F2806x USB Bootloader

USER'S GUIDE

Copyright

Texas Instruments
13905 University Boulevard
Sugar Land, TX 77479
http://www.ti.com/c2000

Revision Information

This is version 2.07.00.00 of this document, last updated on Sun Aug 25 15:30:38 IST 2024.

Table of Contents

1   Introduction

The Texas Instruments® boot loader is a small piece of code that can be programmed in flash to act as an application loader as well as an update mechanism for applications running on a C2000 microcontroller. The boot loader is designed to upgrade the application using the USB Device Firmware Upgrade (DFU) mechanism. 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.

Source Code Overview

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

`aes_config_opts.h`	Configuration header for aes encryption and decryptions routines.
`bl_entrytable.asm`	Bootloader and Application Entry points.
`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.h`	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_hooks.h`	Provides hooks for user supplied functions.
`bl_sym.h`	C prototypes for linker and assembly symbols.
`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 required to implement the USB DFU device.
`bl_usbfuncs.h`	Prototypes for the functions provided in bl_usbfuncs.c.
`crc.h`	Prototypes CPU and VCU CRC routines.

`F2806x_BL_MemCopy.c`	Memory copy functions for use in both normal and failsafe boot modes.
`F2806x_CodeStartBranch.asm`	Special Code start branch that supports two flash entry locations.
`F2806x_CSMPasswords.asm`	Code Security passwords. ONLY MODIFY IF YOU WANT TO LOCK THE DEVICE!
`Flash2806x_API_Config.h`	Flash API configuration header.
`Flash2806x_API_Library.h`	Flash API function prototypes.
`usbdfu.h`	Type definitions, labels related to the USB Device Firmware Upgrade class boot loader.

Practical Use

This bootloader is an example piece of software and should be treated as a guide on implementing your own or used as a starting point for an end application bootloader. As with any embedded code you should do your best to understand how each piece of the solution works. The bootloader may be used unmodified with end application code that follows the same memory map and structure as the bl_app example project.

To evaluate the DFU USB Bootloader example you should follow these steps:

Import the boot_loader and bl_app projects into CCS. The projects can be found in examples∖bl_app and MWare∖boot_loader.

Build each of the projects

Load the bootloader project into the flash of the target device and execute it.

Connect the target device's USB port to the host PC.

When Windows asks about drivers, point it to the boot_usb.inf driver file found in MWare∖windows_drivers. Two options will be presented: Select the first if you have a 32 bit system and the second if you are using a 64 bit system.

Run dfuprog -e to enumerate the DFU enabled devices attached to the system. If your device shows up everything is working correctly.

Run dfuprog -c to clear the flash of the target device.

Run dfuprog -f bl_app_i.hex to load the bl_app example application to the board. You may optionally append the -r option to reset the target device and boot the bl_app application after programming is complete.

The bootloader solution requires that the application to be loaded follow a predefined memory map, have entry addresses and application signatures (CRC) at predefined locations that are known by both the bootloader and the end application. The end application must also be passed to the

dfuprog utility in a special hex image file format. The bl_app example automatically does this when the project is built using the bl_app_hex.cmd. Please refer to this file for the options one must pass to the hex2000 utility to correctly generate a hex file for the dfuprog software. These specially formatted hex files are the only supported file format for programming the device using the bootloader.

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. The start-up code is contained in `rts2800_bl.lib`, with `F28069_bl.cmd` containing the linker script used to place the vector table, code segment, and data segments into the appropriate locations in memory.

At boot time one of two things can happen depending on how the bootloader is configured. During a normal boot, the typical boot28 assembly code initializes the processor copies cinit and pinit and jumps to the bootloader's main function. There the bootloader checks for a valid application or a forced update, and then jumps to the application or starts the bootloader. The bootloader can also be built with a failsafe mode that minimizes chances that the bootloader could be corrupted. If this mode is enabled and the failsafe condition occurs (discussed in greater detail later in this guide), the bootloader will begin execution in the boot28_f file and perform the same initializations it would during a normal boot. Instead of jumping to main though the bootloader will enter the FailsafeEntry function (found in bl_usb.c). By definition there is no application present in the failsafe mode, so the bootloader is immediately started in this case.

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. Once the boot loader calls the application, all SRAM becomes usable by the application.

After a reset, the start-up code checks to see if an application update should be performed by calling `CheckForceUpdate()`. If an update is not required, the bootloader then check for a valid application. At a minimum the bootloader must find a valid address at the application entry pointer pAppEntry. A secondary CRC check can also optionally be perfomed using linker generated CRC tables. For more information on linker generated CRC tables please see SPRU513d Assembly Language Tools User Guide. Assuming an update is required, it is at this point that the bootloader kernel is copied from flash to SRAM. After the copy has completed the microcontroller is initialized by calling `ConfigureUSB()` after which the function `UpdaterUSB()` configures the USB interface for device mode.

The check for an application update (in `CheckForceUpdate()`) consists of checking the beginning of the application area and optionally checking the state of a GPIO pin. The 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 application can call the boot loader in order to perform an application-directed update. In this case, the boot loader assumes that the application has already configured the USB peripheral. This allows the boot loader to use the peripheral as is to perform the update. The boot loader also assumes that the interrupt to the core has been left enabled as well, which means that that application should not call DINT before calling the boot loader. Once the application calls the boot loader, the boot loader copies itself to SRAM, branches to the SRAM copy of the boot loader, and starts the update by calling `UpdaterUSB()`. The application should enter the bootloader via the AppUpdaterUSB function whose address can be found via the pBootEntry pointer (in bl_entrytable.asm).

3   USB Device (DFU) 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.

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 specified 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 transferring control to the USB boot loader by making a call to AppUpdaterUSB().

3.1 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 trigger a state change. During download, for example, the device enters DNLOAD_SYNC state after receiving a block of data and remains there until the data has been processed and a DFU_GETSTATUS request is received at which point the state changes to DNLOAD_IDLE.

DFU_CLRSTATUS
This request is used to reset any error condition reported by the DFU device. If an error is reported 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.
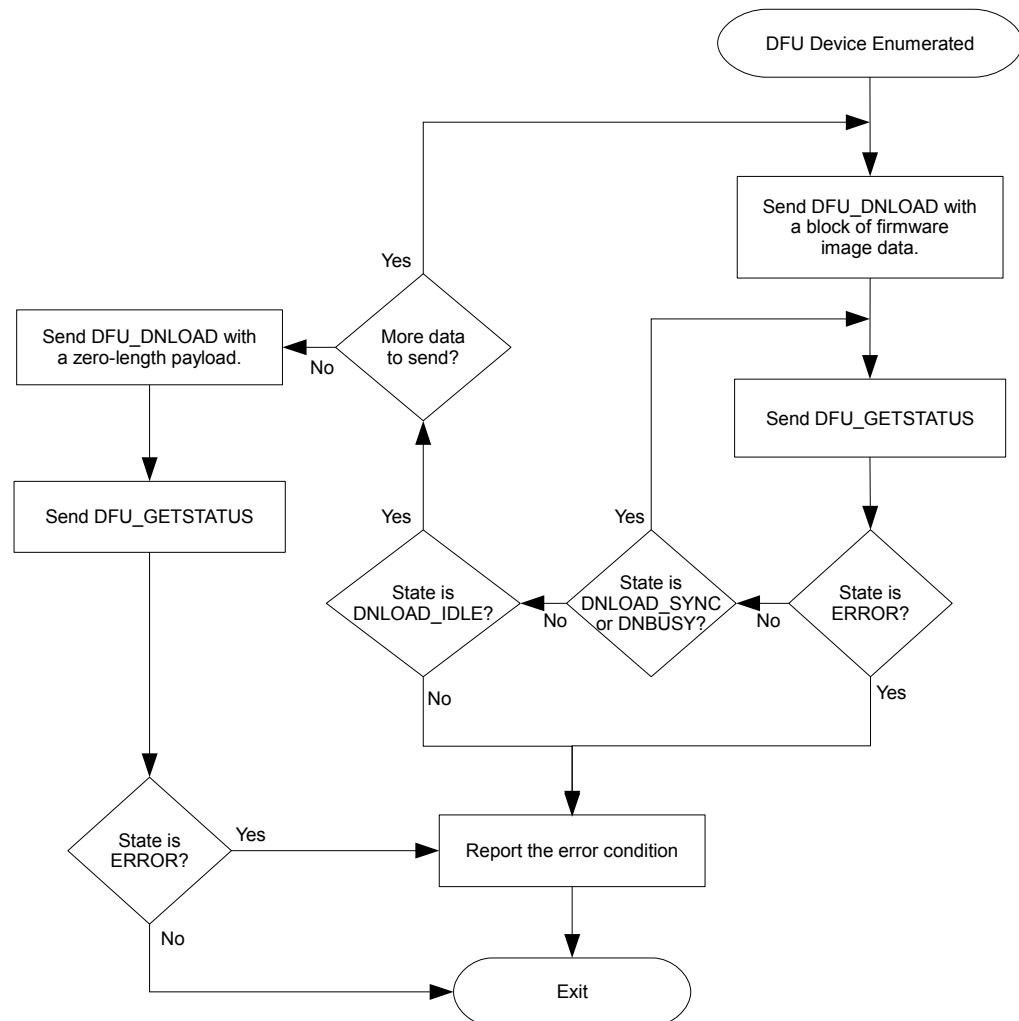
3.2   DFU States

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

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

DNLOAD_SYNC   After each DFU_DNLOAD request is received, DNLOAD_SYNC state is entered. This state remains in effect until the host issues a DFU_GETSTATUS request at which point the state will change to DNLOAD_IDLE if the last download operation has completed or DNBUSY otherwise.

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

DNBUSY   This state is reported if a DFU_GETSTATUS request is received while a block of downloaded data is still being processed. The host must refrain from issuing another DFU_GETSTATUS request for a time specified in the structure returned following the request. After this time, the device state reverts to DNLOAD_SYNC.
                          To reduce the USB 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 signaled by the host sending a DFU_DNLOAD request with a 0 length payload. When this request is received, the DFU device transitions from state DNLOAD_IDLE to MANIFEST_SYNC. This state indicates that the complete firmware image has been received and the device is waiting for a DFU_GETSTATUS request before finalizing programming of the image.
                          The USB 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.

3.3 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  │
                                                    └─────────────────────────┘
```


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

Setting the address that a downloaded binary should be flashed to.

Setting the address and size of the area of flash whose contents are to be uploaded.

Erasing sections of the flash.

Querying the size of flash and writeable area addresses. The USB 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 TI-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 TI 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 TI-specific commands may make use of them to offer additional functionality that would not otherwise be available.

3.4 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 TI 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 TI 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 TI-specific DFU commands, send the following IN request to the DFU interface:

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 0x20 respectively.

3.5 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 word flash blocks. To convert a word address to a block address, merely divide by 1024. The start address must always be on a 1024 word boundary. For example if you wanted to program data at address 0x3D8000 (start of flash) the block number sent would be 0xF60.

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. Keep in mind that while data is transferred over USB in bytes it is programmed into the device flash as words (2 bytes). The image size field should contain the number of bytes to be programmed (not words).

Data following this command in the following format 0xXX, 0xYY will be programmed into flash as 0xXXYY (i.e. the most significant byte of a word should be transmitted first).

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 words flash blocks. To convert a word 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 1KW 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 word flash blocks. To convert
                   a word address to a block address, merely divide by 1024. The start address
                   must always be on a 1024 word boundary. The region size is transmitted in
                   words.
                   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 suc-
                   cessful, the returned bStatus value will be OK (0x00), otherwise it will be er-
                   rCheckErased (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 word flash blocks. To convert
                   a word address to a block address, merely divide by 1024. The start address
                   must always be on a 1024 word boundary.
                   The size of the region to erase is expressed in terms of 1024 word flash blocks.

                   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 the PARTID register
    //
    unsigned long ulPartInfo;

    //
    // Information on the part class and revision as read
    // from the CLASSID
    //
    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;

    //
    // Features supported by the bootloader
    //
    tLong ulFeatures;

    //
    // If true the device is locked and flash reads/writes are disallowed
    //
    unsigned char ucLocked;
}
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
```

DFU_CMD_CSM   This command may be sent to the USB boot loader to cause it to lock or unlock the device's code security module. Unlocking the device is accomplished by first sending the CSM command with all fields set to 0. When the device receives this command it will generate a seed which can be retreived by performing a DFU Upload. The host should then perfom encryption on this seed using a pre-negotiated key. To finish authentication and unlock the device the host should again send a CSM command but with the key size field populated and the encrypted seed following the command. Success can be checked by using the info command. The device will automatically lock when reset if the CSM locations aren't all 0xFF, but the device can also be manually locked by sending the CSM command with the lock field a non-zero value.

The public release of the bootloader does not contain this feature.

The format of the command is as follows:

```
unsigned char ucData[8];

ucData[0] = DFU_CMD_RESET (0x08)
ucData[1] = Reserved - set to 0x00
ucData[2] = Lock device if true
ucData[3] = Key size following this command
ucData[4] = Reserved - set to 0x00
ucData[5] = Reserved - set to 0x00
ucData[6] = Reserved - set to 0x00
ucData[7] = Reserved - set to 0x00
```

4   Customization

This bootloader contains many advanced features that are on par with many professionally supported 3rd party bootloaders. All of the features can be turned on or off by configuring #defines in bl_config.h.

ENABLE_WATCHDOG

If defined the watchdog will be enabled and kicked in the main update loop.

ENABLE_CSM_CONTROL

If defined the bootloader will include mechanisms to lock and unlock the code security module. AES encryption libraries are used to do a seed key exchange to authenticate the host. The passwords to be programmed into flash can be found immediately below this #define. Please note that the public release of the bootloader doesn't contain the encryption library neccessary to implement this feature due to export restrictions. Please contact TI if you are interested in this feature.

ENABLE_READ

Enables the upload of flash memory contents. Applications that want their code to be secure should not define this, but keep in mind the programmed code cannot then be verified.

ENABLE_CRC_CHECK

If defined, before booting the application the bootloader will examine the ulAppSig location for a pointer to a linker generated CRC table in the application. The bootloader will then iterate though this table checking each CRC record. If all records pass then the application is booted.

ENABLE_FAILSAFE

If defined the bootloader will generate two flash entry points and relocate itself in flash during erase operations. This minimizes the critical time where if power is pulled the device becomes bricked. This is an EXTREMELY advanced feature and should only be used by those with a great deal of embedded programming experience.

ENABLE_DECRYPTION

Enables a call to DecryptData in bl_decrypt.c whenever data to be programmed is recieved. The DecryptData function is a stub that can be populated with the users own decryption routine.

ENABLE_UPDATE_CHECK

If defined before checking for a valid application, the bootloader will check a GPIO pin for a given state. If the state matches the defined state in bl_config.h the bootloader will start and wait for an update to be performed.

5   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.

The configuration options are:

`CRYSTAL_FREQ`	This defines the crystal frequency used by the microcontroller running the boot loader. This frequency is used to properlly setup PLL2 for operation with the USB controller. The user should ensure that 120MHz divided by the crystal frequency is an integer.
`APP_START_ADDRESS`	The starting address of the application. ■ This value must be defined.
`APP_END_ADDRESS`	The ending address of the application. ■ This value must be defined.
`FLASH_PAGE_SIZE`	The size of a single, erasable page in the flash. This must be a power of 2. The default value of 32KB (16K Words) represents the page size for the internal flash on all C2000 MCUs. ■ This value must be defined.
`FLASH_END_ADDR`	The starting address of flash.
`FLASH_END_ADDR`	The ending address of flash.
`USB_VENDOR_ID`	The USB vendor ID published by the DFU device. This value is the TI vendor ID. Change this to the vendor ID you have been assigned by the USB-IF.
`USB_PRODUCT_ID`	The USB device ID published by the DFU device. If you are using your own vendor ID, chose a device ID that is different from the ID you use in non-update operation. If you have sublicensed TI's vendor ID, you must use an assigned product ID here.
`USB_DEVICE_ID`	Selects the BCD USB device release number published in the device descriptor.
`USB_MAX_POWER`	Sets the maximum power consumption that the DFU device will report to the USB host in the configuration descriptor. Units are milliamps.
`USB_BUS_POWERED`	Specifies whether the DFU device reports to the host that it is self-powered (defined as 0) or bus-powered (defined as 1).

`USB_HAS_MUX`	Specifies whether the target board uses a multiplexer to select between USB host and device modes.
`USB_MUX_PIN`	Specifies the GPIO pin number used to select between USB host and device modes. Valid values are 0 through 64. ■ This value must be defined if `USB_HAS_MUX` is defined.
`USB_MUX_DEVICE`	Specifies the state of the GPIO pin required to select USB device-mode operation. Valid values are 0 (low) or 1 (high). ■ This value must be defined if `USB_HAS_MUX` is defined.
`BL_HW_INIT_FN_HOOK`	Performs application-specific low level hardware initialization on system reset. If hooked, this function will be called immediately after the boot loader code relocation completes. An application may perform any required low-level hardware initialization during this function. Note that the system clock has not been set when this function is called. Initialization that assumes the system clock is set may be performed in the `BL_INIT_FN_HOOK` function instead.
`BL_INIT_FN_HOOK`	Performs application-specific initialization on system reset. If hooked, this function will be called during boot loader initialization to perform any board- or application-specific initialization which is required. The function is called following a reset immediately after the selected boot loader peripheral has been configured and the system clock has been set.
`BL_REINIT_FN_HOOK`	Performs application-specific reinitialization on boot loader entry via SVC. If hooked, this function will be called during boot loader reinitialization to perform any board- or application-specific initialization which is required. The function is called following boot loader entry from an application, after any system clock rate adjustments have been made.
`BL_START_FN_HOOK`	Informs an application that a download is starting. If hooked, this function will be called when a firmware download is about to begin. The function is called after the first data packet of the download is received but before it has been written to flash.

`BL_PROGRESS_FN_HOOK`	Informs an application of download progress. If hooked, this function will be called periodically during a firmware download to provide progress information. The function is called after each data packet is received from the host. Parameters provide the number of bytes of data received and, in cases other than Ethernet update, the expected total number of bytes in the download (the TFTP protocol used by the Ethernet boot loader does not send the final image size before the download starts so in this case the ulTotal parameter is set to 0 to indicate that the size is unknown).
`BL_END_FN_HOOK`	Informs an application that a download has completed. If hooked, this function will be called when a firmware download has just completed. The function is called after the final data packet of the download has been written to flash.
`BL_DECRYPT_FN_HOOK`	Allows an application to perform in-place data decryption during download. If hooked, this function will be called to perform in-place decryption of each data packet received during a firmware download.

- This value takes precedence over `ENABLE_DECRYPTION`. If both are defined, the hook function defined using `BL_DECRYPT_FN_HOOK` is called rather than the previously-defined `DecryptData()` stub function.

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

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

6   Failsafe Mode

The purpose of the failsafe bootloader mode is twofold. Because the flash sector size is 0x4000kw and the bootloader is only ~0x1500kw large a tradition bootloader which protects the sector it resides in would waste a sizable piece of otherwise usable flash memory. To overcome this limitation the bootloader must copy itself into another flash sector such that its original flash sector can be erased and reused for the user application. This is the first reason for the failsafe mode.

During any bootloading scenario there exists a time during erase where the device can potentially be corrupted and caused to not boot correctly. This can happen if power if removed during an erase or program operation of Sector A (the location of the BootROM boot vector). To minimize the chances of this occuring the erase and program of sector are ordered such that an erase of Sector A is immediately followed by a reprogramming of the reset vector. This is the second reason for the failsafe mode.

To better understand the failsafe mode, lets walk through a typical erase operation. Keep in mind that while in failsafe mode the erase command's address and size arguments are ignored and the entire flash is always erased.

Erase Sectors B-G. This erases most of the application leaving only the bootloader and reset vector in flash. Device will boot normally if power is removed at this point.

Copy the bootloader into Sector G.

Erase Sector A and program reset vector to point to the failsafe entry point in flash Sector G. During this operation if power is removed the device may not boot on the next powerup. If power is removed after this operation completes successfully but before the bootloader is copied back into Sector H, then the device will boot into the failsafe mode which requires an application update.

Erase Sector H, the original location of the bootloader.

Copy the bootloader back into Sector H.

Erase Sector A and program reset vector to point to the normal entry point in flash Sector H. During this operation if power is removed the device may not boot on the next powerup. If power is removed after this operation completes successfully, then the device will boot into the normal mode and will require and application update because the pAppEntry vector will be null.

After these operations complete the flash will be completely erased (except for the area ocupied by the bootloader) and ready for the application to be programmed.

6.1   Program and Data Allocation

To accomplish the above functionality the bootloader has been carefully partitioned to ensure that it is as robust as possible. There are 3 main allocations of program code:

Program code that runs from flash in Sector H (normal boot, application check, normal memory copy).

Program code that runs from flash in Sector G (failsafe boot, failsafe memory copy).

Program code that runs from SRAM and is copied from Sector G or H depending on whether the device is in failsafe mode at the time or not.

Please refer to figure 5.1 for a graphic representation of the memory with failsafe mode enabled.
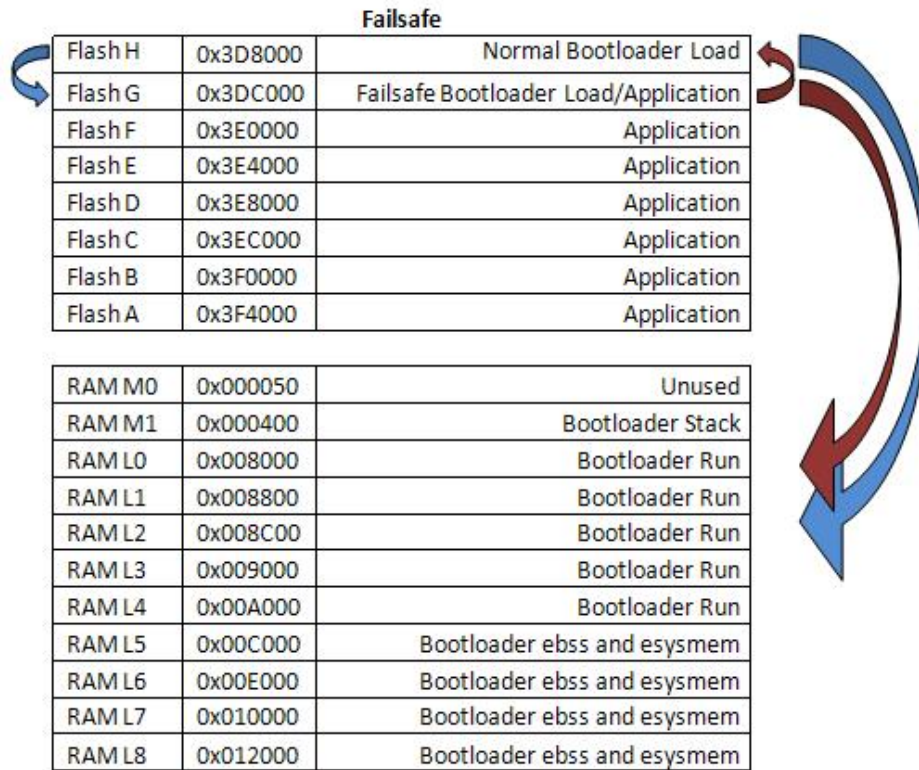
Failsafe

Flash H	0x3D8000	Normal Bootloader Load
Flash G	0x3DC000	Failsafe Bootloader Load/Application
Flash F	0x3E0000	Application
Flash E	0x3E4000	Application
Flash D	0x3E8000	Application
Flash C	0x3EC000	Application
Flash B	0x3F0000	Application
Flash A	0x3F4000	Application

RAM M0	0x000050	Unused
RAM M1	0x000400	Bootloader Stack
RAM L0	0x008000	Bootloader Run
RAM L1	0x008800	Bootloader Run
RAM L2	0x008C00	Bootloader Run
RAM L3	0x009000	Bootloader Run
RAM L4	0x00A000	Bootloader Run
RAM L5	0x00C000	Bootloader ebss and esysmem
RAM L6	0x00E000	Bootloader ebss and esysmem
RAM L7	0x010000	Bootloader ebss and esysmem
RAM L8	0x012000	Bootloader ebss and esysmem

Figure 6.1: Failsafe Bootloader Mode Memory Allocation

6.2 Memory Copies

Care must be taken when modifing the bootloader to ensure everything lines up in memory where it should. Several of the copies performed are based on the normal and failsafe copies of code being spaced 0x4000 apart. During the copy of bootloader image from Sector H to Sector G the entire bootloader image is copied up in the address space by 0x4000. The failsafe initilization functions in the rts2800_bl library expect the cinit and pinit tables to be at a location 0x4000 higher than where the original ones were linked to. When the failsafe flash entry functions are copied they are copied to 0x3DE000. If modifications are made to the bootloader that increase its size, care should be taken to ensure that the bootloader when copied to Sector G doesn't overlap the failsafe flash entry functions at 0x3DE000. Please refer to figure 5.2 for a graphical represenation of the copies that occur.
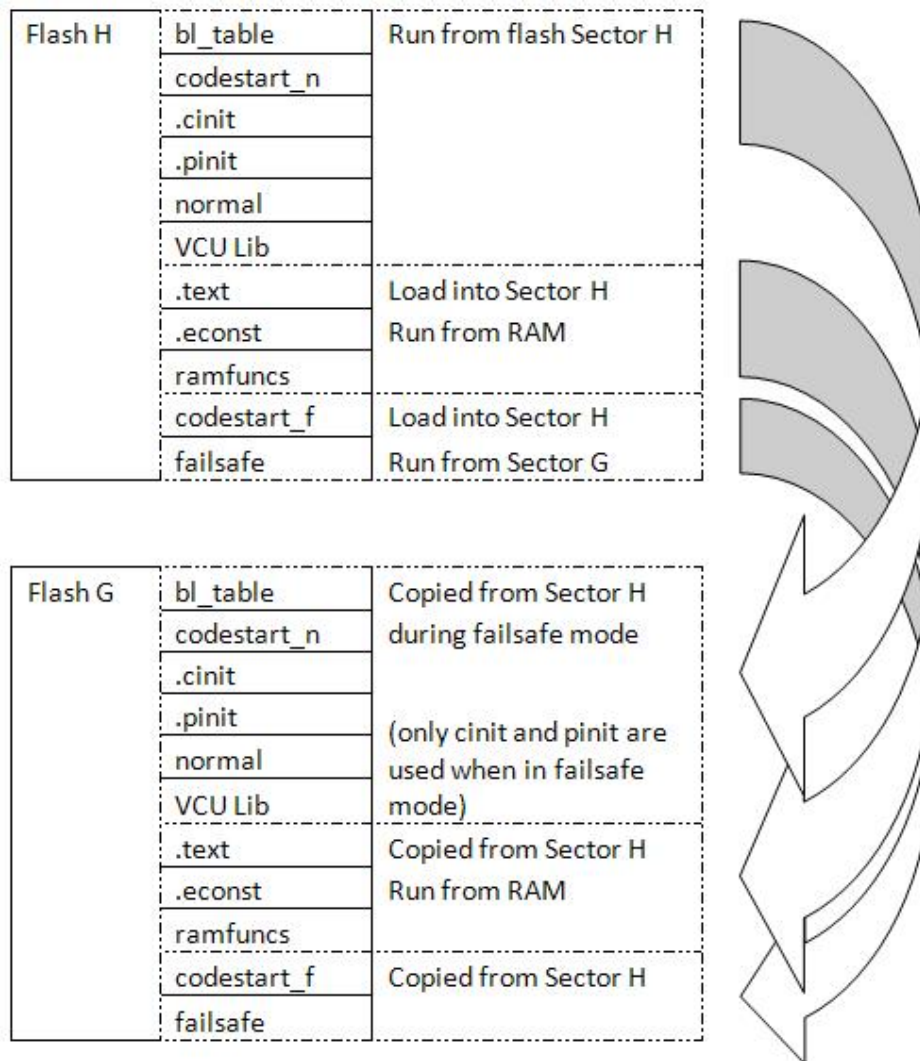
Figure 6.2: Failsafe Bootloader Memory Copies

7   Source Details

7.1   Decryption Functions

Functions

void DecryptData (unsigned char ∗pucBuffer, unsigned long ulSize)

7.1.1   Detailed Description

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

7.1.2   Function Documentation

7.1.2.1

unsigned char ∗ pucBuffer,

unsigned long ulSize) Performs an in-place decryption of downloaded data.

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.

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

Prototype:
```
DecryptData void DecryptData(
```

Returns None.

7.2   Update Check Functions

Functions

unsigned char CheckCRCTable (volatile CRC_TABLE ∗pCRCTable)

unsigned long CheckForceUpdate (void)

unsigned long CheckGPIOForceUpdate (void)

7.2.1 Detailed Description

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

7.2.2 Function Documentation

7.2.2.1

volatile CRC_TABLE ∗ pCRCTable) Checks a Linker Generated CRC table against memory

Parameters *pCRCTable* CRC Table to be checked

This function iterates through a CRC table performing the checks specified against the memory map.

Prototype:
```
CheckCRCTable unsigned char CheckCRCTable(
```

Returns Returns 1 on failure, 0 on success

Referenced by AppCheck().

7.2.2.2 unsigned long CheckForceUpdate (

)

Checks if an update is needed or is being requested.

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.

References CheckGPIOForceUpdate().

Referenced by AppCheck().

7.2.2.3 unsigned long CheckGPIOForceUpdate (

)

Checks a GPIO for a forced update.

This function checks the state of a GPIO to determine if a update is being requested.

Returns Returns a non-zero value if an update is being requested and zero otherwise.

Referenced by CheckForceUpdate().

7.3 USB Device Functions

Data Structures

void tConfigDescriptor

tLong

tShort

tString0Descriptor

tStringDescriptor

tUSBRequest

Macros

readLong(ptr)

readShort(ptr)

writeLong(ptr, value)

writeShort(ptr, value)

Functions

void AppCheck (void)

void AppUpdaterUSB (void)

void CompactBuffer (unsigned char ∗pucBuffer, unsigned long ulBufferSize)

void ConfigureUSB (void)

void ConfigureUSBInterface (void)

void CopyBLGtoH (void)

void CopyBLHtoG (void)

void FailsafeEntry (void)

void FlashAPICallback (void)

unsigned long FlashAPIIndex (unsigned long ulAddress, unsigned long ulSize)

void FlashAPIInit (void)

tBoolean FlashRangeCheck (unsigned long ulStart, unsigned long ulLength)

void HandleConfigChange (unsigned long ulInfo)

void HandleDisconnect (void)

void HandleEP0Data (unsigned long ulSize)

void HandleRequestDnloadIdle (tUSBRequest ∗pUSBRequest)

void HandleRequestDnloadSync (tUSBRequest ∗pUSBRequest)

void HandleRequestError (tUSBRequest ∗pUSBRequest)

void HandleRequestIdle (tUSBRequest ∗pUSBRequest)

void HandleRequestManifestSync (tUSBRequest ∗pUSBRequest)

void HandleRequests (tUSBRequest ∗pUSBRequest)

void HandleRequestUploadIdle (tUSBRequest ∗pUSBRequest)

void HandleReset (void)

void HandleSetAddress (void)

void main (void)

tBoolean ProcessDFUDnloadCommand (tDFUDownloadHeader ∗pcCmd, unsigned long ulSize)

void ProgramBLFailsafeVector (void)

void ProgramBLNormalVector (void)

void SendDFUState (void)

void SendDFUStatus (void)

tBoolean SendUploadData (unsigned short usLength, tBoolean bAppendHeader)

void UpdaterUSB (void)

__interrupt void USB0DeviceIntHandler (void)

void USBBLInit (void)

void USBBLSendDataEP0 (unsigned char ∗pucData, unsigned long ulSize)

void USBBLStallEP0 (void)

void WatchdogEnable (void)

7.3.1 Detailed Description

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

7.3.2 Macro Definition Documentation

7.3.2.1

ptr) This define is used to read data from a tLong variable. This is a is a workaround specific to C2000 devices.

Referenced by FlashRangeCheck(), HandleRequestDnloadIdle(), HandleRequestUploadIdle(), HandleSetAddress(), and ProcessDFUDnloadCommand().

7.3.2.2

ptr) This define is used to read data from a tShort variable. This is a is a workaround specific to C2000 devices.

Referenced by HandleRequestDnloadIdle(), HandleRequestIdle(), HandleRequests(), HandleRequestUploadIdle(), and ProcessDFUDnloadCommand().

7.3.2.3

ptr,

value) This define is used to write data to a tLong variable. This is a is a workaround specific to C2000 devices.

Referenced by ConfigureUSBInterface().

7.3.2.4

ptr,

value) This define is used to write data to a tShort variable. This is a is a workaround specific to C2000 devices.

Referenced by ConfigureUSBInterface(), and SendUploadData().

7.3.3 Data Structure Documentation

7.3.3.1 tConfigDescriptor

Definition:
```
typedef struct
{
```

```
        unsigned char bLength;
        unsigned char bDescriptorType;
        tShort 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.

7.3.3.2   tLong

Definition:

```
    typedef struct
    {
        tShort LSW;
        tShort MSW;
    }
    tLong
```

Members:

> ***LSW***
>
> ***MSW***

Description:

> This struct is used to ensure that data passed up from the driver layer is interpretted correctly by the protocol stack. This is a workaround specifically for C2000 devices.

7.3.3.3 tShort

Definition:
```
typedef struct
{
    unsigned short LSB;
    unsigned short MSB;
}
tShort
```

Members:
> *LSB*
> *MSB*

Description:
> This struct is used to ensure that data passed up from the driver layer is interpretted correctly by the protocol stack. This is a workaround specifically for C2000 devices.

7.3.3.4 tString0Descriptor

Definition:
```
typedef struct
{
    unsigned char bLength;
    unsigned char bDescriptorType;
    tShort 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).

7.3.3.5 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.

7.3.3.6   tUSBRequest

Definition:

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

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of TI information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

TI products are not authorized for use in safety-critical applications (such as life support) where a failure of the TI product would reasonably be expected to cause severe personal injury or death, unless officers of the parties have executed an agreement specifically governing such use. Buyers represent that they have all necessary expertise in the safety and regulatory ramifications of their applications, and acknowledge and agree that they are solely responsible for all legal, regulatory and safety-related requirements concerning their products and any use of TI products in such safety-critical applications, notwithstanding any applications-related information or support that may be provided by TI. Further, Buyers must fully indemnify TI and its representatives against any damages arising out of the use of TI products in such safety-critical applications.

TI products are neither designed nor intended for use in military/aerospace applications or environments unless the TI products are specifically designated by TI as military-grade or "enhanced plastic." Only products designated by TI as military-grade meet military specifications. Buyers acknowledge and agree that any such use of TI products which TI has not designated as military-grade is solely at the Buyer's risk, and that they are solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI products are neither designed nor intended for use in automotive applications or environments unless the specific TI products are designated by TI as compliant with ISO/TS 16949 requirements. Buyers acknowledge and agree that, if they use any non-designated products in automotive applications, TI will not be responsible for any failure to meet such requirements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		**Applications**	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DLP® Products	www.dlp.com	Broadband	www.ti.com/broadband
DSP	dsp.ti.com	Digital Control	www.ti.com/digitalcontrol
Clocks and Timers	www.ti.com/clocks	Medical	www.ti.com/medical
Interface	interface.ti.com	Military	www.ti.com/military
Logic	logic.ti.com	Optical Networking	www.ti.com/opticalnetwork
Power Mgmt	power.ti.com	Security	www.ti.com/security
Microcontrollers	microcontroller.ti.com	Telephony	www.ti.com/telephony
RFID	www.ti-rfid.com	Video & Imaging	www.ti.com/video
RF/IF and ZigBee® Solutions	www.ti.com/lprf	Wireless	www.ti.com/wireless

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265
Copyright © 2024, Texas Instruments Incorporated