libDFU Documentation

Release 1.0.0

ANSSI

Oct 10, 2019

CONTENTS

Contents

The LibDFU project aim to implement a complete USB DFU (Direct Firmware Update) device-side automaton.

This automaton should permit to any device to support both DFU upload and download modes. This stack does not aim to implement the USB control stack or the USB driver, and request these two components to exists. The current implementation of libDFU is compatible with the Wookey STM32F439 driver API, and should be easily portable with other drivers.

CHAPTER

ONE

OVERVIEW

1.1 Principles

The DFU (Device Firmware Update) is a USB class defined by the USB consortium in order to define a vendor-independent, device-independent mechanism to update USB devices.

The DFU protocol can be seen as a communication media to download or upload firmwares from/to a device.

> **Caution:** The DFU protocol is not responsible for the firmware image security (integrity, authenticity and so on)

The DFU protocol is based on a relatively big state automaton defined in the USB DFU class description

1.2 Limitations

The DFU protocol only maintain a connected channel between the host and the device through which they can exchange data.

The channel is **not** secured, not authenticated at DFU stack level.

The DFU protocol is using the control endpoint to communicate, without requiring a dedicated endpoint to transmit or receive data. As a consequence, the USB URB (USB data blocks) size are limited by the control endpoint specific constraints.

This endpoint also manage the control part of the protocol.

CHAPTER

TWO

API

2.1 The DFU functional API

2.1.1 Initializing the stack

Initialize the DFU library is made through two main functions:

```
#include "dfu.h"

void dfu_early_init(void);

mbed_error_t dfu_init(uint8_t **buffer,
                      uint16_t max_size);
```

the early init step is called before the task ends its initialization phase using sys_init(INIT_DONE) syscall. This syscall declare all the requested ressources that can only be declared at initialization time. This include the USB device memory mapping.

The init step initialize the DFU stack context. At the end of this function call, the DFU stack is in DFU_IDLE mode, ready to receive DFU request from the host.

Caution: Even if the DFU stack internal is ready for handling DFU requests, these requests are executed by the dfu_exec_automaton() function that need to be executed

The task has to declare a buffer and a buffer size that will be used by the DFU stack to hold firmware chunks during the UPLOAD and DOWNLOAD states.

The buffer size depend on the task constraints but **must be a multiple of the control plane USB URB size** (usually 64 bytes length).

Note: Bigger the buffer is, faster the DFU stack is

2.1.2 Interacting with the storage backend

Accessing the backend is not under the direct responsability of the DFU stack. Although, the stack need to request backend write and/or read access in DOWNLOAD and UPLOAD states.

To allow flexibility in how the storage backend is handled, the task has to decrlare the following functions:

```
uint8_t dfu_backend_write(uint8_t ** volatile data,
                          const uint16_t      data_size,
                          uint16_t            blocknum);

uint8_t dfu_backend_read(uint8_t *data, uint16_t data_size);

void dfu_backend_eof(void);
```

The *dfu_backend_write()* function is called by the DFU stack when a firmware chunk has been received. This function is then responsible of the communication with the storage manager (SDIO, EMMC or any storage backend), and should return 0 if the storage has acknowledge correctly the chunk write.

The *dfu_backend_read()* function is called by the DFU stack when the host is requesting a firmware chunk from the device. In UPLOAD mode, the host is reading sequencially the firmware. The task (and/or the storage manager) is responsible for returning the correct chunk of data for each successive *dfu_backend_read()* call. This an be done, for example, using a base address and a counter.

The *dfu_backend_eof()* is called when the host has finished to send the firmware data chunks. The application and the storage manager can decide to execute any action during this event, if needed.

> **Danger:** These functions **must** be defined by the application or the link step will fail to find these three symbols

As storage backend access may be slow, the DFU stack can handle asynchronous write and read actions. This is done in the implementation of dfu_backend_write() and dfu_backend_read() where the task has to request asyncrhonous write and/or read (using DMA or IPC for example).

To inform the DFU stack that the storage access is terminated, two functions are defined in the DFU stack:

```
#include "dfu.h"

void dfu_load_finished(uint32_t bytes_read);
void dfu_store_finished(void);
```

> **Caution:** When using asynchronous read and write, the task has to update its main loop to detect the end of read and end of write and execute these functions.

2.2 About the poll timeout

The Poll timeout defines the minimum period (in milliseconds) of the DFU_GET_STATUS requests of the host. Depending on the write acces cost and the load of the device, this value may vary to avoid usless DFU_GET_STATUS requests to which the DFU stack has to respond a BUSY state.

If another task is costly in the overall device operating system, this flag can also be increased to avoid timeout.

2.3 Executing the DFU automaton

The DFU stack automaton is executed in main thread using the following function

```
#include "dfu.h"
mbed_error_t dfu_exec_automaton(void);
```

A basic usage of the automaton would be:

```
mbed_error_t ret;
while (1) {
    ret = dfu_exec_automaton();
    if (ret != MBED_ERROR_NONE) {
       /* action to decide */
    }
}
```

the automaton execution may returns:

- MBED_ERROR_INVSTATE: the command received should not happen in this state of the DFU automaton
- MBED_ERROR_TOOBIG: the input file size is too big
- MBED_ERROR_UNSUPPORTED_COMMAND: command received is not supported by the DFU stack configuration

When handling asynchronous read and write, the main loop would look like:

```
/* set by asynchronous handler*/
uint32_t data_read;
bool flag_read_finished;
bool flag_write_finished;

while (1) {
   /* inform the DFU stack of backend end of read/write */
   if (flag_read_finished) {
      dfu_load_finished(data_read);
      data_read = 0;
      flag_read_finished = false;
   }
   if (flag_write_finished) {
      dfu_store_finished();
      flag_write_finished = false;
   }
   ret = dfu_exec_automaton();
   if (ret != MBED_ERROR_NONE) {
      /* action to decide */
   }
}
```

CHAPTER

THREE

FAQ