

# **MSP432E4 SimpleLink™ Microcontrollers Bootloader (BSL)**

## Contents

1	Introduction .....	2
	1.1 Source Code Overview .....	2
2	Start-up Code .....	4
3	Serial Update .....	6
	3.1 BSL Hardware Setup Overview .....	6
	3.2 Packet Handling .....	7
	3.3 Transport Layer .....	7
	3.4 Serial Commands .....	8
	3.5 Serial Command Responses .....	11
	3.6 Serial Bootloader Protocol Sequence .....	12
4	Ethernet Update .....	16
5	CAN Update .....	16
	5.1 CAN Bus Clocking .....	16
	5.2 CAN Commands .....	17
6	USB Device (DFU) Update .....	18
	6.1 USB Device Firmware Upgrade Overview .....	18
	6.2 USB Download Commands .....	21
7	Customization .....	25
8	Configuration .....	26
9	Source Details .....	37
	9.1 Autobaud Functions .....	37
	9.2 CAN Functions .....	38
	9.3 Decryption Functions .....	39
	9.4 Ethernet Functions .....	39
	9.5 File System Functions .....	39
	9.6 I <sup>2</sup> C Functions .....	40
	9.7 Main Functions .....	41
	9.8 Packet Handling Functions .....	41
	9.9 SSI Functions .....	43
	9.10 UART Functions .....	44
	9.11 Update Check Functions .....	45
	9.12 USB Device Functions .....	45
10	References .....	50

### Trademarks

SimpleLink is a trademark of Texas Instruments.  
 Arm, Cortex are registered trademarks of Arm Limited.  
 All other trademarks are the property of their respective owners.

## 1 Introduction

The bootloader is a small piece of code that can be programmed at the beginning of flash to act as an application loader as well as an update mechanism for applications running on a SimpleLink™ MSP432E4 Arm® Cortex®-M4-based microcontroller. The bootloader can be built to use the UART, SSI, I<sup>2</sup>C, CAN, Ethernet, or USB ports to update the code on the microcontroller. The bootloader is customizable through source code modifications, or simply deciding at compile time which routines to include. Because full source code is provided, the bootloader can be completely customized.

---

**NOTE:** For UART, I<sup>2</sup>C, and SSI the ROM bootloader is fixed for UART0, I<sup>2</sup>C0, and SSI0. If the application uses any other instance of the peripheral, then it must be loaded into the flash by an emulator.

---

When using the ROM bootloader the following pins are preconfigured by the ROM bootloader for the UART0, I<sup>2</sup>C0, and SSI0 peripherals (see [Table 1](#)).

**Table 1. ROM Bootloader Default Pin Configuration**

Pin Name	Peripheral	Peripheral Function
PA0	UART0	UART receive pin for the target device
PA1	UART0	UART transmit pin for the target device
PB2	I <sup>2</sup> C0	I <sup>2</sup> C serial clock pin. An external pull up is required.
PB3	I <sup>2</sup> C0	I <sup>2</sup> C serial data pin. An external pull up is required
PA2	SSI0	SSI slave clock input pin
PA3	SSI0	SSI slave chip select input pin
PA4	SSI0	SSI slave data input pin connected to master output
PA5	SSI0	SSI slave data output pin connected to master input

Three update protocols are used. On UART, SSI, I<sup>2</sup>C, and CAN, a custom protocol is used to communicate with the download utility to transfer the firmware image and program it into flash. When using Ethernet or USB DFU, however, different protocols are employed. On Ethernet, the standard bootstrap protocol (BOOTP) is used and, for USB DFU, updates are performed through the standard Device Firmware Upgrade (DFU) class.

### 1.1 Source Code Overview

The following is an overview of the organization of the source code provided with the bootloader.

#### **bl\_autobaud.c**

The code for performing the auto-baud operation on the UART port. This is separate from the remainder of the UART code so that the linker can remove it when it is not used.

#### **bl\_can.c**

The functions for performing a firmware update through the CAN port.

#### **bl\_can.h**

Definitions used by the CAN update routine.

#### **bl\_check.c**

The code to check if a firmware update is required, or if a firmware update is being requested by the user.

#### **bl\_check.h**

Prototypes for the update check code.

#### **bl\_commands.h**

The list of commands and return messages supported by the bootloader.

**bl\_config.c**

A dummy source file used to translate the bl\_config.h C header file into a header file that can be included in assembly code. This is needed for the Keil tool chain since it is not able to pass assembly source code through the C preprocessor.

**bl\_config.h.tmpl**

A template for the bootloader configuration file. This contains all of the possible configuration values.

**bl\_decrypt.c**

The code to perform an in-place decryption of the downloaded firmware image. No decryption is actually performed in this file; this is simply a stub that can be expanded to perform the require decryption.

**bl\_decrypt.h**

Prototypes for the in-place decryption routines.

**bl\_enet.c**

The functions for performing a firmware update through the Ethernet port.

**bl\_fs.c**

The functions to provide simple FAT file system support.

**bl\_fs.h**

Prototypes for the file system functions.

**bl\_i2c.c**

The functions for transferring data through the I<sup>2</sup>C port.

**bl\_i2c.h**

Prototypes for the I<sup>2</sup>C transfer functions.

**bl\_link.ld**

The linker script used when the gcc compiler is being used to build the bootloader.

**bl\_link.sct**

The linker script used when the rvmdk compiler is being used to build the bootloader.

**bl\_link.xcl**

The linker script used when the ewarm compiler is being used to build the bootloader.

**bl\_link\_ccs.cmd**

The linker script used when the ccs compiler is being used to build the bootloader.

**bl\_main.c**

The main control loop of the bootloader.

**bl\_packet.c**

The functions for handling the packet processing of commands and responses.

**bl\_packet.h**

Prototypes for the packet handling functions.

**bl\_ssi.c**

The functions for transferring data through the SSI port.

**bl\_ssi.h**

Prototypes for the SSI transfer functions.

**bl\_startup\_ccs.s**

The start-up code used when the ccs compiler is being used to build the bootloader.

**bl\_startup\_ewarm.S**

The start-up code used when the ewarm compiler is being used to build the bootloader.

**bl\_startup\_gcc.S**

The start-up code used when the gcc compiler is being used to build the bootloader.

**bl\_startup\_rvmdk.S**

The start-up code used when the rvmdk compiler is being used to build the bootloader.

**bl\_uart.c**

The functions for transferring data through the UART port.

**bl\_uart.h**

Prototypes for the UART transfer functions.

**bl\_usb.c**

Main functions implementing the USB DFU protocol bootloader.

**bl\_usbfuncs.c**

A cut-down version of the USB library containing support for enumeration and the endpoint 0 transactions required to implement the USB DFU device.

**bl\_usbfuncs.h**

Prototypes for the functions provided in bl\_usbfuncs.c.

**usbdfu.h**

Type definitions, labels related to the USB Device Firmware Upgrade class bootloader.

## 2 Start-up Code

The start-up code contains the minimal set of code required to configure a vector table, initialize memory, copy the bootloader from flash to SRAM, and execute from SRAM. Because some tool chain-specific constructs are used to indicate where the code, data, and bss segments reside in memory, each supported tool chain has its own separate file that implements the start-up code. The start-up code is contained in the following files:

- bl\_startup\_ewarm.S (IAR Embedded Workbench)
- bl\_startup\_gcc.S (GNU GCC)
- bl\_startup\_rvmdk.S (Keil RV-MDK)
- bl\_startup\_ccs.s (Texas Instruments Code Composer Studio)

Accompanying the start-up code for each tool chain are linker scripts that are used to place the vector table, code segment, data segment initializers, and data segments in the appropriate locations in memory. The scripts are located in the following files:

- bl\_link.lids (GNU GCC)
- bl\_link.sct (Keil RV-MDK)
- bl\_link.xcl (IAR Embedded Workbench)
- bl\_link\_ccs.cmd (TI Code Composer Studio)

The bootloader's code and its corresponding linker script use a memory layout that exists entirely in SRAM. This means that the load address of the code and read-only data are not the same as the execution address. This memory map allows the bootloader to update itself since it is actually running from SRAM only. The first part of SRAM is used as the copy space for the bootloader while the rest is reserved for stack and read/write data for the bootloader. After the bootloader calls the application, all SRAM becomes usable by the application.

The vector table of the Cortex-M4 microprocessor contains four required entries: the initial stack pointer, the reset handler address, the NMI handler address, and the hard fault handler address. Upon reset, the processor loads the initial stack pointer and then starts executing the reset handler. The initial stack pointer is required since an NMI or hard fault can occur at any time; the stack is required to take those interrupts since the processor automatically pushes eight items onto the stack.

The Vectors array contains the bootloader's vector table which varies in size based on the addition of the auto-baud feature or USB DFU support. These options requires additional interrupt handlers expand the vector table to populate the relevant entries. Since the bootloader executes from SRAM and not from flash, tool chain-specific constructs are used to provide a hint to the linker that this array is located at 0x2000.0000.

The `IntDefaultHandler` function contains the default fault handler. This is a simple infinite loop, effectively halting the application if any unexpected fault occurs. The application state is, therefore, preserved for examination by a debugger. If desired, a customized bootloader can provide its own handlers by adding the appropriate handlers to the Vectors array.

After a reset, the start-up code copies the bootloader from flash to SRAM, branches to the copy of the bootloader in SRAM, and checks to see if an application update should be performed by calling `CheckForceUpdate()`. If an update is not required, the application is called. Otherwise the functions that are called are based on the mode of operation for the bootloader. For UART, SSI, and I<sup>2</sup>C, the microcontroller is initialized by calling `ConfigureDevice()` and then the boot load calls the serial control loop `Updater()`. For Ethernet, the microcontroller is initialized by calling `ConfigureEnet()` and then the bootloader calls the Ethernet control loop `UpdateBOOTP()`. For CAN, the microcontroller is initialized by calling `ConfigureCAN()` and then the bootloader calls the CAN control loop `UpdaterCAN()`. For USB, the microcontroller is initialized by calling `ConfigureUSB()` after which the function `UpdaterUSB()` configures the USB interface for device mode.

The check for an application update (in `CheckForceUpdate()`) consists of checking the beginning of the application area and optionally checking the state of a GPIO pin. The application is assumed to be valid if the first location is a valid stack pointer (that is, it resides in SRAM, and has a value of 0x2xxx.xxxx), and the second location is a valid reset handler address (that is, it resides in flash, and has a value of 0x000x.xxxx, where the value is odd). If either of these tests fail, then the application is assumed to be invalid and an update is forced. The GPIO pin check can be enabled with `ENABLE_UPDATE_CHECK` in the `bl_config.h` header file, in which case an update can be forced by changing the state of a GPIO pin (for example, with a push button). If the application is valid and the GPIO pin is not requesting an update, the application is called. Otherwise, an update is started by entering the main loop of the bootloader.

Additionally, the application can call the bootloader in order to perform an application-directed update. In this case, the bootloader assumes that the application has already configured the peripheral that it will use for the update. This allows the bootloader to use the peripheral as is to perform the update. The bootloader also assumes that the interrupt to the core has been left enabled as well, which means that that application should not call `IntMasterDisable()` before calling the bootloader. After the application calls the bootloader, the bootloader copies itself to SRAM, branches to the SRAM copy of the bootloader, and starts the update by calling `Updater()` (for UART, SSI, and I<sup>2</sup>C), `UpdateBOOTP()` (for Ethernet), `AppUpdaterCAN()` (for CAN) or `AppUpdaterUSB()` (for USB). The `SVCALL` entry of the vector table contains the location of the application-directed update entry point.

### 3 Serial Update

When performing an update through a serial port (UART, SSI, or I<sup>2</sup>C), ConfigureDevice() is used to configure the selected serial port, making it ready to be used to update the firmware. Then, Updater() sits in an endless loop accepting commands and updating the firmware when requested. All transmissions from this main routine use the packet handler functions (SendPacket(), ReceivePacket(), AckPacket(), and NakPacket()). After the update is complete, the bootloader can be reset by issuing a reset command to the bootloader.

When a request to update the application comes through and FLASH\_CODE\_PROTECTION is defined, the bootloader first erases the entire application area before accepting the binary for the new application. This prevents a partial erase of flash from exposing any of the code before the new binary is downloaded to the microcontroller. The bootloader itself is left in place so that it does not boot a partially erased program. After all of the application flash area has been successfully erased, the bootloader proceeds with the download of the new binary. When FLASH\_CODE\_PROTECTION is not defined, the bootloader only erases enough space to fit the new application that is being downloaded.

In the event that the bootloader itself needs to be updated, the bootloader must replace itself in flash. An update of the bootloader is recognized by performing a download to address 0x0000.0000. The bootloader operates differently based on the setting of FLASH\_CODE\_PROTECTION. When FLASH\_CODE\_PROTECTION is defined and the download address indicates a bootloader update, the bootloader protects any application code already on the microcontroller by erasing the entire application area before erasing and replacing itself. If the process is interrupted at any point, either the old bootloader remains present in the flash and does not boot the partial application or the application code will have already been erased. When FLASH\_CODE\_PROTECTION is not defined, the bootloader only erases enough space to fit its own code and leaves the application intact.

#### 3.1 BSL Hardware Setup Overview

Figure 1 is referenced to illustrate the serial bootloader setup. The BSL scripser tool is used on the PC host to perform the update of the application firmware image on the target device. The [BSL Rocket tool](#) serves as an interface tool between the PC and the target device to update the application firmware using UART, I<sup>2</sup>C, and SSI interfaces.

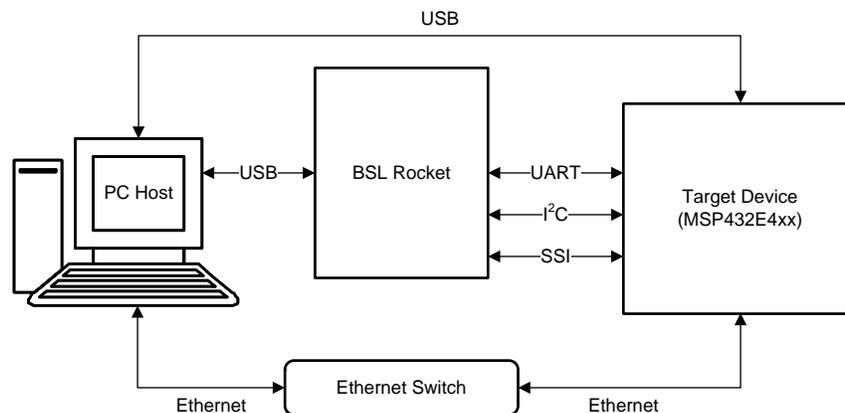
To perform a USB update, a USB cable must be connected between the PC and the target device.

To perform an update over Ethernet, a generic 10/100 Mbps switch is required. The PC and the board with the target device are connected to the switch which in turn may be connected to the LAN network.

---

**NOTE:** The BSL Rocket tool is required only when using UART, I<sup>2</sup>C, and SSI as the interface for performing application firmware update. It is not required when using either USB or Ethernet for performing application firmware update.

---



**Figure 1. Generic System Setup**

## 3.2 Packet Handling

The bootloader uses well-defined packets to ensure reliable communications with the update program. The packets are always acknowledged or not acknowledged by the communicating devices. The packets use the same format for receiving and sending packets. This includes the method used to acknowledge successful or unsuccessful reception of a packet. While the actual signaling on the serial ports is different, the packet format remains independent of the method of transporting the data.

The bootloader uses the `SendPacket()` function to send a packet of data to another device. This function encapsulates all of the steps necessary to send a valid packet to another device including waiting for the acknowledge or not-acknowledge from the other device. The following steps must be performed to successfully send a packet:

1. Send out the size of the packet that will be sent to the device. The size is always the size of the data + 2.
2. Send out the checksum of the data buffer to help ensure proper transmission of the command. The checksum algorithm is implemented in the `Checksum()` function provided and is simply a sum of the data bytes.
3. Send out the actual data bytes.
4. Wait for a single byte acknowledgment from the device that it either properly received the data or that it detected an error in the transmission.

Received packets use the same format as sent packets. The bootloader uses the `ReceivePacket()` function in order to receive or wait for a packet from another device. This function does not take care of acknowledging or not-acknowledging the packet to the other device. This allows the contents of the packet to be checked before sending back a response. The following steps must be performed to successfully receive a packet:

1. Wait for nonzero data to be returned from the device. This is important as the device may send zero bytes between a sent and received data packet. The first nonzero byte received will be the size of the packet that is being received.
2. Read the next byte which will be the checksum for the packet.
3. Read the data bytes from the device. There will be (packet size – 2 bytes) of data sent during the data phase. For example, if the packet size was 3, then there is only 1 byte of data to be received.
4. Calculate the checksum of the data bytes and ensure if it matches the checksum received in the packet.
5. Send an acknowledge or not-acknowledge to the device to indicate the successful or unsuccessful reception of the packet.

The steps necessary to acknowledge reception of a packet are implemented in the `AckPacket()` function. Acknowledge bytes are sent out whenever a packet is successfully received and verified by the bootloader.

A not-acknowledge byte is sent out whenever a sent packet is detected to have an error, usually as a result of a checksum error or just malformed data in the packet. This allows the sender to re-transmit the previous packet.

## 3.3 Transport Layer

The bootloader supports updating through the I<sup>2</sup>C, SSI, and UART ports which are available on MSP432E4 microcontrollers. The SSI port has the advantage of supporting higher and more flexible data rates but it also requires more connections to the microcontroller. The UART has the disadvantage of having slightly lower and possibly less flexible rates. However, the UART requires fewer pins and can be easily implemented with any standard UART connection. The I<sup>2</sup>C interface also provides a standard interface, only uses two wires, and can operate at comparable speeds to the UART and SSI interfaces.

### 3.3.1 I<sup>2</sup>C Transport

The I<sup>2</sup>C handling functions are `I2CSend()`, `I2CReceive()`, and `I2CFlush()` functions. The connections required to use the I<sup>2</sup>C port are the following pins: I2CSCL and I2CSDA. The device communicating with the bootloader must operate as the I<sup>2</sup>C master and provide the I2CSCL signal. The I2CSDA pin is open drain and can be driven by either the master or the slave I<sup>2</sup>C device.

### 3.3.2 SSI Transport

The SSI handling functions are SSISend(), SSIReceive(), and SSIFlush(). The connections required to use the SSI port are the following four pins: SSITx, SSIRx, SSIClk, and SSIFss. The device communicating with the bootloader is responsible for driving the SSIRx, SSIClk, and SSIFss pins, while the MSP432E4 microcontroller drives the SSITx pin. The format used for SSI communications is the Motorola format with SPH set to 1 and SPO set to 1 (see the device-specific data sheet for more information on this format). The SSI interface has a hardware requirement that limits the maximum rate of the SSI clock to be at most 1/12 the frequency of the microcontroller running the bootloader.

### 3.3.3 UART Transport

The UART handling functions are UARTSend(), UARTReceive(), and UARTFlush(). The connections required to use the UART port are the following two pins: U0Tx and U0Rx. The device communicating with the bootloader is responsible for driving the U0Rx pin on the MSP432E4 microcontroller, while the MSP432E4 microcontroller drives the U0Tx pin.

While the baud rate is flexible, the UART serial format is fixed at 8 data bits, no parity, and one stop bit. The baud rate used for communication can either be auto-detected by the bootloader, if the auto-baud feature is enabled, or it can be fixed at a baud rate supported by the device communicating with the bootloader. The only requirement on baud rate is that the baud rate should be no more than 1/32 the frequency of the microcontroller that is running the bootloader. This is the hardware requirement for the maximum baud rate for a UART on any MSP432E4 microcontroller.

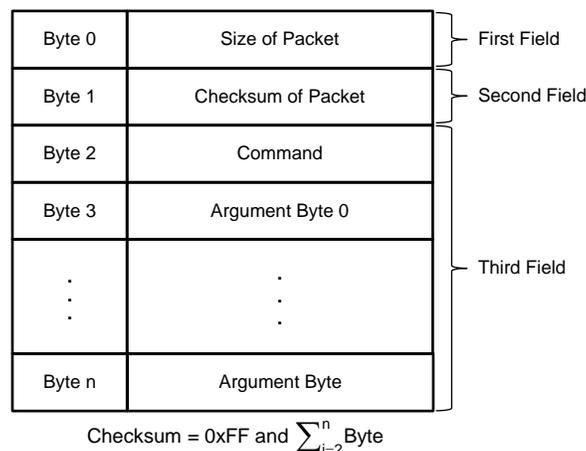
When using a fixed baud rate, the frequency of the crystal connected to the microcontroller must be specified. Otherwise, the bootloader cannot configure the UART to operate at the requested baud rate.

The bootloader provides a method to automatically detect the baud rate being used to communicate with it. This automatic baud rate detection is implemented in the UARTAutoBaud() function. The auto-baud function attempts to synchronize with the updater application and indicates if it is successful in detecting the baud rate or if it failed to properly detect the baud rate. The bootloader can make multiple calls to UARTAutoBaud() to attempt to retry the synchronization if the first call fails. In the example bootloader provided, when the auto-baud feature is enabled, the bootloader will wait forever for a valid synchronization pattern from the host.

## 3.4 Serial Commands

The serial bootloader uses a well defined packet structure for commands and responses to establish communication between the programmer and the target device. The size and response for each of the commands is fixed, so that the programmer and target can perform the most efficient communication.

Each command packet is made up of three fields (see [Figure 2](#)).



**Figure 2. Generic Command Packet Structure**

The first field of the packet is 1 byte that contains the size of the packet including the first field. The second field of the packet is also 1 byte and contains the checksum formed over the rest of the packet. The third field is the command followed by its arguments. This field can have a variable size based on the command and the arguments applicable to the specific command. The descriptions of the commands that are used by the custom protocol on the UART, SSI, and I<sup>2</sup>C ports follow.

### COMMAND\_PING

This command is used to receive an acknowledge from the bootloader indicating that communication has been established. [Figure 3](#) shows the command format, and it has a single byte in the third field.

Byte 0	Size = 0x03
Byte 1	Checksum = 0x20
Byte 2	Command = 0x20

**Figure 3. Ping Command**

### COMMAND\_DOWNLOAD

This command is sent to the bootloader to indicate where to store data and how many bytes will be sent by the `COMMAND_SEND_DATA` commands that follow. The command consists of two 32-bit values that are both transferred MSB first. The first 32-bit value is the address to start programming data into, while the second is the 32-bit size of the data that will be sent. This command also triggers an erase of the full application area in the flash or possibly the entire flash depending on the address used. This causes the command to take longer to send the ACK or NAK in response to the command. This command should be followed by a `COMMAND_GET_STATUS` to ensure that the program address and program size were valid for the microcontroller running the bootloader.

[Figure 4](#) shows the command format.

Byte 0	Size = 0x0B
Byte 1	Checksum = 0xFF
Byte 2	Command = 0x21
Byte 3	Program Address [31:24]
Byte 4	Program Address [23:16]
Byte 5	Program Address [15:8]
Byte 6	Program Address [7:0]
Byte 7	Program Size [31:24]
Byte 8	Program Size [23:16]
Byte 9	Program Size [15:8]
Byte 10	Program Size [7:0]

**Figure 4. Download Command**

### COMMAND\_RUN

This command is sent to the bootloader to transfer execution control to the specified address. The command is followed by a 32-bit value, transferred MSB first, that is the address to which execution control is transferred.

Figure 5 shows the command format.

Byte 0	Size = 0x07
Byte 1	Checksum = 0xXX
Byte 2	Command = 0x22
Byte 3	Run Address [31:24]
Byte 4	Run Address [23:16]
Byte 5	Run Address [15:8]
Byte 6	Run Address [7:0]

**Figure 5. Run Command**

### COMMAND\_GET\_STATUS

This command returns the status of the last command that was issued. Typically, this command should be received after every command is sent to ensure that the previous command was successful or, if unsuccessful, to properly respond to a failure. The command requires one byte in the data of the packet and the bootloader should respond by sending a packet with one byte of data that contains the current status code.

Figure 6 shows the command format.

Byte 0	Size = 0x03
Byte 1	Checksum = 0x23
Byte 2	Command = 0x23

**Figure 6. Get Status Command**

### COMMAND\_SEND\_DATA

This command should only follow a COMMAND\_DOWNLOAD command or another COMMAND\_SEND\_DATA command, if more data is needed. Consecutive send data commands automatically increment the address and continue programming from the previous location. The transfer size is limited by the size of the receive buffer in the bootloader (as configured by the BUFFER\_SIZE parameter). The command terminates programming when the number of bytes indicated by the COMMAND\_DOWNLOAD command has been received. Each time this function is called, it should be followed by a COMMAND\_GET\_STATUS command to ensure that the data was successfully programmed into the flash. If the bootloader sends a NAK to this command, the bootloader does not increment the current address which allows for retransmission of the previous data.

Figure 7 shows the command format.

Byte 0	Size = 0xYY
Byte 1	Checksum = 0xXX
Byte 2	Command = 0x24
Byte 3	Data0 [7:0]
Byte 4	Data0 [15:8]
Byte 5	Data0 [23:16]
Byte 6	Data0 [31:24]
⋮	⋮
Byte 4N+3	DataN [7:0]
Byte 4N+4	DataN [15:8]
Byte 4N+5	DataN [23:16]
Byte 4N+6	DataN [31:24]

**Figure 7. Send Data Command**

### COMMAND\_RESET

This command is used to tell the bootloader to reset. This is used after downloading a new image to the microcontroller to cause the new application or the new bootloader to start from a reset. The normal boot sequence occurs and the image runs as if from a hardware reset. It can also be used to reset the bootloader if a critical error occurs and the host device wants to restart communication with the bootloader.

The bootloader responds with an ACK signal to the host device before actually executing the software reset on the microcontroller running the bootloader. This informs the updater application that the command was received successfully and the part will be reset.

Figure 8 shows the format of the command.

Byte 0	Size = 0x03
Byte 1	Checksum = 0x25
Byte 2	Command = 0x25

**Figure 8. COMMAND\_RESET packet structure on the bus**

```
unsigned char ucCommand[1];
ucCommand[0] = COMMAND_RESET;
```

### 3.5 Serial Command Responses

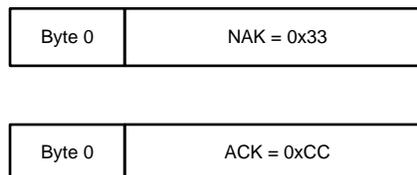
There are two types of response packets generated. The first type of response packet (Type-1) is an ACK or NAK response, which is generated as a single byte packet when a command packet is sent. The second type of response packet (Type-2) is specific to the GET\_STATUS command packet and has the same structure as that of a command packet.

- ACK Response packet: The ACK response packet is sent by either the target device in response to a

command packet or by the programmer in response to the Type-2 response packet. It is generated when the command packet or a Type-2 packet has no checksum error and the parameters are defined correctly.

- **NAK Response packet:** The NAK response packet is sent by either the target device in response to a command packet or by the programmer in response to the Type-2 response packet. It is generated when the command packet or a Type-2 packet has either a checksum error or the parameters are not defined correctly.

The structure of the ACK and NAK response packet is shown in [Figure 9](#).



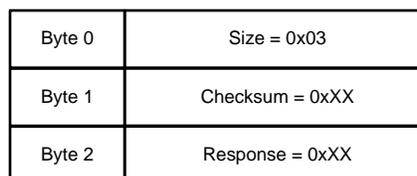
**Figure 9. ACK and NAK Response**

The Type-2 response packet is sent by the target device to the programmer in response to the GET\_STATUS command packet and its structure is shown in [Figure 10](#). The valid values for the Type-2 response packet are defined in [Table 2](#).

---

**NOTE:** The checksum value is not specified as this should be computed per the equation in [Figure 2](#) on Byte-2 and placed in Byte-1 location

---



**Figure 10. Type-2 Response**

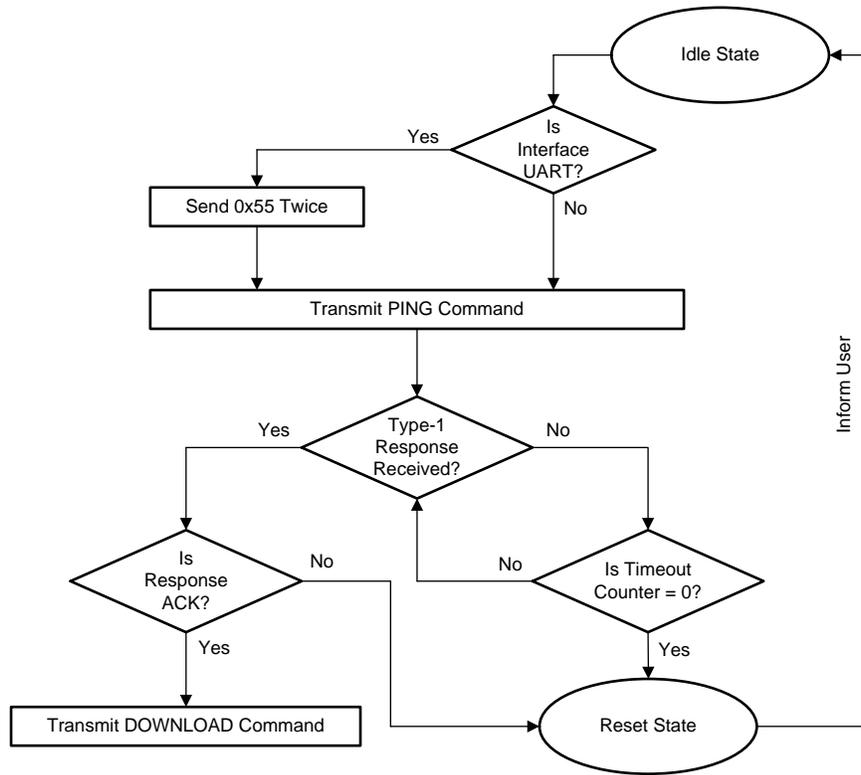
**Table 2. Response Value Description**

Response Packet	Value	Comments
COMMAND_RET_SUCCESS	0x40	If the previous command was defined and format was correct
COMMAND_RET_UNKNOWN_CMD	0x41	If the previous command was an unknown command
COMMAND_RET_INVALID_CMD	0x42	If the previous command had a format error
COMMAND_RET_INVALID_ADR	0x43	If the previous download address was an invalid address
COMMAND_RET_FLASH_FAIL	0x44	If the attempt to program or erase the flash failed
COMMAND_RET_CRC_FAIL	0x45	If the CRC check of the image failed, if CRC check has been enabled

### 3.6 Serial Bootloader Protocol Sequence

Having described the command and response packet types and structures, this section elaborates on the serial bootloader protocol sequence that is used between the programmer and the target device to download and execute an application image. While the steps are identical for all three serial interfaces of UART, I<sup>2</sup>C and SSI, there is one additional step that is handled in UART called Auto-Baud. This step is done so that the programmer and target device can synchronize on the baud rate to be used.

[Figure 11](#) through [Figure 14](#) shows how the BSL Rocket communicates with the target device to download the application image along with required error handling.



**Figure 11. Serial Protocol Sequence-1**

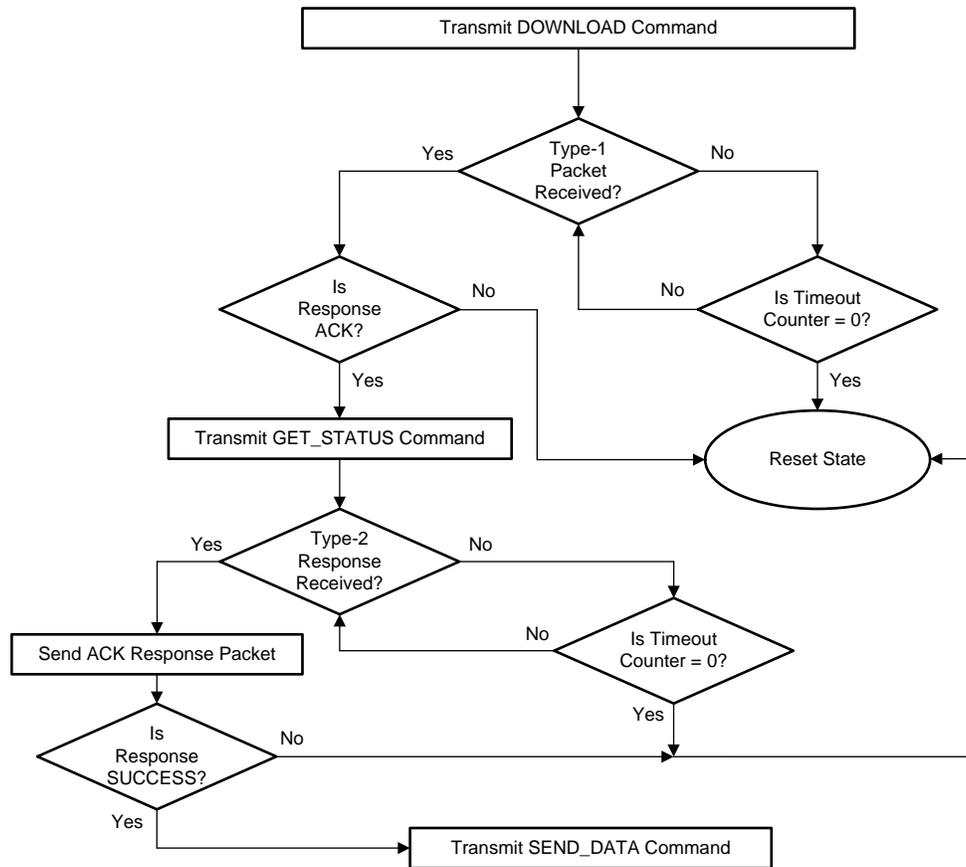


Figure 12. Serial Protocol Sequence-2

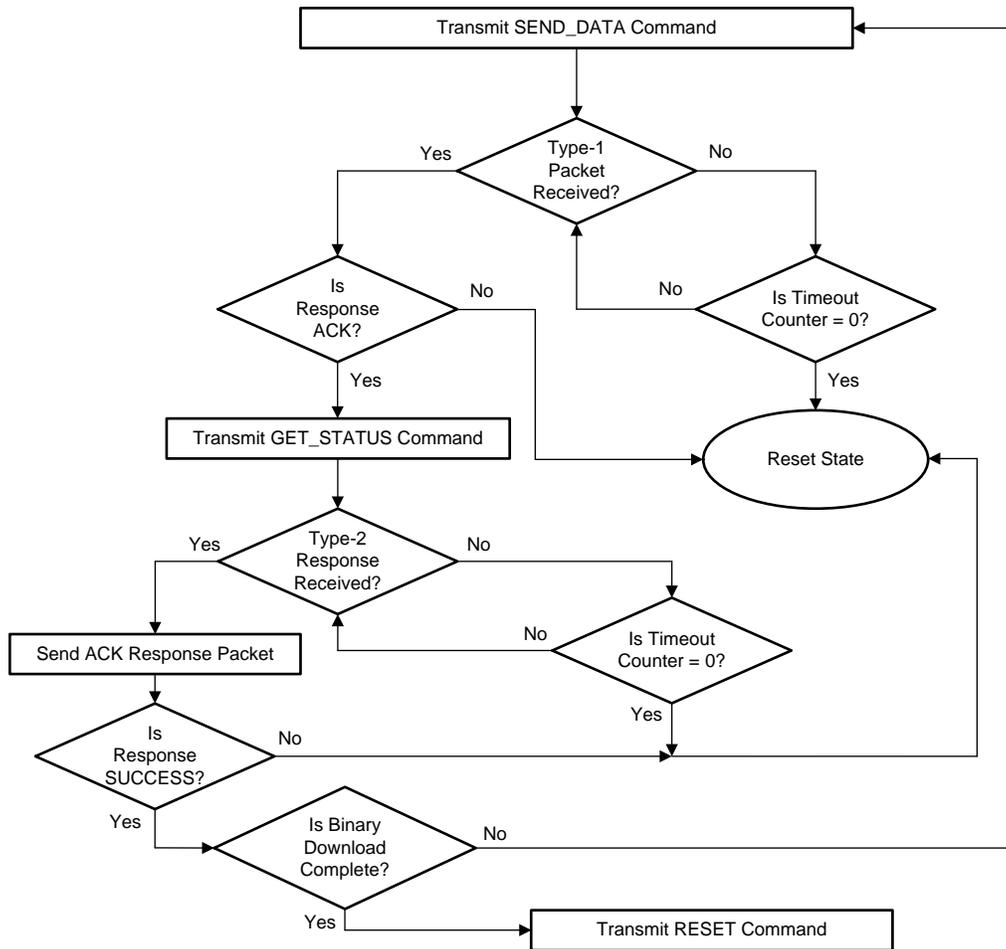


Figure 13. Serial Protocol Sequence-3

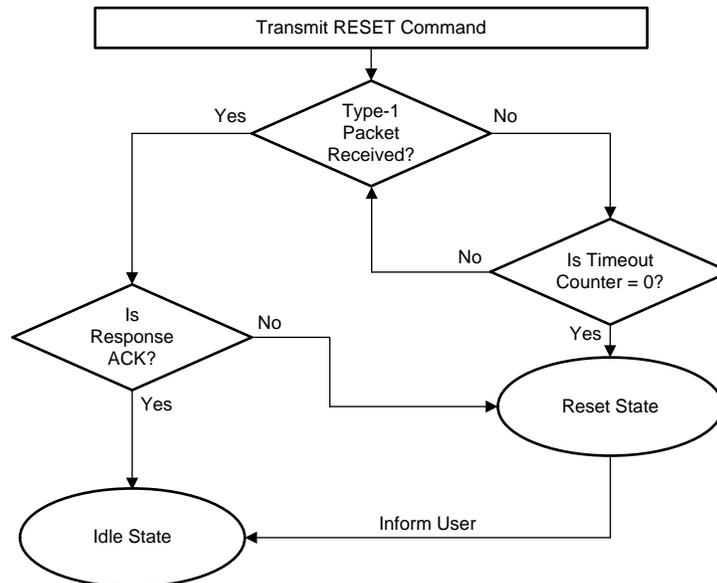


Figure 14. Serial Protocol Sequence-4

## 4 Ethernet Update

When performing an Ethernet update, `ConfigureEnet()` is used to configure the Ethernet controller, making it ready to be used to update the firmware. Then, `UpdateBOOTP()` begins the process of the firmware update.

The bootstrap protocol (BOOTP) is a predecessor to the DHCP protocol and is used to discover the IP address of the client, the IP address of the server, and the name of the firmware image to use. BOOTP uses UDP/IP packets to communicate between the client and the server; the bootloader acts as the client. First, it will send a BOOTP request using a broadcast message. When the server receives the request, it will reply, thereby informing the client of its IP address, the IP address of the server, and the name of the firmware image. When this reply is received, the BOOTP protocol has completed.

Then, the trivial file transfer protocol (TFTP) is used to transfer the firmware image from the server to the client. TFTP also uses UDP/IP packets to communicate between the client and the server, and the bootloader also acts as the client in this protocol. As each data block is received, it is programmed into flash. When all data blocks are received and programmed, the device is reset, causing it to start running the new firmware image.

The uIP stack is used to implement the UDP/IP connections. The TCP support is not needed and is therefore disabled, greatly reducing the size of the stack.

The Ethernet controller will be configured to use the MAC address stored in the USER0/UART1 data registers or the MAC address that is provided in the bootloader configuration file (`bl_config.h`). When using the MAC address from USER0/USER1, it will be interpreted as a MAC address of U0B0:U0B1:U0B2:U1B0:U1B1:U1B2 (where U0B0 is USER0 bits 7-0, or byte 0, U0B1 is USER0 bits 15-8, or byte 1, and so on).

---

**NOTE:** When using the Ethernet update, the bootloader can not update itself since there is no mechanism in BOOTP to distinguish between a firmware image and a bootloader image. Therefore, the bootloader does not know if a given image is a new bootloader or a new firmware image. It assumes that all images provided are firmware images.

---

The following IETF specifications define the protocols used by the Ethernet update mechanism:

RFC951 (<http://tools.ietf.org/html/rfc951.html>) defines the bootstrap protocol.

RFC1350 (<http://tools.ietf.org/html/rfc1350.html>) defines the trivial file transfer protocol.

## 5 CAN Update

When performing a CAN update the bootloader calls `ConfigureCAN()` to configure the CAN controller and prepare the bootloader to update the firmware. The CAN update mechanism allows the bootloader to be entered from a functioning CAN application as well from startup when no application has been downloaded to the microcontroller. The bootloader provides the main routine for performing the CAN update in the `UpdaterCAN()` function which is used in both cases.

When the device enters the bootloader from a running CAN network, the bootloader does not reconfigure the CAN clocks or bit timing and will assume that they have been configured as expected by the firmware update device. The bootloader assumes that the application has taken the device off of the CAN network by putting it in "Init mode" but left the CAN bit timings untouched. When the bootloader is run without an application, it is necessary to configure the CAN bit rate using the default CAN clocking which uses the `#define` values `CAN_BIT_RATE` and `CRYSTAL_FREQ`. These settings must be identical to the CAN bit rate settings used by the application. When the last data is received, the CAN update application must issue an explicit `LM_API_UPD_RESET` command to restart the device.

### 5.1 CAN Bus Clocking

There are two global definitions that are required to configure the CAN bootloader to meet the application's timing requirements. They are both used to determine how the CAN bit rate is configured based on the clock provided to the CAN controller as well as the desired bit rate. The `CAN_BIT_RATE` value sets the transfer rate for data on the CAN bus in bits per second. The other value, `CRYSTAL_FREQ`, is used to set the input frequency to the CAN controller.

## 5.2 CAN Commands

The CAN firmware update provides a short list of commands that are used during the firmware update operation. The definitions for these commands are provided in the file `bl_can.h`. The description of each of these commands is covered in the rest of this section.

### LM\_API\_UPD\_PING

This command is used to receive an acknowledge command from the bootloader indicating that communication has been established. This command has no data. If the device is present it will respond with a `LM_API_UPD_PING` back to the CAN update application.

### LM\_API\_UPD\_DOWNLOAD

This command sets the base address for the download as well as the size of the data to write to the device. This command should be followed by a series of `LM_API_UPD_SEND_DATA` that send the actual image to be programmed to the device. The command consists of two 32-bit values that are transferred LSB first. The first 32-bit value is the address to start programming data into, while the second is the 32-bit size of the data that will be sent. This command also triggers an erasure of the full application area in the flash. This flash erase operation causes the command to take longer to send the `LM_API_UPD_ACK` in response to the command which should be taken into account by the CAN update application.

The format of the command is as follows:

```
unsigned char ucData[8];
ucData[0] = Download Address [7:0];
ucData[1] = Download Address [15:8];
ucData[2] = Download Address [23:16];
ucData[3] = Download Address [31:24];
ucData[4] = Download Size [7:0];
ucData[5] = Download Size [15:8];
ucData[6] = Download Size [23:16];
ucData[7] = Download Size [31:24];
```

### LM\_API\_UPD\_SEND\_DATA

This command should only follow a `LM_API_UPD_DOWNLOAD` command or another `LM_API_UPD_SEND_DATA` command when more data is needed. Consecutive send data commands automatically increment the address and continue programming from the previous location. The transfer size is limited to 8 bytes at a time based on the maximum size of an individual CAN transmission. The command terminates programming when the number of bytes indicated by the `LM_API_UPD_DOWNLOAD` command have been received. The CAN bootloader will send a `LM_API_UPD_ACK` in response to each send data command to allow the CAN update application to throttle the data going to the device and not overrun the bootloader with data.

The format of the command is as follows:

```
unsigned char ucData[8];
ucData[0] = Data[0];
ucData[1] = Data[1];
ucData[2] = Data[2];
ucData[3] = Data[3];
ucData[4] = Data[4];
ucData[5] = Data[5];
ucData[6] = Data[6];
ucData[7] = Data[7];
```

### LM\_API\_UPD\_RESET

This command is used to tell the CAN bootloader to reset the microcontroller. This is used after downloading a new image to the microcontroller to cause the new application or the new bootloader to start from a reset. The normal boot sequence occurs and the image runs as if from a hardware reset. It can also be used to reset the bootloader if a critical error occurs and the CAN update application needs to restart communication with the bootloader.

## 6 USB Device (DFU) Update

When performing a USB update, the bootloader calls `ConfigureUSB()` to configure the USB controller and prepare the bootloader to update the firmware. The USB update mechanism allows the bootloader to be entered from a functioning application as well as from startup when no application has been downloaded to the microcontroller. The bootloader provides the main routine for performing the USB update in the `UpdaterUSB()` function which is used in both cases.

When the USB bootloader is invoked from a running application, the bootloader will reconfigure the USB controller to publish the required descriptor set for a Device Firmware Upgrade (DFU) class device. If the main application had previously been offering any USB device class, it must remove the device from the bus by calling `USBDevDisconnect()` prior to entering the bootloader.

The USB bootloader also assumes that the main application is using the PLL as the source of the system clock.

The USB bootloader allows a USB host to upgrade the firmware on a USB device. To make use of it, therefore, the board running the bootloader must be capable of acting as a USB device. Firmware upgrade of boards which operate solely as USB hosts is not supported by the USB DFU class or the USB bootloader.

### 6.1 USB Device Firmware Upgrade Overview

The USB bootloader enumerates as a Device Firmware Upgrade (DFU) class device. This standard device class specifies a set of class-specific requests and a state machine that can be used to download and flash firmware images to a device and, optionally, upload the existing firmware image to the USB host. The full specification for the device class can be downloaded from the USB Implementer's Forum website at [http://www.usb.org/developers/docs/devclass\\_docs/DFU\\_1.1.pdf](http://www.usb.org/developers/docs/devclass_docs/DFU_1.1.pdf).

All communication with the DFU device takes place using the USB control endpoint, endpoint 0. The device publishes a standard device descriptor with vendor, product and device revisions as specified in the `bl_config.h` header file used to build the bootloader binary. It also publishes a single configuration descriptor and a single interface descriptor where the interface class of `0xFE` indicates an application-specific class and the subclass of `0x01` indicates "Device Firmware Upgrade". Attached to the interface descriptor is a DFU Functional Descriptor which provides information to the host on DFU-specific device capabilities such as whether the device can perform upload operations and what the maximum transfer size for upload and download operations is.

DFU functions are initiated by means of a set of class-specific requests. Each request, which follows the standard USB request format, performs some operation and moves the DFU device between a series of well-defined states. Additional requests allow the host to query the current state of the device to determine whether, for example, it is ready to receive the next block of download data.

A DFU device may operation in one of two modes: "Run Time" mode or "DFU" mode. In "Run Time" mode, the device publishes the DFU interface and functional descriptors alongside any other descriptors that the device requires for normal operation. It does not, however, need to respond to any DFU class-specific requests other than `DFU_DETACH` which indicates that it should switch to "DFU" mode.

In "DFU" mode, the device supports all DFU functionality and can perform upload and download operations as specified in its DFU functional descriptor.

The USB bootloader supports only "DFU" mode operation. If an main application wishes to publish DFU descriptors and respond to the `DFU_DETACH` request, it can cause a switch to "DFU" mode on receiving a `DFU_DETACH` request by removing itself from the USB bus using a call to `USBDevDisconnect()` before transferring control to the USB bootloader by making a call through the `SVC` vector in the usual manner.

### 6.1.1 DFU Requests

Requests supported by the USB bootloader are:

#### **DFU\_DNLOAD**

This OUT request is used to send a block of binary data to the device. The DFU class specification does not define the content and format of the binary data but typically this is either binary data to be written to some position in the device's flash memory or a device-specific command. The request payload size is constrained by the maximum packet size specified in the DFU functional descriptor. In this implementation, that maximum is set to 1024 bytes.

After sending a DFU\_DNLOAD request, the host must poll the device status and wait until the state reverts to DNLOAD\_IDLE before sending another request. If the host wishes to indicate that it has finished sending download data, it sends a DFU\_DNLOAD request with a payload length of 0.

#### **DFU\_UPLOAD**

This IN request is used to request a block of binary data from the device. The data returned is device-specific but will typically be the contents of a region of the device's flash memory or a device-specific response to a command previously sent through a DFU\_DNLOAD request. As with DFU\_DNLOAD, the maximum amount of data that can be requested is governed by the maximum packet size specified in the DFU functional descriptor, here 1024 bytes.

#### **DFU\_GETSTATUS**

This IN request allows the host to query the current status of the DFU device. It is typically used during download operations to determine when it is safe to send the next block of data. Depending upon the state of the DFU device, this request may also trigger a state change. During download, for example, the device enters DNLOAD\_SYNC state after receiving a block of data and remains there until the data has been processed and a DFU\_GETSTATUS request is received at which point the state changes to DNLOAD\_IDLE.

#### **DFU\_CLRSTATUS**

This request is used to reset any error condition reported by the DFU device. If an error is reported through the response to a DFU\_GETSTATUS request, that error condition is cleared when this request is received and the device returns to IDLE state.

#### **DFU\_GETSTATE**

This IN request is used to query the current state of the device without triggering any state change. The single byte of data returned indicates the current state of the DFU device.

#### **DFU\_ABORT**

This request is used to cancel any partially complete upload or download operation and return the device to IDLE state in preparation for some other request.

### 6.1.2 DFU States

During operation, the DFU device transitions between a set of class-defined states. The host must query the current state to determine when a new operation can be performed or to determine the cause of any errors reported. These states are:

#### **IDLE**

The IDLE state indicates to the host that the DFU device is ready to start an upload or download operation.

#### **DNLOAD\_SYNC**

After each DFU\_DNLOAD request is received, DNLOAD\_SYNC state is entered. This state remains in effect until the host issues a DFU\_GETSTATUS request at which point the state will change to DNLOAD\_IDLE if the last download operation has completed or DNBUSY otherwise.

#### **DNLOAD\_IDLE**

This state indicates that a download operation is in progress and that the device is ready to receive another DFU\_DNLOAD request with the next block of data.

### **DNBUSY**

This state is reported if a DFU\_GETSTATUS request is received while a block of downloaded data is still being processed. The host must refrain from issuing another DFU\_GETSTATUS request for a time specified in the structure returned following the request. After this time, the device state reverts to DNLOAD\_SYNC.

To reduce the USB bootloader image size, this state is not supported. Instead of reporting DNBUSY, the USB bootloader remains in state DNLOAD\_SYNC until the previous data has been processed then transitions to DNLOAD\_IDLE on receipt of the first DFU\_GETSTATUS request following completion of block programming.

### **MANIFEST\_SYNC**

The end of a download operation is signaled by the host sending a DFU\_DNLOAD request with a 0 length payload. When this request is received, the DFU device transitions from state DNLOAD\_IDLE to MANIFEST\_SYNC. This state indicates that the complete firmware image has been received and the device is waiting for a DFU\_GETSTATUS request before finalizing programming of the image.

The USB bootloader programs downloaded blocks as they are received so does not need to perform any additional processing after all blocks have been received. It also reports that it is "manifest tolerant", indicating to the host that it will still respond to requests after a download has completed. As a result, the device will transition from this state to IDLE when the DFU\_GETSTATUS request is received.

### **MANIFEST**

This state indicates to the host that the device is programming a previously received firmware image and is entered on receipt of a DFU\_GETSTATUS request while a device that is not manifest tolerant is in MANIFEST\_SYNC state.

This state is not used by the USB bootloader since it is manifest tolerant and reverts to IDLE state after completion of a download.

### **MANIFEST\_WAIT\_RESET**

This state indicates that a device which is not manifest tolerant has finished writing a downloaded image and is waiting for a USB reset to signal it to boot the new firmware.

This state is not used by the USB bootloader since it is manifest tolerant and reverts to IDLE state after completion of a download.

### **UPLOAD\_IDLE**

Following receipt of a DFU\_UPLOAD request, the device remains in this state until it receives another DFU\_UPLOAD request asking for less than the maximum transfer size of data. This indicates that the upload is complete and the device will transition back to IDLE state.

### **ERROR**

The ERROR state is entered when some error occurs.

The device remains in this state until the host sends a DFU\_CLRSTATUS request at which point the state reverts to IDLE and that error code, which is reported in the data returned in response to DFU\_GETSTATUS, is cleared.

### 6.1.3 Typical Firmware Download Sequence

Figure 15 shows a typical firmware image download sequence from the perspective of the host application.

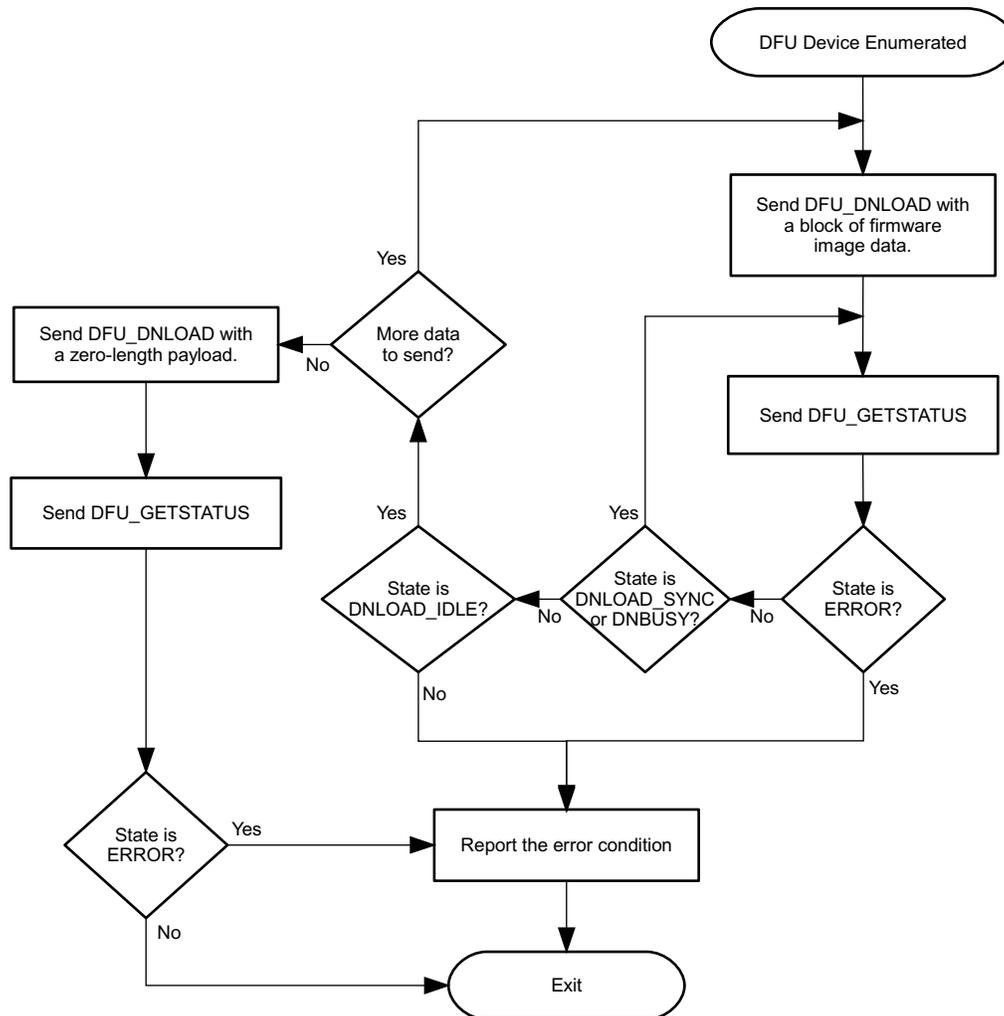


Figure 15. Firmware Download Sequence

## 6.2 USB Download Commands

The DFU class specification provides the framework necessary to download and upload firmware files to the USB device but does not specify the actual format of the binary data that is transferred. As a result, different device implementations have used different methods to perform operations which are not defined in the standard such as:

- Setting the address that a downloaded binary should be flashed to.
- Setting the address and size of the area of flash whose contents are to be uploaded.
- Erasing sections of the flash.
- Querying the size of flash and writeable area addresses.

The USB bootloader implementation employs a small set of commands which can be sent to the DFU device through a DFU\_DNLOAD request when the device is in IDLE state. Each command takes the form of an 8 byte structure which defines the operation to carry out and provides any required additional parameters.

To ensure that a host application which does not have explicit support for device-specific commands can still be used to download binary firmware images to the device, the protocol is defined such that only a single 8 byte header structure need be placed at the start of the binary image being downloaded. This header and the DFU-defined suffix structure can both be added using the supplied “dfuwrap” command-line application, hence providing a single binary that can be sent to a device running the MSP432E4 USB bootloader using a standard sequence of DFU\_DNLOAD requests with no other embedded commands or device-specific operations required. An application which does understand the MSP432E4-specific commands may make use of them to offer additional functionality that would not otherwise be available.

### 6.2.1 Querying Command Support

Since the device-specific commands defined here are sent to the DFU device in DFU\_DNLOAD requests, the possibility exists that sending them to a device which does not understand the protocol could result in corruption of that device’s firmware. To guard against this possibility, the MSP432E4 USB bootloader supports an additional USB request which is used to query the device capabilities and allow a host to determine whether or not the device supports the MSP432E4 commands. A device which does not support the commands will either stall the request or return unexpected data.

To determine whether a target DFU device supports the device-specific DFU commands, send the following IN request to the DFU interface:

**Table 3. Request to Determine Device-Specific Commands**

bmRequest-Type	bRequest	wValue	wIndex	wLength	Data
10100001b	0x42	0x23	Interface	4	Protocol Info

Where the protocol information returned is a 4 byte structure, the first 2 bytes of which are 0x4D and 0x4C, and where the second group of 2 bytes indicates the protocol version supported, currently 0x01 and 0x00 respectively.

### 6.2.2 Download Command Definitions

The following commands may be sent to the USB bootloader as the first 8 bytes of the payload to a DFU\_DNLOAD request. The bootloader will expect any DFU\_DNLOAD request received while

in IDLE state to contain a command header but does not look for command unless the state is IDLE. This allows an application which is unaware of the command header to download a DFU-wrapped binary image using a standard sequence of multiple DFU\_DNLOAD and DFU\_GETSTATUS requests without the need to insert additional command headers during the download.

The commands defined here and their parameter block structures can be found in header file usbdfu.h. In all cases where multi-byte numbers are specified, the numbers are stored in little-endian format with the least significant byte in the lowest addressed location. The following definitions specify the command byte ordering unambiguously but care must be taken to ensure correct byte swapping if using the command structure types defined in usbdfu.h on big-endian systems.

#### DFU\_CMD\_PROG

This command is used to provide the USB bootloader with the address at which the next download should be written and the total length of the firmware image which is to follow. This structure forms the header that is written to the DFU-wrapped file generated by the dfuwrap tool.

The start address is provided in terms of 1024 byte flash blocks. To convert a byte address to a block address, merely divide by 1024. The start address must always be on a 1024 byte boundary.

This command may be followed by up to 1016 bytes of firmware image data, this number being the maximum transfer size minus the 8 bytes of the command structure.

The format of the command is as follows:

```
unsigned char ucData[8];
ucData[0] = DFU_CMD_PROG (0x01)
ucData[1] = Reserved - set to 0x00
ucData[2] = Start Block Number [7:0];
```

```
ucData[3] = Start Block Number [15:8];
ucData[4] = Image Size [7:0];
ucData[5] = Image Size [15:8];
ucData[6] = Image Size [23:16];
ucData[7] = Image Size [31:24];
```

### DFU\_CMD\_READ

This command is used to set the address range whose content will be returned on subsequent DFU\_UPLOAD requests from the host.

The start address is provided in terms of 1024 byte flash blocks. To convert a byte address to a block address, merely divide by 1024. The start address must always be on a 1024 byte boundary.

To read back a the contents of a region of flash, the host should send a DFU\_DNLOAD request with command DFU\_CMD\_READ, start address set to the 1KB block start address and length set to the number of bytes to read. The host should then send one or more DFU\_UPLOAD requests to receive the current flash contents from the configured addresses. Data returned will include an 8-byte DFU\_CMD\_PROG prefix structure unless the prefix has been disabled by sending a DFU\_CMD\_BIN command with the bBinary parameter set to 1. The host should, therefore, be prepared to read 8 bytes more than the length specified in the READ command if the prefix is enabled.

By default, the 8-byte prefix is enabled for all upload operations. This is required by the DFU class specification which states that uploaded images must be formatted to allow them to be directly downloaded back to the device at a later time.

The format of the command is as follows:

```
unsigned char ucData[8];
ucData[0] = DFU_CMD_READ (0x02)
ucData[1] = Reserved - set to 0x00
ucData[2] = Start Block Number [7:0];
ucData[3] = Start Block Number [15:8];
ucData[4] = Image Size [7:0];
ucData[5] = Image Size [15:8];
ucData[6] = Image Size [23:16];
ucData[7] = Image Size [31:24];
```

### DFU\_CMD\_CHECK

This command is used to check a region of flash to ensure that it is completely erased.

The start address is provided in terms of 1024 byte flash blocks. To convert a byte address to a block address, merely divide by 1024. The start address must always be on a 1024 byte boundary. The length must also be a multiple of 4.

To check that a region of flash is erased, the DFU\_CMD\_CHECK command should be sent with the required start address and region length set then the host should issue a DFU\_GETSTATUS request. If the erase check was successful, the returned bStatus value will be OK (0x00), otherwise it will be errCheckErased (0x05).

The format of the command is as follows:

```
unsigned char ucData[8];
ucData[0] = DFU_CMD_CHECK (0x03)
ucData[1] = Reserved - set to 0x00
ucData[2] = Start Block Number [7:0];
ucData[3] = Start Block Number [15:8];
ucData[4] = Region Size [7:0];
ucData[5] = Region Size [15:8];
ucData[6] = Region Size [23:16];
ucData[7] = Region Size [31:24];
```

### DFU\_CMD\_ERASE

This command is used to erase a region of flash.

The start address is provided in terms of 1024 byte flash blocks. To convert a byte address to a block address, merely divide by 1024. The start address must always be on a 1024 byte boundary. The length must also be a multiple of 4. The size of the region to erase is expressed in terms of flash blocks. The block size can be determined using the DFU\_CMD\_INFO command.

The format of the command is as follows:

```
unsigned char ucData[8];
ucData[0] = DFU_CMD_ERASE (0x04)
ucData[1] = Reserved - set to 0x00
ucData[2] = Start Block Number [7:0];
ucData[3] = Start Block Number [15:8];
ucData[4] = Number of Blocks [7:0];
ucData[5] = Number of Blocks [15:8];
ucData[6] = Reserved - set to 0x00
ucData[7] = Reserved - set to 0x00
```

### DFU\_CMD\_INFO

This command is used to query information relating to the target device and programmable region of flash. The device information structure, `tDFUDeviceInfo`, is returned on the next DFU\_UPLOAD request following this command.

The format of the command is as follows:

```
unsigned char ucData[8];
ucData[0] = DFU_CMD_INFO (0x05)
ucData[1] = Reserved - set to 0x00
ucData[2] = Reserved - set to 0x00
ucData[3] = Reserved - set to 0x00
ucData[4] = Reserved - set to 0x00
ucData[5] = Reserved - set to 0x00
ucData[6] = Reserved - set to 0x00
ucData[7] = Reserved - set to 0x00
/*****
//
// Payload returned in response to the DFU_CMD_INFO command.
//
// This structure is returned in response to the first DFU_UPLOAD
// request following a DFU_CMD_INFO command. Note that byte ordering
// of multi-byte fields is little-endian.
//
/*****
typedef struct
{
//
// The size of a flash block in bytes.
//
unsigned short usFlashBlockSize;
//
// The number of blocks of flash in the device. Total
// flash size is usNumFlashBlocks * usFlashBlockSize.
//
unsigned short usNumFlashBlocks;
//
// Information on the part number, family, version and
// package as read from SYSCTL register DID1.
//
unsigned long ulPartInfo;
//
// Information on the part class and revision as read
// from SYSCTL DID0.
//
```

```

unsigned long ulClassInfo;
//
// Address 1 byte above the highest location the boot
// loader can access.
//
unsigned long ulFlashTop;
//
// Lowest address the boot loader can write or erase.
//
unsigned long ulAppStartAddr;
}
PACKED tDFUDeviceInfo;

```

### DFU\_CMD\_BIN

By default, data returned in response to a DFU\_UPLOAD request includes an 8 byte DFU\_CMD\_PROG prefix structure. This ensures that an uploaded image can be directly downloaded again without the need to further wrap it but, in cases where pure binary data is required, can be awkward. The DFU\_CMD\_BIN command allows the upload prefix to be disabled or enabled under host control.

The format of the command is as follows:

```

unsigned char ucData[8];
ucData[0] = DFU_CMD_BIN (0x06)
ucData[1] = 0x01 to disable upload prefix, 0x00 to enable
ucData[2] = Reserved - set to 0x00
ucData[3] = Reserved - set to 0x00
ucData[4] = Reserved - set to 0x00
ucData[5] = Reserved - set to 0x00
ucData[6] = Reserved - set to 0x00
ucData[7] = Reserved - set to 0x00

```

### DFU\_CMD\_RESET

This command may be sent to the USB bootloader to cause it to perform a soft reset of the board. This will reboot the system and, assuming that the main application image is present, run the main application. A reboot will also take place if a firmware download operation completes and the host issues a USB reset to the DFU device.

The format of the command is as follows:

```

unsigned char ucData[8];
ucData[0] = DFU_CMD_RESET (0x07)
ucData[1] = Reserved - set to 0x00
ucData[2] = Reserved - set to 0x00
ucData[3] = Reserved - set to 0x00
ucData[4] = Reserved - set to 0x00
ucData[5] = Reserved - set to 0x00
ucData[6] = Reserved - set to 0x00
ucData[7] = Reserved - set to 0x00

```

## 7 Customization

The bootloader allows for customization of its features as well as the interfaces used to update the microcontroller. This allows the bootloader to include only the features that are needed by the application. There are two types of features that can be customized. The first type are the features that are conditionally included or excluded at compile time. The second type of customizations are more involved and include customizing the actual code that is used by the bootloader.

The bootloader can be modified to have any functionality. As an example, the main loop can be completely replaced to use a different set of commands and still use the packet and transport functions from the bootloader. The method of detecting a forced update can be modified to suit the needs of the application when toggling a GPIO to detect an update request may not be the best solution. If the bootloader's packet format does not meet the needs of the application, it can be completely replaced by replacing ReceivePacket(), SendPacket(), AckPacket(), and NakPacket().

The bootloader also provides a method to add a new serial transmission interface beyond the UART, SSI, and I<sup>2</sup>C that are provided by the bootloader. For the packet functions to use the new transport functions, the `SendData()`, `ReceiveData()`, and `FlushData()` defines need to be modified to use the new functions. For example:

```
#ifndef FOO_ENABLE_UPDATE
#define SendData      FooSend
#define FlushData     FooFlush
#define ReceiveData   FooReceive
#endif
```

would use the functions for the hypothetical Foo peripheral.

The combination of these customizable features provides a framework that allows the bootloader to define whatever protocol it needs, or use any port that it needs to perform updates of the microcontroller.

## 8 Configuration

The following defines are used to configure the operation of the bootloader. These defines are located in the `bl_config.h` header file, for which there is a template (`bl_config.h.tmpl`) provided with the bootloader.

### CRYSTAL\_FREQ

This defines the crystal frequency used by the microcontroller running the bootloader. If this is unknown at the time of production, then use the `UART_AUTOBAUD` feature to properly configure the UART.

This value must be defined if using the UART for the update and not using the auto-baud feature, and when using CAN, USB, or Ethernet for the update.

If using CAN, only a 1 MHz, 2 MHz, 4 MHz, 5 MHz, 6 MHz, 8 MHz, 10 MHz, 12 MHz, or 16 MHz crystal can be used.

### APP\_START\_ADDRESS

The starting address of the application. This must be a multiple of page boundary. A vector table is expected at this location, and the perceived validity of the vector table (stack located in SRAM, reset vector located in flash) is used as an indication of the validity of the application image.

This value must be defined.

The flash image of the bootloader must not be larger than this value.

### VTABLE\_START\_ADDRESS

The address at which the application locates its exception vector table. This must be a multiple of page boundary. Typically, an application will start with its vector table and this value should be set to `APP_START_ADDRESS`. This option is provided to cater for applications which run from external memory which may not be accessible by the NVIC (the vector table offset register is only 30 bits long).

This value must be defined.

### FLASH\_PAGE\_SIZE

The size of a single, erasable page in the flash. This must be a power of 2. The default value is set to 16384 for MSP432E4 MCUs, and this value should be overridden only if configuring a bootloader to access external flash devices with a page size different from this.

This value must be defined.

### FLASH\_RSVD\_SPACE

The amount of space at the end of flash to reserve. This must be a multiple of page boundary. This reserved space is not erased when the application is updated, providing nonvolatile storage that can be used for parameters.

### STACK\_SIZE

The number of words of stack space to reserve for the bootloader.

This value must be defined.

## **BUFFER\_SIZE**

The number of words in the data buffer used for receiving packets. This value must be at least 3. If using auto-baud on the UART, this must be at least 20. The maximum usable value is 65 (larger values will result in unused space in the buffer).

This value must be defined if updating through UART, SSI, or I<sup>2</sup>C .

## **ENABLE\_BL\_UPDATE**

Enables updates to the bootloader. Updating the bootloader is an unsafe operation since it is not fully fault tolerant (losing power to the device partway through could result in the bootloader no longer being present in flash). The bootloader can not be updated through the Ethernet port.

## **CHECK\_CRC**

Enables runtime CRC checking in the bootloader. If this label is not defined, the bootloader will transfer control to a main application image if the initial stack pointer and reset vector of the image appear to be valid in flash. No additional checking is performed. When this label is defined, however, the firmware image's CRC32 value is checked against a value stored in a header at the top of the image's vector table and the firmware is only booted if the calculated CRC matches the value in the header. If the header is absent or the calculated CRC does not match the expected value, the bootloader retains control and waits for a new firmware image to be downloaded. As a special case to aid debugging, the image will be booted if the header is found and the length field is set to 0xFFFFFFFF, regardless of the value of the CRC32 field. This debug behavior can be disabled by also defining ENFORCE\_CRC.

To use the CHECK\_CRC option, firmware images must be built with an 8 word header appended at the top of the interrupt vector table. This header must have the first 4 words populated as follow:

- Word 0: 0xFF01FF02
- Word 1: 0xFF02FF03
- Word 2: Length of the firmware binary image in bytes.
- Word 3: CRC32 calculated over entire image except for the 4 bytes occupied by this word. The CRC calculation includes the header marker words and the length word.

The remaining 4 words in the header are reserved for future extensions and should be set to 0xFFFFFFFF.

This header should be added by reserving 8 additional words above all the required entries in the project's vector table (typically in the startup C or assembler file depending upon the toolchain in use) and initializing the first two to be the required marker words and the remainder to value 0xFFFFFFFF. The length and CRC32 value can be inserted by passing the firmware binary as the input to the binpack.exe tool found in the tools directory of your SimpleLink MSP432E SDK release.

## **ENFORCE\_CRC**

This definition may be used in conjunction with CHECK\_CRC and will remove the debug behaviour described above. When both CHECK\_CRC and ENFORCE\_CRC are defined, the bootloader will only boot a main firmware image if it contains a valid image information header at the top of the vector table and if the CRC32 calculated for the image matches the CRC32 in the image information header.

## **FLASH\_CODE\_PROTECTION**

This definition will cause the the bootloader to erase the entire flash on updates to the bootloader or to erase the entire application area when the application is updated. This erases any unused sections in the flash before the firmware is updated.

## **ENABLE\_DECRYPTION**

Enables the call to decrypt the downloaded data before writing it into flash. The decryption routine is empty in the reference bootloader source, which simply provides a placeholder for adding an actual decryption algorithm. Although this option is retained for backwards compatibility, it is recommended that a decryption function be specified using the newer hook function mechanism and BL\_DECRYPT\_FN\_HOOK instead.

## **ENABLE\_MOSCFAIL\_HANDLER**

Enables support for the MOSCFAIL handler in the NMI interrupt.

**FLASH\_PATCH\_COMPATIBLE**

Enables support for the code to cooperate with the flash patch that is preloaded into the flash of certain MSP432E4 devices. This support should only be enabled for devices that require it.

Enabling this feature causes the bootloader to be loaded at 0x1000, despite the fact that debuggers attempt to load it from 0x0. Therefore, LM Flash Programmer must be used to program the bootloader into flash when using this feature.

**ENABLE\_UPDATE\_CHECK**

Enables the pin-based forced update check. When enabled, the bootloader will go into update mode instead of calling the application if a pin is read at a particular polarity, forcing an update operation. In either case, the application is still able to return control to the bootloader in order to start an update. For applications which need to perform more complex checking than is possible using a single GPIO, a hook function may be provided using BL\_CHECK\_UPDATE\_FN\_HOOK instead.

**FORCED\_UPDATE\_PERIPH**

The GPIO module to enable in order to check for a forced update. This will be one of the SYSCTL\_RCGC2\_GPIOx values, where x is replaced with the port name (such as B). The value of x should match the value of x for FORCED\_UPDATE\_PORT.

This value must be defined if ENABLE\_UPDATE\_CHECK is defined.

**FORCED\_UPDATE\_PORT**

The GPIO port to check for a forced update. This will be one of the GPIO\_PORTx\_BASE values, where x is replaced with the port name (such as B). The value of x should match the value of x for FORCED\_UPDATE\_PERIPH.

This value must be defined if ENABLE\_UPDATE\_CHECK is defined.

**FORCED\_UPDATE\_PIN**

The pin to check for a forced update. This is a value between 0 and 7.

This value must be defined if ENABLE\_UPDATE\_CHECK is defined.

**FORCED\_UPDATE\_POLARITY**

The polarity of the GPIO pin that results in a forced update. This value should be 0 if the pin should be low and 1 if the pin should be high.

This value must be defined if ENABLE\_UPDATE\_CHECK is defined.

**FORCED\_UPDATE\_WPU FORCED\_UPDATE\_WPD**

This enables a weak pullup or pulldown for the GPIO pin used in a forced update. Only one of FORCED\_UPDATE\_WPU or FORCED\_UPDATE\_WPD should be defined, or neither if a weak pullup or pulldown is not required.

**FORCED\_UPDATE\_KEY**

This enables the use of the GPIO\_LOCK mechanism for configuration of protected GPIO pins (for example JTAG pins). If this value is not defined, the locking mechanism is not be used. The only legal values for this feature are GPIO\_LOCK\_KEY for Fury devices and GPIO\_LOCK\_KEY\_DD for all other devices except Sandstorm devices, which do not support this feature.

**UART\_ENABLE\_UPDATE**

Selects the UART as the port for communicating with the bootloader.

**UART\_AUTOBAUD**

Enables automatic baud rate detection. This can be used if the crystal frequency is unknown, or if operation at different baud rates is desired.

This value or UART\_FIXED\_BAUDRATE must be defined if UART\_ENABLE\_UPDATE is defined.

**UART\_FIXED\_BAUDRATE**

Selects the baud rate to be used for the UART.

This value or UART\_AUTOBAUD must be defined if UART\_ENABLE\_UPDATE is defined.

**UART\_CLOCK\_ENABLE**

Selects the clock enable for the UART peripheral module for the bootloader.

This value must be defined as SYSCTL\_RCGCUART\_Rx, where x is replaced with the module instance number (such as 0).

**UARTx\_BASE**

Selects the base address of the UART peripheral module for the bootloader.

This value must be defined as UARTx\_BASE, where x is replaced with the module instance number (such as 0).

**UART\_RXPIN\_CLOCK\_ENABLE**

Selects the clock enable for the GPIO corresponding to UART RX pin.

This value must be defined as SYSCTL\_RCGCGPIO\_Rx, where x is replaced with the GPIO module instance number (such as 0 for GPIO Port A).

**UART\_RXPIN\_BASE**

Selects the base address for the GPIO corresponding to UART RX pin.

This value must be defined as GPIO\_PORTx\_BASE, where x is replaced with the GPIO module port name (such as A for GPIO Port A).

**UART\_RXPIN\_PCTL**

Selects the port control value for the GPIO corresponding to UART RX pin.

This value must be defined as per the pin mux value given in the signal description table for the RX pin of the UART module.

**UART\_RXPIN\_POS**

Selects the pin number for the GPIO corresponding to UART RX pin.

This value must be between 0 and 7 as per the pin assignment value given in the signal description table for the RX pin of the UART module.

**UART\_TXPIN\_CLOCK\_ENABLE**

Selects the clock enable for the GPIO corresponding to the UART TX pin.

This value must be defined as SYSCTL\_RCGCGPIO\_Rx, where x is replaced with the GPIO module instance number (such as 0 for GPIO Port A).

**UART\_TXPIN\_BASE**

Selects the base address for the GPIO corresponding to the UART TX pin.

This value must be defined as GPIO\_PORTx\_BASE, where x is replaced with the GPIO module port name (such as A for GPIO Port A).

**UART\_TXPIN\_PCTL**

Selects the port control value for the GPIO corresponding to UART TX pin.

This value must be defined as per the pin mux value given in the signal description table for the UART module's TX pin.

**UART\_TXPIN\_POS**

Selects the pin number for the GPIO corresponding to UART TX pin.

This value must be between 0 and 7 as per the pin assignment value given in the signal description table for the TX pin of the UART module.

**SSI\_ENABLE\_UPDATE**

Selects the SSI port as the port for communicating with the bootloader.

**SSI\_CLOCK\_ENABLE**

Selects the clock enable for the SSI peripheral module for the bootloader.

This value must be defined as SYSCTL\_RCGCSSI\_Rx, where x is replaced with the module instance number (such as 0).

**SSIx\_BASE**

Selects the base address of the SSI peripheral module for the bootloader.

This value must be defined as SSIx\_BASE, where x is replaced with the module instance number (such as 0).

**SSI\_CLKPIN\_CLOCK\_ENABLE**

Selects the clock enable for the GPIO corresponding to SSI CLK pin.

This value must be defined as SYSCTL\_RCGCGPIO\_Rx, where x is replaced with the GPIO module instance number (such as 0 for GPIO Port A).

**SSI\_CLKPIN\_BASE**

Selects the base address for the GPIO corresponding to SSI CLK pin.

This value must be defined as GPIO\_PORTx\_BASE, where x is replaced with the GPIO module port name (such as A for GPIO Port B).

**SSI\_CLKPIN\_PCTL**

Selects the port control value for the GPIO corresponding to the SSI CLK pin.

This value must be defined as per the pin mux value given in the signal description table for the SSI module's CLK pin.

**SSI\_CLKPIN\_POS**

Selects the pin number for the GPIO corresponding to SSI CLK pin.

This value must be between 0 and 7 as per the pin assignment value given in the signal description table for the SSI module's CLK pin.

**SSI\_FSSPIN\_CLOCK\_ENABLE**

Selects the clock enable for the GPIO corresponding to SSI FSS pin.

This value must be defined as SYSCTL\_RCGCGPIO\_Rx, where x is replaced with the GPIO module instance number (such as 0 for GPIO Port A).

**SSI\_FSSPIN\_BASE**

Selects the base address for the GPIO corresponding to SSI FSS pin.

This value must be defined as GPIO\_PORTx\_BASE, where x is replaced with the GPIO module port name (such as A for GPIO Port A).

**SSI\_FSSPIN\_PCTL**

Selects the port control value for the GPIO corresponding to the SSI FSS pin.

This value must be defined as per the pin mux value given in the signal description table for the SSI FSS pin.

**SSI\_FSSPIN\_POS**

Selects the pin number for the GPIO corresponding to SSI FSS pin.

This value must be between 0 and 7 as per the pin assignment value given in the signal description table for the SSI FSS pin.

**SSI\_MISOPIN\_CLOCK\_ENABLE**

Selects the clock enable for the GPIO corresponding to SSI MISO pin.

This value must be defined as SYSCTL\_RCGCGPIO\_Rx, where x is replaced with the GPIO module instance number (such as 0 for GPIO Port A).

**SSI\_MISOPIN\_BASE**

Selects the base address for the GPIO corresponding to SSI MISO pin.

This value must be defined as GPIO\_PORTx\_BASE, where x is replaced with the GPIO module port name (such as A for GPIO Port A).

**SSI\_MISOPIN\_PCTL**

Selects the port control value for the GPIO corresponding to the SSI MISO pin.

This value must be defined as per the pin mux value given in the signal description table for the SSI MISO pin.

**SSI\_MISOPIN\_POS**

Selects the pin number for the GPIO corresponding to SSI MISO pin.

This value must be between 0 and 7 as per the pin assignment value given in the signal description table for the SSI module's MISO pin.

**SSI\_MOSIPIN\_CLOCK\_ENABLE**

Selects the clock enable for the GPIO corresponding to the SSI MOSI pin.

This value must be defined as SYSCTL\_RCGCGPIO\_Rx, where x is replaced with the GPIO module instance number (such as 0 for GPIO Port A).

**SSI\_MOSIPIN\_BASE**

Selects the base address for the GPIO corresponding to SSI MOSI pin.

This value must be defined as GPIO\_PORTx\_BASE, where x is replaced with the GPIO module port name (such as A for GPIO Port A).

**SSI\_MOSIPIN\_PCTL**

Selects the port control value for the GPIO corresponding to the SSI MOSI pin.

This value must be defined as per the pin mux value given in the signal description table for the SSI MOSI pin.

**SSI\_MOSIPIN\_POS**

Selects the pin number for the GPIO corresponding to SSI MOSI pin.

This value must be between 0 and 7 as per the pin assignment value given in the signal description table for the SSI MOSI pin.

**I2C\_ENABLE\_UPDATE**

Selects the I<sup>2</sup>C port as the port for communicating with the bootloader.

**I2C\_SLAVE\_ADDR**

Specifies the I<sup>2</sup>C address of the bootloader.

This value must be defined if I2C\_ENABLE\_UPDATE is defined.

**I2C\_CLOCK\_ENABLE**

Selects the clock enable for the I<sup>2</sup>C peripheral module for the bootloader.

This value must be defined as SYSCTL\_RCGCI2C\_Rx, where x is replaced with the module instance number (such as 0).

**I2Cx\_BASE**

Selects the base address of the I<sup>2</sup>C peripheral module for the bootloader.

This value must be defined as I2Cx\_BASE, where x is replaced with the module instance number (such as 0).

**I2C\_SCLPIN\_CLOCK\_ENABLE**

Selects the clock enable for the GPIO corresponding to I<sup>2</sup>C SCL pin.

This value must be defined as SYSCTL\_RCGCGPIO\_Rx, where x is replaced with the GPIO module instance number (such as 1 for GPIO Port B).

**I2C\_SCLPIN\_BASE**

Selects the base address for the GPIO corresponding to I<sup>2</sup>C SCL pin.

This value must be defined as GPIO\_PORTx\_BASE, where x is replaced with the GPIO module port name (such as B for GPIO Port B).

**I2C\_SCLPIN\_PCTL**

Selects the port control value for the GPIO corresponding to the I<sup>2</sup>C SCL pin.

This value must be defined as per the pin mux value given in the signal description table for the I<sup>2</sup>C SCL pin.

**I2C\_SCLPIN\_POS**

Selects the pin number for the GPIO corresponding to I<sup>2</sup>C SCL pin.

This value must be between 0 and 7 as per the pin assignment value given in the signal description table for the I<sup>2</sup>C SCL pin.

**I2C\_SDAPIN\_CLOCK\_ENABLE**

Selects the clock enable for the GPIO corresponding to the I<sup>2</sup>C SDA pin.

This value must be defined as SYSCTL\_RCGCGPIO\_Rx, where x is replaced with the GPIO module instance number (such as 1 for GPIO Port B).

**I2C\_SDAPIN\_BASE**

Selects the base address for the GPIO corresponding to I<sup>2</sup>C SDA pin.

This value must be defined as GPIO\_PORTx\_BASE, where x is replaced with the GPIO module port name (such as B for GPIO Port B).

**I2C\_SDAPIN\_PCTL**

Selects the port control value for the GPIO corresponding to the I<sup>2</sup>C SDA pin.

This value must be defined as per the pin mux value given in the signal description table for the I<sup>2</sup>C module's SDA pin.

**I2C\_SDAPIN\_POS**

Selects the pin number for the GPIO corresponding to I<sup>2</sup>C SDA pin.

This value must be between 0 and 7 as per the pin assignment value given in the signal description table for the I<sup>2</sup>C SDA pin.

**ENET\_ENABLE\_UPDATE**

Selects Ethernet update through the BOOTP/TFTP protocol.

**ENET\_ENABLE\_LEDS**

Enables the use of the Ethernet status LED outputs to indicate traffic and connection status.

**ENET\_MAC\_ADDRx (x = 0 to 5)**

Specifies the hard coded MAC address for the Ethernet interface. There are six individual values (ENET\_MAC\_ADDR0 to ENET\_MAC\_ADDR5) that provide the six bytes of the MAC address, where ENET\_MAC\_ADDR0 to ENET\_MAC\_ADDR2 provide the organizationally unique identifier (OUI) and ENET\_MAC\_ADDR3 to ENET\_MAC\_ADDR5 provide the extension identifier. If these values are not provided, the MAC address is extracted from the user registers.

**ENET\_BOOTP\_SERVER**

Specifies the name of the BOOTP server from which to request information. The use of this specifier allows a board-specific BOOTP server to co-exist on a network with the DHCP server that may be part of the network infrastructure. The BOOTP server provided by Texas Instruments requires that this be set to "msp432e4".

**USB\_ENABLE\_UPDATE**

Selects USB update through Device Firmware Update class.

**USB\_VENDOR\_ID**

The USB vendor ID published by the DFU device. This value is the TI MSP432E4 vendor ID. Change this to the vendor ID you have been assigned by the USB-IF.

This value must be defined if USB\_ENABLE\_UPDATE is defined.

**USB\_PRODUCT\_ID**

The USB device ID published by the DFU device. If you are using your own vendor ID, chose a device ID that is different from the ID you use in nonupdate operation. If you have sublicensed TI's vendor ID, you must use an assigned product ID here.

This value must be defined if USB\_ENABLE\_UPDATE is defined.

**USB\_DEVICE\_ID**

Selects the BCD USB device release number published in the device descriptor.

This value must be defined if USB\_ENABLE\_UPDATE is defined.

**USB\_MAX\_POWER**

Sets the maximum power consumption that the DFU device will report to the USB host in the configuration descriptor. Units are milliamps.

This value must be defined if USB\_ENABLE\_UPDATE is defined.

**USB\_BUS\_POWERED**

Specifies whether the DFU device reports to the host that it is self-powered (defined as 0) or bus-powered (defined as 1).

**USB\_HAS\_MUX**

Specifies whether the target board uses a multiplexer to select between USB host and device modes.

**USB\_MUX\_PERIPH**

Specifies the GPIO peripheral containing the pin which is used to select between USB host and device modes. The value is of the form SYSCTL\_RCGC2\_GPIOx, where GPIOx represents the required GPIO port.

This value must be defined if USB\_HAS\_MUX is defined.

**USB\_MUX\_PORT**

Specifies the GPIO port containing the pin which is used to select between USB host and device modes. The value is of the form GPIO\_PORTx\_BASE, where PORTx represents the required GPIO port.

This value must be defined if USB\_HAS\_MUX is defined.

**USB\_MUX\_PIN**

Specifies the GPIO pin number used to select between USB host and device modes. Valid values are 0 to 7.

This value must be defined if USB\_HAS\_MUX is defined.

**USB\_MUX\_DEVICE**

Specifies the state of the GPIO pin required to select USB device-mode operation. Valid values are 0 (low) or 1 (high).

This value must be defined if USB\_HAS\_MUX is defined.

**USB\_VBUS\_CONFIG**

Specifies whether the target board requires configuration of the pin used for VBUS.

**USB\_VBUS\_PERIPH**

Specifies the GPIO peripheral containing the pin which is used for VBUS. The value is of the form SYSCTL\_RCGCGPIO\_Rx, where the Rx represents the required GPIO port.

This value must be defined if USB\_VBUS\_CONFIG is defined.

**USB\_VBUS\_PORT**

Specifies the GPIO port containing the pin which is used for VBUS. The value is of the form GPIO\_PORTx\_BASE, where PORTx represents the required GPIO port.

This value must be defined if USB\_VBUS\_CONFIG is defined.

**USB\_VBUS\_PIN**

Specifies the GPIO pin number used for VBUS. Valid values are 0 to 7.

This value must be defined if USB\_VBUS\_CONFIG is defined.

**USB\_ID\_CONFIG**

Specifies whether the target board requires configuration of the pin used for ID.

**USB\_ID\_PERIPH**

Specifies the GPIO peripheral containing the pin which is used for ID. The value is of the form SYSCTL\_RCGCGPIO\_Rx, where the Rx represents the required GPIO port.

This value must be defined if USB\_ID\_CONFIG is defined.

**USB\_ID\_PORT**

Specifies the GPIO port containing the pin which is used for ID. The value is of the form GPIO\_PORTx\_BASE, where PORTx represents the required GPIO port.

This value must be defined if USB\_ID\_CONFIG is defined.

**USB\_ID\_PIN**

Specifies the GPIO pin number used for ID. Valid values are 0 to 7.

This value must be defined if USB\_ID\_CONFIG is defined.

**USB\_DP\_CONFIG**

Specifies whether the target board requires configuration of the pin used for DP.

**USB\_DP\_PERIPH**

Specifies the GPIO peripheral containing the pin which is used for DP. The value is of the form SYSCTL\_RCGCGPIO\_Rx, where the Rx represents the required GPIO port. This value must be defined if USB\_DP\_CONFIG is defined.

**USB\_DP\_PORT**

Specifies the GPIO port containing the pin which is used for DP. The value is of the form GPIO\_PORTx\_BASE, where PORTx represents the required GPIO port.

This value must be defined if USB\_DP\_CONFIG is defined.

**USB\_DP\_PIN**

Specifies the GPIO pin number used for DP. Valid values are 0 to 7. This value must be defined if USB\_DP\_CONFIG is defined.

**USB\_DM\_CONFIG**

Specifies whether the target board requires configuration of the pin used for DM.

**USB\_DM\_PERIPH**

Specifies the GPIO peripheral containing the pin which is used for DM. The value is of the form SYSCTL\_RCGCGPIO\_Rx, where the Rx represents the required GPIO port.

This value must be defined if USB\_DM\_CONFIG is defined.

#### **USB\_DM\_PORT**

Specifies the GPIO port containing the pin which is used for DM. The value is of the form GPIO\_PORTx\_BASE, where PORTx represents the required GPIO port.

This value must be defined if USB\_DM\_CONFIG is defined.

#### **USB\_DM\_PIN**

Specifies the GPIO pin number used for DM. Valid values are 0 to 7.

This value must be defined if USB\_DM\_CONFIG is defined.

#### **CAN\_ENABLE\_UPDATE**

Selects an update through the CAN port.

#### **CAN\_REQUIRES\_PLL**

Indicates that the CAN peripheral operates from a fixed divide of the PLL output, meaning that the PLL must be enabled. This is required by some older devices, but must not be used on newer devices. Consult the device data sheet to determine if the CAN peripheral operates from a fixed 8-MHz clock derived from the PLL (meaning this option must be used) or if it operates from the system clock (meaning this option must not be used).

#### **CAN\_UART\_BRIDGE**

Enables the UART to CAN bridging for use when the CAN port is selected for communicating with the bootloader.

#### **CAN\_RX\_PERIPH**

The GPIO module to enable in order to configure the CAN0 Rx pin. This will be one of the SYSCTL\_RCGC2\_GPIOx values, where x is replaced with the port name (such as B). The value of x should match the value of x for CAN\_RX\_PORT. This value must be defined if CAN\_ENABLE\_UPDATE is defined.

#### **CAN\_RX\_PORT**

The GPIO port used to configure the CAN0 Rx pin. This will be one of the GPIO\_PORTx\_BASE values, where x is replaced with the port name (such as B). The value of x should match the value of x for CAN\_RX\_PERIPH.

This value must be defined if CAN\_ENABLE\_UPDATE is defined.

#### **CAN\_RX\_PIN**

The GPIO pin that is shared with the CAN0 Rx pin. This is a value between 0 and 7.

This value must be defined if CAN\_ENABLE\_UPDATE is defined.

#### **CAN\_TX\_PERIPH**

The GPIO module to enable in order to configure the CAN0

Tx pin. This will be one of the SYSCTL\_RCGC2\_GPIOx values, where x is replaced with the port name (such as B). The value of x should match the value of x for CAN\_TX\_PORT.

This value must be defined if CAN\_ENABLE\_UPDATE is defined.

#### **CAN\_TX\_PORT**

The GPIO port used to configure the CAN0 Tx pin. This will be one of the GPIO\_PORTx\_BASE values, where x is replaced with the port name (such as B). The value of x should match the value of x for CAN\_TX\_PERIPH.

This value must be defined if CAN\_ENABLE\_UPDATE is defined.

#### **CAN\_TX\_PIN**

The GPIO pin that is shared with the CAN0 Tx pin. This is a value of 0 to 7.

This value must be defined if CAN\_ENABLE\_UPDATE is defined.

### **CAN\_BIT\_RATE**

The bit rate used on the CAN bus. This must be one of 20000, 50000, 125000, 250000, 500000, or 1000000. The CAN bit rate must be less than or equal to the crystal frequency divided by 8 (CRYSTAL\_FREQ / 8).

This value must be defined if CAN\_ENABLE\_UPDATE is defined.

### **BL\_HW\_INIT\_FN\_HOOK**

Performs application-specific low-level hardware initialization on system reset. If hooked, this function is called immediately after the bootloader code relocation completes. An application may perform any required low-level hardware initialization during this function. The system clock has not been set when this function is called. Initialization that assumes the system clock is set may be performed in the BL\_INIT\_FN\_HOOK function instead.

### **BL\_INIT\_FN\_HOOK**

Performs application-specific initialization on system reset.

If hooked, this function will be called during bootloader initialization to perform any board- or application-specific initialization which is required. The function is called following a reset immediately after the selected bootloader peripheral has been configured and the system clock has been set.

### **BL\_REINIT\_FN\_HOOK**

Performs application-specific reinitialization on bootloader entry through SVC. If hooked, this function will be called during bootloader reinitialization to perform any board- or application-specific initialization which is required. The function is called following bootloader entry from an application, after any system clock rate adjustments have been made.

### **BL\_START\_FN\_HOOK**

Informs an application that a download is starting. If hooked, this function will be called when a firmware download is about to begin. The function is called after the first data packet of the download is received but before it has been written to flash.

### **BL\_PROGRESS\_FN\_HOOK**

Informs an application of download progress. If hooked, this function will be called periodically during a firmware download to provide progress information. The function is called after each data packet is received from the host. Parameters provide the number of bytes of data received and, in cases other than Ethernet update, the expected total number of bytes in the download (the TFTP protocol used by the Ethernet bootloader does not send the final image size before the download starts so in this case the ulTotal parameter is set to 0 to indicate that the size is unknown).

### **BL\_END\_FN\_HOOK**

Informs an application that a download has completed. If hooked, this function will be called when a firmware download has just completed. The function is called after the final data packet of the download has been written to flash.

### **BL\_DECRYPT\_FN\_HOOK**

Allows an application to perform in-place data decryption during download. If hooked, this function will be called to perform in-place decryption of each data packet received during a firmware download.

This value takes precedence over ENABLE\_DECRYPTION. If both are defined, the hook function defined using BL\_DECRYPT\_FN\_HOOK is called rather than the previously-defined DecryptData() stub function.

### **BL\_CHECK\_UPDATE\_FN\_HOOK**

Allows an application to force a new firmware download. If hooked, this function will be called during bootloader initialization to determine whether a firmware update should be performed regardless of whether a valid main code image is already present. If the function returns 0, the existing main code image is booted (if present), otherwise the bootloader will wait for a new firmware image to be downloaded.

This value takes precedence over `ENABLE_UPDATE_CHECK` if both are defined. If you wish to perform a GPIO check in addition to any other update check processing required, the GPIO code must be included within the hook function itself.

#### **BL\_FLASH\_ERASE\_FN\_HOOK**

Allows an application to replace the flash block erase function. If hooked, this function will be called whenever a block of flash is to be erased. The function must erase the block and wait until the operation has completed. The size of the block which will be erased is defined by `FLASH_BLOCK_SIZE`.

#### **BL\_FLASH\_PROGRAM\_FN\_HOOK**

Allows an application to replace the flash programming function. If hooked, this function will be called to program the flash with firmware image data received during download operations. The function must program the supplied data and wait until the operation has completed.

#### **BL\_FLASH\_CL\_ERR\_FN\_HOOK**

Allows an application to replace the flash error clear function.

If hooked, this function must clear any flash error indicators and prepare to detect access violations that may occur in a future erase or program operations.

#### **BL\_FLASH\_ERROR\_FN\_HOOK**

Reports whether or not a flash access violation error has occurred. If hooked, this function will be called after flash erase or program operations. The return code indicates whether or not an access violation error occurred since the last call to the function defined by `BL_FLASH_CL_ERR_FN_HOOK`, with 0 indicating no errors and nonzero indicating an error.

#### **BL\_FLASH\_SIZE\_FN\_HOOK**

Reports the total size of the device flash. If hooked, this function will be called to determine the size of the supported flash device. The return code is the number of bytes of flash in the device. This does not take into account any reserved space defined through the `FLASH_RSVD_SPACE` value.

#### **BL\_FLASH\_END\_FN\_HOOK**

Reports the address of the first byte after the end of the device flash. If hooked, this function will be called to determine the address of the end of valid flash. This does not take into account any reserved space defined through the `FLASH_RSVD_SPACE` value.

#### **BL\_FLASH\_AD\_CHECK\_FN\_HOOK**

Checks whether the start address and size of an image are valid. If hooked, this function will be called when a new firmware download is about to start. Parameters provided are the requested start address for the new download and, when using protocols which transmit the image length in advance, the size of the image that is to be downloaded. The return code will be nonzero to indicate that the start address is valid and the image will fit in the available space, or 0 if either the address is invalid or the image is too large for the device.

## **9 Source Details**

### **9.1 Autobaud Functions**

The following functions are provided in `bl_autobaud.c` and are used to perform autobauding on the UART interface.

```
void GPIOIntHandler (void)
```

```
int UARTAutoBaud (uint32_t *pui32Ratio)
```

### 9.1.1 GPIOIntHandler

Handles the UART Rx GPIO interrupt.

#### Prototype

```
void GPIOIntHandler(void)
```

#### Description

When an edge is detected on the UART Rx pin, this function is called to save the time of the edge. These times are later used to determine the ratio of the UART baud rate to the processor clock rate.

#### Returns

None

### 9.1.2 UARTAutoBaud

Performs auto-baud on the UART port.

#### Prototype

```
int UARTAutoBaud(uint32_t *pui32Ratio)
```

#### Parameters

*pui32Ratio* is the ratio of the processor's crystal frequency to the baud rate being used by the UART port for communications.

#### Description

This function attempts to synchronize to the updater program that is trying to communicate with the bootloader. The UART port is monitored for edges using interrupts. When enough edges are detected, the bootloader determines the ratio of baud rate and crystal frequency needed to program the UART.

#### Returns

Returns a value of 0 to indicate that this call successfully synchronized with the other device communicating over the UART, and a negative value to indicate that this function did not successfully synchronize with the other UART device.

## 9.2 CAN Functions

The following functions are provided in `bl_can.c` and are used to perform an update over the CAN interface.

```
void AppUpdaterCAN (void) void ConfigureCAN (void) void UpdaterCAN (void)
```

### 9.2.1 AppUpdaterCAN

This is the application entry point to the CAN updater.

#### Prototype

```
void AppUpdaterCAN(void)
```

#### Description

This function should only be entered from a running application and not when running the bootloader with no application present.

#### Returns

None

### 9.2.2 ConfigureCAN

Generic configuration is handled in this function.

#### Prototype

void ConfigureCAN(void)

#### Description

This function is called by the start up code to perform any configuration necessary before calling the update routine.

#### Returns

None

### 9.2.3 UpdaterCAN

This is the main routine for handling updating over CAN.

#### Prototype

void UpdaterCAN(void)

#### Description

This function accepts bootloader commands over CAN to perform a firmware update over the CAN bus. This function assumes that the CAN bus timing and message objects have been configured elsewhere.

#### Returns

None

### 9.3 Decryption Functions

The following functions are provided in `bl_decrypt.c` and are used to optionally decrypt the firmware data as it is received.

void DecryptData (uint8\_t \*pui8Buffer, uint32\_t ui32Size)

#### 9.3.1 DecryptData

Performs an in-place decryption of downloaded data.

#### Prototype

void DecryptData(uint8\_t \*pui8Buffer, uint32\_t ui32Size)

#### Parameters

*pui8Buffer* is the buffer that holds the data to decrypt.

*ui32Size* is the size, in bytes, of the buffer that was passed in through the *pui8Buffer* parameter.

#### Description

This function is a stub that could provide in-place decryption of the data that is being downloaded to the device.

#### Returns

None

### 9.4 Ethernet Functions

The following functions are provided in `bl_enet.c` and are used to perform an update over the Ethernet interface.

### 9.5 File System Functions

The following functions are provided in `bl_fs.c` and are used to provide very basic support for reading from a FAT file system.

## 9.6 I<sup>2</sup>C Functions

The following functions are provided in `bl_i2c.c` and are used to communicate over the I<sup>2</sup>C interface.

`void I2CFlush (void)`

`void I2CReceive (uint8_t *pui8Data, uint32_t ui32Size)`

`void I2CSend (const uint8_t *pui8Data, uint32_t ui32Size)`

### 9.6.1 I2CFlush

Waits until all data has been transmitted by the I<sup>2</sup>C port.

**Prototype** `void I2CFlush(void)`

#### Description

This function waits until all data written to the I<sup>2</sup>C port has been read by the master.

#### Returns

None

### 9.6.2 I2CReceive

Receives data over the I<sup>2</sup>C port.

#### Prototype

`void I2CReceive(uint8_t *pui8Data, uint32_t ui32Size)`

#### Parameters

*pui8Data* is the buffer to read data into from the I<sup>2</sup>C port.

*ui32Size* is the number of bytes provided in the *pui8Data* buffer that should be written with data from the I<sup>2</sup>C port.

#### Description

This function reads back *ui32Size* bytes of data from the I<sup>2</sup>C port, into the buffer that is pointed to by *pui8Data*. This function does not return until *ui32Size* number of bytes have been received. This function waits until the I<sup>2</sup>C slave port has been properly addressed by the I<sup>2</sup>C master before reading the first byte of data from the I<sup>2</sup>C port.

#### Returns

None

### 9.6.3 I2CSend

Sends data over the I<sup>2</sup>C port.

#### Prototype

`void I2CSend(const uint8_t *pui8Data, uint32_t ui32Size)`

#### Parameters

*pui8Data* is the buffer containing the data to write out to the I<sup>2</sup>C port.

*ui32Size* is the number of bytes provided in *pui8Data* buffer that will be written out to the I<sup>2</sup>C port.

#### Description

This function sends *ui32Size* bytes of data from the buffer pointed to by *pui8Data* through the I<sup>2</sup>C port. The function waits until the I<sup>2</sup>C slave port has been properly addressed by the I<sup>2</sup>C master device before sending the first byte.

#### Returns

None

## 9.7 Main Functions

The following functions are provided in bl\_main.c and comprise the main bootloader application.

void ConfigureDevice (void)

void Updater (void)

### 9.7.1 ConfigureDevice

Configures the microcontroller.

#### Prototype

void ConfigureDevice(void)

#### Description

This function configures the peripherals and GPIOs of the microcontroller, preparing it for use by the bootloader. The interface that has been selected as the update port is configured, and auto-baud is performed if required.

#### Returns

None

### 9.7.2 Updater

This function performs the update on the selected port.

**Prototype** void Updater(void)

#### Description

This function is called directly by the bootloader or it is called as a result of an update request from the application.

#### Returns

Never returns.

## 9.8 Packet Handling Functions

The following functions are provided in bl\_packet.c and are used to process the data packets in the custom serial protocol.

void AckPacket (void)

uint32\_t CheckSum (const uint8\_t \*pui8Data, uint32\_t ui32Size)

void NakPacket (void)

int ReceivePacket (uint8\_t \*pui8Data, uint32\_t \*pui32Size)

int SendPacket (uint8\_t \*pui8Data, uint32\_t ui32Size)

### 9.8.1 AckPacket

Sends an Acknowledge packet.

#### Prototype

void AckPacket(void)

#### Description

This function is called to acknowledge that a packet has been received by the microcontroller.

#### Returns

None

### 9.8.2 CheckSum

Calculates an 8-bit checksum

#### Prototype

```
uint32_t CheckSum(const uint8_t *pui8Data, uint32_t ui32Size)
```

#### Parameters

*pui8Data* is a pointer to an array of 8-bit data of size *ui32Size*.

*ui32Size* is the size of the array that will run through the checksum algorithm.

#### Description

This function simply calculates an 8-bit checksum on the data passed in.

#### Returns

Returns the calculated checksum.

### 9.8.3 NakPacket

Sends a no-acknowledge packet.

#### Prototype

```
void NakPacket(void)
```

#### Description

This function is called when an invalid packet has been received by the microcontroller, indicating that it should be retransmitted.

#### Returns

None

### 9.8.4 ReceivePacket

Receives a data packet.

#### Prototype

```
int ReceivePacket(uint8_t *pui8Data, uint32_t *pui32Size)
```

#### Parameters

*pui8Data* is the location to store the data that is sent to the bootloader.

*pui32Size* is the number of bytes returned in the *pui8Data* buffer that was provided.

#### Description

This function receives a packet of data from specified transfer function.

#### Returns

Returns zero to indicate success while any nonzero value indicates a failure.

### 9.8.5 SendPacket

Sends a data packet.

#### Prototype

```
int
```

```
SendPacket(uint8_t *pui8Data, uint32_t ui32Size)
```

#### Parameters

*pui8Data* is the location of the data to be sent.

*ui32Size* is the number of bytes to send.

### Description

This function sends the data provided in the *pui8Data* parameter in the packet format used by the bootloader. The caller only needs to specify the buffer with the data that needs to be transferred. This function addresses all other packet formatting issues.

### Returns

Returns zero to indicate success while any nonzero value indicates a failure.

## 9.9 SSI Functions

The following functions are provided in `bl_ssi.c` and are used to communicate over the SSI interface.

```
void SSIFlush (void)
```

```
void SSIReceive (uint8_t *pui8Data, uint32_t ui32Size)
```

```
void SSISend (const uint8_t *pui8Data, uint32_t ui32Size)
```

### 9.9.1 SSIFlush

Waits until all data has been transmitted by the SSI port.

#### Prototype

```
void SSIFlush(void)
```

#### Description

This function waits until all data written to the SSI port has been read by the master.

#### Returns

None

### 9.9.2 SSIReceive

Receives data from the SSI port in slave mode.

#### Prototype

```
void SSIReceive(uint8_t *pui8Data, uint32_t ui32Size)
```

#### Parameters

*pui8Data* is the location to store the data received from the SSI port.

*ui32Size* is the number of bytes of data to receive.

#### Description

This function receives data from the SSI port in slave mode. The function does not return until *ui32Size* number of bytes have been received.

#### Returns

None

### 9.9.3 SSISend

Sends data through the SSI port in slave mode.

#### Prototype

```
void SSISend(const uint8_t *pui8Data, uint32_t ui32Size)
```

#### Parameters

*pui8Data* is the location of the data to send through the SSI port.

*ui32Size* is the number of bytes of data to send.

#### Description

This function sends data through the SSI port in slave mode. This function does not return until all bytes are sent.

#### Returns

None

### 9.10 UART Functions

The following functions are provided in `bl_uart.c` and are used to communicate over the UART interface.

`void UARTFlush (void)`

`void UARTReceive (uint8_t *pui8Data, uint32_t ui32Size)`

`void UARTSend (const uint8_t *pui8Data, uint32_t ui32Size)`

#### 9.10.1 UARTFlush

Waits until all data has been transmitted by the UART port.

##### Prototype

`void UARTFlush(void)`

##### Description

This function waits until all data written to the UART port has been transmitted.

##### Returns

None

#### 9.10.2 UARTReceive

Receives data over the UART port.

##### Prototype

`void`

`UARTReceive(uint8_t *pui8Data, uint32_t ui32Size)`

##### Parameters

*pui8Data* is the buffer to read data into from the UART port.

*ui32Size* is the number of bytes provided in the *pui8Data* buffer that should be written with data from the UART port.

##### Description

This function reads back *ui32Size* bytes of data from the UART port, into the buffer that is pointed to by *pui8Data*. This function does not return until *ui32Size* number of bytes have been received.

##### Returns

None

#### 9.10.3 UARTSend

Sends data over the UART port.

##### Prototype

```
void UARTSend(const uint8_t *pui8Data, uint32_t ui32Size)
```

#### Parameters

*pui8Data* is the buffer containing the data to write out to the UART port.

*ui32Size* is the number of bytes provided in *pui8Data* buffer that will be written out to the UART port.

#### Description

This function sends *ui32Size* bytes of data from the buffer pointed to by *pui8Data* through the UART port.

#### Returns

None

### 9.11 Update Check Functions

The following functions are provided in `bl_check.c` and are used to check if a firmware update is required.

```
uint32_t CheckForceUpdate (void)
```

#### 9.11.1 CheckForceUpdate

Checks if an update is needed or is being requested.

#### Prototype

```
uint32_t CheckForceUpdate(void)
```

#### Description

This function detects if an update is being requested or if there is no valid code presently located on the microcontroller. This is used to tell whether or not to enter update mode.

#### Returns

Returns a nonzero value if an update is needed or is being requested and zero otherwise.

### 9.12 USB Device Functions

The following data structures and functions are provided in `bl_usb.c` and `bl_usbfuncs.c` and are used to communicate over the USB interface.

#### Data Structures

```
tConfigDescriptor
```

```
tString0Descriptor
```

```
tStringDescriptor
```

```
tUSBRequest
```

#### Functions

```
void AppUpdaterUSB (void)
```

```
void ConfigureUSB (void)
```

```
void ConfigureUSBInterface (void)
```

```
void HandleRequests (tUSBRequest *psUSBRequest)
```

```
bool ProcessDFUDnloadCommand (tDFUDownloadHeader *psCmd, uint32_t ui32Size)
```

```
void UpdaterUSB (void)
```

```
void USBBLInit (void)
```

```
void USBBLSendDataEP0 (uint8_t *pui8Data, uint32_t ui32Size)
```

```
void USBBLStallEP0 (void)
```

```
void USBConfigurePins (void)
```

### 9.12.1 tConfigDescriptor

#### Definition

```
typedef struct
{
    uint8_t bLength;
    uint8_t bDescriptorType;
    uint16_t wTotalLength;
    uint8_t bNumInterfaces;
    uint8_t bConfigurationValue;
    uint8_t iConfiguration;
    uint8_t bmAttributes;
    uint8_t bMaxPower;
}
tConfigDescriptor
```

#### Members

*bLength* The length of this descriptor in bytes. All configuration descriptors are 9 bytes long.

*bDescriptorType* The type of the descriptor. For a configuration descriptor, this is USB\_DTYPE\_CONFIGURATION (2).

*wTotalLength* The total length of data returned for this configuration. This includes the combined length of all descriptors (configuration, interface, endpoint and class- or vendor- specific) returned for this configuration.

*bNumInterfaces* The number of interface supported by this configuration.

*bConfigurationValue* The value used as an argument to the SetConfiguration standard request to select this configuration.

*iConfiguration* The index of a string descriptor describing this configuration.

*bmAttributes* Attributes of this configuration.

*bMaxPower* The maximum power consumption of the USB device from the bus in this configuration when the device is fully operational. This is expressed in units of 2 mA so, for example, 100 represents 200 mA.

#### Description

This structure describes the USB configuration descriptor as defined in USB 2.0 specification Section 9.6.3. This structure also applies to the USB other speed configuration descriptor defined in Section 9.6.4.

### 9.12.2 tString0Descriptor

#### Definition

```
typedef struct
{
    uint8_t bLength;
    uint8_t bDescriptorType;
    uint16_t wLANGID[1];
}
tString0Descriptor
```

#### Members

*bLength* The length of this descriptor in bytes. This value will vary depending upon the number of language codes provided in the descriptor.

*bDescriptorType* The type of the descriptor. For a string descriptor, this is USB\_DTYPE\_STRING (3).

*wLANGID* The language code (LANGID) for the first supported language. This descriptor may support multiple languages, in which case, the number of elements in the wLANGID array increases, and bLength is updated accordingly.

## Description

This structure describes the USB string descriptor for index 0 as defined in USB 2.0 specification section 9.6.7. The number of language IDs is variable and can be determined by examining `bLength`. The number of language IDs present in the descriptor is given by  $((bLength - 2) / 2)$ .

### 9.12.3 tStringDescriptor

#### Definition

```
typedef struct
{
    uint8_t bLength;
    uint8_t bDescriptorType;
    uint8_t bString;
}
tStringDescriptor
```

#### Members

*bLength* The length of this descriptor in bytes. This value is 2 greater than the number of bytes comprising the UNICODE string that the descriptor contains.

*bDescriptorType* The type of the descriptor. For a string descriptor, this is `USB_DTYPE_STRING` (3).

*bString* The first byte of the UNICODE string. This string is not NULL terminated. Its length (in bytes) can be computed by subtracting 2 from the value in the `bLength` field.

#### Description

This structure describes the USB string descriptor for all string indexes other than 0 as defined in USB 2.0 specification Section 9.6.7.

### 9.12.4 tUSBRequest

#### Definition

```
typedef struct
{
    uint8_t bmRequestType;
    uint8_t bRequest;
    uint16_t wValue;
    uint16_t wIndex;
    uint16_t wLength;
}
tUSBRequest
```

#### Members

*bmRequestType* Determines the type and direction of the request.

*bRequest* Identifies the specific request being made.

*wValue* Word-sized field that varies according to the request.

*wIndex* Word-sized field that varies according to the request; typically used to pass an index or offset.

*wLength* The number of bytes to transfer if there is a data stage to the request.

#### Description

The standard USB request header as defined in section 9.3 of the USB 2.0 specification.

### 9.12.5 AppUpdaterUSB

This is the application entry point to the USB updater.

#### Prototype

```
void AppUpdaterUSB(void)
```

**Description**

This function should only be entered from a running application and not when running the bootloader with no application present. If the calling application supports any USB device function, it must remove itself from the USB bus prior to calling this function. This function assumes that the calling application has already configured the system clock to run from the PLL.

**Returns**

None

**9.12.6 ConfigureUSB**

Generic configuration is handled in this function.

**Prototype**

```
void ConfigureUSB(void)
```

**Description**

This function is called by the start up code to perform any configuration necessary before calling the update routine. It is responsible for setting the system clock to the expected rate and setting flash programming parameters prior to calling `ConfigureUSBInterface()` to set up the USB hardware and place the DFU device on the bus.

**Returns**

None

**9.12.7 ConfigureUSBInterface**

Configure the USB controller and place the DFU device on the bus.

**Prototype**

```
void ConfigureUSBInterface(void)
```

**Description**

This function configures the USB controller for DFU device operation, initializes the state machines required to control the firmware update and places the device on the bus in preparation for requests from the host. It is assumed that the main system clock has been configured at this point.

**Returns**

None

**9.12.8 HandleRequests**

Handle USB requests sent to the DFU device.

**Prototype**

```
void HandleRequests(tUSBRequest *psUSBRequest)
```

**Parameters**

*psUSBRequest* is a pointer to the USB request that the device has been sent.

**Description**

This function is called to handle all nonstandard requests received by the device. This will include all the DFU endpoint 0 commands along with the MSP432E4-specific request we use to query whether the device supports our flavor of the DFU binary format. Incoming DFU requests are processed by request handlers specific to the particular state of the DFU connection. This state machine implementation is chosen to keep the software as close as possible to the USB DFU class documentation.

**Returns**

None

### 9.12.9 ProcessDFUDnloadCommand

Process device-specific commands passed through DFU download requests.

#### Prototype

```
bool ProcessDFUDnloadCommand(tDFUDownloadHeader *psCmd, uint32_t ui32Size)
```

#### Parameters

*psCmd* is a pointer to the first byte of the DFU\_DNLOAD payload that is expected to hold a command.

*ui32Size* is the number of bytes of data pointed to by *psCmd*. This function is called when a DFU download command is received while in **STATE\_IDLE**. New downloads are assumed to contain a prefix structure containing one of several MSP432E4-specific commands and this function is responsible for parsing the download data and processing whichever command is contained within it.

#### Returns

Returns true on success or false on failure.

### 9.12.10 UpdaterUSB

This is the main routine for handling updating over USB.

This function forms the main loop of the USB DFU updater. It polls for commands sent from the USB request handlers and is responsible for erasing flash blocks, programming data into erased blocks and resetting the device.

#### Prototype

```
void UpdaterUSB (void)
```

#### Returns

None

### 9.12.11 USBBLInit

Initialize the bootloader USB functions.

This function initializes the bootloader USB functions and places the DFU device onto the USB bus.

#### Prototype

```
void USBBLInit (void)
```

#### Returns

None

### 9.12.12 USBBLSendDataEP0

This function requests transfer of data to the host on endpoint zero.

#### Prototype

```
void USBBLSendDataEP0 (uint8_t* pui8Data, uint32_t ui32Size)
```

#### Parameters

*pui8Data* is a pointer to the buffer to send through endpoint zero.

*ui32Size* is the amount of data to send in bytes.

#### Description

This function handles sending data to the host when a custom command is issued or nonstandard descriptor has been requested on endpoint zero.

#### Returns

None

### 9.12.13 USBBLStallEP0

This function generates a stall condition on endpoint zero.

**Prototype**

```
void USBBLStallEP0(void)
```

**Description**

This function is typically called to signal an error condition to the host when an unsupported request is received by the device. It should be called from within the callback itself (in interrupt context) and not deferred until later since it affects the operation of the endpoint zero state machine.

**Returns**

None

### 9.12.14 USBConfigurePins

Initialize the pins used by USB functions.

**Prototype** void USBConfigurePins(void)

**Description**

This function configures the pins for USB functions depending on defines from the bl\_config.h file.

**Returns**

None

## 10 References

The following related documents, software and tools are available for user reference

1. [SimpleLink™ MSP432E4 Software Development Kit](#)
2. [MSP432 MCU BSL Scripter](#)
3. [BSL Rocket Programmer](#)

## Revision History

NOTE: Page numbers for previous revisions may differ from page numbers in the current version.

Changes from October 28, 2017 to November 16, 2017	Page
• Changed document title .....	2
• Corrected the link to the USB DFU specification in <a href="#">Section 6.1</a> , <i>USB Device Firmware Upgrade Overview</i> .....	18

## IMPORTANT NOTICE FOR TI DESIGN INFORMATION AND RESOURCES

Texas Instruments Incorporated ("TI") technical, application or other design advice, services or information, including, but not limited to, reference designs and materials relating to evaluation modules, (collectively, "TI Resources") are intended to assist designers who are developing applications that incorporate TI products; by downloading, accessing or using any particular TI Resource in any way, you (individually or, if you are acting on behalf of a company, your company) agree to use it solely for this purpose and subject to the terms of this Notice.

TI's provision of TI Resources does not expand or otherwise alter TI's applicable published warranties or warranty disclaimers for TI products, and no additional obligations or liabilities arise from TI providing such TI Resources. TI reserves the right to make corrections, enhancements, improvements and other changes to its TI Resources.

You understand and agree that you remain responsible for using your independent analysis, evaluation and judgment in designing your applications and that you have full and exclusive responsibility to assure the safety of your applications and compliance of your applications (and of all TI products used in or for your applications) with all applicable regulations, laws and other applicable requirements. You represent that, with respect to your applications, you have all the necessary expertise to create and implement safeguards that (1) anticipate dangerous consequences of failures, (2) monitor failures and their consequences, and (3) lessen the likelihood of failures that might cause harm and take appropriate actions. You agree that prior to using or distributing any applications that include TI products, you will thoroughly test such applications and the functionality of such TI products as used in such applications. TI has not conducted any testing other than that specifically described in the published documentation for a particular TI Resource.

You are authorized to use, copy and modify any individual TI Resource only in connection with the development of applications that include the TI product(s) identified in such TI Resource. NO OTHER LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE TO ANY OTHER TI INTELLECTUAL PROPERTY RIGHT, AND NO LICENSE TO ANY TECHNOLOGY OR INTELLECTUAL PROPERTY RIGHT OF TI OR ANY THIRD PARTY IS GRANTED HEREIN, including but not limited to any patent right, copyright, mask work right, or other intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information regarding or referencing third-party products or services does not constitute a license to use such products or services, or a warranty or endorsement thereof. Use of TI Resources may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

TI RESOURCES ARE PROVIDED "AS IS" AND WITH ALL FAULTS. TI DISCLAIMS ALL OTHER WARRANTIES OR REPRESENTATIONS, EXPRESS OR IMPLIED, REGARDING TI RESOURCES OR USE THEREOF, INCLUDING BUT NOT LIMITED TO ACCURACY OR COMPLETENESS, TITLE, ANY EPIDEMIC FAILURE WARRANTY AND ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT OF ANY THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

TI SHALL NOT BE LIABLE FOR AND SHALL NOT DEFEND OR INDEMNIFY YOU AGAINST ANY CLAIM, INCLUDING BUT NOT LIMITED TO ANY INFRINGEMENT CLAIM THAT RELATES TO OR IS BASED ON ANY COMBINATION OF PRODUCTS EVEN IF DESCRIBED IN TI RESOURCES OR OTHERWISE. IN NO EVENT SHALL TI BE LIABLE FOR ANY ACTUAL, DIRECT, SPECIAL, COLLATERAL, INDIRECT, PUNITIVE, INCIDENTAL, CONSEQUENTIAL OR EXEMPLARY DAMAGES IN CONNECTION WITH OR ARISING OUT OF TI RESOURCES OR USE THEREOF, AND REGARDLESS OF WHETHER TI HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

You agree to fully indemnify TI and its representatives against any damages, costs, losses, and/or liabilities arising out of your non-compliance with the terms and provisions of this Notice.

This Notice applies to TI Resources. Additional terms apply to the use and purchase of certain types of materials, TI products and services. These include; without limitation, TI's standard terms for semiconductor products (<http://www.ti.com/sc/docs/stdterms.htm>), [evaluation modules](#), and [samples](http://www.ti.com/sc/docs/sampterm.htm) (<http://www.ti.com/sc/docs/sampterm.htm>).

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265  
Copyright © 2017, Texas Instruments Incorporated