

# Application Update Using the USB Device Firmware Upgrade Class

## Application Note



# Copyright

Copyright © 2009 Texas Instruments, Inc. All rights reserved. Stellaris and StellarisWare are registered trademarks of Texas Instruments. ARM and Thumb are registered trademarks, and Cortex is a trademark of ARM Limited. Other names and brands may be claimed as the property of others.

Texas Instruments  
108 Wild Basin, Suite 350  
Austin, TX 78746  
Main: +1-512-279-8800  
Fax: +1-512-279-8879  
<http://www.luminarymicro.com>



# Table of Contents

Introduction .....	4
DFU Overview .....	4
Device Operating Modes .....	4
DFU Descriptors .....	5
Device Descriptor.....	5
Configuration Descriptor .....	5
Interface Descriptor.....	5
DFU Functional Descriptor.....	5
DFU Requests .....	7
DFU State Machine .....	8
Stellaris DFU Binary Protocol .....	9
Stellaris DFU Binary Protocol Query.....	10
Deviations from the DFU Specification .....	15
DFU Library for Windows.....	15
Window Messages.....	16
Data Structures.....	16
API Functions.....	18
Conclusion .....	32
References .....	32

## Introduction

Use of the standard USB Device Firmware Upgrade (DFU) class on Stellaris® USB-enabled microcontrollers offers a convenient and fast method of replacing main application images in microcontroller flash on boards configured to operate as USB devices. This application note provides a brief overview of the DFU class, describes the Stellaris implementation in the USB boot loader (boot\_usb) application, and also describes the host-side “Imdfu” dynamic link library for Windows.

## DFU Overview

The USB DFU class defines mechanisms that can be used by USB devices to write new firmware application images to their internal storage and, optionally, to return the current firmware image to the host. The class specifies a functional descriptor that the DFU device must return as part of its configuration descriptor, a set of class-specific requests, and a state machine which controls the download or upload operations.

The basic DFU specification defines the following operations:

- Firmware image download to a device (write operations)
- Firmware image upload from a device (read operations)
- Device status and state queries

It does not, however, specify the following:

- Address selection for downloaded or uploaded images
- The ability to erase blocks of flash
- The ability to check that areas of memory are erased
- Methods to query the target device storage parameters (for example, flash size and writeable region addresses)

While these operations are not explicitly mentioned in the specification, they are typically supported by DFU-device manufacturers via device-specific commands embedded in downloaded binary data. In this implementation, care has been taken to ensure that a host-side application which is unaware of these binary commands can still download a suitably wrapped binary file to the device using nothing more than standard DFU download requests. More details of this are provided in “Stellaris DFU Binary Protocol” on page 9.

## Device Operating Modes

Unlike most device classes, the DFU class specifies two operating modes for the device: Run Time mode and DFU mode, each of which publishes a different set of USB descriptors to the host. In Run Time mode, the device operates using its normal USB class descriptors (for example, a printer operates as a printer and a mass storage device offers mass storage device services), but also publishes an additional interface descriptor and functional descriptor publicizing the fact that it is DFU-capable. In this mode, no actual firmware upload or download is supported, but the DFU specification defines how a host may signal the device to indicate that it should switch into the second mode in preparation for DFU operation.

In DFU mode, the device no longer publishes its standard device descriptors but, instead, reports only its DFU capabilities, all of which are accessed via a single interface and using the control endpoint, endpoint 0. The device descriptor published in this mode often contains a different product ID (PID) than the descriptor published in Run Time mode, thus ensuring that the connected USB host only loads the DFU device driver when the device is in DFU mode. In this mode, firmware download (write) and upload (read) operations are possible. On exit from DFU mode, the device typically reboots and runs the main application image, reverting to Run Time mode operation.

**Note:** The remainder of this document refers to DFU mode only.

## DFU Descriptors

In DFU mode, the device publishes the following USB descriptors:

- Device
- Configuration
- Interface
- DFU Functional

### Device Descriptor

A standard device descriptor is published with the vendor ID assigned to the manufacturer of the device (0x1CBE for Stellaris examples) and a product ID which is typically different from the one published during Run Time mode operation. For the Stellaris USB boot loader, product ID 0x00FF is used.

The bDeviceClass and bDeviceSubclass fields of the device descriptor are each set to 0x00 indicating that the class and subclass are defined at the interface level.

### Configuration Descriptor

A standard configuration descriptor is published by the device. In DFU mode, the bNumInterfaces field must be set to 1 indicating that a single interface is present.

### Interface Descriptor

A single interface descriptor is published with the DFU interface identified by bInterfaceClass set to 0xFE, bInterfaceSubClass set to 0x01, and bInterfaceProtocol set to 0x02. The DFU specification allows different programmable areas of the device memory to be identified through the use of multiple alternate settings but the Stellaris USB boot loader only supports a single alternate setting for the interface.

The interface supports no endpoints (bNumEndpoints is set to 0) since all DFU communication is carried out via the default control endpoint, endpoint 0.

### DFU Functional Descriptor

The only device class-specific descriptor published by the device is the DFU functional descriptor which offers information on the DFU capabilities of the device and the size of data blocks which may

be written to or read from the device during download or upload operations. Table 1-1 shows the DFU functional descriptor fields.

**Table 1-1. DFU Functional Descriptor Fields**

Offset	Field	Size	Value	Description
0	bLength	1	0x09	Size of this descriptor in bytes.
1	bDescriptorType	1	0x21	DFU FUNCTIONAL descriptor type.
2	bmAttributes	1	Bit Mask	<p>This field provides bit flags indicating DFU-specific device capabilities.</p> <p>Bit [7:4] – Reserved</p> <p>[3] – bitWillDetach (1) indicates that the device will detach and reattach automatically on receipt of the DFU_DETACH request whereas (0) indicates that the host must issue a USB reset after the DFU_DETACH request.</p> <p>Bit [2] – bManifestationTolerant (0) indicates that the device must be reset after completion of a firmware download operation whereas (1) indicates that the device remains responsive and capable of receiving further DFU requests after a download ends.</p> <p>Bit [1] – bitCanUpload (1) indicates that the device is capable of uploading data to the host and (0) indicates that this is not supported.</p> <p>Bit [0] – bitCanDnload (1) indicates that the device supports firmware download operations and (0) indicates that download is not supported.</p> <p>In the Stellaris® USB boot loader, this field is set to 0x07 to indicate that the device is “manifestation tolerant” and that it can perform both upload and download operations.</p>
3	wDetachTimeOut	2	Number	This field defines the number of milliseconds that the device waits for a USB reset after receiving the DFU_DETACH request. The Stellaris USB boot loader does not support DFU_DETACH (since this is a Run Time mode request) and the field is set to 0xFFFF.
5	wTransferSize	2	Number	Defines the maximum number of bytes that the device can accept or provide in each control endpoint transaction. For the Stellaris® USB boot loader, this value is 1024.
7	bcdDFUVersion	2	0x110	Identifies this device as DFU 1.1 compliant.

## DFU Requests

All communication between the USB host and DFU device is made via a group of seven DFU-defined requests sent using the default control endpoint, endpoint 0. Table 1-2 shows the DFU-defined requests.

**Table 1-2. DFU-Defined Requests**

bmRequestType	bRequest	wValue	wIndex	wLength	Data
00100001b	DFU_DETACH	wTimeout	Interface	Zero	None
00100001b	DFU_DNLOAD	wBlockNum	Interface	Length	Firmware data
10100001b	DFU_UPLOAD	Zero	Interface	Length	Firmware data
10100001b	DFU_GETSTATUS	Zero	Interface	6	Status
00100001b	DFU_CLRSTATUS	Zero	Interface	Zero	None
10100001b	DFU_GETSTATE	Zero	Interface	1	State
00100001b	DFU_ABORT	Zero	Interface	Zero	None

### ■ DFU\_DETACH

This request is not supported by the Stellaris USB boot loader since it applies only to Run Time mode operation. The request instructs the device that it should switch into DFU mode either immediately (if the device DFU functional descriptor bitWillDetach attribute bit is set) or following the next USB reset.

### ■ DFU\_DNLOAD

The DFU\_DNLOAD request is used to send binary firmware data to the device. This may include target-specific commands in addition to the firmware data since the DFU specification does not define the actual content or meaning of the payload of the download request. The size of data passed with the request must be less than or equal to the wTransferSize specified in the DFU functional descriptor (1024 for the Stellaris implementation). After each transfer, the host must issue a DFU\_GETSTATUS and wait until the previous transfer has been processed prior to sending the next block of data.

### ■ DFU\_UPLOAD

The DFU\_UPLOAD request is used to retrieve the existing firmware image from the device. As in the download case, each transfer can be up to wTransferSize bytes. DFU\_UPLOAD may also be used to retrieve device-specific information in response to a command embedded in a previous DFU\_DNLOAD request.

### ■ DFU\_GETSTATUS

The DFU\_GETSTATUS request returns information on the state of the DFU device and also acts as a synchronization mechanism during download operations. The request must be made following each DFU\_DNLOAD and no more download requests issued until the returned state indicates that the device can accept more download data.

This request returns a six-byte structure containing the fields shown in Table 1-3.

**Table 1-3. DFU\_GETSTATUS Fields**

Offset	Field	Size	Value	Description
0	bStatus	1	Number	An indication of the device status as a result of the execution of the most recent request. The value 0 indicates that no error condition is present.
1	bwPollTimeout	3	Number	The minimum time that the host should wait before sending another DFU_GETSTATUS while polling for completion of a DFU_DNLOAD operation.
4	bState	1	Number	The state that the device will transition to immediately after sending this response.
5	iString	1	Index	The index of any status description string that the device may offer. 0 indicates that no string is available.

■ **DFU\_CLRSTATUS**

In cases where DFU\_GETSTATUS has reported an error (non-zero value) in the bStatus field, this request must be sent to clear the error condition before the host can issue any further download or upload requests.

■ **DFU\_GETSTATE**

The DFU\_GETSTATE request is similar to DFU\_GETSTATUS in that it returns the current state of the device. Unlike DFU\_GETSTATUS, however, it does not cause any state machine transitions and only returns the current device state and not the status information indicating the source of any error.

■ **DFU\_ABORT**

The DFU\_ABORT request will return the device to IDLE state and abort any partially-complete upload or download operation.

## DFU State Machine

The DFU specification defines not only the collection of requests that a DFU device must support but also a detailed state machine that the device must implement. For each of the states, the specification defines the action to be performed on receipt of each of the DFU requests and also various USB events, and also the state that is to be transitioned to following that request or stimulus. The Stellaris USB boot loader implementation is designed to closely follow the state machine definition from the specification, making it straightforward to understand the code after reading the specification. A couple of deviations from the specification do exist to facilitate download command extensions and these are described later in this document. The following states are used while the device is operating in DFU mode:

■ **dfuIDLE**

The device enters dfuIDLE state whenever it first enters DFU mode and on completion of a download or upload operation (unless the device is not “manifestation tolerant” – see dfuMANIFEST state for more details). In this state, it is ready to start a new operation.



**■ dfuDNLOAD-SYNC**

Once a DFU\_DNLOAD request has been received, the device enters this state and remains here until DFU\_GETSTATUS is received at which point it will transition to dfuDNLOAD-IDLE assuming the download request processing has completed.

**■ dfuDNLOAD-IDLE**

When a previous download request has completed and the device is ready to accept another transfer of binary data, this state is entered.

**■ dfuUPLOAD-IDLE**

Following a DFU\_UPLOAD request, this device transitions from dfuIDLE to this state and remains there until all the requested upload data has been transmitted to the host. In this state, further DFU\_UPLOAD requests may be issued by the host to retrieve subsequent data blocks. When all data has been transferred, the device transitions back to dfuIDLE.

**■ dfuERROR**

At any time when an error occurs, the device transitions into dfuERROR state. A DFU\_GETSTATUS request will result in details of the error being returned to the host. To clear the error, DFU\_CLRSTATUS must be sent which will result in the device transitioning back to dfuIDLE in preparation for the start of a new operation.

**■ dfuMANIFEST-SYNC**

Once a download operation completes and all data has been received by the device (as indicated by the host sending a DFU\_DNLOAD request with 0 in the wLength field of the request structure), this state is entered. The next DFU\_GETSTATUS request causes the Stellaris USB boot loader to transition back to dfuIDLE (since it is “manifestation tolerant”).

**■ dfuMANIFEST**

Once a download operation is completed and the host has sent a DFU\_GETSTATUS request, a device which is not “manifestation tolerant” will enter this state during which it will finalize the programming of the new firmware. Since the Stellaris USB boot loader is “manifestation tolerant” it does not support states dfuMANIFEST or dfuMANIFEST-WAIT-RESET. At the end of a download, the device transitions to dfuMANIFEST-SYNC and, on completion of programming, directly back to dfuIDLE.

**■ dfuMANIFEST-WAIT-RESET**

This state is used by devices which are not manifestation tolerant to indicate that a download has completed and that the device is waiting for the host to issue a USB reset and cause the device to boot the new firmware.

## **Stellaris DFU Binary Protocol**

The Stellaris DFU boot loader implementation supports several commands which may be sent to the target device to perform operations which are outside the scope of the existing DFU specification. This protocol is defined in such a way, however, that a host application which is unaware of it may still be used to download and upload firmware images. Using protocol commands, however, the application may access additional features such as the ability to erase specific regions of flash, query device parameters or download binary data to particular addresses.

Each command in the Stellaris DFU binary protocol is sent to the target device as a DFU\_DNLOAD request with an 8 byte payload. The first byte of the payload is a command identifier and the

following bytes are command-specific. The Stellaris device expects that any DFU\_DNLOAD request received while in state dfuIDLE will contain a command header. In other states, however, commands are not parsed, thus allowing a host application unaware of the command protocol to download a correctly formatted firmware image in multiple transfers without the need to inject commands into the download stream. In this case “correctly formatted” means that the image has been wrapped with a DFU suffix and a command prefix structure containing the 8 byte command indicating that binary data is being sent. This wrapper can be added by using the “dfuwrap” command line application which is included in StellarisWare™ releases for USB-capable Stellaris evaluation kits such as the EK-LM3S3748.

## Stellaris DFU Binary Protocol Query

Since the Stellaris DFU binary protocol involves sending data via DFU\_DNLOAD requests, it is desirable to be able to determine whether or not a target device supports the protocol before attempting to use it. Sending such commands to a device which does not expect them could cause corruption of the device firmware so a special request is supported by the Stellaris DFU boot loader allowing a client to determine whether the protocol is supported. It is assumed that a correct response to this request, rather than a stall, indicates that the protocol is supported.

The request is an IN request to the DFU interface on the control endpoint, endpoint 0 containing specific values for the bRequest (USB\_DFU\_REQUEST\_LUMINARY, 0x42), wValue (REQUEST\_LUMINARY\_VALUE, 0x23) and wLength (sizeof(tDFUQueryLuminaryProtocol), 4) parameters. A device supporting the Stellaris DFU binary protocol is expected to respond to a request containing these known values with a 4 byte structure (tDFUQueryLuminaryProtocol) containing two marker pattern bytes (LM\_DFU\_PROTOCOL\_MARKER, 0x4C4D) and a version number. Receipt of the marker bytes by the host indicates that the protocol is supported and the version number (currently LM\_DFU\_PROTOCOL\_VERSION\_1, 0x0001) can be used to determine the set of features supported.

The protocol support request is shown in Table 1-4.

**Table 1-4. Protocol Support Request**

bmRequest	Type bRequest	wValue	wIndex	wLength	Data
10100001b	0x42	0x23	Interface	4	A structure containing a 2 byte marker and 2 byte version number is returned by the device.

A device supporting the protocol must respond with the following packed structure:

```
typedef struct
{
    unsigned short usMarker;           // Set to LM_DFU_PROTOCOL_MARKER
    unsigned short usVersion;          // Set to LM_DFU_PROTOCOL_VERSION_1
}
tDFUQueryLuminaryProtocol;
```

## Stellaris DFU Binary Protocol Commands

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 dfu\_IDLE state to contain a command header but will not look for command unless the state is dfu\_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 the usbdfu.h header file. 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 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 SYSTCTL register DID1.  
//  
unsigned long ulPartInfo;  
//  
// Information on the part class and revision as read  
// from SYSTCTL 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
```

## Deviations from the DFU Specification

The Stellaris USB boot loader contains a couple of small deviations from the DFU specification. It is not expected that these differences will materially impact host software accessing the device without knowledge of the Stellaris DFU binary protocol.

- **State dfuDNBUSY is not supported.** After each DFU\_DNLOAD request, the device transitions to dfuDNLOAD\_SYNC state and remains there until the previously downloaded data has been processed at which point it transitions back to dfuDNLOAD\_IDLE state. This change was made to reduce the image size since it means that timers do not need to be supported. The specification suggests that dfuDNBUSY state basically results when the host sends DFU\_GETSTATUS too frequently while the device is programming a block of flash yet the Stellaris implementation is capable of responding to DFU\_GETSTATUS while programming is ongoing.
- **The device will transition back to state dfuIDLE on completion of a DFU\_DNLOAD request which contained a Stellaris-specific command other than DFU\_CMD\_PROG rather than transitioning into dfuDNLOAD\_IDLE state.** If the previous DFU\_DNLOAD request contained binary data to be written to flash, the state transitions to dfuDNLOAD\_IDLE as required by the specification. By doing this, we guarantee that the device is ready to accept a new command once a previous command is complete yet we maintain the expected state transitions while flashing an image to the microcontroller.
- **The Stellaris USB boot loader does not support run time states (appIDLE and appDETACH).** If an application wishes to support both run time and DFU modes, it must include the software necessary to respond to at least the DFU\_DETACH request and transfer control to the boot loader.

## DFU Library for Windows

The DFU Library for Windows is a DLL offering a high-level application interface allowing communication with attached USB-DFU equipped Stellaris devices. Functions are provided to allow the host application to determine the number and type of DFU devices currently attached to the system, to query DFU-related parameters from those devices, to download new firmware images, to upload existing images and to erase sections of the DFU device flash memory.

This DLL is included as part of the device driver which is installed when the DFU device is first placed on the host's USB bus. The device driver is part of the SW-EK-USB-windrivers package which can be downloaded from the software updates page on the [www.luminarymicro.com/products/software\\_updates.html](http://www.luminarymicro.com/products/software_updates.html) web site.

The dfuprog example application which is part of the SW-EK-USB-win "Windows-side examples for USB kits" package is also downloadable from the web site and makes use of the LMUSB DLL interface as does the latest version of the LM Flash Programmer.

## Window Messages

Various functions in the LMDFU library allow the caller to provide a window handle which receives periodic status messages during time-consuming operations. The library sends the messages shown in Table 1-5.

**Table 1-5. Windows Messages**

Message	WPARAM	LPARAM	Description
WM_DFU_DOWNLOAD	Total transfer count at completion.	tLMDFUHandle	A download operation has started.
WM_DFU_UPLOAD	Total transfer count at completion.	tLMDFUHandle	An upload operation has started.
WM_DFU_VERIFY	Total transfer count at completion.	tLMDFUHandle	A verify operation has started.
WM_DFU_ERASE	Total transfer count at completion.	tLMDFUHandle	An erase operation has started.
WM_DFU_PROGRESS	Transfers completed	tLMDFUHandle	Provides progress information on the current operation. The completed transfer count will increment to the value provided in the download, upload, verify or erase message sent at the start of the operation.
WM_DFU_ERROR	0	tLMDFUHandle	An error was detected and the current operation has been aborted.
WM_DFU_COMPLETE	0	tLMDFUHandle	The previous operation has completed successfully.

## Data Structures

### tLMDFUDeviceInfo

This structure is returned from a call to LMDFUDeviceOpen and provides information about the opened device.

```
typedef struct
{
    unsigned short  usVID;
    unsigned short  usPID;
    unsigned short  usDevice;
    unsigned short  usDetachTimeOut;
    unsigned short  usTransferSize;
    unsigned char   ucDFUAttributes;
    unsigned char   ucManufacturerString;
    unsigned char   ucProductString;
    unsigned char   ucSerialString;
    unsigned char   ucDFUInterfaceString;
    bool            bSupportsLuminaryExtensions;
    bool            bDFUMode;
    unsigned long   ulPartNumber;
    char            cRevisionMajor;
    char            cRevisionMinor;
}
tLMDFUDeviceInfo;
```



usVID	Vendor ID published in the device descriptor.
usPID	Product ID published in the device descriptor.
usDevice	BCD device release number published in the device descriptor.
usDetachTimeOut	Device detach timeout published in the DFU functional descriptor.
usTransferSize	Maximum number of bytes that the device can accept per control-write transaction as published in the DFU functional descriptor.
ucDFUAttributes	Contains the device attributes from the DFU functional descriptor. Bits in this value indicate whether the device is capable of upload and/or download, and whether it is able to continue communication after completing a download (whether it is “manifestation tolerant”).
ucManufacturerString	Index of the manufacturer name string published in the device descriptor.
ucProductString	Index of the product name string published in the device descriptor.
ucSerialString	Index of the serial number string published in the device descriptor.
ucDFUInterfaceString	Index of the interface string published in the DFU interface descriptor.
bSupportsLuminaryExtensions	Set to true if the DFU device supports the Stellaris DFU binary protocol or false otherwise.
bDFUMode	Set to true if the device is currently operating in DFU mode or false if operating in runtime mode.
ulPartNumber	Hexadecimal number indicating the Stellaris part number on the target device. This value is only valid if bSupportsLuminaryExtensions is true. For example, if the device contains an lm3s3748 part, this field will be set to 0x3748.
cRevisionMajor	Hexadecimal number indicating the major revision of the Stellaris part on the target device. Major revision ‘A’ is represented by 0x00, ‘B’ by 0x01 and so on. This value is only valid if bSupportsLuminaryExtensions is true.
cRevisionMinor	Hexadecimal number indicating the minor revision of the Stellaris part on the target device. This value is only valid if bSupportsLuminaryExtensions is true.

### **tLMDFUPParams**

This structure is returned in response to a call to LMDFUPParamsGet and provides information on the writable area of the flash address space on the device.

```
typedef struct
{
    unsigned short usFlashBlockSize;
    unsigned short usNumFlashBlocks
    unsigned long ulFlashTop;
```

```
    unsigned long ulAppStartAddr;
}
tLMDFUParams;
```

usFlashBlockSize	Size of an individual flash block on the device.
usNumFlashBlocks	Total number of blocks of flash in the device. The total flash size is (usNumFlashBlocks * usFlashBlockSize).
ulFlashTop	Address 1 byte above the highest location that the DFU boot loader can access. This will typically be at the very top of flash but some implementations may reserve some space at the top of flash for parameter storage in which case this will be reflected in ulFlashTop.
ulAppStartAddr	Lowest address that the DFU boot loader can write or erase.

## API Functions

### Functions

Click any of the function names in the list below to go directly to the full description of that function.

- [LMDFUInit on page 18](#)
- [LMDFUDeviceOpen on page 19](#)
- [LMDFUDeviceClose on page 20](#)
- [LMDFUDeviceStringGet on page 20](#)
- [LMDFUDeviceASCIIStringGet on page 21](#)
- [LMDFUParamsGet on page 22](#)
- [LMDFUIsValidImage on page 23](#)
- [LMDFUDownload on page 24](#)
- [LMDFUDownloadBin on page 26](#)
- [LMDFUErase on page 27](#)
- [LMDFUBlackCheck on page 28](#)
- [LMDFUUpload on page 29](#)
- [LMDFUStatusGet on page 31](#)
- [LMDFUErrorStringGet on page 32](#)

### LMDFUInit

Initializes the DLL for use by the host application.

### Prototype

```
tLMDFUErr
LMDFUInit(void)
```

### Parameters

None.

### Description

This function must be called by the host application before any other entry point in the library. It initializes the global data required to access DFU devices.

### Returns

Returns `DFU_OK` on success.

## LMDFUDeviceOpen

Opens a DFU device and returns a handle to the caller.

### Prototype

```
tLMDFUErr  
LMDFUDeviceOpen(int iDeviceIndex,  
tLMDFUDeviceInfo *psDevInfo,  
tLMDFUHandle *phHandle)
```

### Parameters

iDeviceIndex	Zero-based index indicating which DFU-capable device is to be opened.
psDevInfo	Structure which will be filled in with information relating to the DFU device which has been opened.
phHandle	Points to storage which will be written with a valid DFU device handle on success.

### Description

This function opens a DFU device and returns information on the device state and capabilities.

Note that this function will open DFU devices that are currently in runtime mode. The caller must ensure that a device is in DFU mode prior to making any further requests which are not supported in runtime mode. Currently, this DLL does not contain a function to cause a switch from runtime to DFU mode and it is assumed that DFU devices used with this DLL will have some means to allow the user to start them in DFU mode. For Stellaris evaluation kits, the DFU mode may be entered by resetting the board with the select button pressed.

The returned psDevInfo structure contains a flag, bDFUMode which is set to true if the device is operating in DFU mode or false if operating in runtime mode.

Handles allocated by this function must be closed using a matching call to LMDFUDeviceClose().

LMDFUDeviceOpen() and LMDFUDeviceClose() may be used to enumerate DFU devices on the bus by opening devices in a loop, incrementing iDeviceIndex until DFU\_ERR\_NOT\_FOUND is returned.

### Returns

Returns one of the following error codes:

- DFU\_OK if the operation completed without errors.
- DFU\_ERR\_INVALID\_ADDR if psDevInfo or phHandle is NULL.
- DFU\_ERR\_MEMORY if it was not possible to allocate system memory to support the request.
- DFU\_ERR\_NOT\_FOUND if a DFU device with index iDeviceIndex could not be found attached to the system.
- DFU\_ERR\_UNKNOWN if the device was found but an error was reported while trying to open it.

## LMDFUDeviceClose

Closes a DFU device and, optionally, returns it to its runtime configuration.

### Prototype

```
tLMDFUErr  
LMDFUDeviceClose(tLMDFUHandle hHandle,  
                 bool bReset)
```

### Parameters

hHandle	Handle of the DFU device which is to be closed. This handle was previously returned from a call to LMDFUDeviceOpen().
bReset	Indicates whether to leave the device in DFU mode (false) or reset it and return to runtime mode (true).

### Description

This function closes a DFU device previously opened using a call to LMDFUDeviceOpen(). If the bReset parameter is true, the device is reset and returned to its run time mode of operation. If bReset is false, the device is left in DFU mode and can be reopened again without the need to perform a mode switch.

### Returns

Returns one of the following error codes:

- DFU\_OK if the operation completed without errors.
- DFU\_ERR\_HANDLE if hHandle is NULL.

## LMDFUDeviceStringGet

Retrieves a Unicode string descriptor from a DFU device.

### Prototype

```
tLMDFUErr  
LMDFUDeviceStringGet(tLMDFUHandle hHandle,  
                    unsigned char ucStringIndex,  
                    unsigned short usLanguageID,  
                    char *pcString,  
                    unsigned short *pusStringLength)
```

### Parameters

hHandle	Handle of the DFU device from which the string is to be queried. This handle was previously returned from a call to LMDFUDeviceOpen().
ucStringIndex	Index of the string that is to be queried and found in either a USB device descriptor or the tLMDFUDeviceInfo structure returned following a call to LMDFUDeviceOpen().
usLanguageID	ID of the language for the returned string. This ID must exist in the device's string table.

pcString	Points to a buffer into which the returned Unicode string will be written.
pusStringLen	Points to a variable which contains the size of the pcString buffer (in bytes) on entry. If the string is read, this variable is updated to show the number of bytes written into the pcString buffer.

### Description

This function retrieves Unicode strings from the DFU device. If the requested string is available in the chosen language, `DFU_OK` is returned and the string is written into the supplied pcString buffer and \*pusStringLen updated to provide the length of the returned string in bytes.

### Returns

Returns one of the following error codes:

- `DFU_OK` if the operation completed without errors.
- `DFU_ERR_HANDLE` if hHandle is NULL.
- `DFU_ERR_INVALID_ADDR` if pcString or pusStringLen is NULL.
- `DFU_ERR_NOT_FOUND` if the string requested cannot be found.

### LMDFUDeviceASCIIStringGet

Retrieves an ASCII string descriptor from a DFU device.

### Prototype

```
tLMDFUErr
LMDFUDeviceASCIIStringGet(tLMDFUHandle hHandle,
                           unsigned char ucStringIndex,
                           char *pcString,
                           unsigned short *pusStringLen)
```

### Parameters

hHandle	Handle of the DFU device from which the string is to be queried. This handle was previously returned from a call to <code>LMDFUDeviceOpen()</code> .
ucStringIndex	Index of the string that is to be queried and found in either a USB device descriptor or the <code>tLMDFUDeviceInfo</code> structure returned following a call to <code>LMDFUDeviceOpen()</code> .
pcString	Points to a buffer into which the returned Unicode string will be written.
pusStringLen	Points to a variable which contains the size of the pcString buffer on entry. If the string is read, this variable is updated to show the number of characters written into the pcString buffer.

### Description

This function retrieves a string descriptor from the DFU device using the first language supported by the device. If the requested string is available, the Unicode descriptor is converted to 8-bit

ASCII and written into the buffer pointed to by pcString and \*pusStringLength is updated to provide the length of the returned string.

### Returns

Returns one of the following error codes:

- DFU\_OK if the operation completed without errors.
- DFU\_ERR\_HANDLE if hHandle is NULL.
- DFU\_ERR\_INVALID\_ADDR if pcString or pusStringLength is NULL.
- DFU\_ERR\_NOT\_FOUND if the string requested cannot be found.

### LMDFUPParamsGet

Query DFU download-related parameters from a Stellaris device.

### Prototype

```
tLMDFUErr  
LMDFUPParamsGet(tLMDFUHandle hHandle,  
                tLMDFUPParams *psParams)
```

### Parameters

- |          |  |
|----------|--|
| hHandle  | Handle of the DFU device whose parameters are being queried. This handle was previously returned from a call to LMDFUDeviceOpen(). |
| psParams | Points to a structure which will be written with the device's DFU parameters.  |

### Description

This function allows an application to query various parameters related to the Stellaris DFU device that it has opened. These parameters are the size of a flash block, the number of blocks the device supports and the address region that is writeable.

The bottom of the flash address range is always considered read-only since this contains the DFU boot loader code itself. Depending upon the application, a section of flash at the top of the range may also be reserved for persistent application parameter storage and marked read-only to the DFU host.

### Returns

Returns one of the following error codes:

- DFU\_OK if the operation completed without errors.
- DFU\_ERR\_HANDLE if hHandle is NULL.
- DFU\_ERR\_INVALID\_ADDR if psParams is NULL.
- DFU\_ERR\_UNSUPPORTED if the target DFU device does not support Stellaris DFU binary protocol extensions.
- DFU\_ERR\_TIMEOUT if the control transaction times out.

- `DFU_ERR_STALL` if the device stalls the request indicating an error.
- `DFU_ERR_DISCONNECTED` if the device has been disconnected.
- `DFU_ERR_UNKNOWN` if an unexpected error is reported by the device.

## **LMDFUIsValidImage**

Determines whether the supplied firmware image is a correctly formatted DFU image.

### **Prototype**

```
tLMDFUErr  
LMDFUIsValidImage(tLMDFUHandle hHandle,  
                  unsigned char *pcDFUImage,  
                  unsigned long ulImageLen,  
                  bool *pbLuminaryFormat)
```

### **Parameters**

- |                               |   |
|-------------------------------|---|
| <code>hHandle</code>          | Handle of the DFU device which the firmware image is destined to be used with. This handle was previously returned from a call to <code>LMDFUDeviceOpen()</code> .  |
| <code>pcDFUImage</code>       | Points to the first byte of the firmware image to check.  |
| <code>ulImageLen</code>       | Number of bytes in the image data pointed to by <code>pcDFUImage</code> .   |
| <code>pbLuminaryFormat</code> | Pointer which will be written to true if the supplied data appears to start with a valid Luminary Micro DFU prefix structure. If the data ends in a valid DFU suffix structure but does not contain the Luminary Micro prefix, this value will be written to false. |

### **Description**

This function checks a provided binary to determine whether it is a correctly formatted DFU image or not. A valid image contains a 16 byte suffix with a checksum and the IDs of the intended target device. An image is considered to be valid if the following criteria are met:

- The CRC of the whole block calculates to 0 (i.e. the CRC of all but the last 4 bytes equals the CRC stored in the last 4 bytes).
- The "DFU" suffix marker exists at the correct place at the end of the data block.
- The vendor and products IDs read from the expected positions in the DFU suffix match the VID and PID of the device whose handle is passed.

Additionally, if these conditions are met, the data is examined for the presence of a Luminary Micro DFU prefix. This structure contains the address at which to flash the image and also the length of the payload.

The `pbLuminaryFormat` pointer is written to true if the following additional criteria are met:

- The first byte of the data block is 0x01.

- The unsigned long in bytes 4 through 7 of the image matches the value of the length of the block minus the DFU suffix (length read from the suffix itself) and Luminary Micro prefix (8 bytes).
- The unsigned short in bytes 2 and 3 of the image forms a sensible flash block number (address / 1024) for a Stellaris device.

## Returns

Returns one of the following error codes:

- `DFU_OK` if the operation completed without errors and the passed image contains a valid DFU suffix structure.
- `DFU_ERR_HANDLE` if `hHandle` is `NULL`.
- `DFU_ERR_INVALID_ADDR` if `pcDFUImage` or `pbLuminaryFormat` is `NULL`.
- `DFU_ERR_UNSUPPORTED` if the device VID and PID do not match those in the DFU image suffix structure.
- `DFU_ERR_INVALID_FORMAT` if the firmware image provided does not appear to contain a valid DFU suffix structure.

## LMDFUDownload

Downloads a DFU-formatted firmware image to a target device.

## Prototype

```
tLMDFUErr
LMDFUDownload(tLMDFUHandle hHandle,
               unsigned char *pcDFUImage,
               unsigned long ulImageLen,
               bool bVerify,
               bool bIgnoreIDs,
               HWND hwndNotify)
```

## Parameters

<code>hHandle</code>	Handle of the DFU device to which a new firmware image is to be downloaded. This handle was previously returned from a call to <code>LMDFUDeviceOpen()</code> .
<code>pcDFUImage</code>	Pointer to the first byte of the DFU-formatted firmware image to download.
<code>ulImageLen</code>	Length of the firmware image pointed to by <code>pcDFUImage</code> . This is the length of the whole image including the Luminary Micro prefix and DFU suffix structures.
<code>bVerify</code>	Should be set to true if the download is to be verified (by reading back the image and checking it against the original data) or false if verification is not necessary.
<code>bIgnoreIDs</code>	Should be set to true if the DFU image is to be downloaded regardless of the fact that the DFU suffix contains a VID or PID that differs from the target



device. If set to false, the call will fail with `DFU_ERR_UNSUPPORTED` if the device VID and PID do not match the values in the DFU suffix.

**hwndNotify** Handle of a window to which periodic notifications will be sent indicating the progress of the operation. If NULL, no status notifications will be sent.

## Description

This function downloads a DFU-formatted binary to the device. A valid binary contains both the standard DFU footer suffix structure and also a Luminary Micro prefix informing the device of the address to which the image is to be written. If the data passed does not appear to contain this information, `DFU_ERR_INVALID_FORMAT` will be returned and the image will not be written to the device flash.

This function is synchronous and will not return until the operation is complete. To receive periodic status updates during the operation, a window handle may be provided. This window will receive `WM_DFU_PROGRESS` messages during the download operation allowing an application to update its user interface accordingly.

To flash a pure binary image without the DFU suffix or Luminary Micro prefix, use `LMDFUDownloadBin()` instead of this function.

Note that the Luminary Micro prefix structure contains the address at which the image is to be flashed so no parameter exists here to provide this information.

## Returns

Returns one of the following error codes:

- `DFU_OK` if the operation completed without errors.
- `DFU_ERR_DNLOAD_FAIL` if the device reported an error during image download.
- `DFU_ERR_VERIFY_FAIL` if `bVerify` is true and the image read back after download does not match the image originally sent.
- `DFU_ERR_CANT_VERIFY` if the device is not manifestation tolerant and does not return to idle state after the download completes.
- `DFU_ERR_HANDLE` if `hHandle` is NULL.
- `DFU_ERR_INVALID_ADDR` if `pcDFUImage` is NULL.
- `DFU_ERR_UNSUPPORTED` if the target DFU device does not support download operations or, if `blgnoreIDs` is false and the device VID and PID do not match those in the DFU image suffix structure.
- `DFU_ERR_INVALID_FORMAT` if the firmware image provided does not appear to contain a valid DFU suffix structure.
- `DFU_ERR_TIMEOUT` if the control transaction times out.
- `DFU_ERR_STALL` if the device stalls the request indicating an error.

- `DFU_ERR_DISCONNECTED` if the device has been disconnected.
- `DFU_ERR_UNKNOWN` if an unexpected error is reported by the device.

## **LMDFUDownloadBin**

Downloads a binary firmware image to a target device.

### **Prototype**

```
tLMDFUErr  
LMDFUDownloadBin(tLMDFUHandle hHandle,  
                 unsigned char *pcBinaryImage,  
                 unsigned long ulImageLen,  
                 unsigned long ulStartAddr,  
                 bool bVerify,  
                 HWND hwndNotify)
```

### **Parameters**

<code>hHandle</code>	Handle of the DFU device to which a new firmware image is to be downloaded. This handle was previously returned from a call to <code>LMDFUDeviceOpen()</code> .
<code>pcBinaryImage</code>	Pointer to the first byte of the binary firmware image to download. This image must not contain a Luminary Micro DFU prefix structure.
<code>ulImageLen</code>	Length of the firmware image pointed to by <code>pcBinaryImage</code> .
<code>ulStartAddr</code>	Flash address at which the image is to be written. If this parameter is set to 0 and the target device supports Stellaris DFU binary extensions, the binary image will be downloaded to the currently-configured application start address.
<code>bVerify</code>	Should be set to true if the download is to be verified (by reading back the image and checking it against the original data) or false if verification is not necessary.
<code>hwndNotify</code>	Handle of a window to which periodic notifications will be sent indicating the progress of the operation. If NULL, no status notifications will be sent.

### **Description**

This function downloads a pure binary image containing no DFU suffix or Luminary Micro header to the device at an address supplied by the caller.

This function is synchronous and will not return until the operation is complete. To receive periodic status updates during the operation, a window handle may be provided. This window will receive `WM_DFU_PROGRESS` messages during the download operation allowing an application to update its user interface accordingly.

To flash a DFU-formatted image, use `LMDFUDownload()` instead of this function.

## Returns

Returns one of the following error codes:

- `DFU_OK` if the operation completed without errors.
- `DFU_ERR_DNLOAD_FAIL` if the device reported an error during image download.
- `DFU_ERR_VERIFY_FAIL` if `bVerify` is true and the image read back after download does not match the image originally sent.
- `DFU_ERR_CANT_VERIFY` if the device does not return to idle state after the download completes.
- `DFU_ERR_HANDLE` if `hHandle` is `NULL`.
- `DFU_ERR_INVALID_ADDR` if `pcBinaryImage` is `NULL` or `ulStartAddr` does not coincide with the start of a flash block or any part of the image would fall outside the range of flash addresses that are writeable.
- `DFU_ERR_UNSUPPORTED` if the target DFU device does not support Stellaris DFU binary protocol extensions.
- `DFU_ERR_TIMEOUT` if the control transaction times out.
- `DFU_ERR_STALL` if the device stalls the request indicating an error.
- `DFU_ERR_DISCONNECTED` if the device has been disconnected.
- `DFU_ERR_UNKNOWN` if an unexpected error is reported by the device.

## LMDFUErase

Erases a section of the device flash.

## Prototype

```
tLMDFUErr  
LMDFUErase(tLMDFUHandle hHandle,  
            unsigned long ulStartAddr,  
            unsigned long ulEraseLen,  
            bool bVerify,  
            HWND hwndNotify)
```

## Parameters

<code>hHandle</code>	Handle of the DFU device whose flash is to be erased. This handle was previously returned from a call to <code>LMDFUDeviceOpen()</code> .
<code>ulStartAddr</code>	Address of the first byte of flash to be erased. This must correspond to a flash block boundary, typically multiples of 1024 bytes. If this parameter is set to 0 the entire writeable flash region will be erased.
<code>ulEraseLen</code>	Number of bytes of flash to erase. This must be a multiple of the flash block size, typically 1024 bytes.

bVerify	Should be set to true if the erase is to be verified (by reading back the flash blocks and ensuring that all bytes contain 0xFF) or false if verification is not necessary.
hwndNotify	Handle of a window to which periodic notifications will be sent indicating the progress of the operation. If NULL, no status notifications will be sent.

### Description

This function erases a section of the device flash and, optionally, checks that the resulting area has been correctly erased before returning.

The function is synchronous and will not return until the operation is complete. To receive periodic status updates during the operation, a window handle may be provided. This window will receive WM\_DFU\_PROGRESS messages during the erase operation allowing an application to update its user interface accordingly.

The start address provided must correspond to the start of a flash block within the writeable address region and the length must indicate an integral number of blocks. The block size, number of blocks and writeable region addresses can be determined by calling `LMDFUParamsGet()`.

### Returns

Returns one of the following error codes:

- `DFU_OK` if the operation completed without errors.
- `DFU_ERR_DNLOAD_FAIL` if bVerify is true and the blank check following erase failed.
- `DFU_ERR_HANDLE` if hHandle is NULL.
- `DFU_ERR_INVALID_ADDR` if ulStartAddr does not coincide with the start of a flash block.
- `DFU_ERR_INVALID_SIZE` if ulEraseLen is not a multiple of the flash block size.
- `DFU_ERR_UNSUPPORTED` if the target DFU device does not support Stellaris DFU binary protocol extensions.
- `DFU_ERR_TIMEOUT` if the control transaction times out.
- `DFU_ERR_STALL` if the device stalls the request indicating an error.
- `DFU_ERR_DISCONNECTED` if the device has been disconnected.
- `DFU_ERR_UNKNOWN` if an unexpected error is reported by the device.

### LMDFUBlankCheck

Checks a section of the device flash to ensure that it has been erased.

### Prototype

```
tLMDFUErr  
LMDFUBlankCheck(tLMDFUHandle hHandle,  
                unsigned long ulStartAddr,
```

```
unsigned long ulLen)
```

**Parameters**

hHandle	Handle of the DFU device whose flash is to be checked. This handle was previously returned from a call to <code>LMDFUDeviceOpen()</code> .
ulStartAddr	Address of the first byte of flash to check. This must correspond to a flash block boundary, typically multiples of 1024 bytes. If this parameter is set to 0 the entire writeable flash region will be checked.
ulLen	Number of bytes of flash to check. This must be a multiple of 4.

**Description**

This function checks a region of the device flash and reports whether or not it is blank (with all bytes containing value 0xFF).

The function is synchronous and will not return until the operation is complete.

The start address provided must correspond to the start of a flash block within the writeable address region. Writeable region addresses and block size can be determined by calling `LMDFUParamsGet()`.

**Returns**

Returns one of the following error codes:

- `DFU_OK` if the operation completed without errors.
- `DFU_ERR_DNLOAD_FAIL` if the region described by `ulStartAddr` and `ulLen` is not completely blank.
- `DFU_ERR_HANDLE` if `hHandle` is `NULL`.
- `DFU_ERR_INVALID_ADDR` if `ulStartAddr` is not a multiple of 1024.
- `DFU_ERR_INVALID_SIZE` if `ulLen` is not a multiple of 4.
- `DFU_ERR_UNSUPPORTED` if the target DFU device does not support Stellaris DFU binary protocol extensions.
- `DFU_ERR_TIMEOUT` if the control transaction times out.
- `DFU_ERR_STALL` if the device stalls the request indicating an error.
- `DFU_ERR_DISCONNECTED` if the device has been disconnected.
- `DFU_ERR_UNKNOWN` if an unexpected error is reported by the device.

**LMDFUUpload**

Reads back a section of the device flash.

**Prototype**

```
tLMDFUErr  
LMDFUUpload(tLMDFUHandle hHandle,  
            unsigned char *pcBuffer,  
            unsigned long ulStartAddr,  
            unsigned long ulImageLen,  
            bool bRaw,  
            HWND hwndNotify)
```

**Parameters**

hHandle	Handle of the DFU device whose flash image is to be read. This handle was previously returned from a call to <code>LMDFUDeviceOpen()</code> .
pcBuffer	Points to a buffer of at least <code>ullImageLen</code> bytes into which the returned data will be written. If <code>bRaw</code> is set to false, the buffer must be 24 bytes longer than the actual data requested to accommodate the DFU prefix and suffix which are added during the upload process.
ulStartAddr	Address of the first byte of flash to be read.
ullImageLen	Number of bytes of flash to read. If <code>bRaw</code> is set to false, this length must be increased by 24 bytes to accommodate the DFU prefix and suffix added during the upload process.
bRaw	Indicates whether the returned image will be wrapped in a Luminary Micro prefix and DFU standard suffix. If false, the wrappers will be omitted and the raw data returned. If true, the wrappers will be included allowing the returned image to be written to a device later by calling <code>LMDFUDownload()</code> .
hwndNotify	Handle of a window to which periodic notifications will be sent indicating the progress of the operation. If NULL, no status notifications will be sent.

**Description**

This function reads back a section of the device flash into a buffer supplied by the caller. The data returned may be either raw data containing no DFU control prefix and suffix or a DFU-wrapped image suitable for later download via a call to `LMDFUDownload()`. If a DFU-wrapped image is requested, the buffer pointed to by `pcBuffer` must be 24 bytes larger than the number of bytes of device flash which is to be read. For example, to read 1024 bytes of flash wrapped as a DFU image, `ullImageLen` must be set to  $(1024 + 24)$  and `pcBuffer` allocated accordingly.

The function is synchronous and will not return until the operation is complete. To receive periodic status updates during the operation, a window handle may be provided. This window will receive `WM_DFU_PROGRESS` messages during the erase operation allowing an application to update its user interface accordingly.

**Returns**

Returns one of the following error codes:

- `DFU_OK` if the operation completed without errors.

- `DFU_ERR_HANDLE` if `hHandle` is `NULL`.
- `DFU_ERR_INVALID_ADDR` if `ulStartAddr` is not a multiple of 1024.
- `DFU_ERR_INVALID_SIZE` if the buffer provided is too small to hold the image prefix and suffix (when `bRaw` is `false`).
- `DFU_ERR_UNSUPPORTED` if the target DFU device does not support Stellaris DFU binary protocol extensions.
- `DFU_ERR_TIMEOUT` if the control transaction times out.
- `DFU_ERR_STALL` if the device stalls the request indicating an error.
- `DFU_ERR_DISCONNECTED` if the device has been disconnected.
- `DFU_ERR_UNKNOWN` if an unexpected error is reported by the device.

## LMDFUStatusGet

Queries the current status of the DFU device.

### Prototype

```
tLMDFUErr  
LMDFUStatusGet(tLMDFUHandle hHandle,  
               tDFUStatus *pStatus )
```

### Parameters

<code>hHandle</code>	Handle of the DFU device whose status is being requested. This handle was previously returned from a call to <code>LMDFUDeviceOpen()</code> .
<code>pStatus</code>	Points to storage which will be written with the DFU status returned by the device.

### Description

This call may be made to receive detailed error status from the connected DFU device. The status value returned is an error code as defined in section 6.2.1 of the USB Device Firmware Upgrade Specification.

### Returns

Returns one of the following error codes:

- `DFU_OK` if the operation completed without errors.
- `DFU_ERR_HANDLE` if `hHandle` is `NULL`.
- `DFU_ERR_INVALID_ADDR` if `pStatus` is `NULL`.
- `DFU_ERR_TIMEOUT` if the control transaction times out.
- `DFU_ERR_STALL` if the device stalls the request indicating an error.

- `DFU_ERR_DISCONNECTED` if the device has been disconnected.
- `DFU_ERR_UNKNOWN` if an unexpected error is reported by the device.

## **LMDFUErrorStringGet**

Returns an ASCII string describing the passed error code.

### **Prototype**

```
char *  
LMDFUErrorStringGet( tLMDFUErr eError )
```

### **Parameters**

`eError` is the error code whose description is being queried.

### **Description**

This function is provided for debug and diagnostic purposes. It maps the return code from an LMDFU function into a human readable string suitable for, for example, debug trace output.

### **Returns**

Returns a pointer to an ASCII string describing the return code.

## **Conclusion**

USB Device Firmware Upgrade allows the high speed of the USB interface to be used to quickly update application binary images on a target device. Making use of the Luminary Micro DFU DLL on Windows, DFU functionality can very easily be added to new or existing host applications supporting those target devices.

## **References**

Documents used in the generation of this application note include:

- *Universal Serial Bus Device Class Specification for Device Firmware Upgrade Version 1.1*  
[http://www.usb.org/developers/devclass\\_docs/DFU\\_1.1.pdf](http://www.usb.org/developers/devclass_docs/DFU_1.1.pdf)
- *Universal Serial Bus Specification, Revision 2.0*  
[http://www.usb.org/developers/docs/usb\\_20\\_040908.zip](http://www.usb.org/developers/docs/usb_20_040908.zip)
- *Stellaris® Boot Loader User's Guide* – An element of each StellarisWare® Firmware Development Package downloadable from [http://www.luminarymicro.com/products/software\\_updates.html](http://www.luminarymicro.com/products/software_updates.html)



## 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

Amplifiers	<a href="http://amplifier.ti.com">amplifier.ti.com</a>
Data Converters	<a href="http://dataconverter.ti.com">dataconverter.ti.com</a>
DLP® Products	<a href="http://www.dlp.com">www.dlp.com</a>
DSP	<a href="http://dsp.ti.com">dsp.ti.com</a>
Clocks and Timers	<a href="http://www.ti.com/clocks">www.ti.com/clocks</a>
Interface	<a href="http://interface.ti.com">interface.ti.com</a>
Logic	<a href="http://logic.ti.com">logic.ti.com</a>
Power Mgmt	<a href="http://power.ti.com">power.ti.com</a>
Microcontrollers	<a href="http://microcontroller.ti.com">microcontroller.ti.com</a>
RFID	<a href="http://www.ti-rfid.com">www.ti-rfid.com</a>
RF/IF and ZigBee® Solutions	<a href="http://www.ti.com/lprf">www.ti.com/lprf</a>

### Applications

Audio	<a href="http://www.ti.com/audio">www.ti.com/audio</a>
Automotive	<a href="http://www.ti.com/automotive">www.ti.com/automotive</a>
Broadband	<a href="http://www.ti.com/broadband">www.ti.com/broadband</a>
Digital Control	<a href="http://www.ti.com/digitalcontrol">www.ti.com/digitalcontrol</a>
Medical	<a href="http://www.ti.com/medical">www.ti.com/medical</a>
Military	<a href="http://www.ti.com/military">www.ti.com/military</a>
Optical Networking	<a href="http://www.ti.com/opticalnetwork">www.ti.com/opticalnetwork</a>
Security	<a href="http://www.ti.com/security">www.ti.com/security</a>
Telephony	<a href="http://www.ti.com/telephony">www.ti.com/telephony</a>
Video & Imaging	<a href="http://www.ti.com/video">www.ti.com/video</a>
Wireless	<a href="http://www.ti.com/wireless">www.ti.com/wireless</a>

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265  
Copyright © 2009, Texas Instruments Incorporated