MODULAR BOOTLOADER FRAMEWORK FOR SILICON LABS C8051FXXX MICROCONTROLLERS

1. Introduction

A bootloader enables field updates of device firmware without the need for dedicated, external programming hardware. All Silicon Labs C8051Fxxx 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 field updates of the application firmware. 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, CAN, or USB. 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 re-use most of the code as-is across different Silicon Labs MCU families, and to be able to use it with various communication channels. Additional related application notes describe the interface-specific implementation details for various communication channels such as CAN and LIN. These documents are available at http://www.silabs.com/products/mcu/Pages/ApplicationNotes.aspx

2. Modular Bootloader Framework Overview

The modular bootloader framework consists of the following components:

- Target Bootloader Firmware
- Master Programmer Firmware or PC Software
- Active Data Source Software
- Hex or Binary Application Firmware Image

For communication channels that can be directly accessed by the PC (such as UART or USB), the Master Programmer firmware can be bypassed and the PC software can directly program the target MCU. In another scenario where the Master Programmer firmware can be interfaced with a storage medium such as an SD card that contains the application binary image, the PC software can be bypassed. The different possible firmware update setup configurations are shown in Figure 1, Figure 2, and Figure 3. The current release of the modular bootloader framework supports option 2 only.

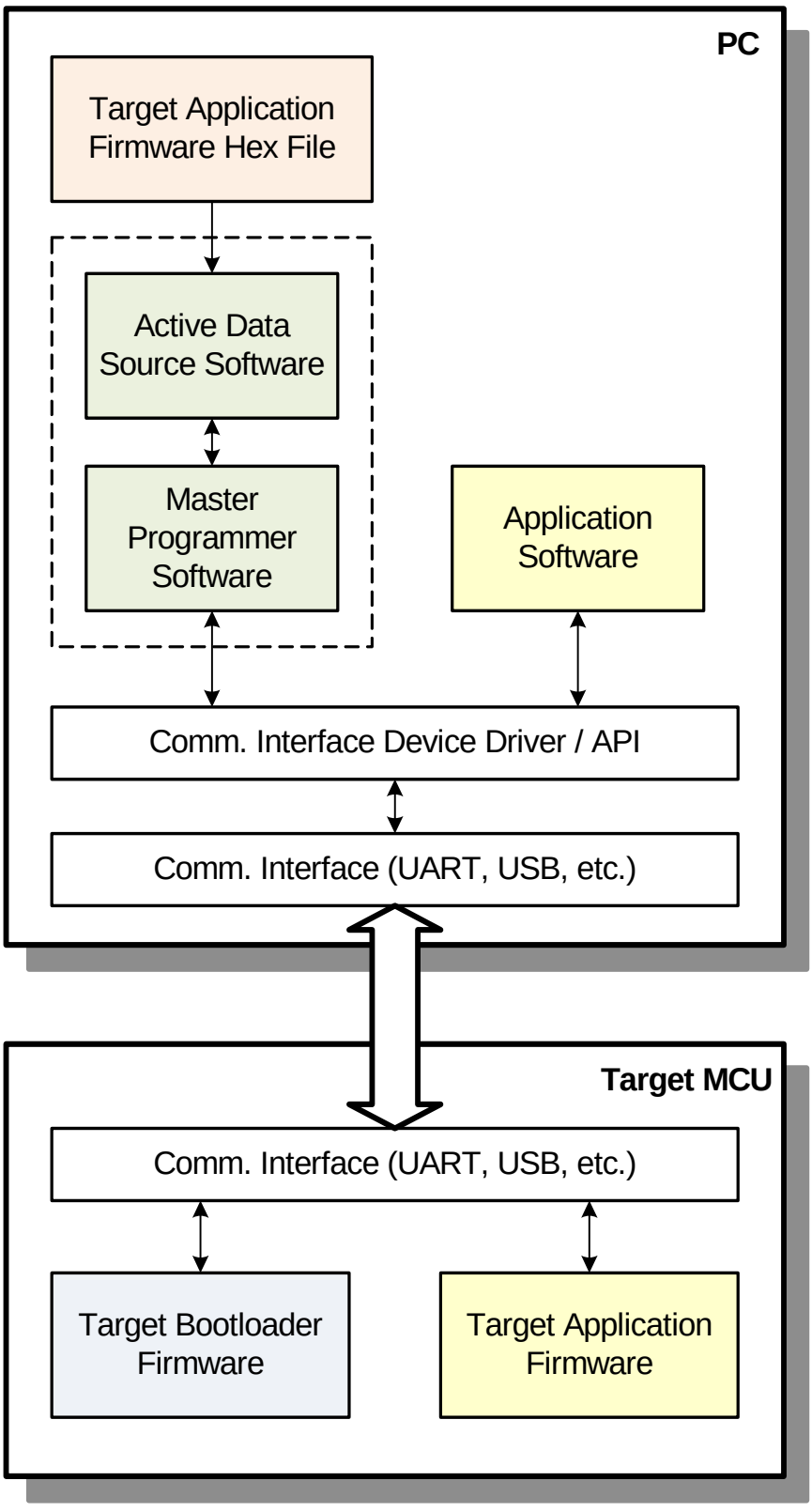


Figure 1. Firmware Update Setup—Option 1

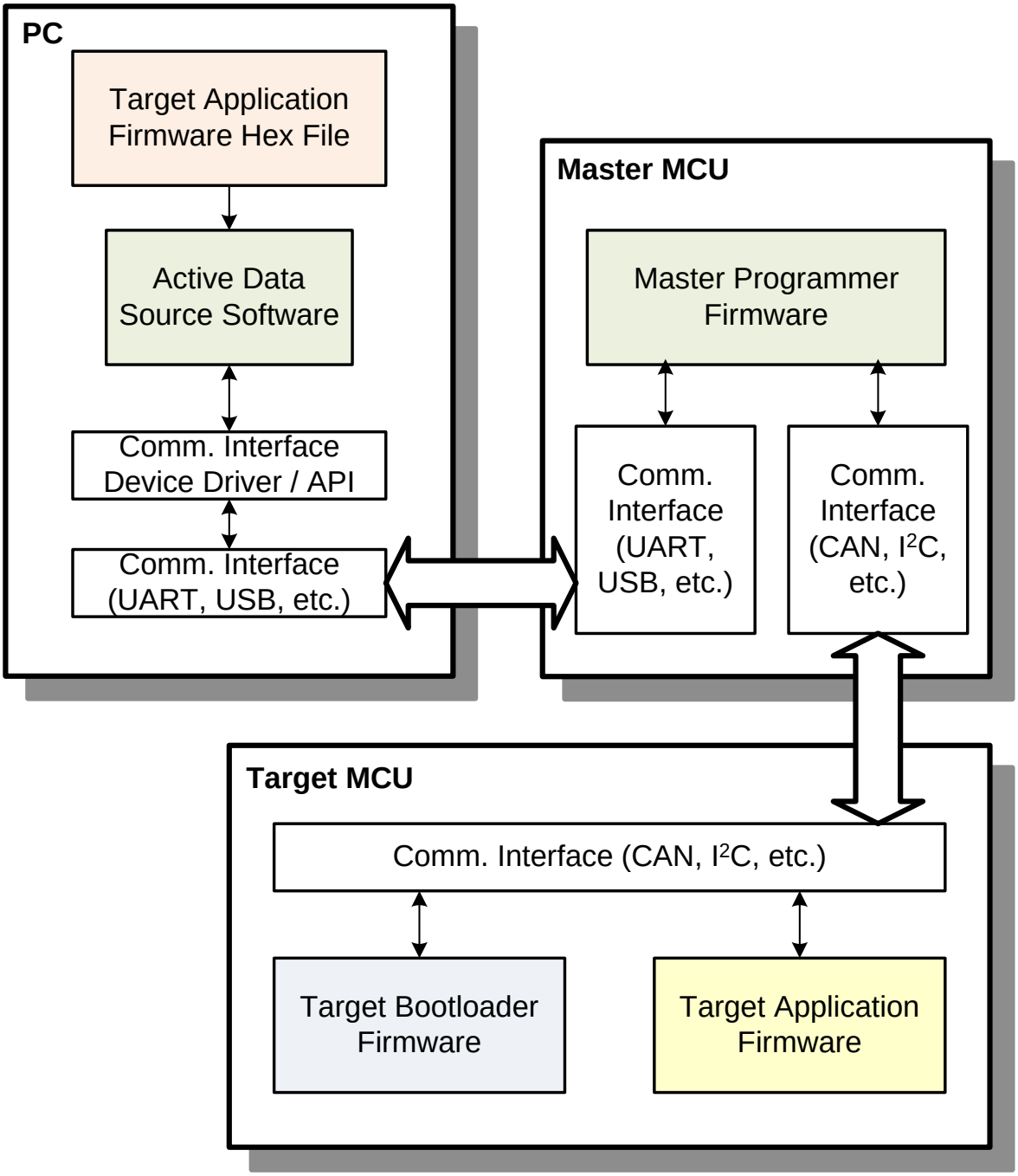


Figure 2. Firmware Update Setup—Option 2

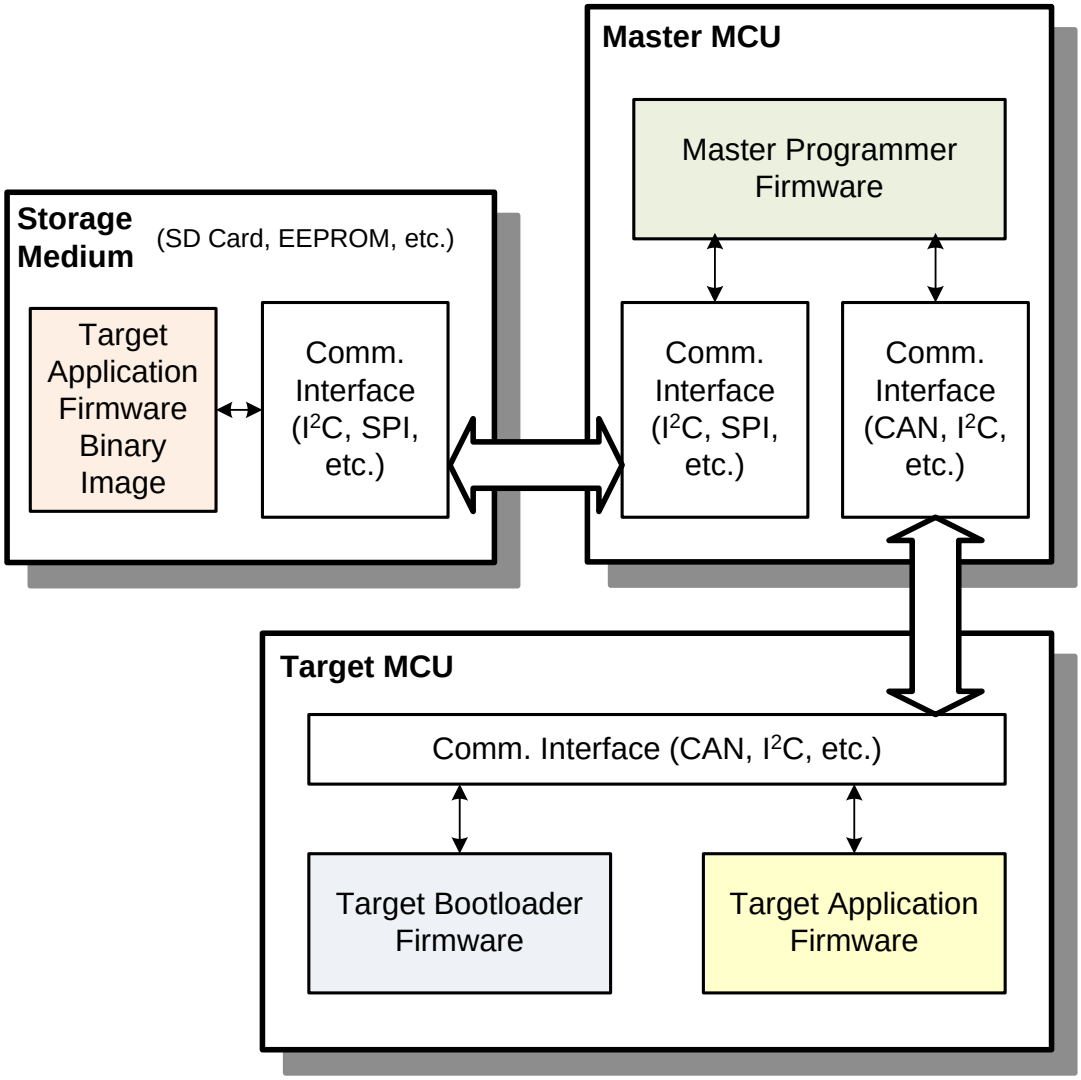


Figure 3. Firmware Update Setup—Option 3

3. Target Bootloader Firmware Design

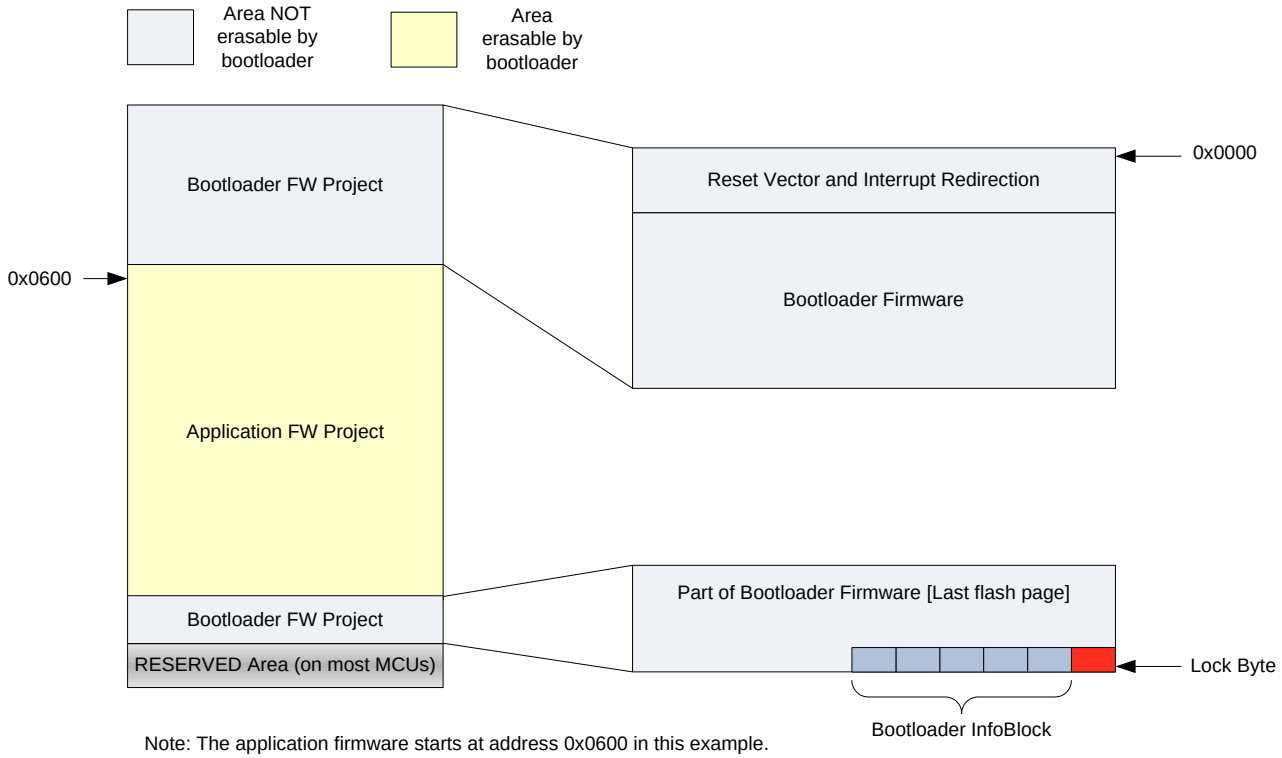
The target “bootloader” firmware “loads” (runs) on the target MCU on each “boot” (reset) and is responsible for reprogramming the MCU. The bootloader firmware communicates with the master programmer via the chosen communication channel to receive the new application firmware image that is to be programmed.

3.1. Target Bootloader Firmware Memory Usage

The target bootloader is split and placed in two sections of flash memory to efficiently utilize the code space available. These two sections are:

- Lower flash memory page(s) starting at address 0x000000.
- Last page of flash memory (the one that includes the lock byte).

The bootloader must occupy the area starting from 0x000000 because a device reset should always execute the bootloader first. The bootloader can then determine if it should launch the application or stay in bootload mode. Instead of placing the entire bootloader contiguously starting at 0x000000, one page of the bootloader firmware is placed on the last page (the page with the lock byte) because that page is not erasable by firmware on Silicon Labs MCUs, and thus cannot be used for the application firmware. The size of the bootloader depends on the complexity of the communication protocol used. Figure 4 shows the bootloader firmware code memory map. See Figure 6 in Section 6 for an application firmware memory map.



Note: The application firmware starts at address 0x0600 in this example.

Figure 4. Target Bootloader Firmware Memory Map

A few bytes at the end of the bootloader firmware space are used to store key information about the bootloader. The application firmware also stores some key information about the application at the end of its allocated space. The purpose and content of these "InfoBlocks" are described in Section 3.2.9 and Section 3.2.10. Because the bootloader and application firmware run exclusive of each other, there are no restrictions on RAM usage for either of the firmware projects.

3.2. Target Bootloader Firmware Features

The bootloader firmware implements a set 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. These fault-tolerance enhancements and other features are described in the following sections.

3.2.1. Self-Update Prevention

The bootloader firmware contains code that includes boundary checks to prevent erasure of the bootloader firmware itself. The *Erase_Page* and *Write_Page* commands work only for addresses that are within the application firmware code space. This boundary check ensures that the bootloader is never erased in case of an accidental command from the master programmer.

3.2.2. Interrupt Redirection

The hardware interrupt vectors from page 0 are redirected by the bootloader into the application space, which allows the application firmware to use those interrupts as desired and enables the updating of interrupt vectors when updating the application image. All interrupts except the Reset interrupt are redirected using unconditional LJMP instructions. The Reset interrupt causes the MCU to enter the bootloader and perform the signature check before jumping to the reset vector of the application firmware. Because all hardware interrupts are redirected to the application space using the static code on page 0 (which is never erased), errors during an application firmware download that could render the device non-bootloadable are prevented. Figure 5 shows the interrupt redirection control flow.

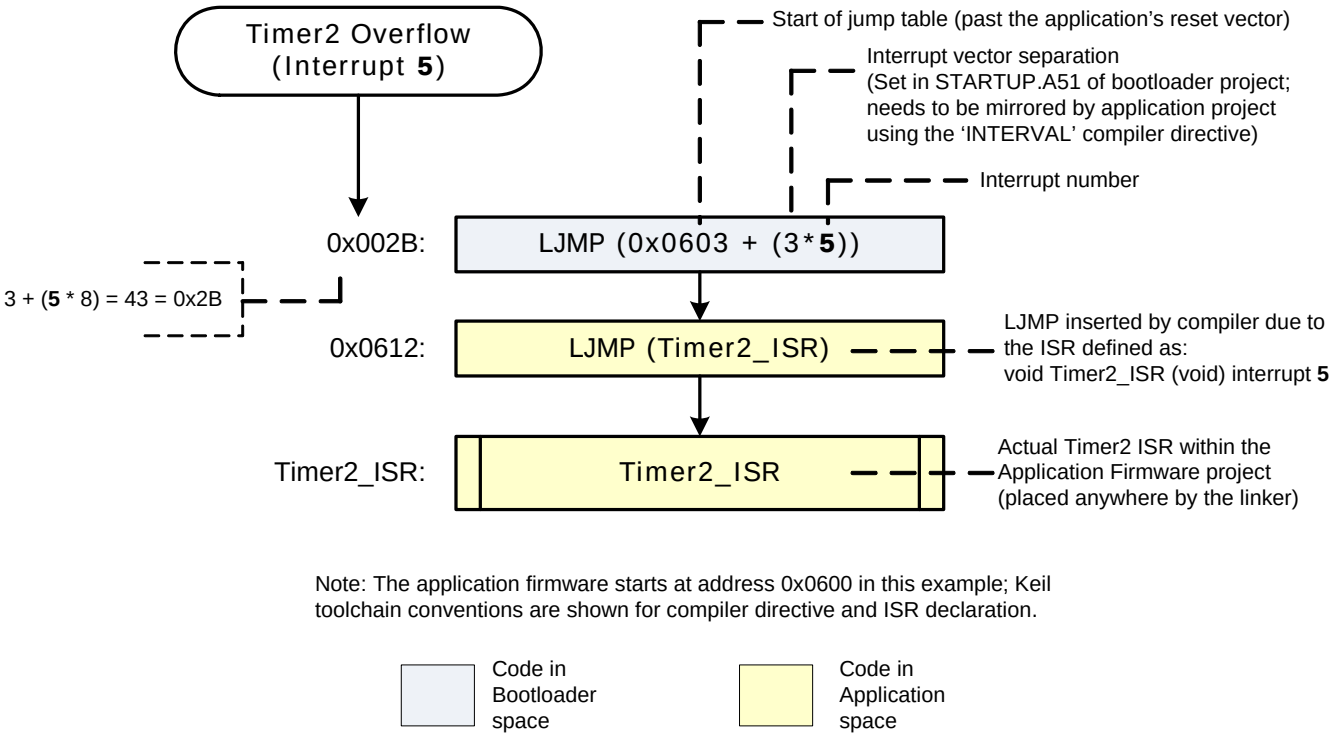


Figure 5. Interrupt Redirection Control Flow for Target Application Firmware

3.2.3. Post-Reset Signature Check

One of the most important fault-tolerant features included in this bootloader is the post-reset signature check. On a device reset, the bootloader code that is first executed checks whether a specific 16-bit signature value exists in the last two bytes of the application code space. If this check passes, the application is started by the bootloader. If it fails, the bootloader considers the application invalid and waits for a new application firmware image. To ensure that the device is never rendered non-bootloadable, the application firmware update process is designed such that the page containing the signature is the first to be erased and the signature value is the last data to be written.

3.2.4. Non-Resident Flash Keys

One of the causes of Flash memory corruption is random code execution, which can happen due to an out-of-spec power ramp-up, among other factors. To reduce the possibility of Flash corruption due to random code execution, the Flash unlock key codes are not stored in the non-volatile code memory within the bootloader firmware. In the absence of these key codes, the MCU will not allow any Flash page to be erased or changed, and any attempt to do so will cause a Flash Error Reset. During a firmware update process, the Flash key codes are sent to the target bootloader firmware along with the `TGT_Erase_Flash_Page` and `TGT_Write_Flash_Bytes` commands to be held in RAM until that specific command completes execution. When that command is completed, the target bootloader clears the key codes from RAM, which minimizes the amount of time it has valid key codes and reduces the possibility of flash corruption. More detailed information about the causes of flash corruption and ways to prevent it can be found in this MCU Knowledge Base article: http://cp-siliconlabs.kb.net/article.aspx?article=87183&p=4120

3.2.5. 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 is performed during the firmware update process. The 16-bit CCITT CRC polynomial is implemented on the bootloader software and the firmware. If the 16-bit page CRC calculated on the software image does not match the corresponding CRC reported by the device, the firmware update process is aborted, and the user is notified (if in interactive mode).

3.2.6. Fail-Safe Firmware Update Option

Once a valid application firmware image is present (indicated by a valid signature), the bootloader launches the application firmware on device reset. In this case, the normal procedure to re-enter bootload mode is to make the application firmware call the bootloader. Because the application firmware is an unknown, it may become stuck in a state where it is not able to call the bootloader (due to a firmware bug), which could render the device non-bootloadable. To prevent this situation, a fail-safe option is included in the bootloader firmware that checks a port pin state on device reset. If this GPIO pin is asserted at that time, the bootloader will not proceed with the signature check and will stay in bootload mode instead. This option is enabled by default and can be modified to use a different pin or disabled as desired by the user.

3.2.7. Code Banking Support

The master programmer and the bootloader firmware use linear 32-bit addresses to support code-banked devices that have more than 64 kB of flash memory. For target MCUs with code banking, the API function `FLASH_ByteRead` should be implemented in the target bootloader firmware to provide read access to the code banked flash memory. This function need not be implemented for MCUs that have 64 kB or lesser code space because the target bootloader firmware can read code memory by using a code pointer (translated to MOVC by the compiler).

3.2.8. Polled-Mode Comm Implementation

The communication interface can be typically implemented as polled-mode or by using an interrupt service routine (ISR). The target bootloader implements the communication interface as polled-mode to avoid any additional latency for that application-mode interrupt. If implemented using an ISR, that interrupt cannot be unconditionally redirected to the application space; additional delay would be incurred to determine the MCU mode (bootload mode or application mode) before redirecting the interrupt to the right ISR. It also means a location in RAM needs to be reserved for use by the bootloader to store the MCU mode even while the application is running. Whereas, in a polled-mode implementation, the entire RAM space is usable by the application firmware, thus making application design much simpler.

3.2.9. Target Bootloader Firmware Info Block

The target bootloader firmware includes a series of bytes that are stored at the end of the bootloader space adjacent to the lock byte on the last flash page. This 'InfoBlock' contains key details about the target bootloader that can be read by the master programmer using the GetInfo command. Table 1 shows the InfoBlock structure along with the byte offset locations with respect to the last byte of bootloader space.

Table 1. Target Bootloader Firmware InfoBlock

Byte Offset (from lock byte location)	Field	Notes
–1	InfoBlock Length	This byte is included in the count.
–2	Bootloader FW Version Low (0)	Example: Low = 2 and High = 1 will yield v1.2
–3	Bootloader FW Version High (1)	
–4	Product Code	
–5	BL Type	See Table 2.
–6	Flash Page Size Code	If n, actual value = 2^n.
–7	BL Packet/Buffer Size Code	Upper nibble[7:4] = Packet Size; If n, actual value = 2^n Lower nibble[3:0] = Buffer Size; If n, actual value = 2^n; Min: 3 (= 8 bytes)
–8	CRC Type	See Table 3.
–9	Application FW Start Addr Low (0)	24-bit linear start address of the application firmware.
–10	Application FW Start Addr (1)	
–11	Application FW Start Addr High (2)	
–12	Application FW End Addr Low (0)	24-bit linear end address of the application firmware.
–13	Application FW End Addr (1)	
–14	Application FW End Addr High (2)	
–15	Device Serial Number Byte0 (0)	Lower two bytes of device serial number (for multi-drop buses)
–16	Device Serial Number Byte1 (1)	
–17	BL-specific Byte (0)	[OPTIONAL] Additional serial number bytes or other BL-specific bytes.
–18	…	

The `Flash Page Size Code` and `BL Buffer Size Code` are encoded using power-of-two encoding (code = n; value = 2^n). The BL Type and CRC Type are lookup table interpretations described in Table 2 and Table 3. Bit 7 (MSB) of CRC Type denotes if the CRC is 16-bit (= 0) or 32-bit (= 1). Some ranges of values of these types are reserved to be defined by Silicon Labs, while other values can be defined by users.

Table 2. Bootloader Type Lookup Table

BL Type	Filename	Description
0x00		Reserved
0x01	BL001	User-defined
…		User-defined
0x7F	BL127	User-defined
0x80	BL128	UART
0x81	BL129	CAN
0x82	BL130	LIN
0x83	BL131	I2C
0x84	BL132	SPI
0x85	BL133	Reserved
…	…	Reserved
0xFE	BL254	Reserved
0xFF	BL255	Reserved

Table 3. CRC Type Lookup Table

CRC Type	Filename	Description
0x00		Reserved
0x01	CRC001	16-bit user-defined CRC
…		16-bit user-defined CRC
0x3F	CRC063	16-bit user-defined CRC
0x40	CRC064	CCITT-16
…		Reserved
0x7F	BL127	Reserved
0x80	CRC128	32-bit user-defined CRC
…		32-bit user-defined CRC
0xBF	CRC191	32-bit user-defined CRC
0xC0	CRC192	Reserved
…		Reserved
0xFF	CRC255	Reserved

3.2.10. Target Application Firmware InfoBlock

The target application firmware also includes an InfoBlock that is stored at the end of the application space. This InfoBlock contains key details about the target application that can be read by the master programmer using the GetInfo command. Some of the information here is duplicated from the bootloader InfoBlock. The reason for this redundancy is to allow the master programmer to communicate appropriately with the target MCU by reading the application InfoBlock that would be contained within the application hex file. Table 4 shows the InfoBlock structure along with the byte offset locations with respect to the last byte of application space.

Table 4. Application Firmware Info Block

Byte Offset (from Last Byte of App FW)	Field	Notes
0	Signature Byte Low (0)	The signature bytes are used by the bootloader to determine if the application image is valid. These two bytes are not part of the Application FW InfoBlock and are not included in the InfoBlock Length count.
–1	Signature Byte High (1)	
–2	InfoBlock Length	This byte is included in the count.
–3	Product Code	Example: v1.2 ≥ Low = 2 and High = 1.
–4	BL Type	
–5	Flash Page Size Code	If n, actual value = 2^n.
–6	BL Packet/Buffer Size Code	Upper nibble[7:4] = Packet Size; If n, actual value = 2^n Lower nibble[3:0] = Buffer Size; If n, actual value = 2^n; Min: 3 (= 8 bytes)
–7	Application FW Version Low (0)	Example: v1.2 ≥ Low = 2 and High = 1.
–8	Application FW Version High (1)	
–9	Device Serial Number Byte0 (0)	Lower two bytes of device serial number (for multi-drop buses)
–10	Device Serial Number Byte1 (1)	
–11	BL-specific Byte (0)	[OPTIONAL] Additional serial number bytes or other BL-specific bytes.
–12	…	

3.3. Target Bootloader Firmware Commands

The interface between the master programmer and the target bootloader is implemented as a command-response protocol. The set of commands supported by the target bootloader firmware are:

- TGT_Enter_BL_Mode
- TGT_Get_Info
- TGT_Erase_Flash_Page
- TGT_Write_Signature
- TGT_Reset_MCU
- TGT_Get_Page_CRC
- TGT_Write_Flash_Bytes

Appendix A includes the Master to Target interface specification with details about each of the above commands. All commands are fixed length of 8 bytes except for TGT_Write_Flash_Bytes, which has a subsequent data payload that can be as big as the BL buffer size.

3.4. Target Bootloader Communication Interface

The target bootloader is designed to be modular, with the communication functions in a separate source file. The bootloader core calls these functions by using a pre-defined interface. This way, the core code does not have to be modified for different communication interfaces. The following three functions need to be implemented for any given communication interface:

1. void Comm_Init (void): Initializes the comm interface.
2. U8 Comm_Wait_For_Rx_Bytes (U8 numbytes_to_read): Waits to receive the specified number of bytes from the master and stores them in the global receive buffer (Rx_Buf). Also returns the first received byte as the return value (this is the command code). If the specified number of bytes is greater than the maximum payload in a packet (if applicable), then the subsequent packets are retrieved after sending back TGT_RSP_OK responses to the master after each packet has been received. No response is sent after receiving the last packet.
3. void Comm_Send_Tx_Bytes (U16 numbytes_to_send): Sends the specified number of bytes from the global transmit buffer (Tx_Buf) to the master; this blocking function waits for all the bytes to be sent before returning to the caller. If the specified number of bytes is greater than the maximum payload in a packet (if applicable), then the subsequent packets are sent after receiving TGT_RSP_OK responses from the master after each packet has been sent. No response is expected from the master after the last packet is sent.

4. Data Source Design

The data source provides the application firmware image to the master programmer. This data source can either be "passive" or "active". A passive data source stores a hex or binary image of the application firmware image that needs to be read and interpreted by the master programmer. An example would be a hex file stored in an SD card. This approach involves either more code on the master programmer to interpret a hex file or more programming time as a binary image will specify all bytes of the application firmware unlike a hex file.

An active data source reads in a hex file and provides a page-by-page data stream to the master programmer upon request. This reduces master programmer complexity and improves programming efficiency. In this release of the module bootloader framework, the master programmer only supports an active data source, with passive data source support planned for a future release.

4.1. Active Date Source Software Features

An active data source reads in a hex file and provides a page-by-page data stream to the master programmer upon request. This reduces master programmer complexity because the master does not have to include code to interpret a hex file. It improves programming efficiency (reduces programming time) because only those pages that are specified in the hex file are programmed into the target device instead of programming the entire target application space. It also allows the target application to use one or more pages as non-volatile storage (calibration constants, etc.) that will not be erased on a firmware update.

Active data sources support the following commands:

- SRC_Get_ Info
- SRC_Get_NextPage_Info
- SRC_Get_NextPage
- SRC_Disp_TGT_Info
- SRC_Disp_Error

Appendix B includes the Master-DataSource interface specification with details about each supported DataSource command.

4.2. Silicon Labs MCU Serial Bootloader Data Source Software

The Silicon Labs MCU Serial Bootloader Data Source software included with this modular bootloader framework is an example of an active data source software. The source code in Visual C# is included in the code package. It implements the following features:

- Reads and interprets Intel hex files
- Supports code banking by accepting up to four hex files (one for each bank)
- Keeps track of the flash pages that are specified in the hex file when decoding the hex file; uses this information to send only those pages to the master programmer when requested
- Uses RS232 (COM port) to communicate between the PC and the master programmer with the following settings: 115200 baud, 8-N-1, and no flow control
- Waits for and responds to master commands; see Appendix B for details
- Bootloading cannot be initiated from the data source software; it has to be initiated by the master programmer
- Decodes and displays the Target Application InfoBlock contained in the application hex file
- Receives and displays target MCU information (Bootloader InfoBlock + Application InfoBlock) via the master
- Displays bootload progress and allows viewing of raw data sent to the master

5. Master Programmer Design

The master programmer can be implemented on an MCU as firmware or as PC software. In the current release of the Modular Bootloader Framework, only Master Programmer Firmware is included, with a Master Programmer PC Software planned for a future release. Implementing the master programmer as MCU firmware is more flexible compared to PC software because it can be used for so many more target communication interfaces (such as UART, CAN, and I^2C) than the PC (which would be limited to RS232 and USB). But, for applications that already have a communication interface between the target MCU and the PC, the PC software approach is more efficient as it eliminates an intermediate MCU.

5.1. Master Programmer Interface

The master programmer needs to communicate with target MCUs and also with a data source that contains the target application firmware image that needs to be programmed into the target MCU.

Appendix A includes the Master-Target interface specification with details about each supported Target command. All commands are fixed length of 8 bytes except for TGT_Write_Flash_Bytes,which has a subsequent data payload that can be as big as the BL buffer size.

Appendix B includes the Master to DataSource interface specification with details about each supported DataSource command.

5.2. Master Programmer Firmware

The master programmer firmware example included with the modular bootloader framework is partitioned into the following source modules:

- Core code that includes the main program loop; portable across Silicon Labs MCU families
- Communication interface (UART) code that sends Data Source commands and receives responses
- Communication interface (CAN, I^2C, etc.) code that sends Target commands and receives responses
- Validation code that validates responses from the data source and the target

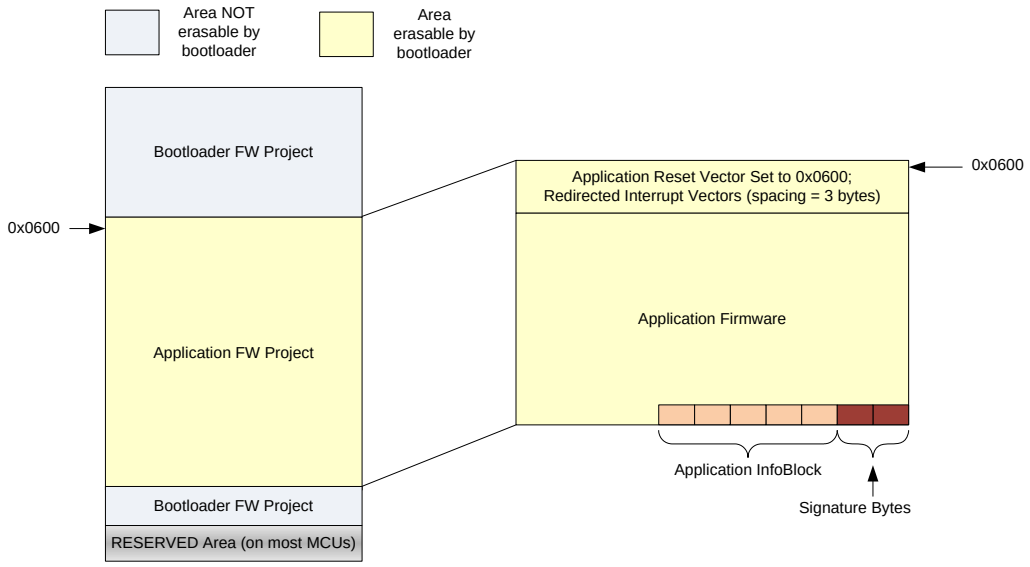
The master programmer firmware follows the steps listed in Section 6 to update the target application firmware.

6. 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. Change the reset vector of the application project from address 0x0000 to the starting address of the application (0x0600 in the template example). For the Keil compiler, this can be done by adding a customized version of the default STARTUP.A51 file with the modified reset vector specified using a CSEG directive (CSEG AT 0 CSEG AT 0600H).
2. Add a compiler command line option to let the compiler know that the interrupt vectors need to start at the new reset vector address. This is because all hardware interrupt vectors from page 0 are redirected by the bootloader project to the vector table on the first page of the application project. Also, add an option to make the vector spacing 3 bytes (instead of the 8-byte standard) for better code efficiency and for compatibility with the bootloader's interrupt redirection. For the Keil compiler, these can be done using INTVECTOR and INTERVAL directives. An example is shown in the template project: `INTVECTOR(0x600) INTERVAL(3)`
3. Add a linker command line option to locate the application firmware within the application firmware code area and to place the Application InfoBlock at the end of the application space, with two bytes reserved for the bootloader to store the signature bytes. An example is shown in the template project where the application space is between addresses 0x0600 and 0xF9FF: `CODE(0x600-0xF9FD, ?CO?FXXX_TARGETAPP_INFOBLOCK(0xF9F5))`
4. [Optional] Add code to recognize the `TGT_Enter_BL_Mode` command and to create a flash error reset upon a match to enter bootload mode. In the CAN bootloader's application template example, support functions are included to make this easier.
5. [Optional] Check hardware design and designate a pin that can be used by the bootloader as the fail-safe firmware update option trigger. This pin can be easily shared between the bootloader and application as it is only read once by the bootloader following a reset. For example, a push-button switch in the product can be used to put the MCU in bootload mode if held down while the MCU is reset using a power cycle or a reset button.

Once the above changes are applied, the application firmware should resemble the memory map shown in Figure 6.



Note: The application firmware starts at address 0x0600 in this example.

Figure 6. Target Application Firmware Memory Map

7. Firmware Update Steps

The target MCU needs to be programmed with the bootloader via the JTAG/C2 interface before a firmware update can take place via the chosen communication interface. During development, JTAG/C2 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. Erase entire code space and program just the bootloader firmware image on the target MCU using the JTAG or C2 programming interface.
2. Erase entire code space and program a combined bootloader + application firmware image on the target MCU using the JTAG or C2 programming interface. A combined image can be created by manually appending the application hex image to the bootloader hex image; also, the signature bytes need to be manually added to the hex image.

The first method is simpler, but a subsequent bootload step is needed before the product can be used in the end application. As long as a bootloader is present on the target, it can be invoked to update the application firmware. The firmware update process is described in detail in the flow diagram in Figure 7. Because of the order of the steps taken with respect to the signature bytes (erased first, written last), there are no adverse consequences if the firmware update process is interrupted at any stage for reasons such as target power loss or accidental comm cable unplug. If that happens, the target MCU can be reset (or power cycled) and the update process can be attempted again.

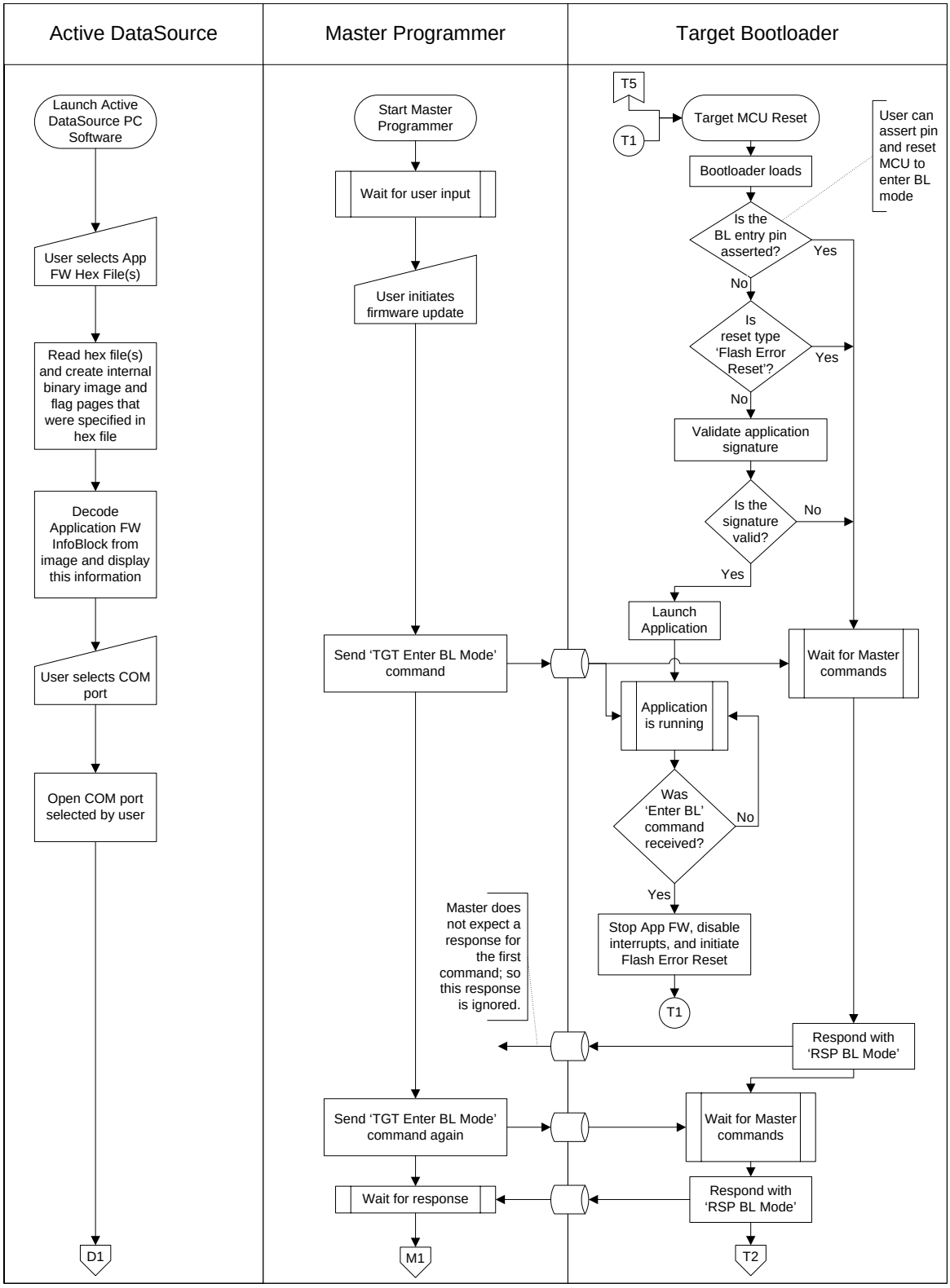


Figure 7. Firmware Update Process Flow Diagram (Page 1 of 4)

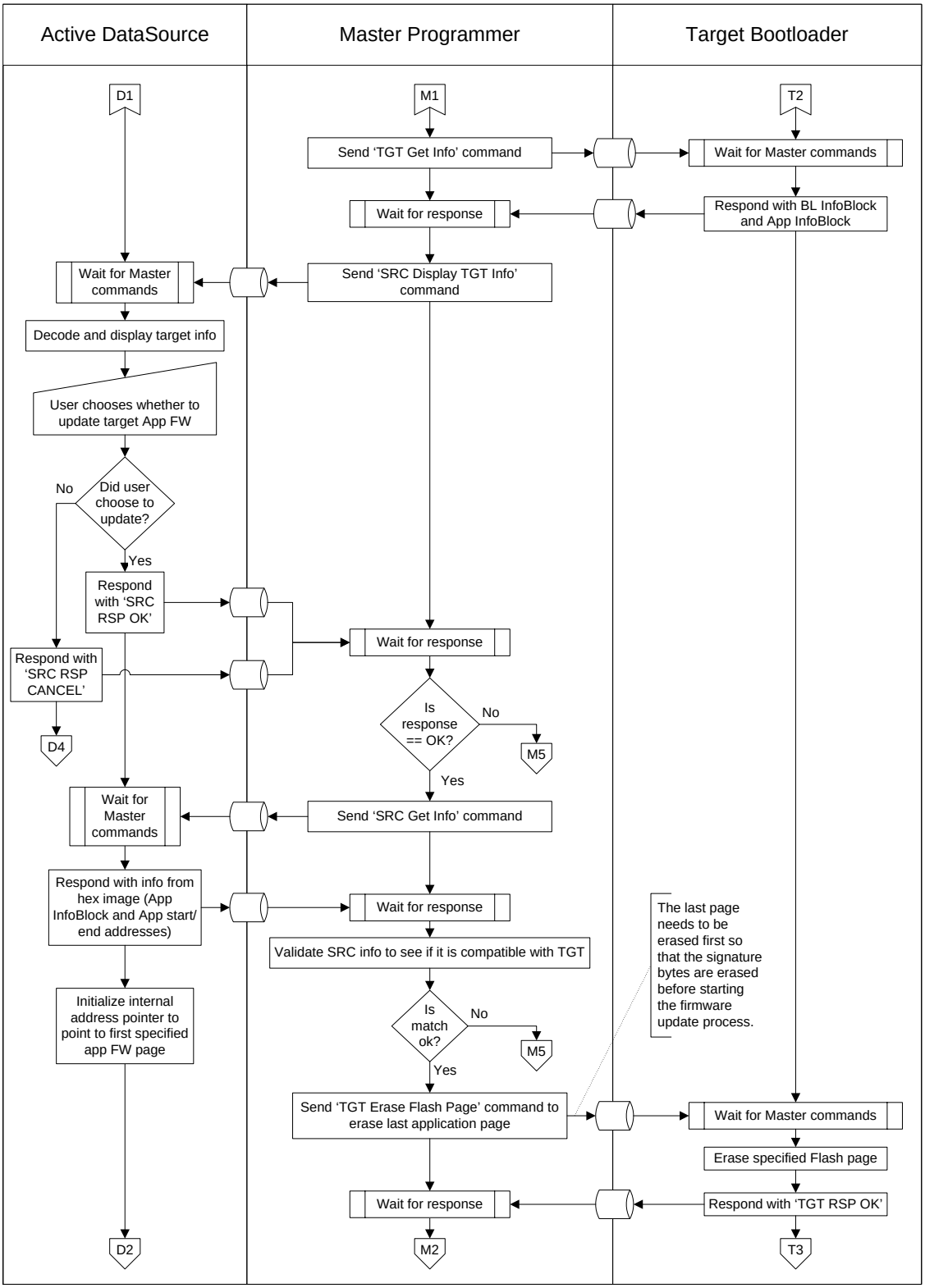


Figure 8. Firmware Update Process Flow Diagram (Page 2 of 4)

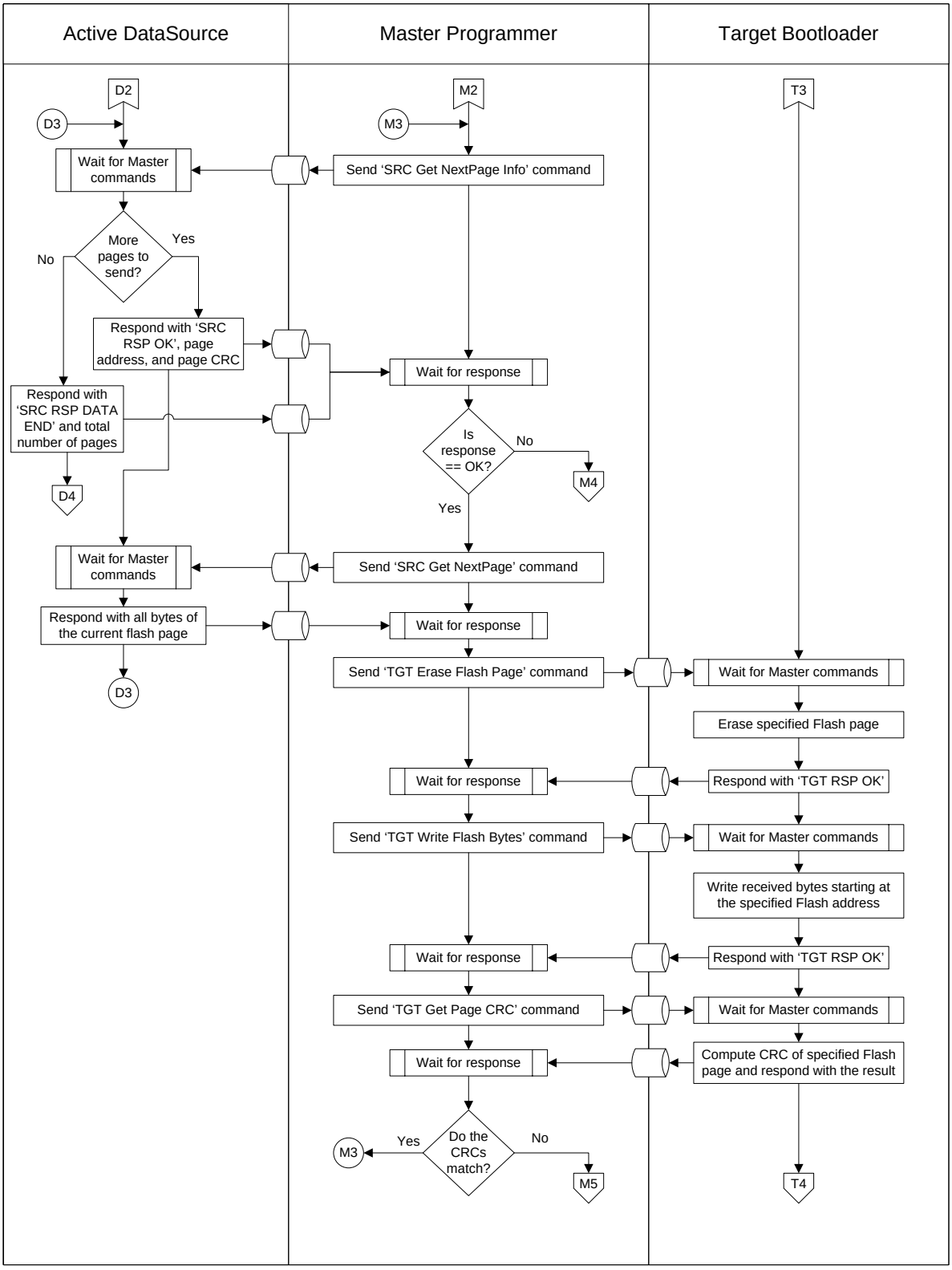


Figure 9. Firmware Update Process Flow Diagram (Page 3 of 4)

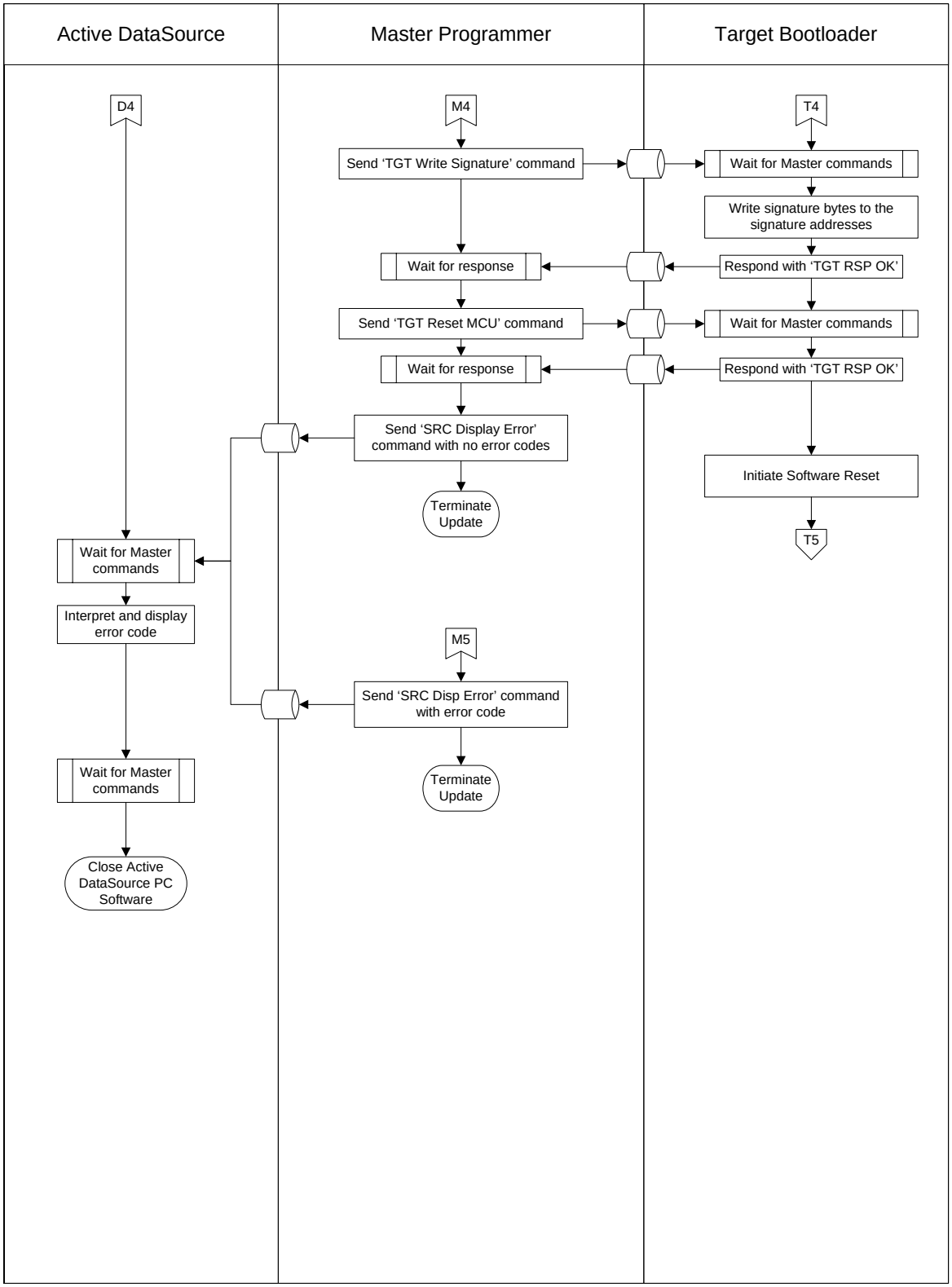


Figure 10. Firmware Update Process Flow Diagram (Page 4 of 4)

APPENDIX A—TARGET INTERFACE

Commands Sent from Master to Target:

Command Name	Command Code	Parameters						
	byte0	byte1	byte2	byte3	byte4	byte5	byte6	byte7
TGT_Enter_BL_Mode	0x10	[dev_serial0]	[dev_serial1]	…	…	…	…	[dev_serial6]
TGT_Get_Info	0x20	xx	xx	xx	xx	xx	xx	xx
TGT_Erase_-Flash_Page	0x30	key_code0	key_code1	addr0	addr1	addr2	Reserved	Reserved
TGT_Write_Signature	0x40	key_code0	key_code1	sigbyte0	sigbyte1	Reserved	Reserved	Reserved
TGT_Reset_MCU	0x50	Reserved	Reserved	xx	xx	xx	xx	xx
TGT_Get_Page_CRC	0x60	addr0	addr1	addr2	Reserved	xx	xx	xx

Note: All the above commands are of fixed length of 8 bytes.

Command Name	Command Code	Parameters						
	byte0	byte1	byte2	byte3	byte4	byte5	byte6	byte7
TGT_Write_Flash_-Bytes	0x70	key_code0	key_code1	addr0	addr1	addr2	numbytes0	numbytes1

Note: Master will wait for an RSP code from target after the above command before sending the data bytes. “numbytes” counts all bytes that will be sent by the master after it receives a TGT_RSP_OK code; Min: 1; Max: BL buffer size

Responses Sent from Target to Master:

Command Name	Response				
	byte0	byte1	byte2	byte3	byte4
TGT_Enter_BL_Mode	RSP code				
TGT_Erase_Flash_Page	RSP code				
TGT_Write_Signature	RSP code				
TGT_Reset_MCU	RSP code				
TGT_Get_Page_CRC	RSP code	crc byte0	crc byte1	[crc byte2]	[crc byte3]
TGT_Write_Flash_Bytes	RSP code				
Note: The one-byte response codes (RSP code) are specified in the file "Fxxx_Target_Interface.h"					

Command Name	Response									
	byte0	byte1	byte2	…	byteN	byte(N+1)	byte(N+2)	…	…	byte(N+M)
TGT_Get_Info	RSP code	BL IB byte0	BL IB byte1	…	BL IB byte (N-1)	App IB byte0	App IB byte1	…	…	App IB byte(M-1)
Notes: The one-byte response codes (RSP code) are specified in the file "Fxxx_Target_Interface.h" BL IB = Bootloader InfoBlock; App IB = Application InfoBlock N = Bootloader InfoBlock length; this is specified by BL IB byte0 M = Application InfoBlock length; this is specified by App IB byte0; if the application image is not valid, App IB byte0 is set to 0 and M = 1.										

APPENDIX B—MASTER–DATA SOURCE INTERFACE

Commands Sent from Master to Data Source

Command Name	Command Code
	byte0
SRC_Get_Info	0x01
SRC_Get_NextPage_Info	0x02
SRC_Get_NextPage	0x03
SRC_Disp_Error	0x8y
Note: y = error code to be displayed.	

Command Name	Command Code	Parameters								
	byte0	byte1	byte2	…	byteN	byte(N+1)	byte(N+2)	…	…	byte(N+M)
SRC_ Disp_ TGT_Info	0x04	BL IB byte0	BL IB byte1	…	BL IB byte(N-1)	App IB byte0	App IB byte1	…	…	App IB byte(M-1)

Notes: BL IB = Bootloader InfoBlock; App IB = Application InfoBlock
N = Bootloader InfoBlock length; this is specified by BL IB byte0
M = Application InfoBlock length; this is specified by App IB byte0; if the application image is not valid, App IB byte0 is set to 0 and M = 1.

Responses Sent from DataSource to Master:

Command Name	Response							
	byte0	**byte1**	**byte2**	**byte3**	**byte4**	**byte5**	**byte6**	**byte7**
SRC_Get_ NextPage_ Info	RSP code	Page addr0	Page addr1	Page addr2	Page crc0	Page crc1	[Page crc2]	[Page crc3]
SRC_Disp_ Error	RSP code							
SRC_Disp_ TGT_Info	RSP code							

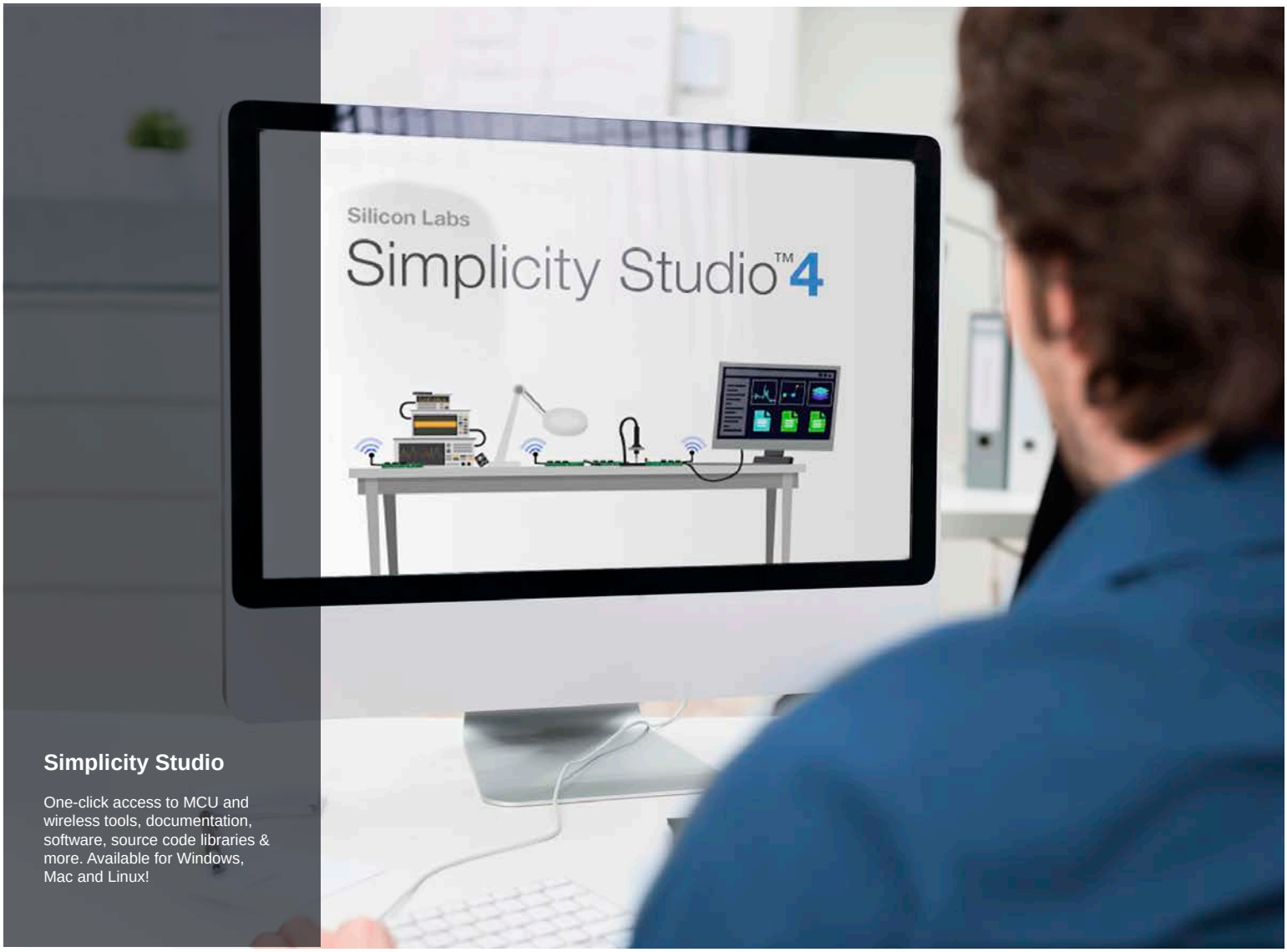
Notes:
1. The one-byte response codes (RSP code) are specified in the file "Fxxx_DataSource_Interface.h"
2. For SRC_Get_NextPage_Info, if RSP code is SRC_RSP_DATA_END, then 'Page addr0' is replaced by total number of specified pages.

Command Name	Response					
	byte0	**byte1**	**byte2**	**byte3**	**byteT**	**byte(T+1)**
SRC_Get_NextPage	RSP code	Page byte0	Page byte1	…	Page byte(T-1)	RSP code

Notes: The one-byte response codes (RSP code) are specified in the file "Fxxx_DataSource_Interface.h"
T = Flash page size

Command Name	Response									
	byte0	**byte1**	**…**	**byteM**	**byte (M+1)**	**byte (M+2)**	**byte (M+3)**	**byte (M+4)**	**byte (M+5)**	**byte(M+6)**
SRC_Get_Info	RSP code	App IB byte0	…	App IB byte(M-1)	App FW start addr0	App FW start addr1	App FW start addr2	App FW end addr0	App FW end addr1	App FW end addr2

Notes: The one-byte response codes (RSP code) are specified in the file "Fxxx_DataSource_Interface.h"
BL IB = Bootloader InfoBlock; App IB = Application InfoBlock
M = Application InfoBlock length; this is specified by App IB byte0.
The source waits for an SRC_RSP_OK from the master after sending the first two bytes before it sends the remaining bytes.



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