Universal Serial Bus Power Delivery Firmware Update Specification

Revision 1.0
September 15, 2016

CONTENTS

TABLES

FIGURES

Specification Work Group Chairs / Specification Editors

Apple	Co-Chair	Colin Whitby-Strevens	
Specwerkz	Co-Chair	Bob Dunstan	
Intel Corporation	Editor	Stephanie Wallick	

Specification Work Group Contributors

Advanced Micro Devices	Joseph Scanlon		
Analogix Semiconductor, Inc.	Mehran Badii	Greg Stewart	
Apple	William Ferry	Scott Jackson	Karthik Raj Kaliannan
	Robert Walsh	Kevin Hsiue	
Canyon Semiconductor	YuHung Lin		
Chrontel, Inc.	David Tsai		
Cypress Semiconductor	Jagadeesan Raj	Subu Sankaran	
Dell Inc.	Marcin Nowak		
DisplayLink (UK) Ltd.	Richard Petrie		
Ellisys	Tim Wei	Abel Astley	
Fairchild Semiconductor	Oscar Freitas	Joseph Bauman	
Google Inc.	Mark Hayter	Vincent Palatin	David Schneider
Intel Corporation	Abdul Ismail	Christine Krause	Brad Saunders
Lattice Semiconductor Corp	Babu Mailachalam		
MCCI Corporation	Terry Moore		
Microchip Technology Inc.	Shannon Cash	John Sisto	
Microsoft Corporation	Michelle Bergeron	Anthony Chen	Tatu Tomppo
MQP Electronics Ltd.	Sten Carlsen		
NXP Semiconductors	Vijendra Kuroodi		
Renesas Electronics Corp.	Philip Leung	Toshifumi Yamaoka	
STMicroelectronics	Nathalie Ballot	Chekib Hammami	
Total Phase	Chris Yokum		
VIA Technologies, Inc.	Jay Tseng		

Revision History

Revision	Date	Description
1.0	September 15, 2016	Initial Release

1 Introduction

1.1 Purpose

This specification defines a common method for updating the firmware in a **USB PD** capable device.

The USB Power Delivery Firmware Update Specification is guided by the following principles:

- Define a common method to update the firmware in a PD capable device such as a PD charger or a USB Type-C Alternate Mode device
- Define a secure method designed to thwart installation of compromised firmware
- Complement and work in congruence with existing **USB DFU** Class implementations

1.2 Scope

This specification is intended as a supplement to the existing *USB Power Delivery* specification. It addresses only the elements required to implement and support firmware update using USB Power Delivery.

Normative information is provided to allow interoperability of components designed to this specification. Informative information, when provided, may illustrate possible design implementations.

1.3 Related Documents

USB PD	*Universal Serial Bus Power Delivery Specification, Revision 3.0, Version 1.0a, March 25, 2016 (referred to in this document as the USB PD Specification) (available at:* http://www.usb.org/developers/docs*).*
USB Type-C	*Universal Serial Bus Type-C Cable and Connector Specification, Revision 1.2, March 25, 2016 (referred to in this document as the USB Type-C Specification)(available at:* http://www.usb.org/developers/docs*).*
USB Type-C Bridge	*Universal Serial Bus Type-C Bridge Specification, Revision 1.0, June 15, 2016, (available at* http://www.usb.org/developers/docs*).*
USB DFU	*USB Device Class Specification for Device Firmware Upgrade, Version 1.1, August 5, 2004, (available at* http://www.usb.org/developers/docs*).*

1.4 Conventions

1.4.1 Precedence

If there is a conflict between text, figures, and tables, the precedence shall be tables, figures, and then text.

1.4.2 Keywords

The following keywords differentiate between the levels of requirements and options.

1.4.2.1 Conditional Normative

Conditional Normative is a keyword used to indicate a feature that is mandatory when another related feature has been implemented. Designers are mandated to implement all such requirements, when the dependent features have been implemented, to ensure interoperability with other compliant Products.

1.4.2.2 Deprecated

Deprecated is a keyword used to indicate a feature, supported in previous releases of the specification, which is no longer supported.

1.4.2.3 Informative

Informative is a keyword that describes information with this specification that intends to discuss and clarify requirements and features as opposed to mandating them.

1.4.2.4 May

May is a keyword that indicates a choice with no implied preference.

1.4.2.5 N/A

N/A is a keyword that indicates that a field or value is not applicable and has no defined value and shall not be checked or used by the recipient.

1.4.2.6 Normative

Normative is a keyword that describes features that are mandated by this specification.

1.4.2.7 Optional/Optionally/Optional Normative

Optional, ***Optionally***, and ***Optional Normative*** are equivalent keywords that describe features not mandated by this specification. However, if an ***Optional*** feature is implemented, the feature shall be implemented as defined by this specification.

1.4.2.8 Reserved

Reserved is a keyword indicating reserved bits, bytes, words, fields, and code values that are set-aside for future standardization. Their use and interpretation may be specified by future extensions to this specification and, unless otherwise stated, shall not be utilized or adapted by vendor implementation. A ***Reserved*** bit, byte, word, or field shall be set to zero by the sender and shall be ignored by the receiver. ***Reserved*** field values shall not be sent by the sender and, if received, shall be ignored by the receiver.

1.4.2.9 Shall/Normative

Shall and ***Normative*** are keywords indicating a mandatory requirement. Designers are mandated to implement all such requirements to ensure interoperability with other compliant Products.

1.4.2.10 Should

Should is a keyword indicating flexibility of choice with a preferred alternative. Equivalent to the phrase “it is recommended that”.

1.4.3 Numbering

Numbers that are immediately followed by a lowercase “b” (e.g., 01b) are binary values. Numbers that are immediately followed by a lowercase “h” (e.g., 3Ah) are hexadecimal values. Numbers not immediately followed by either a “b”, or “h” are decimal values.

1.5 Terms and Abbreviations

This section defines the terms and abbreviations used throughout this document.

Table 1-1: Terms and Abbreviations

Term	Description
Alternate Mode	Operation defined by a vendor or standards organization that is associated with a SVID assigned by the USB-IF. Entry into and exit from an Alternate Mode is controlled by the **USB PD** Structured VDM Enter Mode and Exit Mode commands, respectively.
CC	Configuration Channel (CC) used in the discovery, configuration and management of connections across a **USB Type-C** cable.
Direct Connect	The host's DFP is connected directly to the PD Device's UFP with no USB hub in between, either via a cable or without (e.g., thumb drive).
Downstream Facing Port (DFP)	Defined in **USB PD.**
Electronically Marked Cable	A **USB Type-C** cable that uses **USB PD** to provide the cable's characteristics.
LSB	Least significant byte
MSB	Most significant byte
PD Device	A device that supports **USB PD**.
PD Sink	A Sink that is managed using **USB PD**.
PD Source	A Source that is managed using **USB PD**.
PDFU	Power Delivery Firmware Update.
PDFU Depot	A place or entity that stores firmware images.
PDFU Initiator	The system that retrieves a firmware image from the PDFU Depot and provides it to the PDFU Responder.
PDFU Responder	The system that receives a firmware image.
Port Partner	Refers to the port (PD Device or host) that another port is attached to.
Request	USB PD Firmware Update Request (as specified in **USB PD**).
Response	USB PD Firmware Update Response (as specified in **USB PD**).
Silent Failure	Exiting the PDFU Flow prior to successful firmware image update without first notifying the user that an error has occurred.
Silent Update	The updating of a firmware image without first informing a user.
Sink	Port consuming power from VBUS; most commonly a device.
Source	Port providing power over VBUS; most commonly a Host or Hub DFP.
Type-C Port	The USB port associated to a **USB Type-C** receptacle. This includes the USB signaling, CC logic, multiplexers and other associated logic.

Term	Description
Upstream Facing Port (UFP)	Defined in **USB PD.**
VCONN-powered accessory	An accessory that is powered from VCONN to operate in an Alternate Mode.

2 Overview

This section contains no Normative requirements.

2.1 Introduction

This specification describes the architecture and methodology to load a single consolidated firmware image using USB Power Delivery. The management of multiple firmware images is outside the scope of this specification. Product vendors can implement a means of managing multiple firmware images provided that those means do not conflict with or alter the specifications described herein.

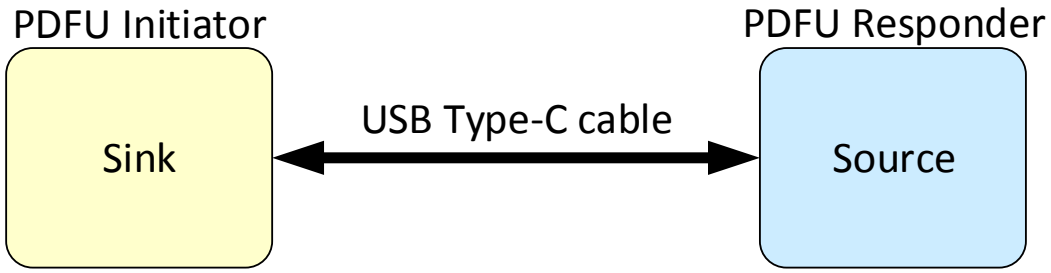
2.2 Use Cases

The use cases for PD Firmware Update are described in four topology scenarios for firmware update. The first three are for PDFU Responders that are directly connected to the PDFU Initiator. The last is for a PDFU Responder connected behind a Hub with an integrated **USB Type-C Bridge**.

2.2.1 Scenario 1 – Updating a PD Source over a USB Type-C Cable

Figure 2-1 shows a firmware update topology scenario where the PDFU Initiator is a PD Sink (e.g. a laptop) and the PDFU Responder is a PD Source (e.g. AC/DC power adapter). Note that the PDFU Initiator and PDFU Responder are not likely to have appropriate USB data connectivity in this scenario. Thus, **USB DFU** is not likely to be an option for firmware update.

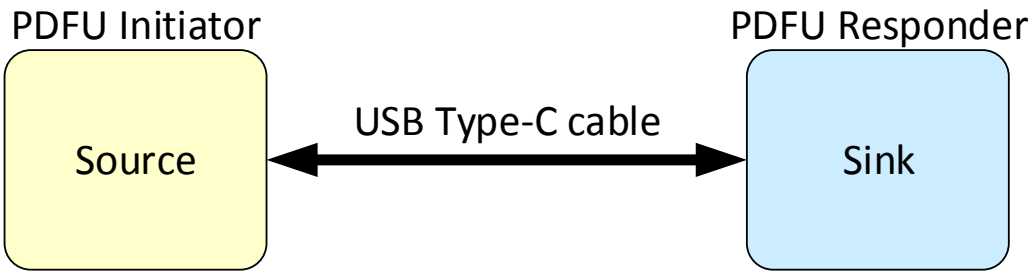
Figure 2-1 Scenario 1



2.2.2 Scenario 2 – Updating a PD Sink over a USB Type-C Cable

Figure 2-2 shows a firmware update topology scenario where the PDFU Initiator is a PD Source (e.g. a desktop capable of powering peripheral devices) and the PDFU Responder is a PD Sink (e.g. Captive Cable Sink or Sink with Alternate Mode Adapter). Note that the PDFU Initiator and PDFU Responder may also have USB data connectivity and could potentially use **USB DFU** for firmware update as well.

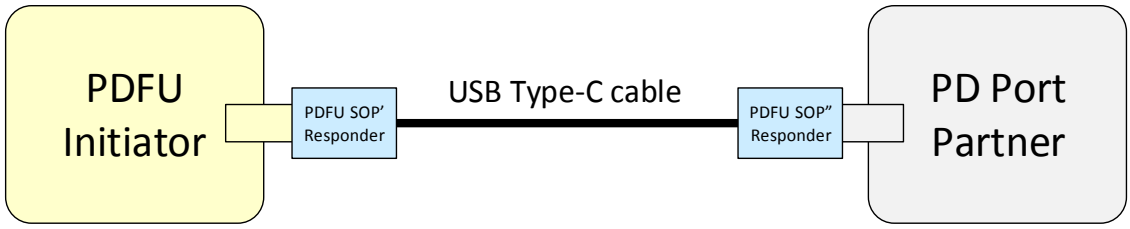
Figure 2-2 Scenario 2



2.2.3 Scenario 3 – Updating a USB Type-C Cable

Figure 2-3 shows a firmware update topology scenario where the PDFU Responders are the ends of a PD cable (SOP' and SOP'').

Figure 2-3 Scenario 3



In this scenario, the following conditions have to be met:

- The PDFU Initiator supplies VCONN.
- The PDFU Initiator, the PDFU Responder(s) in the cable, and the system at the far end of the cable to the PDFU Initiator all implement **USB PD** Revision 3.0a or later.
- The **USB Type-C** cable can tolerate the error case where only one end updates successfully.

This scenario has the following limitations:

- If the cable has an isolated SOP' at each end (i.e. only one end is powered at a time – no VCONN wire in the cable), the user has to reverse the cable in the middle of the update process.
- A PD Port Partner has to be connected to the end of the cable. The Port Partner does not participate in firmware update but must support **USB PD** Revision 3.0 or later.
- The PDFU Initiator has to enter into an explicit contract with the PD Port Partner before it can initiate firmware update.

2.2.4 Scenario 4 – Update via a USB Hub

In this firmware update topology scenario, the PDFU Responder is downstream of a Hub.

Figure 2-4 shows Scenario 4a, where the PDFU Responder connected behind the Hub is a PD Source or PD Sink (Scenarios 1 and 2 respectively).

Figure 2-4 Scenario 4a

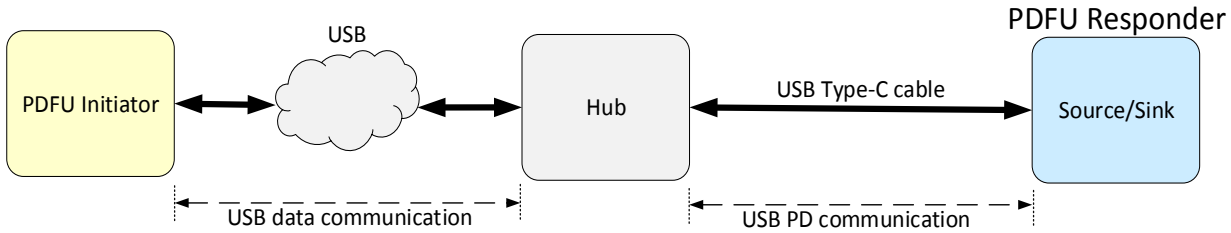
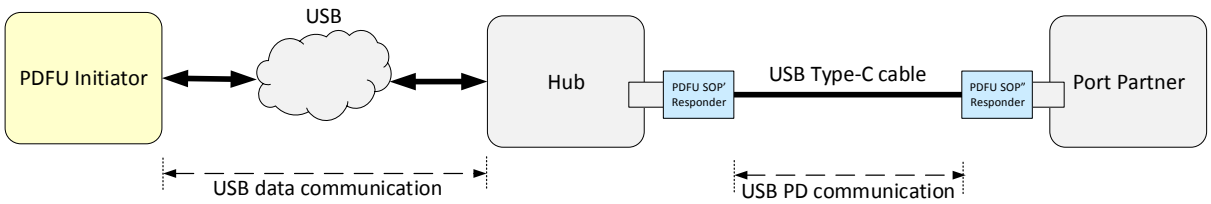


Figure 2-5 shows Scenario 4b, where the PDFU Responder connected behind the Hub is either end of a PD cable (SOP' or SOP").

Figure 2-5 Scenario 4b

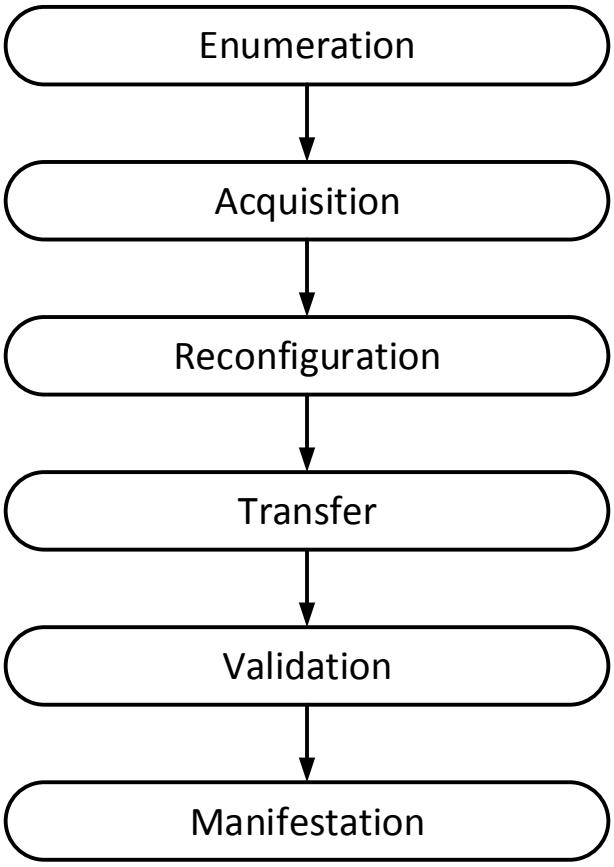


These scenarios require a Hub that implements a **USB Type-C Bridge**. If the Hub does not implement a **USB Type-C Bridge**, then the PDFU Initiator has no means of communicating with the PDFU Responder and will not be able to detect whether or not the PDFU Responder needs a firmware update.

2.3 PD Firmware Update Flow

The PD Firmware Update Flow (PDFU Flow) consists of six phases. Figure 2-6 shows the order in which these phases occur.

Figure 2-6 PDFU Flow



2.3.1 Enumeration

In this phase, the PDFU Initiator queries the PDFU Responder. The PDFU Responder then responds with vendor and device identification and firmware version information.

Enumeration takes place after a **USB PD** connection has been established and a **USB PD** Explicit Contract has been negotiated. Enumeration can be performed at any time while in an Explicit Contract.

2.3.2 Acquisition

In this phase, the PDFU Initiator uses the vendor and device identification and firmware version information to query the appropriate PDFU Depot for a firmware update. The PDFU Depot either responds with a new firmware update image, or indicates that there is no appropriate update, in which case the PDFU Flow terminates.

2.3.3 Reconfiguration

In this phase, the PDFU Initiator and the PDFU Responder perform any necessary reconfiguration in order to prepare for the transfer of the new firmware image.

2.3.4 Transfer

In this phase, the new firmware image is transferred from the PDFU Initiator to the PDFU Responder.

2.3.5 Validation

In this phase, the PDFU Responder validates the new firmware image (including verifying a vendor-dependent signature) and reports success or failure to the PDFU Initiator.

2.3.6 Manifestation

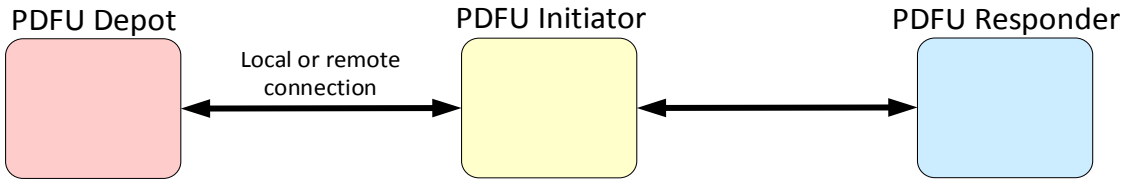
In this phase, the PDFU Responder switches to using the new firmware image. Any necessary reconfiguration is performed, typically initiated by the PDFU Initiator performing a **USB PD** Hard Reset.

3 Architecture

3.1 Overview

Figure 3-1 shows the general topology of a system that includes a PDFU Depot, PDFU Initiator, and PDFU Responder.

Figure 3-1 General PDFU Topology



3.1.1 PDFU Depot

A PDFU Depot is a collection of one or more firmware images. A PDFU Depot may be accessed by multiple parties. This allows firmware updates between PDFU Initiators and PDFU Responders from different vendors (e.g. PDFU Responder from Vendor A can be updated by PDFU Initiator from Vendor B).

A PDFU Depot can take a number of different forms:

- Local to PDFU Initiator (e.g. memory stick).
- Remote (manufacturer specific) – the firmware image is accessed and downloaded using a vendor-dependent protocol.
- Remote (centralized for all manufacturers) – the firmware image is accessed and downloaded using a standard protocol.
- Remote (manufacturer specific with centralized index) – the index and the firmware image are accessed and downloaded using a standard protocol.

This specification does not currently define the interface between the PDFU Initiator and the PDFU Depot. However, this is anticipated to be defined in a future version of this specification.

3.1.1.1 File Naming and Hierarchy in Local Depot

To assist with firmware image identification, a local Depot shall have a top level file directory with the name PDFU. Within this directory, there may be any combination of sub-directories, firmware images and other files. A firmware image shall have a file name in the following format (except for systems that only support 8.3 format file names, see below):

convenience string-iiii-pppp-vvvvvvvvvvvvvvvv-bb-yyyymmddhhmmss.pdfu

where the individual fields are as described in Table 3-1.

Table 3-1: Firmware Image File Name Fields

Name	Meaning
Convenience string	An arbitrary string selected by the vendor, not containing a hyphen character. Example: Acme Inc. 60W power adapter
iiii	Vendor ID represented as a string of four hexadecimal digits
pppp	Product ID represented as a string of four hexadecimal digits
vvvvvvvvvvvvvvvv	Firmware Version represented as a string of sixteen hexadecimal digits. The first four hexadecimal digits correspond to *FWVersion1*, the second four hexadecimal digits correspond to *FWVersion2*, the third four hexadecimal digits correspond to *FWVersion3* and the final four hexadecimal digits correspond to *FWVersion4.*
bb	Image bank in binary coded decimal. 00 if the PDFU Responder does not support multiple image banks.
yyyymmddhhmmss	Timestamp in international date/time format. Leading zeros for months, date, hours, minutes and seconds shall be present.

All alpha characters in the hexadecimal strings, the timestamp and the file extension may be represented in upper case or in lower case.

Example file name:
Acme Inc. 60W power adapter-ac12-006b-0001000101010103-20160401093212.pdfu

In the case of systems that only support 8.3 file names, the file name shall be IIIIPPPP.PDU.

3.1.2 PDFU Initiator

A PDFU Initiator is a system (typically a laptop or PC) that has at least one **USB Type-C** port that supports the USB PD Firmware Update Requests defined in this specification. A PDFU Initiator also contains the following:

- An optional user interface.
- A means of accessing a PDFU Depot.

A PDFU Initiator does the following:

1. Enumerates PDFU Responders.
2. Queries the appropriate PDFU Depot using the information received during enumeration.
3. Applies a policy whether or not to initiate a firmware image update.
4. Retrieves a firmware image from the PDFU Depot, converts it from hexadecimal to binary (if necessary), and verifies the CRC.
5. Sends the firmware image to the PDFU Responder.
6. Validates whether or not firmware update was successful.

7. Optionally manifests the update of PDFU Responder firmware by initiating Hard Reset (i.e. in order to minimize the disruptive effect of Manifestation, a PDFU Initiator may delay manifestation of new firmware until after the PDFU Responder is reset in the course of normal use).

3.1.3 PDFU Responder

A PDFU Responder is a system that has at least one **USB Type-C** port that supports the USB PD Firmware Update Responses defined in this specification.

This specification supports four possible architectures for storing firmware images within the EEPROM of a PDFU Responder. Architecture 3 or Architecture 4 is recommended to reduce the risk of limited or no Application functionality after a failed firmware update.

These architectures are informative, for reference, and are not intended to form an exhaustive list. Variations of each of the four architectures are possible.

The architectures are described in terms of an Application Image, providing the full functionality of the device, the PDFU Responder functionality and a Bootloader. The Bootloader is firmware that is always present and is entered on power reset and, optionally, from the Application. Its main functions include verifying the Application Image is correct (e.g. by means of an image checksum). If the Application Image is not correct, then, where the device contains more than one Application image, the Bootloader selects a fallback image and repeats the check. If the selected Application Image is correct and is executed from RAM, then the Bootloader copies it from the EEPROM to the RAM. If the selected Application Image is correct then, on completion of the optional copy and any other low-level initialization, the Bootloader transfers control to the Application. The Bootloader includes support for all Phases of PDFU so that, if there is no correct Application Image, it is capable of downloading a new Application Image. The Bootloader may transfer control to the newly loaded Application directly as part of the Manifestation Phase or may rely on the PDFU Initiator performing a Hard Reset.

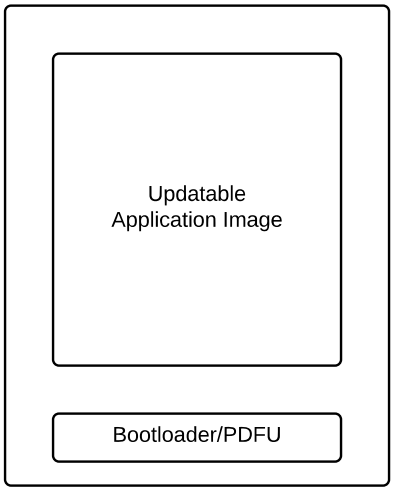
This version of this specification does not support the update of firmware images for separate applications. Vendor-dependent means may be used for combining firmware images for multiple applications into a single image for downloading via PDFU.

The Application Image always contains support for the Enumeration and Reconfiguration Phases of PDFU, and may contain support for all the phases of PDFU, allowing the Application Functionality to continue during the Transfer and Validation Phases of PDFU. If it does not contain support for all phases of PDFU then it transfers control to the Bootloader in the Reconfiguration Phase so that the Bootloader can continue with the Transfer, Validation and Manifestation Phases.

3.1.3.1 Architecture 1 - Bootloader with limited PD support and fully-featured application

Figure 3-2 illustrates Architecture 1.

Figure 3-2 Architecture 1 - Bootloader with limited PD support and fully-featured application



This is the simplest supported image storage architecture. This architecture comprises a single Application image and a Bootloader. The Bootloader contains support for all of the PDFU phases.

Key features of Architecture 1 are:

- The PDFU Responder switches to the Bootloader during the Reconfiguration phase, the Bootloader then downloads the new image into the Updatable Application Image area during the Transfer phase and, once Validation has completed successfully, the Bootloader switches to the new firmware image stored in the Updatable Application Image space during the Manifestation phase.
- Device functionality is limited to that provided by the Bootloader while the firmware image is being downloaded.
- In the event that the firmware download process fails for some reason, device functionality is limited to that provided by the Bootloader. In other words, firmware update has to be attempted again and has to succeed before full device functionality can be restored.

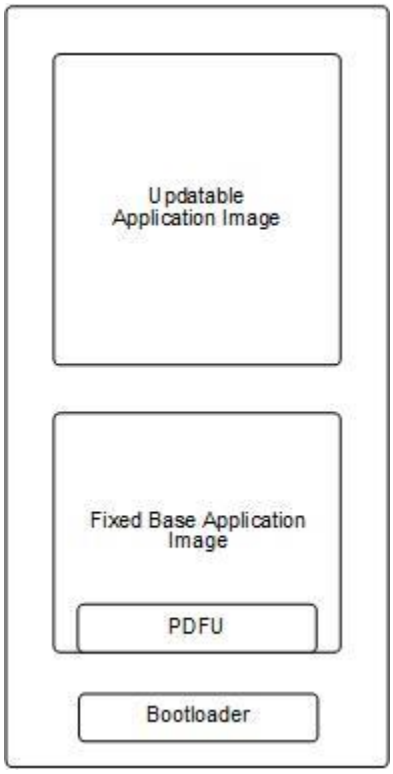
If the device executes code from RAM instead of EEPROM, then it is possible for the Application to incorporate all of the PDFU Responder functionality. In this case, the PDFU Responder in the Application can download a new firmware version into the EEPROM while maintaining full application functionality. At the Manifestation Phase, the Application switches to the Bootloader which then copies the new Application Image into RAM.

As in the previous case, if the firmware download fails for some reason, then on a power reset, the Bootloader finds that the Updatable Application Image is corrupt and waits for a new image to be downloaded.

3.1.3.2 Architecture 2 - Fixed base application with updatable application image

Figure 3-3 illustrates Architecture 2.

Figure 3-3 Architecture 2 - Fixed base application with updatable application image



This architecture comprises Bootloader, a Fixed Base Application Image that contains a PDFU Responder and an Updatable Application Image.

Key features of Architecture 2 are:

- The Updatable Application Image supports the Enumeration and Reconfiguration Phases of PDFU. In the Reconfiguration Phase, it switches to the PDFU Responder in the Fixed Base Application Image.
- The PDFU Responder in the Fixed Base Application Image overwrites the current firmware image in the Updatable Application Image area with a new firmware image during the Transfer Phase.
- The PDFU Responder in the Fixed Base Application verifies the new image during the Validation Phase. If Validation is successful, then it switches to the new firmware image stored in the Updatable Application image space during the Manifestation phase, or the device is reset, the Bootloader is entered, and it transfers to the new firmware image in the Updatable Application image space.
- Device functionality is limited to that provided by the Fixed Base Application while the firmware image is being downloaded.
- In the event that the firmware download process fails for some reason, device functionality falls back to that provided by the Fixed Base Application. In other words, while much functionality continues to be provided, firmware update has to be

attempted again and has to succeed before the latest device functionality can be restored.

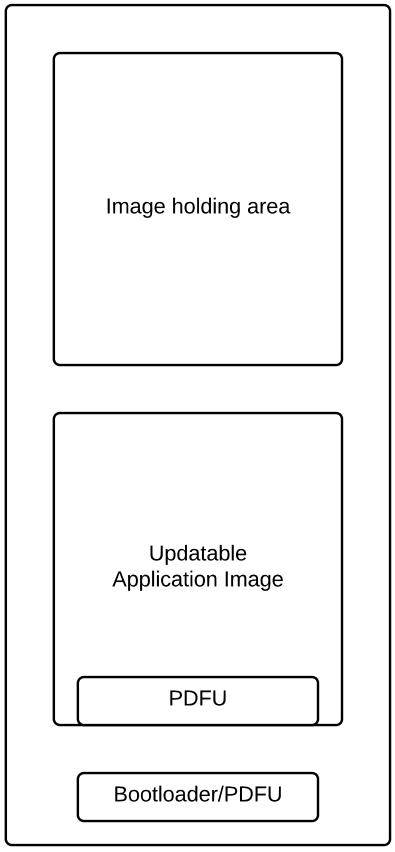
If the device executes code from RAM instead of EEPROM, then it is possible for the Application to incorporate all of the PDFU Responder functionality. In this case, the PDFU Responder in the Application can download a new firmware version into the EEPROM while maintaining full application functionality. At the Manifestation Phase, the Application switches to the Bootloader which then copies the new Application Image into RAM.

As in the previous case, if the firmware fails for some reason, then on a power reset, the Bootloader finds that the Updatable Application Image is corrupt and falls back to the Fixed Base Application.

3.1.3.3 Architecture 3 - Single application image with holding area for new firmware

Figure 3-4 illustrates Architecture 3.

Figure 3-4 Architecture 3 - Single application image with holding area for new firmware



This architecture comprises an Updatable Application Image area and an Image Holding area in which to receive a new image. Both the Bootloader and the Application image contain the full PDFU Responder functionality

Key features of Architecture 3 are:

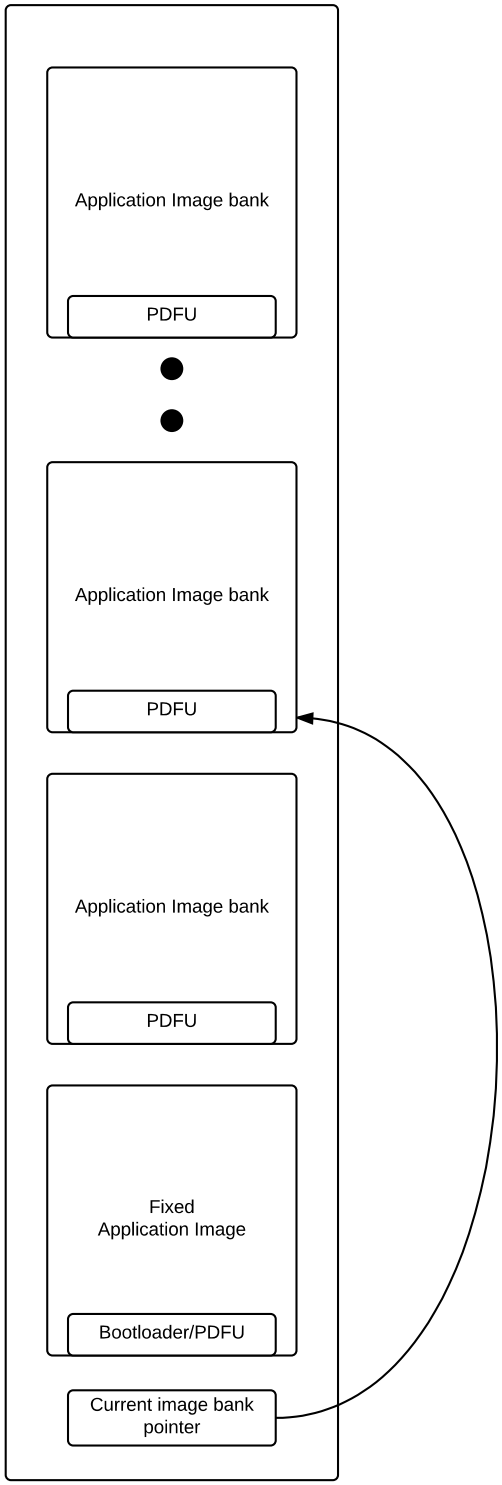
- The PDFU Responder in the Application Image downloads a new image into the Image Holding Area, while maintaining full Application functionality.
- The PDFU Responder in the Application Image validates the new image, and, if Validation is successful, switches to the Bootloader at the start of the Manifestation phase. The Bootloader copies the new firmware image from the Image Holding area into the Updatable Application Image area and transfers control to it in the Manifestation phase.
- In the event that the firmware download process fails for some reason, device functionality continues with that provided by the current version in the Updatable Application Image.

- In the event that the image copy fails for some reason the Bootloader finds the Updateable Application Image is corrupt and uses its copy of PDFU to download a new image.

3.1.3.4 Architecture 4 - Multiple fully-featured application images

Figure 3-5 illustrates Architecture 4.

Figure 3-5 Architecture 4 - Multiple fully-featured application images



This architecture comprises a number of Image Area banks, each of which can hold a different version of an Application Image. A Current Image Bank pointer indicates which bank contains the most up-to-date version of the firmware. Each Application Image incorporates the full PDFU Responder functionality. Optionally, one of the banks may contain a Fixed Application Image which is never overwritten.

Key features of Architecture 4 are:

- The PDFU Responder is responsible for managing the Image Area banks.
- The PDFU Responder downloads the new Application Image into a different bank during the Transfer phase, and finally, once Validation has completed successfully, updates a pointer to the bank containing the new firmware image during the Manifestation phase.
- Full Device functionality is available while the new firmware image is being downloaded.
- Device functionality falls back to that provided by current image in the event that the firmware download process fails for some reason.
- Multiple levels of fall back are provided in the event of image ROM failure, providing additional robustness.

3.2 Firmware Verification and Validation

3.2.1 PDFU File Prefix

The PDFU File Prefix provides a simple means for a PDFU Initiator to validate that a particular firmware image is appropriate for a PDFU Responder. The PDFU File Prefix is based on the specification for the File Suffix in **USB DFU**.

A PDFU Initiator shall not send the PDFU File Prefix from a firmware image to a PDFU Responder.

All PDFU firmware image files shall contain a PDFU File Prefix at the start of the file. The PDFU File Prefix shall take the form of a string of ASCII hexadecimal digits terminated by an ASCII Carriage return (0x0D) and ASCII line feed (0x0A).

The PDFU File Prefix shall be a hexadecimal representation of the binary information formatted according to Table 3-2 with byte offsets relative to the start of the binary information.

Table 3-2: PDFU File Prefix Data

Offset	Field	Size (Bytes)	Value	Description
0	*dwCRC*	4	Number	The CRC of the entire file, excluding *dwCRC*. Offset 0 (represented by the first two hexadecimal digits) contains the LSB of the *dwCRC* and offset 3 contains the MSB of *dwCRC*
4	*bLength*	1	23	The length of this PDFU File Prefix, including *dwCRC*, after converting to hexadecimal.
5	*ucPDFUSignature*	4	“PDFU”	The unique PDFU signature field in ASCII. Offset 5 contains the ASCII encoding for the letter “P” and offset 8 contains the ASCII encoding for the letter “U”
9	*bcdPDFU*	2	BCD	PDFU specification number. For Revision 1.0, the value of this field is 0100h. Offset 9 contains the LSB of *bcdPDFU* (i.e. 00h) and offset 10 contains the MSB of *bcdPDFU* (i.e. 01h)
11	*idVendor*	2	ID	The USB-IF-issued Vendor ID associated with this file. Offset 11 contains the LSB of *idVendor* and offset 12 contains the MSB of *idVendor*

13	*idProduct*	2	ID	The Product ID associated with this file as assigned by the vendor identified in the *idVendor* field. Offset 13 contains the LSB of *idProduct* and offset 14 contains the MSB of *idProduct*
15	*wVersionDevice1*	2	Number	The most significant component of the firmware version number of the firmware image contained in the file. Offset 15 contains the LSB of *wVersionDevice1* and offset 16 contains the MSB of *wVersionDevice1*
17	*wVersionDevice2*	2	Number	The second-most significant component of the firmware version number of the firmware image contained in the file. Offset 17 contains the LSB of *wVersionDevice2* and offset 18 contains the MSB of *wVersionDevice2*
19	*wVersionDevice3*	2	Number	The third-most significant component of the firmware version number of the firmware image contained in the file. Offset 19 contains the LSB of *wVersionDevice3* and offset 20 contains the MSB of *wVersionDevice3*
21	*wVersionDevice4*	2	Number	The least significant component of the firmware version number of the firmware image contained in the file. Offset 21 contains the LSB of *wVersionDevice4* and offset 22 contains the MSB of *wVersionDevice4*

The CRC stored in the *dwCRC* field shall be calculated over the binary representation of the Prefix Data, excluding the first four bytes, followed by the remaining bytes in the file as presented, starting with the ASCII Carriage return (0x0D) and ASCII line feed (0x0A) that terminates the Prefix. The CRC shall be calculated using the polynomial and method given in Appendix B.

3.2.2 Firmware Signature

All firmware images shall be signed by vendor-dependent means and the signature shall be contained in the firmware image (typically at the end of the image). It is recommended that the signature format be PKCS1 PSS. It is recommended that the hash algorithm be SHA256 or better. The minimum RSA key size should be 2048 bits, but it is recommended that a key size of 3072 bits or greater be used. Note that RSA 2048/3072 indicates a minimum security

recommendation, other methods with equivalent strength, for example ECDSA with P-256 curve, may also be used.

The PDFU Responder shall validate the Firmware Signature during the Validation Phase.

4 PD Firmware Update Flow

The PD Firmware Update Flow (PDFU Flow) comprises the PDFU Initiator and PDFU Responder each following successively the Phases specified in Section 2.3. The PDFU Initiator and PDFU Responder may exit the PDFU Flow from any Phase under the various error conditions, specified below.

The firmware update process is initiated and managed by the PDFU Initiator. A PDFU Responder shall not send any Requests.

A PDFU Responder may terminate firmware update during the Reconfiguration, Transfer and Validation Phases by indicating an error in the *Status* field of an appropriate Response.

During the PDFU Flow there shall be no power state changes, no data role swaps, no power role swaps, and no VCONN role swaps. If a Port receives a request for one of these during the PDFU Flow, it shall respond with a Reject message.

A PDFU Initiator shall only enter the PDFU Flow for SOP if the peer supports **USB PD**. A PDFU Initiator shall only follow the PDFU Flow for SOP' or SOP" if both the peer port and the cable support **USB PD**.

A PDFU Initiator shall follow the PDFU Flow in its entirety for each of SOP, SOP' and SOP" in succession, except that the flow for SOP' and SOP'' is skipped if **USB PD** support is not available as described in the previous paragraph.

When not in any of the PDFU phases, a PDFU Initiator shall enter the PDFU Flow at periodic intervals unless a previous Enumeration phase has determined that there is no PDFU Responder for SOP, SOP' and SOP' as appropriate. The period is not specified, but is left to system policy, possibly with user configurability. A period of one day may be typical.

4.1 PDFU Phases

4.1.1 Enumeration Phase

The PDFU Flow commences with both the PDFU Initiator and PDFU Responder entering the Enumeration phase. The PDFU Initiator and PDFU Responder enter the Enumeration phase on detecting a new connection after the Explicit Contract has been established, any appropriate Data Role Swaps, Power Role Swaps and VCONN swaps have been performed and after any appropriate Alternate Modes have been entered.

4.1.1.1 PDFU Initiator Enumeration Phase

Upon entering the Enumeration Phase, a PDFU Initiator shall attempt to enumerate a PDFU Responder. A PDFU Initiator enumerates a PDFU Responder by sending a GET_FW_ID Request to the PDFU Responder. If the PDFU Initiator does not receive a Response within *tPDFUResponseRcvd* of sending the GET_FW_ID Request, it shall resend the GET_FW_ID Request. If after resending the GET_FW_ID Request *EnumerateResend* times the PDFU Initiator does not receive a response, it shall determine that there is no PDFU Responder (note, this is separately determined for SOP, SOP' and SOP").

A PDFU Initiator exits the Enumeration Phase after either receiving a GET_FW_ID Response or determining that there is no PDFU Responder:

If a GET_FW_ID Response is received and Silent Update is either not permitted or not used, the PDFU Initiator shall inform the user that firmware update is available to take place. If the user allows the firmware update, then the PDFU Initiator shall continue to the Acquisition Phase. If the user does not allow the firmware update, then the PDFU Initiator shall exit the PDFU Flow

If a GET_FW_ID Response is received and Silent Update is used, the PDFU Initiator shall continue on to the Acquisition Phase.

Otherwise, the PDFU Initiator shall exit the PDFU Flow.

After entering the Enumeration Phase other than on a new connection, the PDFU Initiator shall send a GET_FW_ID Request to the SOP PDFU Responder.

4.1.1.2 PDFU Responder Enumeration Phase

The PDFU Responder shall transit from the Enumeration Phase to the Reconfiguration Phase on receipt of the PDFU_INITIATE Request. It shall respond to the PDFU_INITIATE Request with a PDFU_INITIATE Response from within the Reconfiguration Phase (but within *tPDFUResponseSent*).

The PDFU Responder shall exit the PDFU Flow on receipt of any USB PD message that is not a PDFU Request prior to receipt of a GET_FW_ID Request in order to process the message appropriately. Also, the PDFU Responder may exit the PDFU Flow prior to receipt of a GET_FW_ID Request in order to perform non-PDFU activity (e.g. engage in power negotiation). In both cases, the PDFU Responder shall re-enter the PDFU Flow starting at the Enumeration Phase at the completion of the non-PDFU activity.

4.1.2 Acquisition Phase

Only the PDFU Initiator uses the Acquisition Phase.

4.1.2.1 PDFU Initiator Acquisition Phase

4.1.2.1.1 Firmware Image Retrieval

The means by which a PDFU Initiator retrieves a firmware image from a remote PDFU Depot is outside the scope of this version of this specification. The means by which a PDFU Initiator retrieves a firmware image from a local PDFU Depot is described below.

A PDFU Initiator may retrieve a relevant firmware image from local temporary storage, e.g. a flash drive, either under control of a firmware update application, or automatically by scanning the media.

Relevant images are those that have an exact match for the Vendor ID, Product ID and image bank with the information retrieved by GET_FW_ID and for which the Firmware Version is greater than that retrieved by GET_FW_ID when a numerical comparison is performed (see below for details). In the event of more than one such image, the image with the latest timestamp is selected.

After retrieving a firmware image from a PDFU Depot, the PDFU Initiator shall verify that the firmware image is appropriate by validating the PDFU File Prefix as follows:

- Verify that the CRC over the firmware image file matches the *dwCRC* field.
- Verify that the *ucPDFUSignature* field is correct.
- Verify that the *bcdPDFU* field is equal to or lower than the PDFU Specification field returned in the GET_FW_ID response.
- Verify that the *idVendor* field matches the VID returned in the GET FW_ID response.
- Verify that the *idProduct* field matches the PID returned in the GET_FW_ID response.

- Verify that the values in the *wVersionDevice1-4* fields are greater than the values in the *FWVersion1-4* fields returned in the GET_FW_ID response, using the precedence defined in Table 3-2 and in Section 5.3.1.

If validation of the PDFU File Prefix succeeds and a Silent Update is permissible and used, the PDFU Initiator shall continue to the Reconfiguration Phase. Silent Updates shall not be considered permissible unless one or more of the following conditions exist:

- The PDFU Responder advertises that it accepts Silent Updates.
- The user has approved Silent Updates via an implementation-specific mechanism provided by the PDFU Initiator.
- The PDFU Initiator and PDFU Responder are from the same vendor.

Note: A firmware update can be disruptive to normal USB operation either during update or at manifestation. The possibility of such disruption should be taken into account by system policy in deciding whether or not to use a Silent Update.

If Silent Update is not allowed or is not used, then Silent Failure shall not be used. If Silent Update is used, then it is a matter of system policy whether or not to use Silent Failure.

If validation of the PDFU File Prefix succeeds and a Silent Update is not permissible or not used as a result of system policy, then the PDFU Initiator shall display a User Interface inviting the user to continue with the firmware update. The User Interface shall also display any warnings concerning loss of functionality during firmware update and/or prompts for the user to take any necessary preparatory actions (such as ejection of removable media). On confirmation from the user, the PDFU Initiator shall transit to the Reconfiguration Phase. If the user declines to proceed with the firmware update, then the PDFU Initiator shall exit the PDFU Flow.

If validation of the PDFU File Prefix fails:

- If the PDFU Initiator determines that a user has sufficient permissions, the PDFU Initiator should display a User Interface providing an override mechanism allowing the user to force a firmware update even though one or more verification checks fails. The PDFU Initiator shall not perform an override unless the user has explicitly requested it. The override mechanism may allow the PDFU Initiator to select a different image in the Depot. For example, in the case of a local Depot, the PDFU Initiator can display the names of other files with extension .pdfu. Mechanisms for querying a remote Depot are not defined in this version of the specification. The override mechanism shall include warnings that the result may cause loss of device functionality. If the user overrides the check failures, then then the PDFU Initiator shall transit to the Reconfiguration Phase.
- If the PDFU Initiator determines that a user does not have sufficient permissions, the PDFU Initiator shall exit the Firmware Update Flow without displaying a UI (typically this happens if there is no change in firmware version from that already loaded).

4.1.3 Reconfiguration Phase

4.1.3.1 PDFU Initiator Reconfiguration Phase

Upon entering the Reconfiguration Phase, the PDFU Initiator shall transmit a PDFU_INITIATE Request to the PDFU Responder. If the PDFU Initiator does not receive a response within *tPDFUResponseRcvd* of sending the PDFU_INITIATE Request, it shall resend

the PDFU_INITIATE Request. If after resending the PDFU_INITIATE Request *ReconfigureResend* times the PDFU Initiator does not receive a response, it shall exit the PDFU Flow. If either Silent Update is being used but system policy prohibits Silent Failure or Silent Update is not being used, the PDFU Initiator shall indicate an error to the user via the User Interface prior to exiting the PDFU Flow.

If the PDFU Initiator receives a PDFU_INITIATE Response with the *WaitTime* set to zero, then the PDFU Initiator shall perform any further necessary reconfiguration and then transit to the Transfer Phase. A PDFU Initiator shall not send a PDFU_DATA Request until after it transitions to the Transfer Phase.

If the PDFU Initiator receives a PDFU_INITIATE Response with a *WaitTime* greater than zero but less than 255, then the PDFU Initiator shall wait at least the requested amount of time, and then re-transmit the PDFU_INITIATE Request.

If the PDFU Initiator receives a PDFU_INITIATE Response with a *WaitTime* equal to 255, then the PDFU Initiator shall not re-transmit the PDFU_INITIATE Request and shall exit the PDFU Flow.

4.1.3.2 PDFU Responder Reconfiguration Phase

A PDFU Responder shall enter the Reconfiguration Phase after receipt of a PDFU_INITIATE Request in the Enumeration Phase.

If a PDFU Responder is able to complete any necessary reconfiguration within *tPDFUResponseSent* of receiving a PDFU_INTIATE Request, then it shall complete the reconfiguration and transmit a PDFU_INITIATE Response with the *WaitTime* field set to 0. The PDFU_INITIATE Response shall be sent within *tPDFUResponseSent* of receiving the PDFU_INTIATE Request.

If the PDFU Responder is unable to perform all of the necessary reconfiguration within *tPDFUResponseSent* of receiving a PDFU_INTIATE Request, then it shall transmit a PDFU_INITIATE Response with a non-zero *WaitTime* indicating the amount of delay required for it to complete its internal reconfiguration. The PDFU_INITIATE Response shall be sent within *tPDFUResponseSent* of receiving the PDFU_INTIATE Request. The PDFU Responder shall then await a further PDFU_INITIATE Request. If, the PDFU Responder does not receive another PDFU_INITIATE Request within the *WaitTime* plus *tPDFUNextRequestRcvd* after sending the PDFU_INITIATE Response, then the PDFU Responder shall resend the previous PDFU_INITIATE Response a further *ReconfigureResend* times, waiting for the PDFU_INITIATE Request after resending each PDFU_INITIATE Response. If the PDFU Responder does not receive a PDFU_INITITATE Request after the final PDFU_INITIATE Response is sent, then the PDFU Responder shall exit the PDFU Flow, then shall re-enter the PDFU Flow starting from the Enumeration Phase.

After sending a PDFU_INITIATE Response with the *WaitTime* field set to 0, a PDFU Responder shall wait for a PDFU_DATA Request. Upon receiving a PDFU_DATA Request, the PDFU Responder shall transit to the Transfer Phase. If the PDFU Responder does not receive a PDFU_DATA Request within *tPDFUNextRequestRcvd* of sending the PDFU_INITIATE Response, then the PDFU Responder shall resend the previous PDFU_INITIATE Response a further *DataResend* times, waiting for the initial PDFU_DATA Request after resending each PDFU_INITIATE Response. If the PDFU Responder does not receive a PDFU_DATA Request after the final PDFU_INITIATE Response is sent, then the PDFU Responder shall exit the PDFU Flow, then shall re-enter the PDFU Flow starting from the Enumeration Phase.

4.1.4 Transfer Phase

In this phase, a firmware image is transferred from the PDFU Initiator to the PDFU Responder via PDFU_DATA Requests. Firmware images are broken up into Data Blocks. Each Data Block is 256 bytes in length unless it contains the end of a firmware image. A Data Block carrying the end of a firmware image may be less than 256 bytes in length.

The PDFU Initiator starts a firmware image transfer by sending a PDFU_DATA Request that contains a Data Block with the beginning of the firmware image. Upon receiving a PDFU_DATA Request, a PDFU Responder performs flow control using the *WaitTime* and *NumDataNR* fields in the PDFU_DATA Response. The PDFU Initiator continues the transfer by sending subsequent PDFU_DATA Requests to the PDFU Responder abiding by the flow control parameters in the last received PDFU_DATA Response. Firmware image transfer is complete for the PDFU Initiator after it receives a PDFU_DATA Response for the PDFU_DATA Request carrying the last Data Block of the firmware image. Firmware image transfer is complete for the PDFU Responder after it sends a PDFU_DATA Response for the PDFU_DATA Request carrying the last Data Block of the firmware image.

Note: The internal architecture for processing USB PD messages may have bandwidth constraints, for example if the [Type-C Port Controller Interface] is used. The Responder design should use the *NumDataNR* in the PDFU Data Response (indicating the number of PDFU_DATA_NR Requests the PDFU Responder can receive before sending the next PDFU_DATA Response).

Firmware image transfer may be “paused” midway and resumed at a later time. To pause a firmware image transfer, a PDFU Initiator sends the PDFU Responder a PDFU_DATA_PAUSE Request. A PDFU Initiator shall only send a PDFU_DATA_PAUSE Request when a firmware image transfer is in progress (i.e. after the first Data Block of the firmware image is sent, but before the last Data Block is sent).

A PDFU Responder that receives a PDFU_DATA_PAUSE Request shall respond with a PDFU_DATA_PAUSE Response that either rejects (*Status* field = *errREJECT_PAUSE*) or accepts (*Status* field = *OK*) the pause. A PDFU Responder that rejects a pause shall discard the partially received firmware update and exit the PDFU flow, then shall re-enter the PDFU Flow starting from the Enumeration Phase. A PDFU Responder that accepts a pause shall disable any PDFU-related timeouts and shall remain paused in the Transfer Phase until one of the following occurs:

- The PDFU Initiator resumes the firmware image transfer by sending a PDFU_DATA Request that picks up where the transfer previously left off
- The PDFU Responder receives a PDFU_ABORT Request
- The PDFU Responder is disconnected from the PDFU Initiator
- The PDFU Responder undergoes a Hard Reset

If the PDFU Initiator does not receive a PDFU_DATA_PAUSE Response within *tPDFUResponseRcvd* of sending the PDFU_DATA_PAUSE Request, it shall resend the PDFU_DATA_PAUSE Request. If after resending the PDFU_DATA_PAUSE Request *PauseResend* times the PDFU Initiator does not receive a response, it shall exit the PDFU Flow. If either Silent Update is being used but system policy prohibits Silent Failure or Silent Update is not being used, the PDFU Initiator shall indicate an error to the user via the User Interface prior to exiting the PDFU Flow.

Note that the rules and requirements in this specification prohibiting certain other activities during firmware update still apply when the update is paused. For example, if a PDFU

Initiator needs to change the Power Contract in the middle of a firmware update, it aborts the current firmware update, renegotiates the Power Contract, then begins a new firmware update starting from the Enumeration Phase.

The detailed requirements for a PDFU Initiator and PDFU Responder are described in detail below.

4.1.4.1 PDFU Initiator Transfer Phase

Upon entering the Transfer Phase, the PDFU Initiator shall send a PDFU_DATA Request to the PDFU Responder where:

- If the PDFU Initiator is starting a new firmware image transfer, the PDFU_DATA Request shall contain the first 256 bytes of the firmware image in the *DataBlock* field (or the full firmware image if it is less than 256 bytes). The *DataBlockIndex* shall be zero, to indicate that the PDFU_DATA Request carries the first block of data.

- If the firmware update is resuming from partway through a previous firmware image transfer or if the PDFU Responder architecture contains areas that do not need updating, the PDFU_DATA Request may contain a data block other than the first data block. When the PDFU Responder requests Data Blocks out of order, the transfer of the image is considered complete when the PDFU Initiator transmits the final Data Block of the image.

Table 4-1 describes how a PDFU Initiator shall behave after receiving a PDFU_DATA Response as based on the fields in the received Response.

Table 4-1: PDFU Initiator Response

PDFU_DATA Response Payload Field				PDFU Initiator Behavior
Status	*WaitTime*	*NumDataNR*	*DataBlockNum*	
OK	less than 255	greater than zero	N	The PDFU Initiator shall wait *WaitTime* (which may be zero) and then may send up to *NumDataNR* PDFU_DATA_NR Requests without waiting for a PDFU_DATA Response after each one. It shall then send a PDFU_DATA Request Message. It shall always send the final Data Block of the firmware image as a PDFU_DATA Request and not as a PDFU_DATA_NR Request. The PDFU Initiator shall send PDFU_DATA_NR Requests in order of increasing DataBlockIndex, starting from Data Block N in the image indicated in the *DataBlockNum* field in the Response.
OK	less than 255	zero	N	The PDFU Initiator shall wait *WaitTime* (which may be zero) before sending the next PDFU_DATA Request. The next PDFU_DATA Request shall continue the firmware update from Data Block N in the image indicated in the *DataBlockNum* field in the Response.
OK	255	X	X	If the PDFU Initiator has further Data Blocks to be sent i.e. it has not yet transmitted the final Data Block of the image, the PDFU Initiator shall send a PDFU_ABORT to the PDFU Responder and exit the PDFU Flow. If either Silent Update is being used but system policy prohibits Silent Failure or Silent Update is not being used, the PDFU Initiator shall indicate an error to the user via the User Interface prior to exiting the PDFU Flow. If the PDFU Initiator has no further Data Blocks to be sent i.e. it has just transmitted the final Data Block of the image, then the PDFU Initiator shall transit to the Validation Phase.
error (i.e. non-zero)	X	X	X	PDFU Initiator shall exit the PDFU Flow. If either Silent Update is being used but system policy prohibits Silent Failure or Silent Update is not being used, the PDFU Initiator shall indicate an error to the user via the User Interface prior to exiting the PDFU Flow.

X = don’t care

If a PDFU Initiator receives a PDFU_DATA Response with *NumDataNR* set to zero, the PDFU Initiator shall not send any more PDFU_DATA_NR Responses to that PDFU Responder until after it receives a subsequent PDFU_DATA Response with a non-zero value for *NumDataNR*.

A PDFU Initiator shall always use a PDFU_DATA Request to carry the final Data Block of the firmware image. A PDFU Initiator shall end a transfer by either sending a PDFU_DATA Request with zero payload or sending a PDFU_DATA Request with less than 256 bytes of payload.

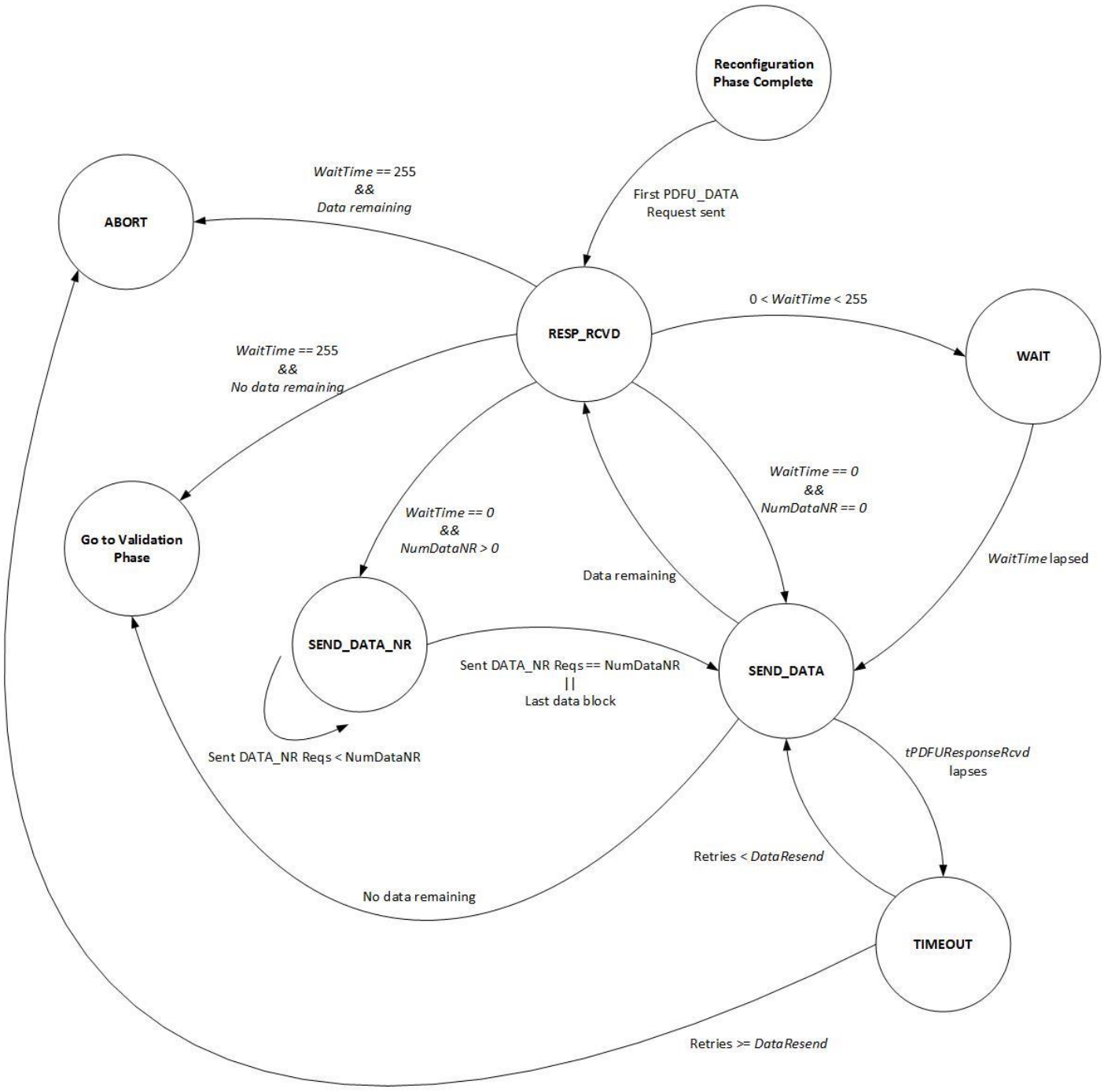
After receipt of a PDFU_DATA Response a PDFU Initiator shall send the following PDFU_DATA, PDFU_DATA_NR or PDFU_DATA_PAUSE Request within the specified *WaitTime* plus *tPDFUNextRequestSent*.

After sending a PDFU_DATA_NR Request, a PDFU Initiator shall send the following PDFU_DATA, PDFU_DATA_NR or PDFU_DATA_PAUSE Request within *tPDFUNextRequestSent*.

If at any point the PDFU Initiator does not receive a response within *tPDFUResponseRcvd* after sending the PDFU_DATA Request, it shall resend the PDFU_DATA Request. If after resending the PDFU_DATA Request *DataResend* times the PDFU Initiator does not receive a PDFU_DATA Response, it shall exit the PDFU Flow. If either Silent Update is being used but system policy prohibits Silent Failure or Silent Update is not being used, the PDFU Initiator shall indicate an error to the user via the User Interface prior to exiting the PDFU Flow.

The actions performed by the PDFU Initiator during the Transfer Phase are illustrated in Figure 4-1.

Figure 4-1 PDFU Initiator Transfer Phase State Diagram



4.1.4.2 PDFU Responder Transfer Phase

A PDFU Responder shall respond to a PDFU_DATA or PDFU_DATA_PAUSE Request with a PDFU_DATA or PDFU_DATA_PAUSE Response respectively, within *tPDFUResponseSent* of receiving the Request. The PDFU Responder shall then wait for a subsequent PDFU_DATA, PDFU_DATA_NR or PDFU_DATA_PAUSE Request.

If the PDFU Responder is expecting more Data Blocks and the PDFU Responder does not receive the next PDFU_DATA, PDFU_DATA_NR, or PDFU_DATA_PAUSE Request within *tPDFUNextRequestRcvd* after sending a PDFU_DATA Response (but not after sending a PDFU_DATA_PAUSE Response), then the PDFU Responder shall resend the previous PDFU_DATA Response. If after resending the PDFU_DATA Response *DataResend* times the PDFU Responder does not receive the next PDFU_DATA, PDFU_DATA_NR or PDFU_DATA_PAUSE Request, the PDFU Responder shall exit the PDFU Flow, then shall re-enter the PDFU Flow starting from the Enumeration Phase.

The PDFU Responder shall respond to all PDFU_DATA and PDFU_DATA_PAUSE Requests received. The PDFU Responder shall not respond to any PDFU_DATA_NR Requests.

Table 4-2 describes how a PDFU Responder shall populate the fields of a PDFU_DATA Response.

Table 4-2: PDFU Responder PDFU_DATA Response Field Values

Condition	PDFU_DATA Response Payload Field			
	Status	*WaitTime*	*NumDataNR*	*DataBlockNum*
Data from PDFU_DATA Request was successfully received and PDFU Responder is ready to receive the next (and only the next) Data Block without delay.	OK	0	0	next Data Block to send
Data from PDFU_DATA Request was successfully received and PDFU Responder is ready to receive the next (and only the next) Data Block after the specified delay.	OK	delay in ms between 1 and 254	0	next Data Block to send
Data from PDFU_DATA Request was successfully received and PDFU Responder is ready to receive multiple Data Blocks without delay.	OK	0	(number of Data Blocks the PDFU Responder is able to receive before sending the next PDFU_DATA Response) minus 1	next Data Block to send
Data from PDFU_DATA Request was successfully received and PDFU Responder is ready to receive multiple Data Blocks after the specified delay.	OK	delay in ms between 1 and 254	(number of Data Blocks the PDFU Responder is able to receive before sending the next PDFU_DATA Response) minus 1	next Data Block to send
The PDFU Responder receives a PDFU_DATA Request with more data in the Data Block than it has room for or is expecting	errADDRESS	255	0	0
The PDFU Responder receives a PDFU_DATA Request with a zero length Data Block but detects that the firmware image is not complete	errNOTDONE	255	0	0
The PDFU Responder encounters an internal error, or has other reason for aborting the firmware update	error value from Table 5-29	255	0	0

The PDFU Responder shall transit to the Validation Phase after either:

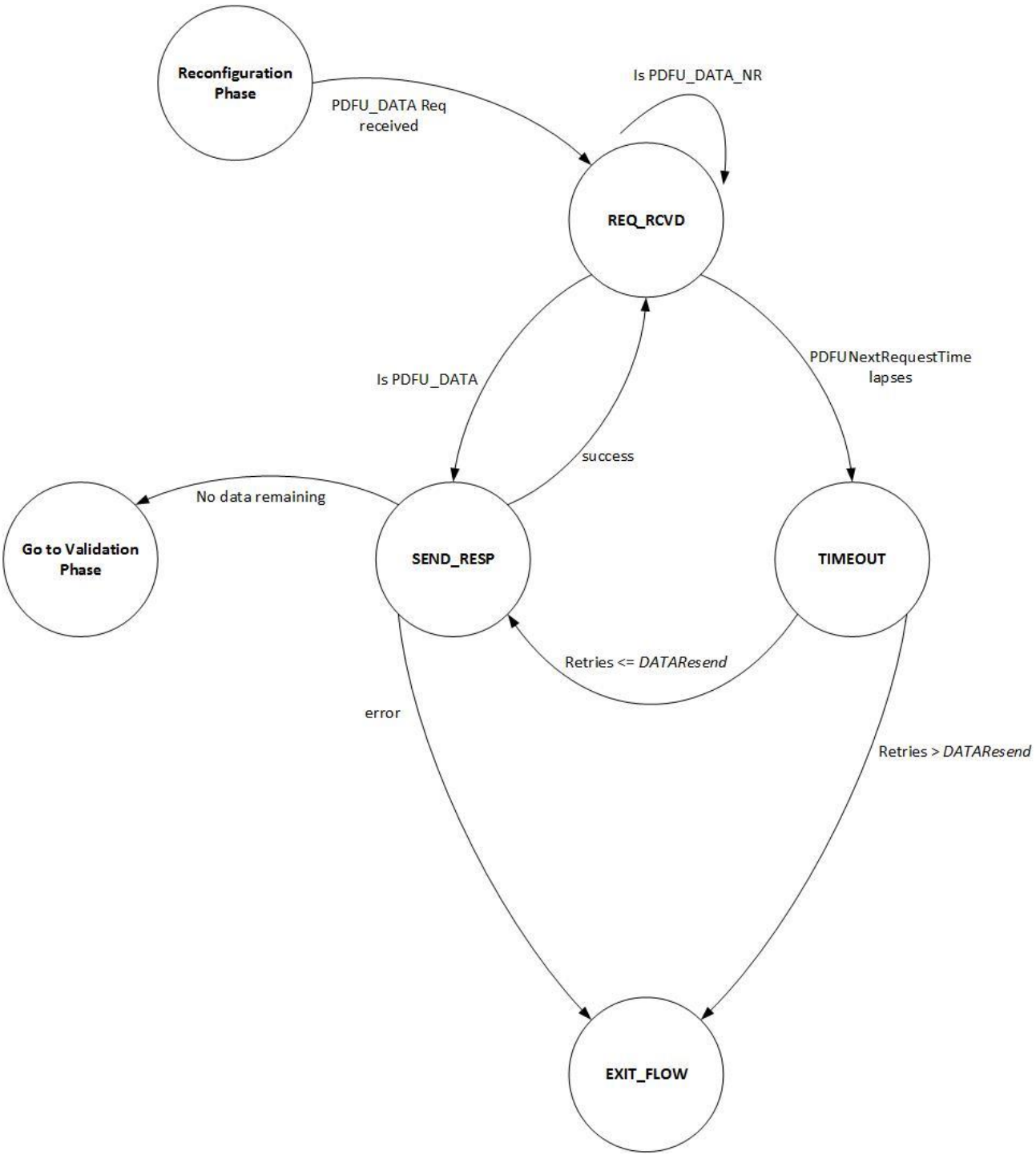
- Sending a PDFU_DATA Response in response to receiving the final block of data; or
- Sending a PDFU_DATA Response in response to a PDFU_DATA Request with a Data Block of zero size that it does not error (with *WaitTime* = 255 in the Response).

The PDFU Responder shall ensure that areas of ROM that are not being updated are not overwritten. Such areas can include fixed areas of EEPROM (such as those used to store the BootROM and/or a Fixed Application Image) or areas used for other copies of the firmware image. Accidental overwriting can occur, for example, in the event of too much data being received or a Data Block index being out of range for the area designated to receive the image. The PDFU Responder may skip downloading such areas by not requesting the corresponding Data Blocks.

Note that PDFU_DATA and PDFU_DATA_NR Requests are not retried in the event of GoodCRC not being received. It is therefore necessary for the PDFU Responder to check the Data Block index in every PDFU_DATA or PDFU_DATA_NR Request received.

The actions performed by the PDFU Responder during the Transfer Phase are illustrated in Figure 4-2.

Figure 4-2 PDFU Responder Transfer Phase State Diagram



4.1.5 Validation Phase

4.1.5.1 PDFU Initiator Validation Phase

Upon entering the Validation Phase, the PDFU Initiator shall transmit a PDFU_VALIDATE Request within *tPDFUNextRequestSent* of receiving the previous PDFU_DATA Response and await the response.

If the PDFU Initiator does not receive a response within *tPDFUResponseRcvd* of sending the PDFU_VALIDATE Request, it shall resend the PDFU_VALIDATE Request. A PDFU Initiator shall not resend a PDFU_VALIDATE Request more than *ValidateResend* times. If after resending the PDFU_VALIDATE Request *ValidateResend* times the PDFU Initiator does not receive a response, it shall exit the PDFU Flow. If either Silent Update is being used but system policy prohibits Silent Failure or Silent Update is not being used, the PDFU Initiator shall indicate an error to the user via the User Interface prior to exiting the PDFU Flow.

If the PDFU Initiator receives a PDFU_VALIDATE Response with a *WaitTime* greater than zero but less than 255, then the PDFU Initiator shall wait the requested amount of time, and then re-transmit the PDFU_VALIDATE Request.

On reception of a PDFU_VALIDATE Response indicating that validation was successful, the PDFU Initiator shall transit to the Manifestation Phase.

On reception of a PDFU_VALIDATE Response indicating that validation failed, the PDFU Initiator shall exit the PDFU Flow. If either Silent Update is being used but system policy prohibits Silent Failure or Silent Update is not being used, the PDFU Initiator shall indicate an error to the user via the User Interface prior to exiting the PDFU Flow.

4.1.5.2 PDFU Responder Validation Phase

After receiving a PDFU_VALIDATE Request, a PDFU Responder shall verify that the received firmware image is correctly signed. It may also perform other vendor-dependent validation and image management.

If the PDFU Responder does not receive a PDFU_VALIDATE Request within *tPDFUNextRequestRcvd* of sending the last PDFU_DATA Response, then the PDFU Responder shall resend the last PDFU_DATA Response sent prior to transitioning to the Validation Phase. If after resending the PDFU_DATA Response *ValidateResend* times the PDFU Responder does not receive a PDFU_VALIDATE Request, the PDFU Responder shall exit the PDFU Flow, then shall re-enter the PDFU Flow starting from the Enumeration Phase.

If the PDFU Responder is unable to complete validation and image management within *tPDFUResponseSent* of receiving the PDFU_VALIDATE Request, then it shall transmit a PDFU_ VALIDATE Response indicating the amount of delay required for it to complete its internal reconfiguration. The PDFU Validate Response shall be sent within *tPDFUResponseSent* of receiving the PDFU_VALIDATE Request. The PDFU Responder shall then await a further PDFU_ VALIDATE Request. If the PDFU Responder does not receive a subsequent PDFU_VALIDATE Request after *WaitTime* plus *tPDFUNextRequestRcvd*, then the PDFU Responder shall resend the PFDU_VALIDATE Response. If after resending the PDFU_VALIDATE Response *ValidateResend* times the PDFU Responder does not receive a PDFU_VALIDATE Request, the PDFU Responder shall exit the PDFU Flow, then shall re-enter the PDFU Flow starting from the Enumeration Phase.

If the PDFU Responder is able to complete validation and image management within *tPDFUResponseSent* of receiving the PDFU_VALIDATE Request, then it shall complete the validation and transmit a PDFU_ VALIDATE Response indicating that no further delay is needed and providing the validation status. The PDFU_VALIDATE Response shall be sent within *tPDFUResponseSent* of receiving the PDFU_VALIDATE Request.

Firmware image management is normally carried out after the image has been validated, and can include copying the image from one location to another, or updating a pointer to the new image bank, etc.

If validation is successful, the PDFU Responder shall transit to the Manifestation Phase after transmitting the PDFU_VALIDATE Response.

If validation or firmware image management fails, then the PDFU Responder shall exit the PDFU Flow, then shall re-enter the PDFU Flow starting from the Enumeration Phase.

4.1.6 Manifestation Phase

This phase follows immediately after successful Validation. There is no separate Request to start or synchronize the actions in this Phase. On entry to the Manifestation Phase, the new firmware image is in place and ready to be used.

4.1.6.1 PDFU Initiator Manifestation Phase

The PDFU Initiator shall initiate a Hard Reset or Cable Reset as appropriate if requested by the PDFU Responder in its GET_FW_ID Response. The PDFU Initiator may also initiate Hard Reset if necessary for internal purposes.

Note: Power provision (either Voltage, Current or both) is sometimes reduced as a result of a Hard Reset.

The effect of Hard Reset is to reset the Port State machine. This has the by-product of an exit from the PDFU Flow and is treated as a normal exit from the PDFU Flow.

If the PDFU Initiator does not issue a Hard Reset, then it shall then exit the PDFU Flow (after issuing the Cable Reset if necessary).

If Silent Update is not being used (see Section 4.1.2.1.1), the PDFU Initiator shall inform the user of the status of the firmware update before exiting the Manifestation Phase.

4.1.6.2 PDFU Responder Manifestation Phase

If the PDFU Responder indicated that Hard Reset or Cable Reset is necessary in its GET_FW_ID Response, then it shall wait for the PDFU Initiator to generate the Reset. The effect of the Reset is to reset the Port State machine and to re-enter the Bootloader. This has the by-product of an exit from the PDFU Flow and is treated as a normal exit from the PDFU Flow.

Note: Power provision (either Voltage, Current or both) is sometimes reduced as a result of a Hard Reset.

If the PDFU Responder indicated that Hard Reset or Cable Reset was not necessary, then the PDFU Responder shall exit the PDFU Flow (transferring control back to the Application), then shall re-enter the PDFU Flow starting from the Enumeration Phase.

A PDFU Responder shall not update its firmware version information in response to a GET_FW_ID Request until after it has exited the PDFU Flow that resulted in a successful update.

4.2 Mitigation to USB Data loss and Power Change

PDFU Initiators and PDFU Responders should take precautions to avoid USB Data loss during firmware update. Such precautions may include:

- Requesting un-mounting of any removable storage where applicable

It should be noted that a Hard Reset results in USB re-enumeration and interrupts any USB traffic taking place at the time.

PDFU Initiators and PDFU Responders should also take precaution against power fluctuations during firmware update. Such precautions may include:

- Reducing power consumed or provided prior to firmware update

4.3 Termination

4.3.1 By PDFU Initiator

A PDFU Initiator may terminate a PDFU Flow by sending a PDFU_ABORT Request at any time, and then exiting the PDFU Flow. This is not shown in the detailed description of the Phases but should be assumed. If a PDFU Initiator is powered off during the PDFU Flow (e.g. when shut down by a user), it is recommended that the PDFU Initiator send a PDFU_ABORT Request before powering off.

A PDFU Responder shall exit the PDFU Flow on receipt of a PDFU_ABORT Request, then shall re-enter the PDFU Flow starting from the Enumeration Phase. It shall not issue a PDFU message in response to a PDFU_ABORT.

If the PDFU Responder receives a PDFU_ABORT Request while it is executing from the Bootloader, then it should verify the integrity of the Application image before transferring control back to the Application, remaining in the Bootloader if there is no valid Application image.

4.3.2 By PDFU Responder

A PDFU Responder may terminate a PDFU Flow during the Transfer Phase by sending an appropriate error in the *Status* field of a PDFU_DATA Response. *WaitTime* and *NumDataNR* shall be zero.

If the PDFU Initiator receives a PDFU_DATA Response with a non-zero value in the *Status* field, it shall exit the PDFU flow. If either Silent Update is being used but system policy prohibits Silent Failure or Silent Update is not being used, the PDFU Initiator shall indicate an error to the user via the User Interface prior to exiting the PDFU Flow.

5 Firmware Update Messages

Firmware Update Messages are used to convey information related to firmware update. A Firmware Update Message consists of a 2-byte Message Header followed by a variable length (including zero) payload. The format for a Message Header is defined in Section 5.1.

There are two types of Firmware Update Messages:

- Requests
- Responses

Requests are defined in Section 5.2. Responses are defined in Section 5.3.

Firmware Update Messages are transmitted between a PDFU Initiator and PDFU Responder using the transfer mechanisms defined in **USB PD** where a Request is sent in the Firmware Update Request Data Block (FRQDB) of a Firmware Update Request Extended Message and a Firmware Update Response is sent in the Firmware Update Response Data Block (FRPDB) of a Firmware Update Response Extended Message.

Multi-byte fields that contain numerical values are formatted at consecutive offsets, and shall be formatted with the LSB at the lower offset and the MSB at the higher offset. Fields that contain firmware data bytes shall be formatted with the bytes in the same order as they are in the file containing the firmware image.

5.1 Header

All Firmware Update Messages shall start with the 2-byte header defined in Table 5-1.

Table 5-1: Firmware Update Message Header

Offset	Field	Size	Reference
0	*ProtocolVersion*	1	Section 5.1.1
1	*MessageType*	1	Section 5.1.2

5.1.1 Protocol Version

This field identifies which version of the USB PD Firmware Update Specification is being used. Table 5-2 shows the valid values for this field. A Product shall not use a Protocol Version value corresponding to a specification revision that it does not support.

Table 5-2: USB PD Firmware Update Protocol Version

Name	Value	Meaning
Reserved	00h	Reserved
V1.0	01h	USB PD Firmware Update Protocol Version 1.0
Reserved	02h-ffh	Reserved

5.1.2 Message Type

This field identifies Firmware Update Message type and shall contain one of the Firmware Update Message Types listed in Table 5-3 (Requests) or Table 5-16 (Responses).

5.2 Requests

Requests are used to send a command to the recipient and/or retrieve data. Request types are listed in Table 5-3.

Table 5-3: Firmware Update Message Request Types

Value	Description
00h – 7Fh	Only used for Responses
80h	Reserved
81h	GET_FW_ID
82h	PDFU_INITIATE
83h	PDFU_DATA
84h	PDFU_DATA_NR
85h	PDFU_VALIDATE
86h	PDFU_ABORT
87h	PDFU_DATA_PAUSE
88h - FEh	Reserved
FFh	VENDOR_SPECIFIC

5.2.1 GET_FW_ID

This Request is used to retrieve information about a PDFU Responder and determine if a firmware update is necessary. The header for a GET_FW_ID Request is defined in Table 5-4. A GET_FW_ID Request has no payload.

Table 5-4: GET_FW_ID Request Header

Offset	Field	Size	Value
0	*ProtocolVersion*	1	V1.0
1	*MessageType*	1	GET_FW_ID

5.2.2 PDFU_INITIATE

This Request is used to initiate firmware update. The header for a PDFU_INITIATE Request is defined in Table 5-5. The Payload for a PDFU_INITIATE Request is defined in Table 5-6.

Table 5-5: PDFU_INITIATE Request Header

Offset	Field	Size	Value
0	*ProtocolVersion*	1	V1.0
1	*MessageType*	1	PDFU_INITIATE

Table 5-6: PDFU_INITIATE Request Payload

Offset	Field	Size	Value
2	*FWVersion1*	2	Most significant component of the firmware version of the update
4	*FWVersion2*	2	Second-most significant component of the firmware version of update
6	*FWVersion3*	2	Third-most significant component of the firmware version of update
8	*FWVersion4*	2	Least significant component of the firmware version of update

5.2.3 PDFU_DATA

This Request is used to transfer a data block from a firmware image to a PDFU Responder where a response is required. The header for a PDFU_DATA Request is defined in Table 5-7. The Payload for a PDFU_DATA Request is defined in Table 5-8.

Table 5-7: PDFU_DATA Request Header

Offset	Field	Size	Value
0	*ProtocolVersion*	1	V1.0
1	*MessageType*	1	PDFU_DATA

Table 5-8: PDFU_DATA Request Payload

Offset	Field	Size	Value
2	*DataBlockIndex*	2	The index into the firmware image of the Data Block being transmitted where each Data Block contains 256 bytes *DataBlockIndex* shall be 0 for the Request carrying the first Data Block of a firmware image and shall increment by 1 for each subsequent Data Block of the firmware image.
4	*DataBlock*	Varies	Data Block containing a section of the firmware image. A data block shall be 256 bytes in length unless it is the last data block in a firmware update transfer. A data block that is the last data block in a firmware update transfer may be less than 256 bytes in length, but shall not exceed 256 bytes.

5.2.4 PDFU_DATA_NR

This Request is used to transfer a data block from a firmware image to a PDFU Responder where a response is not required. The header for a PDFU_DATA_NR Request is defined in Table 5-7. The Payload for a PDFU_DATA_NR Request is defined in Table 5-8.

Table 5-9: PDFU_DATA_NR Request Header

Offset	Field	Size	Value
0	*ProtocolVersion*	1	V1.0
1	*MessageType*	1	PDFU_DATA_NR

Table 5-10: PDFU_DATA_NR Request Payload

Offset	Field	Size	Value
2	*DataBlockIndex*	2	The index into the firmware image of the Data Block being transmitted where each Data Block contains 256 bytes. *DataBlockIndex* shall increment by 1 for each subsequent Data Block of the firmware image.
4	*DataBlock*	Varies	Data Block containing a section of the firmware image. The data block shall be 256 bytes in length.

5.2.5 PDFU_VALIDATE

This Request is used to request validation of a firmware image. The header for a PDFU_VALIDATE Request is defined in Table 5-7. A PDFU_VALIDATE Request has no payload.

Table 5-11: PDFU_VALIDATE Request Header

Offset	Field	Size	Value
0	*ProtocolVersion*	1	V1.0
1	*MessageType*	1	PDFU_VALIDATE

5.2.6 PDFU_ABORT

This Request is used to end firmware image update prematurely. The header for a PDFU_ABORT Request is defined in Table 5-12. A PDFU_ABORT Request has no payload.

Table 5-12: PDFU_ABORT Request Header

Offset	Field	Size	Value
0	*ProtocolVersion*	1	V1.0
1	*MessageType*	1	PDFU_ABORT

5.2.7 PDFU_DATA_PAUSE

This Request is used to pause a firmware image update in the Transfer Phase before the firmware image transfer is complete. The header for a PDFU_DATA_PAUSE Request is defined in Table 5-12. A PDFU_DATA_PAUSE Request has no payload.

Table 5-13: PDFU_DATA_PAUSE Request Header

Offset	Field	Size	Value
0	*ProtocolVersion*	1	V1.0
1	*MessageType*	1	PDFU_DATA_PAUSE

5.2.8 VENDOR_SPECIFIC

This Request is for vendor-specific use. The header for a VENDOR_SPECIFIC Request is defined in Table 5-14. The payload for a VENDOR_SPECIFIC Request is defined by the vendor except that the first 2 bytes shall always contain the VID of the vendor defining the rest of the payload fields.

Table 5-14: VENDOR_SPECIFIC Request Header

Offset	Field	Size	Value
0	*ProtocolVersion*	1	V1.0
1	*MessageType*	1	VENDOR_SPECIFIC

Table 5-15: VENDOR_SPECIFIC Request Payload

Offset	Field	Size	Value
2	*VID*	2	VID allocated by the USB-IF
4	*Vendor Defined*	Varies up to 256	Vendor defined

5.3 Responses

Responses are only sent in response to a Request. Response types are listed in Table 5-16.

Table 5-16: Message Response Types

Value	Description
00h	Reserved
01h	GET_FW_ID
02h	PDFU_INITIATE
03h	PDFU_DATA
04h	Reserved
05h	PDFU_VALIDATE
O6h	Reserved
07h	PDFU_DATA_PAUSE
08h-7Eh	Reserved
7Fh	VENDOR_SPECIFIC
80h – FFh	Only used by Requests

5.3.1 GET_FW_ID

This Response is used to respond to a GET_FW_ID Request. The header for a GET_FW_ID Response is defined in Table 5-17. The Payload for a GET_FW_ID Response is defined in Table 5-18.

Table 5-17: GET_FW_ID Response Header

Offset	Field	Size	Value
0	*ProtocolVersion*	1	V1.0
1	*MessageType*	1	GET_FW_ID

Table 5-18: GET_FW_ID Response Payload

Offset	Field	Size	Value
2	*Status*	1	See Table 5-29
3	*VID*	2	USB-IF assigned Vendor ID
5	*PID*	2	USB-IF assigned Product ID
7	*HWVersion*	1	Hardware Version **Bits** / **Description** 3:0 – Hardware Minor Version 7:4 – Hardware Major Version
8	*SiVersion*	1	Silicon Version **Bits** / **Description** 3:0 – Reserved 7:4 – Silicon Base Version
9	*FWVersion1*	2	Most significant component of the firmware version
11	*FWVersion2*	2	Second-most significant component of the firmware version
13	*FWVersion3*	2	Third-most significant component of the firmware version
15	*FWVersion4*	2	Least significant component of the firmware version
17	*ImageBank*	1	Image bank for which firmware is requested
18	*Flags1*	1	**Bit** / **Description** 0 – Set to 1 to indicate support for PDFU via USB PD Firmware Update flow. Otherwise shall be set to 0. 1 – Set to 1 to indicate support for DFU via **USB DFU**. Otherwise shall be set to 0. 2 – Set to 1 if firmware is not updatable. Otherwise shall be set to 0. 3 – Set to 1 if PDFU Responder allows Silent Updates. Otherwise shall be set to 0. 7:4 – Reserved
19	*Flags2*	1	**Bit** / **Description** 0 – Set to 1 if PDFU Responder is fully functional during firmware update. Otherwise shall be set to 0. 1 – Set to 1 if unplug during firmware update is safe. Otherwise, shall be set to 0. 7:2 – Reserved

<table>
<tr><th>Offset</th><th>Field</th><th>Size</th><th>Value</th></tr>
<tr><td>20</td><td>Flags3</td><td>1</td><td>
<table>
<tr><th>Bit</th><th>Description</th></tr>
<tr><td>0</td><td>Set to 1 if Hard Reset required to complete firmware update. Otherwise shall be set to 0.</td></tr>
<tr><td>1</td><td>Set to 1 if USB functionality is available during firmware update. Otherwise shall be set to 0.</td></tr>
<tr><td>2</td><td>Set to 1 if Alternate Modes are available during firmware update. Otherwise shall be set to 0 if Alternate Modes are not available during firmware update or are not supported.</td></tr>
<tr><td>3</td><td>Set to 1 if PDFU Responder has power limitations during firmware update. Otherwise shall be set to 0.</td></tr>
<tr><td>4</td><td>Set to 1 if PDFU Responder needs more power than is provided in the current Power Contract in order to support firmware update. Otherwise shall be set to 0.
If this bit is set to 1, the PDFU Initiator (Power Source) shall either take suitable action to provide new power to the PDFU Responder (Power Sink) (e.g. by issuing Source Capabilities or requesting the user to connect external power to the PDFU Responder), or abandon the Firmware Update.</td></tr>
<tr><td>7:5</td><td>Reserved</td></tr>
</table>
</td></tr>
<tr><td>21</td><td>Flags4</td><td>1</td><td>
<table>
<tr><th>Bit</th><th>Description</th></tr>
<tr><td>0</td><td>Set to 1 if user must un-mount storage on PDFU Responder before starting firmware update. Otherwise shall be set to 0.</td></tr>
<tr><td>1</td><td>Set to 1 if user must unplug/replug cable to complete firmware update. Otherwise shall be set to 0.</td></tr>
<tr><td>2</td><td>Set to 1 if user must swap cable ends during firmware update. Otherwise shall be set to 0.</td></tr>
<tr><td>3</td><td>Set to 1 if power cycle is required to complete firmware update. Otherwise shall be set to 0.</td></tr>
<tr><td>7:4</td><td>Reserved</td></tr>
</table>
</td></tr>
</table>

The use of the *PID*, *HWVersion*, *SiVersion*, and *FWVersion1-4* fields is specified by the vendor with the USB-IF assigned Vendor ID given in the *VID* field. Later versions in the *HWVersion* and *SiVersion* shall always be arithmetically greater than earlier versions. A later version for *FWVersion1* shall always be arithmetically greater than earlier versions. A later version for *FWVersion2* for a specific *FWVersion1* shall always be arithmetically greater than earlier versions. A later version for *FWVersion3* for a specific *FWVersion1* and *FWVersion2* shall always be arithmetically greater than earlier versions. A later version for *FWVersion4* for a

specific *FWVersion1, FWVersion2* and *FWVersion3* shall always be arithmetically greater than earlier versions.

5.3.2 PDFU_INITIATE

This Response is used to respond to a PDFU_INITIATE Request. The header for a PDFU_INITIATE Response is defined in Table 5-19. The payload for a PDFU_INITIATE Response is defined in Table 5-20.

Table 5-19: PDFU_INITIATE Response Header

Offset	Field	Size	Value
0	*ProtocolVersion*	1	V1.0
1	*MessageType*	1	PDFU_INITIATE

Table 5-20: PDFU_INITIATE Response Payload

Offset	Field	Size	Value
2	*Status*	1	See Table 5-29
3	*WaitTime*	1	Carries a non-zero value when the PDFU Responder is not ready for a firmware update that indicates the time in units of 10ms (e.g. a value of 3 is equal to 30ms) that a PDFU Initiator shall wait before resending a PDFU_INITIATE Request. A value of 0 indicates that the PDFU Responder is ready to initiate firmware update. A value of 255 indicates that the PDFU Responder is unable to initiate firmware update.
4	*MaxImageSize*	3	<table><tr><th>Bit</th><th>Description</th></tr><tr><td>19:0</td><td>Maximum firmware image length in bytes that PDFU Responder can receive.</td></tr><tr><td>23:20</td><td>Reserved. Shall be set to 0.</td></tr></table>

5.3.3 PDFU_DATA

This Response is used to respond to a PDFU_DATA Request. The header for a PDFU_DATA Response is defined in Table 5-21. The payload for a PDFU_DATA Response is defined in Table 5-22.

Table 5-21: PDFU_DATA Response Header

Offset	Field	Size	Value
0	*ProtocolVersion*	1	V1.0
1	*MessageType*	1	PDFU_DATA

Table 5-22: PDFU_DATA Response Payload

Offset	Field	Size	Value
2	*Status*	1	See Table 5-29
3	*WaitTime*	1	0: Indicates that the PDFU Responder is ready to receive more data. 0 < *WaitTime* < 255: Indicates that the PDFU Responder is not ready to receive additional data. The value indicates the time in units of 1ms that a PDFU Initiator shall wait before sending data. 255: Indicates that the PDFU Responder is unable to receive any more data.
4	*NumDataNR*	1	Number of PDFU_DATA_NR Requests the PDFU Responder can receive before sending the next PDFU_DATA Response. Zero indicates that the PDFU Responder cannot receive any PDFU_DATA_NR Requests. Shall be zero if *WaitTime* is non-zero.
5	*DataBlockNum*	2	The Data Block Number of the next PDFU_DATA or PDFU_DATA_NR Request that the PDFU Initiator is to send. Shall be set to zero if *WaitTime* is 255. Note: this may not be the next Data Block to the one last sent

5.3.4 PDFU_VALIDATE

This Response is used to respond to a PDFU_VALIDATE Request. The header for a PDFU_ VALIDATE Response is defined in Table 5-23. The payload for a PDFU_ VALIDATE Response is defined in Table 5-24.

Table 5-23: PDFU_ VALIDATE Response Header

Offset	Field	Size	Value
0	*ProtocolVersion*	1	V1.0
1	*MessageType*	1	PDFU_VALIDATE

Table 5-24: PDFU_VALIDATE Response Payload

Offset	Field	Size	Value
2	*Status*	1	See Table 5-29
3	*WaitTime*	1	0: Indicates that the PDFU Responder was able to validate. 0 < *WaitTime* < 255: Indicates that the PDFU Responder is able to complete validation. The value indicates the time in units of 1ms that a PDFU Initiator shall wait before resending a PDFU_VALIDATE Request. 255: Indicates that the PDFU Responder is unable to validate and that validation is not to be retried.
4	*Flags*	1	This field shall be ignored if *WaitTime* is not 0. **Bit** / **Description** 0 / Set to 1 if validation was successful. Otherwise, shall be set to 0. 7:1 / Reserved. Shall be set to 0.

5.3.5 PDFU_DATA_PAUSE

This Response is used to respond to a PDFU_DATA_PAUSE Request. The header for a PDFU_DATA_PAUSE Response is defined in Table 5-27. The payload for PDFU_DATA_PAUSE Response is vendor defined.

Table 5-25: PDFU_DATA_PAUSE Response Header

Offset	Field	Size	Description
0	*ProtocolVersion*	1	V1.0
1	*MessageType*	1	PDFU_DATA_PAUSE

Table 5-26: PDFU_DATA_PAUSE Response Payload

Offset	Field	Size	Value
2	*Status*	1	See Table 5-29

5.3.6 VENDOR_SPECIFIC

This Response is used to respond to a VENDOR_SPECIFIC Request. The header for a VENDOR_SPECIFIC Response is defined in Table 5-27. The payload for VENDOR_SPECIFIC Response is vendor defined.

Table 5-27: VENDOR_SPECIFIC Response Header

Offset	Field	Size	Description
0	*ProtocolVersion*	1	V1.0
1	*MessageType*	1	VENDOR_SPECIFIC

Table 5-28: VENDOR_SPECIFIC Response Payload

Offset	Field	Size	Value
2	*Status*	1	See Table 5-29
3	*VID*	2	VID allocated by the USB-IF
5	*Vendor defined*	Varies up to 255	Vendor defined

5.4 Response Status

The PDFU Responder shall indicate whether a Request was successfully completed or whether an error was encountered in the *Status* field of the corresponding Response, using one of the values specified in Table 5-29.

Note: the errors with values <80h are intended to align with corresponding errors in **USB DFU**.

Table 5-29: Status Information during Firmware Update

Status	Value	Comment
OK	00h	Request completed successfully or delayed
errTarget	01h	FW not targeted for this device
errFile	02h	Fails vendor-specific verification test
errWrite	03h	Unable to write memory
errERASE	04h	Memory erase function failed
errCHECK_ERASED	05h	Memory erase check failed
errPROG	06h	Program memory function failed
errVERIFY	07h	Program memory failed verification
errADDRESS	08h	Received address is out of range
errNOTDONE	09h	Received PDFU_DATA Request with a zero length Data Block, but the PDFU Responder expects more data
errFIRMWARE	0Ah	Device's firmware is corrupt. It cannot return to normal operations.
Reserved	0Bh	Reserved
Reserved	0Ch	Reserved
errPOR	0Dh	Unexpected power on reset
errUNKNOWN	0Eh	Something went wrong
Reserved	0Fh – 7Fh	Reserved
errUNEXPECTED_HARD_RESET	80h	Used when firmware update starts after a hard reset (enumeration, etc.) that occurred in the middle of firmware update
errUNEXPECTED_SOFT_RESET	81h	Used when firmware update starts after soft reset (new power contract, etc.) that occurred in the middle of firmware update
errUNEXPECTED_REQUEST	82h	PDFU Responder received a request that is not appropriate for the current Phase
errREJECT_PAUSE	83h	PDFU Responder is unable or unwilling to pause a firmware image transfer
Reserved	84h – FFh	Reserved

5.5 Retries

USB PD Messages, including PDFU Firmware Update messages, are not retried in various circumstances when there is a Message transmission failure.

The following summarizes the rules related to retries specified in **USB PD**:

- Cable Plugs do not retry Messages

- Extended Messages of Data Size > *MaxExtendedMsgLegacyLen* that are not Chunked (Chunked flag set to zero) are not retried
- Extended Messages of Data Size ≤ *MaxExtendedMsgLegacyLen* (Chunked flag set to zero or one) are retried
- Extended Messages of Data Size > *MaxExtendedMsgLegacyLen* that are Chunked (Chunked flag set to one) individual Chunks are retried

The rules for resending Requests and Responses and the associated timeouts in this specification have been designed to provide an equivalent level of robustness to retries in **USB PD**.

5.6 Timing and Timeouts

The PDFU message timing and timeouts depends on whether chunking is used or not. If both the PDFU Initiator and PDFU Responder support unchunked Extended Messages, then chunking is not used (the *Chunked* bit in the message is set to 0). If chunking is used (the *Chunked* bit in the message is set to 1) then the message is transmitted in one or more chunks.

Figure 5-1 illustrates where timeouts are applied for PDFU Messages that comprise a single chunk. This figure applies to PDFU Messages with Data Size ≤ *MaxExtendedMsgLegacyLen* bytes, regardless of whether chunking is used or not. It also applies when chunking is not used to PDFU Messages with Data Size > *MaxExtendedMsgLegacyLen* bytes.

Figure 5-1 PDFU Retries for Single Chunk Messages

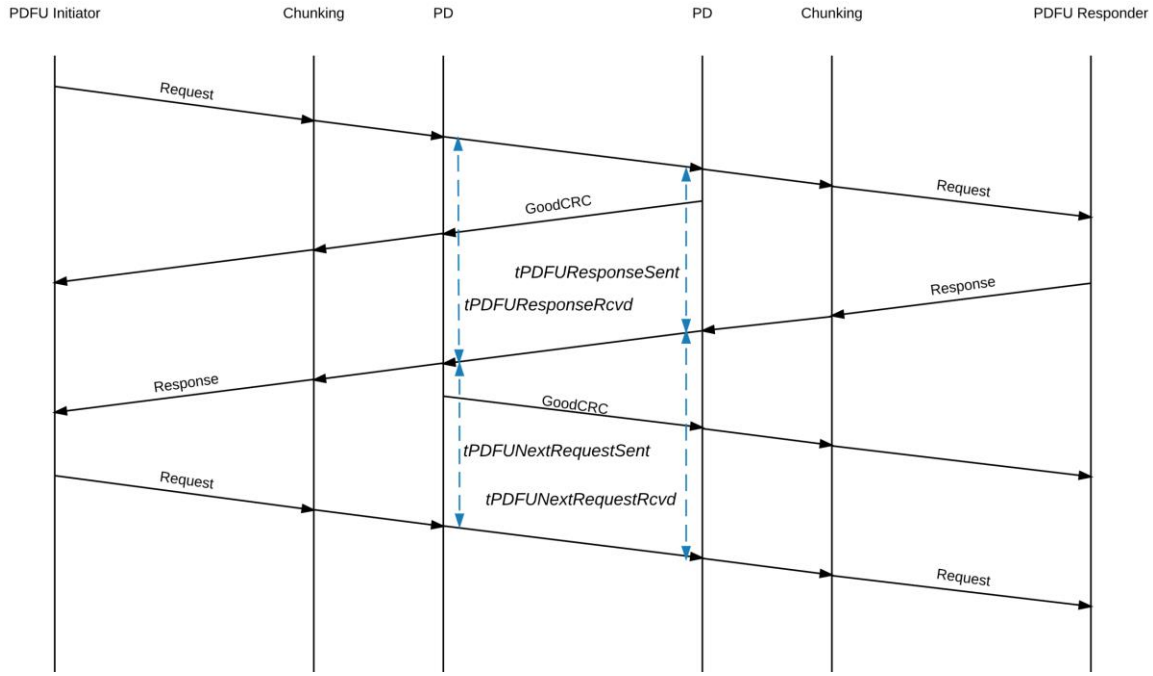
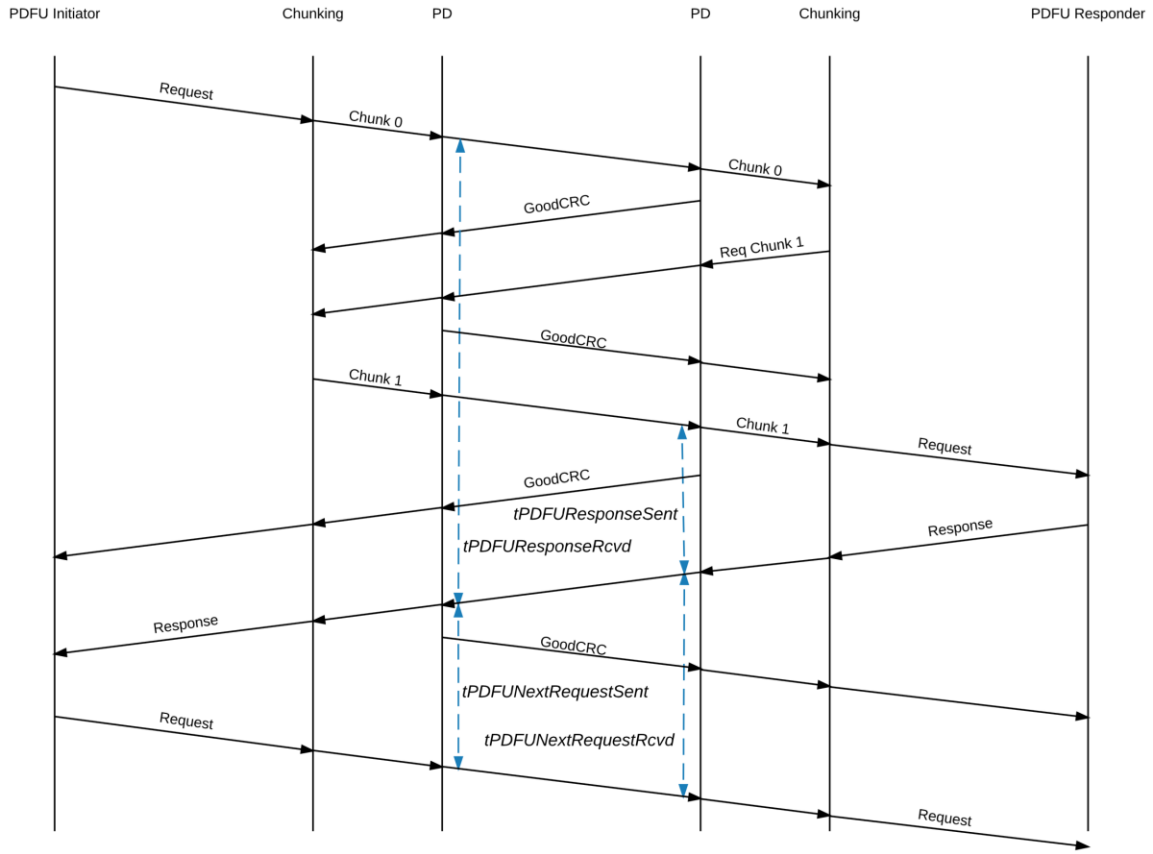


Figure 5-2 illustrates where timeouts are applied for PDFU Messages with Data Size > *MaxExtendedMsgLegacyLen* bytes that are transmitted in a multiple of chunks.

Figure 5-2 PDFU Retries for Multi-Chunk Messages



tPDFUResponseSent and *tPDFUNextRequestSent* are performance requirements, and have similar values to *tReceiverResponse* in USB PD, but with an extra 12 ms allowed for internal bus transfer time of full size Extended Messages in both directions.

tPDFUResponseRcvd and *tPDFUNextRequestRcvd* are robustness timeouts, intended to guard against transmission failures of the individual PDFU message components or corresponding *GoodCRC* messages, particularly in cases where the **USB PD** messages are not retried.

Note: When chunking is used, a VENDOR_SPECIFIC Response that is longer than *MaxExtendedMsgLegacyLen* bytes is transmitted as a sequence of chunks, otherwise all PDFU Responses are transmitted as a single chunk.

5.6.1 PDFU Initiator Timing Parameters

Table 5-30 shows the timing parameters that apply to a PDFU Initiator. The parameter *NumOutgoingChunks* is the number of chunks in the outgoing message (for example, 10

chunks in a full size PDFU_DATA message for timing *tPDFUResponseRcvd*) and the parameter *NumIncomingChunks* is the number of chunked in the following incoming message.

Table 5-30: Timeout Values for a PD Firmware Update PDFU Initiator

Parameter	Timeout Value		Description
	Chunked = 0	**Chunked = 1**	
tPDFUResponseRcvd	54 ms (min) 60 ms (max)	30 x *NumOutgoingChunks* + 30 x *NumIncomingChunks* ms	Timeout for any PDFU Request requiring a response
tPDFUNextRequestSent	27 ms	27 ms	Maximum time between receiving a PDFU Response and waiting a specified WaitTime and sending the next PDFU Request

tPDFUResponseRcvd shall be measured from when the first bit of the PDFU Message Preamble for the first chunk of the outgoing PDFU Request is transmitted to the time when the last bit of the ***EOP*** in final chunk of the expected incoming PDFU Response Message has been received by the Physical Layer.

tPDFUNextRequestSent time shall be measured from the time the last bit of the ***EOP*** in the final chunk of the incoming PDFU Response Message has been received by the Physical Layer until the first bit of the outgoing next PDFU Request Message Preamble has been transmitted by the Physical Layer.

5.6.2 PDFU Responder Timing Parameters

Table 5-31 gives timing values that apply to a PDFU Responder. The parameter *NumOutgoingChunks* is the number of chunks in the outgoing message and the parameter *NumIncomingChunks* is the number of chunked in the following incoming message.

Table 5-31: Timeout Values for a PD Firmware Update PDFU Responder

Parameter	Timeout Value		Description
	Chunked = 0	**Chunked = 1**	
tPDFUNextRequestRcvd	54 ms (min) 60 ms (max)	30 x *NumOutgoingChunks* + 30 x *NumIncomingChunks* ms	Maximum time between sending a Response and waiting a specified WaitTime and receiving the next Request
tPDFUResponseSent	27 ms	27 ms	Timeout for any PDFU Request requiring a response

tPDFUNextRequestRcvd shall be measured from the time the first bit of the PDFU Message Preamble for the first chunk of the outgoing PDFU Response is transmitted to the time when the last bit of the ***EOP*** in the final chunk of the expected incoming next PDFU Request Message has been received by the Physical Layer.

tPDFUResponseSent shall be measured from the time the last bit of the ***EOP*** in the last chunk of the incoming PDFU Request Message has been received by the Physical Layer until the first bit of the outgoing PDFU Response Message Preamble has been transmitted by the Physical Layer.

5.7 Unexpected Requests and Responses

As a result of error conditions or other failures, a PDFU Responder sometimes receives a Request which is not anticipated for the current Phase in which the PDFU Responder is operating.

Table 5-32 indicates the action that a PDFU Responder shall take after receiving an unexpected Request type. When the Request requires a Response, the PDFU Responder shall generate a Response with the *Status* byte set to *errUNEXPECTED_REQUEST*. The PDFU Responder shall then exit the PDFU Flow, then shall re-enter the PDFU Flow starting from the Enumeration Phase. For Reserved Requests, the Response *MessageType* field shall be constructed by masking the Request *MessageType* field with 7Fh.

Table 5-32: Response to Requests

Phase	Request	Expected/ Unexpected	Response
All	VENDOR_SPECIFIC	Vendor specific	If *VID* in Request matches the USB-IF assigned Vendor ID provided in the GET_FW_ID Response, then Vendor defined, otherwise notify as unexpected with *VID* field set to the VID in the Request
Enumeration	GET_FW_ID	Expected	
	PDFU_INITIATE	Expected	
	PDFU_DATA	Unexpected	Notify as unexpected
	PDFU_DATA_NR	Unexpected	Ignore
	PDFU_VALIDATE	Unexpected	Notify as unexpected
	PDFU_ABORT	Expected	
	PDFU_DATA_PAUSE	Unexpected	Ignore
	Reserved	Unexpected	Notify as unexpected
Reconfiguration	GET_FW_ID	Unexpected	Notify as unexpected
	PDFU_INITIATE	Expected	
	PDFU_DATA	Unexpected	If reconfiguration is complete, transit to Transfer Phase and treat as the first PDFU_DATA Request, otherwise notify as unexpected
	PDFU_DATA_NR	Unexpected	Ignore
	PDFU_VALIDATE	Unexpected	Notify as unexpected
	PDFU_ABORT	Expected	
	PDFU_DATA_PAUSE	Unexpected	Ignore
	Reserved	Unexpected	Notify as unexpected
Transfer	GET_FW_ID	Unexpected	Notify as unexpected
	PDFU_INITIATE	Unexpected	If no PDF_DATA request has been received, treat as valid, and provide PDFU_INITIATE Response. If PDFU_DATA request has been received, notify as unexpected
	PDFU_DATA	Expected	
	PDFU_DATA_NR	Expected	

Phase	Request	Expected/ Unexpected	Response
	PDFU_VALIDATE	Unexpected	If transfer is complete (all expected bytes of the firmware image have been received), transit to Validation Phase, otherwise notify as unexpected
	PDFU_ABORT	Expected	
	PDFU_DATA_PAUSE	Expected	
	Reserved	Unexpected	Notify as unexpected
Validation	GET_FW_ID	Unexpected	Notify as unexpected
	PDFU_INITIATE	Unexpected	Notify as unexpected
	PDFU_DATA	Unexpected	Notify as unexpected
	PDFU_DATA_NR	Unexpected	Ignore
	PDFU_VALIDATE	Expected	
	PDFU_ABORT	Expected	
	PDFU_DATA_PAUSE	Unexpected	Ignore
	Reserved	Unexpected	Notify as unexpected
Manifestation	GET_FW_ID	Unexpected	Notify as unexpected
	PDFU_INITIATE	Unexpected	Notify as unexpected
	PDFU_DATA	Unexpected	Notify as unexpected
	PDFU_DATA_NR	Unexpected	Ignore
	PDFU_VALIDATE	Unexpected	Notify as unexpected
	PDFU_ABORT	Expected	
	PDFU_DATA_PAUSE	Unexpected	Ignore
	Reserved	Unexpected	Notify as unexpected

If the PDFU Initiator receives a Response with Status *errUNEXPECTED_REQUEST*, it shall exit the PDFU flow. If either Silent Update is being used but system policy prohibits Silent Failure or Silent Update is not being used, the PDFU Initiator shall indicate an error to the user via the User Interface prior to exiting the PDFU Flow.

If a PDFU Initiator receives a Response that is not a Response to the most recent Request that it issued then it shall ignore the Response and re-issue the most recent Request.

6 Protocol Constants

Constants required by the PDFU Firmware Update Protocol are given in Table 6-1.

Table 6-1: Protocol Constants

Constant	Value
EnumerateResend	10
ReconfigureResend	3
DataResend	3
ValidateResend	3
PauseResend	3

Appendix A Transfer Phase Flow Diagrams (Informative)

Figure A-1 PDFU Initiator Transfer Phase Flow (Informative)

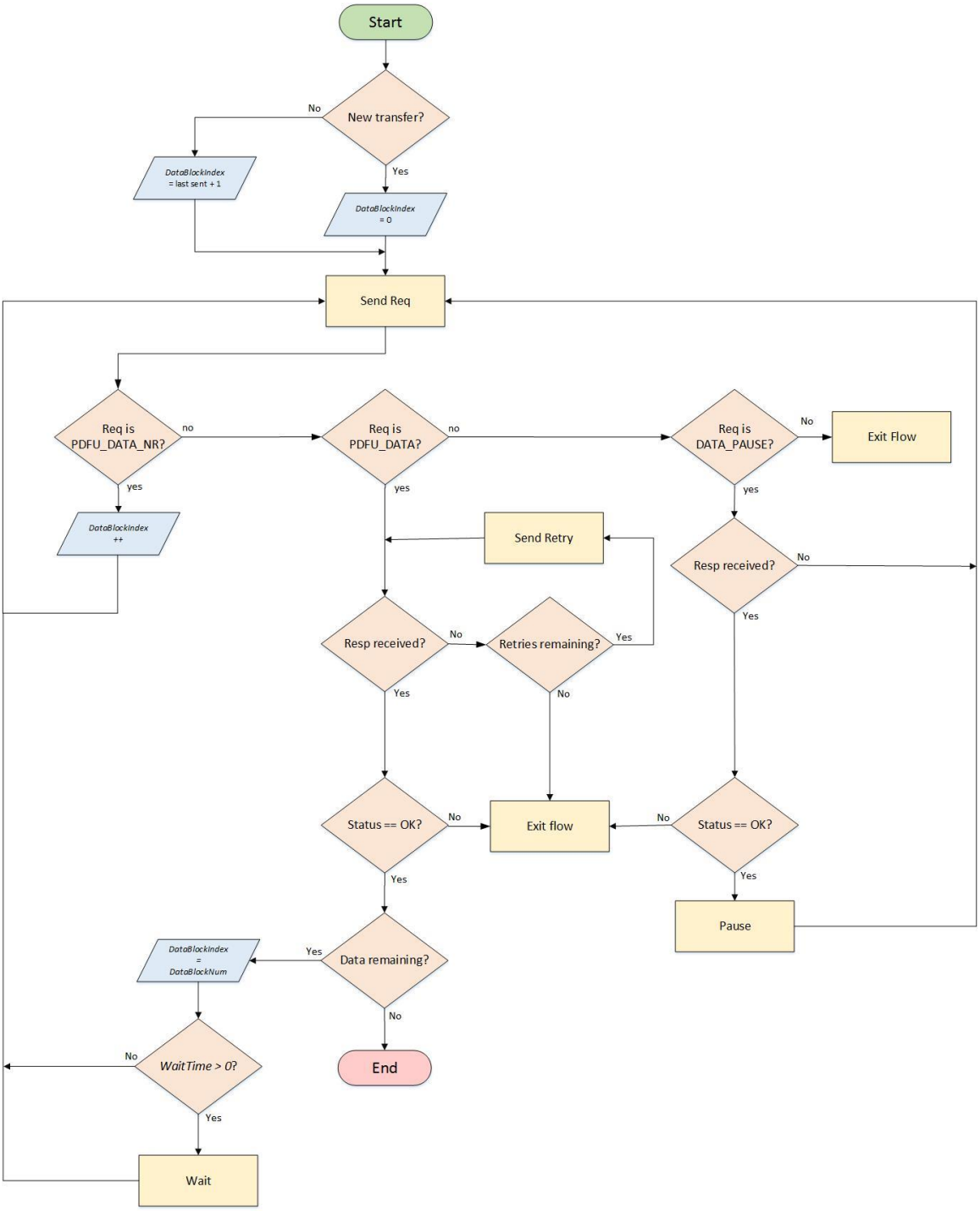
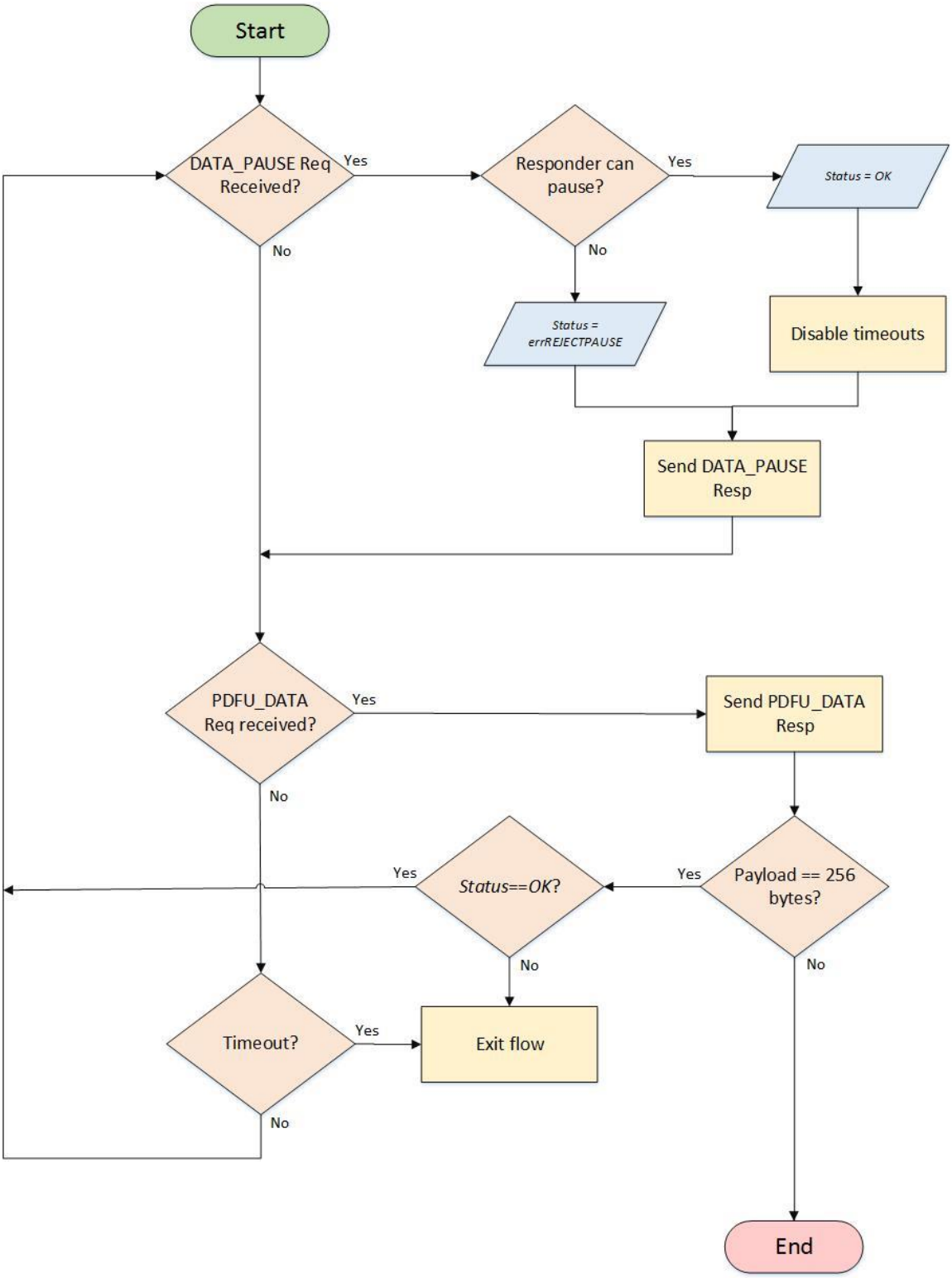


Figure A-2 PDFU Responder Transfer Phase Flow (Informative)



Appendix B PDFU Prefix Reference Code

```
/*****************************************************************\
pdfu.c
This is sample software to demonstrate a simple method of manipulating the
PDFU prefix as specified in the PDFU specification version 1.0. This code
is modified from the equivalent DFU code in order to support the PDFU
Prefix (replacing the DFU Suffix) in hex
The following authors have contributed to this sample code:
      Colin Whitby-Strevens
      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 prefix. To modify a prefix, delete
the current one and then append a new prefix.
\*****************************************************************/
#include <stdio.h>
/* #include <io.h> *** not available in all environments */
#include <sys/stat.h> /* was <sys\stat.h> */
#include <stdarg.h>
#include <string.h>
#include <stdlib.h>
#include <errno.h>
#include <stdbool.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, 0xe963a535, 0x9e6495a3, 0x09b64c2b, 0x7eb17cbd,
  0xf3b97148, 0x84be41de, 0x136c9856, 0x646ba8c0, 0xfa0f3d63, 0x8d080df5,
  0x3c03e4d1, 0x4b04d447, 0xdbbbc9d6, 0xacbcf940, 0x26d930ac, 0x51de003a,
  0xcfba9599, 0xb8bda50f, 0x2f6f7c87, 0x58684c11, 0x98d220bc, 0xefd5102a,
  0x7807c9a2, 0x0f00f934, 0x91646c97, 0xe6635c01, 0x6c0695ed, 0x1b01a57b,
```

```
  0x8bbeb8ea, 0xfcb9887c, 0x4db26158, 0x3ab551ce, 0xa4d1c46d, 0xd3d6f4fb,
  0x44042d73, 0x33031de5, 0xbe0b1010, 0xc90c2086, 0x5edef90e, 0x29d9c998,
  0xb7bd5c3b, 0xc0ba6cad, 0xead54739, 0x9dd277af, 0x0d6d6a3e, 0x7a6a5aa8,
  0xf00f9344, 0x8708a3d2, 0x196c3671, 0x6e6b06e7, 0xf9b9df6f, 0x8ebeeff9,
  0x38d8c2c4, 0x4fdff252, 0xd80d2bda, 0xaf0a1b4c, 0x316e8eef, 0x4669be79,
  0xcc0c7795, 0xbb0b4703, 0x2bb45a92, 0x5cb36a04, 0x9b64c2b0, 0xec63f226,
  0x72076785, 0x05005713, 0x92d28e9b, 0xe5d5be0d, 0x68ddb3f8, 0x1fda836e,
  0x88085ae6, 0xff0f6a70, 0x616bffd3, 0x166ccf45, 0xa7672661, 0xd06016f7,
  0x40df0b66, 0x37d83bf0, 0xbdbdf21c, 0xcabac28a, 0x54de5729, 0x23d967bf,
  0xee0e612c, 0x990951ba, 0x076dc419, 0x706af48f, 0x0edb8832, 0x79dcb8a4,
  0xe0d5e91e, 0x97d2d988, 0xe7b82d07, 0x90bf1d91, 0x1db71064, 0x6ab020f2,
  0x1adad47d, 0x6ddde4eb, 0xf4d4b551, 0x83d385c7, 0xfd62f97a, 0x8a65c9ec,
  0x14015c4f, 0x63066cd9, 0x3b6e20c8, 0x4c69105e, 0xd56041e4, 0xa2677172,
  0xd20d85fd, 0xa50ab56b, 0x35b5a8fa, 0x42b2986c, 0x32d86ce3, 0x45df5c75,
  0xdcd60dcf, 0xabd13d59, 0xc8d75180, 0xbfd06116, 0x21b4f4b5, 0x56b3c423,
  0x2802b89e, 0x5f058808, 0xc60cd9b2, 0xb10be924, 0xc1611dab, 0xb6662d3d,
  0x76dc4190, 0x01db7106, 0x71b18589, 0x06b6b51f, 0x9fbfe4a5, 0xe8b8d433,
  0x9609a88e, 0xe10e9818, 0x7f6a0dbb, 0x086d3d2d, 0x6b6b51f4, 0x1c6c6162,
  0x856530d8, 0xf262004e, 0x8208f4c1, 0xf50fc457, 0x65b0d9c6, 0x12b7e950,
  0x62dd1ddf, 0x15da2d49, 0x8cd37cf3, 0xfbd44c65, 0xa3bc0074, 0xd4bb30e2,
  0x4adfa541, 0x3dd895d7, 0x4369e96a, 0x346ed9fc, 0xad678846, 0xda60b8d0,
  0xaa0a4c5f, 0xdd0d7cc9, 0x5005713c, 0x270241aa, 0x5768b525, 0x206f85b3,
  0xb966d409, 0xce61e49f, 0xb0d09822, 0xc7d7a8b4, 0x59b33d17, 0x2eb40d81,
  0xedb88320, 0x9abfb3b6, 0x03b6e20c, 0x74b1d29a, 0x04db2615, 0x73dc1683,
  0xe3630b12, 0x94643b84, 0xe40ecf0b, 0x9309ff9d, 0x0a00ae27, 0x7d079eb1,
  0x1e01f268, 0x6906c2fe, 0xf762575d, 0x806567cb, 0xfed41b76, 0x89d32be0,
  0x10da7a5a, 0x67dd4acc, 0x17b7be43, 0x60b08ed5, 0xd6d6a3e8, 0xa1d1937e,
  0xd1bb67f1, 0xa6bc5767, 0x3fb506dd, 0x48b2364b, 0x36034af6, 0x41047a60,
  0xdf60efc3, 0xa867df55, 0xcb61b38c, 0xbc66831a, 0x256fd2a0, 0x5268e236,
  0x220216b9, 0x5505262f, 0xc5ba3bbe, 0xb2bd0b28, 0xc2d7ffa7, 0xb5d0cf31,
  0x2cd99e8b, 0x5bdeae1d, 0x756aa39c, 0x026d930a, 0x9c0906a9, 0xeb0e363f,
  0x95bf4a82, 0xe2b87a14, 0x7bb12bae, 0x0cb61b38, 0x7cdcefb7, 0x0bdbdf21,
  0x86d3d2d4, 0xf1d4e242, 0x81be16cd, 0xf6b9265b, 0x6fb077e1, 0x18b74777,
  0x66063bca, 0x11010b5c, 0x8f659eff, 0xf862ae69, 0xa00ae278, 0xd70dd2ee,
  0x4e048354, 0x3903b3c2, 0x4969474d, 0x3e6e77db, 0xaed16a4a, 0xd9d65adc,
  0xa9bcae53, 0xdebb9ec5, 0x47b2cf7f, 0x30b5ffe9, 0x53b39330, 0x24b4a3a6,
  0xbad03605, 0xcdd70693, 0xb3667a2e, 0xc4614ab8, 0x5d681b02, 0x2a6f2b94,
  0xb40bbe37, 0xc30c8ea1, 0x5a05df1b, 0x2d02ef8d};
/**************************************************************************/
unsigned char _prefix[64] = {
  0x00,                     /* dwCRC lsb */
  0x00,                     /* dwCRC --- */
  0x00,                     /* dwCRC --- */
  0x00,                     /* dwCRC msb */
  23,                       /* bLength for this version */
  'P',                      /* ucDfuSignature lsb */
  'D',                      /* ucDfuSignature --- */
  'F',                      /* ucDfuSignature --- */
  'U',                      /* ucDfuSignature msb */
  0x00,                     /* bcdPDFU lo */
  0x01,                     /* bcdPDFU hi */
  0x00,                     /* idVendor lo */
  0x00,                     /* idVendor hi */
  0x00,                     /* idProduct lo */
  0x00,                     /* idProduct hi */
  0x00,                     /* wVersionDevice1 lo */
  0x00,                     /* wVersionDevice1 hi */
  0x00,                     /* wVersionDevice2 lo */
  0x00,                     /* wVersionDevice2 hi */
  0x00,                     /* wVersionDevice3 lo */
  0x00,                     /* wVersionDevice3 hi */
  0x00,                     /* wVersionDevice4 lo */
  0x00                      /* wVersionDevice4 hi */
```

```
};
unsigned char _fileprefix[64];
/*************************************************************************\
\*************************************************************************/
void _fatal(const char *);
void _fatal(const char *_str)
{
  perror(_str);
/*  fcloseall(); */    /* not available on all systems */
  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: pdfu fname [options]\n\n"                                      \
" to check for a prefix use: pdfu fname\n\n"                             \
" to remove a prefix use: pdfu fname -del\n\n"                           \
" to add a prefix use: pdfu fname -did1 val -did2 val -did3              \
val -did4 val -pid val -vid val\n\n"                                     \
" e.g., pdfu myfile -did1 0x0102 -did2 0x0203 -did3 0x0304               \
-did4 0x0405 -pid 2345 -vid 017\n"                                       \
" sets idDevice1 0x0102 idDevice2 0x0203 idDevice3 0x0304                \
idDevice4 0x0405 idProduct 0x0929 idVendor 0x000F\n\n"

#define _getarg(ident,index);                                            \
if (!strcmp(argv[_i], (ident)))                                          \
{                                                                        \
  _write_prefix = 1;                                                     \
  if (argc-1 == _i) _fatal(_usage);                                      \
  _tmpl = strtol(argv[_i+1], &_charp, 0);                                \
  _prefix[(index)] = (unsigned char)(_tmpl & 0x000000FF);                \
  _tmpl /= 256;                                                          \
  _prefix[(index)+1] = (unsigned char)(_tmpl & 0x000000FF);              \
}
/*************************************************************************\
\*************************************************************************/
int main(int argc, const char **argv)
{
  FILE *_fp;
  FILE *_tmpfp;
  int             _remove_prefix = 0;
  int             _write_prefix = 0;
  unsigned long   _filecrc;
  unsigned long   _fullcrc;
  long            _i;
  int             _j;
  int             _plength;
  bool            _foundprefix;
  long            _tmpl;
  char            *_charp;
  unsigned char   _char;

  /* 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]);

  for (_i = 0; _i < _prefix[4]; _i++)
    _fileprefix[_i] = _prefix[_i];

  /* assume that there is a prefix and convert from hex */

fseek(_fp, 0L, SEEK_END);
  _i = ftell(_fp);
  rewind(_fp);
  _foundprefix = false;
  if (_i > 48) /* file is long enough that there might be a real prefix */
  {
    _plength = 23;
    for (_j = 0; _j < _plength; _j++)
    {
      _char = fgetc(_fp);
      if (('0' <= _char) && (_char <= '9'))
        _char = _char - '0';
      else if (('a' <= _char) && (_char <= 'f'))
        _char = _char - 'a' + 10;
      else if (('A' <= _char) && (_char <= 'F'))
        _char = _char - 'A' + 10;
      else
        break;
      if (_j <= sizeof(_fileprefix))
        _fileprefix[_j] = _char << 4;
      _char = fgetc(_fp);
      if (('0' <= _char) && (_char <= '9'))
        _char = _char - '0';
      else if (('a' <= _char) && (_char <= 'f'))
        _char = _char - 'a' + 10;
      else if (('A' <= _char) && (_char <= 'F'))
        _char = _char - 'A' + 10;
      else
        break;
      if (_j <= sizeof(_fileprefix))
        _fileprefix[_j] = _fileprefix[_j] + _char;
      if (_j == 4) /* length byte - length may change in the future */
      {
        _plength = _fileprefix[_j];

       /* printf("length field in file prefix = %d\n", _plength); */
      }
    }
    if (_j == _plength)
    {
      _foundprefix = true;
      printf("found hex prefix\n");
    } else
      printf ("no hex prefix found\n");
  }

  /* compute the CRC after the first 4 bytes */
```

```
  _filecrc = 0xffffffff;

  if (_foundprefix) {
    for (_j = 4; _j < _fileprefix[4]; _j++)
      _crc(_filecrc, _fileprefix[_j]);

    for (; _i-(2*_fileprefix[4]); _i--)
      _crc(_filecrc, (unsigned char) fgetc(_fp));
    printf("file calculated crc: 0x%08lX\n", _filecrc);

    _fullcrc = _filecrc;

    /* store the file crc in the new prefix area for comparison */
    for (_i = 0; _i < 4; _i++) {
      _prefix[_i] = (unsigned char) (_filecrc & 0x000000ff);
      _filecrc /= 256;
    }

    /* and check with the received version using CRC zero technique */
    for (_j = 0; _j < 4; _j++)
      _crc(_fullcrc, _fileprefix[_j]);
    if (_fullcrc == 0)
      printf("File CRC OK\n");
    printf ("file calculated crc including crc bytes: 0x%08lX\n",
_fullcrc);
  }

  /* compute the CRC of everything including the first 4 bytes - assumes
     that there is no CRC */
  rewind(_fp);
  _fullcrc = 0xffffffff;
  fseek(_fp, 0L, SEEK_END);
  _i = ftell(_fp);
  for (; _i; _i--)
    _crc(_fullcrc, (unsigned char) fgetc(_fp));
  printf("full crc: 0x%08lX\n", _fullcrc);

  /* if prefix exists, try to validate it */
  if (_foundprefix)
  {
    /* print out what is in there already */
    printf("   idVendor: 0x%02X%02X\n", (unsigned char) _fileprefix[12],
(unsigned char) _fileprefix[11]);
    printf("  idProduct: 0x%02X%02X\n",
           (unsigned char) _fileprefix[14], (unsigned char)
_fileprefix[13]);
    printf(" idVersion1: 0x%02X%02X\n",
           (unsigned char) _fileprefix[16], (unsigned char)
_fileprefix[15]);
    printf(" idVersion2: 0x%02X%02X\n",
           (unsigned char) _fileprefix[18], (unsigned char)
_fileprefix[17]);
    printf(" idVersion3: 0x%02X%02X\n",
           (unsigned char) _fileprefix[20], (unsigned char)
_fileprefix[19]);
    printf(" idVersion4: 0x%02X%02X\n",
           (unsigned char) _fileprefix[22], (unsigned char)
_fileprefix[21]);

  }

  /* now parse the command arguments to overwrite the default prefix w/
     new values */
```

```
  for (_i = 1; _i < argc; _i++) {
    errno = EINVAL;
    if (!strcmp(argv[_i], "-del"))
      _remove_prefix = 1;
    _getarg("-vid", 11);
    _getarg("-pid", 13);
    _getarg("-did1", 15);
    _getarg("-did2", 17);
    _getarg("-did3", 19);
    _getarg("-did4", 21);
  }

  if (_foundprefix) {

    /* compare the found file prefix to the prefix in memory */
    for (_i = 0; _i < 11; _i++) {
      if (_fileprefix[_i] != _prefix[_i]) break;
    }
    if (_i < 4) {
      printf("bad dwCRC\n");
    }
    else if (_i < 5)
      printf("bad bLength\n");
    else if (_i < 9)
      printf("bad ucDfuSignature\n");
    else if (_i < 11)
      printf("bad bcdPDFU\n");
    if (_i < 11) {

      /* can't remove a prefix if there isn't one there */
      if (_remove_prefix)
        printf("invalid or missing prefix\n");
      _remove_prefix = 0;
    } else {
      printf("valid pdfu prefix found\n");
      errno = EINVAL;
      if (_write_prefix)
        _fatal("delete prefix before making changes\n");
    }
  }

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

  /* remove an existing prefix? */
  if (_foundprefix && _remove_prefix) {
    _tmpfp = fopen("pdfu.tmp", "w+b");
    if (!_tmpfp)
      _fatal("pdfu.tmp");

    /* this is not an exercise in how to do buffered file io ;-) */
    fseek(_fp, 0L, SEEK_END);
    _i = ftell(_fp) - ((_prefix[4] * 2) + 2);  /* size of rest of file,
after the hex prefix */
    if (_i > 0) {
      fseek(_fp, (_prefix[4] * 2) + 2, SEEK_SET);   /* skip the prefix */
      for (; _i; _i--)
        fputc(fgetc(_fp), _tmpfp);
      fclose(_tmpfp);
      fclose(_fp);
      chmod(argv[1], S_IWRITE);
      remove(argv[1]);
```

```
      rename("pdfu.tmp", argv[1]);
      /* warm fuzzies */
      printf("pdfu prefix removed from %s\n", argv[1]); }
    else
      printf("%s too small to contain pdfu prefix\n", argv[1]); exit(0);
  }

  /* prepend a prefix to the file? */
  if (_write_prefix) {

    /* prepend a PDFU prefix */
    _tmpfp = fopen("pdfu.tmp", "w");  // was "w+b"
    if (!_tmpfp)
      _fatal("pdfu.tmp");

    /* calculate CRC */
    _fullcrc = 0xffffffff;

    /* iterate CRC over the prefix less the CRC field */
    for (_i = 4; _i < _prefix[4]; _i++) {
      _crc(_fullcrc, _prefix[_i]);
    }

    _crc(_fullcrc, '\r');
    _crc(_fullcrc, '\n');

    /* and calculate rest of CRC over the original file */
    fseek(_fp, 0L, SEEK_END);
    _i = ftell(_fp);
    rewind(_fp);
    for (; _i; _i--)
      _crc(_fullcrc, (unsigned char) fgetc(_fp));
    printf("full crc: 0x%08lX\n", _fullcrc);

    /* store the CRC */
    for (_i = 0; _i < 4; _i++)
    {
      _prefix[_i] = (unsigned char) (_fullcrc & 0x000000ff);
      _fullcrc /= 256;
    }

    /* write the prefix */
    for (_i = 0; _i < _prefix[4]; _i++) {
      fprintf(_tmpfp, "%02X", _prefix[_i]);
    }
    fprintf(_tmpfp, "\r\n");

    /* copy the file */
    fseek(_fp, 0L, SEEK_END);
    _i = ftell(_fp);  // size of file
    rewind(_fp);

    /* for (; _i; _i--) */
    for (_j = 0; _j < _i; _j++)
      fputc(fgetc(_fp), _tmpfp);
    fclose(_tmpfp);
    fclose(_fp);
    chmod(argv[1], S_IWRITE);
    remove(argv[1]);
    rename("pdfu.tmp", argv[1]);

    /* warm fuzzies */
    printf("pdfu prefix prepended to %s\n", argv[1]); }
```

```
  /* finished */
  fclose(_fp);
  return 0;
}
/* eof */
```