LUMINARY MICRO®

Stellaris® Boot Loader

USER'S GUIDE

SW-BOOTLDR-UG-4423

Legal Disclaimers and Trademark Information

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH LUMINARY MICRO PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN LUMINARY MICRO'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, LUMINARY MICRO ASSUMES NO LIABILITY WHATSOEVER, AND LUMINARY MICRO DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF LUMINARY MICRO'S PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT. LUMINARY MICRO'S PRODUCTS ARE NOT INTENDED FOR USE IN MEDICAL, LIFE SAVING, OR LIFE-SUSTAINING APPLICATIONS.

Luminary Micro may make changes to specifications and product descriptions at any time, without notice. Contact your local Luminary Micro sales office or your distributor to obtain the latest specifications and before placing your product order.

Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Luminary Micro reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them.

Luminary Micro, Inc.
108 Wild Basin, Suite 350
Austin, TX 78746
Main: +1-512-279-8800
Fax: +1-512-279-8879
http://www.luminarymicro.com

Revision Information

This is version 4423 of this document, last updated on April 10, 2009.

Table of Contents

1 Introduction

The Luminary Micro® Stellaris® boot loader 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 Stellaris ARM® Cortex™-M3-based microcontroller. The boot loader can be built to use either the UART0, SSI0, I2C0, CAN, Ethernet or USB ports to update the code on the microcontroller. The boot loader is customizable via source code modifications, or simply deciding at compile time which routines to include. Since full source code is provided, the boot loader can be completely customized.

Three update protocols are utilized. On UART0, SSI0, I2C0, 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, however, different protocols are employed. On Ethernet the standard bootstrap protocol (BOOTP) is used and on USB, updates are performed via the standard Device Firmware Upgrade (DFU) class.

When configured to use UART0 or Ethernet, the LM Flash Programmer GUI can be used to download an application via the boot loader. The LM Flash Programmer utility is available for download from www.luminarymicro.com.

Note:
Building the boot loader requires the use of linker scripts, and building applications that run under its control requires the ability to specify a start address other than the beginning of flash. Neither of these capabilities are available in the evaluation version of Keil RealView Microcontroller Development Kit; therefore, the boot loader is not usable unless the full version is used. Additionally, the linker script specified in the uVision project file for the boot loader is simply ignored, resulting in a successful link of the boot loader but an image that will not operate correctly.

Source Code Overview

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

`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 via 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 boot loader.

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 boot loader 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 via the Ethernet port.
bl_i2c.c	The functions for transferring data via the I2C0 port.
bl_i2c.h	Prototypes for the I2C0 transfer functions.
bl_link.ld	The linker script used when the codered, gcc, or sourcerygxx compiler is being used to build the boot loader.
bl_link.sct	The linker script used when the rvmdk compiler is being used to build the boot loader.
bl_link.xcl	The linker script used when the ewarm compiler is being used to build the boot loader.
bl_main.c	The main control loop of the boot loader.
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 via the SSI0 port.
bl_ssi.h	Prototypes for the SSI0 transfer functions.
bl_startup_codered.S	The start-up code used when the codered compiler is being used to build the boot loader.
bl_startup_ewarm.S	The start-up code used when the ewarm compiler is being used to build the boot loader.
bl_startup_gcc.S	The start-up code used when the gcc compiler is being used to build the boot loader.

bl_startup_rvmdk.S	The start-up code used when the rvmdk compiler is being used to build the boot loader.
bl_startup_sourcerygxx.S	The start-up code used when the sourcerygxx compiler is being used to build the boot loader.
bl_uart.c	The functions for transferring data via the UART0 port.
bl_uart.h	Prototypes for the UART0 transfer functions.
bl_usb.c	Main functions implementing the USB DFU protocol boot loader.
bl_usbfuncs.c	A cut-down version of the USB library containing support for enumeration and the endpoint 0 transactions requied 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 boot loader.

2 Startup Code

The start-up code contains the minimal set of code required to configure a vector table, initialize memory, copy the boot loader 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_codered.S` (Code Red Technologies tools)
- `bl_startup_ewarm.S` (IAR Embedded Workbench)
- `bl_startup_gcc.S` (GNU GCC)
- `bl_startup_rvmdk.S` (Keil RV-MDK)
- `bl_startup_sourcerygxx.S` (CodeSourcery Sourcery G++)

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.ld` (Code Red Technologies tools, GNU GCC, and CodeSourcery Sourcery G++)
- `bl_link.sct` (Keil RV-MDK)
- `bl_link.xcl` (IAR Embedded Workbench)

The boot loader'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 boot loader to update itself since it is actually running from SRAM only. The first part of SRAM is used as the copy space for the boot loader while the rest is reserved for stack and read/write data for the boot loader. Once the boot loader calls the application, all SRAM becomes usable by the application.

The vector table of the Cortex-M3 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 boot loader'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 boot loader 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 boot loader can provide its own handlers by adding the appropriate handlers to the `Vectors` array.

After a reset, the start-up code copies the boot loader from flash to SRAM, branches to the copy of the boot loader 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 boot loader. For UART0, SSI0, and I2C0, 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 boot loader calls the Ethernet control loop `UpdateBOOTP()`. For CAN, the microcontroller is initialized by calling `ConfigureCAN()` and then the boot loader calls the CAN control loop `UpdaterCAN()`. For USB, the microcontroller is initialized by calling `ConfigureUSB` after which the function `UpdaterUSB` places the DFU class device on the bus and handles interaction with the USB host.

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 boot loader.

Additionally, the boot loader can be called by the application in order to perform an application-directed update. In this case, the boot loader assumes that the peripheral in use for the update has already been configured by the application, and must simply be used by the boot loader to perform the update. The boot loader therefore copies itself to SRAM, branches to the SRAM copy of the boot loader, and starts the update by calling `Updater()` (for UART0, SSI0, and I2C0), `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 via a serial port (UART0, SSI0, or I2C0), `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()`). Once the update is complete, the boot loader can be reset by issuing a reset command to the boot loader.

When a request to update the application comes through and **FLASH_CODE_PROTECTION** is defined, the boot loader 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 boot loader itself is left in place so that it will not boot a partially erased program. Once all of the application flash area has been successfully erased, the boot loader proceeds with the download of the new binary. When **FLASH_CODE_PROTECTION** is not defined, the boot loader only erases enough space to fit the new application that is being downloaded.

In the event that the boot loader itself needs to be updated, the boot loader must replace itself in flash. An update of the boot loader is recognized by performing a download to address 0x0000.0000. Once again the boot loader operates differently based on the setting of **FLASH_CODE_PROTECTION**. When **FLASH_CODE_PROTECTION** is defined and the download address indicates an boot loader update, the boot loader 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 boot loader 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 boot loader only erases enough space to fit its own code and leaves the application intact.

3.1 Packet Handling

The boot loader 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 boot loader uses the `SendPacket()` function in order 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 boot loader 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 non-zero 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 non-zero 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 boot loader.

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.

3.2 Transport Layer

The boot loader supports updating via the I2C0, SSI0, and UART0 ports which are available on Stellaris 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 I2C interface also provides a standard interface, only uses two wires, and can operate at comparable speeds to the UART and SSI interfaces.

3.2.1 I2C Transport

The I2C handling functions are `I2CSend()`, `I2CReceive()`, and `I2CFlush()` functions. The connections required to use the I2C port are the following pins: **I2CSCL** and **I2CSDA**. The device communicating with the boot loader must operate as the I2C master and provide the **I2CSCL** signal. The **I2CSDA** pin is open drain and can be driven by either the master or the slave I2C device.

3.2.2 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 boot loader is responsible for driving the **SSIRx**, **SSIClk**, and **SSIFss** pins, while the Stellaris 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 Stellaris Family 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 boot loader.

3.2.3 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 boot loader is responsible for driving the **U0Rx** pin on the Stellaris microcontroller, while the Stellaris 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 boot loader, if the auto-baud feature is enabled, or it can be fixed at a baud rate supported by the device communicating with the boot loader. 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 boot loader. This is the hardware requirement for the maximum baud rate for a UART on any Stellaris microcontroller.

When using a fixed baud rate, the frequency of the crystal connected to the microcontroller must be specified. Otherwise, the boot loader will not be able to configure the UART to operate at the requested baud rate.

The boot loader 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 boot loader can make multiple calls to `UARTAutoBaud()` to attempt to retry the synchronization if the first call fails. In the example boot loader provided, when the auto-baud feature is enabled, the boot loader will wait forever for a valid synchronization pattern from the host.

3.3 Serial Commands

The following commands are used by the custom protocol on the UART0, SSI0, and I2C0 ports:

`COMMAND_PING`

This command is used to receive an acknowledge from the boot loader indicating that communication has been established. This command is a single byte.

The format of the command is as follows:

```
unsigned char ucCommand[1];

ucCommand[0] = COMMAND_PING;
```

COMMAND_DOWNLOAD

This command is sent to the boot loader 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 erasure 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/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 boot loader.

The format of the command is as follows:

```
unsigned char ucCommand[9];

ucCommand[0] = COMMAND_DOWNLOAD;
ucCommand[1] = Program Address [31:24];
ucCommand[2] = Program Address [23:16];
ucCommand[3] = Program Address [15:8];
ucCommand[4] = Program Address [7:0];
ucCommand[5] = Program Size [31:24];
ucCommand[6] = Program Size [23:16];
ucCommand[7] = Program Size [15:8];
ucCommand[8] = Program Size [7:0];
```

COMMAND_RUN

This command is sent to the boot loader 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.

The format of the command is as follows:

```
unsigned char ucCommand[5];

ucCommand[0] = COMMAND_RUN;
ucCommand[1] = Run Address [31:24];
ucCommand[2] = Run Address [23:16];
ucCommand[3] = Run Address [15:8];
ucCommand[4] = Run Address [7:0];
```

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 boot loader should respond by sending a packet with one byte of data that contains the current status code.

The format of the command is as follows:

```
unsigned char ucCommand[1];

ucCommand[0] = COMMAND_GET_STATUS;
```

The following are the definitions for the possible status values that can be returned from the boot loader when COMMAND_GET_STATUS is sent to the the microcontroller.

```
COMMAND_RET_SUCCESS
COMMAND_RET_UNKNOWN_CMD
COMMAND_RET_INVALID_CMD
COMMAND_RET_INVALID_ADD
COMMAND_RET_FLASH_FAIL
```

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 boot loader (as configured by the BUFFER_SIZE parameter). The command terminates programming once 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 boot loader sends a NAK to this command, the boot loader will not increment the current address which allows for retransmission of the previous data.

The format of the command is as follows:

```
unsigned char ucCommand[9];

ucCommand[0] = COMMAND_SEND_DATA
ucCommand[1] = Data[0];
ucCommand[2] = Data[1];
ucCommand[3] = Data[2];
ucCommand[4] = Data[3];
ucCommand[5] = Data[4];
ucCommand[6] = Data[5];
ucCommand[7] = Data[6];
ucCommand[8] = Data[7];
```

COMMAND_RESET

This command is used to tell the boot loader to reset. This is used after downloading a new image to the microcontroller to cause the new application or the new boot loader 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 boot loader if a critical error occurs and the host device wants to restart communication with the boot loader.

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

The format of the command is as follows:

```
unsigned char ucCommand[1];

ucCommand[0] = COMMAND_RESET;
```

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 boot loader 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. Once 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 boot loader also acts as the client in this protocol. As each data block is received, it is programmed into flash. Once all data blocks are received and programmed, the device is reset, causing it to start running the new firmware image.

The uIP stack (`http://www.sics.se/~adam/uip`) 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.

Note:
When using the Ethernet update, the boot loader can not update itself since there is no mechanism in BOOTP to distinguish between a firmware image and a boot loader image. Therefore, the boot loader does not know if a given image is a new boot loader 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 boot loader calls `ConfigureCAN()` to configure the CAN controller and prepare the boot loader to update the firmware. The CAN update mechanism allows the boot loader to be entered from a functioning CAN application as well from startup when no application has been downloaded to the microcontroller. The boot loader provides the main routine for performing the CAN update in the `UpdaterCAN()` function which is used in both cases.

When the device enters the boot loader from a running CAN network, the boot loader will not reconfigure the CAN clocks or bit timing and will assume that they have been configured as expected by the firmware update device. The boot loader 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 boot loader 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 boot loader 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 boot loader 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 once the number of bytes indicated by the LM_API_UPD_DOWNLOAD command have been received. The CAN boot loader 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 boot loader 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[8] = Data[7];
```

LM_API_UPD_RESET	This command is used to tell the CAN boot loader to reset the microcontroller. This is used after downloading a new image to the microcontroller to cause the new application or the new boot loader 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 boot loader if a critical error occurs and the CAN update application needs to restart communication with the boot loader.

6 USB Update

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

When the USB boot loader is invoked from a running application, the boot loader 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 boot loader.

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

The USB boot loader allows a USB host to upgrade the firmware on a USB device. To make use of it, therefore, the board running the boot loader 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 boot loader.

6.1 USB Device Firmware Upgrade Overview

The USB boot loader 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 web site at `http://www.usb.org/developers/devclass_docs#approved`.

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 boot loader 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 specificed in its DFU functional descriptor.

The USB boot loader 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 transfering control to the USB boot loader by making a call through the SVC vector in the usual manner.

6.1.1 DFU Requests

Requests supported by the USB boot loader are as follow:

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 will be 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 via 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 triffer 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 via 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 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

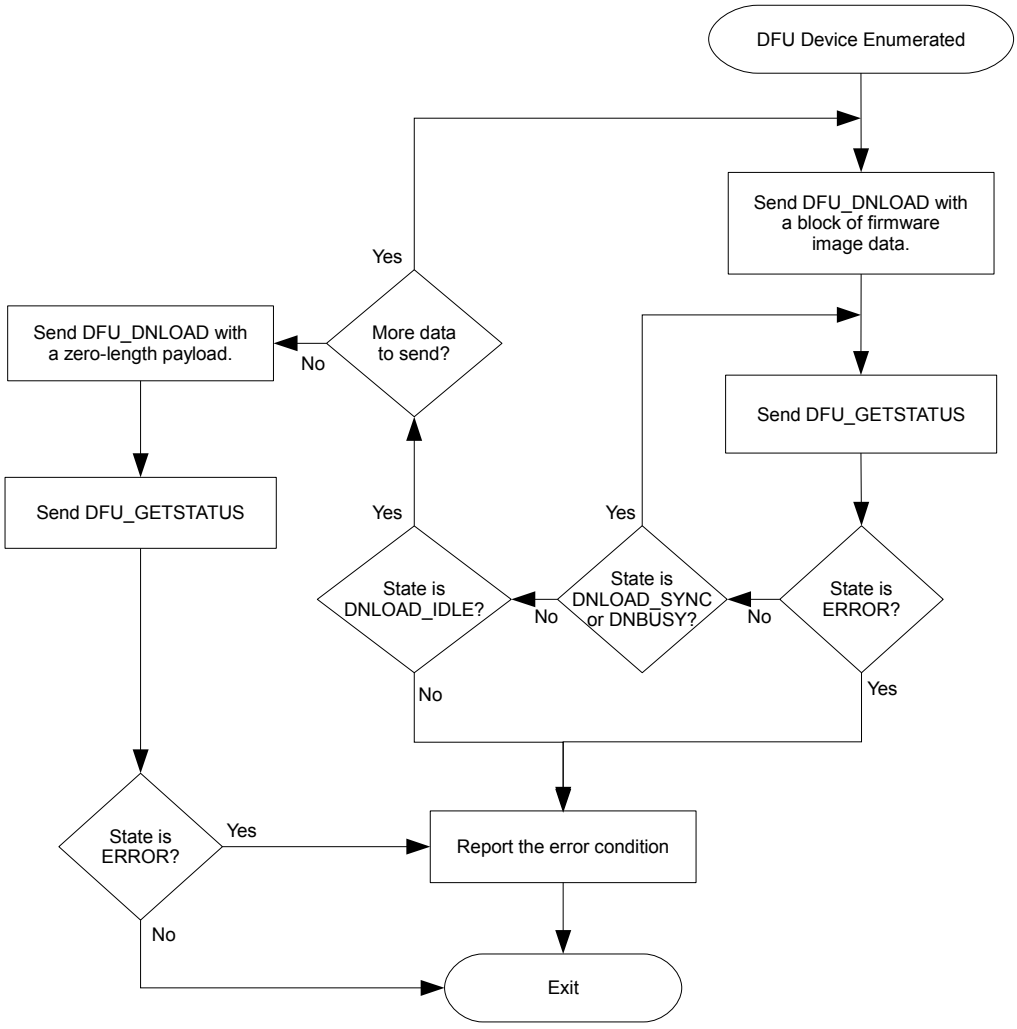
During operation, the DFU device transitions between a set of class-defined states. The host must query the current state to determine when a new operation can be performed or to determine the cause of any errors reported. These states are:

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 recieved, 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 boot loader image size, this state is not supported. Instead of reporting DNBUSY, the USB boot loader 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 signalled 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 recieved and the device is waiting for a DFU_GETSTATUS request before finalizing programming of the image. The USB boot loader 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 once 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 boot loader 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 boot loader 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

The following flow chart illustrates a typical firmware image download sequence from the perspective of the host application.


DFU Device Enumerated
Send DFU_DNLOAD with a block of firmware image data.
Send DFU_GETSTATUS
State is ERROR?
Yes
No
State is DNLOAD_SYNC or DNBUSY?
Yes
No
State is DNLOAD_IDLE?
Yes
No
More data to send?
Yes
No
Send DFU_DNLOAD with a zero-length payload.
Send DFU_GETSTATUS
State is ERROR?
Yes
No
Report the error condition
Exit

6.2 Luminary-Specific 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 transfered. 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 boot loader implementation employs a small set of commands which can be sent to the DFU device via 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 Luminary-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 Luminary Micro USB boot loader using a standard sequence of DFU_DNLOAD requests with no other embedded commands or device-specific operations required. An application which does understand the Luminary-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 Luminary USB boot loader 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 Luminary 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 Luminary-specific DFU commands, send the following IN request to the DFU interface:

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 two bytes of which are 0x4D, 0x4C and where the second group of two 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 boot loader as the first 8 bytes of the payload to a DFU_DNLOAD request. The boot loader will expect any DFU_DNLOAD request received while in IDLE state to contain a command header but will 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 boot loader 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 is 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 boot loader 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. Note that 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 boot loader allows for customization of its features as well as the interfaces used to update the microcontroller. This allows the boot loader 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 boot loader.

The boot loader 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 boot loader. 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 boot loader's packet format does not meet the needs of the application, it can be completely replaced by replacing `ReceivePacket()`, `SendPacket()`, `AckPacket()`, and `NakPacket()`.

The boot loader also provides a method to add a new serial transmission interface beyond the UART, SSI, and I2C that are provided by the boot loader. In order 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:

```
#ifdef 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 boot loader to define whatever protocol it needs, or use any port that it needs to perform updates of the microcontroller.

8 Configuration

There are a number of defines that are used to configure the operation of the boot loader. These defines are located in the `bl_config.h` header file, for which there is a template (`bl_config.h.tmpl`) provided with the boot loader.

The configuration options are:

`CRYSTAL_FREQ`

This defines the crystal frequency used by the microcontroller running the boot loader. 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 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.

`BOOST_LDO_VOLTAGE`

This enables the boosting of the LDO voltage to 2.75V. For boot loader configurations that enable the PLL (in other words, using the Ethernet port) on a part that has the PLL errata, this should be enabled. This applies to revision A2 of Fury-class devices.

`APP_START_ADDRESS`

The starting address of the application. This must be a multiple of 1024 bytes (making it aligned to a 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 boot loader must not be larger than this value.

`FLASH_RSVD_SPACE`

The amount of space at the end of flash to reserve. This must be a multiple of 1024 bytes (making it aligned to a page boundary). This reserved space is not erased when the application is updated, providing non-volatile storage that can be used for parameters.

`STACK_SIZE`

The number of words of stack space to reserve for the boot loader.

- 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 is unused when updating via the Ethernet port. ■ This value must be defined.
ENABLE_BL_UPDATE	Enables updates to the boot loader. Updating the boot loader is an unsafe operation since it is not fully fault tolerant (losing power to the device partway through could result in the boot loader no longer being present in flash). The boot loader can not be updated via the Ethernet port.
FLASH_CODE_PROTECTION	This definition will cause the the boot loader to erase the entire flash on updates to the boot loader 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 boot loader source, which simply provides a placeholder for adding an actual decryption algorithm.
ENABLE_UPDATE_CHECK	Enables the pin-based forced update check. When enabled, the boot loader 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 boot loader in order to start an update.
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.
UART_ENABLE_UPDATE	Selects the UART as the port for communicating with the boot loader.
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.
UART_FIXED_BAUDRATE	Selects the baud rate to be used for the UART.
SSI_ENABLE_UPDATE	Selects the SSI port as the port for communicating with the boot loader.
I2C_ENABLE_UPDATE	Selects the I2C port as the port for communicating with the boot loader.
I2C_SLAVE_ADDR	Specifies the I2C address of the boot loader. ■ This value must be defined if I2C_ENABLE_UPDATE is defined.
CAN_ENABLE_UPDATE	Selects an update via the CAN port.
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 between 0 and 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.
ENET_ENABLE_UPDATE	Selects an update via the Ethernet port.
ENET_ENABLE_LEDS	Enables the use of the Ethernet status LED outputs to indicate traffic and connection status.
ENET_MAC_ADDR?	Specifies the hard coded MAC address for the Ethernet interface. There are six individual values (ENET_MAC_ADDR0 through ENET_MAC_ADDR5) that provide the six bytes of the MAC address, where ENET_MAC_ADDR0 though ENET_MAC_ADDR2 provide the organizationally unique identifier (OUI) and ENET_MAC_ADDR3 through ENET_MAC_ADDR5 provide the extension identifier. If these values are not provided, the MAC address will be 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 Luminary Micro requires that this be set to "stellaris".
USB_ENABLE_UPDATE	Selects an update via the USB interface.
USB_VENDOR_ID	Sets the vendor ID published to the USB host in the idVendor field of the device descriptor.
USB_PRODUCT_ID	Sets the product ID published to the USB host in the idProduct field of the device descriptor.
USB_DEVICE_ID	Sets the device release number published to the USB host in the bcdDevice field of the device descriptor.
USB_MAX_POWER	Sets the maximum power consumption that the USB DFU device will report to the host in the bMaxPower field of the configuration descriptor. The units are milliamps and this value is divided by 2 to derive the actual number published in the descriptor (where the value is expressed in 2mA units).
USB_BUS_POWERED	Determines whether the DFU device reports to the USB host that it is self powered (0) or bus powered (1) via the bmAttributes field of the configuration descriptor.
USB_HAS_MUX	Determines whether the target board uses a multiplexer to select between USB host and device modes. If this label is defined, the target does use a mux and the fields USB_MUX_PERIPH, USB_MUX_PORT, USB_MUX_PIN and USB_MUX_DEVICE must also be defined to allow the boot loader to correctly switch the board into USB device mode.
USB_MUX_PERIPH	Defines the GPIO peripheral containing the pin which is used to select between USB host and device modes. The value is of the form SYSCTL_PERIPH_GPIOx where GPIOx represents the required GPIO port. This label is required if USB_HAS_MUX is defined.
USB_MUX_PORT	Defines the GPIO peripheral containing the pin which is used to select between USB host and device modes. The value is of the form GPOP_PORTx_BASE where PORTx represents the required GPIO port. This label is required if USB_HAS_MUX is defined.
USB_MUX_PIN	Defines the GPIO pin number used to select between USB host and device modes. Valid values are 0 through 7. This label is required if USB_HAS_MUX is defined.

USB_MUX_DEVICE	Defines the state of the GPIO pin required to select USB device-mode operation. Valid values are 0 (low) or 1 (high). This label is required if USB_HAS_MUX is defined.

9 Source Details

9.1 Autobaud Functions

Functions

- void GPIOIntHandler (void)
- int UARTAutoBaud (unsigned long *pulRatio)

9.1.1 Detailed Description

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

9.1.2 Function Documentation

9.1.2.1 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.

9.1.2.2 UARTAutoBaud

Performs auto-baud on the UART port.

Prototype:

```
int
UARTAutoBaud(unsigned long *pulRatio)
```

Parameters:

pulRatio 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 boot loader. The UART port is monitored for edges using interrupts. Once enough edges are detected, the boot loader 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

Functions

- void AppUpdaterCAN (void)
- void CANInit (void)
- unsigned long CANMessageGetRx (unsigned char *pucData, unsigned long *pulMsgID)
- void CANMessageSetRx (void)
- void CANMessageSetTx (unsigned long ulId, const unsigned char *pucData, unsigned long ulSize)
- unsigned long CANReceive (unsigned char *pucData, unsigned long *pulSize)
- void CANSend (const unsigned char *pucData, unsigned long ulSize)
- void ConfigureCAN (void)
- void ConfigureCANInterface (unsigned long ulSetTiming)
- void UpdaterCAN (void)

9.2.1 Detailed Description

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

9.2.2 Function Documentation

9.2.2.1 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 boot loader with no application present.

Returns:

None.

9.2.2.2 CANInit

Initializes the CAN controller after reset.

Prototype:

```
void
CANInit(void)
```

Description:

After reset, the CAN controller is left in the disabled state. However, the memory used for message objects contains undefined values and must be cleared prior to enabling the CAN controller the first time. This prevents unwanted transmission or reception of data before the message objects are configured. This function must be called before enabling the controller the first time.

Returns:

None.

9.2.2.3 CANMessageGetRx

This function reads data from the receive message object.

Prototype:

```
unsigned long
CANMessageGetRx(unsigned char *pucData,
                unsigned long *pulMsgID)
```

Parameters:

pucData is a pointer to the buffer to store the data read from the CAN controller.

pulMsgID is a pointer to the ID that was received with the data.

Description:

This function will reads and acknowledges the data read from the message object used to receive all CAN firmware update messages. It will also return the message identifier as this

holds the API number that was attached to the data. This message identifier should be one of the LM_API_UPD_* definitions.

Returns:
The number of valid bytes returned in the *pucData* buffer or 0xffffffff if data was overwritten in the buffer.

9.2.2.4 CANMessageSetRx

This function configures the message object used to receive commands.

Prototype:
```
void
CANMessageSetRx(void)
```

Description:
This function configures the message object used to receive all firmware update messages. This will not actually read the data from the message it is used to prepare the message object to receive the data when it is sent.

Returns:
None.

9.2.2.5 CANMessageSetTx

This function sends data using the transmit message object.

Prototype:
```
void
CANMessageSetTx(unsigned long ulId,
                const unsigned char *pucData,
                unsigned long ulSize)
```

Parameters:
ulId is the ID to use with this message.
pucData is a pointer to the buffer with the data to be sent.
ulSize is the number of bytes to send and should not be more than 8 bytes.

Description:
This function will reads and acknowledges the data read from the message object used to receive all CAN firmware update messages. It will also return the message identifier as this holds the API number that was attached to the data. This message identifier should be one of the LM_API_UPD_* definitions.

Returns:
None.

9.2.2.6 CANReceive

Receives data over the CAN interface.

Prototype:

```
unsigned long
CANReceive(unsigned char *pucData,
           unsigned long *pulSize)
```

Parameters:

pucData is the buffer to read data into from the CAN interface.

pulSize is a pointer to the number of bytes provided in the *pucData* buffer that should be written with data from the CAN interface.

Description:

This function reads back *ulSize* bytes of data from the CAN interface, into the buffer that is pointed to by *pucData*. This function will not return until *pulSize* number of bytes have been received.

Returns:

None.

9.2.2.7 CANSend

Sends data over the CAN interface.

Prototype:

```
void
CANSend(const unsigned char *pucData,
        unsigned long ulSize)
```

Parameters:

pucData is the buffer containing the data to write out to the CAN interface.

ulSize is the number of bytes provided in *pucData* buffer that will be written out to the CAN interface.

Description:

This function sends *ulSize* bytes of data from the buffer pointed to by *pucData* via the CAN interface.

Returns:

None.

9.2.2.8 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.

9.2.2.9 ConfigureCANInterface

Configures the CAN interface.

Prototype:
```
void
ConfigureCANInterface(unsigned long ulSetTiming)
```

Parameters:
ulSetTiming determines if the CAN bit timing should be configured.

Description:
This function configures the CAN controller, preparing it for use by the boot loader. If the *ulSetTiming* paramter is 0, the bit timing for the CAN bus will be left alone. This occurs when the boot loader was entered from a running application that already has configured the timing for the system. When *ulSetTiming* is non-zero the bit timing will be set to the defaults defined in the `bl_config.h` file in the project.

Returns:
None.

9.2.2.10 UpdaterCAN

This is the main routine for handling updating over CAN.

Prototype:
```
void
UpdaterCAN(void)
```

Description:
This function accepts boot loader 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

Functions

- void DecryptData (unsigned char *pucBuffer, unsigned long ulSize)

9.3.1 Detailed Description

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

9.3.2 Function Documentation

9.3.2.1 DecryptData

Performs an in-place decryption of downloaded data.

Prototype:

```
void
DecryptData(unsigned char *pucBuffer,
            unsigned long ulSize)
```

Parameters:

pucBuffer is the buffer that holds the data to decrypt.

ulSize is the size, in bytes, of the buffer that was passed in via the pucBuffer 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

Functions

- char BOOTPThread (void)
- void ConfigureEnet (void)
- void SysTickIntHandler (void)
- void UpdateBOOTP (void)

9.4.1 Detailed Description

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

9.4.2 Function Documentation

9.4.2.1 BOOTPThread

Handles the BOOTP process.

Prototype:

```
char
BOOTPThread(void)
```

Description:

This function contains the proto-thread for handling the BOOTP process. It first communicates with the BOOTP server to get its boot parameters (IP address, server address, and file name), then it communicates with the TFTP server on the specified server to read the firmware image file.

Returns:

None.

9.4.2.2 ConfigureEnet

Configures the Ethernet controller.

Prototype:

```
void
ConfigureEnet(void)
```

Description:

This function configures the Ethernet controller, preparing it for use by the boot loader.

Returns:

None.

9.4.2.3 SysTickIntHandler

Handles the SysTick interrupt.

Prototype:

```
void
SysTickIntHandler(void)
```

Description:

This function is called when the SysTick interrupt occurs. It simply keeps a running count of interrupts, used as a time basis for the BOOTP and TFTP protocols.

Returns:

None.

9.4.2.4 UpdateBOOTP

Starts the update process via BOOTP.

Prototype:

```
void
UpdateBOOTP(void)
```

Description:
This function starts the Ethernet firmware update process. The BOOTP (as defined by RFC951 at http://tools.ietf.org/html/rfc951) and TFTP (as defined by RFC1350 at http://tools.ietf.org/html/rfc1350) protocols are used to transfer the firmware image over Ethernet.

Returns:
Never returns.

9.5 I2C Functions

Functions

- void I2CFlush (void)
- void I2CReceive (unsigned char *pucData, unsigned long ulSize)
- void I2CSend (const unsigned char *pucData, unsigned long ulSize)

9.5.1 Detailed Description

The following functions are provided in `bl_i2c.c` and are used to communicate over the I2C interface.

9.5.2 Function Documentation

9.5.2.1 I2CFlush

Waits until all data has been transmitted by the I2C port.

Prototype:
```
void
I2CFlush(void)
```

Description:
This function waits until all data written to the I2C port has been read by the master.

Returns:
None.

9.5.2.2 I2CReceive

Receives data over the I2C port.

Prototype:
```
void
I2CReceive(unsigned char *pucData,
           unsigned long ulSize)
```

Parameters:

pucData is the buffer to read data into from the I2C port.

ulSize is the number of bytes provided in the *pucData* buffer that should be written with data from the I2C port.

Description:

This function reads back *ulSize* bytes of data from the I2C port, into the buffer that is pointed to by *pucData*. This function will not return until *ulSize* number of bytes have been received. This function will wait till the I2C Slave port has been properly addressed by the I2C Master before reading the first byte of data from the I2C port.

Returns:

None.

9.5.2.3 I2CSend

Sends data over the I2C port.

Prototype:

```
void
I2CSend(const unsigned char *pucData,
        unsigned long ulSize)
```

Parameters:

pucData is the buffer containing the data to write out to the I2C port.

ulSize is the number of bytes provided in *pucData* buffer that will be written out to the I2C port.

Description:

This function sends *ulSize* bytes of data from the buffer pointed to by *pucData* via the I2C port. The function will wait till the I2C Slave port has been properly addressed by the I2C Master device before sending the first byte.

Returns:

None.

9.6 Main Functions

Functions

- void ConfigureDevice (void)
- void Updater (void)

9.6.1 Detailed Description

The following functions are provided in `bl_main.c` and comprise the main boot loader application.

9.6.2 Function Documentation

9.6.2.1 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 boot loader. The interface that has been selected as the update port will be configured, and auto-baud will be performed if required.

Returns:

None.

9.6.2.2 Updater

This function performs the update on the selected port.

Prototype:

```
void
Updater(void)
```

Description:

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

Returns:

Never returns.

9.7 Packet Handling Functions

Functions

- void AckPacket (void)
- unsigned long CheckSum (const unsigned char *pucData, unsigned long ulSize)
- void NakPacket (void)
- int ReceivePacket (unsigned char *pucData, unsigned long *pulSize)
- int SendPacket (unsigned char *pucData, unsigned long ulSize)

9.7.1 Detailed Description

The following functions are provided in `bl_packet.c` and are used to process the data packets in the custom serial protocol.

9.7.2 Function Documentation

9.7.2.1 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.

9.7.2.2 CheckSum

Calculates an 8-bit checksum

Prototype:

```
unsigned long
CheckSum(const unsigned char *pucData,
         unsigned long ulSize)
```

Parameters:
pucData is a pointer to an array of 8-bit data of size ulSize.
ulSize 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.

9.7.2.3 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.

9.7.2.4 ReceivePacket

Receives a data packet.

Prototype:

```
int
ReceivePacket(unsigned char *pucData,
              unsigned long *pulSize)
```

Parameters:

pucData is the location to store the data that is sent to the boot loader.

pulSize is the number of bytes returned in the pucData buffer that was provided.

Description:

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

Returns:

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

9.7.2.5 SendPacket

Sends a data packet.

Prototype:

```
int
SendPacket(unsigned char *pucData,
           unsigned long ulSize)
```

Parameters:

pucData is the location of the data to be sent.

ulSize is the number of bytes to send.

Description:

This function sends the data provided in the *pucData* parameter in the packet format used by the boot loader. 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 non-zero value indicates a failure.

9.8 SSI Functions

Functions

- void SSIFlush (void)
- void SSIReceive (unsigned char *pucData, unsigned long ulSize)
- void SSISend (const unsigned char *pucData, unsigned long ulSize)

9.8.1 Detailed Description

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

9.8.2 Function Documentation

9.8.2.1 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.

9.8.2.2 SSIReceive

Receives data from the SSI port in slave mode.

Prototype:

```
void
SSIReceive(unsigned char *pucData,
           unsigned long ulSize)
```

Parameters:

pucData is the location to store the data received from the SSI port.

ulSize is the number of bytes of data to receive.

Description:

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

Returns:

None.

9.8.2.3 SSISend

Sends data via the SSI port in slave mode.

Prototype:

```
void
SSISend(const unsigned char *pucData,
        unsigned long ulSize)
```

Parameters:

pucData is the location of the data to send through the SSI port.

ulSize is the number of bytes of data to send.

Description:

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

Returns:

None.

9.9 UART Functions

Functions

- void UARTFlush (void)
- void UARTReceive (unsigned char *pucData, unsigned long ulSize)
- void UARTSend (const unsigned char *pucData, unsigned long ulSize)

9.9.1 Detailed Description

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

9.9.2 Function Documentation

9.9.2.1 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.

9.9.2.2 UARTReceive

Receives data over the UART port.

Prototype:

```
void
UARTReceive(unsigned char *pucData,
            unsigned long ulSize)
```

Parameters:

pucData is the buffer to read data into from the UART port.

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

Description:

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

Returns:

None.

9.9.2.3 UARTSend

Sends data over the UART port.

Prototype:

```
void
UARTSend(const unsigned char *pucData,
         unsigned long ulSize)
```

Parameters:

pucData is the buffer containing the data to write out to the UART port.

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

Description:

This function sends *ulSize* bytes of data from the buffer pointed to by *pucData* via the UART port.

Returns:

None.

9.10 Update Check Functions

Functions

- unsigned long CheckForceUpdate (void)

9.10.1 Detailed Description

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

9.10.2 Function Documentation

9.10.2.1 CheckForceUpdate

Checks if an update is needed or is being requested.

Prototype:

```
unsigned long
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 non-zero value if an update is needed or is being requested and zero otherwise.

9.11 USB Functions

Data Structures

- tConfigDescriptor
- tString0Descriptor
- tStringDescriptor
- tUSBRequest

Functions

- void AppUpdaterUSB (void)
- void ConfigureUSB (void)
- void ConfigureUSBInterface (void)
- void HandleRequests (tUSBRequest *pUSBRequest)
- tBoolean ProcessDFUDnloadCommand (tDFUDownloadHeader *pcCmd, unsigned long ulSize)
- void UpdaterUSB (void)

9.11.1 Detailed Description

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

9.11.2 Data Structure Documentation

9.11.2.1 tConfigDescriptor

Definition:

```
typedef struct
{
    unsigned char bLength;
    unsigned char bDescriptorType;
    unsigned short wTotalLength;
    unsigned char bNumInterfaces;
    unsigned char bConfigurationValue;
    unsigned char iConfiguration;
    unsigned char bmAttributes;
    unsigned char 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 will be 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 2mA so, for example, 100 represents 200mA.

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.

9.11.2.2 tString0Descriptor

Definition:

```
typedef struct
{
    unsigned char bLength;
    unsigned char bDescriptorType;
    unsigned short 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 will be USB_DTYPE_STRING (3).

wLANGID The language code (LANGID) for the first supported language. Note that this descriptor may support multiple languages, in which case, the number of elements in the wLANGID array will increase and bLength will be updated accordingly.

Description:

This structure describes the USB string descriptor for index 0 as defined in USB 2.0 specification section 9.6.7. Note that 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).

9.11.2.3 tStringDescriptor

Definition:

```
typedef struct
{
    unsigned char bLength;
    unsigned char bDescriptorType;
    unsigned char bString;
}
tStringDescriptor
```

Members:

bLength The length of this descriptor in bytes. This value will be 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 will be 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.

9.11.2.4 tUSBRequest

Definition:

```
typedef struct
{
    unsigned char bmRequestType;
    unsigned char bRequest;
    unsigned short wValue;
    unsigned short wIndex;
    unsigned short 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.

9.11.3 Function Documentation

9.11.3.1 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 boot loader 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.

9.11.3.2 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.

9.11.3.3 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.

9.11.3.4 HandleRequests

Handle USB requests sent to the DFU device.

Prototype:

```
void
HandleRequests(tUSBRequest *pUSBRequest)
```

Parameters:

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

Description:

This function is called to handle all non-standard requests received by the device. This will include all the DFU endpoint 0 commands along with the Luminary-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.

9.11.3.5 ProcessDFUDnloadCommand

Process Luminary-specific commands passed via DFU download requests.

Prototype:

```
tBoolean
ProcessDFUDnloadCommand(tDFUDownloadHeader *pcCmd,
                        unsigned long ulSize)
```

Parameters:

pcCmd is a pointer to the first byte of the **DFU_DNLOAD** payload that is expected to hold a command.

ulSize is the number of bytes of data pointed to by *pcCmd*. 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 Luminary-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.

9.11.3.6 void UpdaterUSB (void)

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.

Returns:

None.

Company Information

Founded in 2004, Luminary Micro, Inc. designs, markets, and sells ARM Cortex-M3-based microcontrollers (MCUs). Austin, Texas-based Luminary Micro is the lead partner for the Cortex-M3 processor, delivering the world's first silicon implementation of the Cortex-M3 processor. Luminary Micro's introduction of the Stellaris family of products provides 32-bit performance for the same price as current 8- and 16-bit microcontroller designs. With entry-level pricing at $1.00 for an ARM technology-based MCU, Luminary Micro's Stellaris product line allows for standardization that eliminates future architectural upgrades or software tool changes.

Luminary Micro, Inc.
108 Wild Basin, Suite 350
Austin, TX 78746
Main: +1-512-279-8800
Fax: +1-512-279-8879
http://www.luminarymicro.com
sales@luminarymicro.com

Support Information

For support on Luminary Micro products, contact:

support@luminarymicro.com
+1-512-279-8800, ext 3