Bootloader Library Help

MPLAB Harmony Integrated Software Framework v1.07.01

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).

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`.

Library File:

The Bootloader Library archive (`.a`) file is installed with MPLAB Harmony.

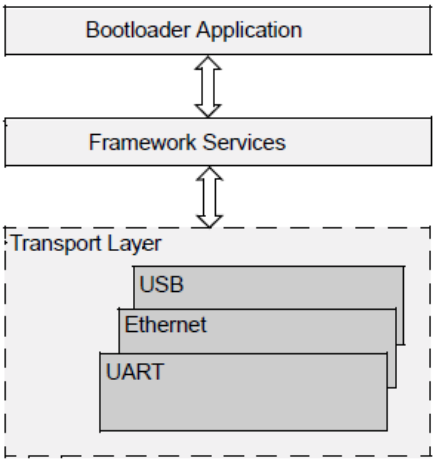
Please refer to the What is MPLAB Harmony? section for how the Bootloader Library interacts with the framework.

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



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.

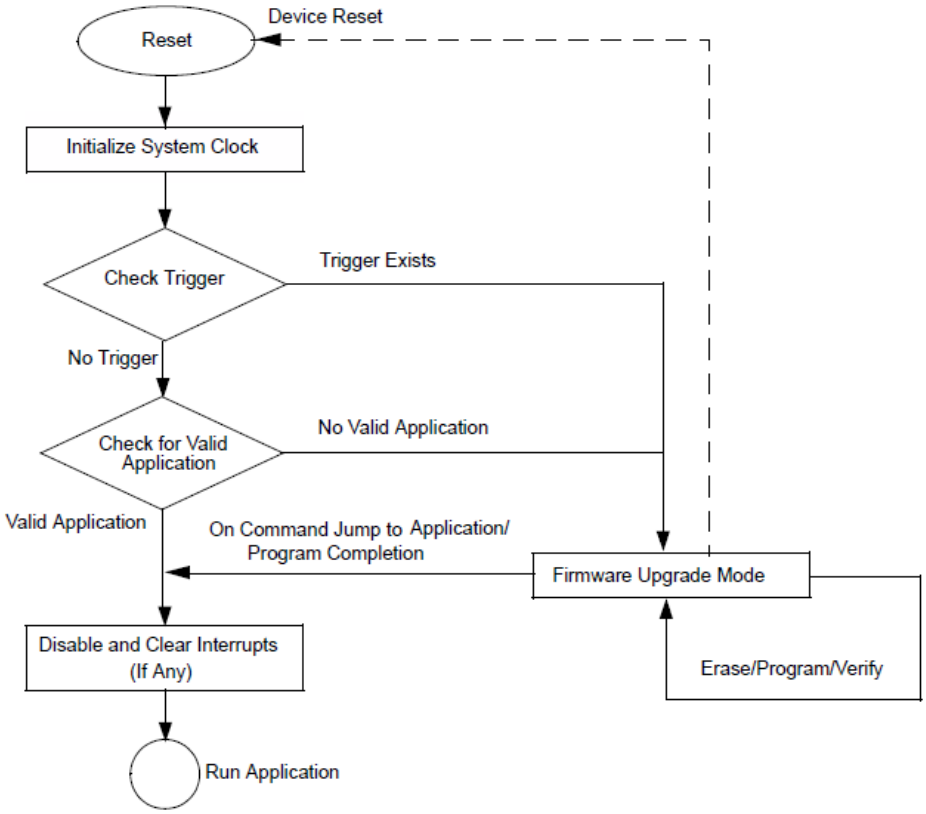
How the Library Works

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. To manually force the Bootloader into the firmware upgrade mode, press and hold the switch, S3, on the Explorer 16 Development Board during power-up. On PIC32 starter kits, press and hold the switch, SW3, during power-up. While in firmware upgrade mode, the LED labeled D5 on the Explorer 16 Development Board and the LED labeled LED3 on the PIC32 starter kit will blink.

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 or SD card 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

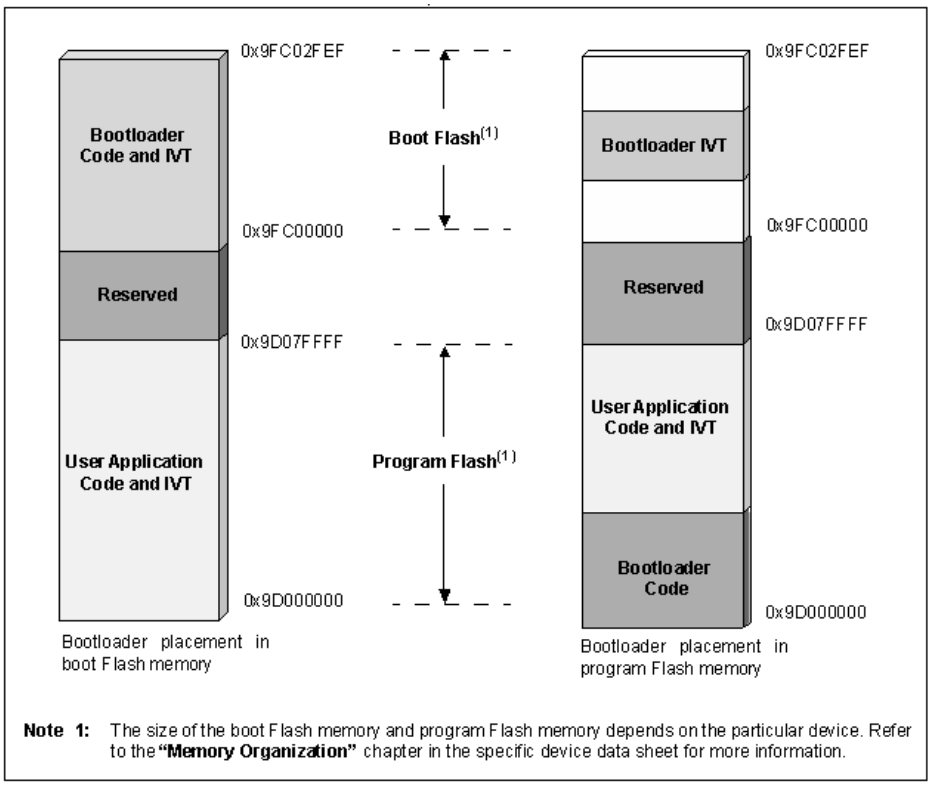
Describes the two schemes for Bootloader placement in memory.

Description

Figure 1 illustrates two schemes for the Bootloader placement based on the size of the Bootloader. Bootloaders that are smaller in size are placed within the PIC32 boot Flash memory. Fitting the Bootloader application 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. 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.

Figure 1: Bootloader Placement



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.

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:

- Applications Helps > Bootloader Demonstrations
- Applications Help > Examples > Peripheral Library Examples > dma_led_pattern

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

Provides information on using the Bootloader application.

Description

Use the following procedure to run the UART/USB HID/Ethernet Bootloader:

1. Select the specific hardware setup for the selected Bootloader.
2. With a debugger or programmer, program the Bootloader into the device using Release Build.
3. Run `PIC32UBL.exe`, which is located in the `..\PC_Application\` folder.
4. Depending on the Bootloader workspace used, enable either the USB, Serial port, or Ethernet using the Communication Settings menu in the PC application:

- For the Serial Port Bootloader, the default baud rate is 115200
- For the USB Bootloader, the default values of Vendor ID number (VID) and Product ID number (PID) are 0x4D8 and 0x03C, respectively
- For the Ethernet Bootloader, the default IP address is 192.168.1.11

5. If you are using the Ethernet Bootloader, set the IP address and subnet mask of the PC to 192.168.1.12 and 255.255.255.0, respectively.
6. To enter the firmware upgrade mode, use the procedure as described in **Entering the Firmware Upgrade Mode** in Basic Flow of the Bootloader.
7. Depending on the type of Bootloader programmed, connect the serial cable, micro-USB cable, or the Ethernet (RJ-45) crossover cable to the Explorer 16 Development Board.
8. Click **Connect**. The device connects and the Bootloader version information is read.
9. Click **Load Hex file**, and browse to the `Demo_Application` folder.
10. Select the specific demonstration application file, `Demo_App_Explorer16.hex/Demo_App_PIC32_Starter_Kits.hex`, located in the `Firmware\Demo_Application` folder.
11. Click **Erase** to erase the device.
12. Click **Program** to program the previously loaded `.hex` file into the device Flash.
13. Click **Verify** to verify the Flash contents. If the Flash is written correctly, the “Verification successful” message is displayed in the console.
14. Click **Run Application**. Optionally, steps 11, 12 and 13 can be performed as a single action by clicking **Erase-Program-Verify**.
15. Once the device starts running the programmed application firmware, the PC application will not be able to communicate with the device. To reconnect to the Bootloader, return to step 6.

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 C32 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.

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.

1. Create a new text file and save it with a `.ld` file extension.
2. Add the new `*.ld` file to your project. The new `*.ld` file appears in the project tree.
3. Starting with the default linker script is easier than starting from the scratch. Use a text editor to copy the contents of the `\pic32mx\lib\ldscripts\elf32pic32mx.x` default linker script into the newly created `*.ld` file. The `INCLUDE procdefs.ld` directive should be replaced with the contents of the device-specific

`\pic32mx\lib\proc\device\procdefs.ld` portion of the linker script. The path `\pic32mx\lib` is located within the folder where the XC32 compiler tools are installed.

4. Edit the newly created `*.ld` file to remap the linker script memory regions `exception_mem`, `kseg0_boot_mem`, `kseg1_boot_mem` and `kseg0_program_mem` into the program Flash reserved for the user application. The `exception_mem` must align on a 4K address boundary of the program Flash as shown in Example 11.

Note: For more information on linker script memory regions, refer to the *"MPLAB® Assembler, Linker and Utilities for PIC32 MCUs User's Guide"* (DS50001833) and the *"MPLAB® C Compiler for PIC32 MCUs User's Guide"* (DS50001686).

Example 11: Lines to Modify in Application Linker Script

```
/*************************************************************************
 * Processor-specific object file. Contains SFR definitions.
 *************************************************************************/
INPUT("processor.o")

/*************************************************************************
 * For interrupt vector handling
 *************************************************************************/
PROVIDE(_vector_spacing = 0x00000001);
/* _ebase_address value must be same as the ORIGIN value of exception_mem (see below) */
_ebase_address = 0x9D000000;

/*************************************************************************
 * Memory Address Equates
 *************************************************************************/
 /* Equate _RESET_ADDR to the ORIGIN value of kseg1_boot_mem (see below) */
_RESET_ADDR             = (0x9D000000 + 0x1000 + 0x970);

/*Map _BEV_EXCPT_ADDR and _DBG_EXCPT_ADDR in to kseg1_boot_mem (see below) */
/* Place _BEV_EXCPT_ADDR at an offset of 0x380 to _RESET_ADDR */
/* Place _DBG_EXCPT_ADDR at an offset of 0x480 to _RESET_ADDR */

_BEV_EXCPT_ADDR         = (0x9D000000 + 0x1000 + 0x970 + 0x380);
_DBG_EXCPT_ADDR         = (0x9D000000 + 0x1000 + 0x970 + 0x480);

_DBG_CODE_ADDR          = 0xBFC02000;
_DBG_CODE_SIZE          = 0xFF0     ;
_GEN_EXCPT_ADDR         = _ebase_address + 0x180;

/*************************************************************************
 * Memory Regions
 *
 * Memory regions without attributes cannot be used for orphaned sections.
 * Only sections specifically assigned to these regions can be allocated
 * into these regions.
 *************************************************************************/
MEMORY
{
  /* IVT is mapped into the exception_mem. ORIGIN value of exception_mem must align with 4K
     address boundary. Keep the default value for the length */

  exception_mem       : ORIGIN = 0x9D000000, LENGTH = 0x1000

  /* Place kseg0_boot_mem adjacent to exception_mem. Keep the default value for the length */
  kseg0_boot_mem      : ORIGIN = (0x9D000000 + 0x1000), LENGTH = 0x970

  /* C Start-up code is mapped into kseg1_boot_mem. Place kseg1_boot_mem adjacent to
     kseg0_boot_mem. Keep the default value for the length */
  kseg1_boot_mem      : ORIGIN = (0x9D000000 + 0x1000 + 0x970), LENGTH = 0x490

  /*All C files (Text and Data) are mapped into kseg0_program_mem. Place kseg0_program_mem
    adjacent to kseg1_boot_mem. Change the length of kseg0_program_mem as required. In this
    example, 512 KB Flash size is shrunk as follows: */

  kseg0_program_mem (rx):ORIGIN = (0x9D000000 + 0x1000 + 0x970 + 0x490),
```

```
    LENGTH = (0x80000 - (0x1000 + 0x970 + 0x490))


  debug_exec_mem             : ORIGIN = 0xBFC02000, LENGTH = 0xFF0
  config3                    : ORIGIN = 0xBFC02FF0, LENGTH = 0x4
  config2                    : ORIGIN = 0xBFC02FF4, LENGTH = 0x4
  config1                    : ORIGIN = 0xBFC02FF8, LENGTH = 0x4
  config0                    : ORIGIN = 0xBFC02FFC, LENGTH = 0x4
  kseg1_data_mem       (w!x) : ORIGIN = 0xA0000000, LENGTH = 0x20000
  sfrs                       : ORIGIN = 0xBF800000, LENGTH = 0x100000
  configsfrs                 : ORIGIN = 0xBFC02FF0, LENGTH = 0x10
}
```

5. Set the value of `_ebase_address` to the `ORIGIN` value of `exception_mem`.
6. Change the values of memory addresses `_RESET_ADDR`, `_BEV_EXCPT_ADDR` and `_DBG_EXCPT_ADDR`, so that all these addresses fall inside `kseg1_boot_mem`.
7. Clean and build the application project to get a remapped application image.

Note: The modified application linker script file is not suitable for building the application in debug or stand-alone (to use without Bootloader) mode.

The next step following the application remap is informing the Bootloader about the new location and reset address of the user application in the program Flash.

The Bootloader code provides compile time options for this purpose. The macros, `APP_FLASH_BASE_ADDRESS` and `APP_FLASH_END_ADDRESS`, define the start and the end addresses of the program Flash reserved for the user application. The Bootloader performs an erase or program operation only if the target address of the Flash is within these addresses. Therefore, the user must set new start and end address values to these macros following the application remap. Set the addresses so that `exception_mem`, `kseg0_boot_mem`, `kseg1_boot_mem`, and `kseg0_program_mem` defined in the `procdefs.ld` file of the user application are within these addresses, as shown in Example 12.

Example 12: Lines to Modify in Bootloader Code

```
/* APP_FLASH_BASE_ADDRESS and APP_FLASH_END_ADDRESS reserves program Flash for the application
*/
/* Rule:
        1) The memory regions kseg0_program_mem, kseg0_boot_mem, exception_mem and
        kseg1_boot_mem of the application linker script must fall within APP_FLASH_BASE_ADDRESS
        and APP_FLASH_END_ADDRESS

        2) The base address and end address must align on 4K address boundary
*/

#define APP_FLASH_BASE_ADDRESS                                      0x9D000000
#define APP_FLASH_END_ADDRESS                                      0x9D07FFFF

/* Address of  the Flash from where the application starts executing */
/* Rule: Set APP_FLASH_BASE_ADDRESS to _RESET_ADDR value of application linker script */

#define USER_APP_RESET_ADDRESS     (0x9D000000 + 0x1000 + 0x970)
```

The macro, `USER_APP_RESET_ADDRESS`, specifies the reset address of the user application. The Bootloader branches to this address when it must run the user application. The value of this macro must be changed to `_RESET_ADDR` defined in the `procdefs.ld` file of the user application project. The Bootloader project must be recompiled and programmed into the PIC32 device after modifying these macros.

Bootloader Configurations

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

Description

The Bootloader code provides a few macros for configuring the settings while compiling. Table 3 through Table 6 list these macros and their usage. Depending on user requirements, the values in these macros may need to be changed.

Table 3: General Macros

Macro	Usage
APP_FLASH_BASE_ADDRESS	Base address of the program Flash reserved for the user application. The address value must point to the beginning of a 4K Flash page.
APP_FLASH_END_ADDRESS	End address of the program Flash reserved for the user application. The address value must point to the end of a 4K Flash page.
USER_APP_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.

Table 4: UART Bootloader Macro

Macro	Usage
DEFAULT_BAUDRATE	Sets the UART baud rate.

Table 5: USB HID Bootloader Macros

Macro	Usage
USB_VENDOR_ID	Sets the vendor ID.
USB_PRODUCT_ID	Sets the product ID.

Table 6: Ethernet Bootloader Macros

Macro	Usage
MY_DEFAULT_MAC_BYTE1 MY_DEFAULT_MAC_BYTE2 MY_DEFAULT_MAC_BYTE3 MY_DEFAULT_MAC_BYTE4	Sets the MAC address.
MY_DEFAULT_IP_ADDR_BYTE1 MY_DEFAULT_IP_ADDR_BYTE2 MY_DEFAULT_IP_ADDR_BYTE3 MY_DEFAULT_IP_ADDR_BYTE4	Sets the IP Address.

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>/build/framework/bootloader/bootloader.X`	Include this project to bring in the library and all associated functions. There are two build configuration options available for this library: • SW – Software-only configuration for all Microchip PIC32 microcontrollers • HW – Hardware acceleration configuration for PIC32 devices that have a hardware encryption module For both configurations, adjust the properties if 16-bit code is desired, and set the appropriate version of microcontroller.
`<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
N/A	No optional files are available for this library.

Module Dependencies

The Bootloader Library does not depend on any other modules.

Bootloader Plug-in for MPLAB X IDE

Describes the Bootloader plug-in.

Description

The Bootloader plug-in is a PC host application, which can be used to communicate with the Bootloader firmware of a PIC32 device.

Note: This plug-in is required for UART, USB HID and Ethernet Bootloaders.

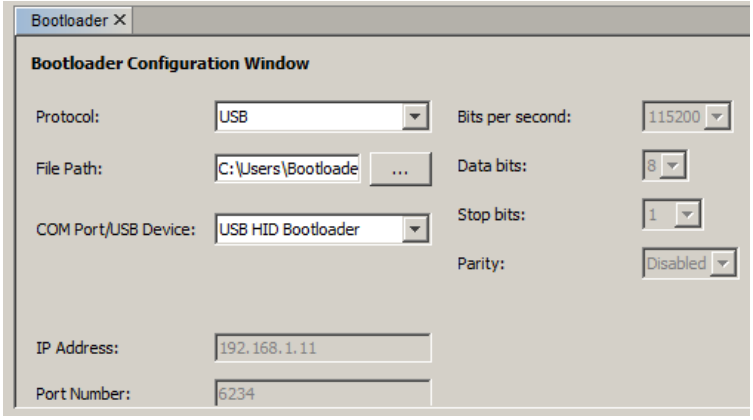
Installing the Bootloader Plug-in

1. Open MPLAB X IDE and select *Tools* > *Plugins*.
2. Select the **Available Plugins** tab and select **Bootloader** from the list, and then click **Install**.
3. When the Plugin Installer window appears, click **Next**.
4. Accept the license terms, and then click **Install**.
5. If prompted to verify the certificate, click **Continue**.
6. When the *Installation completed successfully* message appears, click **Finish** to complete the installation.

Using the Bootloader Plug-in

This application is used to perform erase and programming operations.

1. In MPLAB X IDE, select *Window* > *Bootloader*. The Bootloader dialog will appear, as shown in the following figure.



2. Depending on the Bootloader configuration, enable either the USB, serial port (UART), or Ethernet (UDP) using the **Protocol** drop-down.
3. Choose the correct hex file in the **File Path:** by either typing it directly or by using the file navigation icon (...). The reference application project for the Bootloader demonstrations are located in `<install-dir>/apps/examples/peripherals/dma/dma_led_pattern`.
4. Choose the desired **COM Port/USB Device**:
   - UART Bootloader - The default baud rate is 115200. The Data bits should be set to 8, the Stop bits should be set to 1, and Parity should be disabled.
   - USB HID Bootloader - The default values of Vendor ID number (VID) and Product ID number (PID) are 0x4D8 and 0x03C, respectively. If the USB is connected and running, the device in the COM Port/USB Device drop-down list will appear as USB HID Bootloader; otherwise, this field is empty.
   - Ethernet Bootloader - The default IP address is 192.168.1.11. Set the IP address and subnet mask of the PC to 192.168.1.12 and 255.255.255.0, respectively.
5. Once properly configured, click the Bootloader icon ( ), which is located in the MPLAB X IDE toolbar. The application will then erase the device, program the Flash, and reset the device. Once completed, the device will be running the programmed application.

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

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
} 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

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.

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