Universal Serial Bus

Device Class Specification for

Device Firmware Upgrade

Version 1.1
Aug 5, 2004

Intellectual Property Disclaimer

THIS SPECIFICATION IS PROVIDED “AS IS” WITH NO WARRANTIES WHATSOEVER INCLUDING ANY WARRANTY OF MERCHANTABILITY, FITNESS FOR ANY PARTICULAR PURPOSE, OR ANY WARRANTY OTHERWISE ARISING OUT OF ANY PROPOSAL, SPECIFICATION, OR SAMPLE.

A LICENSE IS HEREBY GRANTED TO REPRODUCE AND DISTRIBUTE THIS SPECIFICATION FOR INTERNAL USE ONLY. NO OTHER LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY OTHER INTELLECTUAL PROPERTY RIGHTS IS GRANTED OR INTENDED HEREBY.

AUTHORS OF THIS SPECIFICATION DISCLAIM ALL LIABILITY, INCLUDING LIABILITY FOR NFRINGEMENT OF PROPRIETARY RIGHTS, RELATING TO IMPLEMENTATION OF INFORMATION IN THIS SPECIFICATION. AUTHORS OF THIS SPECIFICATION ALSO DO NOT WARRANT OR REPRESENT THAT SUCH IMPLEMENTATION(S) WILL NOT INFRINGE SUCH RIGHTS.

Contributors

Trenton Henry	SMSC
David Rivenburg	SMSC
Dan Stirling	MCCI
Bob Nathan	NCR
Bill Belknap	NCR
Mats Webjörn	MCCI
Bill Dellar	Systech
Neil Winchester	SMSC
Steve McGowan	Intel
Tom Green	Microsoft
Ivo Bettens	Symbol Technologies
Mark McCoy	Anchor Chips
John Stafford	Systech
Keith Gudger	Atmel
Greg Kroah-Hartman	PSC Inc.

Contents

1. INTRODUCTION 7

1.1 Related Documents 7

1.2 Terms and Abbreviations 7

B. DFU FILE SUFFIX

1. Introduction

This document describes proposed requirements and specifications for Universal Serial Bus (USB) devices that support the Device Firmware Upgrade (DFU) capability.

1.1 Related Documents

The following related documents are available from WWW.USB.ORG:

- Universal Serial Bus Specification 1.0, January 19, 1996
- Universal Serial Bus Common Class Specification 1.0, December 16, 1997

1.2 Terms and Abbreviations

The meanings of some words have been stretched to suit the purposes of this document. These definitions are intended to clarify the discussions that follow.

DFU	(n) Device Firmware Upgrade
Firmware	(n) Executable software stored in a write-able, nonvolatile memory on a USB device.
Upgrade	(v) To overwrite the firmware of a device.
	(n)(1) The act of overwriting the firmware of a device.
	(n)(2) New firmware intended to replace a device's existing firmware.
Download	(v) To transmit information from host to device.
Upload	(v) To transmit information from device to host.

2. Overview

Users that have purchased USB devices require the ability to upgrade the firmware of those devices with improved versions as they become available from manufacturers. Device Firmware Upgrade is the mechanism for accomplishing that task. Any class of USB device can exploit this capability by supporting the requirements specified in this document.

This document focuses on installing product enhancements and patches to devices that are already deployed in the field. Other potential uses for the firmware upgrade capability are beyond the scope of this document.

Because it is impractical for a device to concurrently perform both DFU operations and its normal run-time activities, those normal activities must cease for the duration of the DFU operations. Doing so means that the device must change its operating mode; i.e., a printer is **not** a printer while it is undergoing a firmware upgrade; it is a PROM programmer. However, a device that supports DFU is not capable of changing its mode of operation on its own volition. External (human or host operating system) intervention is required.

There are four distinct phases required to accomplish a firmware upgrade:

1. Enumeration: The device informs the host of its capabilities. A DFU class-interface descriptor and associated functional descriptor embedded within the device's normal run-time descriptors serves this purpose and provides a target for class-specific requests over the control pipe.
2. Reconfiguration: The host and the device agree to initiate a firmware upgrade. The host issues a USB reset to the device, and the device then exports a second set of descriptors in preparation for the Transfer phase. This deactivates the run-time device drivers associated with the device and allows the DFU driver to reprogram the device's firmware unhindered by any other communications traffic targeting the device.
3. Transfer: The host transfers the firmware image to the device. The parameters specified in the functional descriptor are used to ensure correct block sizes and timing for programming the nonvolatile memories. Status requests are employed to maintain synchronization between the host and the device.
4. Manifestation: Once the device reports to the host that it has completed the reprogramming operations, the host issues a USB reset to the device. The device re-enumerates and executes the upgraded firmware.

The device's vendor ID, product ID, and serial number can be used to form an identifier used by the host operating system to uniquely identify the device. However, certain operating systems may use only the vendor and product IDs reported by a device to determine which drivers to load, regardless of the device class code reported by the device. (Host operating systems typically do not expect a device to change classes.) Therefore, to ensure that only the DFU driver is loaded, it is considered necessary to change the *idProduct* field of the device when it enumerates the DFU descriptor set. This ensures that the DFU driver will be loaded in cases where the operating system simply matches the vendor ID and product ID to a specific driver.

Note This document does not attempt to specify how a vendor might alter the device's product ID except to suggest that adding one, setting the high bit, or using FFFFh are all valid possibilities. Vendors may use any scheme that they choose.

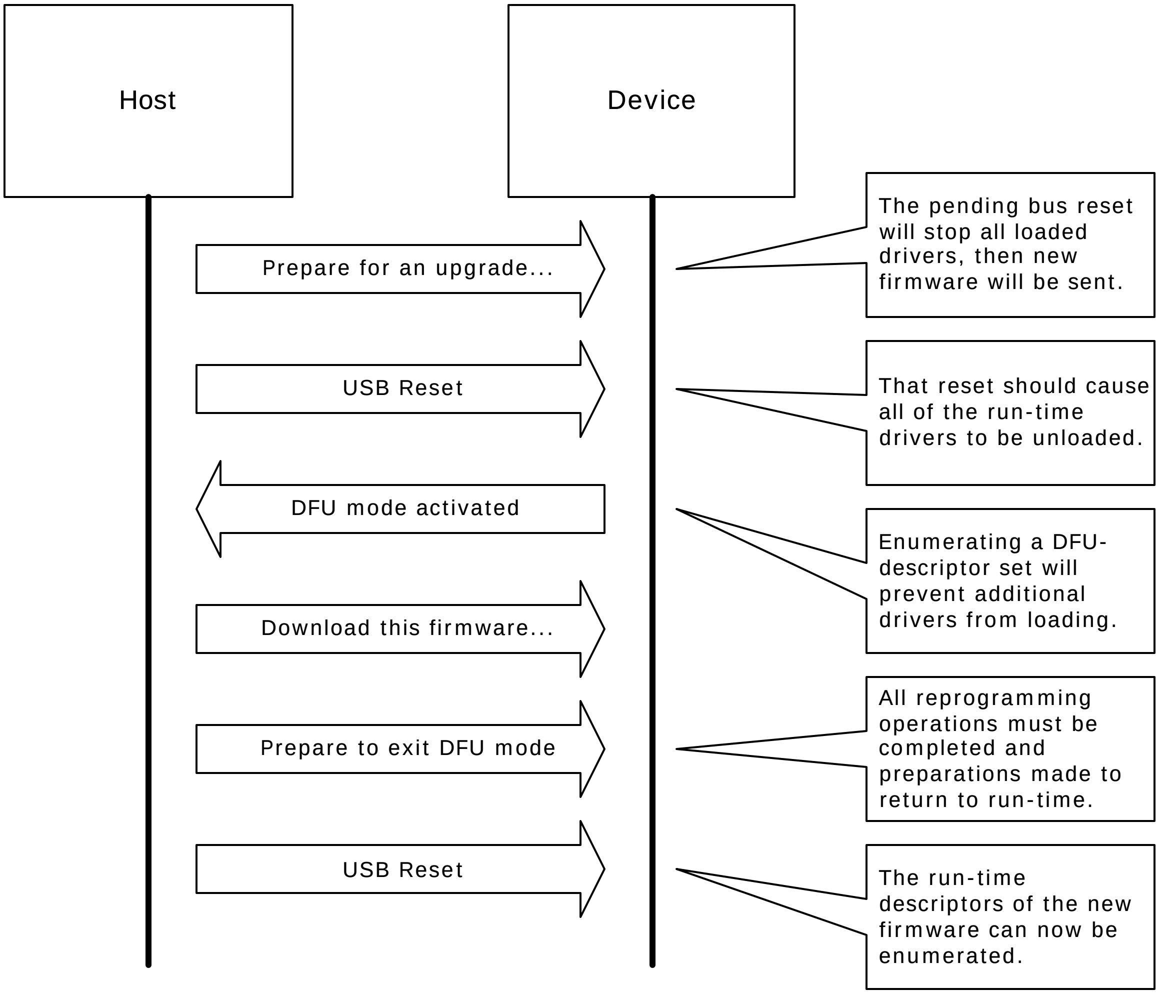
Host
Device
Prepare for an upgrade...
USB Reset
DFU mode activated
Download this firmware...
Prepare to exit DFU mode
USB Reset
The pending bus reset will stop all loaded drivers, then new firmware will be sent.
That reset should cause all of the run-time drivers to be unloaded.
Enumerating a DFU-descriptor set will prevent additional drivers from loading.
All reprogramming operations must be completed and preparations made to return to run-time.
The run-time descriptors of the new firmware can now be enumerated.

Figure 2.1 Stylized DFU session

3. Requests

A number of DFU class-specific requests are employed to accomplish the upgrade operations. The following table summarizes the DFU class-specific requests. Details concerning each of these requests are explained in subsequent sections of this document.

Table 3.1 Summary of DFU Class-Specific Requests

bmRequestType	bRequest	wValue	wIndex	wLength	Data
00100001b	DFU_DETACH	*wTimeout*	Interface	Zero	None
00100001b	DFU_DNLOAD	*wBlockNum*	Interface	Length	Firmware
10100001b	DFU_UPLOAD	Zero	Interface	Length	Firmware
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

Table 3.2 DFU Class-Specific Request Values

bRequest	Value	Protocol
DFU_DETACH	0	1
DFU_DNLOAD	1	2
DFU_UPLOAD	2	2
DFU_GETSTATUS	3	1*, 2
DFU_CLRSTATUS	4	2
DFU_GETSTATE	5	1*, 2
DFU_ABORT	6	2

* optional

4. Enumeration Phase

It is very important to note that the device exposes **two** distinct and independent descriptor sets, one each at the appropriate time:

- Run-time descriptor set
- DFU mode descriptor set

4.1 Run-Time Descriptor Set

During normal run-time operation, the device exposes its normal set of descriptors. However, the following additional descriptors are inserted within each run-time configuration that supports DFU:

- A single DFU class interface descriptor
- A single functional descriptor

4.1.1 Run-Time Device and Configuration Descriptors

The run-time descriptor set exposes the device's normal run-time device and configuration descriptors. The *bNumInterfaces* field of configuration descriptor of each configuration that supports DFU is incremented by one to accommodate the addition of the run-time DFU interface.

4.1.2 Run-Time DFU Interface Descriptor

No endpoint descriptors are present because DFU uses only the control endpoint. This provides sufficient information for the host to recognize that the device is capable of performing firmware upgrade operations. It also provides the means for initiating such operations over the default control pipe.

The DFU class interface is typically the last interface enumerated for each run-time configuration. However, there is no requirement for this interface to occupy any specific position.

Note Depending upon the device's run-time descriptors, this additional 'dangling' interface may cause some operating systems to load a DFU driver even though it is rarely used.

Table 4.1 Run-Time DFU Interface Descriptor

Offset	Field	Size	Value	Description
0	*bLength*	1	09h	Size of this descriptor, in bytes.
1	*bDescriptorType*	1	04h	INTERFACE descriptor type.
2	*bInterfaceNumber*	1	Number	Number of this interface.
3	*bAlternateSetting*	1	00h	Alternate setting. Must be zero.
4	*bNumEndpoints*	1	00h	Only the control pipe is used.
5	*bInterfaceClass*	1	FEh	Application Specific Class Code
6	*bInterfaceSubClass*	1	01h	Device Firmware Upgrade Code
7	*bInterfaceProtocol*	1	01h	Runtime protocol.
8	*iInterface*	1	Index	Index of string descriptor for this interface.

4.1.3 Run-Time DFU Functional Descriptor

This descriptor is identical for both the run-time and the DFU mode descriptor sets.

Table 4.2 DFU Functional Descriptor

Offset	Field	Size	Value	Description
0	*bLength*	1	09h	Size of this descriptor, in bytes.
1	*bDescriptorType*	1	21h	DFU FUNCTIONAL descriptor type.
2	*bmAttributes*	1	Bit mask	DFU attributes Bit 7..4: reserved Bit 3: device will perform a bus detach-attach sequence when it receives a DFU_DETACH request. The host must not issue a USB Reset. (*bitWillDetach*) 0 = no 1 = yes Bit 2: device is able to communicate via USB after Manifestation phase. (*bitManifestationTolerant*) 0 = no, must see bus reset 1 = yes Bit 1: upload capable (*bitCanUpload*) 0 = no 1 = yes Bit 0: download capable (*bitCanDnload*) 0 = no 1 = yes

3	*wDetachTimeOut*	2	Number	Time, in milliseconds, that the device will wait after receipt of the DFU_DETACH request. If this time elapses without a USB reset, then the device will terminate the Reconfiguration phase and revert back to normal operation. This represents the maximum time that the device can wait (depending on its timers, etc.). The host may specify a shorter timeout in the DFU_DETACH request.
5	*wTransferSize*	2	Number	Maximum number of bytes that the device can accept per control-write transaction.
7	*bcdDFUVersion*	2	BCD	Numeric expression identifying the version of the DFU Specification release.

4.2 DFU Mode Descriptor Set

After the host and device agree to perform DFU operations, the host re-enumerates the device. It is at this time that the device exports the DFU descriptor set, which contains:

- A DFU device descriptor
- A single configuration descriptor
- A single interface descriptor (including descriptors for alternate settings, if present)
- A single functional descriptor

These are the only descriptors that the device may expose after reconfiguration. The reason is to prevent any other device drivers from being loaded by the host operating system.

4.2.1 DFU Mode Device Descriptor

This descriptor is only present in the DFU mode descriptor set.

Table 4.3 DFU Device Descriptor

Offset	Field	Size	Value	Description
0	*bLength*	1	12h	Size of this descriptor, in bytes.
1	*bDescriptorType*	1	01h	DEVICE descriptor type.
2	*bcdUSB*	2	0100h	USB specification release number in binary coded decimal.
4	*bDeviceClass*	1	00h	See interface.
5	*bDeviceSubClass*	1	00h	See interface.

6	*bDeviceProtocol*	1	00h	See interface.
7	*bMaxPacketSize0*	1	8,16,32,64	Maximum packet size for endpoint zero.
8	*idVendor*	2	ID	Vendor ID. Assigned by the USB-IF.
10	*idProduct*	2	ID	Product ID. Assigned by manufacturer.
12	*bcdDevice*	2	BCD	Device release number in binary coded decimal.
14	*iManufacturer*	1	Index	Index of string descriptor.
15	*iProduct*	1	Index	Index of string descriptor.
16	*iSerialNumber*	1	Index	Index of string descriptor.
17	*bNumConfigurations*	1	01h	One configuration only for DFU.

4.2.2 DFU Mode Configuration Descriptor

This descriptor is identical to the standard configuration descriptor described in the USB specification version 1.0, with the exception that the *bNumInterfaces* field must contain the value 01h.

4.2.3 DFU Mode Interface Descriptor

This is the descriptor for the only interface available when operating in DFU mode. Therefore, the value of the *bInterfaceNumber* field is always zero.

Table 4.4 DFU Mode Interface Descriptor

Offset	Field	Size	Value	Description
0	*bLength*	1	09h	Size of this descriptor, in bytes.
1	*bDescriptorType*	1	04h	INTERFACE descriptor type.
2	*bInterfaceNumber*	1	00h	Number of this interface.
3	*bAlternateSetting*	1	Number	Alternate setting. *
4	*bNumEndpoints*	1	00h	Only the control pipe is used.
5	*bInterfaceClass*	1	FEh	Application Specific Class Code
6	*bInterfaceSubClass*	1	01h	Device Firmware Upgrade Code
7	*bInterfaceProtocol*	1	02h	DFU mode protocol.

8	*iInterface*	1	Index	Index of string descriptor for this interface.

* Alternate settings can be used by an application to access additional memory segments. In this case, it is suggested that each alternate setting employ a string descriptor to indicate the target memory segment; e.g., “EEPROM”. Details concerning other possible uses of alternate settings are beyond the scope of this document. However, their use is intentionally not restricted because the authors anticipate that implementers will devise additional creative uses for alternate settings.

4.2.4 DFU Functional Descriptor

This descriptor is identical to the run-time DFU functional descriptor. See 4.1.3 for details.

5. Reconfiguration Phase

An operator initiates a firmware upgrade operation by executing an application on the host. This application requires the operator to specify the device that will be upgraded and the firmware image file that will be transferred to that device. That is, the operator indicates to the application "Download **this** file into **that** device."

For more information about the file suffix that simplifies the task of ensuring that a file is compatible with a specific device, see Appendix B.

Once the operator has identified the device and supplied the filename, the host and the device must negotiate to perform the upgrade. The negotiation proceeds as follows:

- The host issues a DFU_DETACH request on the control endpoint EP0.
- The host issues a USB reset to the device.
- The device enumerates the DFU descriptor set, as described previously.

5.1 The DFU_DETACH Request

When bit 3 in *bmAttributes* (*bitWillDetach*) is set the device will generate a detach-attach sequence on the bus when it sees this request. Otherwise, the device starts a timer counting the amount of time specified, in milliseconds, in the *wDetachTimeout* field. If the device detects a USB reset while this timer is running, then DFU operating mode is enabled by the device; i.e., when USB reset signaling is detected, perform as normal unless this timer is running, in which case switch into DFU mode and stop the timer.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
00100001b	DFU_DETACH	*wTimeout**	Interface	Zero	None

* The *wTimeout* field is specified in units of milliseconds and represents the amount of time that the device should wait for the pending USB reset before giving up and terminating the operation. *wTimeout* should not contain a value larger than the value specified in *wDetachTimeout*.

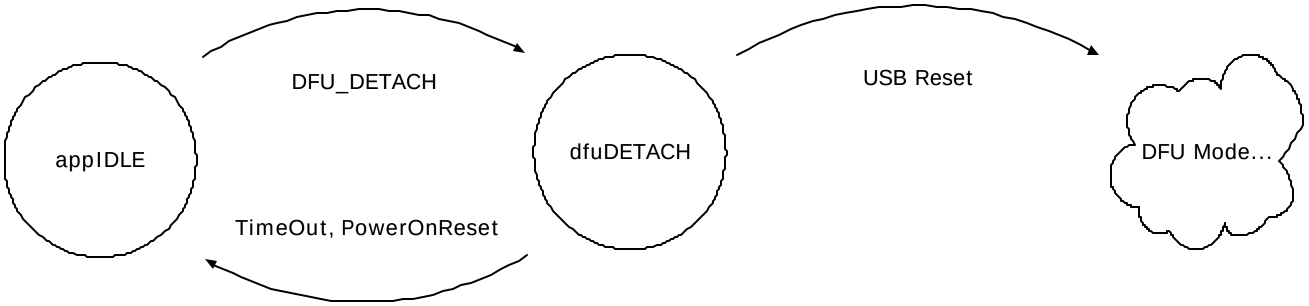

Figure 5.1 Example state transition during reconfiguration

6. Transfer Phase

The Transfer phase begins after the device has processed the USB reset and exported the DFU descriptor set. Both firmware downloads and uploads can take place during this phase.

6.1 Downloading

Firmware images for specific devices are, by definition, vendor specific. It is therefore required that target addresses, record sizes, and all other information relative to supporting an upgrade are encapsulated within the firmware image file. It is the responsibility of the device manufacturer and the firmware developer to ensure that their devices can consume these encapsulated data. With the exception of the DFU file suffix, the content of the firmware image file is irrelevant to the host. The host simply slices the firmware image file into N pieces and sends them to the device by means of control-write operations on the default control endpoint.

```
N = ((F – S) / O) + 1
```

where

F is the size of the file, specified in bytes.

S is the size of the suffix, specified in bytes.

O is the transfer size, specified in bytes.

Note The optimum transfer size, O, is in the range between *bMaxPacketSize0* and *wTransferSize*, inclusive. The actual value depends upon the host operating system.

The host continues the transfer by sending the payload packets on the control endpoint until the entire file has been transferred or the device reports an error.

The device uses the standard NAK mechanism for flow control, if necessary, while the content of its nonvolatile memories is updated. If the device detects an error, it signals the host by issuing a STALL handshake on the control endpoint. The host then sends a DFU class-specific request, called DFU_GETSTATUS, on the control endpoint to determine the nature of the problem.

There are three general mechanisms by which a device receives a firmware image from a host:

1. The first mechanism is to receive the entire image into a buffer and perform the actual programming during the Manifestation phase.
2. The second mechanism is to accumulate a block of firmware data, erase an equivalent size block of memory, and write the block into the erased memory.
3. The third mechanism is a variation of the second. In the third method, a large portion of memory is erased, and small firmware blocks are written, one at a time, into the empty memory space. This is necessary when the erasure granularity of the memory is larger than the available buffer size.

All three of these techniques are accommodated by virtue of the dynamic values specified in the *bwPollTimeout* field and closed-loop, host-driven state transitions. For more information about *bwPollTimeout*, see the DFU_GETSTATUS request. For more information about the DFU file suffix, see Appendix B.

6.1.1 DFU_DNLOAD Request

The firmware image is downloaded via control-write transfers initiated by the DFU_DNLOAD class-specific request. The device specifies the maximum number of bytes per transfer via the *wTransferSize* field of the functional descriptor. The host sends between *bMaxPacketSize0* and *wTransferSize* bytes to the device in a control-write transfer. Following each downloaded block, the host solicits the device status with the DFU_GETSTATUS request.

After the final block of firmware has been sent to the device and the status solicited, the host sends a DFU_DNLOAD request with the *wLength* field cleared to 0 and then solicits the status again. If the result indicates that the device is ready and there are no errors, then the Transfer phase is complete and the Manifestation phase begins. However, some devices may buffer the entire firmware image in volatile memory, programming the nonvolatile memories while in the dfuMANIFEST state. It is possible that some devices, during the Manifestation phase, can be rendered incapable of communicating over the USB during the reprogramming operations. The bit *bitManifestationTolerant* of the *bmAttributes* field is cleared to indicate this to the host and prevent it from sending packets to the device during the Manifestation phase.

bmRequestType	bRequest	wValue	WIndex	wLength	Data
00100001b	DFU_DNLOAD	*wBlock-Num**	Interface	Specified by USB**	Firm-ware

* The *wBlockNum* field is a block sequence number. It increments each time a block is transferred, wrapping to zero from 65,535. It is used to provide useful context to the DFU loader in the device.

** The *wLength* field indicates the total number of bytes in this transfer, according to USB version 1.0. This value should not exceed the value specified in the *wTransferSize* field.

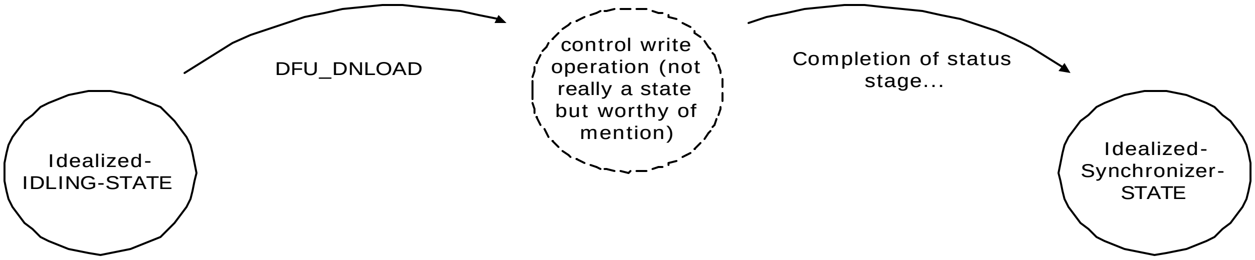
DFU_DNLOAD
control write operation (not really a state but worthy of mention)
Completion of status stage...
Idealized-IDLING-STATE
Idealized-Synchronizer-STATE

Figure 6.1 Example state transition using DFU_DNLOAD to initiate a transfer

6.1.1.1 Zero Length DFU_DNLOAD Request

The host sends a DFU_DNLOAD request with the *wLength* field cleared to 0 to the device to indicate that it has completed transferring the firmware image file. This is the final payload packet of a download operation.

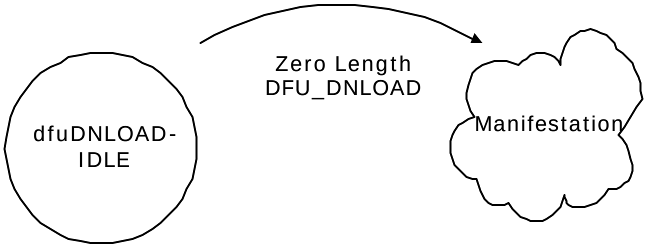

Figure 6.2 Using the zero length DFU_DNLOAD request to terminate a download

6.1.2 DFU_GETSTATUS Request

The host employs the DFU_GETSTATUS request to facilitate synchronization with the device.

BmRequestType	bRequest	wValue	wIndex	wLength	Data
10100001b	DFU_GETSTATUS	Zero	Interface	6	Status

The device responds to the DFU_GETSTATUS request with a payload packet containing the following data:

Offset	Field	Size	Value	Description
0	*bStatus*	1	Number	An indication of the status resulting from the execution of the most recent request.
1	*bwPollTimeout*	3	Number	Minimum time, in milliseconds, that the host should wait before sending a subsequent DFU_GETSTATUS request. *
4	*bState*	1	Number	An indication of the state that the device is going to enter immediately following transmission of this response. (By the time the host receives this information, this is the current state of the device.)
5	*iString*	1	Index	Index of status description in string table. **

* The purpose of this field is to allow the device to dynamically adjust the amount of time that the device expects the host to wait between the status phase of the next DFU_DNLOAD and the subsequent solicitation of the device's status via DFU_GETSTATUS. This permits the device to vary the delay depending on its need to erase memory, program the memory, etc.

** The *iString* field is used to reference a string describing the corresponding status. The device can make these strings available to the host by means of the GET_DESCRIPTOR (STRING) standard request. However, the host may reference its own string table instead.

The device status is defined as follows:

Status	Value	Suggested String
OK	0x00	No error condition is present.
errTARGET	0x01	File is not targeted for use by this device.
errFILE	0x02	File is for this device but fails some vendor-specific verification test.
errWRITE	0x03	Device is unable to write memory.
errERASE	0x04	Memory erase function failed.
errCHECK_ERASED	0x05	Memory erase check failed.
errPROG	0x06	Program memory function failed.
errVERIFY	0x07	Programmed memory failed verification.
errADDRESS	0x08	Cannot program memory due to received address that is out of range.
errNOTDONE	0x09	Received DFU_DNLOAD with *wLength* = 0, but device does not think it has all of the data yet.
errFIRMWARE	0x0A	Device's firmware is corrupt. It cannot return to run-time (non-DFU) operations.
errVENDOR	0x0B	*iString* indicates a vendor-specific error.
errUSBR	0x0C	Device detected unexpected USB reset signaling.
errPOR	0x0D	Device detected unexpected power on reset.
errUNKNOWN	0x0E	Something went wrong, but the device does not know what it was.
errSTALLEDPKT	0x0F	Device stalled an unexpected request.

The device state is defined as follows:

State	Value	Meaning
appIDLE	0	Device is running its normal application.
appDETACH	1	Device is running its normal application, has received the DFU_DETACH request, and is waiting for a USB reset.
dfuIDLE	2	Device is operating in the DFU mode and is waiting for requests.
dfuDNLOAD-SYNC	3	Device has received a block and is waiting for the host to solicit the status via DFU_GETSTATUS.
dfuDNBUSY	4	Device is programming a control-write block into its nonvolatile memories.
dfuDNLOAD-IDLE	5	Device is processing a download operation. Expecting DFU_DNLOAD requests.
dfuMANIFEST-SYNC	6	Device has received the final block of firmware from the host and is waiting for receipt of DFU_GETSTATUS to begin the Manifestation phase; or device has completed the Manifestation phase and is waiting for receipt of DFU_GETSTATUS. (Devices that can enter this state after the Manifestation phase set *bmAttributes* bit *bitManifestationTolerant* to 1.)
dfuMANIFEST	7	Device is in the Manifestation phase. (Not all devices will be able to respond to DFU_GETSTATUS when in this state.)
dfuMANIFEST-WAIT-RESET	8	Device has programmed its memories and is waiting for a USB reset or a power on reset. (Devices that must enter this state clear *bitManifestationTolerant* to 0.)
dfuUPLOAD-IDLE	9	The device is processing an upload operation. Expecting DFU_UPLOAD requests.
dfuERROR	10	An error has occurred. Awaiting the DFU_CLRSTATUS request.

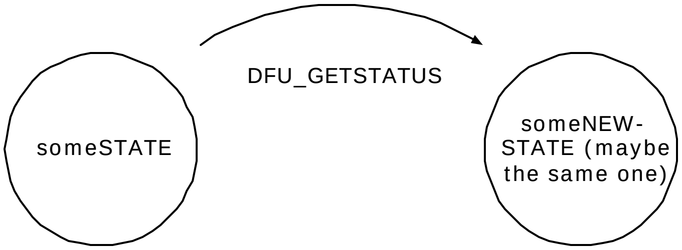

Figure 6.3 Example state transition using DFU_GETSTATUS

6.1.3 DFU_CLRSTATUS Request

Any time the device detects an error and reports an error indication status to the host in the response to a DFU_GETSTATUS request, it enters the dfuERROR state. The device cannot transition from the dfuERROR state, after reporting any error status, until after it has received a DFU_CLRSTATUS request. Upon receipt of DFU_CLRSTATUS, the device sets a status of OK and transitions to the dfuIDLE state. Only then is it able to transition to other states.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
00100001b	DFU_CLRSTATUS	Zero	Interface	Zero	None

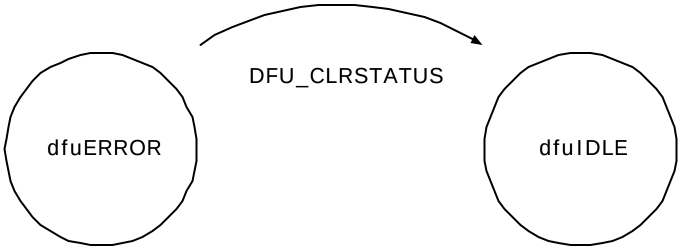

Figure 6.4 Example state transition using DFU_CLRSTATUS to acknowledge an error

6.1.4 DFU_ABORT Request

The DFU_ABORT request enables the host to exit from certain states and return to the DFU_IDLE state. The device sets the OK status on receipt of this request. For more information, see the corresponding state transition summary.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
00100001b	DFU_ABORT	Zero	Interface	Zero	None

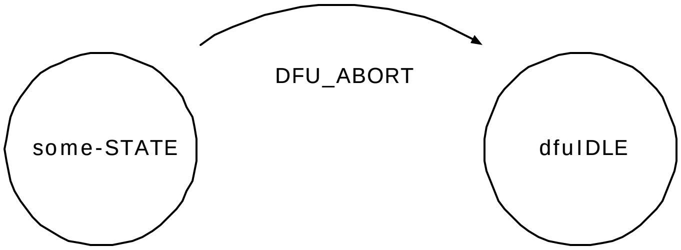

Figure 6.5 Example state transition using DFU_ABORT to terminate a transfer

6.1.5 DFU_GETSTATE Request

This request solicits a report about the state of the device. The state reported is the current state of the device with no change in state upon transmission of the response. The values specified in the *bState* field are identical to those reported in DFU_GETSTATUS.

BmRequestType	bRequest	wValue	wIndex	wLength	Data
10100001b	DFU_GETSTATE	Zero	Interface	1	State

The device responds to the DFU_GETSTATE request with a payload packet containing the following data:

Offset	Field	Size	Value	Description
0	bState	1	Number	Indicates the current state of the device.

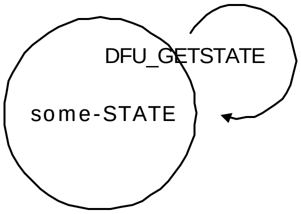

Figure 6.6 Example state transition using DFU_GETSTATE

6.2 Uploading

The purpose of upload is to provide the capability to retrieve and archive a device's firmware. Uploading firmware is, by definition, the inverse of a download, meaning that the uploaded image must be usable in a subsequent download. The host sends DFU_UPLOAD requests to the device until it responds with a short frame as an end of file (EOF) indicator. The device is responsible for selecting the address range to upload and formatting the firmware image appropriately. The host must append the DFU file suffix to the uploaded image. If the host, for some reason, wants to terminate the transfer, it can do so by sending a DFU_ABORT request.

6.2.1 DFU_UPLOAD Request

The DFU_UPLOAD request is employed by the host to solicit firmware from the device.

bmRequestType	bRequest	wValue	wIndex	wLength	Data
10100001b	DFU_ UPLOAD	*BlockNum**	Interface	Length of upload data**	Firm-ware

* The *wValue* field contains a block sequence number. It increments each time a block is transferred, wrapping to zero from 65,535. It is used to provide useful context to the host.

** The *wLength* field indicates the maximum number of bytes of upload data to transfer. This value should not exceed the value specified in the *wTransferSize* field.

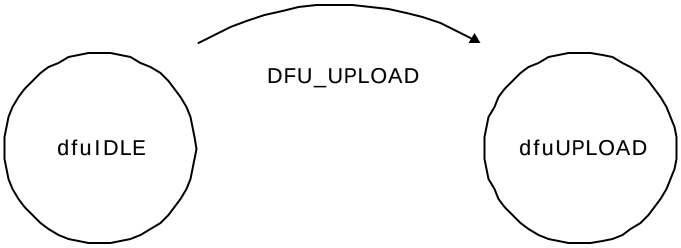

Figure 6.7 Example state transition using DFU_UPLOAD to initiate a transfer

7. Manifestation Phase

After the zero length DFU_DNLOAD request terminates the Transfer phase, the device is ready to manifest the new firmware. As described previously, some devices may accumulate the firmware image and perform the entire reprogramming operation at one time. Others may have only a small amount remaining to be reprogrammed, and still others may have none. Regardless, the device enters the dfuMANIFEST-SYNC state and awaits the solicitation of the status report by the host. Upon receipt of the anticipated DFU_GETSTATUS, the device enters the dfuMANIFEST state, where it completes its reprogramming operations.

Following a successful reprogramming, the device enters one of two states: dfuMANIFEST-SYNC or dfuMANIFEST-WAIT-RESET, depending on whether or not it is still capable of communicating via USB. The host is aware of which state the device will enter by virtue of the *bmAttributes* bit *bitManifestationTolerant*. If the device enters dfuMANIFEST-SYNC (*bitMainfestationTolerant* = 1), then the host issues the DFU_GETSTATUS request, and the device enters the dfuIDLE state. At that point, the host can perform another download, solicit an upload, or issue a USB reset to return the device to application run-time mode. If, however, the device enters the dfuMANIFEST-WAIT-RESET state (*bitManifestationTolerant* = 0), then if *bitWillDetach* = 1 the device generates a detach-attach sequence on the bus, otherwise (*bitWillDetach* = 0) the host must issue a USB reset to the device. After the bus reset the device will evaluate the firmware status and enter the appropriate mode.

A. Interface State Summary

This appendix summarizes the state transitions involved with firmware upgrade operations. It does not attempt to address the normal USB device states.

A.1 Interface State Transition Diagram

This diagram summarizes the DFU interface states and the transitions between them. The events that trigger state transitions can be thought of as arriving on multiple "input tapes" as in the classic Turing machine concept. These multiple conceptual input tapes, or streams, are as follows:

- The control pipe – presents USB DFU class-specific request events to the state machine. USB protocol events, such as completion of the status stage, are also presented on this stream.
- The USB electrical signaling – presents USB reset events to the machine. (For purposes of this document, other signals, such as suspend/resume, are not considered.)
- The power supply to the device – presents power-on events to the device.
- The device peripherals and firmware – timeout, physical hardware error, data content error, completion of peripheral, and memory, operations are examples of the events presented on this stream.

The DFU class-specific requests that the device is required to accept while in any given state are illustrated in the following figure. If the device receives a request, and there is no transition defined for that request (for whatever state the device happens to be in when the request arrives), then the device stalls the control pipe and enters the dfuERROR state. E.g., if the device is in the dfuDNLOAD-SYNC state and a DFU_CLRSTATUS request is received, then the device will stall the control pipe and enter the dfuERROR state.

Note There are two exceptional states with respect to state transitions caused by errors. If an unexpected request arrives while the device is in either the appIDLE or appDETACH states, then the transition to the dfuERROR state does not occur.

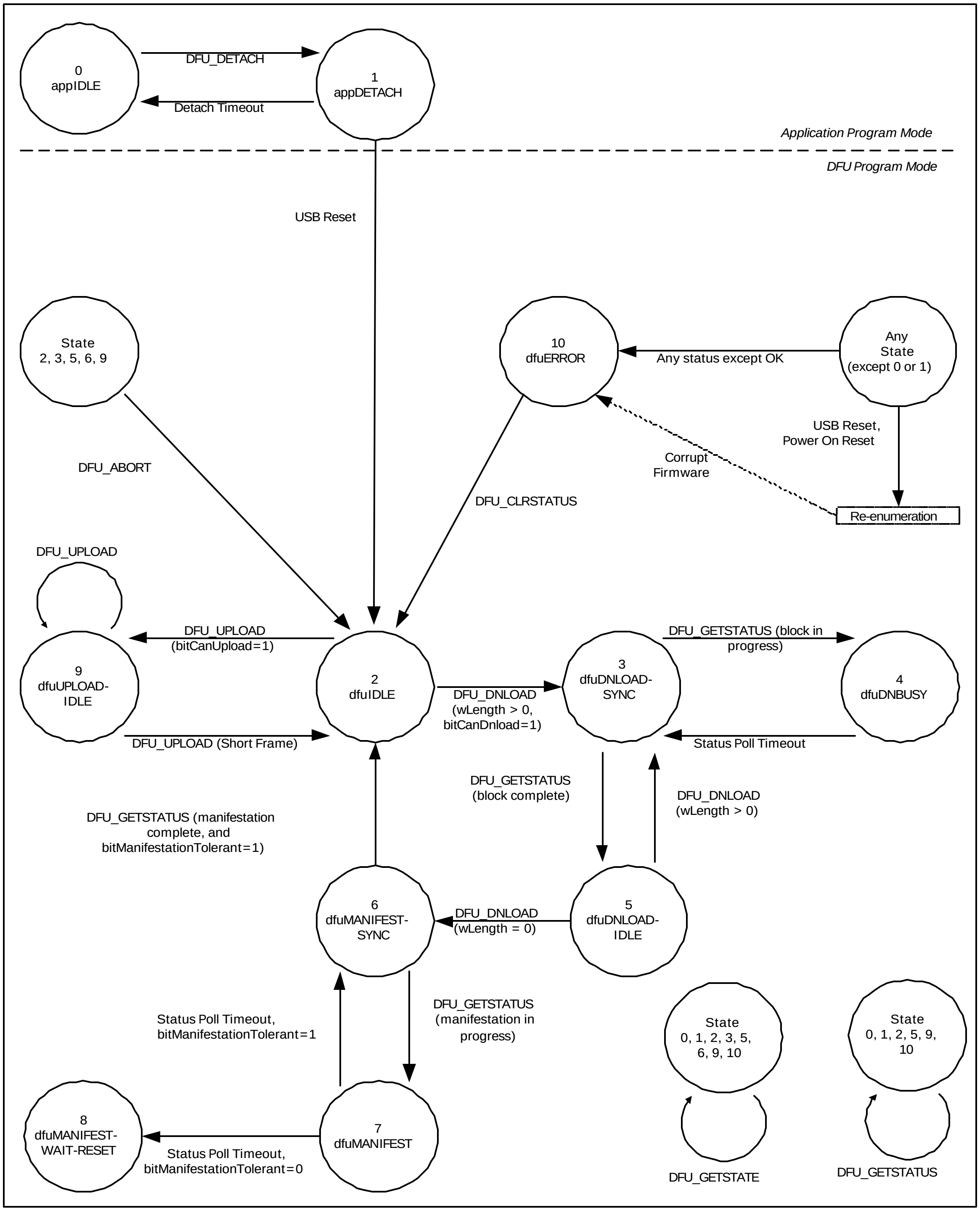

Figure A.1 Interface state transition diagram

A.2 Interface State Transition Summary

The following tables summarize the events that cause state transitions, the actions taken upon detection of the event by the device, and the new state that is entered after the action is performed.

A.2.1 State 0 appIDLE

Event	Action	Next State
Receipt of the DFU_DETACH request	The host sends this request to the device to initiate the DFU process. If *bitWillDetach* is set the device generates a detach-attach sequence on the bus. Otherwise, the device starts the detach timer.	appDETACH
Receipt of the DFU_GETSTATUS request	Support for this request while in the appIDLE state is optional. If it is supported, then the device returns the status response, and *bwPollTimeout* is ignored. Otherwise, the request is treated like any other unsupported request.	appIDLE
Receipt of the DFU_GETSTATE request	Support for this request while in the appIDLE state is optional. If it is supported, then the device returns the status response, and *bwPollTimeout* is ignored. Otherwise, the request is treated like any other unsupported request.	appIDLE
Receipt of any other DFU class-specific request	Device stalls the control pipe.	appIDLE

A.2.2 State 1 appDETACH

Event	Action	Next State
Receipt of the DFU_GETSTATUS request	Device returns the status response; *bwPollTimeout* is ignored.	appDETACH
Receipt of the DFU_GETSTATE request	Device returns the state response.	appDETACH
Receipt of any other DFU class-specific request	Device stalls the control pipe.	appIDLE
Timeout of the device's detach timer	Device does nothing except return to the appIDLE state. A subsequent USB reset will not initiate DFU.	appIDLE
Power on reset	Restart. The device loses all context concerning DFU and operates normally.	appIDLE
USB reset signaling detected	If the device's detach timer is still running (which it should be, or the device would not be in the appDETACH state), then the device prepares to enumerate the DFU descriptor set and enters DFU mode.	dfuIDLE

A.2.3 State 2 dfuIDLE

Event	Action	Next State
Receipt of the DFU_DNLOAD request; *wLength* > 0, and *bitCanDnload* = 1	This is the start of a download block. The device handles the control-write transaction.	dfuDNLOAD-SYNC
Receipt of the DFU_DNLOAD request; *wLength* = 0, or *bitCanDnload* = 0	Device stalls the control pipe. (A zero-length download is not considered useful.)	dfuERROR
Receipt of the DFU_UPLOAD request, and *bitCanUpload* = 1	This is the start of an upload block. The device handles the control-read transaction.	dfuUPLOAD-IDLE
Receipt of the DFU_UPLOAD request, and *bitCanUpload* = 0	Device stalls the control pipe.	dfuERROR
Receipt of the DFU_ABORT request	Do nothing.	dfuIDLE
Receipt of the DFU_GETSTATUS request	Device returns the status response.	dfuIDLE
Receipt of the DFU_GETSTATE request	Device returns the state response.	dfuIDLE
Receipt of any other DFU class-specific request	Device stalls the control pipe.	dfuERROR
USB reset or power on reset and firmware is valid	Re-enumeration. Revert to application firmware.	appIDLE
USB reset or power on reset and firmware is corrupt	Re-enumeration. Remain in DFU mode awaiting recovery attempt by the host.	dfuERROR

A.2.4 State 3 dfuDNLOAD-SYNC

Event	Action	Next State
Receipt of the DFU_GETSTATUS request. (Block transfer still in progress)	Device returns the status response.	dfuDNBUSY
Receipt of the DFU_GETSTATUS request. (Block complete)	Device returns the status response.	dfuDNLOAD-IDLE
Receipt of the DFU_GETSTATE request	Device returns the state response.	dfuDNLOAD-SYNC
Receipt of any other DFU class-specific request	Device stalls the control pipe.	dfuERROR
USB reset or power on reset and firmware is valid	Re-enumeration. Revert to application firmware.	appIDLE
USB reset or power on reset and firmware is corrupt	Re-enumeration. Remain in DFU mode awaiting recovery attempt by the host.	dfuERROR

A.2.5 State 4 dfuDNBUSY

Event	Action	Next State
Receipt of any DFU class-specific request	Device stalls the control pipe.	dfuERROR
bwPollTimeout elapsed	Host can now send DFU_GETSTATUS request.	dfuDNLOAD-SYNC
USB reset or power on reset and firmware is valid	Re-enumeration. Revert to application firmware.	appIDLE
USB reset or power on reset and firmware is corrupt	Re-enumeration. Remain in DFU mode awaiting recovery attempt by the host.	dfuERROR

A.2.6 State 5 dfuDNLOAD-IDLE

Event	Action	Next State
Receipt of the DFU_DNLOAD request; *wLength* > 0	Beginning of a download block. The device handles the control-write transaction.	dfuDNLOAD-SYNC
Receipt of the DFU_DNLOAD request; *wLength* = 0, device agrees	Host is informing the device that there is no more data to download.	dfuMANIFEST-SYNC
Receipt of the DFU_DNLOAD request; *wLength* = 0, but device disagrees	Host and device are not synchronized with respect to the quantity of data to be downloaded. The host must initiate recovery procedures. Device stalls the control pipe.	dfuERROR
Receipt of the DFU_ABORT request	Host is terminating the current download transfer. (Note that if memories have been erased or partially written, the firmware may be corrupt.)	dfuIDLE
Receipt of the DFU_GETSTATUS request	Device returns the status response.	dfuDNLOAD-IDLE
Receipt of the DFU_GETSTATE request	Device returns the state response.	dfuDNLOAD-IDLE
Receipt of any other DFU class-specific request	Device stalls the control pipe.	dfuERROR
USB reset or power on reset and firmware is valid	Re-enumeration. Revert to application firmware.	appIDLE
USB reset or power on reset and firmware is corrupt	Re-enumeration. Remain in DFU mode awaiting recovery attempt by the host.	dfuERROR

A.2.7 State 6 dfuMANIFEST-SYNC

Event	Action	Next State
Receipt of the DFU_GETSTATUS request. Manifestation phase in progress.	Device returns the status response.	dfuMANIFEST
Receipt of the DFU_GETSTATUS request. Manifestation phase complete, and *bitManifestationTolerant* = 1	Device returns the status response.	dfuIDLE
Receipt of the DFU_GETSTATE request	Device returns the state response.	dfuMANIFEST-SYNC
Receipt of any other DFU class-specific request	Device stalls the control pipe.	dfuERROR
USB reset or power on reset and firmware is valid	Re-enumeration. Revert to application firmware.	appIDLE
USB reset or power on reset and firmware is corrupt	Re-enumeration. Remain in DFU mode awaiting recovery attempt by the host.	dfuERROR

A.2.8 State 7 dfuMANIFEST

Event	Action	Next State
Receipt of any DFU class-specific request	Device stalls the control pipe.	dfuERROR
Status poll timeout and *bitManifestationTolerant* = 1	Device that can still communicate via the USB after the Manifestation phase indicated this capability to the host by setting *bmAttributes* bit *bitManifestationTolerant*.	dfuMANIFEST-SYNC
Status poll timeout and *bitManifestationTolerant* = 0	Device that cannot communicate via the USB after the Manifestation phase indicated this limitation to the host by clearing *bmAttributes* bit *bitManifestationTolerant*.	dfuMANIFEST-WAIT-RESET
USB reset or power on reset and firmware is valid	Re-enumeration. Revert to application firmware.	appIDLE
USB reset or power on reset, and firmware is corrupt	Re-enumeration. Remain in DFU mode awaiting recovery attempt by the host.	dfuERROR

A.2.9 State 8 dfuMANIFEST-WAIT-RESET

Event	Action	Next State
Receipt of any DFU class-specific request	If the device could do anything reasonable on the USB, then it would never have entered this state. Do nothing. In fact, the device probably cannot detect the receipt of anything at all. If it could, it would not enter this state.	dfuMANIFEST-WAIT-RESET
USB reset or power on reset and firmware is valid	Re-enumeration. Revert to application firmware.	appIDLE
USB reset or power on reset and firmware is corrupt	Re-enumeration. Remain in DFU mode awaiting recovery attempt by the host.	dfuERROR

A.2.10 State 9 dfuUPLOAD-IDLE

Event	Action	Next State
Receipt of the DFU_UPLOAD request; *wLength* > 0	This is the start of an upload block. The device handles the control-read transaction.	dfuUPLOAD-IDLE
The length of the data transferred by the device in response to a DFU_UPLOAD request is less than *wLength.* (Short frame)	Device finished uploading and completes the control-read operation.	dfuIDLE
Receipt of the DFU_ABORT request	Host is terminating the current upload transfer.	dfuIDLE
Receipt of the DFU_GETSTATUS request	Device returns the status response.	dfuUPLOAD-IDLE
Receipt of the DFU_GETSTATE request	Device returns the state response.	dfuUPLOAD-IDLE
Receipt of any other DFU class-specific request	Device stalls the control pipe.	dfuERROR
USB reset or power on reset and firmware is valid	Re-enumeration. Revert to application firmware.	appIDLE
USB reset or power on reset and firmware is corrupt	Re-enumeration. Remain in DFU mode awaiting recovery attempt by the host.	dfuERROR

A.2.11 State 10 dfuERROR

Event	Action	Next State
Receipt of the DFU_GETSTATUS request	Device returns the status response.	dfuERROR
Receipt of the DFU_GETSTATE request	Device returns the state response.	dfuERROR
Receipt of DFU_CLRSTATUS request	Clear status to OK.	dfuIDLE
Receipt of any other DFU class-specific request	Device stalls the control pipe.	dfuERROR
USB reset or power on reset and firmware is valid	Re-enumeration. Revert to application firmware.	appIDLE
USB reset or power on reset and firmware is corrupt	Re-enumeration. Remain in DFU mode awaiting recovery attempt by the host.	dfuERROR

B. DFU File Suffix

Any file to be downloaded must contain a DFU suffix. The purpose of the DFU suffix is to allow the operating system in general, and the DFU operator interface application in particular, to have a-priori knowledge of whether a firmware download is likely to complete correctly. In other words, these bytes allow the host software to detect and prevent attempts to download incompatible firmware.

The DFU suffix contains the following data:

Offset	Field	Size	Value	Description
-0	*dwCRC*	4	Number	The CRC of the entire file, excluding *dwCRC*. (Calculation specified in the following section).
-4	*bLength*	1	16	The length of this DFU suffix including *dwCRC*.
-5	u*cDfuSignature*	3	uc	The unique DFU signature field.
-8	*bcdDFU*	2	BCD	DFU specification number.
-10	*idVendor*	2	ID	The vendor ID associated with this file. Either FFFFh or must match device's vendor ID.
-12	*idProduct*	2	ID	The product ID associated with this file. Either FFFFh or must match device's product ID.
-14	*bcdDevice*	2	BCD	The release number of the device associated with this file. Either FFFFh or a BCD firmware release or version number.

The *dwCRC*, *bLength*, *ucDfuSignature*, and *bcdDFU* fields will not move in subsequent revisions of this specification. Furthermore, the contents of the *ucDfuSignature* field will remain constant and fixed, and the calculation for the *dwCRC* field will remain fixed as defined in version 1.0 of this specification. If any fields are added to this suffix, they will be added at greater negative offsets than specified in the *bcdDevice* field. The *dwCRC* field is defined as being the first four bytes of the suffix, which makes it the last four bytes of a file to which the suffix has been added.

The offsets are negative. This is a file suffix, and the negative offsets indicate that the last byte of the file is specified in the *dwCRC* field. Note that all multibyte fields are mirror images of their structure. The host must perform an end-for-end swap of the entire suffix of *bLength* bytes to obtain the correct byte ordering. In other words, when the DFU suffix is created, the fields are filled with positive offsets. Then the entire suffix is end-for-end swapped before being appended to the download file. This is done so that the CRC, length byte, and DFU signature will be at a fixed location in all cases, specifically EOF. This allows for possible future revisions of the DFU suffix, or even vendor-specific additions, without difficulty.

The *dwCRC* field contains the CRC as explained in the following paragraphs. The CRC is calculated for all bytes contained in the file, except the *dwCRC* itself.

The *bLength* field is a single-byte length field. In this revision of the DFU specification, the length is 16 (decimal) and includes the four bytes occupied by the *dwCRC* field.

The *ucDfuSignature* field contains three unsigned characters: 44h, 46h, 55h, in that order. In the file, they appear in reverse order, i.e., offset (–5) is 44h, offset (–6) is 46h, and offset (–7) is 55h.

The *bcdDFU* field is a two-byte specification revision number. The value as of this revision of the specification is 0100h, representing version 1.0.

The *idVendor* field may either contain a valid vendor ID, or it may contain FFFFh. If it contains FFFFh, then the file may be sent to any device. The reason to include a DFU suffix with a vendor ID of FFFFh is to maintain a standard file format, or to include a release number specified in the *bcdDevice* field for informational purposes, without enforcing vendor ID matching. If the *idVendor* field contains a value other than FFFFh, then the file contents may only be sent to a device with a matching vendor ID reported in the *idVendor* field of its device descriptor.

The *idProduct* field is ignored if the *idVendor* field contains FFFFh. Otherwise, *idProduct* may either contain a valid product ID, or it may contain FFFFh. If it contains FFFFh, then the file may be sent to any device with a matching vendor ID. If *idProduct* contains a value other than FFFFh, then the file contents may only be sent to a device with matching *idVendor* and *idProduct* fields reported in its device descriptor.

Note Because the *idProduct* field of the DFU-based version of the product may differ from the run-time version, *idProduct* of the suffix should contain the same value as the run-time version of the product. At the time when the host performs a comparison, the DFU descriptor set has not yet been enumerated, and whatever *idProduct* is present in the DFU descriptor set is unavailable.

The *bcdDevice* field may contain FFFFh, or it may contain a BCD number. If it contains FFFFh, it is ignored. If it contains a BCD number, then that number should represent the version of firmware contained in the file. This field is for informational purposes only and does not restrict whether the file may or may not be sent to a device. One possible use of this field is to notify the operator when a download will result in sending a lower firmware version number to the device than the version number that is currently reported by the device. Therefore, it is suggested, but not required, that vendors use this field to record a firmware version number that increases with each revision of firmware.

In no case is the DFU suffix ever sent to the device. The host application verifies that the bytes occupying the *ucDfuSignature* field contain the specified values, and that the CRC over the file matches the *dwCRC* field. If these two criteria are passed, then the host can presume that the firmware upgrade file is intact. The host application then uses the DFU suffix data to perform appropriate validation and screening. During the Transfer phase, the contents of the file are sent, excluding the DFU suffix data.

B. 1 Portable C Source for CRC and DFU Suffix

The following example source code was created by assimilating code from a number of sources. It was edited to illustrate the concepts described in this appendix.

B.1.2 Source Listing

```
/***************************************************************************
\
 dfu.c

 This is sample software to demonstrate a simple method of manipulating
 the DFU suffix as specified in the DFU specification version 1.0.

 The following authors have contributed to this sample code:

 Robert Nathan
 Greg Kroah-Hartman
 Trenton Henry
 Stephen Satchell
 Chuck Foresburg
 Gary S. Brown

 The CRC algorithm derives from the works of the last three authors listed.

 The authors hereby grant developers the right to incorporate any portion
 of this source into their own works, provided that proper credit is given
 to Gary S. Brown, Stephen Satchell, and Chuck Forsberg. Reference the
 following source for the proper format.

 Every attempt has been made to ensure that this source is portable.
 To that end, it uses only ANSI C libraries. Any identifiers that are not
 part of ANSI C have names starting with leading underscores.  The purpose
 is to differentiate what has been "invented" and what was "pre-existing".

 This example cannot modify an existing suffix.  To modify a suffix,
 delete the current one and then append a new suffix.

\***************************************************************************
/
#include <stdio.h>
#include <io.h>
#include <sys\stat.h>
#include <stdarg.h>
#include <string.h>
#include <stdlib.h>
#include <errno.h>

/***************************************************************************
\
 CRC polynomial 0xedb88320 – Contributed unknowingly by Gary S. Brown.

 "Copyright (C) 1986 Gary S. Brown.  You may use this program, or code or
  tables extracted from it, as desired without restriction."

 Paraphrased comments from the original:

 The 32 BIT ANSI X3.66 CRC checksum algorithm is used to compute the 32-bit
 frame check sequence in ADCCP.  (ANSI X3.66, also known as FIPS PUB 71 and
 FED-STD-1003, the U.S. versions of CCITT's X.25 link-level protocol.)
 The polynomial is:
 X^32+X^26+X^23+X^22+X^16+X^12+X^11+X^10+X^8+X^7+X^5+X^4+X^2+X^1+X^0

 Put the highest-order term in the lowest-order bit.  The X^32 term is
 implied, the LSB is the X^31 term, etc.  The X^0 term usually shown as +1)
 results in the MSB being 1.  Put the highest-order term in the lowest-order
 bit.  The X^32 term is implied, the LSB is the X^31 term, etc.  The X^0
 term (usually shown as +1) results in the MSB being 1.

 The feedback terms table consists of 256 32-bit entries.  The feedback
terms
 simply represent the results of eight shift/xor operations for all
 combinations of data and CRC register values.  The values must be right-
 shifted by eight bits by the UPDCRC logic so the shift must be unsigned.

\****************************************************************************
/
unsigned long _crctbl[] = {
    0x00000000, 0x77073096, 0xee0e612c, 0x990951ba, 0x076dc419, 0x706af48f,
    0xe963a535, 0x9e6495a3, 0x0edb8832, 0x79dcb8a4, 0xe0d5e91e, 0x97d2d988,
    0x09b64c2b, 0x7eb17cbd, 0xe7b82d07, 0x90bf1d91, 0x1db71064, 0x6ab020f2,
    0xf3b97148, 0x84be41de, 0x1adad47d, 0x6ddde4eb, 0xf4d4b551, 0x83d385c7,
    0x136c9856, 0x646ba8c0, 0xfd62f97a, 0x8a65c9ec, 0x14015c4f, 0x63066cd9,
    0xfa0f3d63, 0x8d080df5, 0x3b6e20c8, 0x4c69105e, 0xd56041e4, 0xa2677172,
    0x3c03e4d1, 0x4b04d447, 0xd20d85fd, 0xa50ab56b, 0x35b5a8fa, 0x42b2986c,
    0xdbbbc9d6, 0xacbcf940, 0x32d86ce3, 0x45df5c75, 0xdcd60dcf, 0xabd13d59,
    0x26d930ac, 0x51de003a, 0xc8d75180, 0xbfd06116, 0x21b4f4b5, 0x56b3c423,
    0xcfba9599, 0xb8bda50f, 0x2802b89e, 0x5f058808, 0xc60cd9b2, 0xb10be924,
    0x2f6f7c87, 0x58684c11, 0xc1611dab, 0xb6662d3d, 0x76dc4190, 0x01db7106,
    0x98d220bc, 0xefd5102a, 0x71b18589, 0x06b6b51f, 0x9fbfe4a5, 0xe8b8d433,
    0x7807c9a2, 0x0f00f934, 0x9609a88e, 0xe10e9818, 0x7f6a0dbb, 0x086d3d2d,
    0x91646c97, 0xe6635c01, 0x6b6b51f4, 0x1c6c6162, 0x856530d8, 0xf262004e,
    0x6c0695ed, 0x1b01a57b, 0x8208f4c1, 0xf50fc457, 0x65b0d9c6, 0x12b7e950,
    0x8bbeb8ea, 0xfcb9887c, 0x62dd1ddf, 0x15da2d49, 0x8cd37cf3, 0xfbd44c65,
    0x4db26158, 0x3ab551ce, 0xa3bc0074, 0xd4bb30e2, 0x4adfa541, 0x3dd895d7,
    0xa4d1c46d, 0xd3d6f4fb, 0x4369e96a, 0x346ed9fc, 0xad678846, 0xda60b8d0,
    0x44042d73, 0x33031de5, 0xaa0a4c5f, 0xdd0d7cc9, 0x5005713c, 0x270241aa,
    0xbe0b1010, 0xc90c2086, 0x5768b525, 0x206f85b3, 0xb966d409, 0xce61e49f,
    0x5edef90e, 0x29d9c998, 0xb0d09822, 0xc7d7a8b4, 0x59b33d17, 0x2eb40d81,
    0xb7bd5c3b, 0xc0ba6cad, 0xedb88320, 0x9abfb3b6, 0x03b6e20c, 0x74b1d29a,
    0xead54739, 0x9dd277af, 0x04db2615, 0x73dc1683, 0xe3630b12, 0x94643b84,
    0x0d6d6a3e, 0x7a6a5aa8, 0xe40ecf0b, 0x9309ff9d, 0x0a00ae27, 0x7d079eb1,
    0xf00f9344, 0x8708a3d2, 0x1e01f268, 0x6906c2fe, 0xf762575d, 0x806567cb,
    0x196c3671, 0x6e6b06e7, 0xfed41b76, 0x89d32be0, 0x10da7a5a, 0x67dd4acc,
    0xf9b9df6f, 0x8ebeeff9, 0x17b7be43, 0x60b08ed5, 0xd6d6a3e8, 0xa1d1937e,
    0x38d8c2c4, 0x4fdff252, 0xd1bb67f1, 0xa6bc5767, 0x3fb506dd, 0x48b2364b,
    0xd80d2bda, 0xaf0a1b4c, 0x36034af6, 0x41047a60, 0xdf60efc3, 0xa867df55,
    0x316e8eef, 0x4669be79, 0xcb61b38c, 0xbc66831a, 0x256fd2a0, 0x5268e236,
    0xcc0c7795, 0xbb0b4703, 0x220216b9, 0x5505262f, 0xc5ba3bbe, 0xb2bd0b28,
    0x2bb45a92, 0x5cb36a04, 0xc2d7ffa7, 0xb5d0cf31, 0x2cd99e8b, 0x5bdeae1d,
    0x9b64c2b0, 0xec63f226, 0x756aa39c, 0x026d930a, 0x9c0906a9, 0xeb0e363f,
    0x72076785, 0x05005713, 0x95bf4a82, 0xe2b87a14, 0x7bb12bae, 0x0cb61b38,
    0x92d28e9b, 0xe5d5be0d, 0x7cdcefb7, 0x0bdbdf21, 0x86d3d2d4, 0xf1d4e242,
    0x68ddb3f8, 0x1fda836e, 0x81be16cd, 0xf6b9265b, 0x6fb077e1, 0x18b74777,
    0x88085ae6, 0xff0f6a70, 0x66063bca, 0x11010b5c, 0x8f659eff, 0xf862ae69,
    0x616bffd3, 0x166ccf45, 0xa00ae278, 0xd70dd2ee, 0x4e048354, 0x3903b3c2,
    0xa7672661, 0xd06016f7, 0x4969474d, 0x3e6e77db, 0xaed16a4a, 0xd9d65adc,
    0x40df0b66, 0x37d83bf0, 0xa9bcae53, 0xdebb9ec5, 0x47b2cf7f, 0x30b5ffe9,
    0xbdbdf21c, 0xcabac28a, 0x53b39330, 0x24b4a3a6, 0xbad03605, 0xcdd70693,
    0x54de5729, 0x23d967bf, 0xb3667a2e, 0xc4614ab8, 0x5d681b02, 0x2a6f2b94,
    0xb40bbe37, 0xc30c8ea1, 0x5a05df1b, 0x2d02ef8d};
/***************************************************************************
**/

unsigned char _suffix[] = {
    0x00,                       /* bcdDevice lo */
    0x00,                       /* bcdDevice hi */
    0x00,                       /* idProduct lo */
    0x00,                       /* idProduct hi */
    0x00,                       /* idVendor lo */
    0x00,                       /* idVendor hi */
    0x00,                       /* bcdDFU lo */
    0x01,                       /* bcdDFU hi */
    'U',                        /* ucDfuSignature lsb */
    'F',                        /* ucDfuSignature --- */
    'D',                        /* ucDfuSignature msb */
    16,                         /* bLength for this version */
    0x00,                       /* dwCRC lsb */
    0x00,                       /* dwCRC --- */
    0x00,                       /* dwCRC --- */
    0x00                        /* dwCRC msb */
};
/***************************************************************************
\
\***************************************************************************
/
void _fatal(char *);
void _fatal(char *_str)
{
    perror(_str);
    fcloseall();
    abort();
}
/***************************************************************************
\
 The updcrc macro (referred to here as _crc) is derived from an article
 Copyright © 1986 by Stephen Satchell.

 “Programmers may incorporate any or all code into their programs, giving
  proper credit within the source.  Publication of the source routines is
  permitted so long as proper credit is given to Steven Satchell, Satchell
  Evaluations, and Chuck Forsberg, Omen technology."
\***************************************************************************
/
#define _crc(accum,delta)
(accum)=_crctbl[((accum)^(delta))&0xff]^((accum)>>8)
#define _usage                                                    \
 "\nusage: dfu fname [options]\n\n"                               \
 " to check for a suffix use: dfu fname\n\n"                      \
 " to remove a suffix use: dfu fname -del\n\n"                    \
 " to add a suffix use: dfu fname -did val -pid val -vid val\n\n" \
 " e.g., dfu myfile -did 0x0102 -pid 2345 -vid 017\n"             \
 " sets idDevice 0x0102 idProduct 0x0929 idVendor 0x000F\n\n"
#define _getarg(ident,index);                                     \
  if (!strcmp(argv[_i], (ident)))                             \
  {                                                           \
    _write_suffix = 1;                                        \
    if (argc-1 == _i) _fatal(_usage);                         \
    _tmpl = strtol(argv[_i+1], &_charp, 0);                   \
    _suffix[(index)] = (unsigned char)(_tmpl & 0x000000FF);   \
    _tmpl /= 256;                                             \
    _suffix[(index)+1] = (unsigned char)(_tmpl & 0x000000FF); \
  }
/***************************************************************************
\
\***************************************************************************
/
void main(int argc, char **argv)
{
    FILE           *_fp;
    FILE           *_tmpfp;
    int             _remove_suffix = 0;
    int             _write_suffix = 0;
    unsigned long   _filecrc;
    unsigned long   _fullcrc;
    long            _i;
    long            _tmpl;
    char           *_charp;

    /* make sure there is at least one argument */
    errno = EINVAL;
    if (argc < 2)
        _fatal(_usage);

    /* make sure the file is there */
    _fp = fopen(argv[1], "r+b");
    if (!_fp)
        _fatal(argv[1]);

    /* compute the CRC up to the last 4 bytes */
    fseek(_fp, -4L, SEEK_END);
    _i = ftell(_fp);
    rewind(_fp);
    _filecrc = 0xffffffff;
    for (; _i; _i--)
        _crc(_filecrc, (unsigned char) fgetc(_fp));
    /* printf("file crc: 0x%08lX\n", _filecrc); */

    /* compute the CRC of everything including the last 4 bytes */
    _fullcrc = _filecrc;
    for (_i = 0; _i < 4; _i++)
        _crc(_fullcrc, (unsigned char) fgetc(_fp));
    /* printf("full crc: 0x%08lX\n", _fullcrc); */

    /* store the file crc away for comparison */
    for (_i = 12; _i < 16; _i++) {
        _suffix[_i] = (unsigned char) (_filecrc & 0x000000ff);
        _filecrc /= 256;
    }

    /* pretend that a suffix exists and try to validate it */
    fseek(_fp, -16L, SEEK_END);

    /* read in the existing suffix */
    for (_i = 0; _i < 6; _i++)
        _suffix[_i] = (unsigned char) fgetc(_fp);

    /* print out whats in there already */
    printf(" idDevice: 0x%02X%02X\n",
           (unsigned char) _suffix[1], (unsigned char) _suffix[0]);
    printf("idProduct: 0x%02X%02X\n",
           (unsigned char) _suffix[3], (unsigned char) _suffix[2]);
    printf(" idVendor: 0x%02X%02X\n",
           (unsigned char) _suffix[5], (unsigned char) _suffix[4]);

    /* now parse the command arguments to overwrite the suffix w/ new values
*/
    for (_i = 1; _i < argc; _i++) {
        errno = EINVAL;
        if (!strcmp(argv[_i], "-del"))
            _remove_suffix = 1;
        _getarg("-did", 0);
        _getarg("-pid", 2);
        _getarg("-vid", 4);
    }

    /* compare the ‘presumed’ file suffix to the suffix in memory */
    for (_i = 6; _i < sizeof(_suffix); _i++)
        if ((unsigned char) fgetc(_fp) != _suffix[_i])
            break;
    if (_i < 8)
        printf("bad bcdDFU\n");
    else if (_i < 11)
        printf("bad ucDfuSignature\n");
    else if (_i < 12)
        printf("bad bLength\n");
    else if (_i < 16)
        printf("bad dwCRC\n");
    if (_i < 16) {
        /* can't remove a suffix if there isn't one there */
        if (_remove_suffix)
            printf("invalid or missing suffix\n");
        _remove_suffix = 0;
    } else {
        printf("valid dfu suffix found\n");
        errno = EINVAL;
        if (_write_suffix)
            _fatal("delete suffix before making changes\n");
    }

    /* now it is known if a suffix exists, and the important
       information has been printed out.  so, either the user wants
       to delete the suffix, or to add a new one */

    /* remove an existing suffix? */
    if (_remove_suffix) {
        _tmpfp = fopen("dfu.tmp", "w+b");
        if (!_tmpfp)
            _fatal("dfu.tmp");

        /* this is not an exercise in how to do buffered file io ;-) */
        fseek(_fp, -_suffix[11], SEEK_END);
        _i = ftell(_fp);
        if (_i > 0) {
            rewind(_fp);
            for (; _i; _i--)
                fputc(fgetc(_fp), _tmpfp);
            fclose(_tmpfp);
            fclose(_fp);
            chmod(argv[1], S_IWRITE);
            remove(argv[1]);
            rename("dfu.tmp", argv[1]);
            /* warm fuzzies */
            printf("dfu suffix removed from %s\n", argv[1]);
        } else
            printf("%s too small to contain dfu suffix\n", argv[1]);
        exit(0);
    }

    /* append a suffix to the file? */
    if (_write_suffix) {
        /* append a DFU suffix */
        fseek(_fp, 0L, SEEK_END);

        /* write the suffix while iterating the CRC */
        for (_i = 0; _i < sizeof(_suffix) - 4; _i++) {
            _crc(_fullcrc, _suffix[_i]);
            fputc(_suffix[_i], _fp);
        }

        /* and write the CRC, lo to hi */
        /* printf("full crc: 0x%08lX\n", _fullcrc); */
        for (_i = 0; _i < 4; _i++) {
            fputc((unsigned char) (_fullcrc & 0x000000ff), _fp);
            _fullcrc /= 256;
        }

        /* warm fuzzies */
        printf("dfu suffix appended to %s\n", argv[1]);
    }
    /* finished */
    fclose(_fp);
}
/* eof */
```