the new Stack application image should be received and stored at the temporary location by the current Stack application. The "need-to-copy" flag should be set in application#2 metadata indicating that copying operation is required.

The temporary location is half of the flash space that left after Launcher and 2 metadata rows. This implies that Stack application should fit in that half of the flash space.

The destination for the copy operation is obtained as the first row after Launcher (from application#1 metadata).

The source of copy operation is the first row of temporary location that is calculated by the following formula:

```
srcRow = ((((CY_FLASH_NUMBER_ROWS - dstRow - metaRows) + 1) / 2) + dstRow);
```

where:

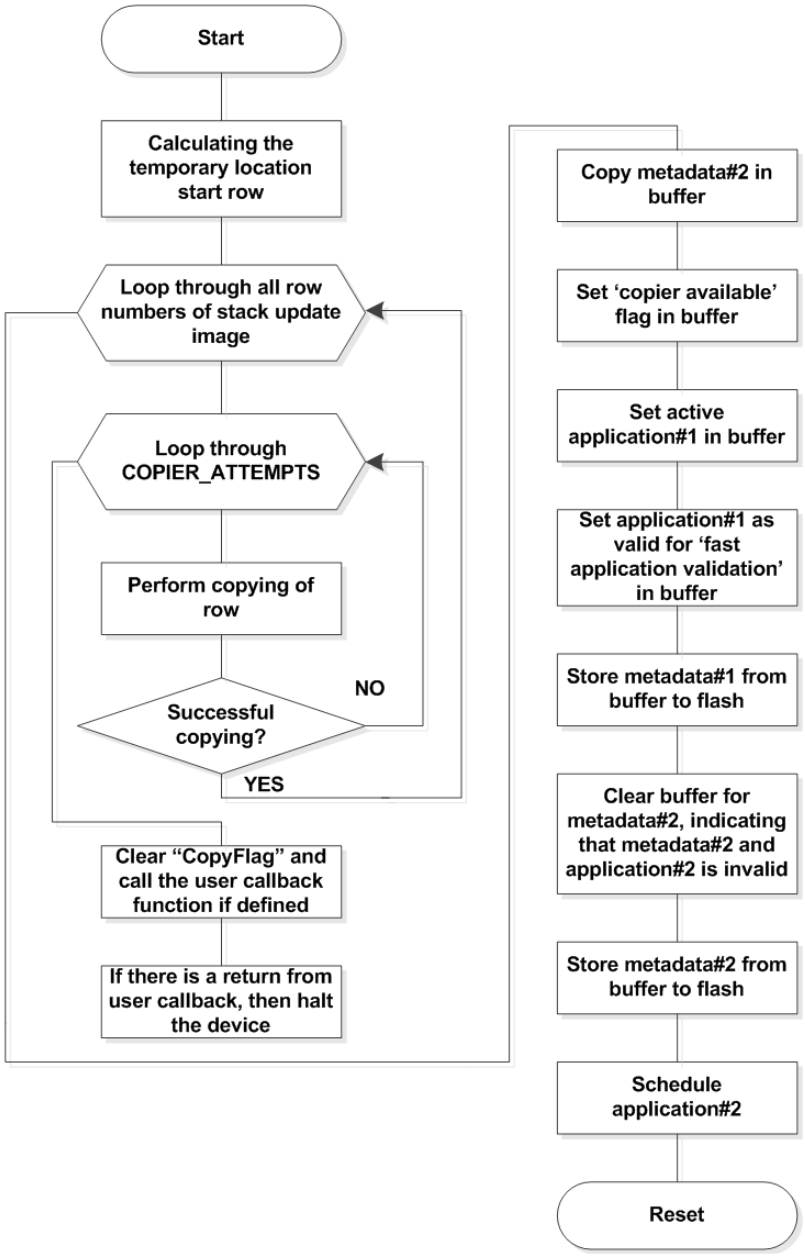
- dstRow - the destination's first row (mentioned above);

- CY_FLASH_NUMBER_ROWS - number of all flash rows;
- metaRows - 2 metadata rows. 1 is added in order to align the result with the alike calculation in a linker script, where the calculation is performed in bytes and rounding-up is performed.

Copying itself means row-by-row copying of the stored application's image. If it fails to copy some row, there is a defined number of attempts (Bootloader_MAX_ATTEMPTS) to try again. In case of no success, a user callback function Bootloader_CopierCallback() is called. If it is not defined, then the device is going to be halted.

After performing the application image copying, in case of success the application#2 metadata is copied in the stead of application#1 metadata and then application#2 metadata is cleared.

A "need-to-copy" flag is cleared just after performing the application#1 image copying from the temporary location to avoid another consecutive copying operation.


Start
Calculating the temporary location start row
Loop through all row numbers of stack update image
Loop through COPIER_ATTEMPTS
Perform copying of row
Successful copying?
NO
YES
Clear "CopyFlag" and call the user callback function if defined
If there is a return from user callback, then halt the device
Copy metadata#2 in buffer
Set 'copier available' flag in buffer
Set active application#1 in buffer
Set application#1 as valid for 'fast application validation' in buffer
Store metadata#1 from buffer to flash
Clear buffer for metadata#2, indicating that metadata#2 and application#2 is invalid
Store metadata#2 from buffer to flash
Schedule application#2
Reset

Wait for command

On device reset, the bootloader can wait for a command from the bootloader host or jump to the application code immediately. If this option is enabled, the bootloader waits for a command from the host until the timeout period specified by the **Wait for command time** parameter occurs. If the bootloader does not receive this command within the specified timeout interval, the active bootloadable project in the flash is executed after the timeout.

Wait for command time

If the bootloader waits for the command to start loading a new bootloadable application after a reset, this is the amount of time it waits before starting the existing bootloadable application. This option is valid only if **Wait for command** is enabled, otherwise it is ignored and grayed out. The zero value is interpreted as wait forever. The default value is a 2 second time out.

Bootloader application version

This parameter provides a 2 byte number to represent the version of the Bootloader application. The default value is 0x0000.

Packet checksum type

This parameter has a couple of options for the type of checksum to use when transferring packets of data between the host and the bootloader. The default value is **Basic summation**.

The basic summation checksum is computed by adding all the bytes (excluding the checksum) and then taking the 2's complement. The other option is CRC-16CCITT – the 16 bit CRC using the CCITT algorithm.

The checksum is computed for the entire packet with the exception of the Checksum and End of Packet fields.

Fast bootloadable application validation

This option controls how the bootloader verifies the application data. If it is disabled, the bootloader computes the bootloadable application checksum every time before starting it. If enabled, the bootloader only computes the checksum the first time and assumes that it remains valid in each future startup.

Bootloader application validation

If this option is enabled, the bootloader validates itself by calculating the checksum and comparing it with the saved one that resides in the internal variable. PSoC Creator generates and puts the exact value into this variable during the post-build step. If validation does not pass, the device halts. If this option is disabled, the bootloader is executed even if it is corrupted. This can lead to unpredictable results.

Note This is true for all types of projects with the exception of the Combination project type. For a Combination project type, Bootload-only Bootloader is validated as part of the whole application with a checksum stored in metadata.

Security key

This option enables the usage of a security key in the "Enter Bootloader" command for additional safety. If this option is enabled, the Bootloader checks whether the security key in the command matches the key entered in the Bootloader Configure dialog. If they do not match, then "Error Data" is returned.

The Bootloader also returns an error if the "Enter Bootloader" command contains a security key, but the **Security key** option is disabled in the Bootloader Configure dialog.

Optional Commands

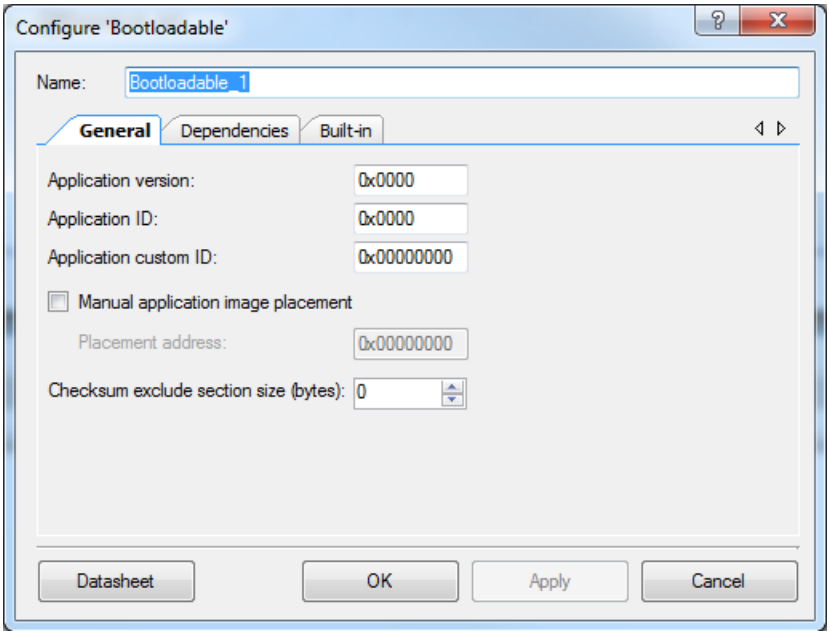
This group of options determines whether or not a corresponding command is supported by the bootloader. If enabled, then the corresponding command is supported. By default all optional commands are supported.

The **Get flash size**, **Send data**, and **Get row checksum** commands are required by the Cypress Bootloader Host tool. These commands might not be used by custom bootloader host tools.

Bootloadable Component Parameters

Drag a Bootloadable component onto your design and double-click it to open the **Configure** dialog.

General Tab



The **General** tab of the Bootloadable component contains the following parameters:

Application version

This parameter provides a 2 byte number to represent the version of the bootloadable application. The default value is 0x0000.

Application ID

This parameter provides a 2 byte number to represent the ID of the bootloadable application. The default value is 0x0000.

Application custom ID

This parameter provides a 4 byte custom ID number to represent anything in the bootloadable application. The default value is 0x00000000.

Manual application image placement

If this option is enabled, PSoC Creator places the bootloadable application image(s) at the location specified by the **Placement address** option. It is also placed according to the rules outlined in the Bootloadable section.

Use this option independently for each of the two bootloadable applications if both of them are referenced in the Dual-application bootloader application.

Placement Address

This option allows you to specify the address where the bootloadable application is placed in memory. This option is only valid if you enable the **Manual application image placement** option; otherwise, it is grayed out. You need to specify the address above the bootloader image and below the metadata area.

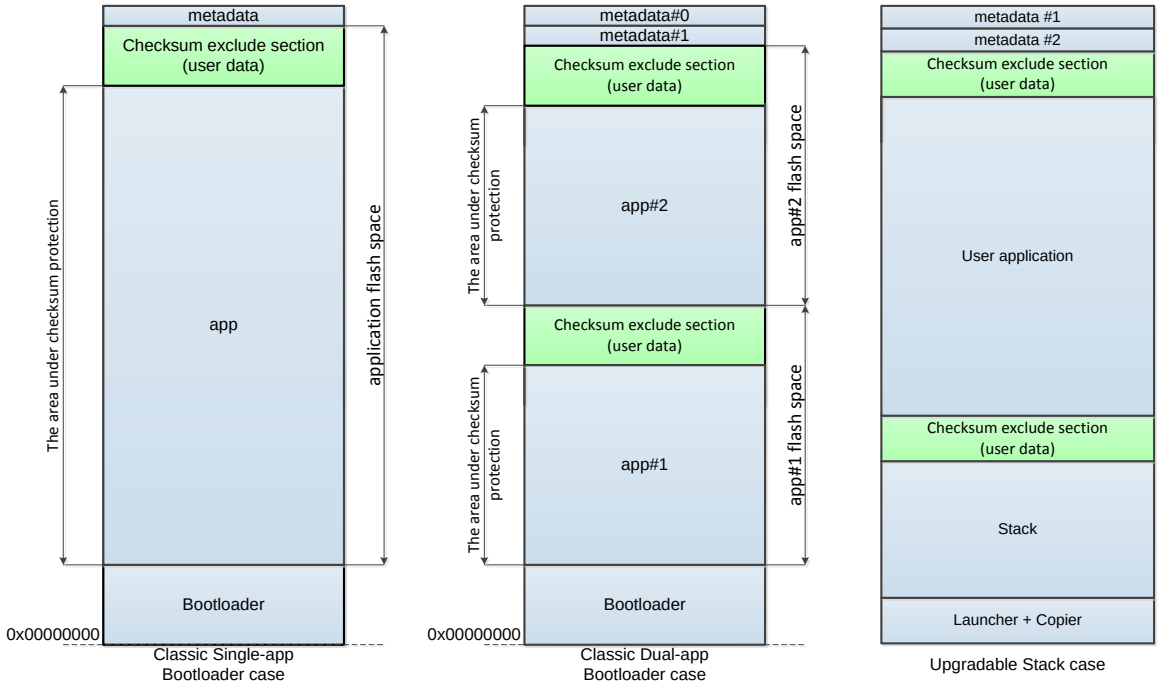
You calculate the placement address by multiplying the number of the flash row (starting from which the image is placed) by the flash row size and adding the result to the flash base address. Align the placement address to the flash row size. See the *Flash and EEPROM* chapter of the *System Reference Guide* for details about flash memory organization.

You get the first available row for the bootloadable application from the associated *<project>.cyacd* file when the **Manual application image placement** option is disabled or can be reported by the Get Flash Size command.

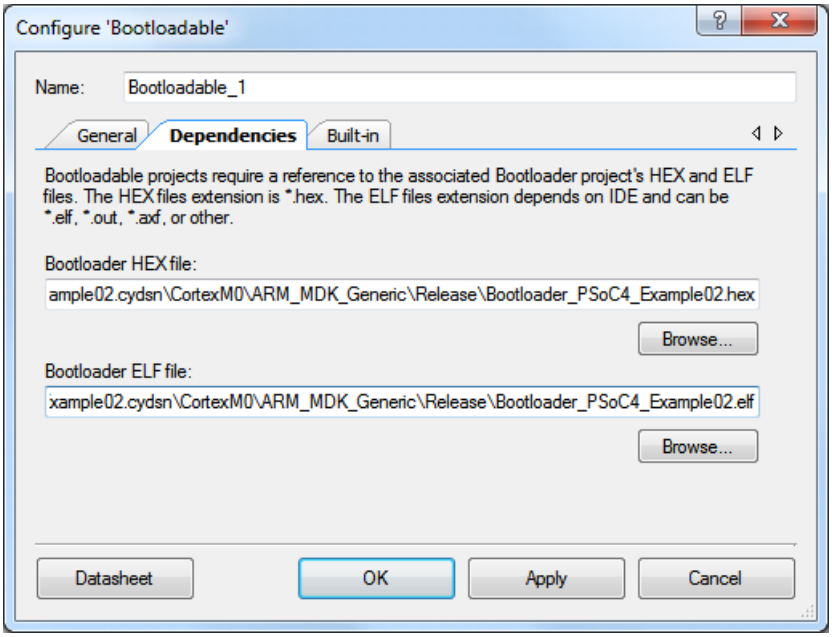
Checksum exclude section size

This option allows you to specify the size of flash section that is intended for user needs and does not take part in Bootloadable Application checksum calculation. Other components could use this section for storing data (for instance, BLE could store its pairing data). Default value is 0x00000000.

Note Do not use Bootloader commands “Send Data” and “Program Row” to update this section.



Dependencies Tab



The **Dependencies** tab of the Bootloadable component contains the following parameters:

Bootloader HEX file

This option allows you to associate a bootloadable project with the bootloader project HEX file. This is necessary so that the build of the bootloadable project gets the information about the bootloader project (for example, properly calculate where it belongs in memory).

Bootloader ELF file

This option allows you to associate a bootloadable project with the bootloader project ELF file. The ELF file extension depends on IDE. For example, PSoC Creator generates ELF files with *.elf extension, while other IDEs produce *.elf, *.out, or *.axf files.

This option is automatically populated with the path to the *.elf file, if it is located in the same folder with the specified HEX file. You can always update this option and specify the path to the ELF file manually.

Note Make sure that HEX and ELF files are generated by the same build process to ensure that they are coherent.

Application Programming Interface

Application Programming Interface (API) routines allow you to configure the component using software. By default, PSoC Creator assigns the instance name "Bootloader_1" to the first instance of a Bootloader component and "Bootloadable_1" to the first instance of a Bootloadable component in a given design. You can rename the instance to any unique value that follows the syntactic rules for identifiers. The instance name becomes the prefix of every global function name, variable, and constant symbol. For readability, the instance names used in the following tables are "Bootloader" and "Bootloadable."

Note Launcher and Combination project types are supported for PSoC 4 and PSoC 5LP device families and are not supported for PSoC 3 device family.

Functions

Classic Bootloader use case specific functions

Function	Description
Bootloader_Start	This function is called to execute the following algorithm.
Bootloader_GetMetadata	Returns the value of the specified field of the metadata section.
Bootloader_ValidateBootloadable	Verifies validation of the specified application.
Bootloader_Exit	Schedules the specified application and performs software reset to launch it.
Bootloader_Calc8BitSum	Computes the 8 bit sum for the specified data.
Bootloader_InitCallback	Initializes the callback functionality.
Bootloadable_Load	Updates the metadata area for the Bootloader to be started on device reset and resets the device.

Launcher-Combination use case specific functions

Function	Description
Bootloader_Initialize	Called for in-application bootloading, to initialize bootloading.
Bootloader_HostLink	Called for in-application bootloading, to process bootloader command from the host.
Bootloader_GetRunningAppStatus	Returns the application number of the currently running application.
Bootloader_GetActiveAppStatus	Returns the application number of the currently active application.
Bootloadable_GetActiveApplication	Gets the application which will be loaded after a next reset event.
Bootloadable_SetActiveApplication	Sets the application which will be loaded after a next reset event.

Function Documentation

void Bootloadable_Load (void)

Schedules the Bootloader/Launcher to be launched and then performs a software reset to launch it.

Returns:

This method will never return. It will load a new application and reset the device.

uint8 Bootloadable_GetActiveApplication (void)

Gets the application which will be loaded after a next reset event. NOTE Intended for the combination project type ONLY!

Returns:

A number of the current active application set in the metadata section.

0 - app#0 is set as active.

1 - app#1 is set as active.

Note:

If neither of the applications is set active, then the API returns 0x02.

cystatus Bootloadable_SetActiveApplication (uint8 appId)

Sets the application which will be loaded after a next reset event.

Theory: This API sets in the Flash (metadata section) the given active application number.

NOTE The active application number is not set directly, but the boolean mark instead means that the application is active or not for the relative metadata. Both metadata sections are updated. For example, if the second application is to be set active, then in the metadata section for the first application there will be a "0" written, which means that it is not active, and for the second metadata section there will be a "1" written, which means that it is active.

NOTE Intended for the combination project type ONLY!

Parameters:

appId	The active application number to be written to flash (metadata section) NOTE Possible values are: 0 - for the first application 1 - for the second application. Any other number is considered invalid.

Returns:

A status of writing to flash operation.

CYRET_SUCCESS - Returned if appId was successfully changed.

CYRET_BAD_PARAM - Returned if the parameter appID passed to the function has the same value as the active application ID

Note:

- The other non-zero value is considered as a failure during writing to flash.

- This API does not update Bootloader_activeApp variable

void Bootloader_Initialize (void)

Used for in-app bootloading. This function updates the global variable Bootloader_runningApp with a running application number.

If the running application number is valid (0 or 1), this function also sets the global variable Bootloader_initVar that is used to determine if the component can process bootloader commands or not. This function should be called once in the application project after a startup.

Returns:

Global variables:

- Bootloader_runningApp
- Bootloader_activeApp
- Bootloader_initVar

This API should be called first to be able to capture the active application number that is actually the running application number.

uint8 Bootloader_GetRunningAppStatus (void)

Used for dual-app or in-app bootloader. Returns the value of the global variable Bootloader_runningApp. This function should be called only after the Bootloader_Initialize() has been called once.

Returns:

The application number that is currently being executed. Values are 0 or 1; other values indicate "not defined".

uint8 Bootloader_GetActiveAppStatus (void)

Used for dual-app or in-app bootloader. Returns the value of the global variable Bootloader_activeApp. This function should be called only after the Bootloader_Initialize() has been called once.

Returns:

The desired application to be executed. Values are 0 or 1; other values indicate "not defined".

uint8 Bootloader_Calc8BitSum (uint32 baseAddr, uint32 start, uint32 size)

This computes an 8-bit sum for the provided number of bytes contained in flash (if baseAddr equals CY_FLASH_BASE) or EEPROM (if baseAddr equals CY_EEPROM_BASE).

Parameters:

baseAddr	CY_FLASH_BASE CY_EEPROM_BASE - applicable only for PSoC 3 / PSoC 5LP devices.
start	The starting address to start summing data.
size	The number of bytes to read and compute the sum.

Returns:

An 8-bit sum for the provided data.

void Bootloader_Start (void)

This function is called to execute the following algorithm:

- Validate the Bootloadable application for the Classic Single-app Bootloader or both Bootloadable/Combination applications for the Classic Dual-app Bootloader/ Launch-only Bootloader (Launcher for short) respectively.
- For the Classic Single-app Bootloader: if the Bootloadable application is valid, then the flow switches to it after a software reset. Otherwise it stays in the Bootloader, waiting for a command(s) from the host.
- For the Classic Dual-app Bootloader: the flow acts according to the switching table (see in the code below) and enabled/disabled options (for instance, auto-switching). NOTE If the valid Bootloadable application is identified, then the control is passed to it after a software reset. Otherwise it stays in the Classic Dual-app Bootloader waiting for a command(s) from the host.
- For the Launcher: the flow acts according to the switching table (see below) and enabled/disabled options. NOTE If the valid Combination application is identified, then the control is passed to it after a software reset. Otherwise it stays in the Launcher forever.
- Validate the Bootloader/Launcher application(s) (design-time configurable, Bootloader application validation option of the component customizer).
- Run a communication subroutine (design-time configurable, the Wait for command option of the component customizer). NOTE This is NOT applicable for the Launcher.
- Schedule the Bootloadable and reset the device.

See Switching logic table for details.

Returns:

This method will never return. It will either load a new application and reset the device or jump directly to the existing application. The CPU is halted, if the validation fails when the "Bootloader application validation" option is enabled. PSoC 3/PSoC 5: The CPU is halted if flash initialization fails.

If the "Bootloader application validation" option is enabled and this method determines that the Bootloader application itself is corrupt, this method will not return, instead it will simply hang the application.

void Bootloader_Exit (uint8 appId)

Schedules the specified application and performs a software reset to launch a specified application.

If the specified application is not valid, the Bootloader (the result of the ValidateBootloadable() function execution returns other than CYRET_SUCCESS, the Bootloader application is launched.

Parameters:

appId	The application to be started:

- Bootloader_EXIT_TO_BTLDR - The Bootloader application will be started on a software reset.
- Bootloader_EXIT_TO_BTLDB;
- Bootloader_EXIT_TO_BTLDB_1 - Bootloadable application # 1 will be started on a software reset.
- Bootloader_EXIT_TO_BTLDB_2 - Bootloadable application # 2 will be started on a software reset. Available only if the "Dual-application" option is enabled in the component customizer.

Returns:

This function never returns.

cystatus Bootloader_ValidateBootloadable (uint8 appId)

Performs the Bootloadable application validation by calculating the application image checksum and comparing it with the checksum value stored in the Bootloadable Application Checksum field of the metadata section.

If the "Fast bootloadable application validation" option is enabled in the component customizer and Bootloadable application successfully passes validation, the Bootloadable Application Verification Status field of the metadata section is updated. Refer to the "Metadata Layout" section for the details.

If the "Fast bootloadable application validation" option is enabled and the Bootloadable Application Verification Status field of the metadata section claims that the Bootloadable

application is valid, the function returns CYRET_SUCCESS without further checksum calculation.

Parameters:

appId	The number of the Bootloadable application should be 0 for the normal bootloader and 0 or 1 for the dual-application bootloader.

Returns:

CYRET_SUCCESS - If the specified the Bootloadable application is valid. CYRET_BAD_DATA is returned if the input parameter is out of the specified range or the calculated checksum does not match the stored checksum.

void Bootloader_SetFlashByte (uint32 address, uint8 runType)

Writes a byte to the specified flash memory location.

Parameters:

address	The address in flash memory where data will be written
runType	The byte to be written.

uint32 Bootloader_GetMetadata (uint8 field, uint8 appId)

Returns the value of the specified field of the metadata section.

Parameters:

field	The field to get data from: Bootloader_GET_BTLDB_CHECKSUM - Bootloadable Application Checksum Bootloader_GET_BTLDB_ADDR - Bootloadable Application Start Routine Address Bootloader_GET_BTLDR_LAST_ROW - Bootloader Last Flash Row Bootloader_GET_BTLDB_LENGTH - Bootloadable Application Length Bootloader_GET_BTLDB_ACTIVE - Active Bootloadable Application Bootloader_GET_BTLDB_STATUS - Bootloadable Application Verification Status

	Bootloader_GET_BTLDR_APP_VERSION - Bootloader Application Version Bootloader_GET_BTLDB_APP_VERSION - Bootloadable Application Version Bootloader_GET_BTLDB_APP_ID - Bootloadable Application ID Bootloader_GET_BTLDB_APP_CUST_ID - Bootloadable Application Custom ID
appId	The number of the Bootlodable application. Should be 0 for the normal bootloader and 0 or 1 for the Dual-application bootloader.

Returns:

The value of the specified field of the specified application.

void Bootloader_InitCallback(Bootloader_callback_type userCallback)

This function initializes the callback functionality.

Parameter:

The user's callback function.

Returns:

None.

void Bootloader_HostLink(uint8 timeOut)

Causes the Bootloader to attempt to read data being transmitted by the host application. If data is sent from the host, this establishes the communication interface to process all requests.

This function is public only for Launcher-Combination architecture. For Classic Bootloader it is static, meaning private.

Parameter:

timeOut - The amount of time to listen for data before giving up. The timeout is measured in 10s of ms. Use 0 for an infinite wait.

Returns:

None

Variables

- uint8 Bootloader_initVar = (0u)
- uint8 Bootloader_runningApp = (2u)
- uint8 Bootloader_isBootloading = (0u)
- uint8 Bootloader_activeApp = Bootloader_MD_BTLDB_ACTIVE_NONE

Variable Documentation

uint8 Bootloader_initVar = (0u)

This variable is intended to indicate that in-application bootloading initialization is done. The initialization itself is performed in the Bootloader_Initialize() function. Once the initialization is done, the variable is set and this prevents the functionality from reinitialization.

uint8 Bootloader_runningApp = (2u)

This variable is intended to keep the current application number. It applies only to in-application bootloading.

uint8 Bootloader_isBootloading = (0u)

This variable is intended to indicate that 'Enter bootloader' command has been received. It applies only to in-application bootloading.

uint8 Bootloader_activeApp = Bootloader_MD_BTLDB_ACTIVE_NONE

This variable is intended to keep the active application number. It applies only to in-application bootloading.

Constants

- Error Codes
- Commands
- Metadata fields

Bootloader's deprecated code

This component contains deprecated code that is not recommended for use but is kept to preserve backward compatibility with the existing designs. The deprecated code should no longer be used in new projects.

The following macro and definitions are NOT recommended for usage:

- Bootloader_BOOTLOADABLE_APP_VALID
- CyBtldr_Start
- Bootloader_NUM_OF_FLASH_ARRAYS
- Bootloader_META_BASE(x)
- Bootloader_META_ARRAY
- Bootloader_META_APP_ENTRY_POINT_ADDR(x)
- Bootloader_META_APP_BYTE_LEN(x)
- Bootloader_META_APP_RUN_ADDR(x)
- Bootloader_META_APP_ACTIVE_ADDR(x)
- Bootloader_META_APP_VERIFIED_ADDR(x)
- Bootloader_META_APP_BLDBL_VER_ADDR(x)
- Bootloader_META_APP_VER_ADDR(x)
- Bootloader_META_APP_ID_ADDR(x)
- Bootloader_META_APP_CUST_ID_ADDR(x)
- Bootloader_META_LAST_BLDR_ROW_ADDR(x)
- Bootloader_META_CHECKSUM_ADDR(x)
- Bootloader_MD_BASE
- Bootloader_MD_ROW
- Bootloader_MD_CHECKSUM_ADDR
- Bootloader_MD_LAST_BLDR_ROW_ADDR
- Bootloader_MD_APP_BYTE_LEN
- Bootloader_MD_APP_VERIFIED_ADDR

- Bootloader_MD_APP_ENTRY_POINT_ADDR
- Bootloader_MD_APP_RUN_ADDR
- Bootloader_MD_CHECKSUM_ADDR
- Bootloader_MD_LAST_BLDR_ROW_ADDR
- Bootloader_MD_APP_BYTE_LEN
- Bootloader_MD_APP_VERIFIED_ADDR
- Bootloader_MD_APP_ENTRY_POINT_ADDR
- Bootloader_MD_APP_RUN_ADDR
- Bootloader_P_APP_ACTIVE(x)
- Bootloader_MD_PTR_CHECKSUM
- Bootloader_MD_PTR_APP_ENTRY_POINT
- Bootloader_MD_PTR_LAST_BLDR_ROW
- Bootloader_MD_PTR_APP_BYTE_LEN
- Bootloader_MD_PTR_APP_RUN_ADDR
- Bootloader_MD_PTR_APP_VERIFIED
- Bootloader_MD_PTR_APP_BLD_BL_VER
- Bootloader_MD_PTR_APP_VER
- Bootloader_MD_PTR_APP_ID
- Bootloader_MD_PTR_APP_CUST_ID
- Bootloader_APP_ADDRESS
- Bootloader_GET_CODE_DATA(idx)
- Bootloader_GET_CODE_WORD(idx)
- Bootloader_META_APP_ADDR_OFFSET
- Bootloader_META_APP_BL_LAST_ROW_OFFSET
- Bootloader_META_APP_BYTE_LEN_OFFSET

- Bootloader_META_APP_RUN_TYPE_OFFSET
- Bootloader_META_APP_ACTIVE_OFFSET
- Bootloader_META_APP_VERIFIED_OFFSET
- Bootloader_META_APP_BL_BUILD_VER_OFFSET
- Bootloader_META_APP_ID_OFFSET
- Bootloader_META_APP_VER_OFFSET
- Bootloader_META_APP_CUST_ID_OFFSET
- Bootloader_GET_REG16(x)
- Bootloader_GET_REG32(x)
- Bootloader_META_APP_CHECKSUM_OFFSET
- Bootloader_META_DATA_SIZE
- appRunType;
- Bootloader_SOFTWARE_RESET
- Bootloader_SetFlashRunType(runType)
- Bootloader_START_APP
- Bootloader_START_BTLDR
- CYDEV_FLASH_BASE

Bootloadable's deprecated code

This component contains deprecated code that is not recommended for use but is kept to preserve backward compatibility with the existing designs. The deprecated code should no longer be used in new projects.

The following macro and definitions are NOT recommended for usage:

- CYBTDLR_SET_RUN_TYPE(x)
- Bootloadable_START_APP
- Bootloadable_START_BTLDR

- Bootloadable_META_DATA_SIZE
- Bootloadable_META_APP_CHECKSUM_OFFSET
- Bootloadable_APP_ADDRESS
- Bootloadable_GET_CODE_WORD(idx)
- Bootloadable_META_APP_ADDR_OFFSET
- Bootloadable_META_APP_BL_LAST_ROW_OFFSET
- Bootloadable_META_APP_BYTE_LEN_OFFSET
- Bootloadable_META_APP_RUN_TYPE_OFFSET
- Bootloadable_META_APP_ACTIVE_OFFSET
- Bootloadable_META_APP_VERIFIED_OFFSET
- Bootloadable_META_APP_BL_BUILD_VER_OFFSET
- Bootloadable_META_APP_ID_OFFSET
- Bootloadable_META_APP_VER_OFFSET
- Bootloadable_META_APP_CUST_ID_OFFSET
- Bootloadable_SetFlashRunType(runType)
- Bootloadable_SetFlashByte(uint32 address, uint8 runType)

Error Codes

Description

Error codes that are returned while communicating with Host

- #define **Bootloader_ERR_KEY**(0x01u)/** The provided key does not match the expected value */
- #define **Bootloader_ERR_VERIFY**(0x02u)/** The verification of flash failed */
- #define **Bootloader_ERR_LENGTH**(0x03u)/** The amount of data available is outside the expected range*/
- #define **Bootloader_ERR_DATA**(0x04u)/** The data is not of the proper form*/
- #define **Bootloader_ERR_CMD**(0x05u)/** The command is not recognized */

- #define **Bootloader_ERR_DEVICE**(0x06u)/** The expected device does not match the detected device*/
- #define **Bootloader_ERR_VERSION**(0x07u)/** The bootloader version detected is not supported*/
- #define **Bootloader_ERR_CHECKSUM**(0x08u)/** The checksum does not match the expected value*/
- #define **Bootloader_ERR_ARRAY**(0x09u)/** The flash array is not valid*/
- #define **Bootloader_ERR_ROW**(0x0Au)/** The flash row is not valid */
- #define **Bootloader_ERR_PROTECT**(0x0Bu)/** The flash row is protected and can not be programmed*/
- #define **Bootloader_ERR_APP**(0x0Cu)/** The application is not valid and cannot be set as active */
- #define **Bootloader_ERR_ACTIVE**(0x0Du)/** The application is currently marked as active*/
- #define **Bootloader_ERR_CALLBACK**(0x0Eu)/** The callback function returns invalid data*/
- #define **Bootloader_ERR_UNK**(0x0Fu)/** An unknown error occurred*/

Commands

Description

Commands for communication with Host

- #define **Bootloader_COMMAND_CHECKSUM**(0x31u) /* Verify the checksum for the bootloadable project*/
- #define **Bootloader_COMMAND_REPORT_SIZE**(0x32u) /* Report the programmable portions of flash */
- #define **Bootloader_COMMAND_APP_STATUS**(0x33u) /* Gets status info about the provided app status*/
- #define **Bootloader_COMMAND_ERASE**(0x34u) /* Erase the specified flash row */
- #define **Bootloader_COMMAND_SYNC**(0x35u) /* Sync the bootloader and host application*/

- #define **Bootloader_COMMAND_APP_ACTIVE**(0x36u) /* Sets the active application*/
- #define **Bootloader_COMMAND_DATA**(0x37u) /* Queue up a block of data for programming*/
- #define **Bootloader_COMMAND_ENTER**(0x38u) /* Enter the bootloader */
- #define **Bootloader_COMMAND_PROGRAM**(0x39u) /* Program the specified row*/
- #define **Bootloader_COMMAND_GET_ROW_CHKSUM**(0x3Au) /* Compute flash row checksum for verification*/
- #define **Bootloader_COMMAND_EXIT**(0x3Bu) /* Exits the bootloader & resets the chip */
- #define **Bootloader_COMMAND_GET_METADATA**(0x3Cu) /* Reports the metadata for a selected application */
- #define **Bootloader_COMMAND_VERIFY_FLS_ROW**(0x45u) /* Verifies data in buffer with specified row in flash*/

Metadata fields

Description

Error codes that are returned while communicating with Host

- #define **Bootloader_GET_BTLDB_CHECKSUM** (1u) /* Bootloadable Application Checksum */
- #define **Bootloader_GET_BTLDB_ADDR** (2u) /* Bootloadable Application Start Routine Address */
- #define **Bootloader_GET_BTLDB_LAST_ROW** (3u) /* Bootloader Last Flash Row */
- #define **Bootloader_GET_BTLDB_LENGTH** (4u) /* Bootloadable Application Length */
- #define **Bootloader_GET_BTLDB_ACTIVE** (5u) /* Active Bootloadable Application */
- #define **Bootloader_GET_BTLDB_STATUS** (6u) /* Bootloadable Application Verification Status */
- #define **Bootloader_GET_BTLDR_APP_VERSION** (7u) /* Bootloader Application Version */
- #define **Bootloader_GET_BTLDB_APP_VERSION** (8u) /* Bootloadable Application Version */
- #define **Bootloader_GET_BTLDB_APP_ID** (9u) /* Bootloadable Application ID */

- #define **Bootloader_GET_BTLDB_APP_CUST_ID** (10u) /* Bootloadable Application Custom ID */
- #define **Bootloader_GET_BTLDB_COPY_FLAG** (11u) /* "need-to-copy” flag */
- #define **Bootloader_GET_BTLDB_USER_DATA** (12u) /* User data */

Data Structure Documentation

Bootloader_ENTER Struct Reference

Data Fields

- uint32 SiliconId
- uint8 Revision
- uint8 BootLoaderVersion [3u]

Sample Firmware Source Code

PSoC Creator provides many example projects that include schematics and example code in the Find Example Project dialog. For component-specific examples, open the dialog from the Component Catalog or an instance of the component in a schematic. For general examples, open the dialog from the Start Page or **File** menu. As needed, use the **Filter Options** in the dialog to narrow the list of projects available to select.

See the "Find Example Project" topic in the PSoC Creator Help for more information.

MISRA Compliance

This section describes the MISRA-C:2004 compliance and deviations for the component. There are two types of deviations defined:

- project deviations – deviations that are applicable for all PSoC Creator components
- specific deviations – deviations that are applicable only for this component

This section provides information on component-specific deviations. Project deviations are described in the *MISRA Compliance* section of the *System Reference Guide* along with information on the MISRA compliance verification environment.

Bootloader Component Specific Deviations:

Rule	Rule Class	Rule Description	Description of Deviation(s)
13.7	R	Boolean operations whose results are invariant shall not be permitted.	For some parts, which have only one flash array, there is a specific if statement that is always false. The generalized implementation approach is used, so there is no differentiation for those parts.
14.3	R	Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment provided that the first character following the null statement is a white-space character.	The null statement is located close to other code: the CyGlobalIntEnable macro is followed by a semi-colon, while its implementation includes a semi-colon. Applicable for PSoC 3/PSoC 5 devices.
14.5	R	The continue statement shall not be used.	A 'continue' statement has been used in 2 places to simplify packet processing.
14.7	R	A function shall have a single point of exit at the end of the function.	Multiple points of exit are used in the function that verifies validity of the bootloadable applications.
19.7	A	A function should be used in preference to a function-like macro.	Deviated since function-like macros are used to allow more efficient code.

Bootloadable Component Specific Deviations:

Rule	Rule Class	Rule Description	Description of Deviation(s)
19.7	A	A function should be used in preference to a function-like macro.	Deviated since function-like macros are used to allow more efficient code.

API Memory Usage

The component memory usage varies significantly, depending on the compiler, device, number of APIs used and component configuration. The following table provides the memory usage for all APIs available in the given component configuration.

The measurements were done with the associated compiler configured in Release mode with optimization set for Size. For a specific design, the map file generated by the compiler can be analyzed to determine the memory usage.

PSoC 3 (Keil_PK51)

Configuration	Flash Bytes	SRAM Bytes
Bootloader	3024	4
Full Bootloader Application[1]	7936	998
Full Bootloadable Application[2]	1792	96
Launch-only Bootloader[3]	-	-
Launcher project[4]	-	-
Bootload-only Bootloader[5]	-	-
Launcher + Combination + Metadata[6]	-	-

1 The measurements for this configuration were done for the entire bootloader project, with the fixed-function based I²C used as communication component and Bootloader component configured for the minimal flash consumption.

2 The measurements for this configuration were done for entire bootloadable project.

3 Launch-only Bootloader means Bootloader component that is configured as Launcher.

4 The whole Launcher project means the whole project size for Launch-only Bootloader and cy_boot component.

5 Bootload-only Bootloader means Bootloader component in Combination project that does bootload operation, it does not include Communication Component.

6 The measurements for this configuration are done for entire combination project (Bootload-only Bootloader, Bootloadable, fixed-function I2C) with the whole Launcher project (Launch-only Bootloader and cy_boot).

PSoC 4 (GCC)

Configuration	PSoC 4000		PSoC 4100 PSoC 4200		PSoC 4100 BLE PSoC 4200 BLE		PSoC 4100M PSoC 4200M	
	Flash Bytes	SRAM Bytes	Flash Bytes	Flash Bytes	Flash Bytes	SRAM Bytes	Flash Bytes	SRAM Bytes
Bootloader	1096	12	940	12	948	12	948	12
Full Bootloader Application[1]	4544	364	4352	448	4352	464	4224	440
Full Bootloadable Application[2]	5568	148	5632	256	5760	256	5760	256
Launch-only Bootloader[3]	620	12	620	12	620	12	620	12
Launcher project[4]	2304	164	2304	264	2304	280	2176	264
Bootload-only Bootloader[5]	1508	14	1508	14	1500	14	1500	14
Launcher + Combination + Metadata[6]	8064	364	8192	448	8192	464	7936	440

PSoC 5LP (GCC)

Configuration	Flash Bytes	SRAM Bytes
Bootloader	1256	12
Full Bootloader Application[1]	4608	613[7]
Full Bootloadable Application[2]	5888	301[7]
Launcher project[4]	2304	333[7]
Bootload-only Bootloader[5]	1910	14
Launcher + Combination + Metadata[6]	8192	613[7]

Callback Functions

Callback functions allow users to execute code from API files that are automatically generated by PSoC Creator. Refer to the PSoC Creator *Component Author Guide* for the more details.

In order to add code to the callback function present in the component's generated sources, perform the following:

- Define a macro to signal the presence of a callback (in *cyapicallbacks.h*). This will make "uncomment" the function call from the component's source code.
- Write the function declaration (in *cyapicallbacks.h*). This will make this function visible by all the project files.

[7] The SRAM usage is shown without space reserved for heap and stack.

- Write the function implementation (in any user file).

Callback Function	Associated Macro	Description
void Bootloader_CopierCallback(void)	LAUNCHER_COPIER_CALLBACK	This function lets the user to define the device's behavior the when the copy operation failed and the device is going to be halted. That callback function could somehow indicate the reasons of the device being halted. See Copier for details.
void Bootloader_Callback (&Bootloader_inPacket, &Bootloader_outPacket)	-	This function lets the user to define his/her own command for communication with the Host. The existing commands are described here: Bootloader Commands

Bootloader_CopierCallback

This feature is intended for the situation when the Copier is available and in case a copy operation fails after some numbers of attempts, then Bootloader_CopierCallback() is called. If it is not defined, then the device is halted.

The following define should be present in *cyapicallbacks.h*:

```
#define LAUNCHER_COPIER_CALLBACK (1u)
#if (0u != LAUNCHER_COPIER_CALLBACK)
  void Bootloader_CopierCallback(void);
#endif /*(0u != LAUNCHER_COPIER_CALLBACK)*/
The following is an example of Bootloader_CopierCallback():
#if (0u != LAUNCHER_COPIER_CALLBACK)
    void Bootloader_CopierCallback(void)
    {
        /* Halt the device */
        CyHalt(0x00);
    }
#endif /*(0u != LAUNCHER_COPIER_CALLBACK)*/
```

Bootloader_Callback

This feature is intended for creation of the user's own command for communication between Host and Bootloader. The callback feature is available only if a callback functionality is initialized by Bootloader_InitCallback() function.

The callback prototype is the following:

```
void User_Callback_Function(Bootloader_in_packet_type* inputPacket,
                            Bootloader_out_packet_type* outputPacket)
```

Parameters:

inputPacket — A pointer on the following input structure:

```
struct
{
    uint8 command;
    uint16 packetLength;
    uint8* pInputBuffer;
} Bootloader_in_packet_type;
```

where

- *command* – a code of a new command that is going to be processed by callback. See the existing commands in Bootloader Commands section. If your command code matches with the existing enabled command, then the existing command will be performed.
- *packetLength* – an input data length in bytes.
- *pInputBuffer* – a pointer on the Bootloader's component internal buffer (the maximum data length is (`Bootloader_SIZEOF_COMMAND_BUFFER- Bootloader_MIN_PKT_SIZE`).

Note By default this is a pointer on the same buffer that *pOutputBuffer* points on, so once something is written by *pOutputBuffer*, the input data by *pInputBuffer* pointer is lost.

outputPacket — A pointer on the following structure, actually the return:

```
struct
{
    cystatus status;
    uint16 packetLength;
    uint8* pOutputBuffer;
} Bootloader_out_packet_type;
```

where:

- *status* – the returned status, the existing error codes are described in Status/Error codes section.
- *packetLength* - a returned data length in bytes.
- *pOutputBuffer* – a pointer on the Bootloader's component internal buffer (the maximum data length is `Bootloader_SIZEOF_COMMAND_BUFFER- Bootloader_MIN_PKT_SIZE).`

Note By default this is a pointer on the same buffer that *pInputBuffer* points on, so once something is written by *pOutputBuffer*, the input data by *pInputBuffer* pointer is lost.

The packet structure is described in the Bootloader Packet Structure section.

The maximum input/output packet length (for instance, Start-of-packet + Command + Data Length + Data + Checksum + End-of-packet) is currently limited to `Bootloader_SIZEOF_COMMAND_BUFFER`. The minimum length is `Bootloader_MIN_PKT_SIZE` bytes.

Note There is a “post processing” after a callback performed in Bootloader component. If the output data length (`outputPacket->packetLength`) exceeds (`Bootloader_SIZEOF_COMMAND_BUFFER-Bootloader_MIN_PKT_SIZE)` bytes and `outputPacket-> pOutputBuffer` points by default on internal buffer or `outputPacket-> pOutputBuffer` points to NULL and `outputPacket->packetLength` is non-zero, then the following error is returned `Bootloader_ERR_CALLBACK`.

The following is an example:

```
#define CALLBACK_COMMAND        (0x60u)
#define CALLBACK_COMMAND_LENGTH (Bootloader_MIN_PKT_SIZE)
#define SOME_RETURN_DATA        (0xAAu)
#define OUTPUT_DATA_LENGTH      (Bootloader_MIN_PKT_SIZE + 1u)

void Bootloader_Callback(Bootloader_in_packet_type* inPacket,
Bootloader_out_packet_type* outPacket)
{
    outPacket->pOutputBuffer = NULL;
    outPacket->packetLength = Bootloader_MIN_PKT_SIZE;

    if (CALLBACK_COMMAND == inPacket->command)
    {
        if (CALLBACK_COMMAND_LENGTH == inPacket->packetLength)
        {
            outPacket->status = CYRET_SUCCESS;
            outPacket->packetLength = OUTPUT_DATA_LENGTH;
            outPacket->pOutputBuffer[0] = SOME_RETURN_DATA;
        }
        else
        {
            /* The command's length is not correct */
            outPacket->status = Bootloader_ERR_LENGTH;
        }
    }
    else
    {
        /* The command is not recognized */
        outPacket->status = Bootloader_ERR_CMD;
    }
}

int main()
{
    Bootloader_Initialize();
    Bootloader_InitCallback(Bootloader_Callback);

    ...
}
```

Note This feature is not available for PSoC3 device family.

Code Examples

PSoC Creator provides numerous example projects that include schematics and example code in the Find Example Project dialog. For component-specific examples, open the dialog from the Component Catalog or an instance of the component in a schematic. For general examples, open the dialog from the Start Page or File menu. As needed, use the Filter Options in the dialog to narrow the list of projects available to select.

Refer to the "Find Example Project" topic in the PSoC Creator Help for more information.

There are more code examples online, check the Code Examples webpage here.

Cypress provides a number of application notes describing how PSoC can be integrated into your design. You can access the Application Notes search webpage here.

Check also the "References" section in this Datasheet.

Switching logic table

Case#	app#1		app#2		Classic Bootloader	Launcher
	Active	Valid	Active	Valid		
1	0	0	0	0	Stay in Bootloader	Stay in Launcher
2	0	0	0	1	Stay in Bootloader	Pass control to Combination app#2
3	0	0	1	0	Stay in Bootloader	Stay in Launcher
4	0	0	1	1	Pass control to Bootloadable app#2	Pass control to Combination app#2
5	0	1	0	0	Stay in Bootloader	Pass control to Combination app#1
6	0	1	0	1	Stay in Bootloader	Pass control to Combination app#1
7	0	1	1	0	Pass control to Bootloadable app#1, except if auto-application switching is disabled, then stay in Bootloader.	Pass control to Combination app#1
8	0	1	1	1	Pass control to Bootloadable app#2	Pass control to Combination app#2
9	1	0	0	0	Stay in Bootloader	Stay in Launcher
10	1	0	0	1	Pass control to Bootloadable app#2, except if auto-application switching is disabled, then stay in Bootloader.	Pass control to Combination app#2
11	1	0	1	0	Stay in Bootloader	Stay in Launcher

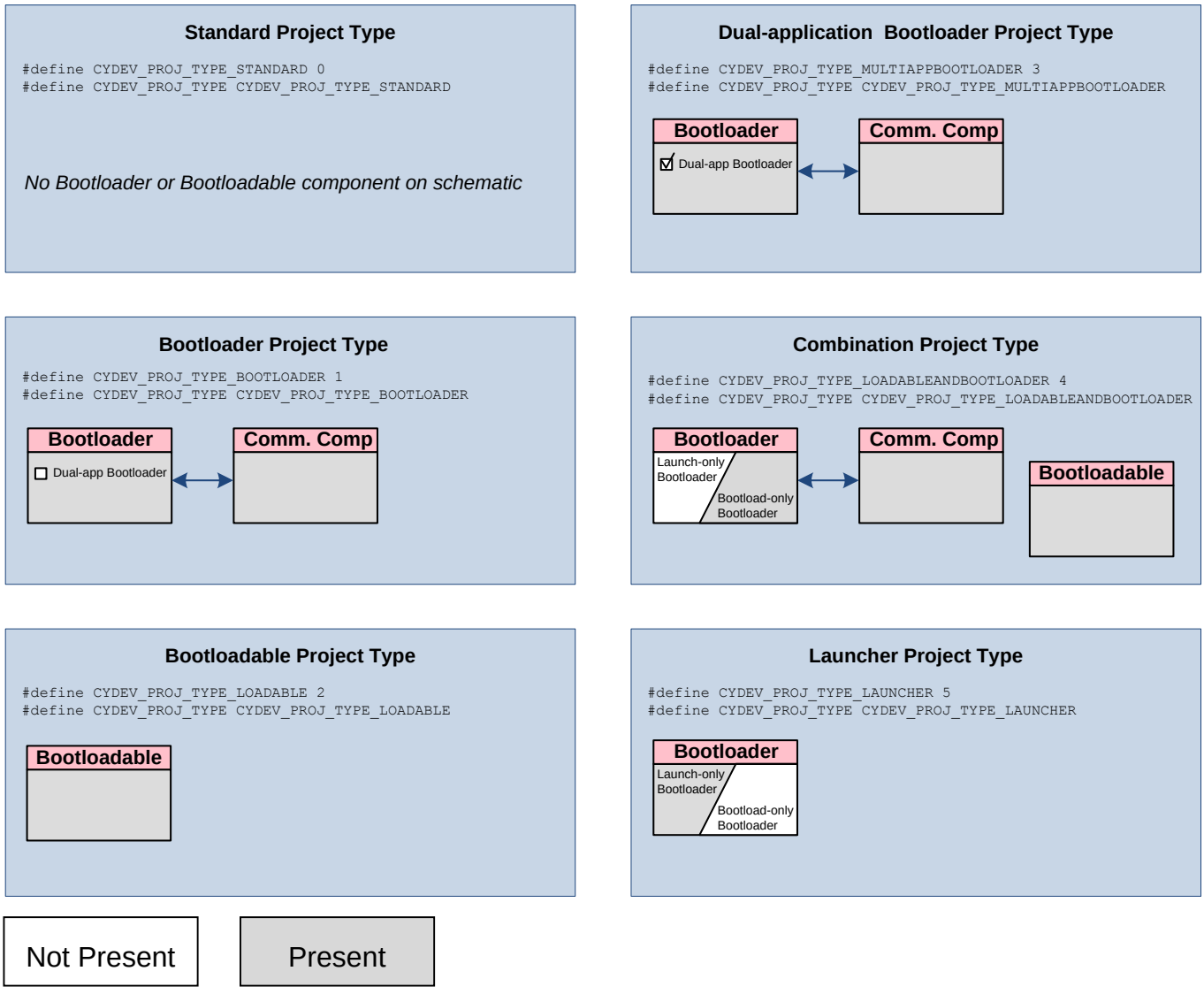
Case#	app#1		app#2		Classic Bootloader	Launcher
	Active	Valid	Active	Valid		
12	1	0	1	1	Pass control to Bootloadable app#2	Pass control to Combination app#2[8]
13	1	1	0	0	Pass control to Bootloadable app#1	Pass control to Combination app#1
14	1	1	0	1	Pass control to Bootloadable app#1	Pass control to Combination app#1
15	1	1	1	0	Pass control to Bootloadable app#1	Pass control to Combination app#1[8]
16	1	1	1	1	Pass control to Bootloadable app#1	Pass control to Combination app#1[8]

8 The invalidation of the non-active application in the metadata section is performed for this case, because there could be a situation due to some error, when both applications are set active (cases #12,15,16). Only one application can be active at the moment.

Functional Description

Bootloader Project Types

One of the new features provided with PSoC Creator 3.2 is an automatically detected project type. It is mainly related to the v1.40 update update of the Bootloader/Bootloadable components. The following diagram shows project type dependency on components on a schematic.



Note Launcher and Combination project types are supported for PSoC 4 and PSoC 5LP device families and are not supported for PSoC 3 device family.

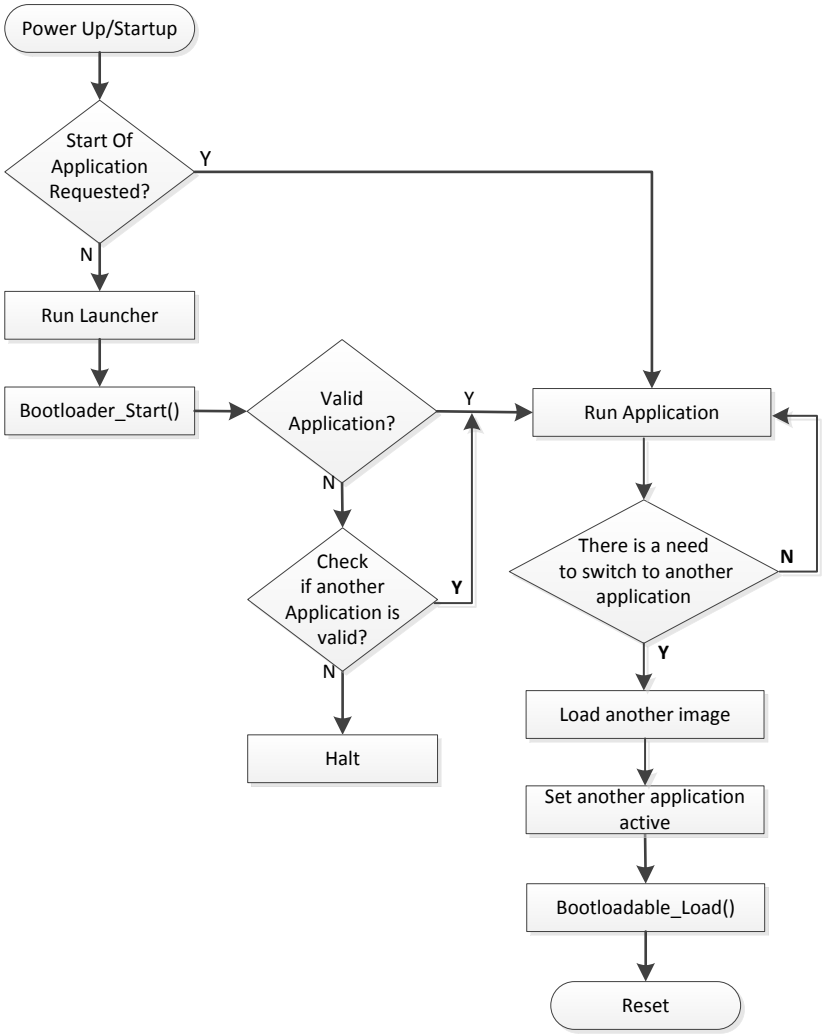
Bootloader and dual-application bootloader project types include features provided in the Bootloader v1.30 component. This functionality is called "Classic Bootloader" to separate it from the new launcher/combination architecture provided in the Bootloader v1.40 component.

Launcher/Combined Application Project Functions

The Launcher performs switching to the active and valid combined application.

Note No bootloading functionality is available for the Launcher. Using the Launcher makes sense only when the application has a Combination project type.

The Combined application is intended as the usual application that in addition is able to perform bootloading for another application. The simplified control flow of launcher and combination application interaction is below. For more details see Appendix A at the end of this document.



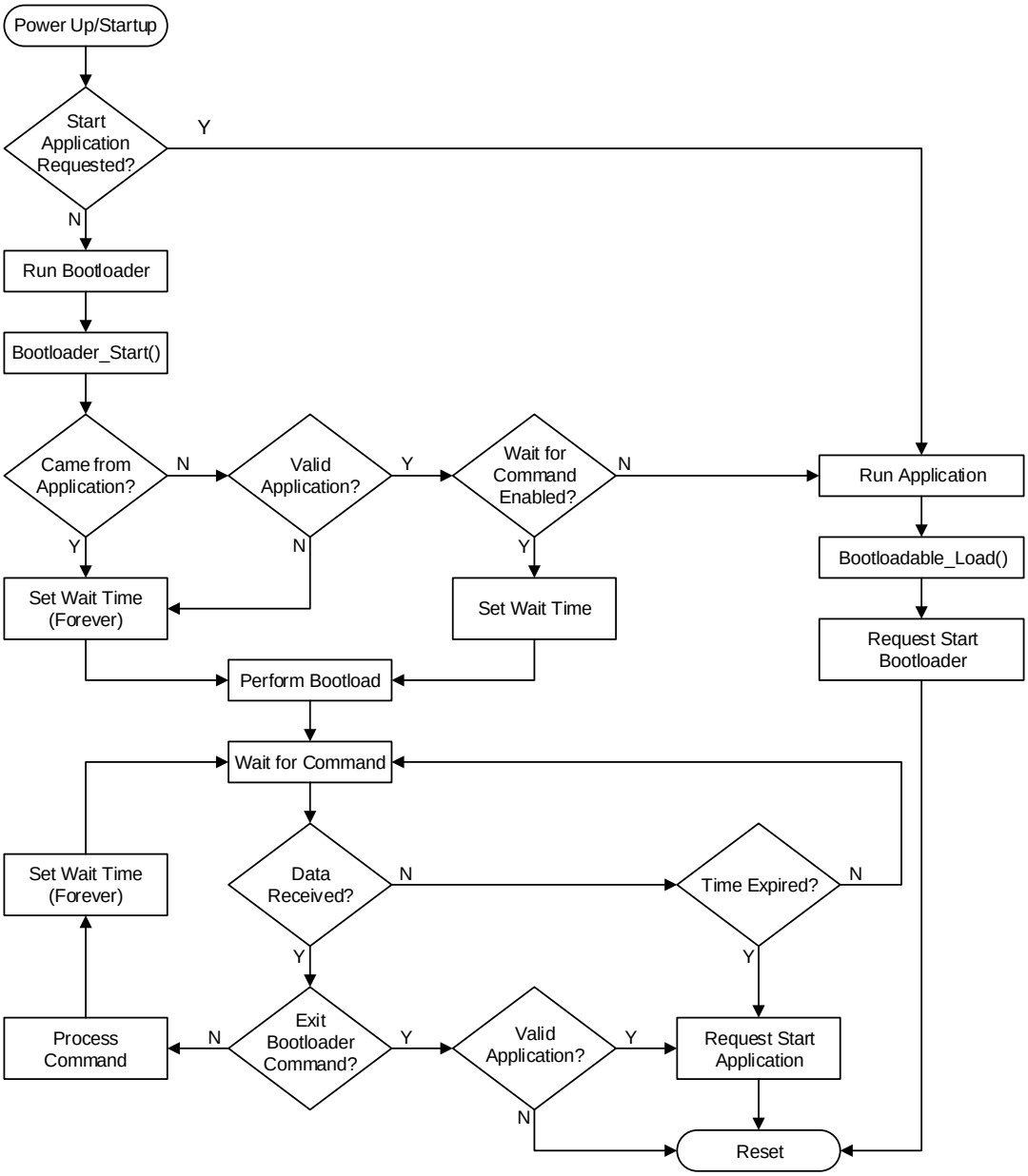
Classic Bootloader and Bootloadable Project Functions

The bootloader project performs overall transfer of a bootloadable project, or new code, to the flash via the bootloader project's communication component. After the transfer, the processor is always reset. At reset time, the bootloader project is also responsible for testing certain conditions and auto-initiating a transfer if the bootloadable project is nonexistent or corrupt.

At startup, the bootloader code loads configuration bytes for its own configuration. It must also initialize the stack and other resources as well as peripherals to do the transfer. When the transfer is complete, control is passed to the bootloadable project with a software reset.

The bootloadable project then loads configuration bytes for its own configuration and reinitializes the stack, as well as other resources and peripherals for its functions. The bootloadable project may call the Bootloadable_Load() function in the bootloadable project to switch to the bootloader application (this results in another software reset).

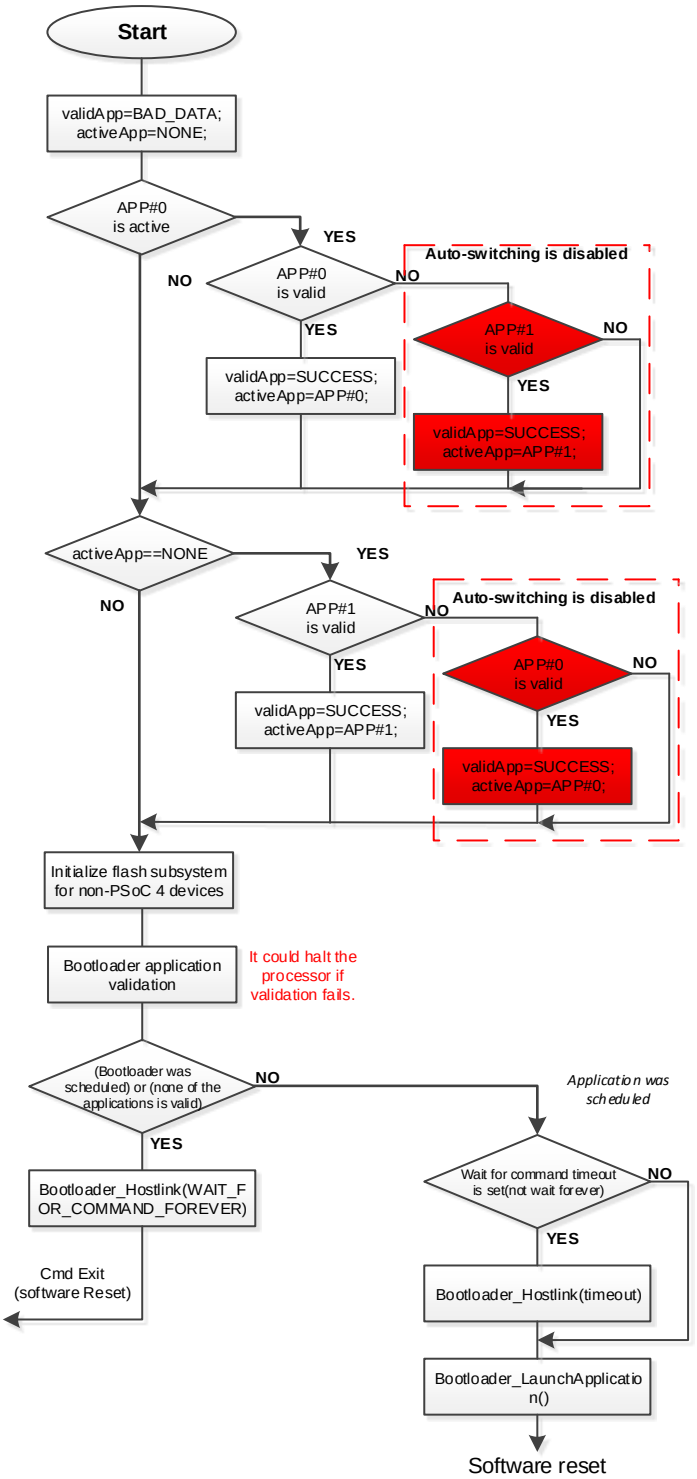
The following diagram shows how the Classic Bootloader for the single application works. The difference between single and dual-applications is in the valid application determination logic that is described in this flow.



When you have finished your development/test cycles and wish to create final images for your bootloader and associated bootloadable applications, make sure to recompile all of the relevant projects using the Release configuration of your IDE.

Bootloader Auto-Switching

The auto-switching feature is available only for the classic dual-application Bootloader and it is the option that disables automatic switching to another valid, but not active, application. In this case the control stays in the Bootloader until it obtains a valid application in which to switch.



Golden Image

The Golden image feature is also for the classic dual-application bootloader only (refusing to write to the certain image in order to retain the working image) is a single check if a given row number is within the following range: startRowNumber ≤ rowNumber ≤ endRowNumber), where startRowNumber is the Golden image's first row, endRowNumber is the Golden image's last row.

Note No bootloading functionality is available for the Launcher. Using the Launcher makes sense only with the application that has the combination project type.

The combined application is intended as a usual application that in addition is able to perform bootloading for another application.

The simplified control flow of launcher and combination applications interaction is below. For more detail see Appendix A.

Bootloader Application

You typically complete a bootloader design project by dragging a Bootloader component and communication component onto the schematic, routing the I/O to pins, setting up clocks, and so on. A project with a Bootloader and a communication component implements the basic bootloader application function of receiving new code and writing it to flash. You add custom functions to a basic bootloader project by dragging other components onto the schematic or by adding source code.

Bootloadable Application

The bootloadable application is actually the code. It is very similar to a normal application type. In this instance, the differences between a bootloadable application and a normal application include:

- a bootloadable is always associated with a bootloader
- a normal application is never associated with a bootloader.

Memory Usage

Bootloader

Normal and bootloader applications reside in flash starting at address zero.

Bootloadable

A bootloadable application occupies flash starting from the next empty flash row to the bootloader application.

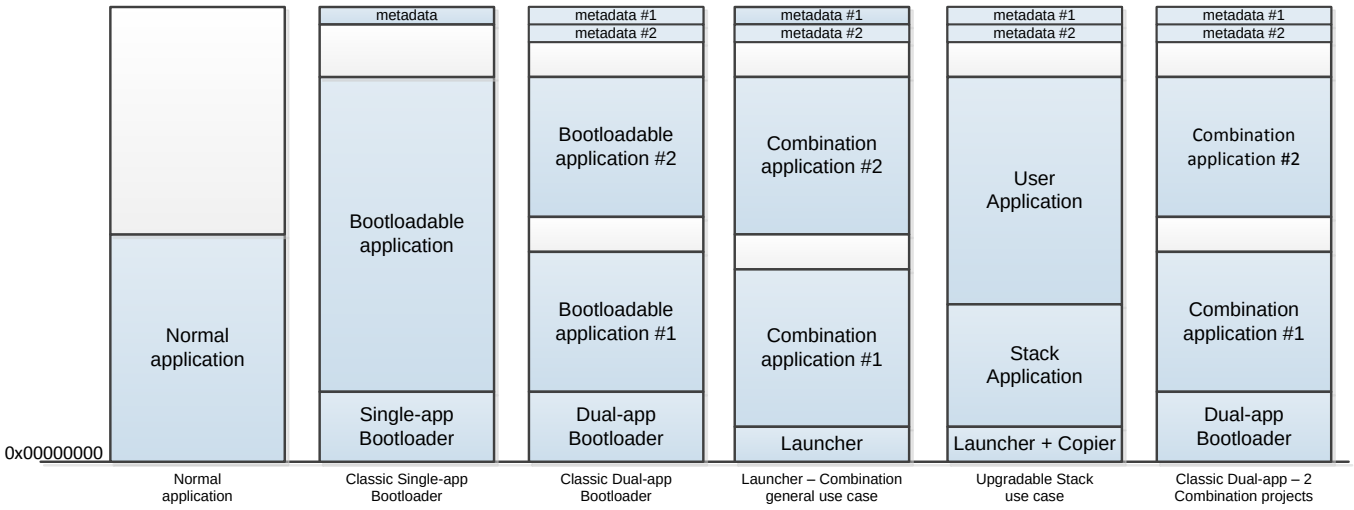
In case of a dual-application bootloader, the first bootloadable application resides above the bootloader application. The second bootloadable application occupies flash starting at the row that is halfway between the start of the first bootloadable application and the end of flash.

If the **Manual application image placement** option in the Bootloadable component Configure dialog is enabled, the bootloadable application is placed at an address specified by the **Placement address** option.

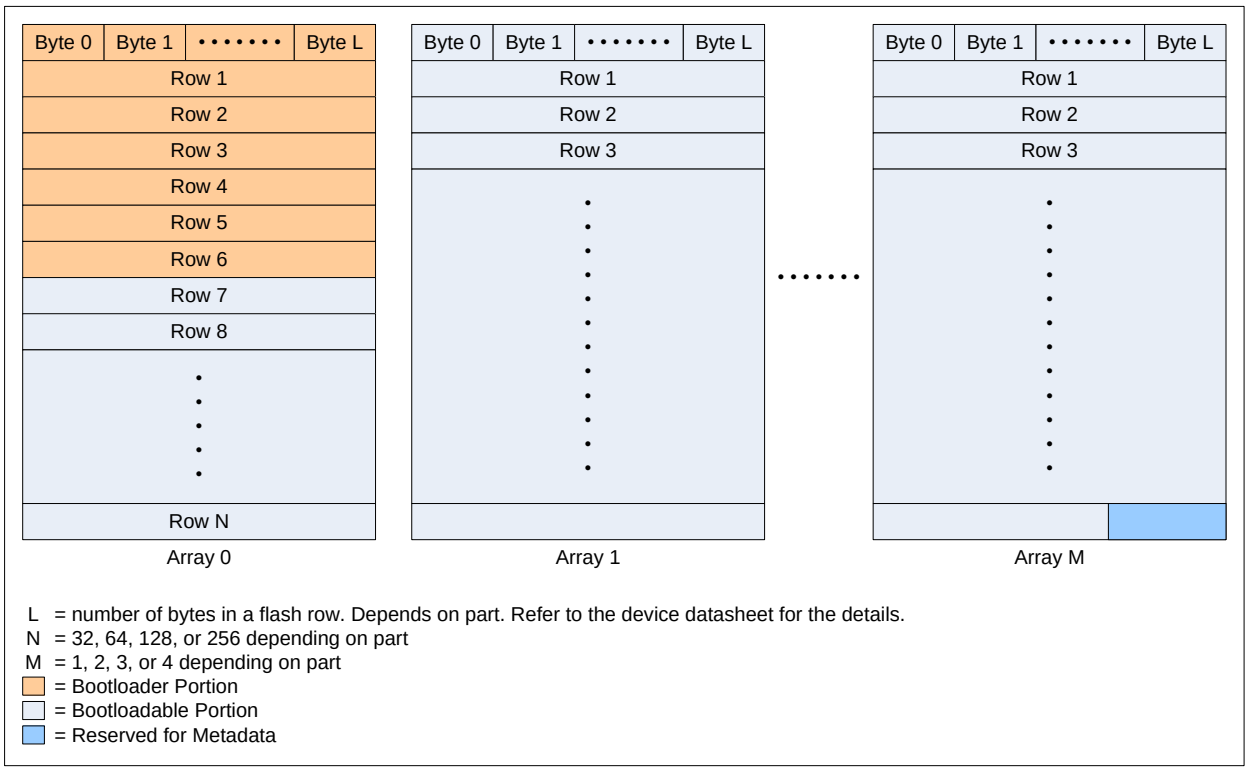
Note In case of a dual-application bootloader, the **Manual application image placement** and **Placement Address** options must be identical for both bootloadable applications.

Note In case of a dual-application bootloader, the **Manual application image placement** and **Placement Address** options are applicable only to the first bootloadable application. The second bootloadable application occupies flash starting at the row that is halfway between the start of the first bootloadable application and the end of flash.

The following diagram shows (from left to right) the memory usage of a normal application, bootloader and bootloadable applications, the dual-application bootloader with two bootloadable applications, and a combination application:



The following diagram shows the device's flash memory layout.



The bootloader project always occupies the bottom X flash rows. X is set so that there is enough flash for the following:

- The vector table for this project, starting at address 0 (except PSoC 3)
- The bootloader project configuration bytes
- The bootloader project code and data
- The checksum for the bootloader portion of flash

The bootloader project configuration bytes are always stored in main flash, never in ECC flash. The relevant option is removed from the bootloader project design-wide resource file.

The bootloader application portion of flash should be protected in the **Flash Protection** tab of the design-wide resource file to make it only be overwritten by downloading via JTAG / SWD.

The bootloadable project occupies flash starting at the first flash row size boundary after the bootloader, and includes:

- The vector table for the project (except PSoC 3)
- The bootloadable project code and data

- 64 bytes of data reserved at the last row (or 64 bytes of data at each of 2 last flash rows for Dual-app and Launcher use cases) of the last flash array to store metadata used by both the bootloader and bootloadable

The bootloadable project's configuration bytes may be stored in the same manner as in a standard project, that is, in either main flash or ECC flash, per settings in the Design-Wide Resources *<project.cydwr>* file.

Device-specific Details

PSoC 3

In PSoC 3, the only "exception vector" is the 3-byte instruction at address 0, which is executed at processor reset. (The interrupt vectors are not in flash – they are supplied by the Interrupt Controller [IC]). So at reset the PSoC 3 bootloader code simply starts executing from flash address 0.

PSoC 5LP and PSoC 4

In the PSoC 5LP / PSoC 4 devices, a table of exception vectors must exist at address 0. (The table is pointed to by the Vector Table Offset Register, at address 0xE000ED08, whose value is set to 0 at reset.) The bootloader code starts immediately after this table.

The table contains the initial stack pointer (SP) value for the bootloader project and the address of the start of the bootloader project code. It also contains vectors for the exceptions and interrupts to be used by the bootloader.

The bootloadable project also has its own vector table which contains that project's starting SP value and first instruction address. When the transfer is complete, as part of passing control to the bootloadable project, the value in the Vector Table Offset Register is changed to the address of the bootloadable project's table.

Metadata Memory Map

The metadata section is a 64-byte block of flash that is used as a common area for both bootloader and bootloadable applications. In the bootloader application, the metadata is placed at row N-1; in case of a dual-application bootloader, the bootloadable application number 1 uses row N-1, and application number 2 uses row N-2 to store its metadata, where N is the total number of rows for the selected device.

Address	PSoC 3	PSoC 4 / PSoC 5LP
0x00	Bootloadable Application Checksum	
0x01	Reserved	Bootloadable Application Start Routine Address
0x02		
0x03	Bootloadable Application Start Routine Address	
0x04		

0x05	Reserved	Last Bootloader Row
0x06		
0x07	Last Bootloader Row	Reserved
0x08		
0x09	Reserved	Bootloadable Application Length
0x0A		
0x0B	Bootloadable Application Length	
0x0C		
0x0D	Reserved	
0x0E		
0x0F		
0x10	Active Bootloadable Application	
0x11	Bootloadable Application Verification Status	
0x12	Bootloader Application Version	
0x13		
0x14	Bootloadable Application ID	
0x15		
0x16	Bootloadable Application Version	
0x17		
0x18	Bootloadable Application Custom ID	
0x19		
0x1A		
0x1B		
0x1C	Reserved	Copy Flag
0x1D	Reserved	Set by Creator to indicate that Copier is available.
0x1E	Reserved	
0x1F		
0x20	Reserved	Checksum exclude section size.
0x21		
0x22		
0x23		
0x24-0x3F	Reserved	

Name	Description
Bootloadable Application Checksum	This is the basic summation checksum that is computed by adding up all the bytes of the bootloadable application image (excluding the metadata section).
Bootloadable Application Start Routine Address	The startup routine address of the bootloadable application. This is STARTUP1 for PSoC 3 and Reset() for PSoC 4 / PSoC 5LP. The linker is free to put these anywhere it wants after the minimum starting address of the application.
Bootloader Last Flash Row	The number of the first flash row occupied by the application, minus 1. **Note** For the second application (in dual-application bootloader and launcher cases), this field contains the first flash row occupied by the second application, minus 1.
Bootloadable Application Length	The size of the bootloadable application in bytes.
Active Bootloadable Application	This field contains information about the active bootloadable application if the **Dual-application bootloader** option is enabled.
Bootloadable Application Verification Status	This field contains the status of the bootloadable application validation when the **Fast bootloadable application validation** option is enabled. The bootloader computes the checksum only for the first time and assumes that it remains valid in each future startup.
Bootloader Application Version	This field contains the application version of the bootloader application. Specified in the bootloader component customizer.
Bootloadable Application ID [9]	This field contains the application ID of the bootloadable application. Specified in the bootloadable component customizer.
Bootloadable Application Version [10]	This field contains the application version of the bootloadable application. Specified in the bootloadable component customizer.
Bootloadable Application Custom ID	This field contains the application custom ID of the bootloadable application. Specified in the bootloadable component customizer.
Copy flag	This field indicates a need to perform the “copy” operation. See Copier section.
Checksum exclude section size	The number of bytes that occupy the user data section that will not be part of the application checksum protection.

Note All fields are stored in the endianness of the processor: big-endian for PSoC 3 and little-endian for PSoC 4/PSoC 5LP.

9 When the bootloader application is the only application in the device (no bootloadable applications are stored), this field reports the number of images the bootloader application expects; 1 for bootloader projects, and 2 for dual-application bootloader projects.

10 For PSoC 3/PSoC 5LP, when the bootloader application is the only application in the device, this field reports whether ECC memory data should be included in the bootloadable application checksum.

PSoC Creator Project Output Files

When either project type – bootloader or bootloadable - is built, an output file is created for that project.

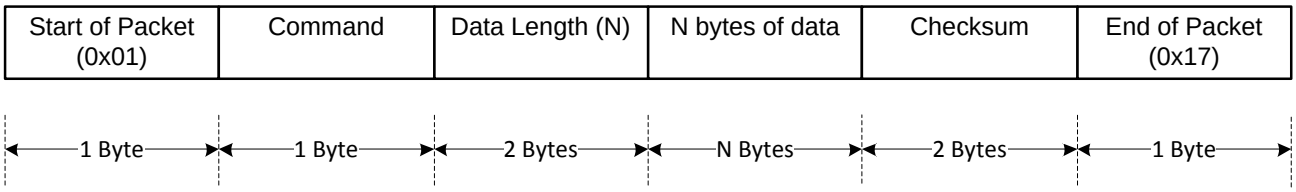
In addition, an output file for both projects, a "combination" file is created when the bootloadable project is built. This file includes both the bootloader and bootloadable projects. This file is typically used to facilitate downloading both projects (via JTAG / SWD) to device flash in a production environment.

Configuration bytes for bootloadable projects may be stored in either main flash or ECC flash. The format of the bootloadable project output file is such that when the device has ECC bytes which are disabled, transfer operations are executed in a shorter time. This is done by interleaving records in the bootloadable main flash address space with records in the ECC flash address space. The bootloader takes advantage of this interleaved structure by programming the associated flash row once – the row contains bytes for both main flash and ECC flash.

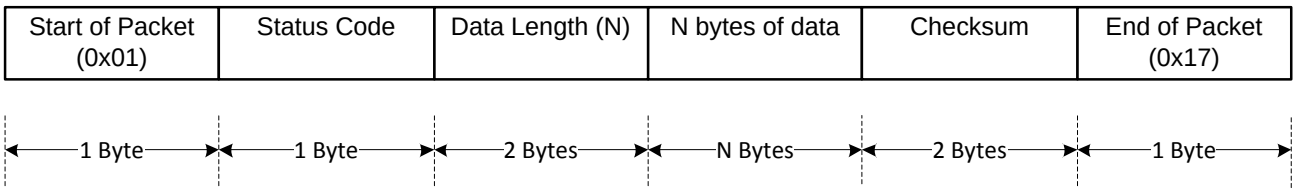
Each project has its own checksum. The checksums are included in the output files during project build time.

Bootloader Packet Structure

Communication packets sent from the Host to the Bootloader have the following structure:



Response packets read from the Bootloader have the following structure:



Status/Error Codes

The possible status/error codes output from the bootloader are:

Status/Error Code	Value	Description
CYRET_SUCCESS	0x00	The command was successfully received and executed.
BOOTLOADER_ERR_LENGTH	0x03	The amount of data available is outside the expected range.
BOOTLOADER_ERR_DATA	0x04	The data is not of the proper form.

Status/Error Code	Value	Description
BOOTLOADER_ERR_CMD	0x05	The command is not recognized.
BOOTLOADER_ERR_CHECKSUM	0x08	The packet checksum does not match the expected value.
BOOTLOADER_ERR_ARRAY	0x09	The flash array ID is not valid
BOOTLOADER_ERR_ROW	0x0A	The flash row number is not valid.
BOOTLOADER_ERR_APP	0x0C	The application is not valid and cannot be set as active.
BOOTLOADER_ERR_ACTIVE	0x0D	The application is currently marked as active or Golden image.
BOOTLOADER_ERR_CALLBACK	0x0E	The callback function returns invalid data.
BOOTLOADER_ERR_UNK	0x0F	An unknown error occurred.

Bootloader Commands

The bootloader supports the following commands. All received bytes that do not start with a command from the set of command bytes are discarded with no response generated. All multi-byte fields are output LSB first.

Note The maximum input/output packet length (for instance, Start-of-packet + Command + Data Length + Data + Checksum + End-of-packet) is currently limited to `Bootloader_SIZEOF_COMMAND_BUFFER` bytes. The minimum length is `Bootloader_MIN_PKT_SIZE` bytes.

Note The time required for the bootloader to execute any command is based on the configuration of the device. Some of the factors that affect the timing include:

- The clock speed at which the part is running
- The toolchain used to build the project
- The optimization settings used during the build
- The number of interrupts running in the background

Bootloader Command Name (Command Code)			
Data Byte (bytes number)	**Response Packet Status Code**	**Response Packet Data (bytes number)**	**Description**
Enter Bootloader (0x38)			
Security Key (6) (optional)	Success Error Command Error Data Error Length Error Checksum	Device JTAG ID (4) Device revision (1) Bootloader version (3)	The bootloader responds to this command with the device information and version of the Bootloader component. Version means the version of the Bootloader component.
Get Flash Size (0x32) (optional)			
Flash Array ID (1)	Success Error Command Error Data Error Length Error Checksum	First available row (2) Last available row (2)	The bootloader responds to this command with the first full row after the bootloader application (the first row of the bootloadable application) and the last flash row in the selected flash array. **Note** In versions prior to v1.40, there was an inconsistency in the returned value for different device families. For PSoC 3 device family Bootloader, the component actually returned that first full row. While for PSoC 4 and PSoC 5LP device families, it returned the last row of the Bootloader application. That is fixed in the Bootloader v1.40 component version, and the first full row after the Bootloader application is returned.
Program Row (0x39)			
Flash Array ID (1) Flash Row Number (2) Data to write (n)	Success Error Command Error Data Error Length Error Checksum Error Flash Row Error Active	N/A	Writes one row of flash data to the device. The data to be written to the flash can be sent in multiple packets using the **Send Data** command. This command may be sent along with the last block of data to program the row.
Erase Row (0x34) (optional)			
Flash Array ID (1) Flash Row Number (2)	Success Error Command Error Data Error Length Error Checksum Error Flash Row Error Active	N/A	Erases the contents of the provided flash row.

Bootloader Command Name (Command Code)			
Data Byte (bytes number)	**Response Packet Status Code**	**Response Packet Data (bytes number)**	**Description**
Get Row Checksum (0x3A) (optional)			
Flash Array ID (1) Flash Row Number (2)	Success Error Command Error Data Error Length Error Checksum	Row checksum (1)	Gets a 1-byte checksum for the contents of the provided row of flash.
Verify Application Checksum (0x31) (optional)			
N/A	Success Error Command Error Data Error Length Error Checksum	Checksum valid (1)	A non-zero return value indicates that the application code flash checksum matches the expected value stored in the flash and therefore the application is valid. A return value of 0 indicates that the checksums do not match and therefore the application is not valid.
Send Data (0x37) (optional)			
Data for Device (n)	Success Error Command Error Data Error Length Error Checksum	N/A	Sends a block of data to the device. This data is buffered up in anticipation of another command that will inform the bootloader what to do with the data. If multiple **Send Data** commands are issued back-to-back, the data is appended to the previous block. This command is used to breakup large transfers into smaller pieces to prevent bus starvation in some protocols.
Sync bootloader (0x35) (optional)			
N/A	N/A	N/A	Resets the bootloader to a clean state, ready to accept a new command. Any data that was buffered is thrown out. This command is only needed if the host and client get out of sync with each other.
Exit Bootloader (0x3B)			

Bootloader Command Name (Command Code)			
Data Byte (bytes number)	**Response Packet Status Code**	**Response Packet Data (bytes number)**	**Description**
N/A	N/A	N/A	Exits from the bootloader by triggering a software reset of the device. Before the software reset is executed, the bootloadable application is verified. If the application passes verification, the application is executed after software reset. If the application fails verification, then execution begins again with the bootloader after the software reset.
Get Metadata (0x03C) (optional)			
Application # (1)	Success Error Application Error Command Error Length Error Data Error Checksum	Metadata (56)	Reports the first 56 bytes of the metadata for a selected application. For more information on metadata see the Metadata section.
Get Application Status (Dual-application bootloader Only) (0x33) (optional)			
Application # (1)	Success Error Length Error Checksum Error Data	App # Valid (1) App # Active (1)	Returns the status of the specified application.
Set Active Application (Dual-application bootloader Only) (0x36)			
Application # (1)	Success Error Application Error Command Error Length Error Data Error Checksum	N/A	The specified bootloadable application is set as active. This command is used to switch between two bootloadable applications.
Verify Row (0x45) (optional)			

Bootloader Command Name (Command Code)			
Data Byte (bytes number)	**Response Packet Status Code**	**Response Packet Data (bytes number)**	**Description**
Flash array ID (1) Flash row number(2) Data to compare with the flash row (n)	Success Error Verify Error Command Error Data Error Length Error Data Error Checksum Error Flash Row	N/A	Compares data to one row of the device internal flash. The data to be written to the flash can be sent in multiple packets using the **Send Data** command.

Bootloader Application and Code Data File Format

The bootloader application and code data (.cyacd) file format stores the bootloadable portion of a design. The file is a header followed by lines of flash data. Excluding the header, each line in the .cyacd file represents an entire row of flash data. The data is stored as ASCII data in big-endian format.

The header record format is as follows:

[4-byte SiliconID][1-byte SiliconRev][1-byte Checksum Type]

The data records have this format:

[1-byte ArrayID][2-byte RowNumber][2-byte DataLength][N-byte Data][1-byte Checksum]

The checksum type in the header indicates the type of checksum used for packets sent between the bootloader host and the bootloader itself. The checksum in the data records is a basic summation computed by summing all bytes (excluding the checksum itself) and then taking the 2's complement.

Bootloader Host Tool

PSoC Creator ships with a bootloader host tool (*bootloader_host.exe*) that you can use to test the bootloader running on a PSoC chip. The bootloader host tool is the application that communicates directly with the bootloader to send new bootloadable images. The bootloader host tool provided is only a development and testing tool.

Source Code

In addition to the host executable itself, much of the source code used is also provided. Use this source code to create your own bootloader host applications. The source code is located in this directory:

<Install Dir>\cybootloaderutils

By default, this directory is:

C:\Program Files\Cypress\PSoC Creator\<Release Version>\PSoC Creator\cybootloaderutils

This source code is broken up into four different modules. These modules provide implementations for the various pieces of functionality required for a bootloader host. Depending on the desired level of control, some or all of these modules can be used in developing a custom bootloader host application.

cybtldr_command.c/h

This module handles construction of packets to send to the bootloader, and the parsing of packets received from the bootloader. It has a single function for constructing each type of packet that the bootloader understands, and a single function for parsing the results for each packet the bootloader can send back.

cybtldr_parse.c/h

This module handles the parsing of the *.cyacd file that contains the bootloadable image to send to the device. It has functions for setting up access to the file, reading the header, reading the row data, and closing the file.

cybtldr_api.c/h

This is a row level API that allows sending a single row of data at a time to the bootloader using a supplied communication mechanism. It has functions for setting up the bootload operation, programming a row, erasing a row, verifying a row, and ending the bootload operation.

cybtldr_api2.c/h

This is a higher level API that handles the entire bootload process. It has functions for programming the device, erasing the device, verifying the device, and aborting the current operation.

Resources

The Bootloader and Bootloadable projects use these device resources:

- The Bootloader component uses both general purpose bits of the reset status (RESET_SR0) register. These bits are necessary to communicate bootloader intents across the software reset boundaries.

- The resources used by the communication component are in the corresponding component datasheet.

Component Errata

This section lists known problems with the Bootloader component.

Cypress ID	Component Version	Problem	Workaround
224638	1.30, 1.40	The "Fast bootloable application validation" option does not work for PSoC 3 and PSoC 5LP devices.	This defect is fixed in version 1.50. If updating the Bootloader component to version 1.50 is not an option, add a CySetTemp() API call before starting the Bootloader component.
226826	1.40	If the 'Initialize EEPROM in HEX file' option was enabled in the PSoC 3 / PSoC 5LP bootloadable project, the bootloader component will fail to receive a complete *.cyacd file. This will most likely damage the bootloadable image stored in flash.	Update to the latest version of the Bootloader / Bootloadable component.

233909	1.40, 1.50	Launcher launches invalid application on PSoC 5LP in Launcher+Combination project type.	Initialize flash subsystem in main for non PSoC 4 devices before calling Bootloader_Start. Example:
			`if (CYRET_SUCCESS != CySetTemp())`
			`{`
			`  CyHalt(0x00u);`
			`}`
			`Bootloader_Start();`

241749	1.40, 1.50	Turning power OFF and ON quickly for several minutes while bootloading may corrupt some data in the flash. This may cause the bootload operation to fail and have to be restarted.	Connect stable power to the device Vdd and Vss pins. For details, refer to the device datasheet as well as AN61290 or AN88619, PSoC Hardware Design Considerations.

Component Changes

This section lists the major changes in the component from the previous version.

Version	Description of Changes	Reason for Changes / Impact
1.40.c	Updated datasheet	Added errata section to document defects 233909 and 241749.
1.40.b	Updated datasheet.	Added errata section to document defect 224638. Added errata section to document defect 226826.
1.40.a	Update datasheet.	Fixed a few typos.
1.40	Removed mention of application type.	No need to select the application type. PSoC Creator does it automatically.
	Renamed multi-application to dual-application.	To provide updated interface information.
	Added the Definition section.	To provide information about terminology.
	Added a description of project types.	To provide information about new features.
	Added descriptions of Golden image support, Auto application switching, and Security key.	To provide updated interface information.
	Updated the description of the section Bootloader application validation	To fix a mistake with the checksum location.
	Updated the Functional Description section with launcher project and combined project functionality.	To update according to the new functionality
	Added the following functions: Bootloader_Initialize() Bootloader_HostLink() Bootloader_GetRunningAppStatus() Bootloader_GetActiveAppStatus() Bootloadable_SetActiveApplication() Bootloadable_GetActiveApplication()	To increase functionality.
	Updated the **Enter Bootloader** command (0x38). Renamed the existing **Verify Row** command (0x3A) as **Get Row Checksum**. Added new **Verify Row** command (0x45).	To provide updated interface information.
	Added ‘Copier’ option.	For BLE OTA Upgradable Stack Example project.
	Added Callback function section.	New feature.
	Added Debugging Bootloader/Bootloadable Projects section.	To provide information about debugging Dual-app Bootloader cases.

Version	Description of Changes	Reason for Changes / Impact
	The command 'Get Flash Size' returns: ▪ The first full row after Bootloader application for Classic Bootloader use case. ▪ The last flash row in the selected flash array. In versions prior to v1.40, there was an inconsistency in the returned value (a) for different device families. For PSoC 3 device family, the Bootloader component actually returned that first full row. While for PSoC 4 and PSoC 5LP device families, it returned the last row of the Bootloader application. That is fixed in the Bootloader v1.40 component version, and the first full row after Bootloader application is returned.	The inconsistency was fixed.
	The following new use cases are added: - Launcher and 2 Combinations; - Upgradable Stack use case; - Classic Dual-app Bootloader with 2 Combination applications;	New features.
	Checksum exclude section feature added. This allows to store the data that are not a part of a checksum protection mechanism. So they could change without breaking an image checksum.	New feature
1.30	Updated the list of supported communication components.	To provide updated interface information.
	Aligned diagram in the Bootloader and Bootloadable Project Functions section with the implementation.	Bootloadable application validation is performed from the bootloader application before switching to it through the software reset.
	Added System Builder support.	To support the new feature in PSoC Creator.
	Added following functions to the Bootloader component: uint32 Bootloader_GetMetadata(uint8 field, uint8 appId) cystatus Bootloader_ValidateBootloadable(uint8 appId) void Bootloader_Exit(uint8 appId) uint8 Bootloader Calc8BitSum(uint32 baseAddr, uint32 start, uint32 size)	Increased functionality.
	Updated Bootloader_Start() for the Multi-Application Bootloader.	To implement the following algorithm: If active bootloadable application is not valid, and the other bootloadable application (inactive) is valid, the last one is started.

Version	Description of Changes	Reason for Changes / Impact
	Fixed an issue when **Verify Row** command was always available independently of the customizer settings.	
	Implemented additional verification.	To make sure that bootloader application is not overwritten during bootloadable application transfer.
	Updated the **Get Flash Size** command implementation.	To address incorrect reply when bootloader application consumes more than one flash array (its size is above 64 KB).
	The flash initialization for the PSoC 3 and PSoC 5LP devices updated to be performed only before flash write.	The startup time (time between reset and main() entry) significantly decreased.
1.20.a	Minor datasheet edits.	Added note for PSoC 4000 devices and flash.
1.20	The **Wait for command time** option was changed to be in units of 100 ms instead of 10 ms units.	**Note** While updating to version 1.20 the **Wait for command time** option value will be automatically increased by 10 times.
	Added **Get Metadata** command.	Reports first 56 bytes of the metadata for a selected application.
	All commands (with the exception of **Exit Bootloader**, and **Sync Bootloader**) are ignored by Bootloader application till the **Enter Bootloader** command is received.	Bootloader application waits for valid traffic (denoted by **Enter Bootloader** command), but not for any traffic. If traffic is received but not a valid bootloader **Enter Bootloader** command, then the timeout expires at the specified time and the bootloadable application is launched.
	Updated the **Dependencies** tab.	Added field to specify Bootloader ELF file.
	Updated *MISRA Compliance* section.	The Bootloader/Bootloadable components were verified for MISRA compliance and have specific deviations described.
1.10	Added *MISRA Compliance* section.	The Bootloader/Bootloadable components were not verified for MISRA compliance.
	Added PSoC 4 device support.	New device support
	Minor datasheet edits	
1.0.a	Datasheet corrections	
1.0	Initial component version	

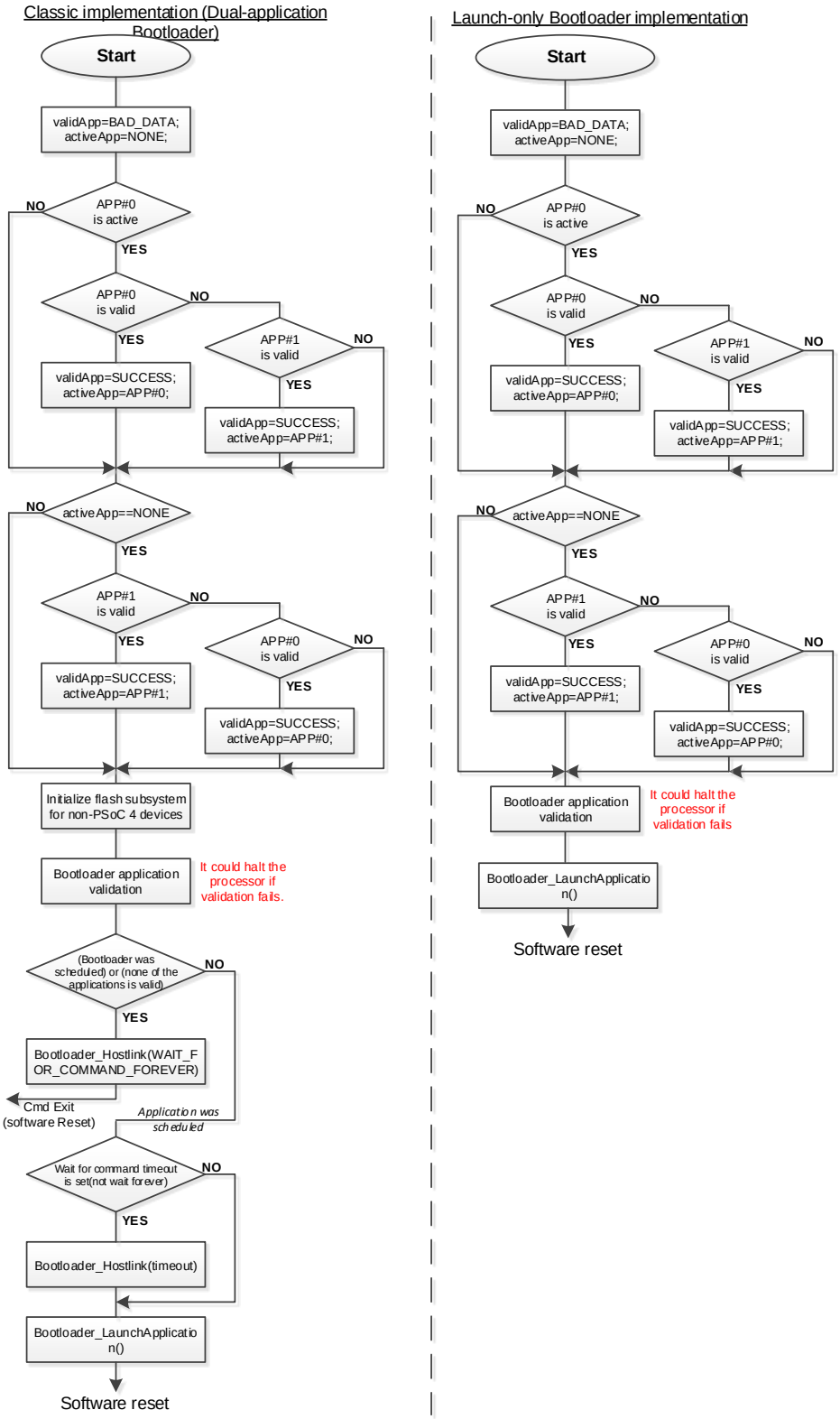
© Cypress Semiconductor Corporation, 2015-2016. This document is the property of Cypress Semiconductor Corporation and its subsidiaries, including Spansion LLC (“Cypress”). This document, including any software or firmware included or referenced in this document (“Software”), is owned by Cypress under the intellectual property laws and treaties of the United States and other countries worldwide. Cypress reserves all rights under such laws and treaties and does not, except as specifically stated in this paragraph, grant any license under its patents, copyrights, trademarks, or other intellectual property rights. If the Software is not accompanied by a license agreement and you do not otherwise have a written agreement with Cypress governing the use of the Software, then Cypress hereby grants you a personal, non-exclusive, nontransferable license (without the right to sublicense) (1) under its copyright rights in the Software (a) for Software provided in source code form, to modify and reproduce the Software solely for use with Cypress hardware products, only internally within your organization, and (b) to distribute the Software in binary code form externally to end users (either directly or indirectly through resellers and distributors), solely for use on Cypress hardware product units, and (2) under those claims of Cypress’s patents that are infringed by the Software (as provided by Cypress, unmodified) to make, use, distribute, and import the Software solely for use with Cypress hardware products. Any other use, reproduction, modification, translation, or compilation of the Software is prohibited.

TO THE EXTENT PERMITTED BY APPLICABLE LAW, CYPRESS MAKES NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS DOCUMENT OR ANY SOFTWARE OR ACCOMPANYING HARDWARE, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. To the extent permitted by applicable law, Cypress reserves the right to make changes to this document without further notice. Cypress does not assume any liability arising out of the application or use of any product or circuit described in this document. Any information provided in this document, including any sample design information or programming code, is provided only for reference purposes. It is the responsibility of the user of this document to properly design, program, and test the functionality and safety of any application made of this information and any resulting product. Cypress products are not designed, intended, or authorized for use as critical components in systems designed or intended for the operation of weapons, weapons systems, nuclear installations, life-support devices or systems, other medical devices or systems (including resuscitation equipment and surgical implants), pollution control or hazardous substances management, or other uses where the failure of the device or system could cause personal injury, death, or property damage (“Unintended Uses”). A critical component is any component of a device or system whose failure to perform can be reasonably expected to cause the failure of the device or system, or to affect its safety or effectiveness. Cypress is not liable, in whole or in part, and you shall and hereby do release Cypress from any claim, damage, or other liability arising from or related to all Unintended Uses of Cypress products. You shall indemnify and hold Cypress harmless from and against all claims, costs, damages, and other liabilities, including claims for personal injury or death, arising from or related to any Unintended Uses of Cypress products.

Cypress, the Cypress logo, Spansion, the Spansion logo, and combinations thereof, PSoC, CapSense, EZ-USB, F-RAM, and Traveo are trademarks or registered trademarks of Cypress in the United States and other countries. For a more complete list of Cypress trademarks, visit cypress.com. Other names and brands may be claimed as property of their respective owners.



Appendix A. Classic vs. Launcher Bootloader Comparison

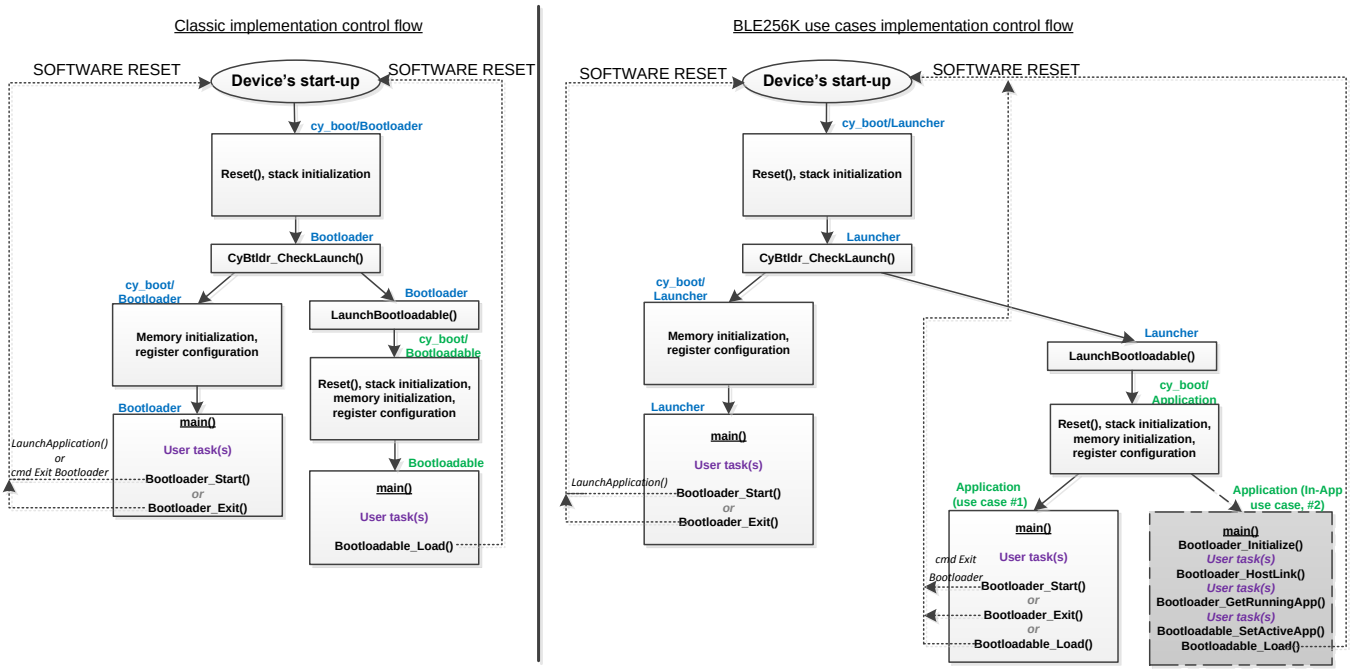


CyBtldr_CheckLaunch() is a type of jumper that chooses either Bootloader or Application(s) to load.

Note For classic implementation (up to Bootloader/Bootloadable component v1.30), the Bootloader component is intended for the Bootloader project type only, while for the Bootloader component v1.40, the Bootloader can also be present in an application (Combination project type) only for the purpose of bootloading.

Bootloader_Start() is a central API that provides launching and bootloading functionality for the classic implementation. Entering Bootloader_Start() API gives no provision for doing any other tasks. Switching to the application by means of a software reset occurs either automatically (in case both active and valid applications are determined and no need for bootloading), or if no valid application is available, then the Bootloader component waits "forever" until the valid application is downloaded and a command for exit for the bootloader is received.

The following is a comparative diagram for the Classic Bootloader flow and BLE256K use cases update flow.



For BLE256K use cases, the Bootloader_Start() API is divided separately into launching and bootloading parts of functionality by compile-time options. Bootloader_Start() in a launcher project type is intended only for the launching functionality. Calling Bootloader_Start() in a combination project type results in starting a bootloading process that lasts until the **Exit Bootloader** command is received.

For BLE256K In-Application use case, Bootloader_Start() is substituted by the following APIs:

- Bootloader_Initialize()
- Bootloader_HostLink()

- Bootloader_GetRunningAppStatus()
- Bootloader_GetActiveAppStatus()
- Bootloadable_SetActiveApplication()
- Bootloadable_GetActiveApplication()

They give a possibility to interleave bootloading with user tasks. Bootloader_Initialize() performs the initialization for the internal variables that keep active the application number, the number of the running application, and the status of bootloading initialization. Bootloader_HostLink() is made public to communicate with the host when there is a need to receive a new image or respond with status information. The communication protocol can be used for other user's purposes. Note that it is a user responsibility to care about performing the bootloading process (keep communication link, enabling/disabling interrupts during programming to flash).