libmassstorage Documentation

Release 0.8.5

ANSSI

May 17, 2019

CONTENTS

Contents

CHAPTER

ONE

OVERVIEW

The LibMassStorage project aim to implement a complete SCSI stack for mass storage devices. This SCSI stack is implemented as a state automaton, executing SCSI commands in the main thread, handling and enqueuing command events in ISR threads.

This stack does not aim to implement the USB control stack or the USB driver, and request these two components to exists. By now, this stack implement the BULK layer implementation of the USB stack.

Note: The BULK support is to be moved as an independent libbulk library in a near future

The current implementation of libMassStorage is compatible with the Wookey STM32F439 driver API, and should be easily portable with other drivers.

The SCSI stack has originally been defined for hard disk drives and is now used for various devices including USB thumb drives. For USB mass-storage device using SCI stack, commands are send by the initiator (the client) to the device (the server), through command blocks named *SCB*.

The server acknowledge each command and may, if the command requires it, send back or receive data on the read and write dedicated USB endpoints (usually, but not always, endpoint 1 and endpoint 2).

Caution: The SCSI protocol is synchronous and request predefined command-response sequences defined in the SCSI standard

1.1 Limitations

Developping a fully-efficient SCSI automaton is complex, as SCSI implementations of various OSes vary and sometimes require proprietary request (typically Microsoft requests a dedicated BULK level string identifier). Moreover, the standard is not made to be fully implemented, as it permits to handle various use cases, from USB mass-storage to fiber-channel and iSCSI protocols. This polymorphism make the SCSI standard complex to understand and to implement.

CHAPTER

TWO

API

2.1 The SCSI functional API

2.1.1 Initializing the stack

Initialize the massstorage library is made through two main functions

```
#include "scsi.h"

uint8_t scsi_early_init(uint8_t*buf, uint16_t buflen);

mbed_error_t scsi_init(void);
```

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 SCSI stack is in SCSI_IDLE mode, ready to receive SCSI command blocks from the host.

Caution: Even if the SCSI stack internal is ready for handling DFU requests, these requests are executed by the scsi_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 data chunks during the READ and WRITE states.

The buffer size depend on the task constraints but **must be a multiple of the endpoint USB URB size** (usually 512 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 scsi_storage_backend_write(uint32_t sector_addr, uint32_t num_sectors);
uint8_t scsi_storage_backend_read(uint32_t sector_addr, uint32_t num_sectors);
uint8_t scsi_storage_backend_capacity(uint32_t *numblocks, uint32_t *blocksize);
```

The *scsi_storage_backend_write()* function is called by the SCSI stack when a data 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 data chunk is at most of buflen size, but the associated SCSI WRITE command may request bigger write. The SCSI stack is responsible of the write split.

The *scsi_storage_backend_read()* function is called by the SCSI stack when the host is requesting data from the device. Again, the SCSI READ command may request more than the buffer capacity. The SCSI stack is also responsible of the data requests split.

The *scsi_storage_backend_capacity()* is called when the SCSI stack is requesting the storage backend capacity. This is usually the consequence of a MODE SENSE SCSI request from the host, to which the SCSI stack return various informations about the device and the SCSI stack itself.

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

Caution: All address and size are in SCSI sectors unit. This information is generally shared with the storage manager, which also manipulate sectors. Althrough, sector size may be translated by the storage manager if needed (e.g. from 512 to 4096 bytes length). OSes usually support from 512 to 4096 bytes sector size.

Backend access, in the SCSI stack, is synchronous and not made for asynchronous read or write.

2.2 Executing the SCSI automaton

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

```
#include "scsi.h"
void scsi_exec_automaton(void);
```

A basic usage of the automaton would be

```
while (1) {
    scsi_exec_automaton();
}
```

2.3 Supported SCSI commands

The SCSI standard is huge and the requested supported commands depend on the SCSI device type, the host Operating System SCSI stack version and some inter-commands dependencies.

Today, this SCSI stack support the following commands:

- FORMAT UNIT
- INQUIRY
- MODE SELECT(6)

- MODE SELECT(10)
- MODE SENSE(6)
- MODE SENSE(10)
- PREVENT ALLOW MEDIUM REMOVAL
- READ FORMAT CAPACITIES
- READ(6)
- READ(10)
- READ CAPACITY(10)
- READ CAPACITY(16)
- READ FORMAT CAPACITIES
- REPORT LUNS
- START STOP UNIT
- SYNCHONIZE CACHE(10)
- TEST UNIT READY
- VERIFY(10)
- WRITE(6)
- WRITE(10)

2.4 Debugging the stack

The SCSI stack can be debugged easily using the SCSI menu of the library in the configuration menu. There is three levels of debug:

- 0: no debug at all. Production mode
- 1: SCSI commands sequence. All SCSI command are printed on the serial interface
- **2: SCSI commands dump and behavior: complex commands (inquiry, etc.) are dumped** on the serial interface. Triggers (data sent, data available) events are printed. amount of data sent or received are also printed.

The debugging is functional only if the kernel serial console is activated.

CHAPTER

THREE

FAQ

3.1 When connected to my JTAG port, there is some regular freeze of the SCSI stack

Beware when keeping the JTAG port connected during the tests of the Wookey board. Incorrectly connected JTAG port of unstable connection may generate noise which can perturbate the high speed I/O ports such as USB High-Speed. Check if the problem still happen without the JTAG and the UART connected

3.2 There is some strange behavior of my USB tools on my host when I try to communicate with the Wookey

If you are currently in a debug state of your Wookey device and if you have regulary reset/disconnect your device from your host, try to:

1. change the USB port on which the device is connected
2. reboot your host, as the USB host stack may have not correctly handle too much unstability on the USB ports