

## I<sup>2</sup>C BOOTLOADER IMPLEMENTATION FOR SILICON LABS SiMXXXXX MICROCONTROLLERS

### 1. Introduction

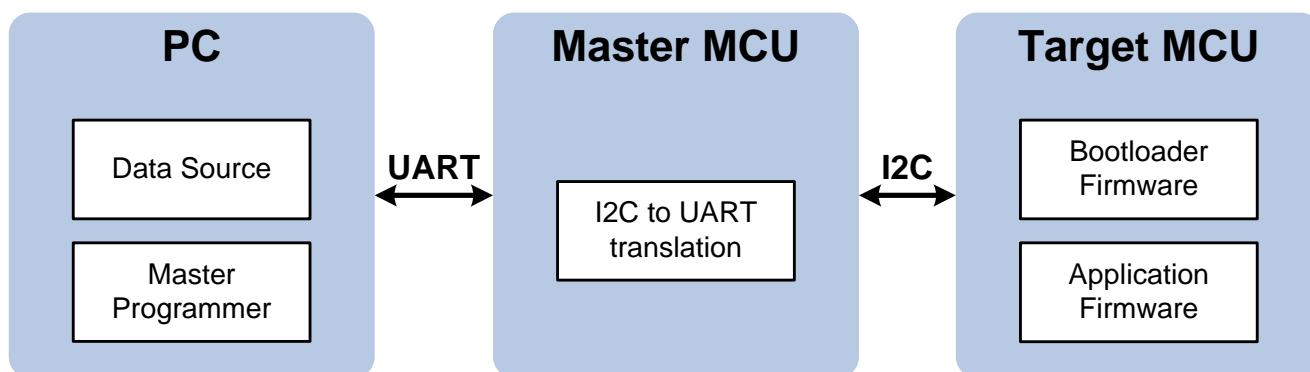
A bootloader enables device firmware upgrades without the need for dedicated, external programming hardware. All Silicon Labs SiMxxxxx MCUs with flash memory are self-programmable, where code running on the MCUs can erase and write other parts of the code memory. A bootloader can be programmed into these 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.

This application note describes an I2C bootloader implementation based on the modular bootloader framework described in Application Note 762 “Modular Bootloader Framework for Silicon Labs SiMxxxxx Microcontrollers”. This document and additional bootloader related application notes are available on [www.silabs.com/32bit-appnotes](http://www.silabs.com/32bit-appnotes).

The bootloader consists of the following components:

- Target MCU
- Master MCU
- Master Programmer
- Data Source

Figure 1 provides a high-level overview of this bootloader system.



**Figure 1. I2C Bootloader Based on the Modular Bootloader Framework**

### 2. Relevant Documentation

Silicon Labs application notes are available on [www.silabs.com/32bit-appnotes](http://www.silabs.com/32bit-appnotes).

- **AN762: Modular Bootloader Framework for Silicon Labs SiMxxxxx Microcontrollers** — Describes the modular bootloader framework, which provides the framework for the I2C bootloader described in this document.
- **AN763: UART and USB Bootloader Implementations for Silicon Labs SiMxxxxx Microcontrollers** — Describes a UART and USB bootloader implementation using the same modular bootloader framework.

## 3. Bootloader Overview

The bootloader consists of the following components:

- **Target MCU** — The end device running an I2C bootloader and application firmware.
- **Master MCU** — Acts as a bridge between the Master Programmer (UART) and the Target MCU (I2C).
- **Master Programmer** — Initiates the firmware update process and transfers the firmware image file stored in the data source to the target MCU.
- **Data Source** — Application firmware image file.

In this implementation of the modular bootloader framework, the Target MCU is an SiM3U167 microcontroller, and data transfer occurs over an I2C communication protocol described in this application note. The Master MCU is an SiM3U167 microcontroller that acts as a bridge, communicating with the Master Programmer over UART and the Target MCU over I2C. The Master Programmer is a PC running a GUI application, and the data source is the Windows File System.

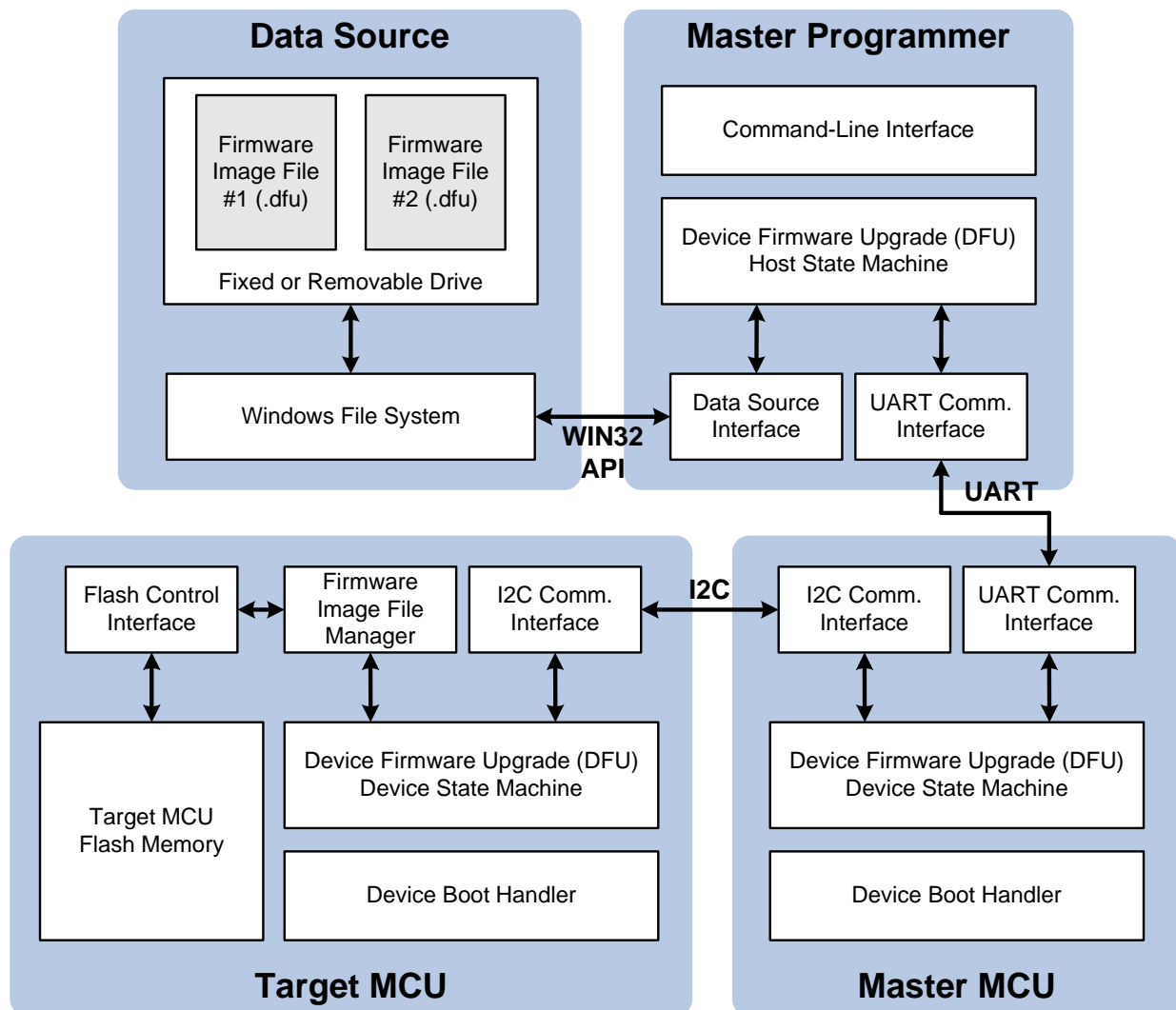


Figure 2. I2C Bootloader Overview



6. Once the Target MCU is programmed following the steps in Section 4.3, connect the USB Debug Adapter to the Master MCU board through its **DEBUG** connector (J31) power it through the **Device USB** port (J13), and move the Master MCU **System Power Select** (SW5) switch to the top USB position.
7. Once both boards are programmed, connect the Master MCU **COM PORT** (J10) to the PC using a mini USB cable.

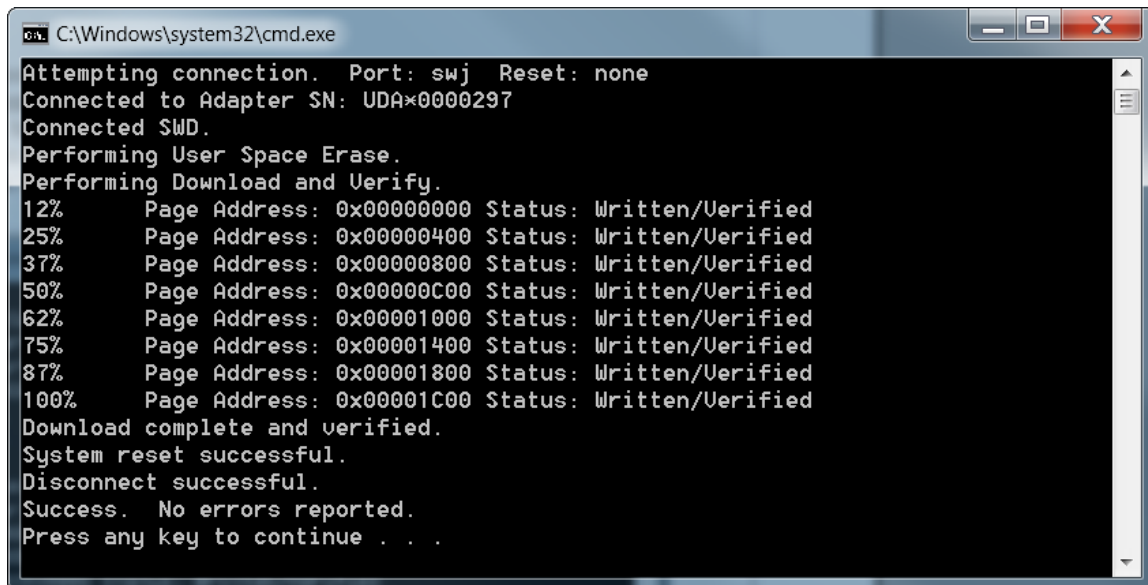
## 4.3. Downloading the Firmware

An uninitialized Target MCU device requires the bootloader firmware to be programmed over the debug interface. Similarly, the Master MCU firmware must be loaded into the Master MCU device. Once the devices are programmed with the appropriate firmware, the application image may be loaded into the Target MCU over the I2C interface.

To program the firmware images to the boards:

1. With the PC connected to the Target MCU board through the USB Debug Adapter, click on the **2.Program\_I2C\_Bootloader.bat** file in the **AN787SWI2C\_Demo** directory to program the firmware image onto the MCU.
2. Swap the debug connection to the Master MCU board.
3. With the PC connected to the Master MCU board through the USB Debug Adapter, click on the **3.Program\_I2C\_Master\_MCU.bat** file in the **AN787SWI2C\_Demo** directory to program the firmware image onto the MCU.

Figure 4 shows the output of the flash programming utility after the Target MCU or Master MCU firmware has been successfully programmed.



```
C:\Windows\system32\cmd.exe
Attempting connection. Port: swj Reset: none
Connected to Adapter SN: UDA*0000297
Connected SMD.
Performing User Space Erase.
Performing Download and Verify.
12%    Page Address: 0x00000000 Status: Written/Verified
25%    Page Address: 0x00000400 Status: Written/Verified
37%    Page Address: 0x00000800 Status: Written/Verified
50%    Page Address: 0x00000C00 Status: Written/Verified
62%    Page Address: 0x00001000 Status: Written/Verified
75%    Page Address: 0x00001400 Status: Written/Verified
87%    Page Address: 0x00001800 Status: Written/Verified
100%   Page Address: 0x00001C00 Status: Written/Verified
Download complete and verified.
System reset successful.
Disconnect successful.
Success. No errors reported.
Press any key to continue . . .
```

Figure 4. Loading the Firmware Images

#### 4.4. Loading the Application Code Using the Graphic Interface

There are two example applications provided to test the bootloader functionality: **Blinky\_Fast** and **Blinky\_Slow**. The two firmware images blink the LEDs on the Target MCU board at different rates to indicate that the bootloader has successfully modified the firmware image.

To download an application image to the Target MCU device:

1. With the PC connected to the Master MCU board using the **COM PORT** interface, click on the **4.Launch\_DFU\_UTILITY.bat** file in the **AN787SWI2C\_Demo** directory to open the GUI software. This software can also be found in **AN787SWHostPC\_Applications\Release**.
2. Select the COM port associated with the device from the drop-down menu.
3. Click the **Query** button to obtain information about the bootloader and the loaded application image.

Figure 5 shows the result of the query command.

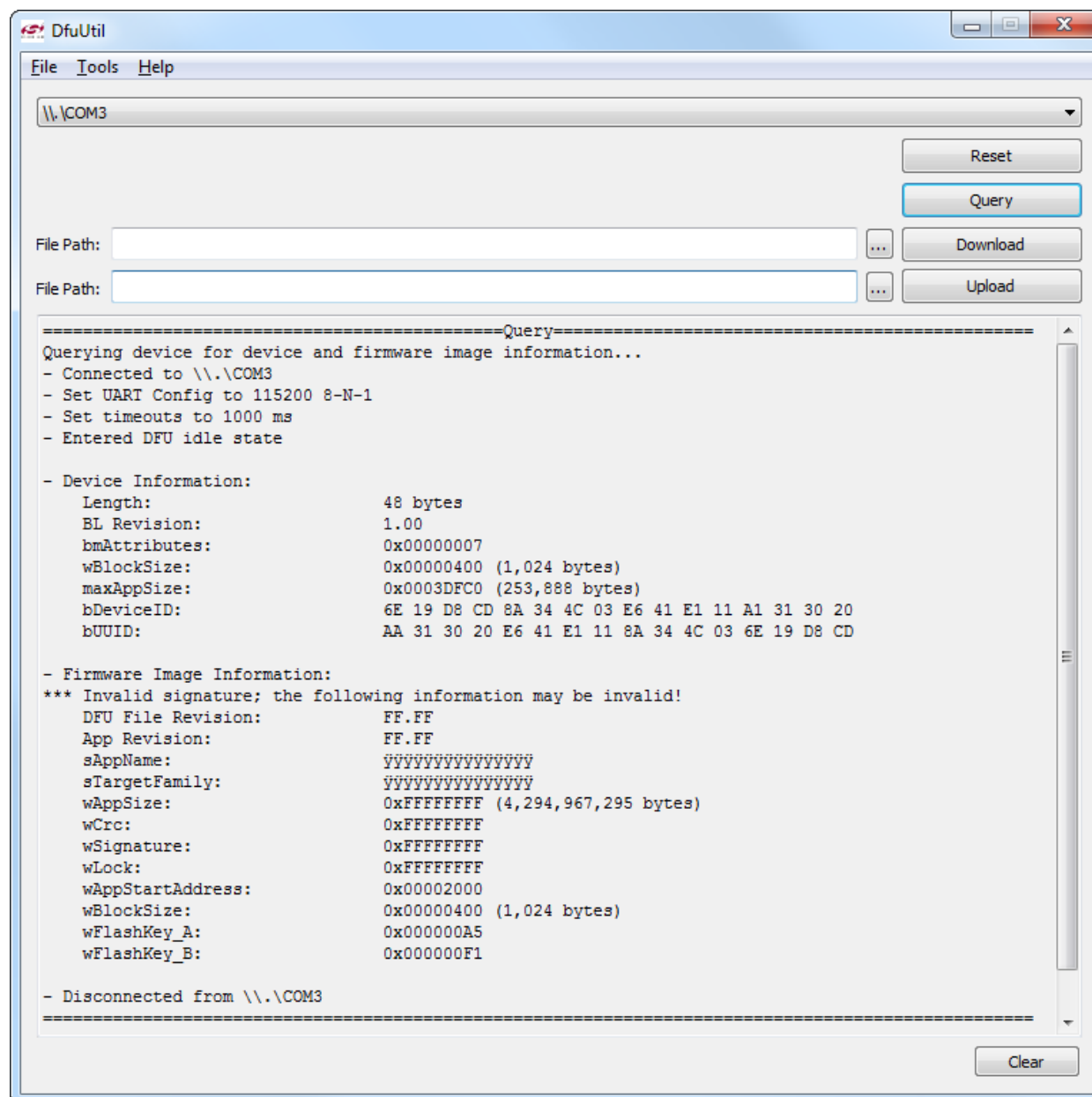
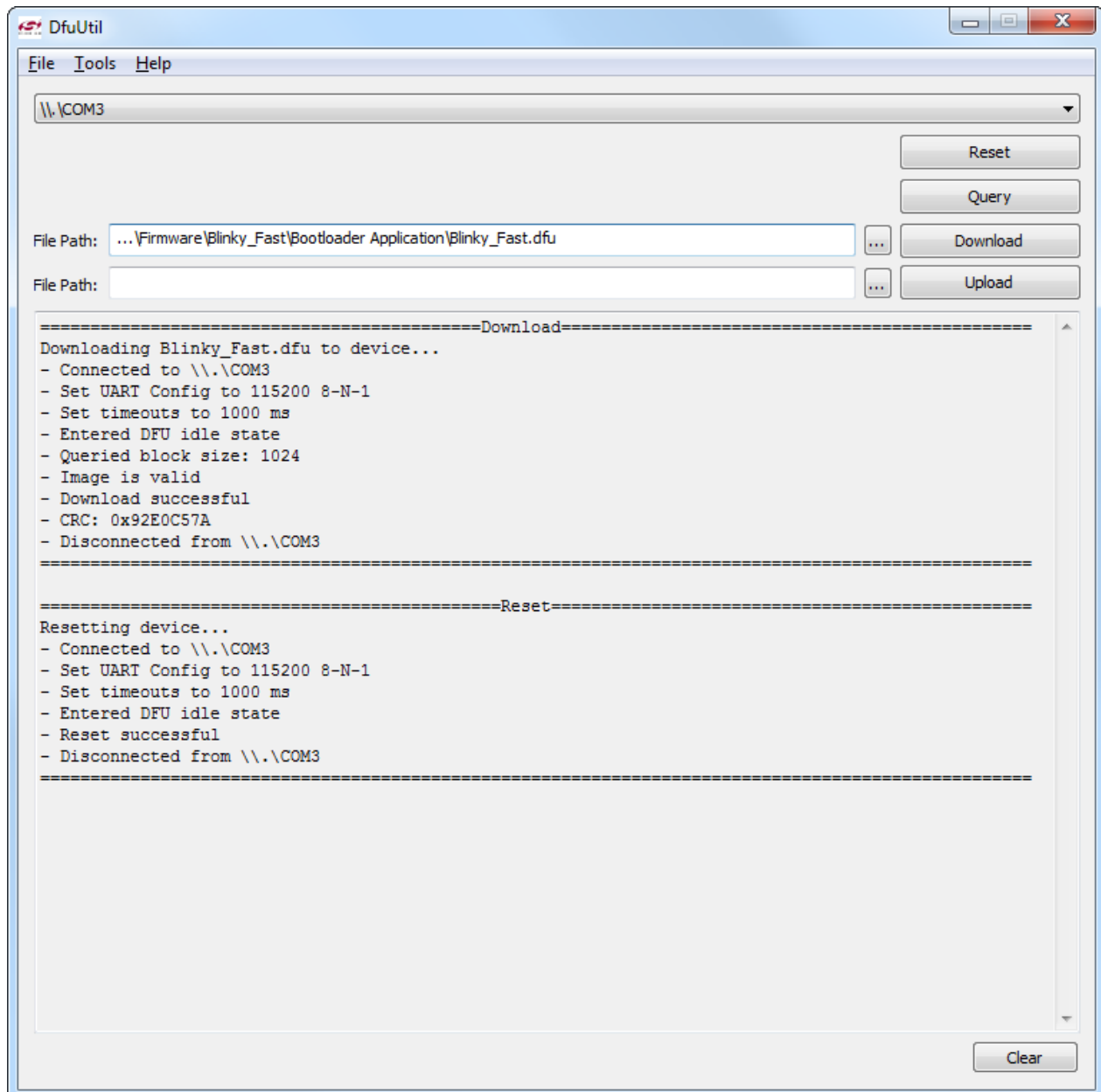


Figure 5. Using the Query Command

4. If the Target MCU has an invalid application image, it will automatically start in bootloader mode. If it does contain a valid application image, it needs to be forced into bootloader mode by holding down SW2 (PB2.8) and pressing and releasing the reset switch (SW1). The Target MCU should now be ready to accept a new firmware image.
5. Browse for the **Blinky\_Fast.dfu** firmware image in the **AN787SW\Firmware\Blinky\_Fast\Bootloader Application** directory and download the image to the Target MCU by clicking the **Download** button. Upon a successful download, click the **Reset** button to begin executing the new code. The LEDs should be blinking at a fast rate.



**Figure 6. Downloading and Resetting the Target MCU**

6. Force the Target MCU into bootloader mode by holding down SW2 (PB2.8) and pressing and releasing the reset switch (SW1). The Target MCU should now be ready to accept a new firmware image.
7. Click the **Query** button and verify that the **sAppName** of the loaded firmware image is **Blinky\_Fast**.

8. Browse for the **Blinky\_Slow.dfu** firmware image in the **AN787SW\Firmware\Blinky\_Slow\Bootloader Application** directory and download the image to the Target MCU by clicking the **Download** button. Upon a successful download, click the **Reset** button to begin executing the new code. The LEDs should be blinking at a slow rate.
9. Force the device into bootloader mode by holding down SW2 (PB2.8) and pressing and releasing the reset switch (SW1) to perform a **Query**, **Upload**, or **Download** command on the device.

## 5. Target MCU Bootloader Implementation

The bootloader for the Target MCU is based on the bootloader framework described AN762. The File Manager and DFU (device firmware update) State Machine are imported unmodified from the framework software. Custom boot handler, communication interface, and flash control interfaces complete the bootloader implementation. Compile time default build options allow the bootloader to be configured for I2C communication. Figure 7 shows a block diagram of the target MCU bootloader.

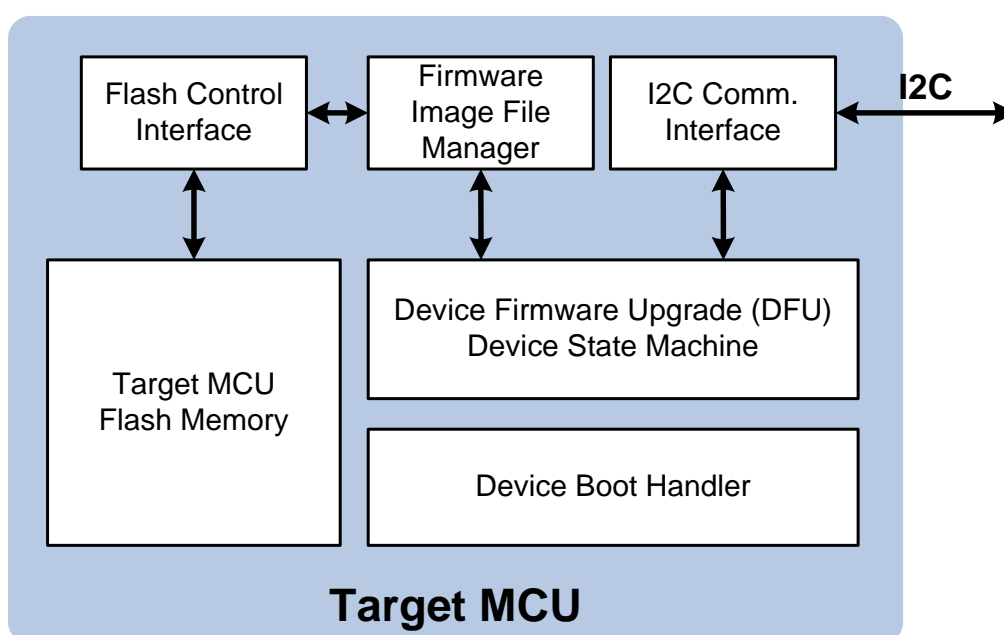


Figure 7. Target MCU Bootloader

### 5.1. Build Options

The framework source code contains support for multiple device families and communications protocols. In this particular implementation, the **MCU\_FAMILY** project variable specifies the device family, and the **COMM\_PROTOCOL** project variable specifies the communication protocol (I2C). The software package includes hex files with these build options set for the SiM3U1xx family and an I2C protocol for use with a SiM3U1xx MCU Card.

Additional project build options, such as the **DEBUG/NDEBUG** define statement, allow the bootloader to print debug statements through the serial wire viewer and can enable, disable, or configure various trigger sources for the bootloader. Build options can be accessed from the project command line (Project Options in µVision4) and from the **device\_userconfig\_sim3u1xx.h** header file. For additional information about framework build options, please see Section 4 of AN762.



## 5.2. Device Boot Handler

The device boot handler performs all the functions required by the framework specification and provides additional functionality to the system. The implementation of the device boot handler can be found in the included **device\_sim3u1xx.c** source file.

### 5.2.1. DEVICE\_Init

The **DEVICE\_Init** routine is called after each device reset and is responsible for initializing the device and checking for the appropriate trigger sources. The **DEVICE\_Init** routine performs the following functions:

1. Disables the watchdog timer and enable the APB clock to all modules.
2. Determines the amount of memory (flash and RAM) in the device and the package option by decoding bits in the device ID or derivative register.
3. Checks all internal, external, and automatic trigger sources to determine if a firmware update is required or has been requested.

Table 2 lists the trigger sources defined in this implementation. This check includes performing a full 32-bit CRC on the application image and verifying that the flash signature written by the bootloader after a successful firmware update operation is present.

**Table 2. Device Boot Handler Trigger Sources**

Type	Source	Enabled by Default
Automatic	Invalid reset vector or stack pointer	Yes
Automatic	Flash signature not found	Yes
Automatic	Application image CRC failure	Yes
External	GPIO trigger pin and any reset	Yes
External	GPIO trigger pin and pin reset	No
External	GPIO trigger pin and POR reset	No
External	Any pin reset	No
Internal	Any system reset	No
Internal	Any software reset	No
Internal	Software Reset and Configuration Word in RAM	No

### 5.2.2. DEVICE\_Restore

The **DEVICE\_Restore** routine restores all device registers modified by **DEVICE\_Init** to their reset values. This includes starting the watchdog timer and restoring the APB clock settings back to their reset value.

### 5.2.3. DEVICE\_InitializeCRC32

The **DEVICE\_InitializeCRC32** routine performs all necessary initializations to begin using the hardware CRC engine for a 32-bit CRC calculation.

### 5.2.4. DEVICE\_UpdateCRC32

The **DEVICE\_UpdateCRC32** routine accepts an 8-bit value and incorporates it into the current 32-bit CRC operation.

### 5.2.5. DEVICE\_ReadCRC32Result

The **DEVICE\_ReadCRC32Result** returns the 32-bit result of the current 32-bit CRC operation.

### 5.2.6. DEVICE\_Fill\_DeviceID\_UUID

The **DEVICE\_Fill\_DeviceID\_UUID** routine reads the Device ID and Unique Identifier from a device-specific location in memory and copies it to a buffer. This function is used by the DFU module when responding to a get information command from the master programmer.

### 5.2.7. DEVICE\_Reset

The **DEVICE\_Reset** routine is typically called in response to a reset request from the master programmer. This function simply performs a software reset.



### 5.2.8. DEVICE\_RedirectInterrupts

The **DEVICE\_RedirectInterrupts** routine is called when transferring control to the user application. The method of interrupt re-direction can vary between devices in the SiMxxxxx family of microcontrollers and is, therefore, implemented as a device-specific function. For the SiM3U1xx implementation, it simply writes the starting address of the application space to the **SCB->VTOR** register.

### 5.2.9. get\_last\_reset\_source

The **get\_last\_reset\_source** routine decodes the reset flags and determines the last reset source. This information is used to detect various bootloader triggers that require a specific reset source. This function varies from the standard si32HAL implementation in that it is optimized for code space.

## 5.3. I2C Comm Interface

The I2C Comm Interface in this implementation is a packet-based protocol that provides guaranteed delivery and reception of error-free data between the DFU state machine and the Master MCU. It meets all the comm interface requirements specified in the bootloader framework, including variable payload size, error checking, acknowledgment, and automatic retransmission. The implementation can be found in the **comm\_i2c\_sim3u1xx.c** file.

### 5.3.1. Packet Format

Figure 8 shows the packet format used in this I2C bootloader protocol. Each packet starts with a start-of-frame character, 0x3A or the ASCII character ':', followed by an 8-bit sequence number. Next, a 16-bit length field is transmitted in little-endian format which specifies the number of bytes in the data field. Next, a CCITT-16 cyclic redundancy check transmitted in little-endian format. The data field containing the packet payload can be zero or more bytes. The CRC calculation includes all packet bytes starting with the start-of-frame until the last byte in the data field. This packet format is used by either side when transmitting information.

Start of Frame	bSequenceNum	wLength	wCRC	Data
':' 0x3A	8-bit sequence number	Length of data field	CCITT-16 CRC	Variable length data field

Figure 8. I2C Frame Format

## 5.3.2. Acknowledgment and Automatic Retransmission

The receiver of a packet is required to return a single-byte acknowledgment packet to the transmitter indicating whether or not the packet was properly received. A receiver will transmit a single byte with value of 0x00 to indicate the packet was properly received and has passed CRC verification. A receiver will transmit a single byte with value of 0xFF to indicate the packet was not properly received due to its length or bit errors detected during CRC verification. If the receiver does not send an acknowledge packet after a programmed timeout, the transmitter should assume the packet was lost. Figure 9 shows an example packet flow under various normal and error conditions.

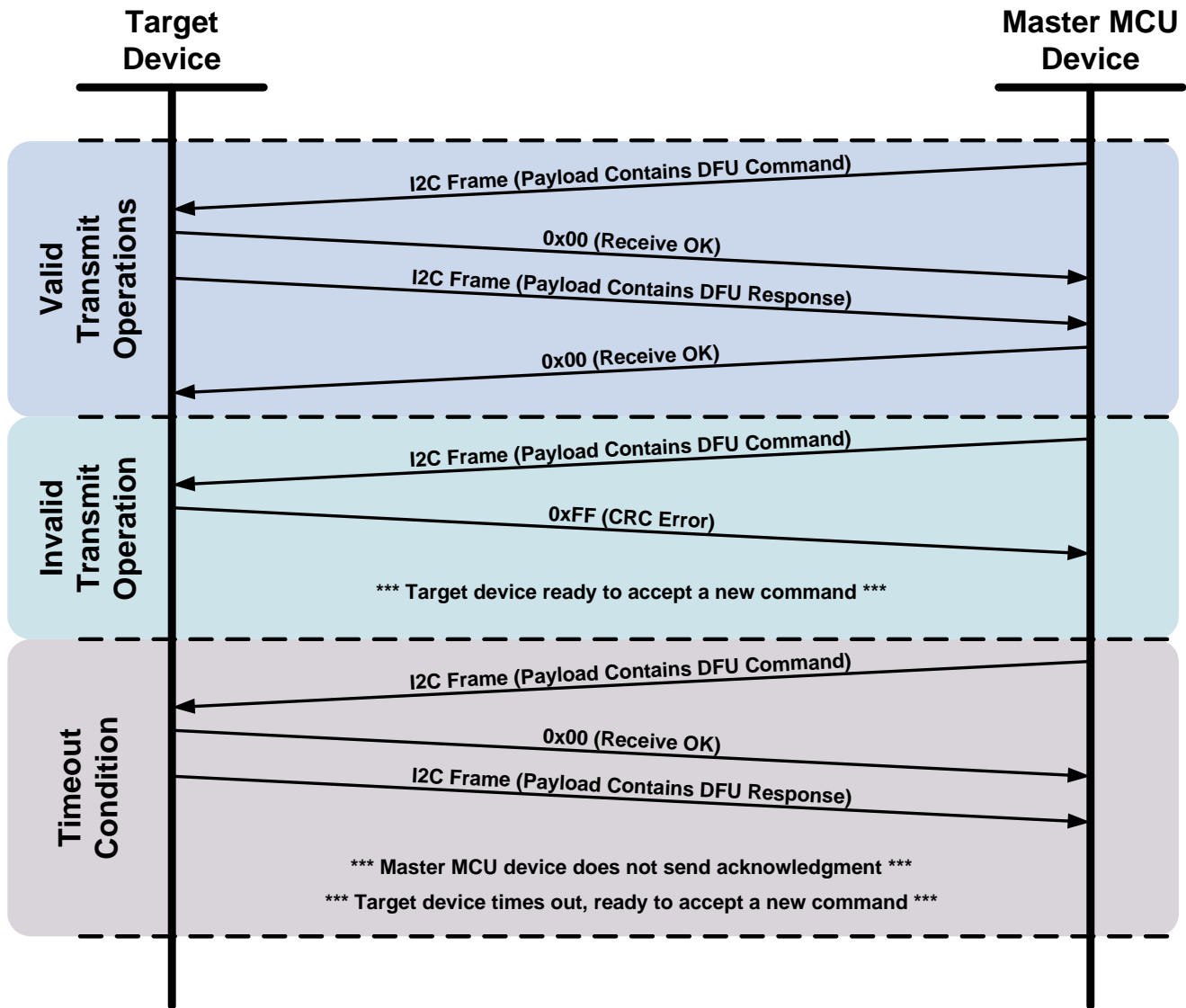


Figure 9. Example I2C Packet Flow

## 5.4. Flash Control Interface

The Flash Control Interface is defined in the `flctl_sim3u1xx.c` source file and provides low-level flash write and erase functionality. The flash keys necessary to initiate any flash write or erase operation are not stored in the Target MCU and are passed in by the master programmer at the beginning of a firmware update in the information block (Block 0) of the DFU firmware image file. The flash keys are stored in RAM and erased upon the successful or unsuccessful termination of the firmware update.

## 6. Master MCU Implementation

The Master MCU firmware is based on the bootloader framework described in AN762. The File Manager and DFU (device firmware update) State Machine are imported from the framework software with a custom boot handler and two communication interfaces. Compile time build options allow the bootloader to be configured for I2C communication. Master MCU firmware has similar structure to the Target MCU bootloader firmware. The main difference is the Master MCU firmware contains a UART communication interface. This UART communication interface has a specific UART packet format and communication packet flow. Figure 10 shows a block diagram of the Master MCU firmware.

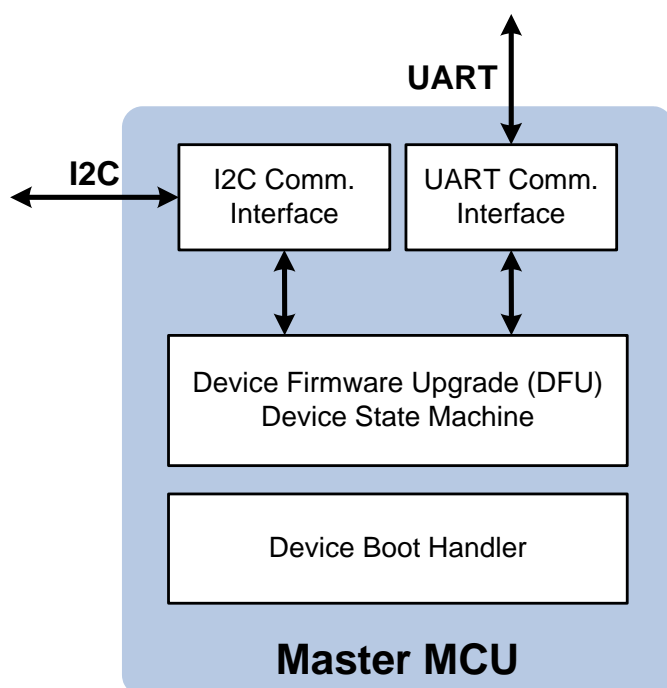


Figure 10. Master MCU Firmware

### 6.1. Device Boot Handler

The device boot handler performs all the functions required by the framework specification and provides additional functionality to the system. The implementation of the device boot handler can be found in the `device_sim3u1xx.c` source file.

## 6.2. UART Comm Interface

The UART Comm Interface in this implementation is a packet-based protocol that provides guaranteed delivery and reception of error-free data between the DFU state machine and the master programmer. It meets all the comm interface requirements specified in the bootloader framework including variable payload size, error checking, acknowledgment, and automatic retransmission. The implementation can be found in the `comm_uart_sim3u1xx.c` source file.

### 6.2.1. Packet Format

Figure 11 shows the packet format used in the UART Comm Interface protocol. Each packet starts with a start-of-frame character, 0x3A or the ASCII character ':', followed by an 8-bit sequence number. Next, a 16-bit length field is transmitted in little-endian format which specifies the number of bytes in the data field. The data field containing the packet payload can be zero or more bytes and is always terminated by a CCITT-16 cyclic redundancy check transmitted in little-endian format. The CRC calculation includes all packet bytes starting with the start-of-frame until the last byte in the data field. This packet format is used by either side when transmitting information.

Start of Frame	bSequenceNum	wLength	Data	wCRC
':' 0x3A	8-bit sequence number	Length of data field	Variable length data field	CCITT-16 CRC

**Figure 11. UART Frame Format**

### 6.2.2. Acknowledgment and Automatic Retransmission

The receiver of a packet is required to return a single-byte acknowledgment packet to the transmitter indicating whether or not the packet was properly received. A receiver will transmit a single byte with value of 0x00 to indicate the packet was properly received and has passed CRC verification. A receiver will transmit a single byte with value of 0xFF to indicate the packet was not properly received due to its length or bit errors detected during CRC verification. If the receiver does not send an acknowledge packet after a programmed timeout, the transmitter should assume the packet was lost. The transmitter should automatically re-transmit any packet that was not successfully received and acknowledged by the receiver. Figure 12 shows an example packet flow under various normal and error conditions.

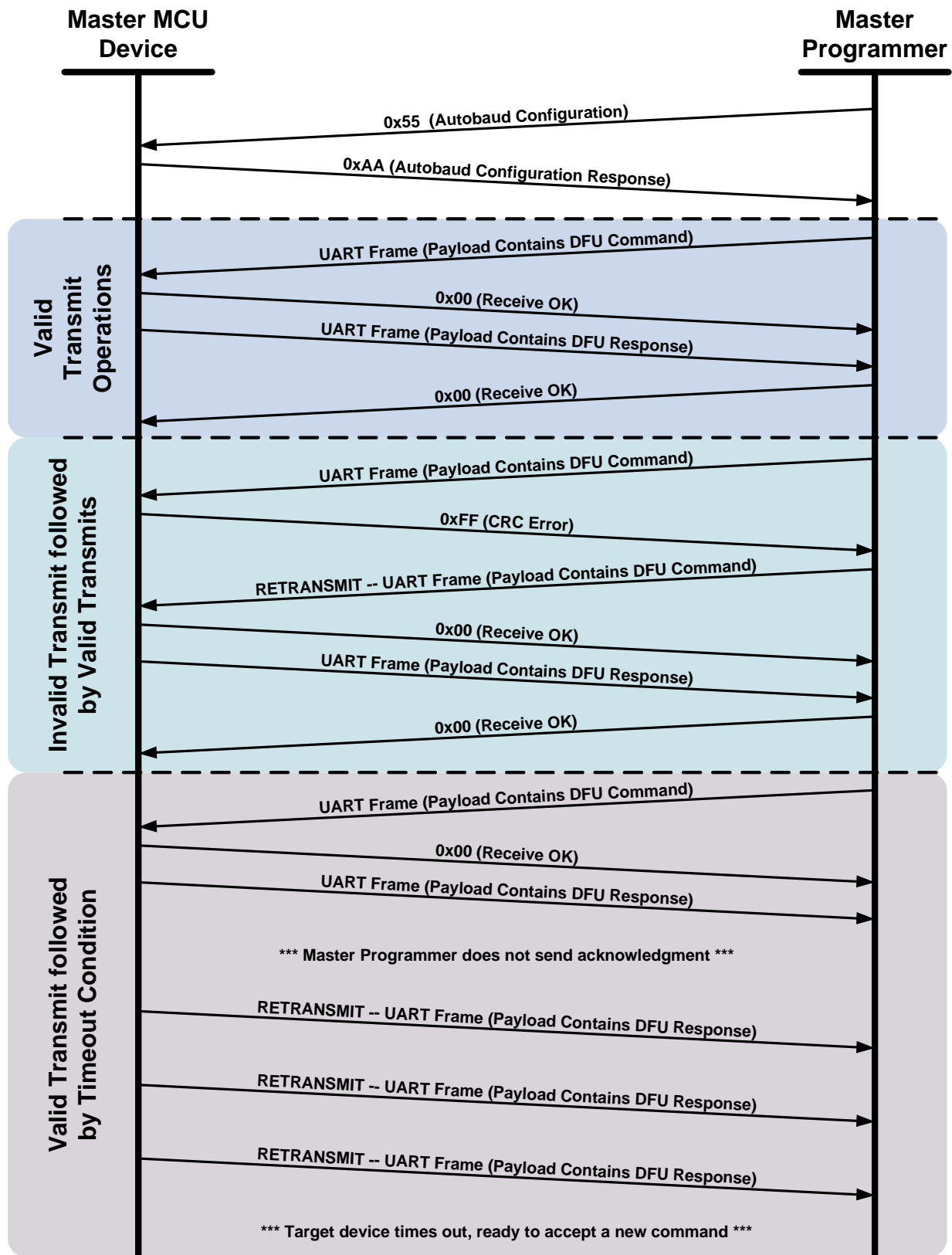


Figure 12. Example UART Packet Flow

## 6.2.3. Baud Rate

The UART interface implementation uses auto-baud detection to configure the baud rate. The master programmer sends a 0x55 (or ASCII 'U') the first time it communicates with the Master MCU. Upon reception of this character, the Master MCU will configure its own baud rate to match the master programmer's baud rate and will transmit an 0xAA character back to the master programmer. Once the Master MCU's baud rate has been set, it remains set and cannot be changed until the next reset.

## 7. Master Programmer and Data Source Implementation

The master programmer and data source are implemented using a PC. The comm interface, DFU state machine, and data source interface are incorporated into a dynamically linked library named **SLAB\_DFU.dll**. The comm interface implementation in this DLL supports guaranteed data transfer over UART or USB. Two user interfaces have been developed for interfacing with the DLL. The **DfuUtil.exe** application provides a graphical user interface for performing firmware updates. The **DfuUtilCL.exe** application provides the same functionality in a command-line application that can be easily automated.

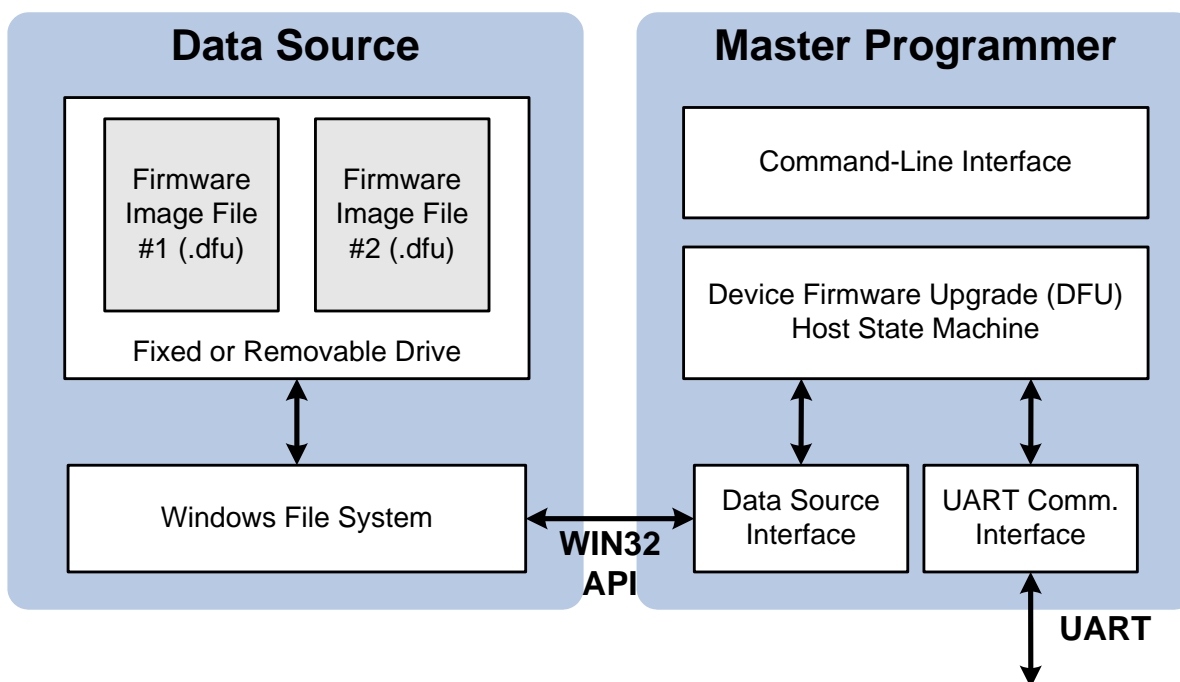


Figure 13. Master Programmer and Data Source

### 7.1. Master Programmer Library (SLAB\_DFU.dll)

The Silicon Labs Device Firmware Upgrade (DFU) Library provides a C application programming interface (API) to transfer the contents of memory from SiMxxxxx MCUs using USB or UART. The library consists of a low-level API to interface directly at the DFU device class level via USB control transfers or UART frame transfers. The library also provides a higher level API to perform full device image uploads and downloads.

### 7.2. Master Programmer Graphical User Interface (DfuUtil.exe)

The master programmer GUI uses **SLAB\_DFU.dll** to communicate with bootloader devices using either the UART or USB comm interface. The program supports Windows XP and later 32-bit and 64-bit with two sets of binaries. The program has four main modes of operation:

1. **Reset** — Send the reset command to the device. If no bootloader trigger sources are present, then the firmware application will run. Otherwise, the device will reset in bootloader mode.

2. **Query** — Upload block 0 from the device to retrieve firmware image information.
3. **Download** — Download the selected binary DFU image to the device code memory.
4. **Upload** — Upload a binary DFU image from the device code memory and save to the specified file.

The master programmer GUI has several options accessed from the file menu by selecting **Tools**→**Options**. These options are persistent and saved in an **options.txt** file.

The program options are:

- **GUID** — Enter the GUID specified in the WinUSB driver INF file to display USB devices that have loaded the driver.
- **Baud** — Enter the baud rate used for UART communications.
- **Timeout** — Enter the control transfer timeout in milliseconds. Enter a larger timeout when using the UART interface with slow baud rates.
- **Check image for device compatibility before download** — Validate the specified binary DFU image on the PC before transferring it to the bootloader. When enabled, the PC checks the following:
  - File size must be a multiple of the block size and be the correct size including block 0 and application firmware blocks.
  - Firmware image application size must not be greater than the maximum application size reported by the device.
  - The signature field must be valid in the image.
  - The application start address must match the address reported by the device.
  - The image block size must match the block size reported by the device.
  - The application CRC must be valid using CRC-32.
- **Reset after download** — Issue a reset command to the device after a successful download to run the firmware image.

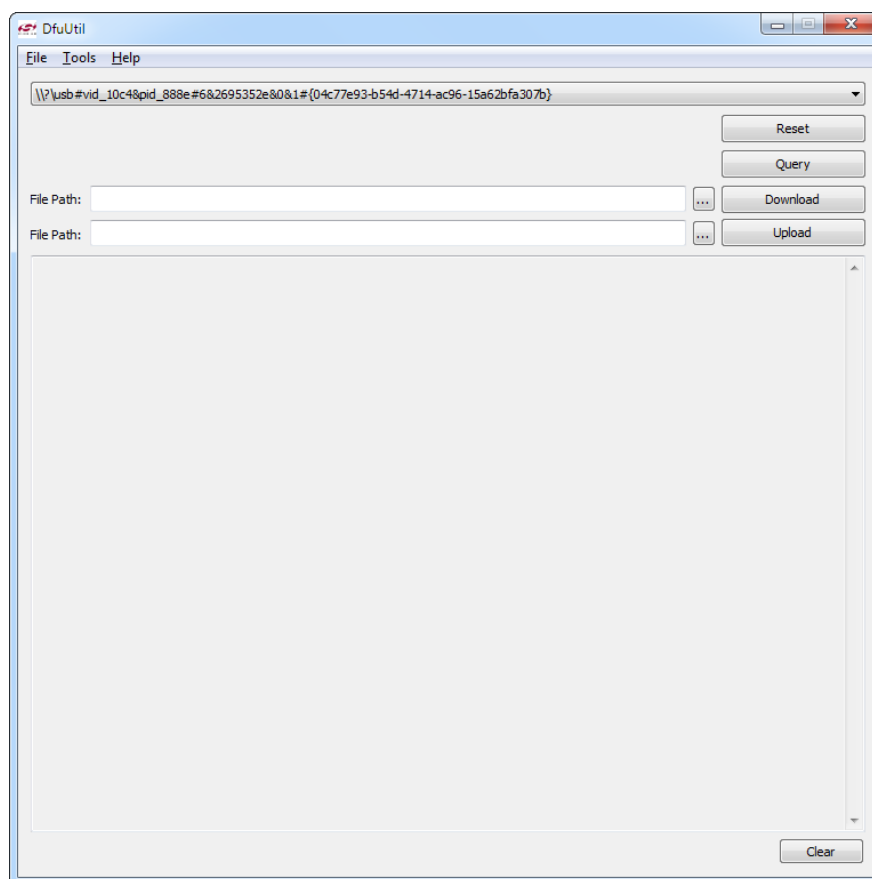


Figure 14. GUI DFU Programmer (DfuUtil.exe)



### 7.3. Master Programmer Command-Line User Interface (DfuUtilCL.exe)

The master programmer command-line program provides the same functionality as the GUI. Run **DfuUtilCL.exe** from the command line with no arguments to display the usage text as seen in Figure 15.

```
Administrator: C:\Windows\system32\cmd.exe
C:\hg32\32bit\DFU\Release\Release\Host\PC_Applications\Release\Win32>DfuUtilCL.exe
Silicon Laboratories DFU Utility 1.0
SLAB_DFU.dll 1.0
=====
Description:
Communicates with a device running the modular DFU bootloader
to query, reset, upload, or download a DFU image.

Usages:
DfuUtilCL -listdevices
DfuUtilCL -query
DfuUtilCL -reset
DfuUtilCL -upload -image<imagepath>
DfuUtilCL -download -image<imagepath> [-checkimage] [-resetafter]]

Communication Parameters:
[-index<deviceindex>] [-path<devicepath>]
[-guid<guidstring>]
[-baud<baudrate>]
[-timeout<timeoutms>]

Parameter Descriptions:
-listdevices      Shows a list of devices, including device index and path
-query           Indicates that the utility will return information about the
                 current image on the DFU device
-reset           Indicates that the utility will reset the bootloader
-upload         Indicates that the utility will upload the DFU image from
                 the DFU device to the specified DFU file
-download       Indicates that the utility will download the specified DFU
                 image to the DFU device
-checkimage      Indicates that the utility will check to see if
                 the application firmware is compatible with the DFU image
                 before downloading
-resetafter      Resets the device after successfully downloading a DFU image
                 to the device
-image          Indicates that a DFU image will be specified
-imagepath       Specifies the DFU image file name for -upload or -download
-index          Indicates that a device index will be specified
-deviceindex     Specifies which DFU device to connect to by device index
-path           Indicates that a device path will be specified
-devicepath      Specifies which DFU device to connect to by device path
-guid           Indicates that a GUID for USB DFU device filtering will be
                 specified
-guidstring      Specifies the GUID for DFU device filtering
-baud           Indicates that a UART baud rate will be specified
-baudrate        Specifies the UART baud rate in bps
-timeout         Indicates that a transfer timeout will be specified
-timeoutms       Specifies the transfer timeout in milliseconds

Default Parameter Values:
deviceindex      - 0
guidstring       - 04C77E93-B54D-4714-AC96-15A62BFA307B
baudrate         - 115200
timeoutms        - 1000

Remarks:
1. Either the -index or the -path switch may be used, but not both.
2. Exactly one of the -listdevices, -query, -reset, -upload, or -download
   switches must be used.
3. By default, this utility will automatically connect to the first DFU
   device by device index (0). Use DfuUtilCL -listdevices to get a list
   of devices. Next, use the -index or -path argument to specify which device
   to connect to.

Examples:
DfuUtilCL -listdevices
DfuUtilCL -query
DfuUtilCL -reset
DfuUtilCL -upload -image(upload.dfu) -path(\\.\COM4) -baud(230400) -timeout(900)
DfuUtilCL -download -image(download.dfu) -index(1)
```

Figure 15. Command-Line DFU Programmer (DfuUtilCL.exe)

#### 7.3.1. Examples

To display a list of devices including device index and device path, use:

```
DfuUtilCL -listdevices
```

To query a device for device and firmware image information for a UART bootloader attached to COM4 using 230400 8-N-1, use:

```
DfuUtilCL -query -path(\\.\COM4) -baud(230400)
```

To download an image (**blinky.dfu**) to the second device in the device list, use:

```
DfuUtilCL -download -image(blinky.dfu) -index(1)
```

To upload an image (**image.dfu**) from the first device in the device list using the default options, use:

```
DfuUtilCL -upload -image(image.dfu)
```

## 8. Creating Bootloader-Aware Applications

In order to use the bootloader to load a firmware image onto the Target MCU, create a new project or modify an existing project and relocate the starting code memory address from 0x0000\_0000 to the application start address (for example, 0x0000\_2000).

### 8.1. Relocating the Application Starting Address

The two included example firmware applications, **Blinky\_Fast** and **Blinky\_Slow**, have been modified from the SiM3U1xx Blinky example in the si32 HAL.

#### 8.1.1. Precision32 IDE

To relocate the starting memory address for a project in Precision32 IDE:

1. Create a new project or import an existing project into Precision32 IDE.
2. Copy **linker\_sim3u1xx\_p32.ld** from the si32 SDK (for example: **C:\SiLabs\32bit\si32-1.1.1\si32Hal\sim3u1xx**) to the project directory.
3. Under the project MCU Linker Target settings, un-check **Manage linker script** and enter the path to the new copy of **linker\_sim3u1xx\_p32.ld** in the **Linker script** text box. The example project has a new **Bootload Application** build configuration that relocates the application start address and generates a hex file.

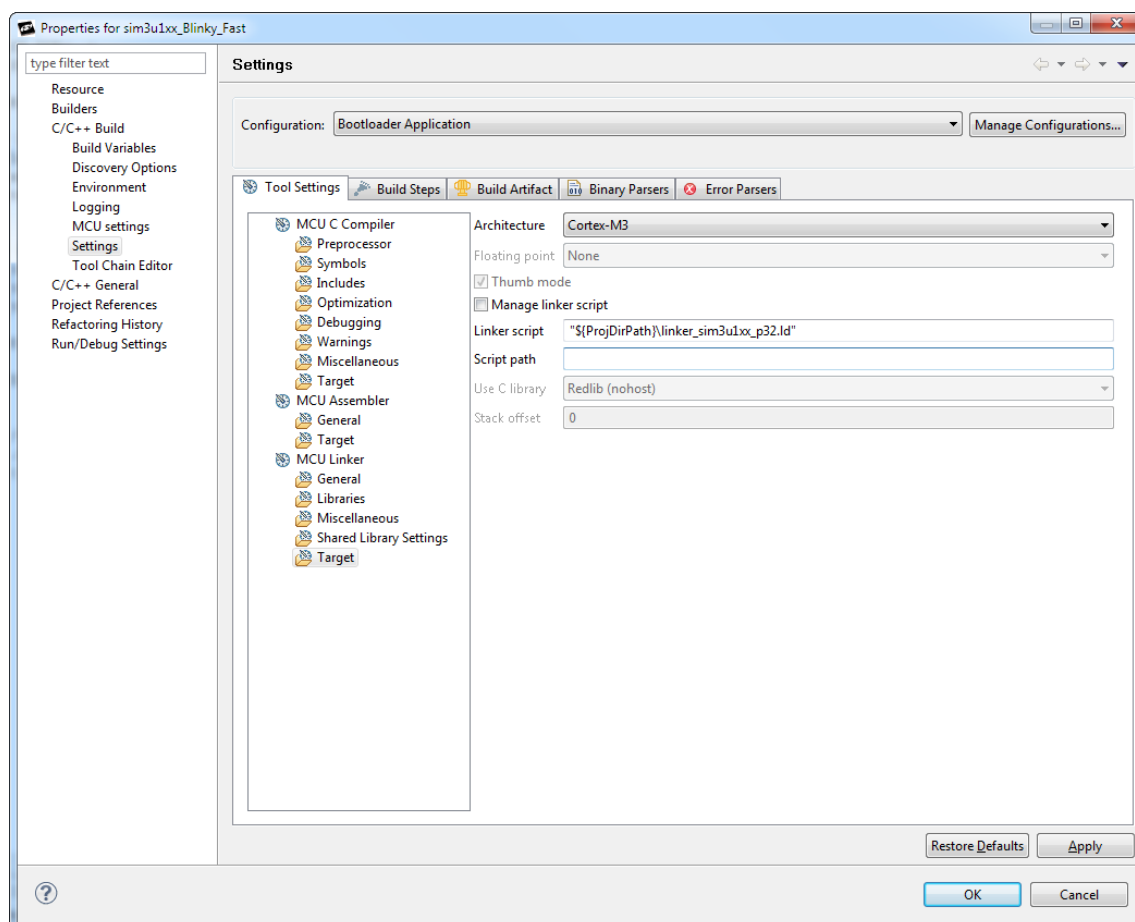


Figure 16. Precision32 Linker Options

4. Modify the new copy of **linker\_sim3u1xx\_p32.ld**. Under the **MFlash256 MEMORYregion**, change the **ORIGIN** variable to the application start address and decrement the **LENGTH** variable by the bootloader size (application start address).

## 8.1.2. Keil µVision

To relocate the starting memory address for a project in Keil µVision:

1. Copy **linker\_sim3u1xx\_arm.sct** and **myLinkerOptions.sct** from the si32 SDK (for example: **C:\SiLabs\32bit\si32-1.1.1\si32Hal\sim3u1xx**) to the project directory.
2. Open the Keil project and open the target options by clicking the **Target Options...** button on the toolbar.
3. Click the **Linker** tab.
4. Modify the **Scatter File** path to point to the new copy of **linker\_sim3u1xx\_arm.sct**.
5. Click the **Edit...** button to open the scatter file in the document viewer.
6. Increase **SI32\_MCU\_FLASH\_BASE** by the bootloader size (application start address).
7. Decrease **SI32\_MCU\_FLASH\_SIZE** by the bootloader size (application start address).

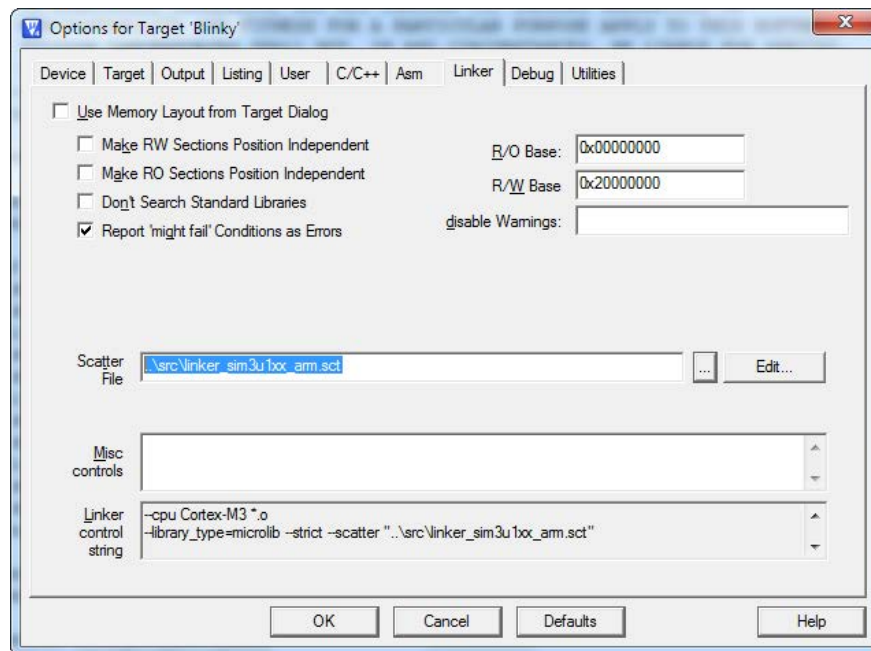


Figure 17. Keil Linker Options

## 8.2. Generating a Hex Image

The master programmer software expects binary DFU files to download and upload. The application firmware must first be built into a hex file (ASCII) and then converted to a binary DFU file using the **HexToDfu.exe** command line program.

### 8.2.1. Precision32 IDE

The **arm-none-eabi-objcopy** executable can be used in the Precision32 IDE to translate an output object file into a hex file format.

To create a hex file:

1. Right-click on the project name in the Project view and select **Options**.
2. Go to **C/C++Build→Settings** and select the **Build Steps** tab.
3. The command-line entered in the **Post-build Steps** section is run after build. Note that '#' begins a comment and that no commands after the '#' sign will be run.
4. Add the following command to **Post-build Steps**:

```
arm-none-eabi-objcopy -O ihex "${BuildArtifactFileName}"  
"${BuildArtifactFileName}.hex";
```

### 8.2.2. Keil $\mu$ Vision

To generate a hex file when building a project in Keil:

1. Open the project file in  $\mu$ Vision.
2. Open the target options by clicking the **Target Options...** button on the toolbar.
3. Click the **Output** tab.
4. Check the **Create HEX File** check box to enable hex file generation.
5. Build the project.

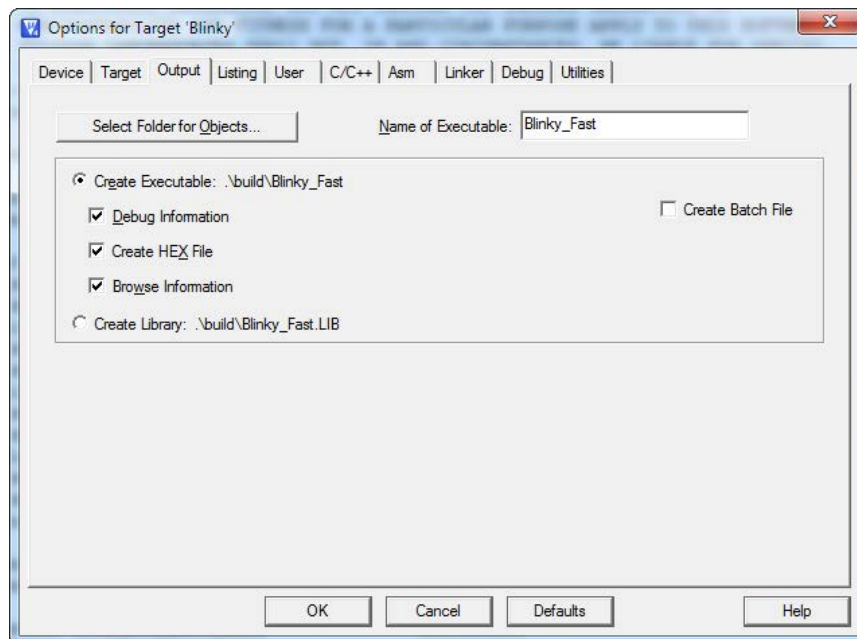


Figure 18. Keil Output Options

### 8.3. Generating a DFU file from a Hex File (HexToDfu.exe)

Once a hex image has been generated with the correct application start address, the next step is to convert the hex image to a binary DFU file using the **HexToDFU** command line program. Run **HexToDfu.exe** from the command line with no arguments to display the usage text as seen in Figure 19.

```
Administrator: C:\Windows\system32\cmd.exe

C:\hg32\32bit\DFU\Release\Release\Host\PC_Applications\Release\Win32>HexToDfu.exe
Silicon Laboratories Hex to DFU File Converter 1.0
=====

Converts an Intel hex file to a Silicon Labs Device
Firmware Upgrade (DFU) file.

HexToDfu -listparts

-listparts      Displays a list of all supported part numbers

HexToDfu [drive:]path\input [[drive:]path\output]
[-part<partnumber>] [-family<target>] [-appname<name>]
[-appversion<version>] [-locked]

input          Specifies the Intel hex file to convert (*.hex).
output         Specifies the DFU file to create (*.dfu).
-part          Indicates that a part number will be specified.
partnumber     Specifies the part number that the created
               DFU image will be used for (see -listparts).
-family        Indicates that a target device family will be
               specified.
target         Specifies the family name that the DFU image is
               compatible with.
-appname       Indicates that a firmware application name will be
               specified.
name           Specifies the firmware application name (15-characters).
-appversion    Indicates that a firmware application version will be
               specified.
version        Specifies the firmware application version
               (BCD major.minor).
-locked        Indicates that the device will be locked after the image
               is downloaded.

Defaults:
output         - input file name with the .dfu extension
part           - SiM3U16x
family         - SiM3U1xx
name           -
version        - 1.00
-locked        - not set

Examples:
HexToDfu -listparts
HexToDfu blinky.hex blinky.dfu -part(SiM3U16x) -family(SiM3U1xx)
               -appname(Blinky) -appversion(1.00) -locked
```

Figure 19. Hex Image to DFU Converter (HexToDfu.exe)

#### 8.3.1. Examples

To display a list of supported part names, use:

```
HexToDfu -listparts
```

The converter uses the part name to generate some of the block 0 information, including flash keys and other device specific information.

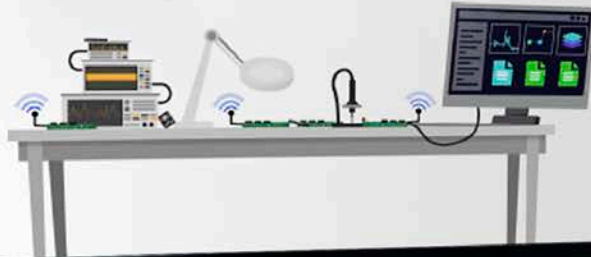
To convert a hex image for **Blinky\_Fast.hex**, use:

```
HexToDfu Blinky_Fast.hex Blinky_Fast.dfu -part(SiM3U16x) -
family(SiM3U1xx) -appname(Blinky_Fast) -appversion(1.00)
```

The part name is only used by the converter, whereas the family string is embedded in the DFU image for verification by the bootloader. The **appname** string and **appversion** number can be retrieved by the **DfuUtil** programs using the **Query** command.

Silicon Labs

# Simplicity Studio™4



## Simplicity Studio

One-click access to MCU and wireless tools, documentation, software, source code libraries & more. Available for Windows, Mac and Linux!



**IoT Portfolio**  
[www.silabs.com/IoT](http://www.silabs.com/IoT)



**SW/HW**  
[www.silabs.com/simplicity](http://www.silabs.com/simplicity)



**Quality**  
[www.silabs.com/quality](http://www.silabs.com/quality)



**Support and Community**  
[community.silabs.com](http://community.silabs.com)

### Disclaimer

Silicon Labs intends to provide customers with the latest, accurate, and in-depth documentation of all peripherals and modules available for system and software implementers using or intending to use the Silicon Labs products. Characterization data, available modules and peripherals, memory sizes and memory addresses refer to each specific device, and "Typical" parameters provided can and do vary in different applications. Application examples described herein are for illustrative purposes only. Silicon Labs reserves the right to make changes without further notice and limitation to product information, specifications, and descriptions herein, and does not give warranties as to the accuracy or completeness of the included information. Silicon Labs shall have no liability for the consequences of use of the information supplied herein. This document does not imply or express copyright licenses granted hereunder to design or fabricate any integrated circuits. The products are not designed or authorized to be used within any Life Support System without the specific written consent of Silicon Labs. A "Life Support System" is any product or system intended to support or sustain life and/or health, which, if it fails, can be reasonably expected to result in significant personal injury or death. Silicon Labs products are not designed or authorized for military applications. Silicon Labs products shall under no circumstances be used in weapons of mass destruction including (but not limited to) nuclear, biological or chemical weapons, or missiles capable of delivering such weapons.

### Trademark Information

Silicon Laboratories Inc.®, Silicon Laboratories®, Silicon Labs®, SiLabs® and the Silicon Labs logo®, Bluegiga®, Bluegiga Logo®, Clockbuilder®, CMEMS®, DSPLL®, EFM®, EFM32®, EFR®, Ember®, Energy Micro, Energy Micro logo and combinations thereof, "the world's most energy friendly microcontrollers", Ember®, EZLink®, EZRadio®, EZRadioPRO®, Gecko®, ISOModem®, Precision32®, ProSLIC®, Simplicity Studio®, SiPHY®, Telegesis, the Telegesis Logo®, USBXpress® and others are trademarks or registered trademarks of Silicon Labs. ARM, CORTEX, Cortex-M3 and THUMB are trademarks or registered trademarks of ARM Holdings. Keil is a registered trademark of ARM Limited. All other products or brand names mentioned herein are trademarks of their respective holders.



**SILICON LABS**

Silicon Laboratories Inc.  
400 West Cesar Chavez  
Austin, TX 78701  
USA

<http://www.silabs.com>