MODULAR BOOTLOADER FRAMEWORK FOR SILICON LABS SIMXXXXX MICROCONTROLLERS

1. Introduction

A bootloader enables device firmware upgrades (DFU) without the need for dedicated, external programming hardware. All Silicon Labs SiMxxxxx MCUs with Flash memory are “self-programmable”, i.e., code running on the MCUs can erase and write other parts of the code memory. A bootloader can be programmed into these devices to enable initial programming or field updates of the application firmware without using a Serial Wire or JTAG adapter. The firmware update is delivered to the MCU via a communication channel that is typically used by the application for its normal operation, such as UART, USB, SPI, I2C, CAN, Ethernet, or over a wireless link.

This application note describes a modular bootloader framework that can be used to implement a bootloader system for any communication channel. The framework is structured in such a way as to be able to both re-use most of the code as-is across different Silicon Labs MCU families and to use it with various communication channels and data sources. Additional related application notes describe the interface-specific implementation details for various communication channels, such as UART or USB. These documents are available at http://www.silabs.com/products/MCU/Pages/ApplicationNotes.aspx

2. SiMxxxxx Modular Bootloader Framework Overview

The SiMxxxxx modular bootloader framework consists of the following components:

- Target MCU
- Master programmer
- Data source

The goal of the bootloader framework, shown in Figure 1, is to provide a mechanism for transferring a firmware image stored in the data source to the target MCU’s flash memory. The framework provides the software necessary to manage the firmware update process and is based on the USB Device Firmware Upgrade specification. To ensure compatibility with all SiMxxxxx devices, all device-specific and protocol-specific functionality is abstracted, making possible a variety of different and inter-operable implementations. For example, a master programmer and data source implemented in a PC application can support communication over USB or UART to accommodate different classes of target MCUs that all utilize the same framework. The master programmer can also be implemented in a dedicated hardware solution to reduce programming time and maintain compatibility with a PC based solution.

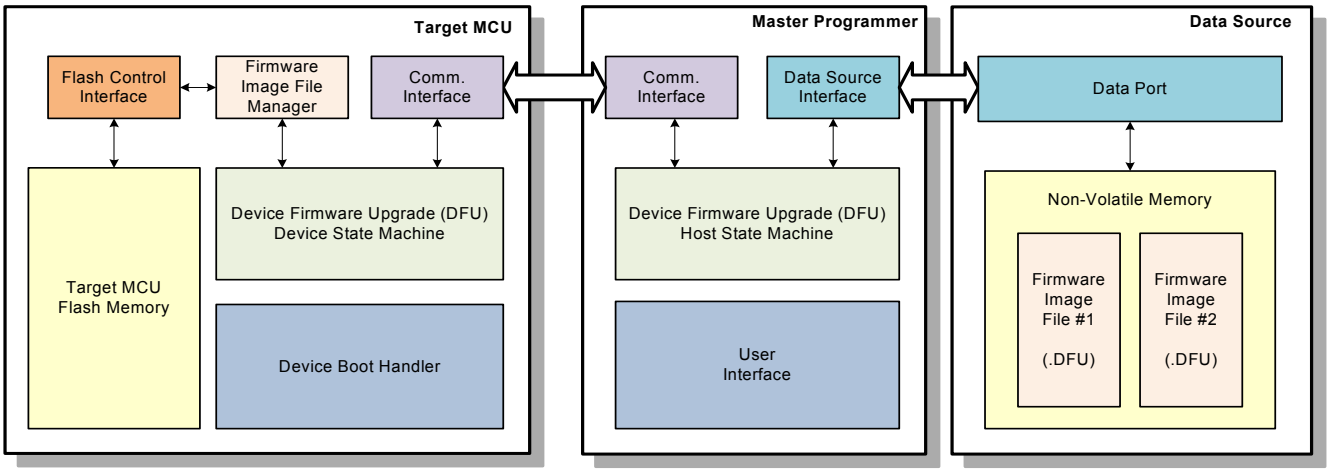

Figure 1. SiMxxxxx Modular Bootloader Framework Overview

3. Related Documentation

This framework description relies upon the USB Device Class Specification for Device Firmware Upgrade available from http://www.usb.org/developers/devclass_docs/DFU_1.1.pdf

Application Note AN763 documents an example implementation of a UART and USB bootloader based on this framework. The target device family is the SiM3U1xx, and the master programmer is implemented as a PC application that uses the Windows file system as its data source. Full source code for the target MCU and master programmer are distributed with the framework code as part of the software accompanying AN762.

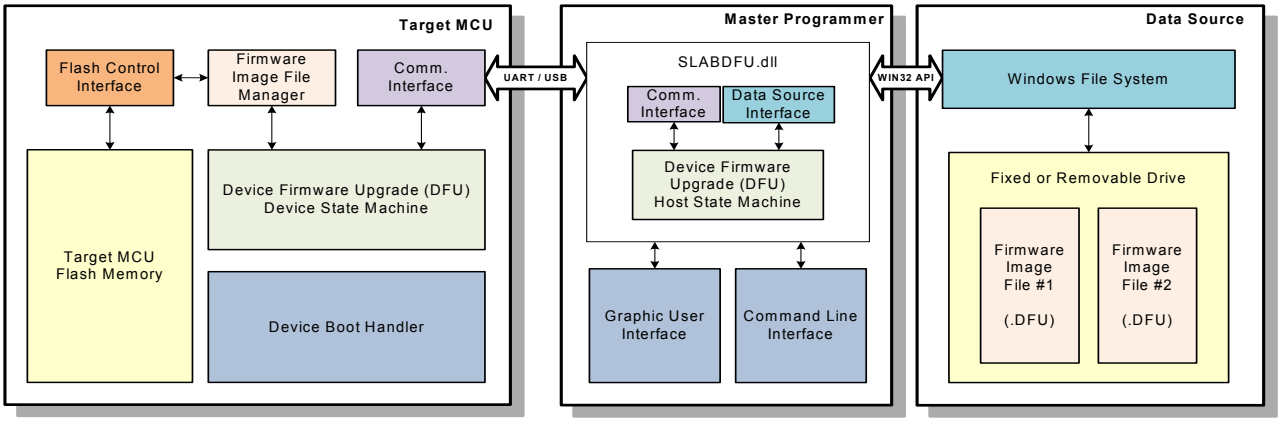

Figure 2. UART/USB Bootloader Implementation

4. Using the SiMxxxxx Bootloader Framework Software

The framework software for the target MCU bootloader distributed with this application note consists of the following modules:

- DFU — hardware and protocol independent implementation of the DFU device state machine. This module is handed code execution during the firmware update process and performs API function calls into COMM and FILEMGR as needed to perform the firmware update. All shared memory buffers reside in the DFU module and are accessed from the other modules with a pointer.
- FILEMGR — hardware and protocol independent DFU file manager responsible for decoding and validating incoming DFU files and committing them to Flash memory. This module is also responsible for assembling a DFU file containing the current firmware image on firmware uploads.
- FLCTL — hardware specific flash module responsible for performing low level Flash operations.
- COMM — hardware specific communication interface responsible for the guaranteed delivery and reception of error-free data packets to support communication between the DFU module and the master programmer.
- DEVICE — hardware specific boot handler responsible for checking the validity of the application image in Flash, checking for trigger sources, and setting the appropriate trigger flag if a firmware update is required.
- MAIN — starting point of code execution and responsible for jumping to the user application unless a firmware update is pending.

In addition, the framework software has a user configuration file *userconfig.h* containing various compile time options. The *global.h* header file contains all symbols defined in the project and is included at the top of every C source file.

5. Target MCU Bootloader Design

The target MCU bootloader is the entry point of code execution following each device reset (or “boot”). The main purpose of the bootloader is to run (or “load”) an application or operating system. The bootloader performs application image verification prior to transferring control and also manages firmware updates. The bootloader resides at the beginning of program memory and co-exists with other applications.

5.1. Target MCU Bootloader Functional Description

Figure 3 shows a block diagram of the target MCU bootloader. The first module executed from reset is the device boot handler, which makes a decision to transfer control to the user application or initiate a firmware upgrade operation. During normal operation, the device boot handler is transparent to the main application. If the device boot handler determines that a firmware upgrade is needed, then control is transferred to the Device Firmware Upgrade (DFU) device state machine. The DFU device state machine receives a firmware image over the communication interface and passes it onto the firmware image file manager which validates the new firmware image and programs it into Flash. The Flash control interface provides low-level write and erase functionality.

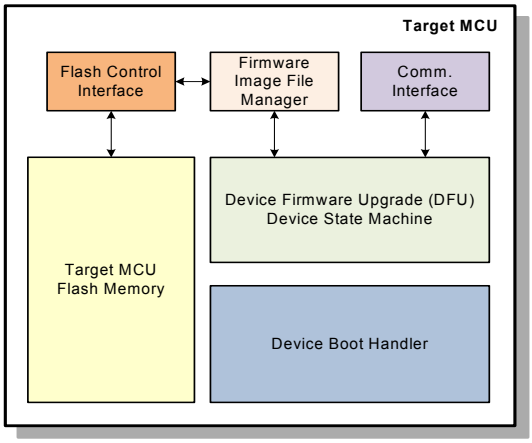

Figure 3. Target MCU Bootloader

5.2. Device Boot Handler

The device boot handler begins executing after each device reset and makes a decision to transfer control to the user application or initiate a firmware upgrade. Each device specific implementation of the device boot handler needs to perform the following functions:

1. Disable watchdog timer and enable the APB clock to all modules.
2. Determine the amount of Flash and RAM in the device. This will later be used to validate incoming firmware image files.
3. Check for any firmware update trigger conditions. The trigger conditions will vary by implementation and are meant to provide a mechanism by which a firmware update can be requested by the main application or by the end user (e.g. a factory reset button on an end device).
4. Validate the main application firmware image. The validation will vary by implementation with a minimum implementation checking only that a valid stack pointer and reset vector address are present. A full-featured implementation can perform a CRC check on the entire application space to determine validity.
5. If a valid application is detected and none of the firmware updated triggers have been set, then the main routine will transfer control to the user application. Otherwise, set the appropriate trigger so that upon exit, control is transferred to the DFU device state machine to perform a firmware update.

5.3. DFU Device State Machine

The DFU device state machine is the primary command interpreter for the device once it has entered bootload mode. Figure 4 shows a layout of the active framework modules when the device is in bootload mode. The DFU device state machine manages the firmware update process and receives commands directly over the communication interface by calling the `COMM_Receive(buffer, length)` function provided by the COMM module.

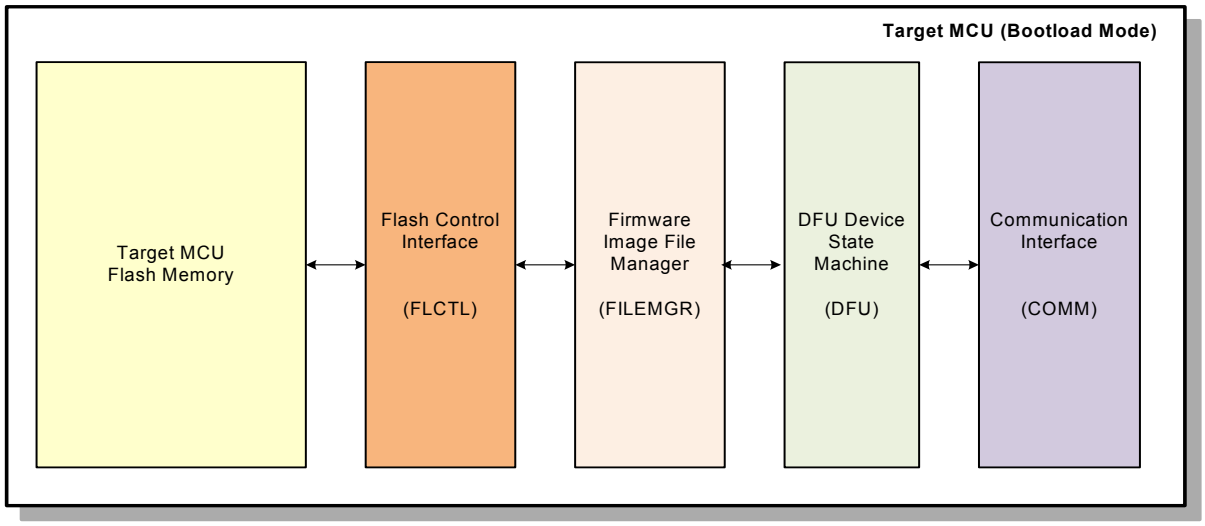

Figure 4. Target MCU Bootloader in Bootload Mode

The two primary operations that can be requested by the master programmer are firmware upload and download. Upon receiving the appropriate command to initiate an operation, the DFU module calls the API functions provided by FILEMGR to recall (upload) or store (download) a firmware image file. The DFU state machine is agnostic of the contents or format of the firmware image file and relies on the FILEMGR module to handle all tasks related to validating and decoding the firmware image file in a download operation or creating a firmware image file in an upload operation. The DFU module sends data back to the host by calling the `COMM_Transmit(buffer, length)` function provided by the COMM module.

The DFU device state machine implementation is based on the firmware upgrade procedure described in the *USB Device Firmware Upgrade Specification, Revision 1.1,* with added flexibility to allow the firmware update to take place over any communication interface. In this implementation, the bitManifestationTolerant defined in the specification will always be set to 1 and the USB Reset has been replaced by the vendor specific DFU_RESET command. The command format, which is based on the USB DFU specification, is shown in Figure 5. All multi-byte fields are encoded in little endian.

\

DFU Command Format

bRequest	wValue	wIndex	wLength	Data
DFU Command	Reserved Set to 0x0000	Reserved Set to 0x0000	The number of bytes in the Data field	Variable Length Data Field

Figure 5. DFU Device State Machine Command Format

Figure 6 shows a typical data exchange between the DFU state machines on the master programmer and the target device. In this simple example, the master programmer is downloading a firmware image file that contains three blocks, with 1024 bytes per block. The firmware image file is representative of a small application, such as blinky, that has 2 kB of executable code and a 1 kB information page. Block 0 of the file is the information page and Block 1 and Block 2 contain executable code.

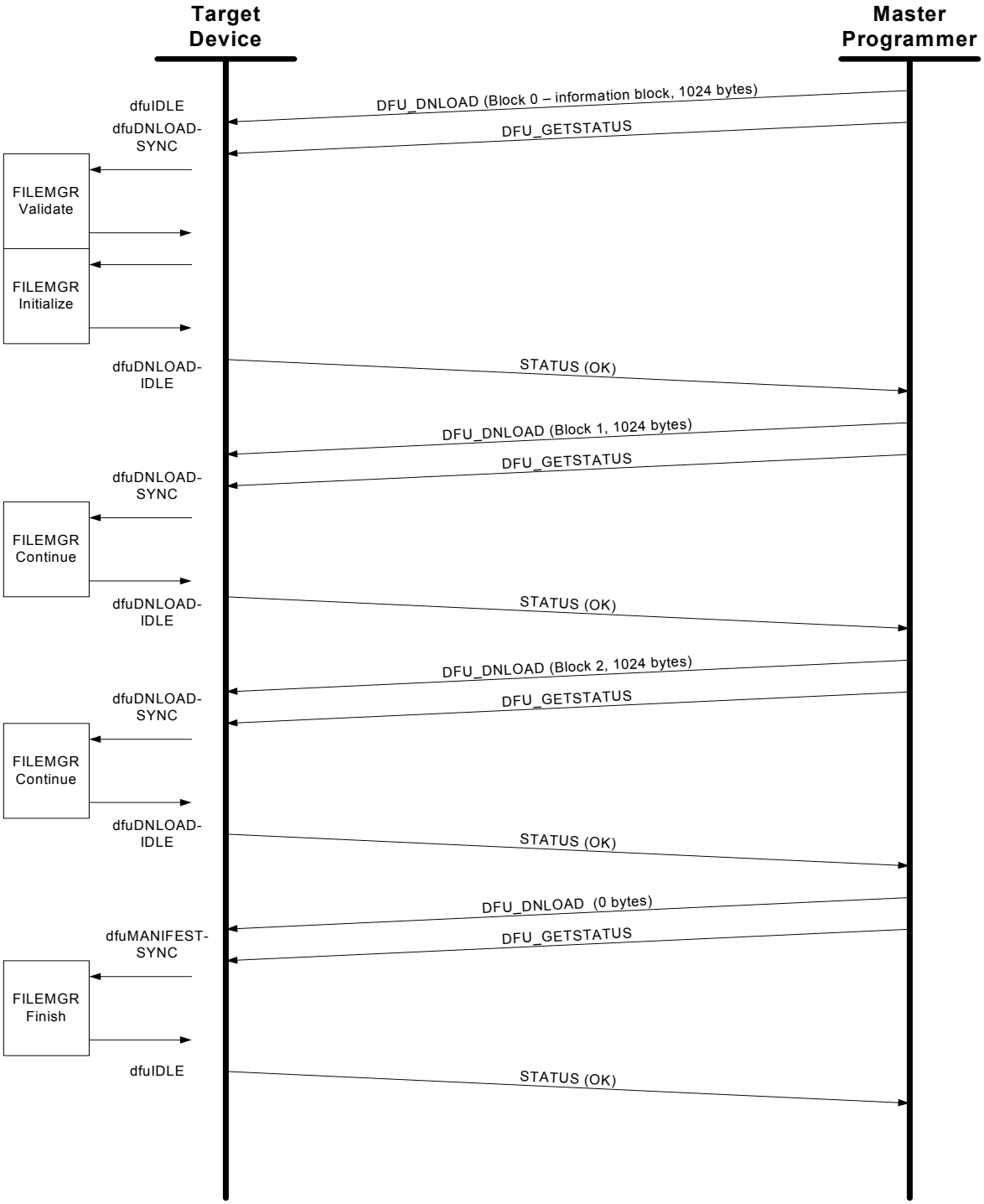

Figure 6. Data Exchange During Download Operation

The DFU device state machine has six states and seven commands used to transition between states. DFU device states and commands are listed in Table 1 and Table 2. The DFU state machine is initialized to the dfuIDLE state if the application space contains a valid firmware image and a firmware upgrade is requested through one of the internal or external trigger sources. If a corrupted application image is detected, then the state machine is initialized to the dfuERROR state. The master programmer must transition the state machine to the dfuIDLE state by sending a DFU_CLRSTATUS command before beginning an upload or download operation. The DFU device state machine is shown in Figure 7. Please see the USB DFU specification for a detailed description of each state and command.

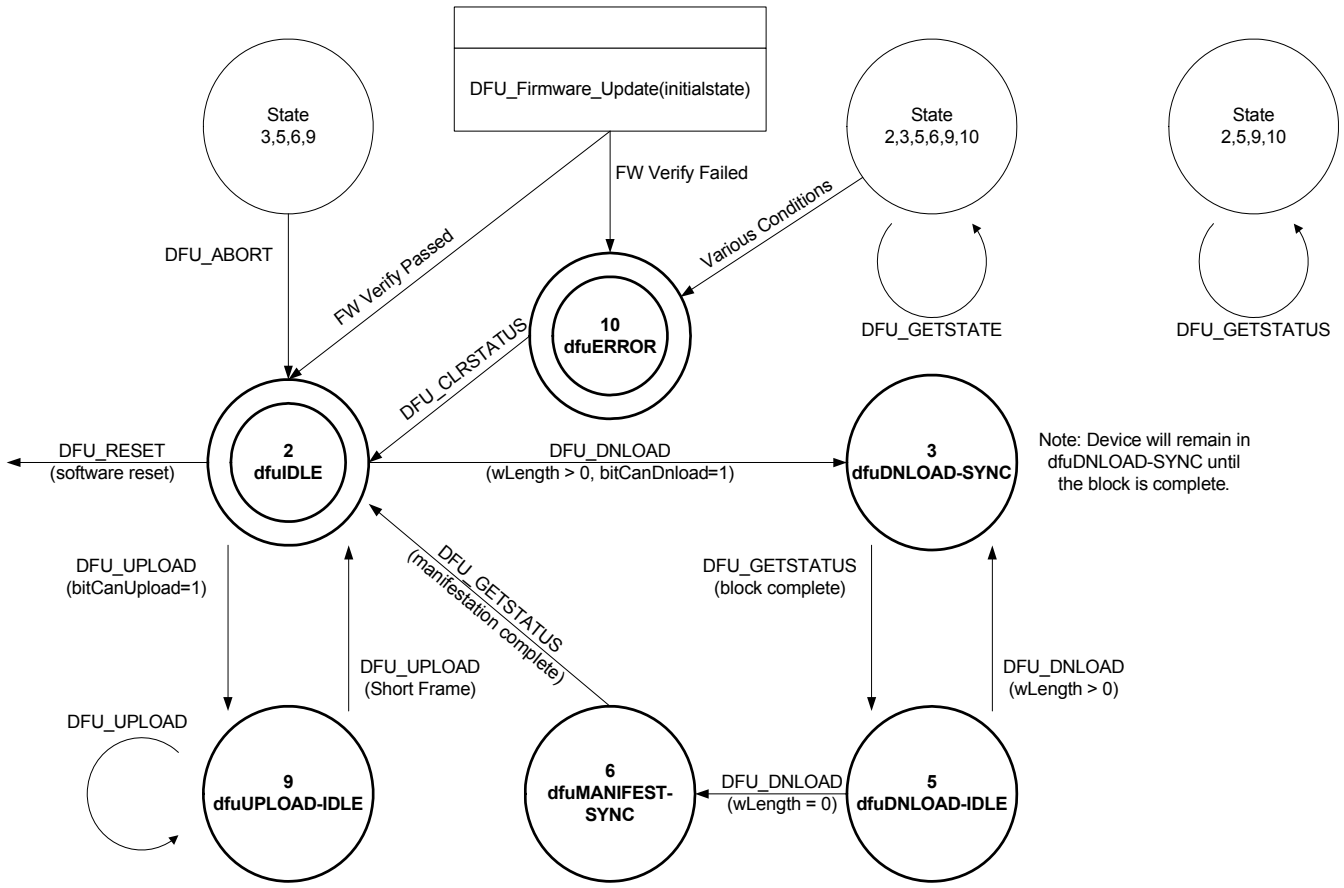

Figure 7. DFU Device State Machine Transition Diagram

Table 1. DFU Device States

State	Number	Description
dfuIDLE	2	The Idle state
dfuDNLOAD-SYNC	3	Used for synchronizing firmware downloads
dfuDNLOAD-IDLE	5	Used for synchronizing firmware downloads. Indicates block complete
dfuMANIFEST-SYNC	6	Provides current state information to the master programmer
dfuUPLOAD-IDLE	9	Used for uploading the firmware image from the target MCU
dfuERROR	10	Used to indicate an error condition

Note: The appIDLE, appDETACH, dfuDNBUSY, dfuMANIFEST, and dfuMANIFEST-WAIT-RESET states are not implemented.

Table 2. DFU Device Commands

Command	Value	Description
DFU_DNLOAD	1	Used for downloading a firmware image file to the target MCU
DFU_UPLOAD	2	Used for uploading the current firmware image from the target MCU
DFU_GETSTATUS	3	Allows the master programmer to obtain detailed information about the error when the target MCU is in the dfuERROR state
DFU_CLRSTATUS	4	Allows the master programmer to clear the error condition
DFU_GETSTATE	5	Provides current state information to the master programmer
DFU_ABORT	6	Allows a firmware upgrade to be aborted
DFU_RESET	7	Used to trigger a software reset on the target MCU

Note: The DFU_DETACH command is not implemented. This command should be supported in the main application when the primary communication interface is USB. The DFU_RESET command is vendor specific.

5.4. Communication Interface (COMM)

The communication interface module (COMM) provides the DFU state machine with a protocol independent API for communicating with the master programmer, and it is responsible for guaranteed delivery and reception of error-free data. The following features should be implemented by the communication interface:

- Variable Payload Size
- Error Checking
- Acknowledgment
- Automatic Retransmission

The COMM interface may be implemented using any communication protocol available to the target MCU. In a USB implementation, the error checking, acknowledgment, and automatic retransmission are automatically handled by hardware. In other protocol implementations, such as UART, these functions are handled in software.

The COMM module implements the following API functions:

- `void COMM_Init (void)`
  Initializes the communication interface.
- `uint32_t COMM_Receive (uint8_t* rx_buff, uint32_t length)`
  Blocking function receives up to <length> bytes over the communication interface and stores them at <rx_buff>. Returns a status code indicating the number of bytes in the receive buffer.
- `uint32_t COMM_Transmit (uint8_t* tx_buff, uint32_t length)`
  Blocking function transmits <length> bytes from <tx_buff> over the communication interface and waits for an acknowledgment. Returns the number of bytes successfully transmitted.

5.5. Firmware Image File Manager

The firmware image file manager is responsible for receiving firmware image files during downloads and for generating firmware image files during uploads. The files generated during uploads must be usable in a subsequent download to allow the master programmer to retrieve and archive a device's firmware prior to downloading a different firmware image.

The FILEMGR module implements the following API functions:

- `uint32_t FILEMGR_Get_Block_Size(void)`
  Returns the block size of the underlying program memory. The block size is typically the Flash sector size.
- `uint32_t FILEMGR_Validate_Dnload(uint8_t* buffer, uint32_t length)`
  Verifies the first block of the firmware image file and erases the entire application space. Returns a status code indicating success or failure.
- `uint32_t FILEMGR_Initialize_Dnload(uint8_t* buffer, uint32_t length)`
  Verifies the first block of the firmware image file and erases the entire application space. Returns a status code indicating success or failure.
- `uint32_t FILEMGR_Continue_Dnload(uint8_t* buffer, uint32_t length)`
  Writes a block of the firmware image file to the application space and verifies that it was properly written. Blocks must be sent consecutively. Returns a status code indicating successful block processing.
- `uint32_t FILEMGR_Finish_Dnload(void)`
  Typically called after a zero-length packet is received indicating that the download is complete. This function verifies the entire application image and returns a status code which indicates success or failure.
- `uint32_t FILEMGR_Start_Upload(uint8_t* buffer, uint32_t length)`
  Fills <buffer> with the first block of the firmware image file. Returns the number of valid bytes in <buffer>.
- `uint32_t FILEMGR_Continue_Upload(uint8_t* buffer, uint32_t length)`
  Copies a block from application space into <buffer>. Blocks must be read consecutively. Returns a status code which includes the number of bytes written into <buffer>. A return value of zero indicates upload complete.

The FILEMGR implementation included in the bootloader framework defines a specific format for the firmware image file. The firmware image file manager will reject any firmware images that do not follow this format and will always return image files in this format in response to an upload command.

FIRMWARE IMAGE FILE INFORMATION BLOCK	uint8_t bDfuFileRevision_Minor;
	uint8_t bDfuFileRevision_Major;
	uint8_t bAppRevision_Minor;
	uint8_t bAppRevision_Major;
	char sAppName[16]; // null terminated string
	char sTargetFamily[16]; // null terminated string
	uint8_t bReserved[12];
	uint32_t wAppSize;
	uint32_t wCrc;
	uint32_t wSignature;
	uint32_t wLock;
	uint32_t wAppStartAddress;
	uint32_t wBlockSize;
	uint32_t wFlashKey_A;
	uint32_t wFlashKey_B;
	
	Pad with 0x00 to achieve a full block with wBlockSize bytes.

EXECUTABLE CODE	Executable code with a maximum of wAppLength bytes. Pad with 0xFF to achieve an integer number of blocks, each block is wBlockSize bytes.

Figure 8. DFU File Format

5.6. Flash Control Interface

The flash control interface provides low-level routines to write and erase flash memory.

The FLCTL module implements the following API functions:

- `uint32_t FLCTL_Init(void)`
  Performs any Flash module initialization.
- `uint32_t FLCTL_Get_Sector_Size(void)`
  Returns the number of bytes in each Flash sector.
- `uint32_t FLCTL_Write(uint8_t* buffer, uint32_t address, uint32_t length)`
  Copies <length> bytes from buffer to flash starting at <address>. Returns number of bytes written.
- `uint32_t FLCTL_PageErase(uint32_t address)`
  Erases one flash page starting at <address>. Returns a status code indicating success or failure.

5.7. Target MCU Flash Memory Usage

Program memory on Cortex-M series MCUs is located at address 0x0000_0000 through 0x1FFF_FFFF. The target MCU's program memory is divided into two sections, one for the bootloader and another for the application. On reset, code execution always begins at 0x0000_0000 and this is where the bootloader is located. The size of the bootloader depends on the implementation. The bootloader also stores information about the application image at the end of Flash. Figure 9 shows the target MCU program memory map.

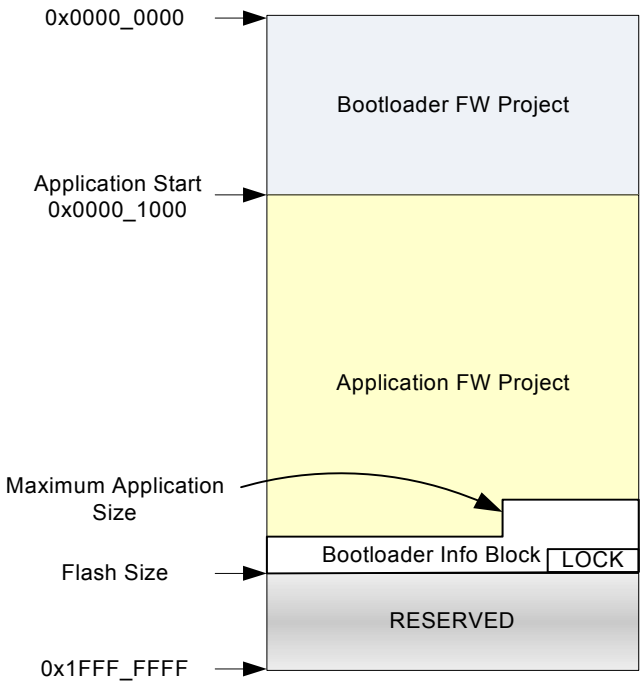

Figure 9. Target MCU Program Memory Map

5.8. Target MCU Bootloader Features

A number of features that improve the robustness of the system and allow it to tolerate unexpected events such as power failures or cable disconnects during the firmware update process are recommended. These fault-tolerance enhancements and other features are described in the following sections.

5.8.1. Self-Update Prevention

To prevent the device from becoming inaccessible over the communication interface, it is recommended that the bootloader be designed to disallow self-updates. This can easily be implemented by including boundary checks on the received firmware image to prevent erasure of the bootloader firmware itself.

5.8.2. Independent Interrupt Vector Table

The Cortex-M architecture makes it very easy to re-direct interrupt vectors. The address of the interrupt vector table is stored in an NVIC register called the vector table offset register. Using a CMSIS compliant library, the offset can be accessed by "SCB->VTOR". The bootloader sets the interrupt vector table offset to the start of the application firmware immediately prior to transferring control to application. Since the bootloader and the application have independent interrupt vector tables, the bootloader itself can utilize interrupts without affecting the interrupt latency of the main application.

5.8.3. Non-Resident Flash Keys

If the MCU experiences an out-of-spec power ramp-up, a noisy external system clock, or other condition which disrupts normal code execution, it is possible for the Flash write/erase routines in the bootloader to be inadvertently

executed. To remove the possibility of Flash corruption due to inadvertent writes and erases, it is recommended that the Flash unlock key codes not reside within the bootloader firmware. In the absence of these key codes, the MCU will not be able to modify the contents of Flash. During a firmware update process, the Flash key codes can be passed to the bootloader through the firmware image file and held in RAM until the firmware update completes. After the firmware update, the bootloader may clear the key codes from RAM, which prevents the possibility of Flash being modified after the new firmware image is programmed.

5.8.4. Page-by-Page CRC Check

To ensure that the image sent is written to Flash memory without any errors, a page-by-page CRC check may be performed during the firmware update process. If the CRC calculated on the contents of flash does not match the CRC of the RAM buffer, the firmware update is aborted, and the device enters the dfuERROR state.

5.8.5. Fail-Safe Bootloader Entry

Under normal circumstances, the bootloader will be entered when application code sets a trigger. If application code ever becomes stuck in a state where it is not able to call the bootloader (due to a firmware bug), a fail-safe bootloader entry method that checks a port pin state on device reset can be implemented. If this GPIO pin is asserted at that time, the bootloader will immediately enter bootload mode.

5.8.6. Application Signature and CRC

In some implementations, a signature and CRC may be appended at the end of the application to allow the bootloader code to verify the application image after each reset.

6. Master Programmer and Data Source Design

The master programmer initiates the firmware update process and transfers the firmware image file stored in the data source to the target MCU. The master programmer can also retrieve the current firmware image from the target MCU for archiving prior to performing a firmware update. The master programmer can be implemented in custom hardware or as a PC application.

6.1. Master Programmer Functional Description

A block diagram of the master programmer is shown in Figure 10. The DFU host state machine communicates with the DFU device state machine over the communication interface to perform firmware uploads and downloads. The data source interface provides a place to read or write the firmware image file. In most implementations, the DFU host state machine will also provide a user interface with progress indicator. This may be in the form of a PC application or an LCD on a custom hardware solution.

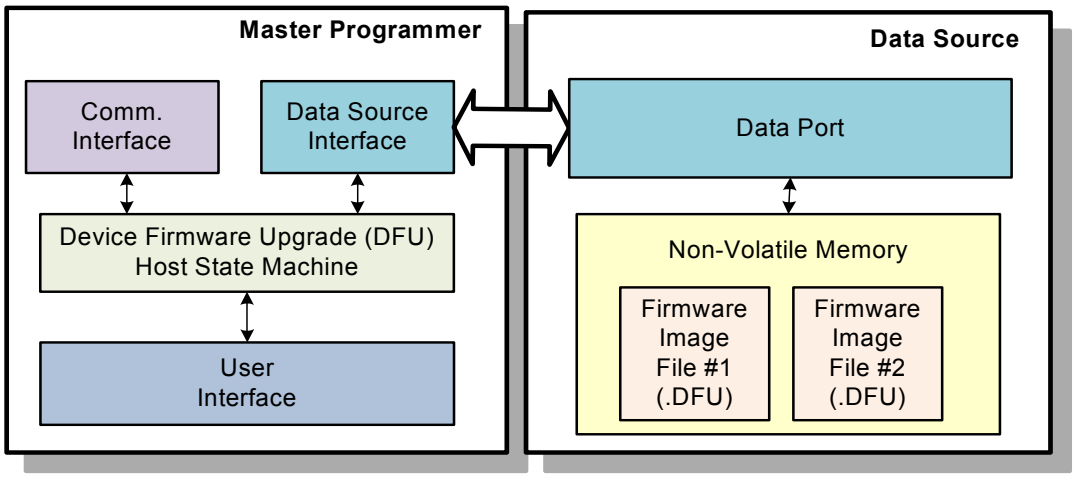

Figure 10. Master Programmer and Data Source

6.2. Communication Interface

The master programmer communication interface should be compatible with the communication interface on the target device.

6.3. Data Source

The data source provides the application firmware image to the master programmer. Example data source implementations include an OS file system, an EEPROM, or an SD flash memory card. The key requirement of the data source is the ability to store and recall firmware image files in non-volatile memory. It is also important that the data source have both read and write capability so that the firmware image file may be updated from time to time or to support device backup and restore operations.

The data source interface can be implemented using a number of protocols that provide the ability to transfer data. It is expected that the data read and written over the data source interface is reliable. The data source interface can vary based on the implementation of the master programmer and the data source with an endless number of implementation possibilities. For example, some systems may benefit from having the latest firmware image file downloaded over a cellular network using a smartphone which is subsequently used to program the target MCU.

6.4. DFU Host State Machine

The DFU host state machine implements the firmware upgrade procedure described in the *USB Device Firmware Upgrade Specification, Revision 1.1,* for a host device. It navigates the DFU device state machine described in detail in *"5.3. DFU Device State Machine" on page 5.* The commands in Table 2 are used to transfer between states to upload or download firmware.

6.5. User Interface

The user interface allows the user to initiate an upload or download operation and provide feedback on the bootloader progress. The user interface will vary by implementation and could be a PC application or an LCD and push-button interface.

7. Making an Application Bootloader-Aware

A stand-alone application that was designed to work on a target MCU can be modified to make it bootloader-aware so that it can co-exist with the bootloader firmware. Because the bootloader does not share any on-chip resources other than code space while the application is active, the modifications needed for the application are minimal. The following steps can be used to make an application bootloader-aware:

1. Modify the starting address to be located at APPLICATION_START. This is typically done using a scatter file.
2. Ensure that the code size does not exceed the maximum application image size.
3. [Optional] Add code to allow the application to trigger a bootload operation.

8. Initial Programming Options

The target MCU needs to be programmed with the bootloader via the SerialWire interface before a firmware update can take place via the chosen communication interface. During development, initial programming can be done using the Silicon Labs IDE and the USB Debug Adapter. For a production environment, many options are available based on production volume and need for serialization. The available programming options can be found here: http://www.silabs.com/products/MCU/Pages/ProgrammingOptions.aspx

There are two ways to initialize the target MCU in a production environment:

1. Program just the bootloader firmware image on the target MCU using the SerialWire programming interface. Application code is subsequently programmed using the bootloader.
2. Program a combined bootloader + application firmware image on the target MCU using the SerialWire programming interface. Application code is programmed at the same time as the bootloader, saving a step in the production flow and allowing the bootloader to be used primary for field updates.

Contact Information

Silicon Laboratories Inc.
400 West Cesar Chavez
Austin, TX 78701
Tel: 1+(512) 416-8500
Fax: 1+(512) 416-9669
Toll Free: 1+(877) 444-3032

Please visit the Silicon Labs Technical Support web page:
https://www.silabs.com/support/pages/contacttechnicalsupport.aspx
and register to submit a technical support request.