Microprocessor Systems Lab 3

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	5 %				
Task 2	10 %				
Task 3	5 %				
Task 4	10 %				
Task 5 [Depth]	10 %				
TA Questioning	10 %				
Documentation and Appearance	50 %				
Total:					

Grader's signature: ____________________

Date: ____________________

→ Laboratory Goals

By completing this laboratory assignment, you will learn to:

1. Communicate with devices using UART,
2. Communicate with devices using SPI,
3. Implement communication using either Polling (Blocking) or Interrupt (Non-Blocking) configurations.

→ Reading and References

R1. Mastering STM32: Chapters 8 (UART), 15 (SPI)

R2. UM1905-stm32f7_HAL_and_LL_Drivers.pdf: Chapters 62 (SPI), 66 (UART)

R3. RM0410-stm32f7_Reference_Manual.pdf: Chapters 34 (USART), 35 (SPI/I2S)

R4. Lab03_UART_SPI_Template.zip: Project Template for Lab 3

R5. STaTS_SPI_Slave_Datasheet.pdf: Example SPI Slave datasheet.

R6. LAB-03_STaTS_Firmware.bin: Firmware for example SPI Slave.

R7. LAB-03_Master_Firmware.bin: Firmware for test SPI Master.

R8. Temperature_Sensor.h: Code for acquiring a temperature reading.

Serial Communication

In order to transport data into or out off a microcontroller from other digital devices (e.g., sensors, computers, other microcontrollers), various communication standards may be used, such as I^2C[1], UART, SPI, CAN, etc. These listed standards are all examples of *Serial Communication* techniques, where data is transferred bit-by-bit. Other standards exist but are not common in microcontroller applications, such as PATA and SCSI along with other various memory access methods, are known as *Parallel Communication*, where byte(s) are transferred over many communication lines at once[2]. For this lab, we will focus on the UART (Universal Asynchronous Receiver Transmitter) and SPI (Serial Peripheral Interface) standards.

Within the serial communication domain, there are two distinct communication methods: synchronous and asynchronous. These are differentiated by whether the data clock is shared between the devices in communication or not. In an asynchronous communication system (e.g., UART), a clock is not shared between the devices; requiring that all devices on the communication bus have knowledge of the expected data baud rate[3] For example: all labs in this course use terminal communication between the STM32F769NI and the computer which is achieved through a UART interface. When configuring the terminal on the computer, the baud rate is explicitly specified to the terminal program as 115200 bps.

[1]The I3C standard has been released towards replacing I^2C, though adaption may take a while

[2]The STM32F769NI has a Flexible Memory Controller (FMC) which contains a parallel interface for memory addressing up to 32-bits wide.

[3]Baud rate for our purposes is the communication *bit rate* or *data rate*, measured in bits per second. In higher-order modulated communication systems, baud rate is also known as the *symbol rate*.

Alternatively, synchronous communications are those where the data clock is shared between the devices on the bus (e.g., I^{2}C, SPI). This allows for the devices receiving the clock to not require an onboard timebase generator but instead requires all devices to have an extra I/O line to receive/emit the clock signal, as well as requiring an additional line connected each device. Sharing the clock between the devices also increases the maximum throughput of the bus, as there is less required overhead for data packet control signals. Synchronous communication techniques commonly have a Master/Slave architecture[4], where one device on the bus is configured as the master and directly controls the communication bus[5], which includes generation and sharing of the data clock. The slaves only communicate when addressed (I^{2}C) or selected (SPI) by the master.

Universal Asynchronous Receiver Transmitter (UART)

The STM32F769NI has several on-board universal synchronous/asynchronous receiver/transmitter (USART), which may be used to generate UART, among other standards. The DISCO board has one built-in virtual UART communication channel over USB which USART1 is configured to use. Each UART interface requires only two signal lines: RX (receive data) and TX (transmit data). These lines are named the same for any device connected, therefore, for device 1 to transmit data to device 2, the TX line of device 1 must be connected to the RX line of device 2 and vice versa. The USARTs can of course be configured through the registers listed in R3, though for this lab, the hardware abstraction layer (HAL) drivers will be used. For each USART configured to operate as a UART device, a `UART_HandleTypeDef` type module handle is required, which has the following defined structure:

```
typedef struct{
        USART_TypeDef *Instance; /* UART registers base address */
        UART_InitTypeDef Init; /* UART communication parameters */
        UART_AdvFeatureInitTypeDef AdvancedInit;
                                     /* UART advanced features configuration */
        ...
} UART_HandleTypeDef;
```

The `UART_InitTypeDef Init` field is another struct containing the following parameters, which set up how the UART operates to exchange data:

```
typedef struct {
        uint32_t BaudRate;
        uint32_t WordLength;
        uint32_t StopBits;
        uint32_t Parity;
        uint32_t Mode;
        uint32_t HwFlowCtl;
        uint32_t OverSampling;
} UART_InitTypeDef;
```

[4]The traditional terminology of "master/slave" is applicable though some view it as offensive. Recently, alternative naming conventions have been arising such as "primary/secondary," "leader/follower," or Python's recent change to "parent/worker."

[5]There are multi-master techniques, we will only focus on single-master SPI.

→ UART Port Setup

In the basic mode, the UART is relatively easy to configure when using the HAL. Three general steps are required:

1. Enable GPIO port for transmitting and receiving
2. Populating the `UART_HandleTypeDef` fields `Instance` and `Init`.
3. Call `HAL_UART_Init()`

Note that item 3 above does not and cannot inherently accomplish item 1 as there are many different configurations as to which GPIO pins to place the UART signals. While this alternatively may have been accomplished through another field in the `UART_HandleTypeDef` (e.g., `GPIOInit`), the authors of the HAL instead have the `HAL_UART_Init()` function call back to the user space function `HAL_UART_MspInit()`, where all GPIO configuration is expected to be done. The callback function `HAL_UART_MspInit()` is provided in the template project R4 in file `uart.c`.

Transmitting and receiving data is also a simple procedure requiring only `HAL_UART_Transmit()` and `HAL_UART_Receive()`. The functionality of `printf()`, `putchar()`, and `getchar()` are all implemented using those two functions (again, see R4 `uart.c`).

◇ Task 1: Two-Terminal UART (Polling)

Write a program that will monitor two serial ports continuously. The STM32F769NI has 8 total UART capable modules, though only 5 are accessible on the DISCO board, one of these being USART1 over USB. The other port used should be USART6. In order to provide a USB connection from USART6, cables are required: A DB9 to USB converter and a DB9 to male header pins. The pinout of the DB9 is listed in Table 1. This table is for the RS-232 standard; only the bolded rows are required for this application. Additionally, the default operation of the UART module produces RX and TX signals that are digitally inverted with respect to the RS-232 standard. Therefore, the RX and TX signals must be inverted prior to connection to the DB9-header pin connector. This inversion may be done in hardware or software.

USART1 should be left as configured in the template project and USART6 should be configured to use 9600 baud and N-8-1 (no parity bits, 8 data bits, 1 stop bit). For reference, USART1 is configured as 115200 baud N-8-1.

Whenever the program detects a character coming in from either of the onboard serial ports, it should echo that character back to both serial ports. This requires two terminal programs to be running; though they do not necessarily have to be on the same computer. When the <ESC> is pressed on either terminal, a brief exit message on both screens should be shown and the program halted. Since continuous polling of both ports is required, `getchar()` may not be used as it will wait indefinitely for a key press.

NOTES:

1. Do not forget to invert the TX and RX signals. If everything else is configured properly, the corresponding terminal will produce predictable but garbage output.
2. Ensure the terminals are configured with the correct baud rate: 115200 and 9600.
3. Failure to connect the Grounds between the DISCO and DB9 will result in either no output or garbage output.

4. Remember that TX1 should be connected to RX2, TX2-RX1.
5. Much of what you need already exists in `uart.c` and `init.c`.
6. The timeout values for `HAL_UART_Transmit()` and `HAL_UART_Receive()` may be very small.

Polling Versus Interrupt Operation

In Task 1, a program was made that continuously checked whether any characters were recieved on USART1 or USART6. This type of implementation is known as *polling*, or continuously (or periodically) checking to see if a specific event has happened. This method has the drawback that to successfully detect an event in complex code at the proper time, checks of the event need to be placed throughout the code. This both makes the code mode difficult to write and maintain but also introduces computational overhead. Alternatively, if only one check for the event is placed within the code and will not continue until the event has happened, then the program is essentially “blocked” from doing anything else.

This functionality can instead be handled by interrupts, or in a “non-blocking” mode. This frees up the additional overhead of polling while also ensuring that the event is dealt with immediately, in cases where there is a time sensitivity (e.g., Real Time systems). For the case of UART character reception, it is desired to trigger interrupts when a character is received by either UART port. To implement the IRQHandler for USART1, the following code should be used:

```
// Handle USB/UART Interrupts with HAL
void USART1_IRQHandler() {
	HAL_UART_IRQHandler(&USB_UART);
}
```

where `USB_UART` is the handle for USART1, defined in the template project. A similar function will need to be written for USART6. Additionally, the callback function `HAL_UART_RxCpltCallback()` will need to be written to handle character reception. Note that the handle of the USART module that triggered the interrupt will be passed to this callback function and would be used to determine the character’s origin. In order receive in this fashion, `HAL_UART_Receive_IT()` will need to be used instead of `HAL_UART_Receive()`.

Table 1: Pinout of RS-232 DB9 to Male Header Pins

Pin #	Signal Name	Cable Color
1	Data Carrier Detect (DCD)	Brown
2	**Receive Data (RX)**	**Red**
3	**Transmit Data (TX)**	**Orange**
4	Data Terminal Read (DTR)	Yellow
5	**Ground (GND)**	**Green**
6	Data Set Ready (DSR)	Blue
7	Request to Send (RTS)	Purple
8	Clear to Send (CTS)	Gray
9	Ring Indicator (RI)	White

⋄ Task 2: Two-Terminal UART (Interrupt)

Reimplement task 1 except using a non-blocking, interrupt structure instead of the polling method. The while loop in the main function **cannot** have any UART commands contained within it.

Once this program is successfully implemented and verified, you will need to find another group with a working version of Task 2 and connect the two USART6 serial ports together. This should allow input on the USB UART of team 1 to be displayed on the USB UART of team 2 and vice versa. Alternatively, this may be accomplished within the team if the team has two development boards.

NOTES:

1. The debugging mode of the IDE may be very useful for this task in order to determine if a character input is being handled properly.
2. Don't forget to use the NVIC to enable the interrupts.
3. `HAL_UART_RxCpltCallback()` will only be called once per one call of `HAL_UART_Receive_IT()`. The receive another character, this command would have to be reissued somehow.

Serial Peripheral Interface (SPI)

As introduced already, one synchronous protocol typically available on microcontrollers is the SPI. This protocol is essentially an alternative to the I^2C. There are significant differences between the two, mainly arising from slave selection and data lines. Active slave selection in I^2C is done via a software address, where the address of a slave is passed first in each data packet to specify which slave is supposed to read from or write to the master. The SPI uses "chip select" (CS) or "slave select" (SS) lines, one for each slave. By using the chip select, the overhead required in the I^2C Bus of sending information to select the slave is avoided, allowing for much higher throughput to achieve; however, this comes at the expense of requiring a CS line for each slave on the bus. Of course, if there are many slaves on the bus, the hardware required to support SPI could become exceedingly burdensome, requiring the use of I^2C instead. A generic hardware layout for each of these standards is given in Figure 1.

In addition to multiple CS lines for the SPI, the SPI also uses two data lines: *Master In, Slave Out* (MISO) and *Master Out, Slave In* (MOSI) as opposed to I^2C using only SDA. The use of MISO and MOSI allows for bidirectional communication to occur, both being clocked by SCLK, again, allowing for higher possible throughput. Data transmission and reception are always done concurrently, that is, for every byte sent, a byte is received. However, distinctions are made for SPI transmitting and SPI receiving only modes, where when only transmitting, data is sent out on MOSI and the data received on MISO is ignored. Similarly, to receive data, dummy data is sent out on MOSI (e.g., 0x00 or 0xFF) while the data received on MISO is stored.

Unlike the I^2C, SPI is less standardized, which results in many different methods to communicate with a slave. The communication requirements of the slave will need to be tailored to individually. Two points of interest are the required behavior of the clock, or the clock polarity, and when to change/latch (read) data, or the clock phase. The clock polarity can either be *Idle High* or *Idle Low*, indicating the state of SCLK when not communicating. The clock phase also has two possibilities: either the first clock edge or second clock edge indicates data capture (rising or falling), with the other edge being the data change

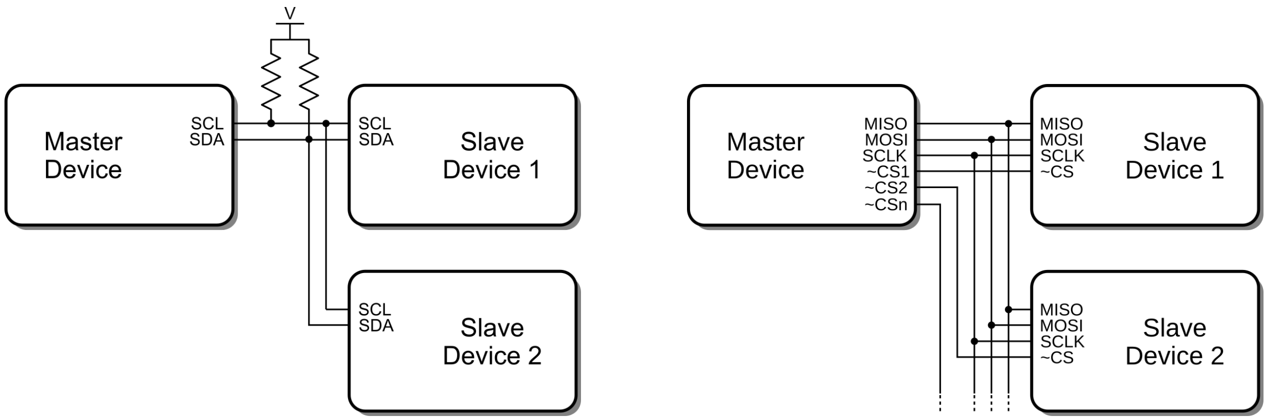

Figure 1: General architectures of (left) I^2C and (right) SPI Buses.

signal. Other common differences between the devices is the absence of either the MISO or MOSI lines, or bidirectional communication on only one of the lines, similar to I^2C.

One aspect that is consistent however, is that selection of the slave is done via pulling the CS lines low. This is indicated by the notation in Figure 1, where "~CS" is shown, the ~ indicating that the signal is *Active Low* as opposed to *Active High*. This implies that when ~CS for the slave is high, the slave will ignore all activity of the bus. This is also commonly denoted with a slash or overline instead: /CS or $\overline{\text{CS}}$. For the SPI modules on the STM32F769NI, only one ~CS line exists, labeled NSS (*Not* Slave Select). Therefore, the module itself is only capable of communicating with one slave by default. To add additional slaves, GPIO pins must be configured independently to act as the required ~CS lines.

→ SPI Port Setup

The following steps are required to configure and use the SPI port:

1. Configure GPIO pins.
2. Configure SPI clock rate, wire mode (full-duplex or single bidirectional line), clock polarity and phase, and master or slave mode. For this lab's purposes, CRC should be disabled.
3. Call `HAL_SPI_Init()`

As was the case for the USART modules, the completion of item 1 is triggered by item 3 through the callback function `HAL_SPI_MspInit()`. In order to send and receive data, the functions `HAL_SPI_Transmit()`, `HAL_SPI_Receive()`, or `HAL_SPI_TransmitReceive()` would be used, or their `_IT()` or `_DMA()` counterparts if operating in interrupt or DMA mode, respectively.

If the slave being used is controlled by the module's NSS line, then no extra commands need to be done to select the slave. If, however, a GPIO output pin is used to select the slave, the pin needs to manually be asserted low prior to calling a transmit/receive function while also disabling the NSS pin. Once the transfer is complete, the GPIO pin should again be raised high.

◇ Task 3: SPI Loopback Interface

Write a program that sets up SPI2 in 8-bit mode, monitors the terminal, echos any character received on the terminal to the SPI bus, and writes any received characters from the SPI bus to the terminal. Wire the SPI bus such that MISO and MOSI are connected together (i.e., the loopback condition). The SPI port should be configured to operate at roughly 1 MHz. The terminal should be split top and bottom such that characters received from the keyboard are written on the top half and characters received from the SPI are written on the bottom half. It is not necessary to scroll both the top half and the bottom half, though the exact implementation is up to you. Implementation of this interface either in a Polling or Interrupt Mode is acceptable.

NOTES:

1. Test your interface by removing the loopback condition and tying MISO to low or high. This should result in returned values of 0x00 or 0xFF, respectively.
2. For this task, clock polarity and phase are not important.

◇ Task 4: Connect to STM32 Slave

Using the datasheet provided in R5, design and implement an SPI master that can interact with the "STaTS" device and perform the following:

1. Reliably send and receive terminal characters to and from the slave device. The received characters should be printed on the bottom half of the master's terminal, respectively.
2. Read the slave's firmware version upon startup.
3. Trigger a temperature measurement and retrieve the result when it is ready. The temperature should be printed on the right side of the terminal to avoid the transmitted and received terminal characters.
4. Change the device ID of the slave.
5. Clear or reset the slave terminal.
6. Pressing ESC in the master's terminal should present a menu where the user can trigger items 2-5. When this menu is active, it is acceptable to ignore characters inputted from the slave's terminal.

NOTES:

- The SPI slave for this task is a second DISCO board. If the group only has one, there is a limited number available from the instructor to borrow.
- Triggering of the temperature measurement may be done by the terminal menu or through a timing interval. If done with a timing interval, the menu option for this may be omitted.
- The SPI slave may contain some bugs. Alert the instructor if the slave is not behaving as described.
- Verify the SPI bus operation using an oscilloscope (or Analog Discovery board, etc.) and include in your report. Using an oscilliscope to debug as well is *highly encouraged*.

◇ Task 5: [Depth] Configure as STM32 Slave

Configure the DISCO board as an SPI slave. The functionality of this slave is a reduced version of the STaTS device from Task 4. The functionality required for this slave is listed below:

- Send and receive terminal characters. Buffering of characters is not necessary (it's okay if keys are pressed quickly if some are lost.).
- All temperature measurement functionality except changing of number of averages. Average the temperature over 32 readings for each triggered measurement.
- Transferring of version numbers and device ID.

Functionality **not** required:

- Terminal character buffering and clearing.
- Slave terminal control (clearing and text attributes).
- Changing of Device ID.
- Changing of the number of temperature averages.
- "DEBUG" line implementation.

Master firmware for testing the developed slave is given in R7; however, the Task 4 program will work as well, assuming it is written correctly.

NOTES:

- The SPI master for this task is a second DISCO board. If the group only has one, there is a limited number available from the instructor to borrow.
- Code for acquiring a single temperature reading is given in R8. The blocking function `getTemperature()` will return a 12-bit measurement of the temperature.
- Although not obvious, an EXTI interrupt is still possible on pins with alternate pins functions enabled. It may be useful to configure the chip select to have an interrupt, using it as a trigger to reset the communication protocol to a known state.
- The "DEBUG" line is not required, but it may be extremely useful in debugging. Using breakpoints on the slave may cause the SPI interface to behave erratically.
- Beware of blindly calling `HAL_SPI_TransmitReceive[_IT]()`. It is very difficult to replace the data stored in the transmit register. `HAL_SPI_Abort[_IT]()` does not accomplish this as it waits for the transmit buffer to be cleared prior to abort.