# Bootloader Library Help

MPLAB Harmony Integrated Software Framework

# Bootloader Library Help

This section describes the Bootloader Library that is available in MPLAB Harmony.

## Introduction

This library provides a software Bootloader Library that is available on the Microchip family of microcontrollers with a convenient C language interface.

### Description

The Bootloader Library can be used to upgrade firmware on a target device without the need for an external programmer or debugger.

A demonstration application, which can be downloaded into the target PIC32 device using the bootloader is included, which provides a personal computer host application to communicate with the bootloader firmware running inside the PIC32 device. This personal computer application is used to perform erase and programming operations.

This library was developed based on the Microchip application note, AN1388 *"PIC32 Bootloader"* (DS01388).

#### Bootloader Theory

A bootloader is a small application that starts the operation of the device. A bootloader does not fully operate the device, but can perform various functions prior to starting the main application. Such functions can include:

- System checks
- Firmware upgrades
- Application integrity verification
- Starting the application

## Using the Library

This topic describes the basic architecture of the Bootloader Library and provides information and examples on its use.

### Description

**Interface Header File**: `bootloader.h`

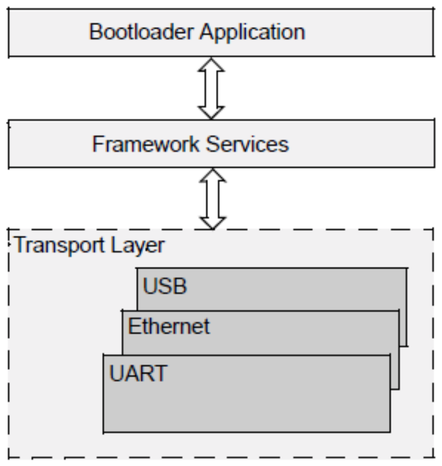
The interface to the Bootloader Library is defined in the `bootloader.h` header file. Any C language source (`.c`) file that uses the Bootloader Library should include `bootloader.h`.

## Abstraction Model

This library provides the low-level abstraction of the Bootloader Library module on the Microchip family of microcontrollers with a convenient C language interface. This topic describes how that abstraction is modeled in the software and introduces the library interface.

### Description

**Bootloader Software Abstraction Block Diagram**



### Bootloader Application

The main purpose of the Bootloader Application layer is to implement the callback functions necessary to enhance the bootloader. The various callback functions are described below, and are used to handle such tasks as forcing Bootloader mode, erasing and programming memory, and any cleanup before jumping to the application.

### Framework Services

The Framework Services provide the main tasks of handling Flash memory. These include:

- Erasing Flash memory
- Programming hex file records into Flash memory
- Computing a CRC check of the Application in Program Memory
- Jumping to the Application

The Framework Services can be enhanced with the callback functions that can handle such things as external memory devices. Doing so requires that the hex files allocate the external memory assignments in a region that would not normally fall into the physical Flash region in the PIC32 device.

The Framework Services take an interface-agnostic approach to the actual communication medium. For the most part, it does not care whether the data comes from a PC Host via USB, UART, or Ethernet, or if it is coming by reading a file from a USB thumb drive or a SD Card. This allows the bootloader to be expanded to handle other communication interfaces with minimal impact to the Framework Services.

### Transport Layer

The Transport Layer, called Datastream in the Bootloader Framework code, is used to implement the specific interface to the source of the hex file.

## Library Overview

The Library Interface routines are divided into various sub-sections, which address one of the blocks or the overall operation of the Bootloader Library module.

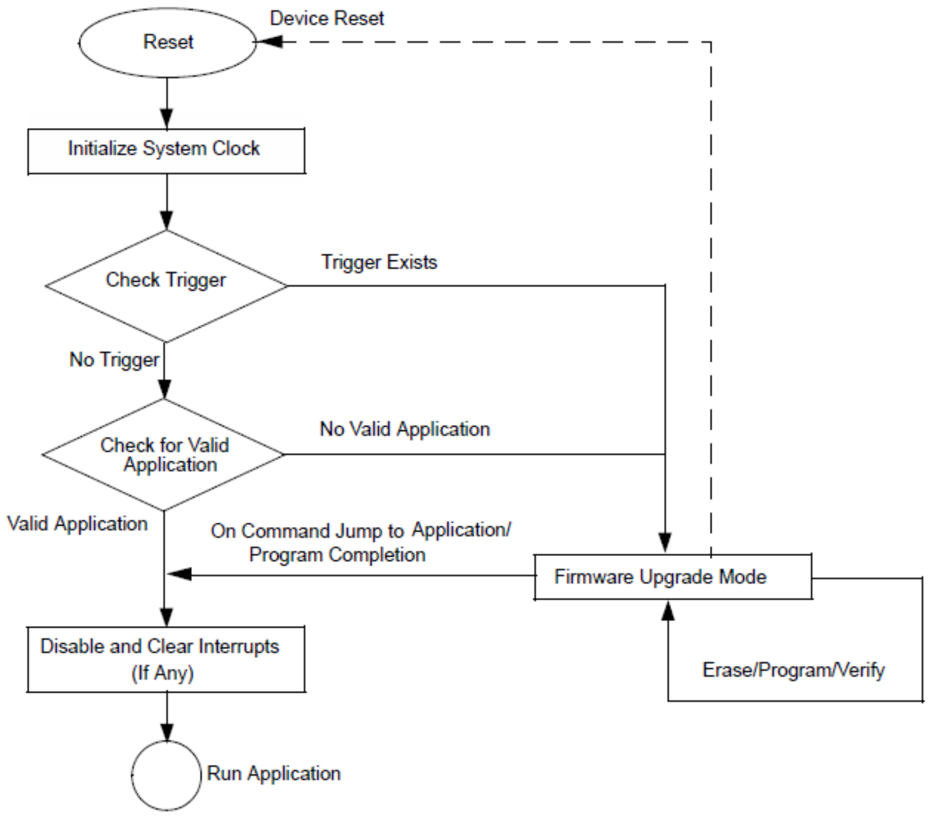
## How the Library Works

This topic provides information on how the library works.

### Description

The bootloader library is implemented using a framework. The bootloader firmware communicates with the personal computer host application by using a predefined communication protocol.

The following figure illustrates the bootloader architecture. The bootloader framework provides several API functions, which can be called by the bootloader application and the data stream layer. The bootloader framework assists the user to easily modify the bootloader application to adapt to different requirements. Refer to the Library Interface section for the available APIs.



## *Basic Flow of the Bootloader*

Describes the basic flow of the PIC32 Bootloader

## Description

The Bootloader code starts executing on a device Reset. If there are no conditions to enter the firmware upgrade mode, the Bootloader starts executing the user application. The Bootloader performs Flash erase/program operations while in the firmware upgrade mode.

### Entering the Firmware Upgrade Mode

On a device Reset, the Bootloader forces itself into the firmware upgrade mode if the content of the user application’s reset vector address is erased. On PIC32 starter kits, press and hold the switch, SW3, during power-up. While in firmware upgrade mode, the LED labeled LED1 on the PIC32 starter kit will blink.

For your custom bootloader, call the BOOTLOADER_ForceBootloadRegister function to register a custom function. From the callback function, the bootloader can check whatever situation is necessary for forcing the bootloader into Bootloader mode.

### Exiting the Firmware Upgrade Mode

For USB HID, Ethernet, or the UART Bootloader, the firmware upgrade mode can be exited either by applying a hard Reset to the device, or by sending a “Jump to Application” command from the PC. For the USB Flash drive Bootloader, the firmware upgrade mode is exited either by a hard Reset or upon completion of firmware programming.

**Note:** The Bootloader should disable and clear any enabled interrupts before running the user application. The stray interrupts from the Bootloader may interfere with the user application and cause the application to fail. If applicable, interrupts and peripherals should be reinitialized in the user application.

## *Bootloader Placement in Memory*

Describes the Bootloader placement in memory.

### Description

The placement of the Bootloader in Flash memory controls several elements of the application and Bootloader interaction. Both have to be placed such that the application will not overwrite the Bootloader, and the Bootloader can properly program the application when it is downloaded.

Figure 1 illustrates two schemes for the Bootloader placement based on the size of the Bootloader. Devices with a large enough Boot Flash memory can place all of the Bootloader within Boot Flash. Fitting the Bootloader within the Boot Flash memory provides the complete program Flash memory for the user application.

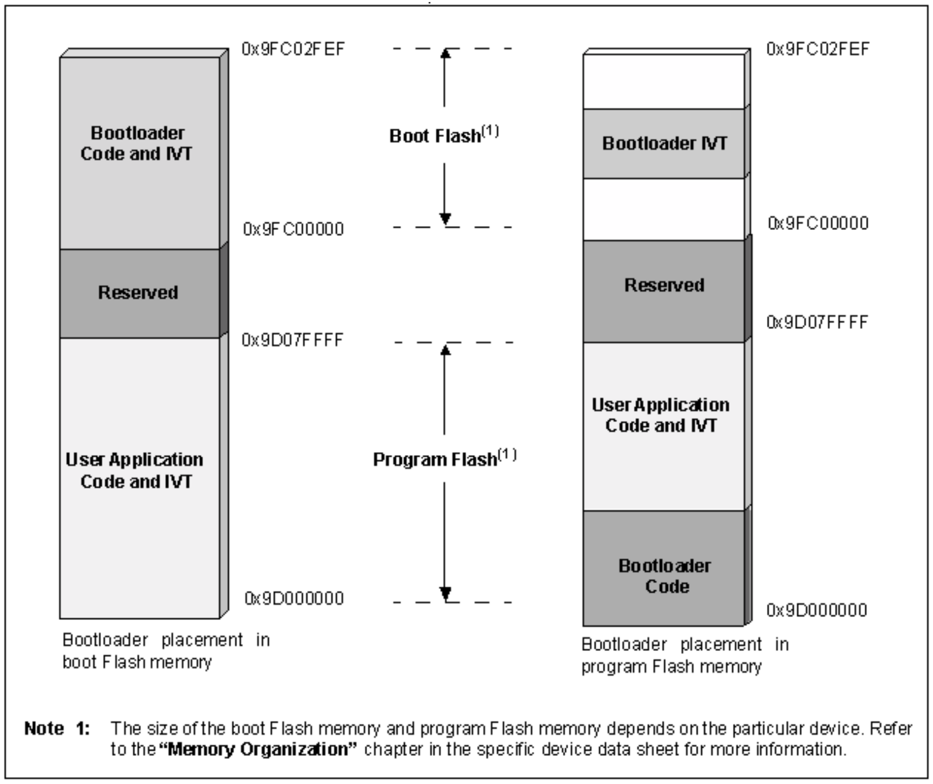
In the case of Bootloaders that exceed the size of PIC32 boot Flash, the Bootloader is split into two parts:

- For devices with 3 KB of Boot Flash, only the reset vector is placed in Boot Flash, with the Interrupt Vector Table (IVT) and C start-up code placed in Program Flash.
- For devices with 12 KB of Boot Flash, the Interrupt Vector Table (IVT) and the C start-up code are placed in Boot Flash, and the remaining portion of the Bootloader is placed inside the Program Flash.

The portion of the Bootloader that resides in Program Flash may be placed at either the beginning or the end of the Program Flash, depending on the needs of the application. The linker scripts generated by Harmony place the Bootloader code at the beginning of Program Flash (0x9D000000).

For PIC32MZ devices, with two 80 KB Boot Flash panels, the Bootloader may or may not fit entirely in one Boot Flash panel. In order to fit some of the Bootloaders, the linker script makes the two Boot Flash panels look like one contiguous Boot Flash memory. Unimplemented areas are blocked using a fill command to the linker.

**Figure 1: Bootloader Placement**


0x9FC02FEF
Bootloader Code and IVT
0x9FC00000
Reserved
0x9D07FFFF
User Application Code and IVT
0x9D000000
Bootloader placement in boot Flash memory
Boot Flash(1)
Program Flash(1)
0x9FC02FEF
Bootloader IVT
0x9FC00000
Reserved
0x9D07FFFF
User Application Code and IVT
Bootloader Code
0x9D000000
Bootloader placement in program Flash memory
Note 1: The size of the boot Flash memory and program Flash memory depends on the particular device. Refer to the "Memory Organization" chapter in the specific device data sheet for more information.

## *Handling Device Configuration Bits*

Provides information on how device Configuration bits are handled.

### Description

The Bootloader does not erase or write the device Configuration words while programming the new application firmware. This is because the device Configuration words settings are shared by both the Bootloader and the user application. Any modification to the device Configuration words may make the Bootloader non-functional. Therefore, it is highly recommended to have common device Configuration words settings for both the user application and the Bootloader.

When combining the Bootloader and Application Hex files in MPLAB X IDE, an error will be generated if the device Configuration words are different. This will be shown as a data conflict error, and the address given will match an address in the device Configuration words. This will indicate which word is in conflict, so that it can be resolved.

The Bootloader's Configuration words can be imported into the Application's MHC configuration file, thus simplifying the process of resolving differences.

## *Demonstration Application*

Provides information on the demonstration applications that can be used with the Bootloader.

### Description

Refer to the following sections for information on the demonstrations that are available for the Bootloader:

- *Volume I: Getting Started With MPLAB Harmony > Applications Helps > Bootloader Demonstrations*
- *Volume I: Getting Started With MPLAB Harmony > Applications Help > Examples > Peripheral Library Examples > dma_led_pattern*

### PIC32MX Bootloaders

For PIC32MX bootloaders, the MHC configuration for the corresponding application must match the bootloader for the interface in use. MHC will adjust the linker script and certain bootloader definitions in source code to maximize the use of the Flash memory. For example, a UART bootloader must be matched with an application configured for UART bootloading. A USB Device bootloader must be matched with an application configured for USB Device bootloading.

To change the type of bootloader interface the demonstration application is configured for, do the following:

1. Open the dma_led_pattern project in MPLAB X IDE.
2. Select the appropriate configuration for the target device.
3. Open MHC.
4. From the Options tab, expand *Harmony Framework Configuration > Bootloader Library*, and select **Use Bootloader Library?**
5. Select the correct interface using the drop-down list next to Bootloader Type.
6. Regenerate the code.
7. Compile the code.
8. The generated `.hex` file can now be used with the corresponding bootloader.

## *Using the Bootloader Application (UART, USB HID, and Ethernet Bootloaders)*

Provides information on using the Bootloader application.

### Description

Refer to *Volume I: Getting Started With MPLAB Harmony > Applications Help > Bootloader Demonstrations > Demonstrations > basic > Running the Demonstration* for information on running the demonstration application.

## *Bootloader Communication Protocol (UART, USB HID, and Ethernet)*

Provides information on the Bootloader communication protocol.

### Description

The PC host application uses a communication protocol to interact with the Bootloader firmware. The PC host application acts as a master and issues commands to the Bootloader firmware to perform specific operations.

## Frame Format

The communication protocol follows the frame format, as shown in Example 1. The frame format remains the same in both directions, that is, from the host application to the Bootloader, and from the Bootloader to the host application.

**Example 1: Frame Format**

```
[<SOH>…]<SOH>[<DATA>…]<CRCL><CRCH><EOT>
Where:
 <...> Represents a byte
 [...] Represents an optional or variable number of bytes
```

The frame starts with a control character, Start of Header (SOH), and ends with another control character, End of Transmission (EOT). The integrity of the frame is protected by two bytes of Cyclic Redundancy Check (CRC)-16, represented by CRCL (low-byte) and CRCH (high-byte).

## Control Characters

Some bytes in the Data field may imitate the control characters, SOH and EOT. The Data Link Escape (DLE) character is used to escape such bytes that could be interpreted as control characters. The Bootloader always accepts the byte following a <DLE> as data, and always sends a <DLE> before any of the control characters.

**Table 1: Control Character Descriptions**

| Control | Hex Value | Description |
|---|---|---|
| `<SOH>` | 0x01 | Marks the beginning of a frame. |
| `<EOT>` | 0x04 | Marks the end of a frame. |
| `<DLE>` | 0x10 | Data link escape. |

## Commands

The PC host application can issue the commands listed in Table 2 to the Bootloader. The first byte in the data field carries the command.

| Command Value in Hexadecimal | Description |
|---|---|
| 0x01 | Read the Bootloader version information. |
| 0x02 | Erase the Flash. |
| 0x03 | Program the Flash. |
| 0x04 | Read the CRC. |
| 0x05 | Jump to the application. |

*Read Bootloader Version Information*

The PC host application request for version information to the Bootloader is shown in Example 2.

**Example 2: Request**

```
[<SOH>…]<SOH>[<0x01>]<CRCL><CRCH><EOT>
```

The Bootloader responds to the PC request for version information in two bytes, as shown in Example 3.

**Example 3: Response**

```
[<SOH>…]<SOH><0x01><MAJOR_VER><MINOR_VER>
<CRCL><CRCH><EOT>

Where:
MAJOR_VER = Major version of the Bootloader
MINOR_VER = Minor version of the Bootloader
```

*Erase Flash*

On receiving the erase Flash command from the PC host application, the Bootloader erases that part of the program Flash, which is allocated for the user application. The request frame from the PC host application to the Bootloader is shown in Example 4.

**Example 4: Request**

```
[<SOH>…]<SOH><0x02><CRCL><CRCH><EOT>
```

The response frame from the Bootloader to the PC host application is shown in Example 5.

**Example 5: Response**

```
[<SOH>…]<SOH><0x02><CRCL><CRCH><EOT>
```

*Program Flash*

The PC host application sends one or multiple hex records in Intel Hex format along with the program Flash command. The MPLAB XC32 C/C++ Compiler generates the image in the Intel Hex format. Each line in the Intel hexadecimal file represents a hexadecimal record. Each hexadecimal record starts with a colon (:) and is in ASCII format. The PC host application discards the colon and converts the remaining data from ASCII to hexadecimal, and then sends the data to the Bootloader. The Bootloader extracts the destination address and data from the hex record, and writes the data into program Flash.

The request frame from the PC host application to the Bootloader is shown in Example 6.

**Example 6: Request**
```
[<SOH>…]<SOH><0x03>[<HEX_RECORD>…]<CRCL>
<CRCH><EOT>
```

Where, `HEX_RECORD` is the Intel Hex record in hexadecimal format.

The response from the Bootloader to the PC host application is shown in Example 7.

**Example 7: Response**
```
[<SOH>…]<SOH><0x03><CRCL><CRCH><EOT>
```

*Read CRC*

The read CRC command is used to verify the content of the program Flash after programming. The request frame from the PC host application to the Bootloader is shown in Example 8.

**Example 8: Request**
```
[<SOH>…]<SOH><0x04><ADRS_LB><ADRS_HB>
<ADRS_UB><ADRS_MB><NUMBYTES_LB><NUMBYTES_HB>
<NUMBYTES_UB><NUMBYTES_MB><CRCL><CRCH><EOT>
```

`ADRS_LB`, `ADRS_HB`, `ADRS_UB` and `ADRS_MB`, as shown in Example 8, represent the 32-bit Flash addresses from where the CRC calculation begins.

`NUMBYTES_LB`, `NUMBYTES_HB`, `NUMBYTES_UB` and `NUMBYTES_MB`, as shown in Example 8, represent the total number of bytes in 32-bit format for which the CRC is to be calculated.

The response from the Bootloader to the PC host application is shown in Example 9.

**Example 9: Response**
```
[<SOH>…]<SOH><0x04><FLASH_CRCL><FLASH_CRCH>
<CRCL><CRCH><EOT>
```

*Jump to Application*

The Jump to Application command from the PC host application commands the Bootloader to execute the application. The request frame from the PC host application to the Bootloader is shown in Example 10.

Example 10: Request
```
[<SOH>…]<SOH><0x05><CRCL><CRCH><EOT>
```

There is no response to this command from the Bootloader because the Bootloader immediately exits the firmware upgrade mode and begins executing the application.

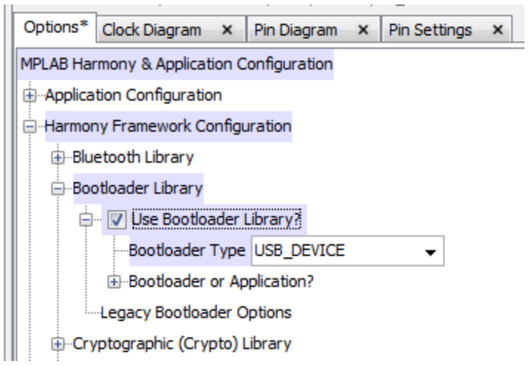
## *Generating the Application Linker Script*

Provides information on generating the application linker script.

## Description

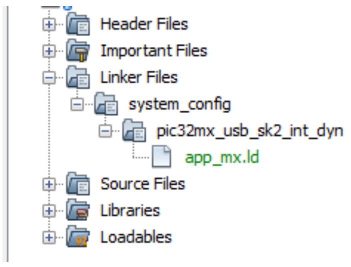
The MPLAB Harmony Configurator (MHC) can generate an application linker script specifically matched to the bootloader type in use. To add the linker script to your application, do the following:

1. In MPLAB X IDE, start MHC, and open the configuration for your application.
2. Navigate to *Harmony Framework Configuration > Bootloader Library* and select **Use Bootloader Library?**, as shown in the following figure.


Options*
Clock Diagram
Pin Diagram
Pin Settings
MPLAB Harmony & Application Configuration
Application Configuration
Harmony Framework Configuration
Bluetooth Library
Bootloader Library
Use Bootloader Library?
Bootloader or Application?
Legacy Bootloader Options
Cryptographic (Crypto) Library

Bootloader Type USB_DEVICE

3. Select the bootloader type in use using the Bootloader Type drop-down. For PIC32MX devices, this option must match between Bootloader and Application configurations. The linker scripts and reset address are configured for each based on the size of the bootloader. Therefore, they must be the same for the bootloader to look in the correct location for the application, and jump to the application's reset address.
4. When the application code is generated, a linker script will be created and inserted into the project folder, as shown in the following figure.



## *Considerations While Moving the Application Image*

Describes the procedure to place a user application into a desired program Flash memory region

### Description

This section describes the procedure to place a user application into a desired program Flash memory region. It must be ensured that the user application's memory region does not overlap with the memory region reserved for the Bootloader.

The bootloader generated by MHC should be considered a starting point for your product's bootloader. As such, adding new features may cause the bootloader to exceed the size intended in the bootloader linker script. In addition, the bootloaders provided in the `apps/bootloader/basic` folder are configured to the `-O1` compiler optimization. It is not the smallest possible size for the bootloader, and turning on the `-Os` or MIPS16/microMIPS code options will reduce the size of the bootloader. If the size of the bootloader changes, the following steps should be performed to adjust both the bootloader and the application in order to make sure that both fit and make best use of the device memory.

1. Determine the new ending address in Program Flash for the bootloader. This can be done either by using the `.map` file generated by the MPLAB XC32 C/C++ Compiler, or by using the ELFViewer plugin for MPLAB X IDE.
2. Round the address up to the nearest page boundary. For devices with 1K page sizes, the address must end on a 0x400 boundary. For devices with 4K page sizes, the address must end on a 0x1000 boundary.
3. Within the bootloader, change the following files:
   - In `btx_mx.ld`, change the length of kseg0_program_mem to the new boundary
   - In `system_config.h`, change the value for BOOTLOADER_FLASH_BASE_ADDRESS to match the virtual address determined previously
4. Within the application, change the linker script, `app_mx.ld`, as follows:
   - Change _ebase_address to the new boundary, plus enough pages to get the exceptions to start at an address on a 4K (0x1000) boundary. This is to align it on the boundary required by the MIPS core.
   - Change _RESET_ADDR to the new boundary previously determined
   - Change the ORIGIN value for kseg0_program_mem to the new boundary
5. Recompile both the bootloader and the application, and test operations.

For PIC32MZ devices, any of the bootloaders can fit entirely into Boot Flash, although some currently take advantage of both panels of Boot Flash. This is done by making the two Boot Flash memories appear as one panel in the linker script, but using a fill command to the linker to block out the area that is not available. When the resulting file is loaded by MPLAB X IDE, you may receive the following warning:

*Warning: C:/microchip/harmony/core/apps/bootloader/basic/firmware/basic.X/dist/udp_pic32mz_ef_sk/production/basic.X.production.hex contains code that is located at addresses that do not exist on the PIC32MZxxx.*

This warning can be ignored, as it is a warning about the fill memory.

📝 **Note:** For more information on linker script memory regions, refer to the *"MPLAB® XC32 Assembler, Linker and Utilities User's Guide"* (DS50002186) and the *"MPLAB® XC32 C/C++ Compiler User's Guide"* (DS50001686).

### *Bootloader Configurations*

Provides information on the macros that are available to configure the Bootloader code during compilation.

### Description

Most of the configuration options for the bootloaders will be done with the driver or middleware-specific options available in MHC. For example:

- UART - Adjusting the port used and the baud rate
- USB - The value of the Vendor and Product IDs
- Ethernet - The default MAC IDs and IP Address to use
- SD Card - The SPI interface that communicates with the SD card

The following table lists macros that are modified in source code only, and their usage. Depending on user requirements, the values in these macros may need to be changed.

**General Macros**

| Macro | Usage |
|---|---|
| `BOOTLOADER_FLASH_BASE_ADDRESS` | Base address of the program Flash reserved for the user application. The address value must point to the beginning of a Flash page, which is dependent upon the Flash page size of the specific device.<br>Note that setting this value to an address below the maximum memory will allow your bootloader to reserve Flash memory as needed. |
| `BOOTLOADER_FLASH_END_ADDRESS` | End address of the program Flash reserved for the user application. The address value must point to the end of a Flash page. |
| `BOOTLOADER_RESET_ADDRESS` | Address of user Reset vector. The Bootloader branches to this address when it must run the user application. |
| `MAJOR_VERSION` | Major version of the Bootloader firmware. |
| `MINOR_VERSION` | Minor version of the Bootloader firmware. |

## Configuring the Library

The configuration of the Bootloader Library is based on the file `system_config.h`.

This header file contains the configuration selection for the Bootloader Library. Based on the selections made, the Bootloader Library will or will not support selected features. These configuration settings will apply to all instances of the Bootloader Library.

This header can be placed anywhere; however, the path of this header needs to be present in the include search path for a successful build. Refer to the Applications Help section for more details.

## Building the Library

This section lists the files that are available in the Bootloader Library.

### Description

The following three tables list and describe the header (`.h`) and source (`.c`) files that implement this library. The parent folder for these files is `<install-dir>/framework/bootloader`.

## Interface File(s)

This table lists and describes the header files that must be included (i.e., using `#include`) by any code that uses this library.

| Source File Name | Description |
|---|---|
| `/src/bootloader.h` | Includes all MPLAB Harmony-compatible function calls for the Bootloader Library. |

## Required File(s)

***All of the required files listed in the following table are automatically added into the MPLAB X IDE project by the MHC when the library is selected for use.***

This table lists and describes the source and header files that must *always* be included in the MPLAB X IDE project to build this library.

| Source File Name | Description |
|---|---|
| `<install-dir>framework/bootloader/src/bootloader.c` | This file contains the core implementation of the Bootloader Library. |
| `<install-dir>framework/bootloader/src/datastream.c` | This file contains the data stream implementation of the Bootloader Library. In addition, a `datastream_<type>.c` file is included with this file. For example, `datastream_usart.c`. |

## Optional File(s)

This table lists and describes the source and header files that may optionally be included if required for the desired implementation.

| Source File Name | Description |
|---|---|
| `datastream_sdcard.c` | Data stream implementation when using the SD Card bootloader configuration. |
| `datastream_udp.c` | Data stream implementation when using the Ethernet UDP bootloader configuration. |
| `datastream_usart.c` | Data stream implementation when using the USART bootloader configuration. |
| `datastream_usb_hid.c` | Data stream implementation when using the USB HID Device bootloader configuration. |
| `datastream_usb_host.c` | Data stream implementation when using the USB Host bootloader configuration. |

## Module Dependencies

The Bootloader Library will depend on the modules used for the communication medium, as listed in the following table.

| Bootloader | Module Dependency and MHC Configuration |
|---|---|
| SD Card | SD Card Driver, configured to use the SPI port and Chip Select lines in use.<br>File System, configured with the following MHC options:<br>• Use File System Auto Mount Feature<br>• Media Type: SYS_FS_MEDIA_TYPE_SD_CARD<br>• FAT File System |
| USART | USART Static Driver |
| UDP | TCP/IP, configured with the following MHC options:<br>• IPv4<br>• Use UDP<br>• One Network Configuration, with interface, host name, and IP address as desired<br>• Enable Stack Deinitialization Operations<br>• Enable Configuration Save/Restore Functionality<br>• Dynamic RAM Size: 20250 |

| | |
|---|---|
| USB Device | USB, configured with the following MHC options:<br>• USB Device<br>• Number of Endpoints Used: 2<br>• Number of Functions: 1<br>• Device Class: HID<br>• Vendor ID: 0x04d8<br>• Product ID: 0x003c<br>• Manufacturer String: As desired<br>• Product String: As desired |
| USB Host | USB, configured with the following MHC options:<br>• USB Host (Recommended)<br>• Use MSD Host Client Driver<br>File System, configured with the following MHC options:<br>• Use File System Auto Mount Feature<br>• Media Type: SYS_FS_MEDIA_TYPE_MSD |

## Bootloader Sizing and Optimization

Provides bootloader sizing and optimization information.

### Description

The example bootloaders provided in the `apps/bootloader/basic` folder have optimization settings for use by most MPLAB XC32 C/C++ Compiler users, which is `-O1`. However, in terms of size, this option does not produce the most optimal code. The following table lists the various bootloaders, and their current size (based on Boot Flash size).

**Flash Usage Based on Optimization in Bytes**

| Peripheral | Device Family | -O1 | -Os | -Os MIPS16 | -Os microAptiv |
|---|---|---|---|---|---|
| SD Card | PIC32MZ(1) | 134968 | 122496 | N/A | 52110 |
| UART | PIC32MX(1) | 7376 | 6892 | 5696 | N/A |
| UART | PIC32MZ(1) | 12796 | 12228 | N/A | 10460 |
| USB Device | PIC32MX | 24944 | 22932 | 15328 | N/A |
| USB Device | PIC32MZ(1) | 38104 | 34976 | N/A | 26164 |
| USB Host | PIC32MX | 76156 | 66968 | 42808 | N/A |
| USB Host | PIC32MZ(1) | 85512 | 75944 | N/A | 53444 |
| UDP | PIC32MX | 73740 | 72416 | 48804 | N/A |
| UDP | PIC32MZ(1) | 90704 | 79856 | N/A | Not Functional(2) |

**Notes:**
1. The Bootloader resides entirely in Boot Flash. No Program Flash is in use.
2. The UDP bootloader does not compile for PIC32MZ devices when microMIPS is selected.

### Application Interrupt Vector Table Placement

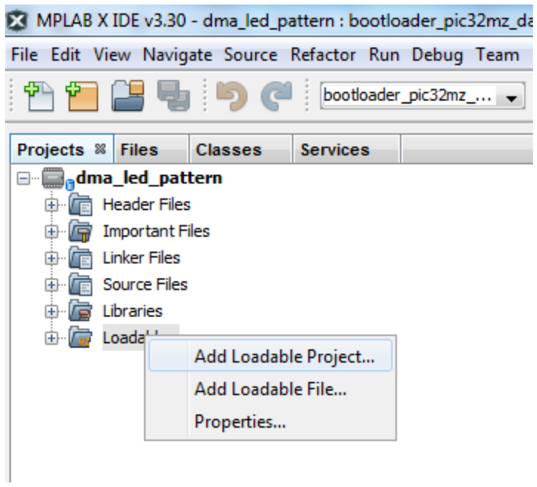
Beginning in MPLAB Harmony v1.08, the application linker scripts are able to have the interrupt vector table within the kseg0_program_mem, along with the rest of the application code. Generally, this means that it can be placed anywhere within the application code, and only the _ebase_address value in the application linker script needs to be updated.

Within the MIPS core, the base address of the interrupt vector table is set by the Ebase register (CP0 Register 15, Select 1). Because of the definition of this register, the interrupt vector table must be aligned on a 4K boundary. This applies to all PIC32 devices. The bootloader will take the start of application Program Memory as the start of the application itself. This means that the interrupt vector table will need to be located at least at the next 4K boundary after that starting address.

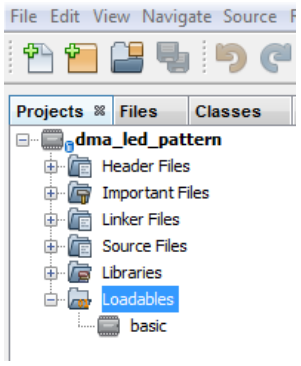
### Debugging with a Bootloader and an Application

When developing a system that uses both an application and a bootloader, it is useful to be able to run both as separate projects, but have them both in the device. This involves setting the bootloader as a loadable project to the application. The steps to add the bootloader as the loadable project, are as follows.

1. Right-click the application project's `Loadables` folder, and select **Add Loadable Project…**.



2. Navigate to the project folder of the Bootloader, selecting the correct configuration if there are multiple configurations, and click **Add**.



3. When the program is built, it will generate a single hex file that contains the combined Bootloader and application.
4. When debugging the program, both will be programmed into the PIC32. However, the Bootloader will be executed, and the PIC32 will halt when it reaches the main function of the application. To step through the Bootloader, it is necessary to set a breakpoint in the bootloader code.

### Commonly Encountered Situations

When creating the Bootloader and the application, you may encounter some of these situations while combining the two, and errors may be generated by the linker.

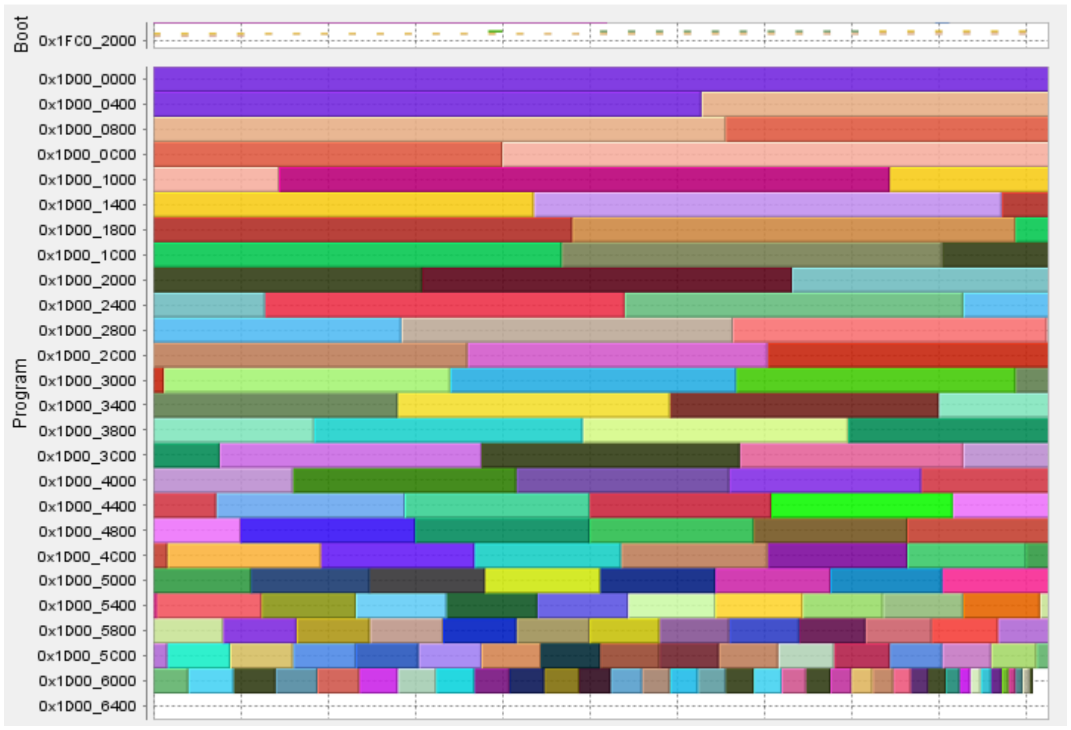
## Optimizing the Bootloader

Provides information for optimizing the Bootloader.

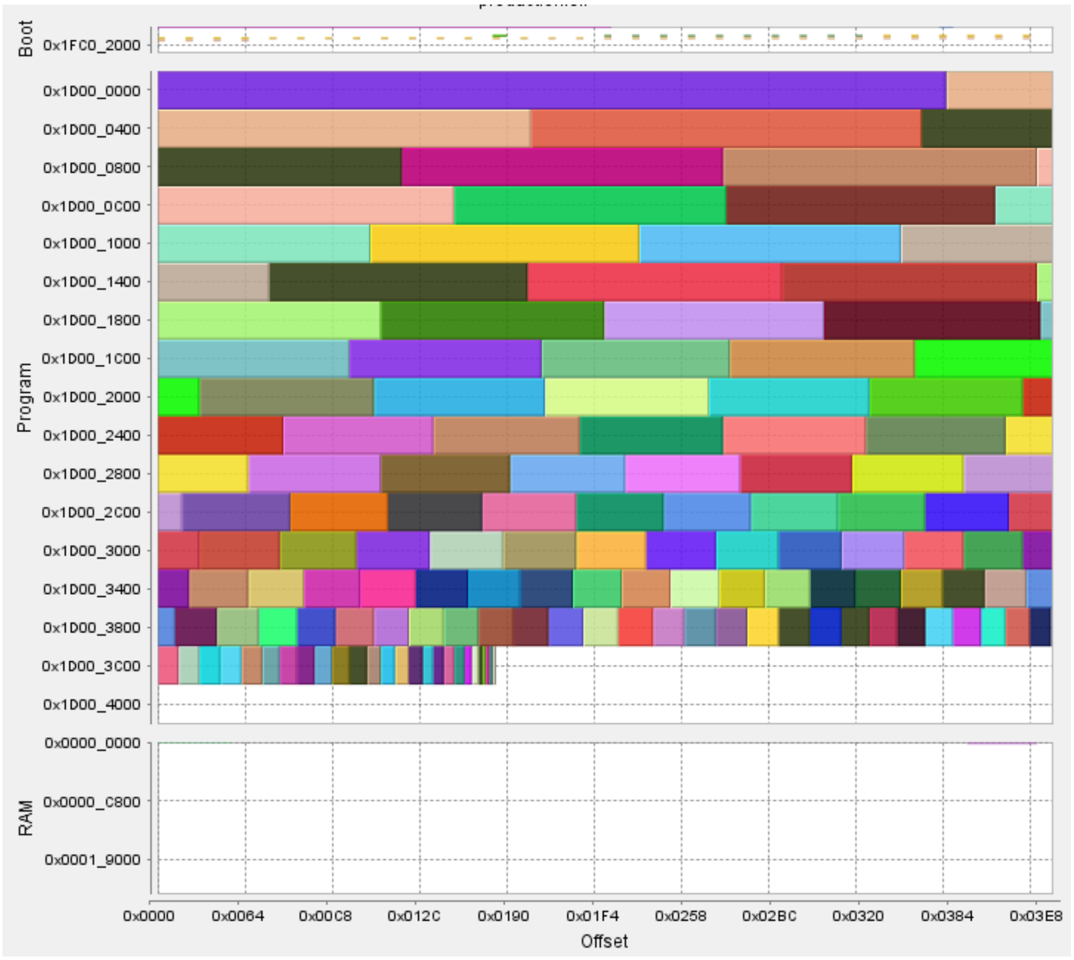
### Description

By selecting a different optimization level, or by adding features to the Bootloader, the size of the Bootloader will change. These changes may require adjustments to the linker scripts of the Bootloader and the Application.

The following example will take a PIC32MX USB Device Bootloader, and set optimizations to save space. When the Bootloader is compiled with the default, `-O1`, optimization, it takes 24944 bytes of Flash. This can be seen in the following figure, which graphically shows the mapping of the Bootloader functions and Flash usage.



When the optimizations are set to `-Os`, and MIPS16 code is generated, the Bootloader code size shrinks to 15328 bytes, as shown in the following figure.



Because the Bootloader now ends at a lower address in Program Flash, the space can now be reclaimed for the application. Due to Flash page sizing, the application linker script must take the ending address of the Bootloader, and round it up to the start address of the next page. In this case, the new start address for the application can be 0x9D004000.

The MIPS32 core requires that Interrupt Vector Tables (IVTs) be placed on a 4 KB boundary, Therefore, the IVT for the Application can now be placed at 0x9D005000, or any address above that in Program Flash that ends in 0x000.

The three lines to change in the application linker script are as follows, with the changes note in **bold** type:

| Original Line of Code | Modified Line of Code |
|---|---|
| `_ebase_address = 0x9D008000;` | `_ebase_address = 0x9D00`**`5`**`000;` |
| `_RESET_ADDR = (0x9D007000);` | `_RESET_ADDR = (0x9D00`**`4`**`000);` |
| `kseg0_program_mem (rx) : ORIGIN = (0x9D000000 + 0x7000), LENGTH = 0x80000 - 0x7000` | `kseg0_program_mem (rx) : ORIGIN = (0x9D000000 + 0x`**`4`**`000), LENGTH = 0x80000 - 0x`**`4`**`000` |

## Library Interface

### a) Functions

| | Name | Description |
|---|---|---|
| | Bootloader_Initialize | Initializes the Bootloader Library. |
| | Bootloader_Tasks | Maintains the Bootloader module state machine. It manages the Bootloader module object list items and responds to Bootloader module primitive events. |

### b) Data Types and Constants

| | Name | Description |
|---|---|---|
| | BOOTLOADER_CLIENT_STATUS | Enumerated data type that identifies the Bootloader module client status. |
| | BOOTLOADER_INIT | A structure used to initialize the bootloader defining the different bootloader. |
| | BOOTLOADER_TYPE | A structure used to initialize the bootloader defining the different bootloader. |
| | MAJOR_VERSION | Bootloader Major Version Shown From a Read Version on PC |
| | MINOR_VERSION | Bootloader Minor Version Shown From a Read Version on PC |
| | BOOTLOADER_BUFFER | This is type BOOTLOADER_BUFFER. |
| | BOOTLOADER_CALLBACK | Pointer to a Bootloader callback function data type . |
| | BOOTLOADER_DATA_CALLBACK | Pointer to a Bootloader callback function data type . |
| | BootInfo | This is variable BootInfo. |

### Description

This section describes the Application Programming Interface (API) functions of the Bootloader Library.

Refer to each section for a detailed description.

## a) Functions

### *Bootloader_Initialize Function*

Initializes the Bootloader Library.

#### File

bootloader.h

#### C

```
void Bootloader_Initialize(const BOOTLOADER_INIT * drvBootloaderInit);
```

#### Returns

If successful, returns a valid handle to a device layer object. Otherwise, it returns SYS_MODULE_OBJ_INVALID.

### Description

This function is used to initialize the Bootloader Library.

### Remarks

This routine must be called before other Bootloader Library functions.

### Preconditions

None.

### Function

void Bootloader_Initialize (const SYS_MODULE_INDEX   moduleIndex,

const SYS_MODULE_INIT    * const moduleInit);

## *Bootloader_Tasks Function*

Maintains the Bootloader module state machine. It manages the Bootloader module object list items and responds to Bootloader module primitive events.

### File

bootloader.h

### C

```
void Bootloader_Tasks();
```

### Returns

None.

### Description

This function maintains the Bootloader module state machine and manages the Bootloader Module object list items and responds to Bootloader Module events. This function should be called from the SYS_Tasks function.

### Remarks

This function is normally not called directly by an application.

### Preconditions

None.

### Example

```
while (true)
{
    Bootloader_Tasks ();

    // Do other tasks
}
```

### Parameters

| Parameters | Description |
|---|---|
| index | Object index for the specified module instance. |

### Function

void Bootloader_Tasks (SYS_MODULE_INDEX index);

## b) Data Types and Constants

## *BOOTLOADER_CLIENT_STATUS Enumeration*

Enumerated data type that identifies the Bootloader module client status.

### File

bootloader.h

### C

```
typedef enum {
  BOOTLOADER_CLIENT_STATUS_CLOSED,
  BOOTLOADER_CLIENT_STATUS_READY,
  BOOTLOADER_CLIENT_STATUS_WRITEFAILURE,
  BOOTLOADER_CLIENT_STATUS_WRITESUCCESS
} BOOTLOADER_CLIENT_STATUS;
```

### Members

| Members | Description |
|---|---|
| BOOTLOADER_CLIENT_STATUS_CLOSED | Client is closed or the specified handle is invalid |
| BOOTLOADER_CLIENT_STATUS_READY | Client is ready |

### Description

Bootloader Client Status

This enumeration defines the possible status of the Bootloader module client. It is returned by the () function.

### Remarks

None.

## *BOOTLOADER_INIT Structure*

A structure used to initialize the bootloader defining the different bootloader.

### File

bootloader.h

### C

```
typedef struct {
  BOOTLOADER_TYPE drvType;
  BOOTLOADER_STATES (* drvTrigger)(void);
} BOOTLOADER_INIT;
```

### Description

Bootloader Initialization Type

This structure holds the bootloader types.

### Remarks

None.

## *BOOTLOADER_TYPE Enumeration*

A structure used to initialize the bootloader defining the different bootloader.

### File

bootloader.h

### C

```
typedef enum {
  TYPE_I2C,
  TYPE_USART,
  TYPE_USB_HOST,
  TYPE_USB_DEVICE,
  TYPE_ETHERNET_UDP_PULL,
  TYPE_SD_CARD
} BOOTLOADER_TYPE;
```

### Description

Bootloader Type

This structure holds the bootloader types.

### Remarks

None.

## *MAJOR_VERSION Macro*

### File

bootloader.h

### C

```
#define MAJOR_VERSION 4    /* Bootloader Major Version Shown From a Read Version on PC */
```

### Description

Bootloader Major Version Shown From a Read Version on PC

## *MINOR_VERSION Macro*

### File

bootloader.h

### C

```
#define MINOR_VERSION 1    /* Bootloader Minor Version Shown From a Read Version on PC */
```

### Description

Bootloader Minor Version Shown From a Read Version on PC

## *BOOTLOADER_BUFFER Union*

### File

bootloader.h

### C

```
typedef union {
  uint8_t buffer[1024];
  struct {
    uint8_t buff1[512];
    uint8_t buff2[512];
  } buffers;
} BOOTLOADER_BUFFER;
```

### Description

This is type BOOTLOADER_BUFFER.

## *BOOTLOADER_CALLBACK Type*

Pointer to a Bootloader callback function data type .

### File

bootloader.h

### C

```
typedef int (* BOOTLOADER_CALLBACK)(void);
```

### Description

Bootloader General Callback Function Pointer

This data type defines a pointer to a Bootloader library callback function.

### Remarks

This is used for callback on the following events:

- ERASE - Erase any additional areas (i.e. SQI, EBI, SPI memories)
- PROGRAM_COMPLETE - No more data coming
- START_APP - Anything that needs to be done prior to starting application
- BLANK_CHECK - Check external devices for erasure
- FORCE_BOOTLOAD - Any checks the bootloader wants to do before launching application

## *BOOTLOADER_DATA_CALLBACK Type*

Pointer to a Bootloader callback function data type .

### File

bootloader.h

### C

```
typedef int (* BOOTLOADER_DATA_CALLBACK)(uint32_t address, uint32_t *data, uint32_t size);
```

### Description

Bootloader Data to Program Callback Function Pointer

This data type defines a pointer to a Bootloader data callback function. This would be used to program data for an area not recognized by the bootloader as being in progam Flash (i.e. SPI Flash).

### Remarks

This is used for callback on the following events:

- DATA_PROGRAM - Data to be programmed
- DATA_VERIFY - Compute the CRC for the given area

## *BootInfo Variable*

### File

bootloader.h

### C

```
const uint8_t BootInfo[2] = { MINOR_VERSION, MAJOR_VERSION };
```

### Description

This is variable BootInfo.

# Files

## Files

| Name | Description |
|---|---|
| bootloader.h | The header file joins all header files used in the Bootloader Library and contains compile options and defaults. |

### Description

This section lists the source and header files used by the Bootloader Library.

## bootloader.h

The header file joins all header files used in the Bootloader Library and contains compile options and defaults.

### Enumerations

| | Name | Description |
|---|---|---|
| | BOOTLOADER_CLIENT_STATUS | Enumerated data type that identifies the Bootloader module client status. |
| | BOOTLOADER_TYPE | A structure used to initialize the bootloader defining the different bootloader. |

### Functions

| | Name | Description |
|---|---|---|
| | Bootloader_Initialize | Initializes the Bootloader Library. |
| | Bootloader_Tasks | Maintains the Bootloader module state machine. It manages the Bootloader module object list items and responds to Bootloader module primitive events. |

### Macros

| | Name | Description |
|---|---|---|
| | MAJOR_VERSION | Bootloader Major Version Shown From a Read Version on PC |
| | MINOR_VERSION | Bootloader Minor Version Shown From a Read Version on PC |

### Structures

| | Name | Description |
|---|---|---|
| | BOOTLOADER_INIT | A structure used to initialize the bootloader defining the different bootloader. |

### Types

| | Name | Description |
|---|---|---|
| | BOOTLOADER_CALLBACK | Pointer to a Bootloader callback function data type . |
| | BOOTLOADER_DATA_CALLBACK | Pointer to a Bootloader callback function data type . |

### Unions

| | Name | Description |
|---|---|---|
| | BOOTLOADER_BUFFER | This is type BOOTLOADER_BUFFER. |

### Variables

| | Name | Description |
|---|---|---|
| | BootInfo | This is variable BootInfo. |

### Description

Module for Microchip Bootloader Library

This header file includes all the header files required to use the Microchip Bootloader Library. Library features and options defined in the Bootloader Library configurations will be included in each build.

## File Name

bootloader.h

## Company

Microchip Technology Inc.

# Index