Microprocessor Systems Lab 6

Checkoff and Grade Sheet

Partner 1 Name: ____________________

Partner 2 Name: ____________________

Grade Component	Max.	Points Awarded Partner 1	Points Awarded Partner 2	TA Init.s	Date
Performance Verification: Task 1	25 %				
Task 2	25 %				
Documentation and Appearance	50 %				
Total:					

→ Laboratory Goals

By completing this laboratory assignment, you will learn to use:

1. the Universal Serial Bus (USB) in Host Mode.

→ Reading and References

R1. Mastering STM32: Chapters 25 (FatFs)

R2. UM1905-stm32f7_HAL_and_LL_Drivers.pdf: Chapter 30 (HCD)

R3. RM0410-stm32f7_Reference_Manual.pdf: Chapter 41 (USB)

R4. UM1720-stm32_USB_Host_Library.pdf: Overview of USB Host Driver included in STM32Cube. Useful.

R5. UM1734-stm32_USB_Device_Library.pdf: Overview of USB Device Driver included in STM32Cube. Include for completeness.

R6. UM1721-stm32_FatFs_Library.pdf: Overview of FatFs Type Filesystem Driver included in STM32Cube.

R7. 32f769i_Discovery_Manual.pdf

R8. LAB-06-USB-Template.zip: Project Template for Lab 6 USB

R9. http://elm-chan.org/fsw/ff/00index_e.html: Documentation for Generic FatFs Driver

Universal Serial Bus (USB)

→ Introduction to USB

USB devices are everywhere. In fact, chances are the device you're reading this document on has some form of USB connector or two in it. Even seemingly oddball interfaces like iPhone/iPad Lightning ports, many proprietary camera connectors, Microsoft Surface dock connectors, and some car interconnections are actually just glorified USB ports, as they incorporate USB to some capacity but have different connector shapes and sometimes extra pins for other functions.

There are a couple different flavors of USB devices, namely hosts, peripherals, and hubs. As the names imply, hosts are devices like laptops, desktops, and tablets that have the USB female connectors, peripherals are endpoint devices like memory sticks, keyboards, printers, STM32F769I-DISCOs, etc., and hubs are those things you plug into a host that gives you more ports. There are currently three major versions of USB, with each version providing greater and greater maximum throughputs. For example, the original USB 1.0 tops at 1.5 Mbit/s (192 kiloB/s), though USB 1.1 revision supports devices up to 12 Mbit/s (1.5 MB/s). USB 2.0 provides up to 480 Mbit/s (60 MB/s) speeds, and USB 3.0 reaches up to 5 Gbit/s (625 MB/s). More recently, two revisions have been published for USB 3. The USB 3.1 specification contained USB 3.1 Gen 1 and USB 3.1 Gen 2 sub-specifications, where USB 3.1 Gen 1 is simply USB 3.0 renamed while USB 3.1 Gen 2 goes up to 10 Gbit/s (1.25 GB/s). In 2017, USB 3.2 was announced, which consists of Gen 1x1 (5 Gbit/s Same as USB 3.0, USB 3.1 Gen 1), Gen 1x2 (10 Gbit/s), Gen 2x1 (10 Gbit/s Same as USB 3.1 Gen 2) and Gen 2x2 (20 Gbit/s). The "x1" and "x2" denotes how

Table 1: Summary of USB Specifications.

USB Specification	Speed [Mbit/s]	Spec Name	Max Current [A]
1.0	1.5	Low Speed (LS)	500
1.1	12	Full Speed (FS)	500
2.0	480	High Speed (HS)	500
3.0 3.1 Gen 1 3.2 Gen 1x1	5,000	SuperSpeed (SS)	900
3.1 Gen 2 3.2 Gen 1x2 3.2 Gen 2x1	10,000	SuperSpeed+ (SS+)	900
3.2 Gen 2x2	20,000	SuperSpeed+ (SS+)	900

many "lanes" the bus uses[1], thereby allowing for doubling of data rates. All USB revisions require that the host implementation be backwards compatible with previous specifications as well; e.g., USB 3.0 host ports can support a USB 1.1 keyboard, albeit at the USB 1.1 data rate. Table 1 summarizes the values listed above.

USB is an asynchronous bus which uses differential signaling[2], USB 1.0, 1.1, and 2.0 use only 4 pins (VDD, D+, D-, GND). USB 3.0 bumped this up to 9 pins, adding RX1-, RX1+, TX1-, TX1+, and another GND to allow for full-duplex serial data transmission; with USB 3.2 additionally providing RX2-, RX2+, TX2-, TX2+, among some other pins. Multiple devices may be connected to the same USB host port as well, generally through the use of USB hubs, which essentially just duplicate the connections and possibly provide external power to supplement the host's limited supply.

There are many different connector types for USB[3] Generally, type-A plugs and connectors are meant for the host end of a USB connection and type-B for the peripheral end. Type-C connectors are reversible and both hosts and devices can have them. There also exists combination plugs like the Micro-AB port on the DISCO board (CN15) that signify the device can act as either a host or peripheral, though this is not very common. It is more common to see dual-role Micro-B (e.g., CN16) connectors instead. Most smartphones with Micro-B connectors are dual role.

Dual-role Micro-USB devices (e.g., the DISCO, smartphones) are referred to as USB On-the-Go (USB OTG), which is a specification defining devifces that can act as USB hosts but are predominantly meant to be USB peripherals. This of course includes smartphones, tablets, the DISCO board, etc.

→ USB Communication

USB devices are allowed to be "hot plugged," where devices may be added and removed from the host at will. This requires that the host must of course continuously monitor the bus for changes, but also be able to identify and correctly select drivers for newly connected devices. This is accomplished through *device descriptors*, which are packets of data that contain many important pieces of information; most notably of

[1]For USB 1.0-3.1, only 1 "lane" exists (e.g., one path for data transmission in a given direction). The new USB Type-C connector has double the amount of lanes available as it is designed to be attached omni-directionally, requiring symmetric plug/socket connections.

[2]Differential signaling is more reliable over longer distances, primarily because of resistance to EM interference.

[3]USB connectors are summarized well here: https://en.wikipedia.org/wiki/USB_(Physical)#Connectors

these are the **bcdUSB** item which notes the USB specification the device conforms to, and **bDeviceClass** and **bDeviceSubClass** which tell the host what type of device it is - indicating the proper driver to use. Additionally, the fields **idVendor**, or **VID**, and **idProduct**, or **PID**, are used to indicate who makes the device and what product it is.

Once the proper driver is selected (assuming said driver exists), then the host may begin communication with the device in order to receive data from it. The device does not ever initial a communication transaction - it always responds to the host. Transfer of information is done though device *endpoints*, which are data buffers designed to be interacted with via the device driver. All devices have an **Endpoint 0**, which serves as a control interface to the device as well as possible data transfer to the host. The format of data to be conveyed from a device to the host is given by *report descriptors*; which essentially indicate the length of a report packet and the meaning of each byte in the report.

→ USB on the STM32

The STM32F769NI may serve as both a peripheral device or host (i.e., is capable of USB OTG). This microcontroller has independent support for USB 2.0 HS OTG[4] and USB 2.0 FS OTG[5]. Both of these support on-chip **PHY**, with the USB 2.0 HS OTG peripheral also supporting **ULPI**.[6]

Note that the STM32 DISCO board has two different kinds of USB ports on it: A Micro-AB port (CN15) and a Micro-B port (CN16). CN16 is currently used for the virtual COM port (it's a peripheral-only port connected through an onboard debugger chip); therefore, CN15 the only USB OTG connected to the STM32F769NI through a **ULPI** connection[7]. That means that we can't really use CN16 for much more than what we are currently using it for (debugging, uploading programs, and UART), but we have full access to CN15.

To configure the STM32F769NI to enable the USB OTG HS connection in **ULPI** mode, many pins are necessary, as shown in Table 2. Although USB is a serial protocol, the **ULPI** implementation used here first conveys the information (in bytes) to be transmitted over the USB link to the external **PHY** chip through a parallel interface via the eight `USB_OTG_HS_ULPI_D#` signals. Each of the pins listed in Table 2 should be configured with the OTG HS alternate function (`GPIO_AF`*x*`_OTG_HS`), in addition to being in alternate function mode (push-pull) with no pull up or down and in high speed mode.

[4]Reminder: USB is backwards compatible, therefore it also supports USB 1.0 and USB 1.1

[5]The datasheet calls this module a USB 2.0 full-speed, though "Full Speed" is USB 1.1. Essentially what this is is a USB 2.0 compliant device with speeds limited to the USB 1.1 specification

[6]**PHY** essentially stands for *physical layer*, where the actual USB connection is provided by the chip. **ULPI** is an external USB controller which communicates with the microcontroller via a parallel interface

[7]CN15 is not directly connected to the STM32F769NI either! This setup uses a **ULPI** connection between the microcontroller and the **PHY** device USB3320 (U20). See the schematic on page 46 of R7.

Table 2: USB HS **ULPI** Pins and Signals

Port Pin	Signal
PA3	USB_OTG_HS_ULPI_D0
PA5	USB_OTG_HS_ULPI_CK
PB0	USB_OTG_HS_ULPI_D1
PB1	USB_OTG_HS_ULPI_D2
PB5	USB_OTG_HS_ULPI_D7
PB10	USB_OTG_HS_ULPI_D3
PB11	USB_OTG_HS_ULPI_D4
PB12	USB_OTG_HS_ULPI_D5
PB13	USB_OTG_HS_ULPI_D6
PC0	USB_OTG_HS_ULPI_STP
PH4	USB_OTG_HS_ULPI_NXT
PI11	USB_OTG_HS_ULPI_DIR

→ STM32 USB Host Library

The STM32Cube software provided by STMicroelectronics contains an abstraction layer (R4) for using the USB instead of interfacing with the HAL or LL drivers, or even more painfully through registers. Additionally, several different peripheral class type drivers are also provided (e.g., Audio, MSC, HID) for use with generic devices. This library significantly simplifies the development of the STM32 device as a USB Host[8]. Unfortunately, the documentation for this library is terse, at best. When used, the library interfaces with the build application as described in Figure 1.

The general structure of a program using the USB Host Library (**USBH**) is as follows:

```
System Initializations
Application Initializations
USBH Driver Initialization
USB Driver Class Registrations: Add device types to handle.
Start USBH Driver
while(1)
        Update USB State: USB_Process()
        Read/Send USB Data
        Other Application Duties
```

A special component of this series of steps is the `USB_Process()` function. This function must be called frequency in order to poll the USB interface for changes (e.g., connect, disconnect, data available, etc.). Essentially, this would be a background process on a multi-process computer. Since multiple processes are not easily handled in this environment, time needs to be explicitly given to this function; hence the repeated calls. This may alternatively be driven by a timer interrupt if consistent timing is required. The `USB_Process()` is capable of triggering the `OTG_HS_IRQ` during certain events which when handled with `HAL_HCD_IRQHandler()`[9] will trigger callbacks either from the USB Host Library core driver or from the

[8] A USB Device Library, R5, is also given but not used in this lab.

[9] `HCD` stands for USB "Host Controller Device."

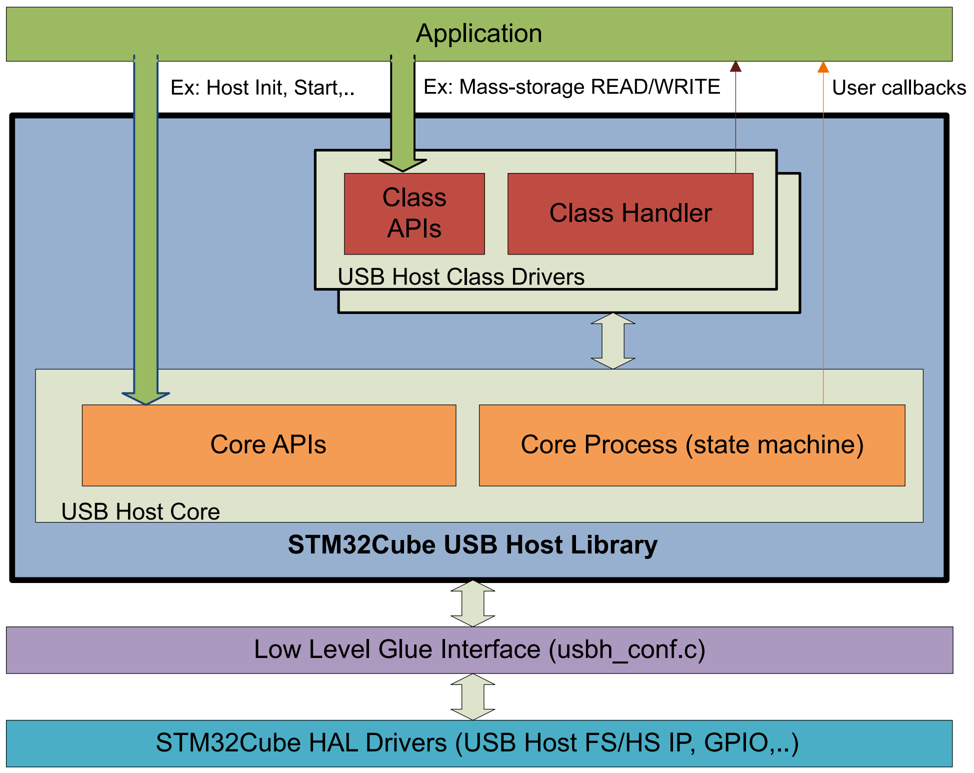
Application
Ex: Host Init, Start,..
Ex: Mass-storage READ/WRITE
User callbacks
Class APIs
Class Handler
USB Host Class Drivers
Core APIs
Core Process (state machine)
USB Host Core
STM32Cube USB Host Library
Low Level Glue Interface (usbh_conf.c)
STM32Cube HAL Drivers (USB Host FS/HS IP, GPIO,..)

Figure 1: Abstraction Layers of USB Application using USB Host Library. Taken from Figure 2 of R4.

specific USB Class[10].

Another interesting implementation note it the initialization of the **USBH** driver requires the provision of a user defined function. This function is called within the function `USB_Process()` on USB device connect, enumeration, and disconnect events, among others. The prototype for the required[11] function is as follows:

```
void USBH_UserProcess(USBH_HandleTypeDef *, uint8_t);
```

where the first argument is the handle for the USB module active and the second argument is a flag denoting the event that occurred. These events are defined in `usbh_core.h` (line 64) as:

```
#define HOST_USER_SELECT_CONFIGURATION          1
#define HOST_USER_CLASS_ACTIVE                  2
#define HOST_USER_CLASS_SELECTED                3
#define HOST_USER_CONNECTION                    4
#define HOST_USER_DISCONNECTION                 5
#define HOST_USER_UNRECOVERED_ERROR             6
```

for the purposes of this lab, the most important one here is `HOST_USER_CLASS_ACTIVE`, which denotes that a device has been attached and enumerated and is ready to use. This may be indicated to the application

[10]The possible callback functions are all listed in R4 in various locations.

[11]This function is not technically required, but it is very useful to the program to have it. Passing a value of `NULL` (0) is also valid.

program though a global variable flag. An example version of this function is given in R4, page 38 (the function `osMessagePut()` doesn't need to be used.).

The `USBH_HandleTypeDef` variable used in `USBH_UserProcess()` is the handle used to denote the USBH driver module active. This needs to be defined by the user but NOT populated as this is done automatically be the USBH initialization function, which also calls the function `USBH_LL_Init()`, which in turn calls the HAL initialization function `HAL_HCD_Init()`, which in turn calls the HAL MSP Initialization function `HAL_HCD_MspInit()`! Through this chain of calls, the whole USB system may be completely configured for use. These are generally provided in a userspace `usbh_conf.c` source file[12] To be able to complete all of these configurations, the `usbh_conf.c/h` also define the HAL HCD handle (`HCD_HandleTypeDef`) which is associated within the `USBH_HandleTypeDef` handle.

◇ Task 1: USB Mouse

Using a Micro-B to USB OTG adapter plugged into CN15, detect a USB mouse and display its connection status on the terminal. This program should be written so that it uses the `USB_OTG_HS` module even though mice and keyboards are generally USB 1.x devices. It should detect when a device has been plugged or unplugged and behave accordingly (i.e., don't poll the mouse when it's not there). Further, the program should at a minimum report the movement of the mouse and button presses. Alternatively, the movement of the mouse can be used to control the terminal cursor, with the buttons providing other functions (draw and clear?).

NOTES:

1. Mice and keyboards are known as *Human Interface Devices* (HID), which is a specific class of usb devices.
2. Take a look at the callback functions for the HID class driver, one or more may be very useful (but not necessarily).
3. Debugging of the USBH driver is built in (emits `printf` messages) and the verbosity can be adjusted within `usbh_core.h` via the defined value for `USBH_DEBUG_LEVEL`.
4. The `x` and `y` values reported by the mouse are changes since last report, not absolute values.
5. Do not continuously poll the mouse once connected. Wait until there is data to be read (Hint: Check the device driver documentation).
6. Similarly, do not try and poll the mouse without it being connected.

[12] A complete version of this file, as well as `usbh_conf.h` is provided in the lab template project R8.

FatFs

→ FatFs Overview

FAT and exFAT are two filesystem types that are used to organize files and directories on physical media. FatFs is a generic library to use FAT and exFAT filesystems on embedded systems. FatFs is “generic” in that it does not handle any of the Input/Output (IO) layer. Instead, it requires that the user provide several functions which FatFs then uses to perform the disk IO. These functions are `disk_status`, `disk_initialize`, `disk_read`, `disk_write`, and `disk_ioctl`. Fortunately, these functions are again provided by STM to support interfacing with these filesystems. The STM FatFs manual, **??**, should be consulted on how to setup the FatFs driver for this microcontroller. The generic FatFs driver documentation, R9, documents the actual filesystem operations.

→ Initializing FatFs

To setup FatFs, the user must provide the functions for FatFs to perform diskio. These are in the struct `Diskio_drvTypeDef`. An instance of this struct, with all the variables initialized, is passed to the function `uint8_t FATFS_LinkDriver(const Diskio_drvTypeDef *drv, char *path)`. The “path” argument should be “0:/” as the first drive connected. The FatFs drivers (`Diskip_drvTypeDef`) are provided by STM and are the applicable one for USB is included in the `FatFs/inc/` directory.

```
typedef struct
{
  DSTATUS (*disk_initialize) (BYTE); /*!< Initialize Disk Drive */
  DSTATUS (*disk_status) (BYTE); /*!< Get Disk Status */
  DRESULT (*disk_read) (BYTE, BYTE*, DWORD, UINT); /*!< Read Sector(s) */
#if _USE_WRITE == 1
                             /*!< Write Sector(s) when _USE_WRITE = 0 */
  DRESULT (*disk_write) (BYTE, const BYTE*, DWORD, UINT);
#endif /* _USE_WRITE == 1 */
#if _USE_IOCTL == 1
                             /*!< I/O control operation when _USE_IOCTL = 1 */
  DRESULT (*disk_ioctl) (BYTE, BYTE, void*);
#endif /* _USE_IOCTL == 1 */

}Diskio_drvTypeDef;
```

◇ Task 2: USB Mass Storage

Again, using a Micro-B to USB OTG adapter plugged into CN15, read a FAT/FAT32 USB 2.0 (HS) flash drive (most flash drives would work) and display its contents on the serial terminal. This program should add on to Task 1 such that the program will respond to the mouse or the flash drive without user intervention. Note that a flash drive in USB terms is a *Mass Storage Class* (MSC). A library for FatFs interaction is provided in the project template. In order to use the FatFs library with the USBH Drivers, some steps must be taken:

- Register the USBH MSC Class with the USBH driver,
- Link the USBH driver handle with the FatFs driver through the `FATFS_LinkDriver()` function,
- Detect if connected device is USBH MSC Class or USBH HID Class,
- Access the drive through the FatFs driver (starting with the `f_mount()` function).

This task should simply read the contents of the root directory of a flash drive and print out each item. The program should also indicate if the item is a directory or a file.

NOTES:

1. Do not repeatedly mount/access the drive; only do it once upon attachment.
2. The generic driver documentation (R9) will be very useful. Warning: the driver supplied within R8 (from STM32CubeF7) is a few versions behind R9.