UART and USB Bootloader Implementations 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”, 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. This application note describes a UART and USB bootloader implementation based on the modular bootloader framework described in Application Note 762 “Modular Booloader Framework for Silicon Labs SiMxxxxx Microcontrollers”. Additional bootloader related application notes are available at http://www.silabs.com/products/MCU/Pages/ApplicationNotes.aspx

2. Bootloader Overview

The bootloader consists of the following components:

- Target MCU
- Master programmer
- Data source

The Target MCU in this bootloader implementation is an SiM3U167 microcontroller, and data transfer occurs over a UART or USB communication protocol described in this application note. The Master Programmer in this implementation is a PC running a GUI application and the data source is the Windows File System.

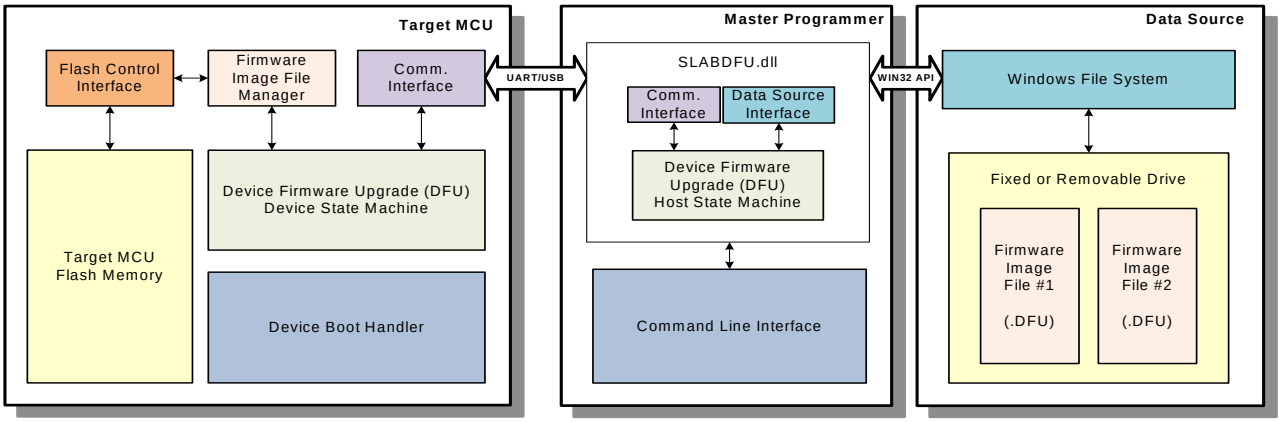


Figure 1. UART/USB Bootloader Based on AN762 Framework

3. How to Use the UART Bootloader (Demonstration)

The following steps demonstrate how to use the UART Bootloader to load two different applications, Blinky_Fast and Blinky_Slow, into the Target MCU.

3.1. Software Setup

Ensure that the latest version of the Silicon Labs CP210x VCP driver is installed. If you are not sure if the VCP driver is installed on your PC, please run the installer available from http://www.silabs.com/products/mcu/Pages/USBtoUARTBridgeVCPDrivers.aspx.

3.2. Hardware Setup

1. Connect the communication port of an SiM3U1xx MCU card to the PC using a mini USB cable.
2. Connect the debug port of an SiM3U1xx MCU Card to a USB debug adapter using a ribbon cable and connect the USB debug adapter to the PC using a standard USB cable.
3. Power the SiM3U1xx MCU Card through its Device USB port (J13) using a mini USB cable. If the PC does not have 3 available USB ports, then the communication port and the debug port may share a single USB port as they will not be used simultaneously.

3.3. Flashing the Bootloader

An uninitialized device requires the bootloader to be flashed over the debug interface. Once the device is programmed with a bootloader, the application image may be loaded over the UART interface. Click on the 2.*Flash_USB_Bootloader.bat* batch file to flash the MCU with the bootloader firmware using the command line flash programming utility. Figure 2 shows the output of the flash programming utility after the bootloader has been successfully programmed into the Target MCU.

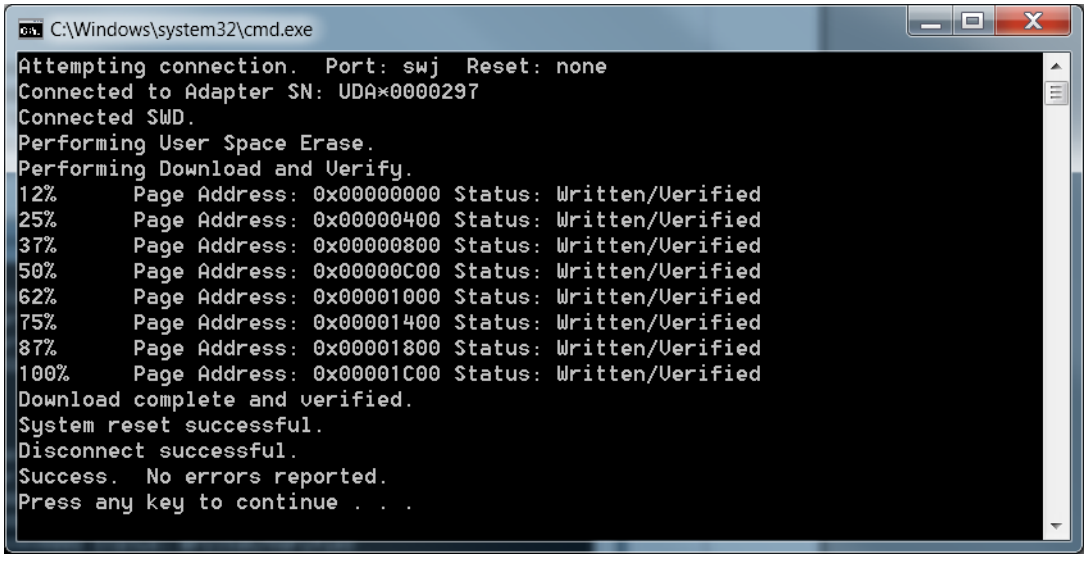


Figure 2. Flash Programming Utility

3.4. Loading the Application Code Using the Graphic Interface

There are two example applications provided to test out the bootloader functionality: Blinky_Fast and Blinky_Slow. The two firmware images blink the LEDs on the MCU card at different rates to allow the user to detect that the bootloader has successfully modified the firmware image.

1. Select the COM port associated with the device.
2. Perform a Query command to obtain information about the bootloader and the loaded application image. Figure 3 shows the result of the query command.

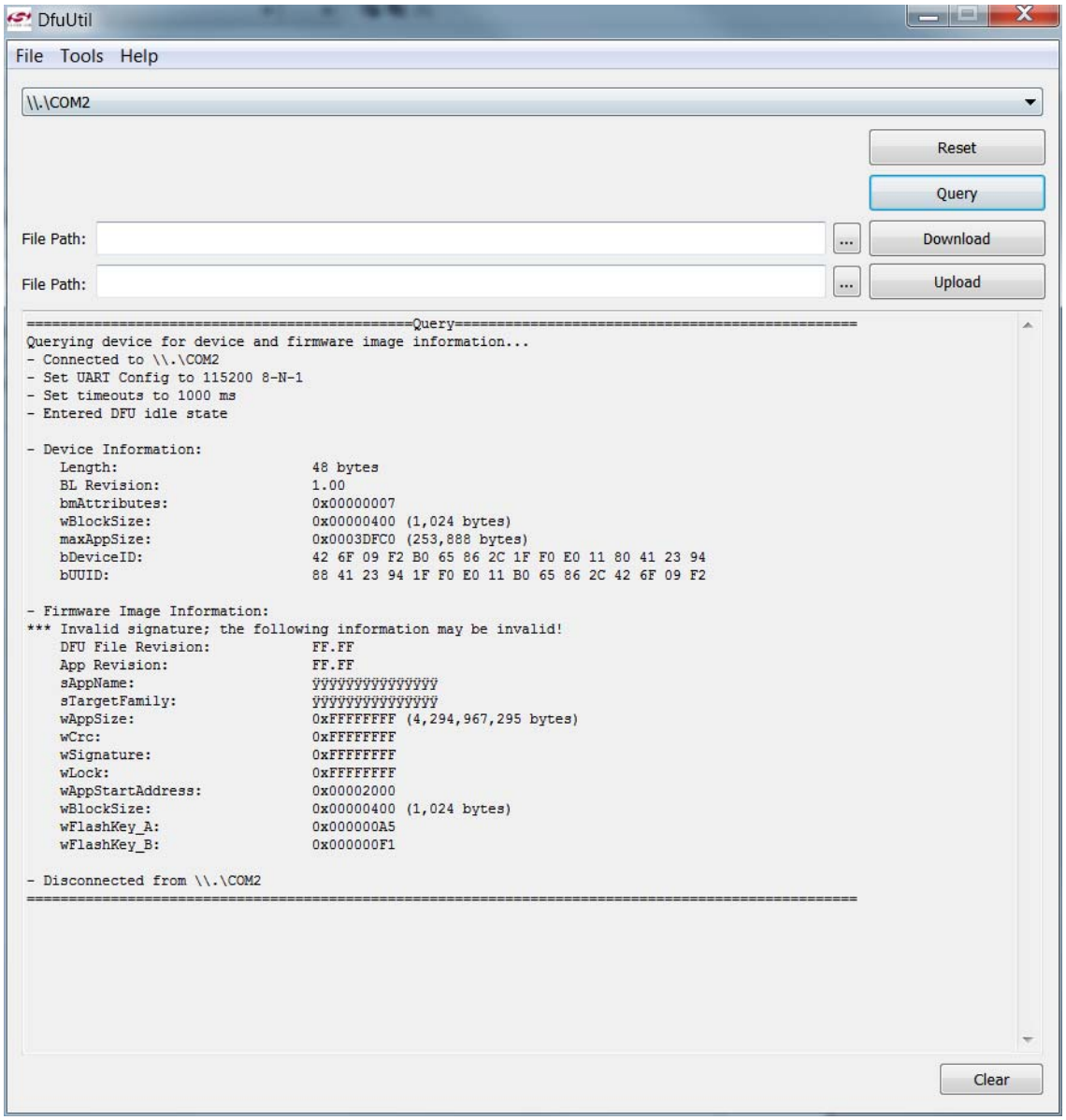


Figure 3. Query Command

3. If the device has an invalid application image, it will automatically be in bootload mode. If it does contain a valid application image, it needs to be forced into bootload mode by holding down SW2 (PB2.8) and pressing and releasing SW1 (reset switch). The target MCU should now be ready to accept a new firmware image.

4. Browse for the “\Firmware\Blinky_Fast\ARM\build\Blinky_Fast.dfu” firmware image file, and perform a download operation. Upon successful download, perform a reset operation to begin executing the new code. The LEDs should be blinking at a fast rate.

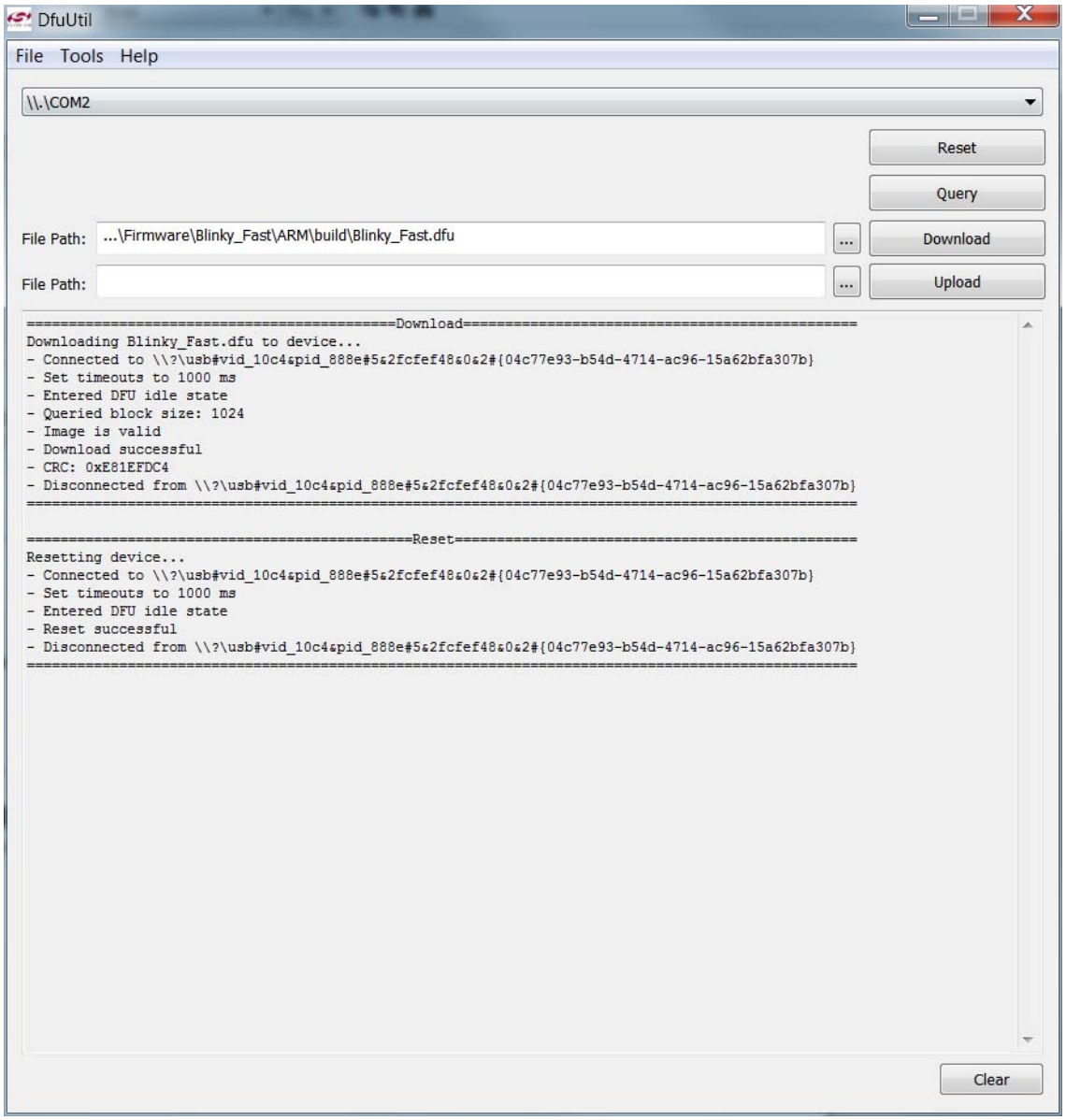


Figure 4. Download and Reset Command

5. Force the device into bootload mode by holding down SW2 (PB2.8) and pressing and releasing SW1 (reset switch). The target MCU should now be ready to accept a new firmware image.
6. Perform a Query command and verify that the sAppName of the loaded firmware image is “Blinky_Fast”.
7. Perform a Download of the “\Firmware\Blinky_Slow\ARM\build\Blinky_Slow.dfu” firmware image file.
8. Perform a Reset to execute the code.
9. The LEDs should now be blinking at a slow rate indicating that firmware has been successfully updated.
10. Force the into bootload mode by holding down SW2 (PB2.8) and pressing and releasing SW1 (reset switch) to perform a Query, Upload, or Download command on the device.

4. How to use the USB Bootloader (Demonstration)

The following steps demonstrate how to use the USB Bootloader to load two different applications, Blinky_Fast and Blinky_Slow, into the Target MCU.

4.1. Software Setup

Install the WinUSB DFU driver to allow the PC to recognize and communicate with the USB DFU device.

1. Double-click the batch file named *1.Install_WinUSB_Driver.bat* in the USB_Demo folder to initiate the driver installation. The Device Driver Installation Wizard shown in Figure 5 should appear on the screen.
2. Click Next to continue.

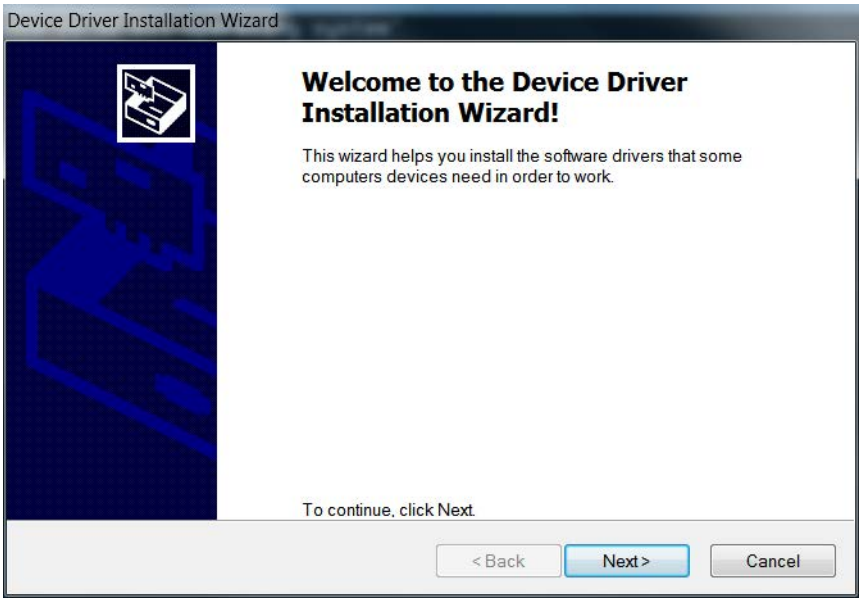


Figure 5. Device Driver Installation Wizard

3. The installer will prompt to confirm if you would like to install this device software. Click the check box and press Install to continue. The security prompt is shown in Figure 6.

Figure 6. Silicon Laboratories Certificate Installation

4. The installer will take a few minutes to install the necessary files. After the necessary files are copied, the dialog shown in Figure 6 is displayed on the screen. Press the Finish button to complete the installation.

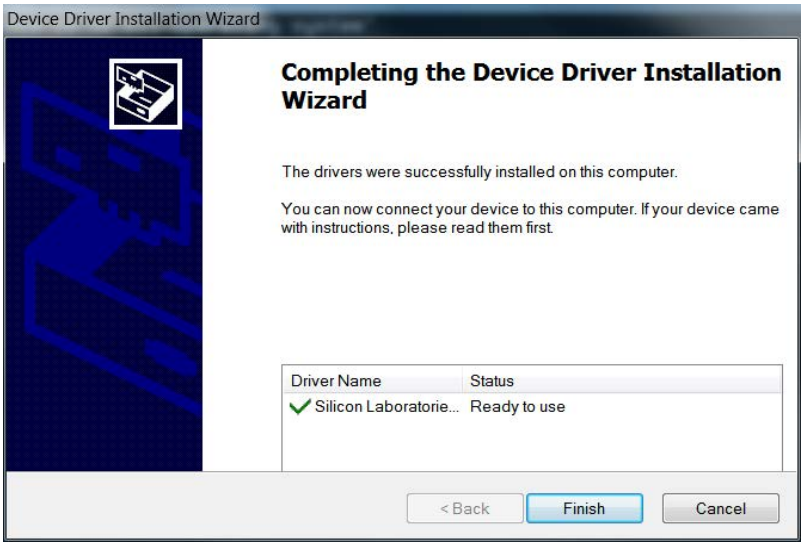


Figure 7. Driver Installation Complete

4.2. Hardware Setup

1. Connect the debug port of an SiM3U1xx MCU Card to a USB debug adapter using a ribbon cable and connect the USB debug adapter to the PC using a standard USB cable.
2. Power the SiM3U1xx MCU Card through its Device USB port (J13) using a mini USB cable.

4.3. Flashing the Bootloader

An uninitialized device requires the bootloader to be flashed over the debug interface. Once the device is programmed with a bootloader, the application image may be loaded over the USB interface. Click on the *Flash_Bootloader_Hex.bat* batch file to flash the MCU with the bootloader firmware using the command line flash programming utility. Figure 8 shows the output of the flash programming utility after the bootloader has been successfully programmed into the Target MCU.

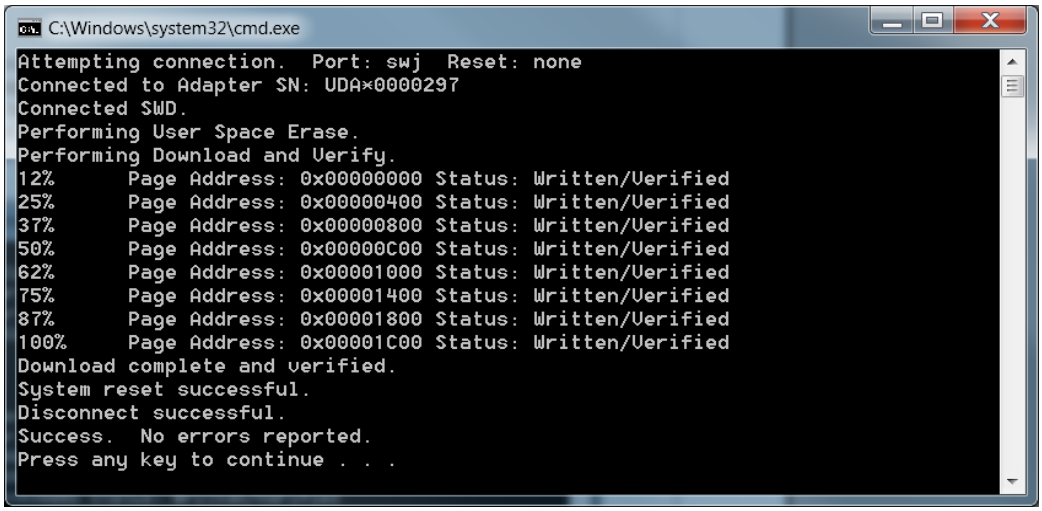


Figure 8. Flash Programming Utility

4.4. Loading the Application Code Using the Graphic Interface

There are two example applications provided to test out the bootloader functionality: Blinky_Fast and Blinky_Slow. The two firmware images blink the LEDs on the MCU card at different rates to allow the user to detect that the bootloader has successfully modified the firmware image.

1. Select the USB device with VID=10C4 and PID=888E.
2. Perform a Query command to obtain information about the bootloader and the loaded application image. Figure 9 shows the result of the query command.

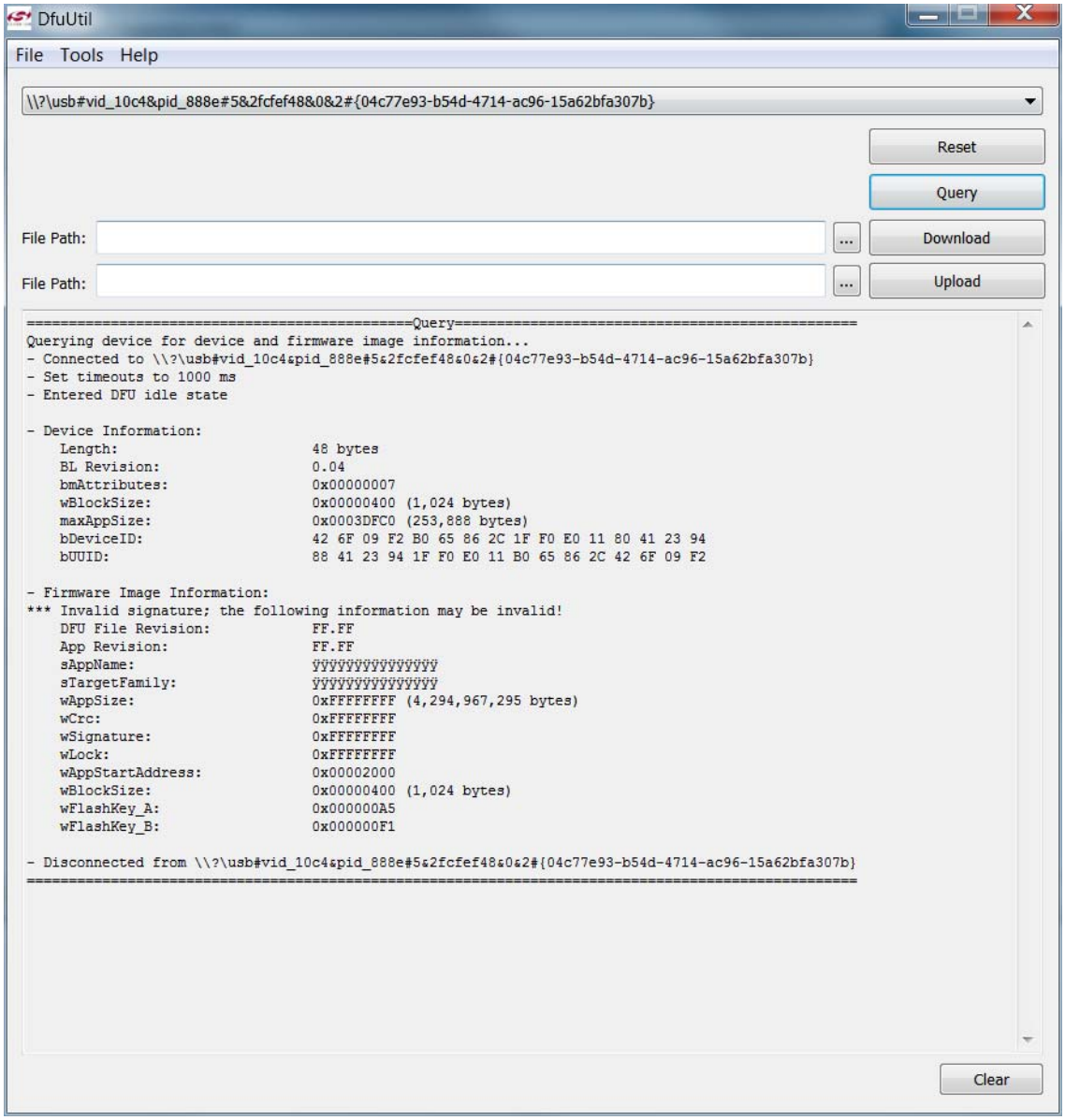


Figure 9. Query Command

3. If the device has an invalid application image, it will automatically be in bootload mode. If it does contain a valid application image, it needs to be forced into bootload mode by holding down SW2 (PB2.8) and pressing and releasing SW1 (reset switch). The target MCU should now be ready to accept a new firmware image.

4. Browse for the “\Firmware\Blinky_Fast\ARM\build\Blinky_Fast.dfu” firmware image file, and perform a download operation. Upon successful download, perform a reset operation to begin executing the new code. The LEDs should be blinking at a fast rate.

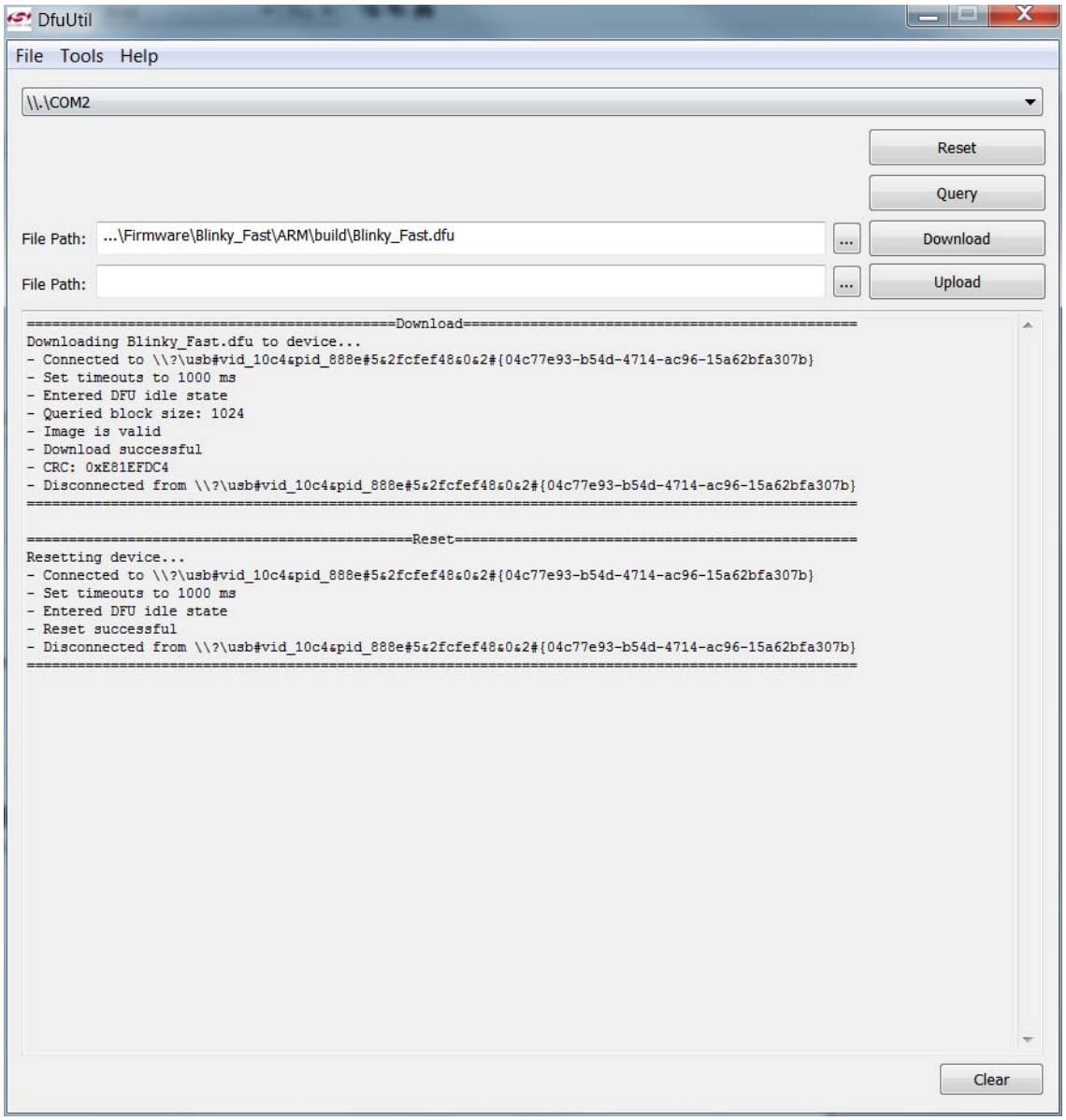


Figure 10. Download and Reset Command

5. Force the device into bootload mode by holding down SW2 (PB2.8) and pressing and releasing SW1 (reset switch). The target MCU should now be ready to accept a new firmware image.
6. Perform a Query command and verify that the sAppName of the loaded firmware image is “Blinky_Fast”.
7. Perform a Download of the “\Firmware\Blinky_Slow\ARM\build\Blinky_Slow.dfu” firmware image file.
8. Perform a Reset to execute the code.
9. The LEDs should now be blinking at a slow rate indicating that firmware has been successfully updated.
10. Force the device into bootload mode by holding down SW2 (PB2.8) and pressing and releasing SW1 (reset switch) to perform a Query, Upload, or Download command on the device.

5. Bootloader Target MCU Implementation

The bootloader for the Target MCU is based on the bootloader framework described in application note 762. The File Manager and DFU (device firmware update) State Machine are imported unmodified from the framework software, and a custom boot handler, communication interface, and flash control interface are added to complete the bootloader implementation. Compile time build options allow the bootloader to be configured for UART or USB communication. Figure 11 shows a block diagram of the target MCU bootloader.

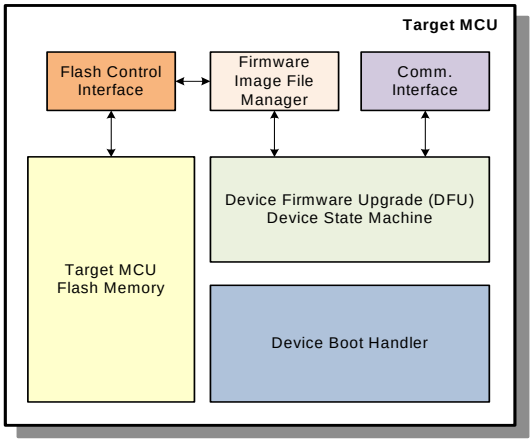


Figure 11. Target MCU Bootloader

5.1. Build Options

The framework source code contains support for multiple device families and communications protocols. In this particular implementation, the device family is specified in a project variable named <MCU_FAMILY>, and the communication protocol (UART or USB) is specified in a project variable named <COMM_PROTOCOL>. Hex files containing the bootloader firmware are distributed for both UART and USB for the SiM3U1xx family and are targeted to run on an 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 uVision4) 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 source repository in the device_sim3u1xx.c source file. The following sections provide a walk-through of the source code.

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 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 1 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 1. 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 gates 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 calculation of a 32-bit CRC.

5.2.4. DEVICE_UpdateCRC32

The DEVICE_UpdateCRC32 accepts an 8-bit value, incorporates it into the current CRC32 operation.

5.2.5. DEVICE_ReadCRC32Result

The DEVICE_ReadCRC32Result returns the 32-bit result of the current CRC32 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 to respond 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 bootload triggers that require a specific reset source. This function varies from the standard HAL implementation in that it is optimized for code space.

5.3. 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.

5.3.1. Packet Format

Figure 12 shows the packet format used in this 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.

\

UART Frame Format

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

Figure 12. UART Frame Format

5.3.2. Acknowledgment and Automatic Retransmission

The receiver of a packet is required to send back to the transmitter a single-byte acknowledgment packet 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 13 shows an example packet flow under various normal and error conditions.

5.3.3. Baud Rate

This UART implementation uses autobaud detection to configure the baud rate. The master programmer sends an 0x55 (or ASCII 'U') the first time it communicates with the Target MCU. Upon reception of this character, the Target 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 Target MCU's baud rate has been set, it remains set and cannot be changed until the next reset.

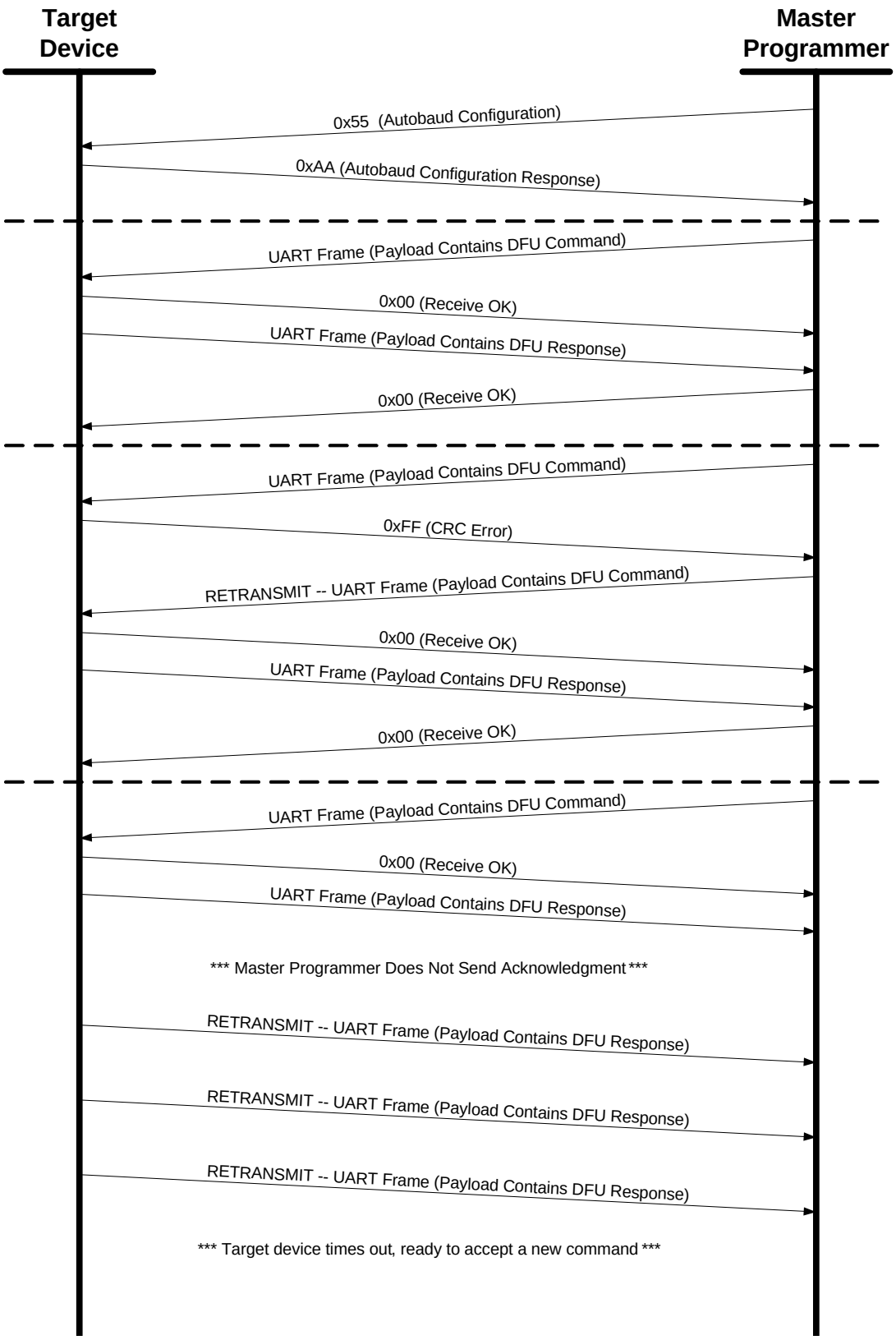


Figure 13. Example UART Packet Flow

5.4. USB Comm Interface

The USB Comm Interface in this implementation is based on the Device Firmware Upgrade (DFU) Device Class version 1.1 with many simplifications to make the Comm interface more generic. The USB Comm Interface provides a mechanism for transmitting and receiving data and is driven by the DFU state machine. The interface passes class (DFU) and vendor (Silicon Labs) control transfer requests to the DFU state machine using the COMM_Receive() function. Similarly, the DFU state machine sends OUT data packets to respond to these requests by calling the COMM_Transmit() function. The USB protocol provides all of the interface requirements specified in the bootloader framework including variable payload size, error checking, acknowledgment, and automatic retransmission.

5.4.1. Source Files

- comm_usb_sim3u1xx.c – Implements the Comm interface for the modular bootloader framework
- comm_usb_sim3u1xx.h – Defines device and Comm specific definitions
- USB0_Descriptor.c – Contains USB descriptor constants
- USB0_StandardRequests.c – Handles USB chapter 9 standard requests
- USB0_ControlRequests.c – Handles vendor and class control requests by passing requests on to the Comm interface
- USB0_ISR.c – Handles USB interrupts

5.4.2. Endpoints

The USB Comm Interface only uses a single endpoint, Endpoint 0, for control transfers. The max packet size for the endpoint is 64 bytes.

5.4.3. Descriptors

The USB implementation contains a USB device descriptor, configuration descriptor, interface descriptor, and two string descriptors (manufacturer and product). The default descriptors contain the following information:

- Vendor ID: 0x10c4 (Silicon Laboratories)
- Product ID: 0x888E (USB modular bootloader)
- Manufacturer String: Silicon Laboratories Inc.
- Product String: DFU Bootloader

These fields can be customized by modifying the constants located in USB0_Descriptor.c.

5.4.4. Control Transfers

All data transfers for the USB implementation make use of control transfers. Control transfers are initiated by the Master Programmer or USB host. Since the USB Comm Interface is driven by the DFU state machine, there must be a mechanism for the USB device to make the USB host wait until data buffers are available. The USB hardware in the target device will automatically NACK requests sent by the host until the firmware loads up the appropriate USB FIFOs and indicates that the USB packet is ready to send or receive.

A control transfer consists of three phases: a setup phase, data phase, and acknowledge phase. During the setup phase, the USB host sends a setup packet containing a request, which includes the request type, request, data phase size and direction. The data phase occurs after the setup phase, and the direction of this phase is dependent on the direction bit in the setup packet. Data is transmitted from the device to the host in an IN data phase and data is transmitted from the host to the device in an OUT data phase. After the data phase, the receiver of the data sends an acknowledgment in the ack phase. The USB implementation handles the ack phase automatically in hardware and is not a part of the USB Comm interface. See Figure 14 below for a diagram describing a USB control transfer.

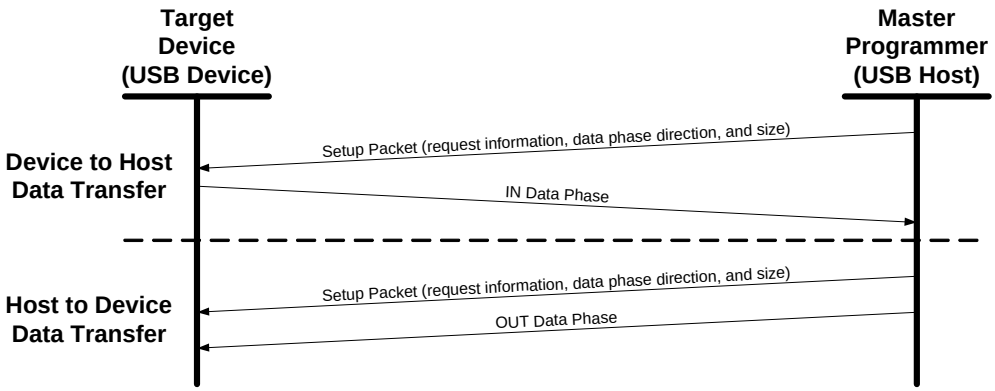


Figure 14. Control Transfer Data Flow

5.4.5. Timeouts

Since all data transfers are host-initiated, the host also controls timeouts for sending data to the device and receiving data from the device. The host will transmit data and automatically retry until the device acknowledges reception of the data or until the host times out. Similarly, the host will automatically attempt to receive data from the device and retry until the device sends data or until the host times out.

5.4.6. Thread Synchronization

Since the USB interface is driven by the host, the target device must handle USB events, such as bus reset, sent OUT packet, and received IN packet, in a USB interrupt service routine. However, since the DFU state machine provides the buffer used to store received data and the buffer containing data to transmit, some thread synchronization must occur between the main thread and USB interrupt thread. The USB Comm Interface must wait to process USB events and packets until after the DFU state machine passes in receive and transmit buffers.

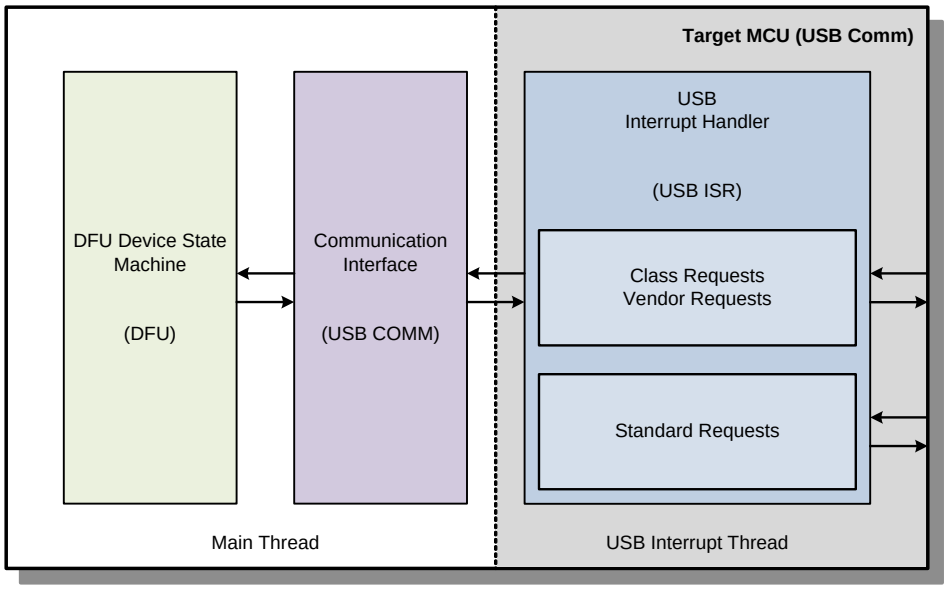


Figure 15. USB Thread Synchronization

The USB ISR handles standard requests such as GET_DESCRIPTOR and SET_ADDRESS completely in the interrupt handler. However, since the class and vendor requests are serviced by the DFU state machine, the USB ISR must defer processing of these requests until the state machine can handle them.

5.4.6.1. Synchronization of Data Phases

All thread synchronization between the USB ISR and main thread occur in the USB0_ControlRequests.c source file. The module provides an interface to pass in a receive or transmit buffer and wait until the host has received or transmitted data into or out of the buffer. The following sections provide a walk through of the source code.

5.4.6.2. USB0_RX_Start

This routine is called from the main thread to pass in a buffer used to store data received in the USB ISR, such as the setup packet and OUT data.

5.4.6.3. USB0_Is_RX_Complete

The main thread calls this routine to poll until the USB ISR returns data to the main thread. On successful completion, this routine returns the number of bytes received.

5.4.6.4. USB0_RX_Complete

The USB ISR calls this routine to indicate that the setup and OUT data phase have completed. In the case of a host-to-device transfer, the receive buffer contains both the setup packet and OUT data.

5.4.6.5. USB0_TX_Start

This routine is called from the main thread to pass in a buffer containing the data to send in the USB ISR, such as the IN data.

5.4.6.6. USB0_Is_TX_Complete

The main thread calls this routine to poll until the USB ISR has completed sending data to the host. On successful completion, this routine returns the number of bytes transmitted.

5.4.6.7. USB0_TX_Complete

The USB ISR calls this routine to indicate that the IN data phase has completed.

5.5. 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 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.

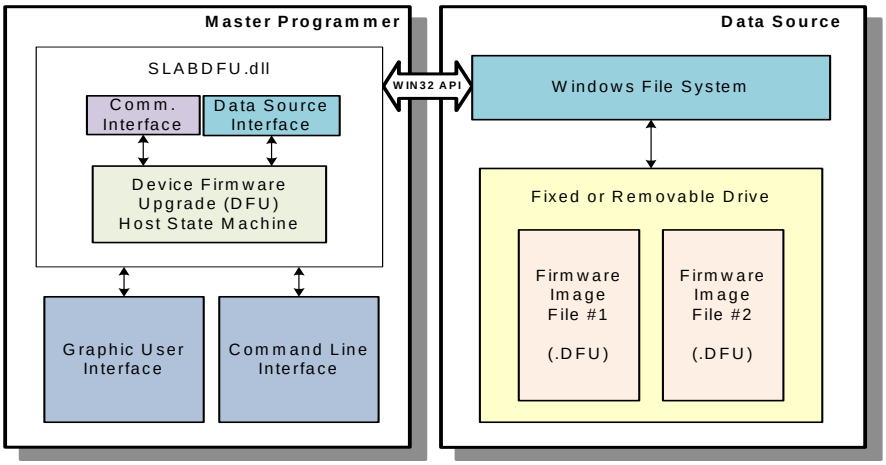


Figure 16. Master Programmer and Data Source

6.1. 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 SiM3xxxx MCUs via 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.

6.2. 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 bootload 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

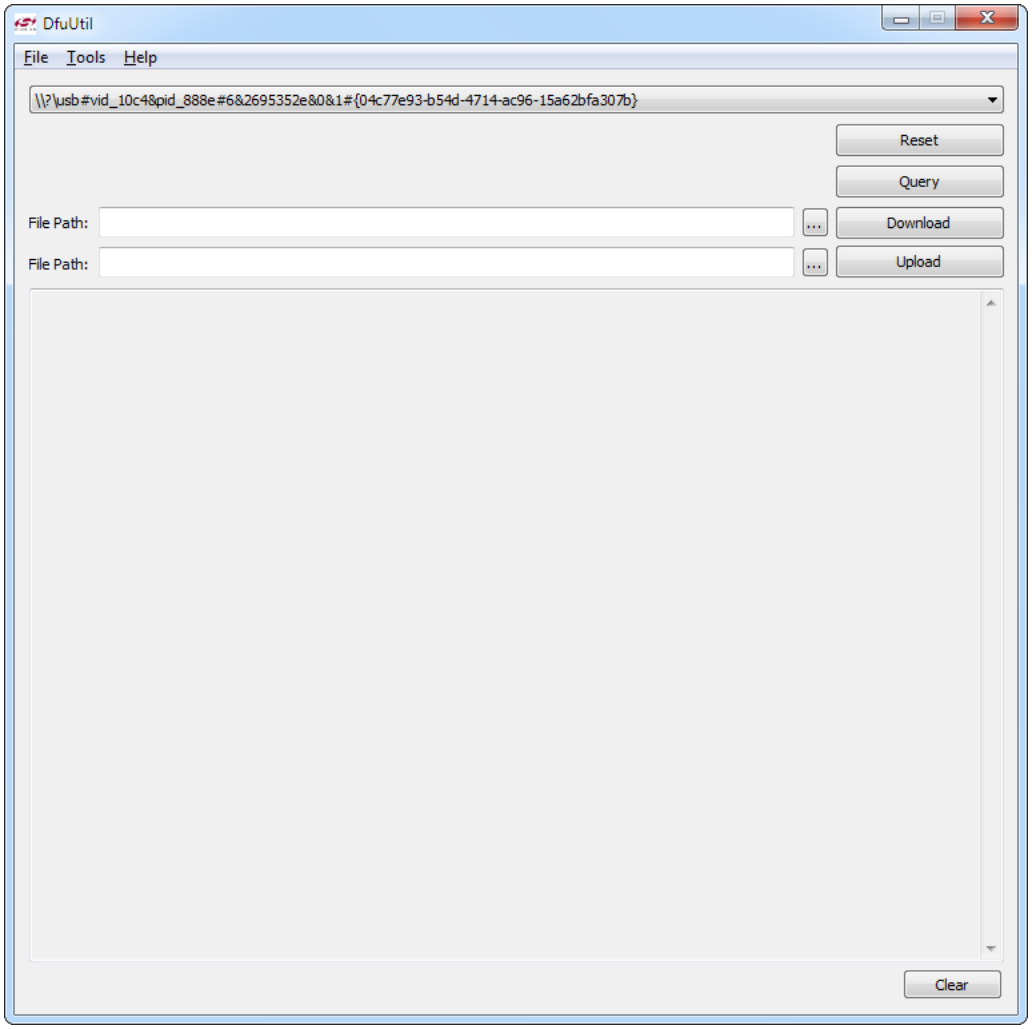


Figure 17. DfuUtil.exe

6.3. 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 18 below.

6.3.1. Examples

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

```
DfuUtilCL -listdevices
```

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

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

To download an image, blinky.dfu, to the second device in the device list, run:

```
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, run:

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

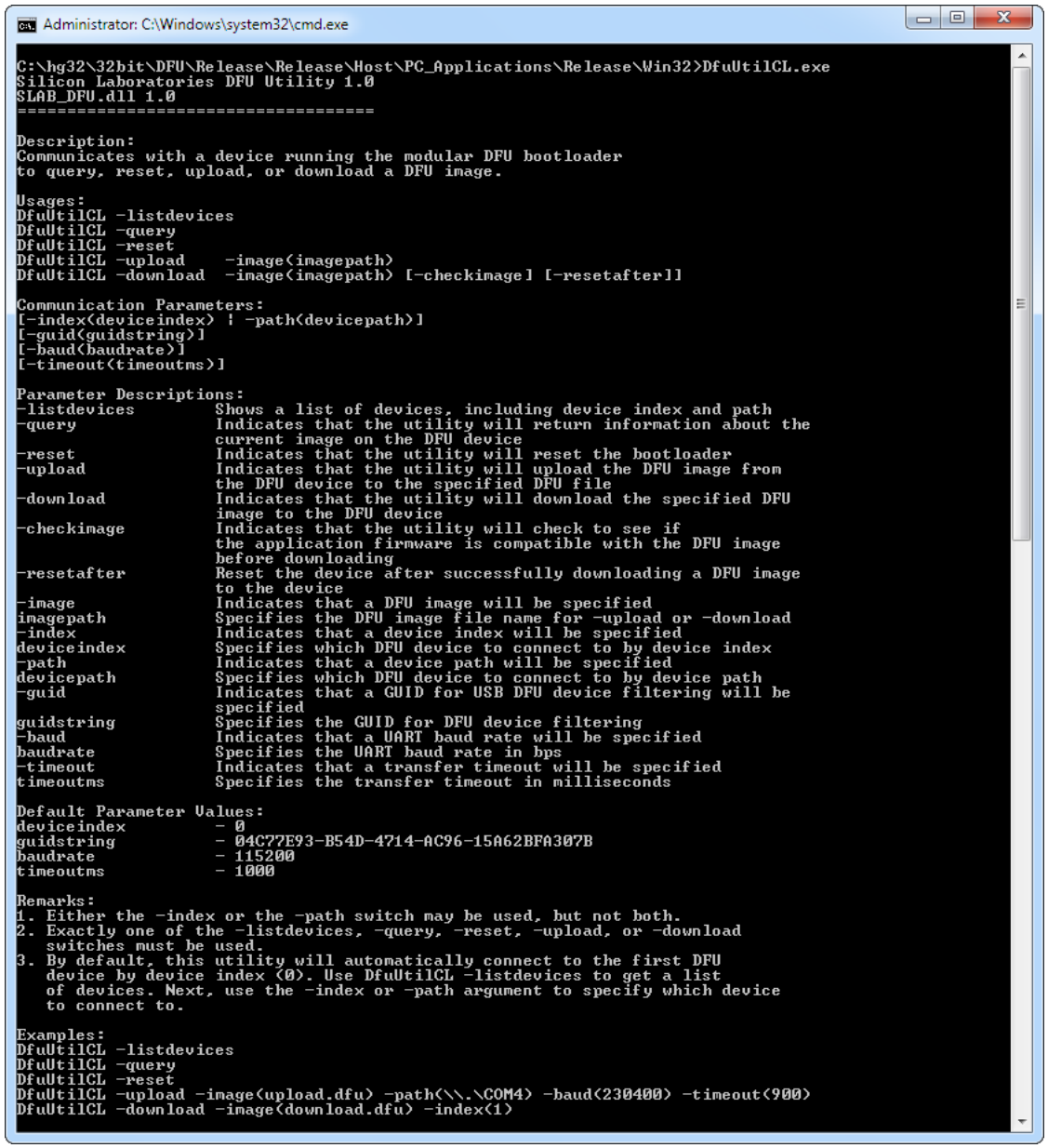


Figure 18. DfuUtilCL.exe

7. Creating Bootloader Aware Applications

In order to use the bootloader to load a firmware image onto the target MCU, the user must create a new project or modify an existing project and relocate the starting code memory address from 0x00000000 to the application start address (for example, 0x00002000).

7.1. Relocating the Application Starting Address

Two example firmware applications are provided, Blinky_Fast and Blinky_Slow, that have been modified from the SiM3U1xx blinky example in the Si32.

7.1.1. Keil µVision

To relocate the starting memory address for a Cortex M3 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).

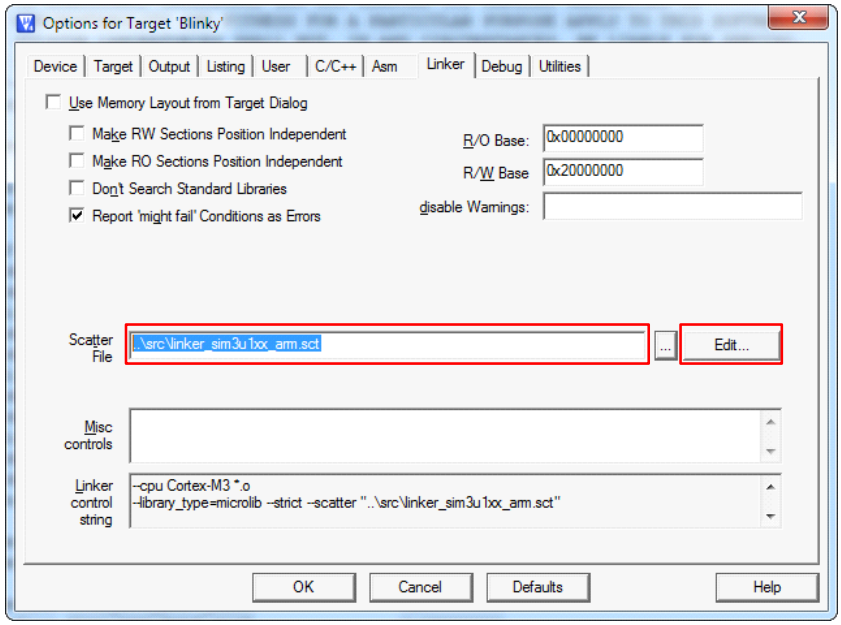
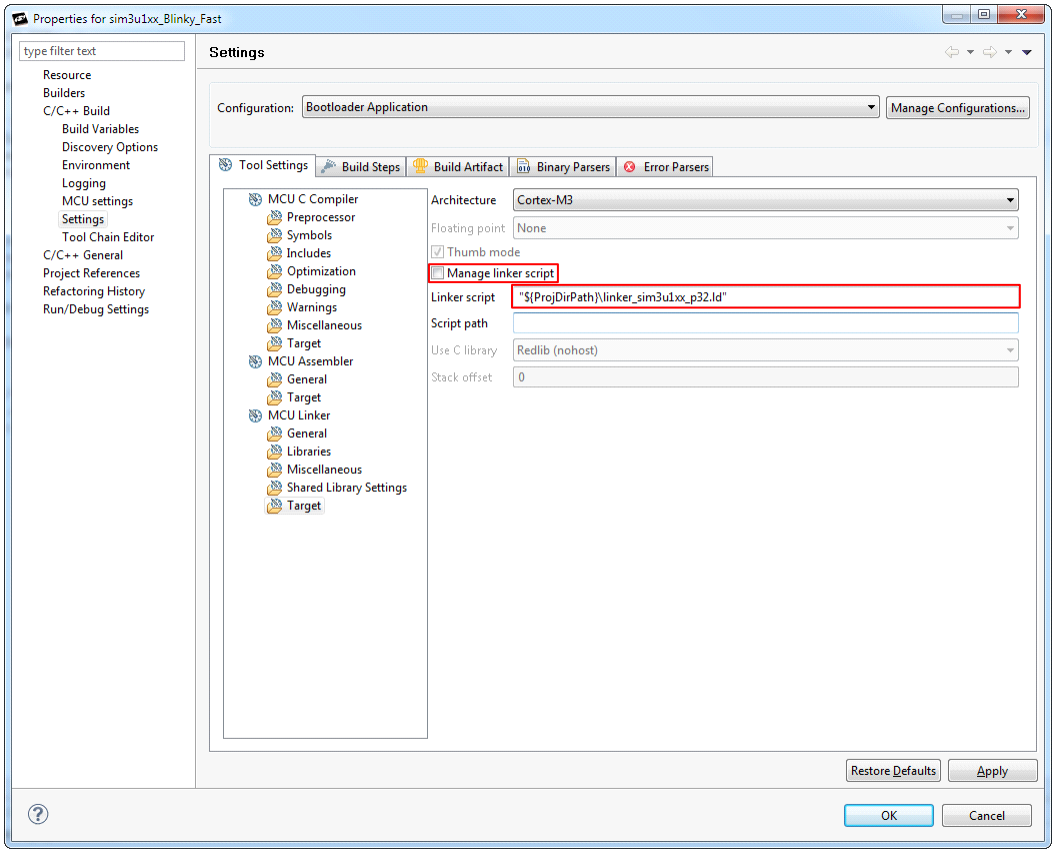


Figure 19. Keil Linker Options

7.1.2. Precision32 IDE

To relocate the starting memory address for a Cortex M3 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, uncheck “Manage linker script” and enter the path to the new copy of linker_sim3u1xx_p32.ld in the “Linker script” edit box. The example projects has a new “Bootload Application” build configuration that relocates the application start address and generates a hex file.



4. Modify the new copy of linker_sim3u1xx_p32.ld:
    a. 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).

7.2. Generating a Hex Image

Users must specify binary DFU files to download and upload. Users must first create a hex file and then convert the ASCII hex file to a binary DFU file using the HexToDfu.exe command line program.

7.2.1. Keil µVision

To generate a hex file when building a project in Keil, the user must perform the following steps:

1. Open the project file in µVision.
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.

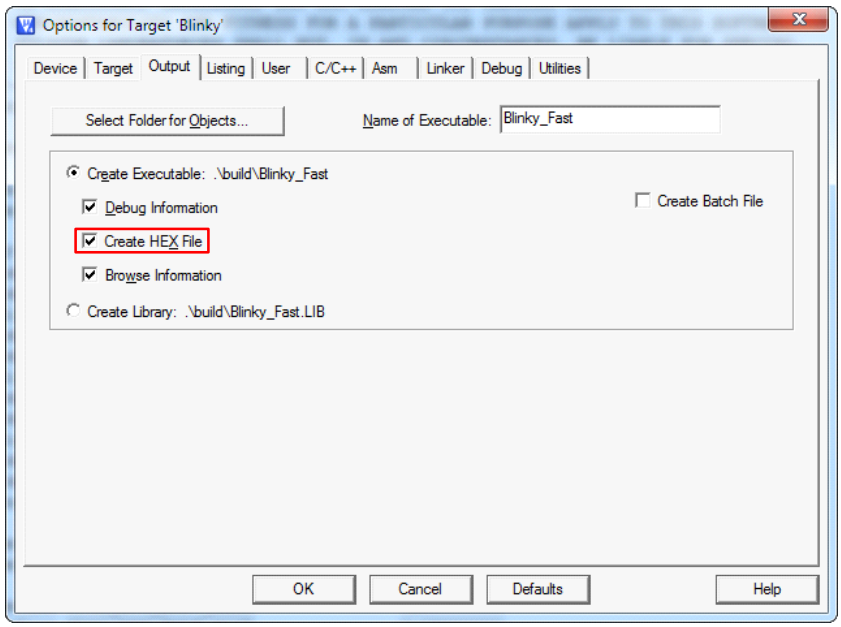


Figure 20. Keil Output Options

7.2.2. Precision32 IDE

Please refer to Knowledge Base #312015 at:

http://cp-siliconlabs.kb.net/article.aspx?article=312015&p=12979

7.3. 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 21 below.

7.3.1. Examples

To display a list of supported part names, run:

```
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, run:

```
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.

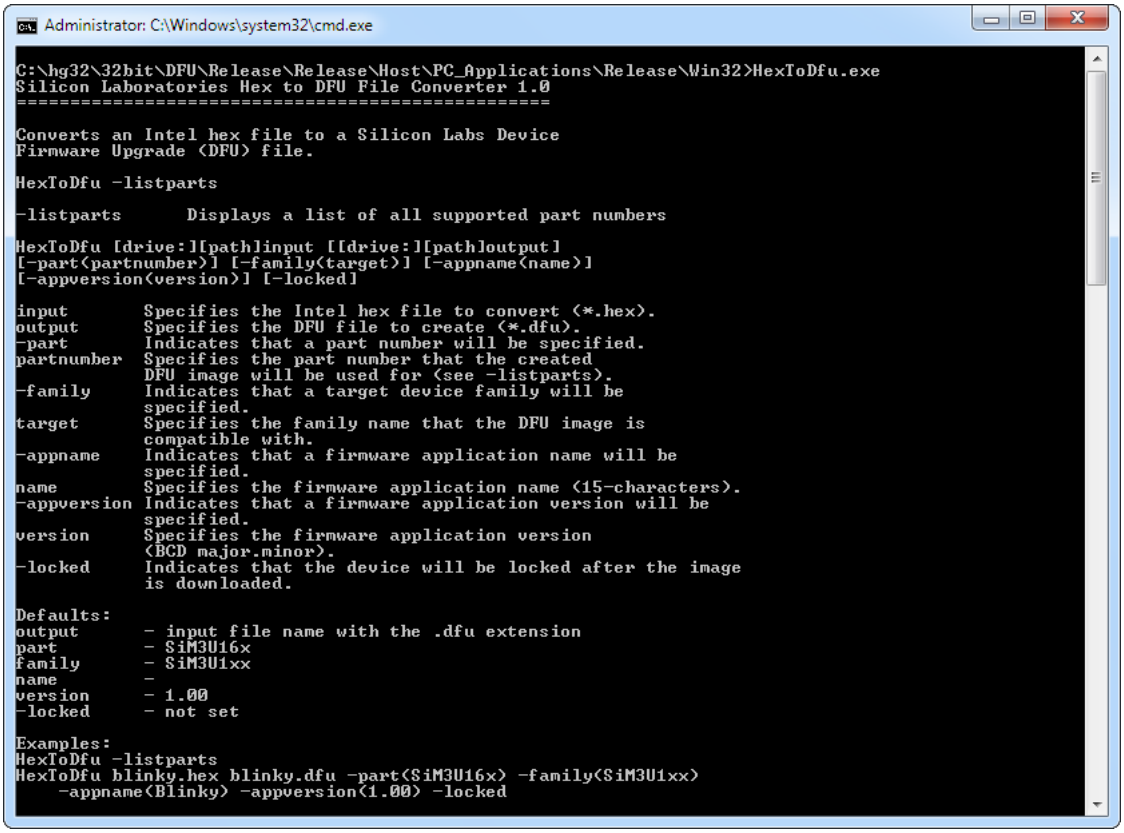
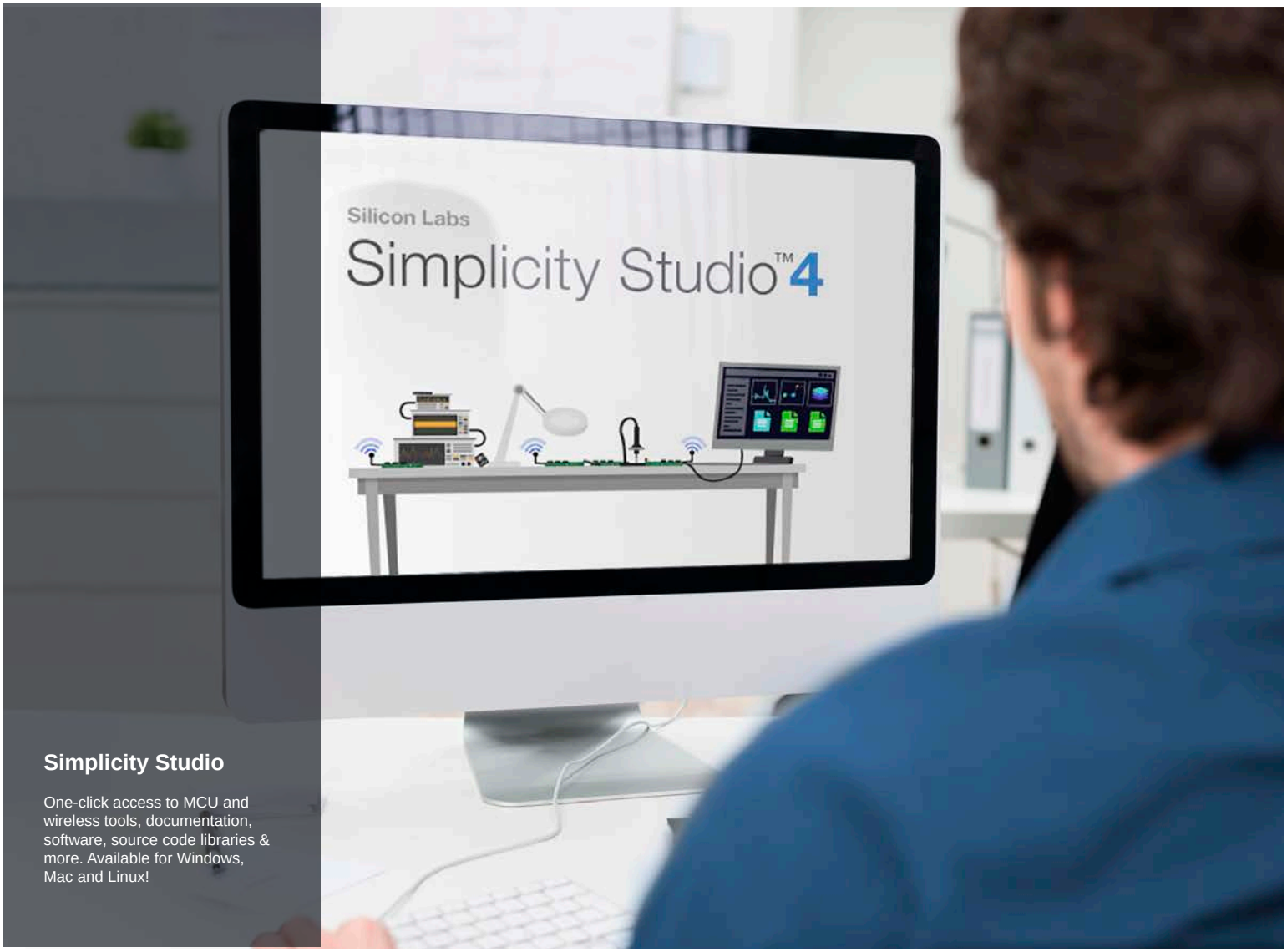


Figure 21. HexToDfu.exe



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

SW/HW
www.silabs.com/simplicity

Quality
www.silabs.com/quality

Support and Community
community.silabs.com

SILICON LABS

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

http://www.silabs.com