

Microprocessor Systems Lab 3 Report

Weiheng Zhuang Ziyu Zhu

Table of Contents

Task 1: Two-Terminal UART	3
Introduction	3
Low Level Description	4
Results and Analysis	6
Task 2: Two-Terminal UART (Interrupt)	8
Introduction	8
Low Level Description	9
Results and Analysis	10
Task 3: SPI Loopback Interface	12
Introduction and High-Level Description	12
Low Level Description	13
Results and Analysis	14
Task 4: Connect to STM32-Based SPI Device	15
Introduction and High-Level Description	15
Low Level Description	16
Receive terminal characters	16
Read the peripheral's firmware	17
Temperature measurement	17
Clear peripheral terminal	18
Change and read the Device ID	19
Results and Analysis	20
Conclusions	24
UART	24
SPI	24
Appendix	25

Task 1: Two-Terminal UART

Introduction

In order to transport data into or out of a microcontroller from other digital devices, various communication standards may be used, such as I2C, UART, SPI, CAN, etc. Within the serial communication domain, there are two distinct communication methods: synchronous and asynchronous. In an asynchronous AC system, the clocks of the devices at both ends of the AC are not shared with each other. A typical example of such sharing is the UART. A very typical example is that the STM32F769NI communicates with the computer terminal, and this interaction uses the UART.

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.

For the first task we are going to do is about using the UART to communicate. There are three options available to us, we chose the first option, using USART6 to transmit signals between the panelists' DISCO boards. Whenever any one of us keyboard input is transmitted to the screen, the other will have the same output on the screen. But for such a task, we cannot use `getchar()` because it is a blocking equation. But we have a better alternative. As mentioned above, the DISCO board uses UART to communicate with the computer, so using UART is also a better solution. When one of the two computers presses ECS, both terminals will stop typing, which can be done by entering an infinite loop after transmitting the information. Then on the receiving end, if it sees a similar result, it will stop outputting.

Low Level Description

Since this is just an echo task, the underlying logic of this task is pretty straightforward. Basically, it is to turn on the GPIO first, initialize the UART, and then call the UART equation. Generally speaking, it is not difficult.

In the given initial file `uart.c`, the interface selection and initialization of GPIO has been given. First, A9 and A10 are used as the TX and RX interfaces of USART1. USART1 is used as the USB_UART handler to communicate with the computer terminal. Then use C6 and C7 as the TX and RX interfaces of USART6. USART6 is mainly needed to connect with another board. At the same time, the clocks of these two USARTs need to be turned on. Next is the initialization of the UART. In the `uart.c` file, we only need to provide the handle of the UART and the BaudRate to complete the initialization.

Then there are the `uart_getchar` equations and the `uart_putchar` equations that have been written in the `uart.c` file. Their corresponding functions are to get a character from RX and send a character to TX. Their basic function is to use `HAL_UART_Receive` and `HAL_UART_Transmit`. After analyzing the files we need in the `uart.c` file, we can start writing the main program. First, you need to initialize the UART, declare two `UART_HandleTypeDef`, corresponding to the IO of the terminal and the IO of another board. Then use `initUart` and `HAL_UART_MspInit` to initialize them.

```
initUart(&USB_UART0, 115200, USART1);  
HAL_UART_MspInit(&USB_UART0);
```

Then we need to write a send and receive functions respectively. The send function needs to read the keyboard input on the terminal first and then send it to another board, so we first use `uart_getchar` to read the terminal input, and then make a judgment.

```
wordss = uart_getchar(&USB_UART0, 1);  
uart_putchar(&DISCO_UART, &wordss);
```

If the sent character is not empty, then use the `uart_getchar` function to send it to another board. on the RX of a board. Then make a judgment, if the input of the

keyboard is ECS, enter an infinite loop. Finally, reset the value of wordss to 0 to prevent infinite sending. Finally, there is the receive function. This equation is easy to write. It accepts the data from RX through the uart_getchar equation,

```
wordsr = uart_getchar(&DISCO_UART, 0);
```

stores it in another variable, and then prints the return value with printf, and then adds a line to judge, If the accepted value is ECS, enter an infinite loop. Putting these two equations into while(1) completes all the requirements of this task.

Results and Analysis

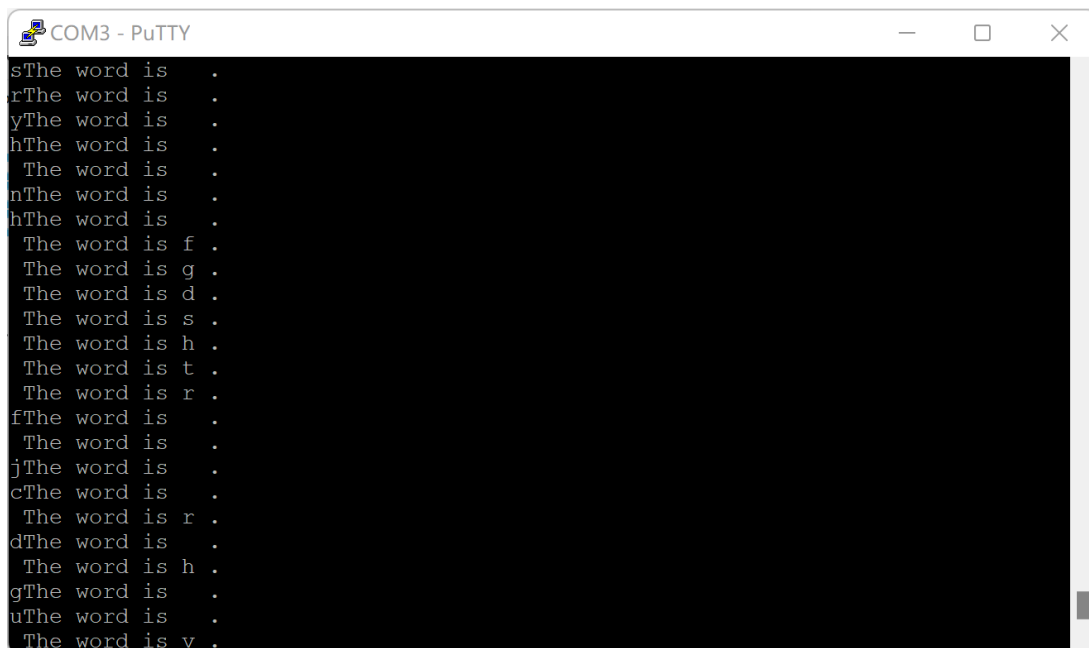
The first and second screenshots show that the normal running sequence from both terminals. The character before sentence is the one received from keyboard. The character in the sentence is received from the other board.

The last 2 screenshots show the exit function on both terminals.



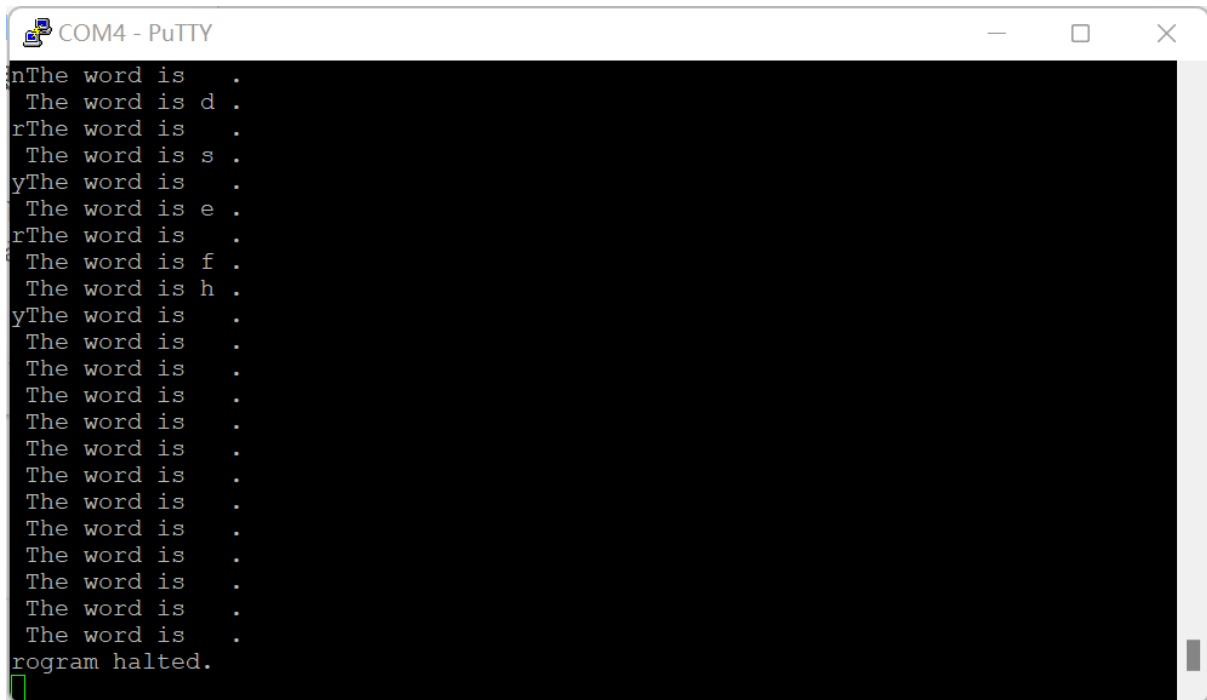
```
COM4 - PuTTY
The word is r .
The word is y .
The word is h .
The word is .
The word is n .
The word is h .
fThe word is .
gThe word is .
dThe word is .
sThe word is .
hThe word is .
tThe word is .
rThe word is .
The word is f .
The word is .
The word is j .
The word is c .
rThe word is .
The word is d .
hThe word is .
The word is g .
The word is u .
vThe word is .
The word is .
```

Figure 1: normal running on terminal 1



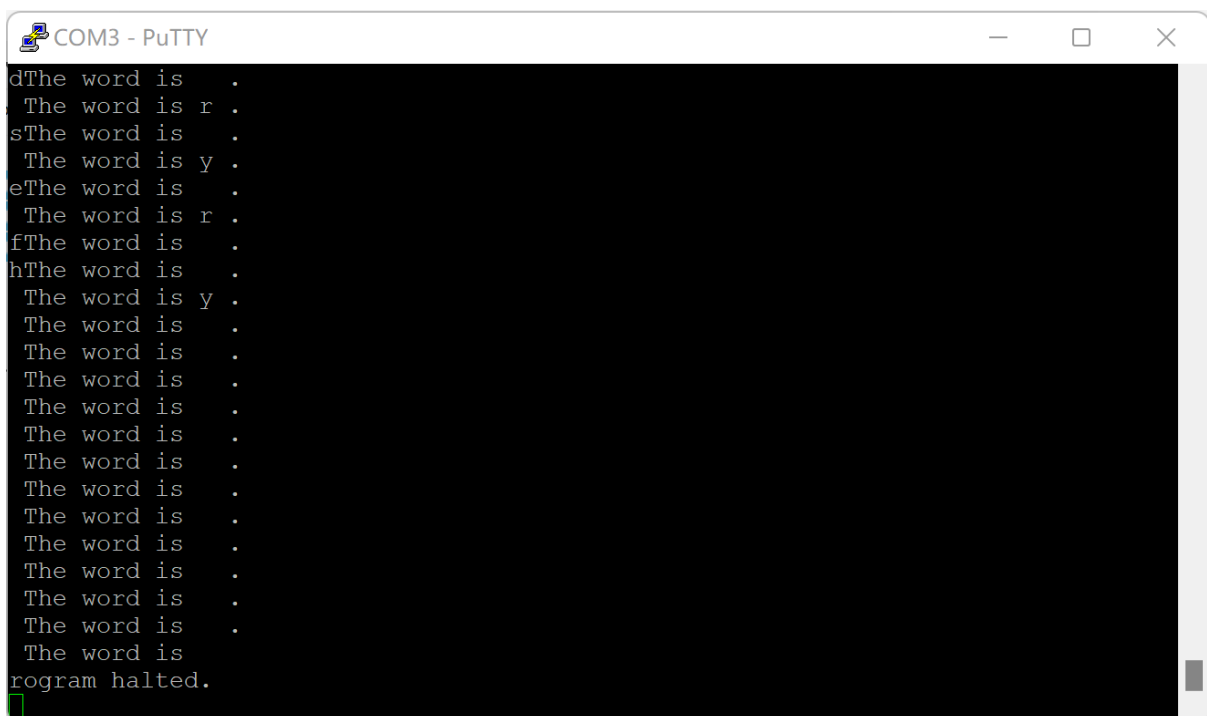
```
COM3 - PuTTY
sThe word is .
rThe word is .
yThe word is .
hThe word is .
The word is .
nThe word is .
hThe word is .
The word is f .
The word is g .
The word is d .
The word is s .
The word is h .
The word is t .
The word is r .
fThe word is .
The word is .
jThe word is .
cThe word is .
The word is r .
dThe word is .
The word is h .
gThe word is .
uThe word is .
The word is v .
```

Figure 2: normal running on terminal 2



```
COM4 - PuTTY
nThe word is .
The word is d .
rThe word is .
The word is s .
yThe word is .
The word is e .
rThe word is .
The word is f .
The word is h .
yThe word is .
The word is .
The word is .
The word is .
The word is .
The word is .
The word is .
The word is .
The word is .
The word is .
program halted.
█
```

Figure 3: Program halted on terminal 1



```
COM3 - PuTTY
dThe word is .
The word is r .
sThe word is .
The word is y .
eThe word is .
The word is r .
fThe word is .
hThe word is .
The word is y .
The word is .
The word is .
The word is .
The word is .
The word is .
The word is .
The word is .
The word is .
The word is .
The word is .
The word is .
program halted.
█
```

Figure 4: Program halted on terminal 2

Task 2: Two-Terminal UART (Interrupt)

Introduction

In Task 1, a program was made that continuously checked whether any characters were received 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 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. For the case of UART character reception, it is desired to trigger interrupts when a character is received by either UART port.

We need to use interrupts in Task2, which is a very common program. But the additional task requirements make this task not very simple to complete. The task requires that we cannot write any code in the while(1) loop of the main() function, that is, we have to use the interrupted HAL_UART_RxCpltCallback() equation throughout to achieve the function we need. One thing to note is that this interrupt function is triggered after the HAL_UART_Receive_IT() function is called. That is to say, we need to call HAL_UART_Receive_IT() in the HAL_UART_RxCpltCallback() equation to achieve a continuous loop process.

Low Level Description

The initialization has been completed in Task 1, so we directly write the main part of the program at the beginning. First we need to complete the two IRQHandler equations. They handle interrupt signals from the terminal and another board respectively.

```
void USART1_IRQHandler()  
{  
    HAL_UART_IRQHandler(&USB_UART0);  
}
```

Then we need to complete the HAL_UART_RxCpltCallback equation. HAL_UART_RxCpltCallback has an input parameter, which is the HandleTypeDef that triggers the interrupt. First of all, we must determine who triggered the interrupt, because whether it is the terminal or another board, their interrupt will trigger the Callback function. The equation will be mainly divided into two parts, the first part is the interrupt of the terminal, and the other part is to handle the interrupt of the other board. The judgment condition here is judged by huart->Instance.

If the judgment condition is from the terminal, we first pass the received character through the HAL_UART_Transmit_IT equation to the RX terminal of another board through the TX terminal. Then we need to judge if this input is the case for ECS, and if so, the program will enter an infinite loop. Then we print the received signal on the terminal and call the HAL_UART_Receive_IT equation to enter the next loop.

```
HAL_UART_Receive_IT(&USB_UART0, (uint8_t *)input_c, 1);
```

If the interrupt comes from another board, we need to immediately determine whether the input signal is ECS. If true, the program will enter an infinite loop. Then we print out the received information. Finally call HAL_UART_Receive_IT to enter the next loop.

```
HAL_UART_Receive_IT(&DISCO_UART, (uint8_t *)input_d, 1);
```

Because the callback equation cannot be entered without calling the HAL_UART_Receive_IT equation, we need to add the initial call condition to the main() function, that is, call the function above and two different parameters respectively.

Results and Analysis

The first and second screenshots show that the normal running sequence from both terminals. The last 2 screenshots show the exit function on both terminals.

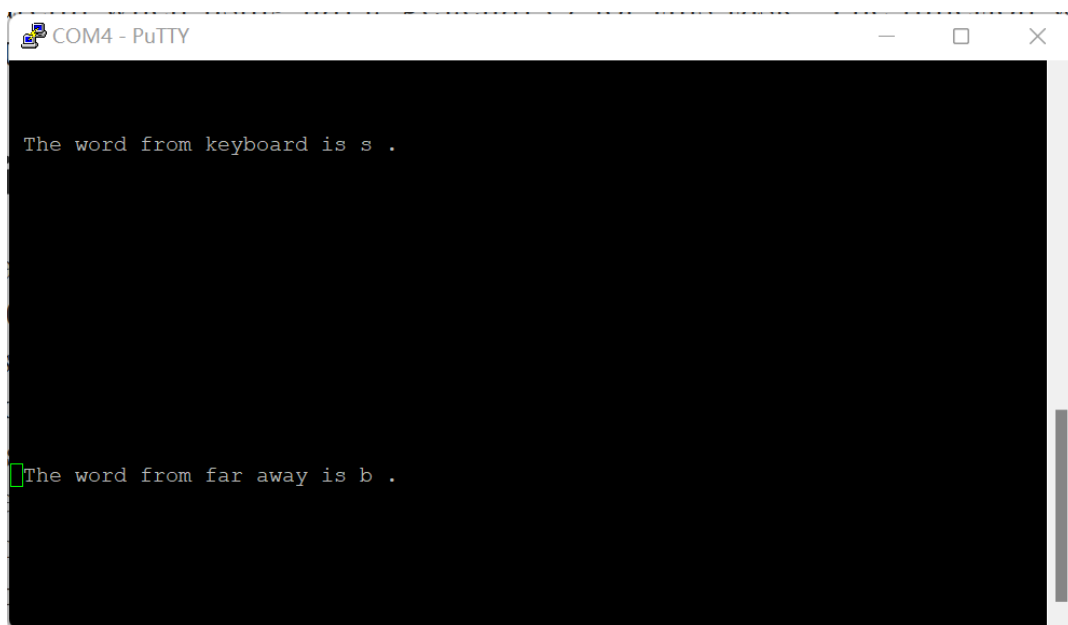


Figure 5: normal running on terminal 1

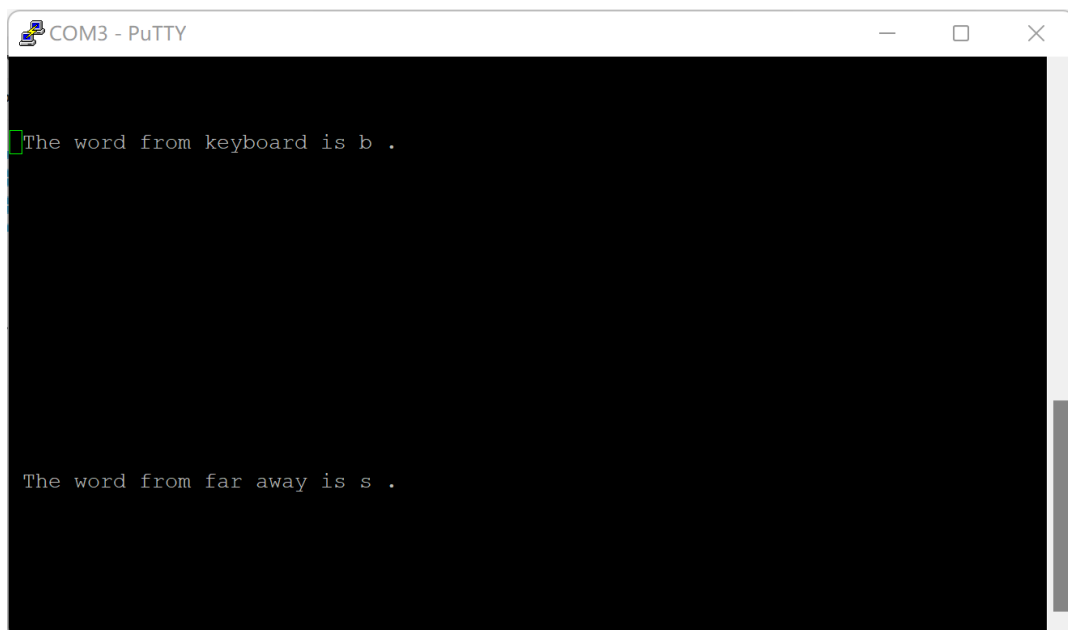


Figure 6: normal running on terminal 2

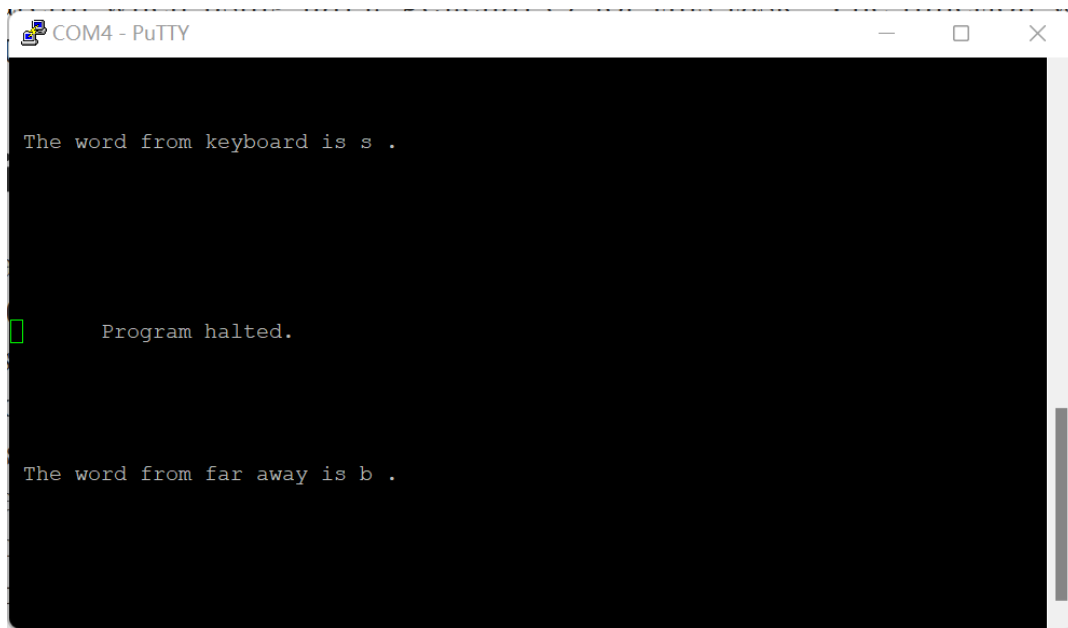


Figure 7: Program halted on terminal 1

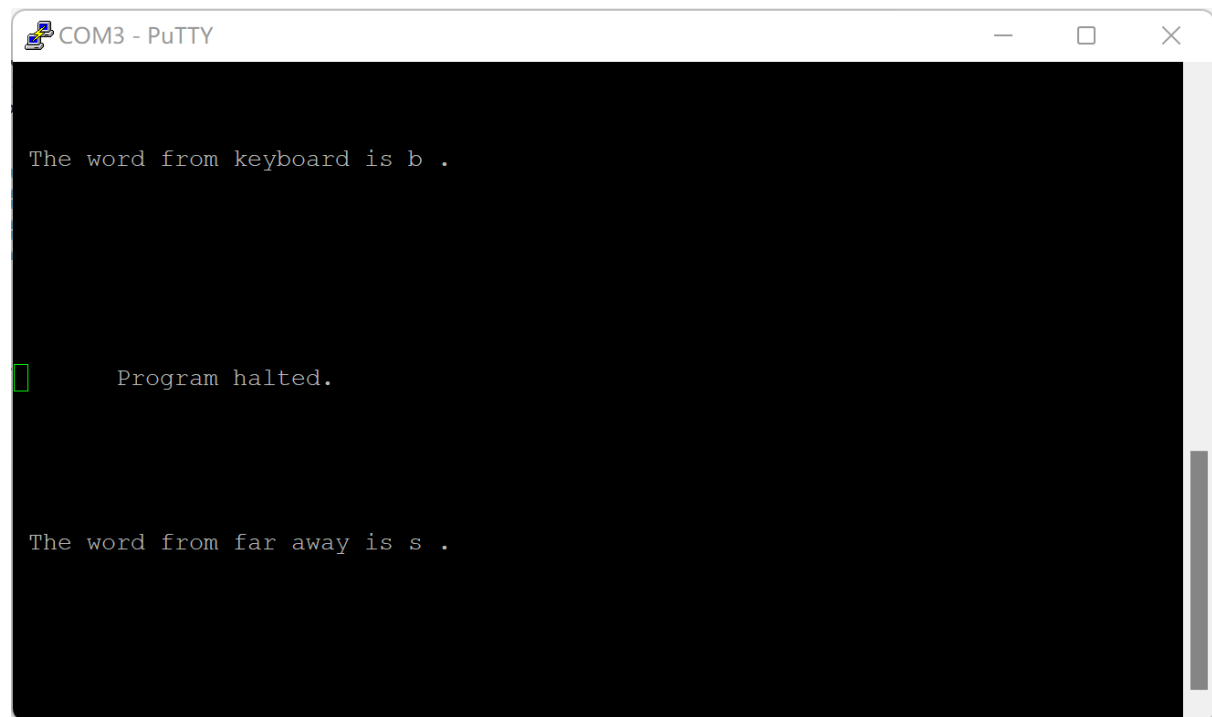


Figure 8: Program halted on terminal 2

Task 3: SPI Loopback Interface

Introduction and High-Level Description

The purpose of this task is to write a program to send and receive the message through the SPI bus. Various systems can be formed under the control of software using SPI. For example, a master controller and several slave controllers, several slave peripherals are connected to each other to form a multi-master system (distributed system), various systems formed by a master controller and one or several slave I/O devices, etc. In most applications, a master controller can be used as the master to control data and transfer the data to one or several slave peripheral devices.

SPI is also divided into two modes in the transmission part, one is Transmit Mode, its specific function is that data can be transmitted to MOSI but MISO data will be ignored. Another is Receive Mode. This mode will transfer unimportant data to MOSI, and the data will be uploaded to MISO and saved. There are shift registers inside the master and slave devices. The master writes one byte of data into the shift register to initiate a transmission. The master sends the data to the slave, and the slave also sends the data in its shift register to the master. The contents of the shift registers are swapped. The read and write operations of peripherals are completed synchronously. If the host only wants to send data to the slave, the data sent by the slave can be ignored; if the host only wants to receive data, the host can send a null byte to drive the slave. machine to send data.

The goal of this task is to write an SPI program that can transmit and receive signals. For this task, the controller is STM32F769NI, and the peripheral is also STM32F769NI. That is to say, we want to write a closed-loop program to read character input from the terminal's keyboard and send the obtained character through SPI. Next, the character is received again via SPI. Because it is sending a message to myself, both of the MISO and MOSI of the SPI will be opened. The initial task of this SPI is relatively simple. After completing the initialization of the SPI and the opening of the GPIO, the input is read through the equation of reading the keyboard input written by the previous Task, and then basically the `HAL_SPI_TransmitReceive()` equation can be used to complete the task.

Low Level Description

The first thing to do is the initialization part of the SPI. First declare a parameter of the SPI_HandleTypeDef attribute to set the parameter attributes of the SPI we need. For most STM32 tools, the first thing is to turn on the clock. SPI is the same, the first step is to open the clock. Then the instance of SPI selects SPI2. Select the mode of SPI as SPI_MODE_MASTER. Select two-way communication and set the data size to 8bit. For the NSS part of the chip select, we choose SPI_NSS_HARD_OUTPUT. For our particular task, a BaudRatePrescaler selection of 8x is sufficient. Finally, we formally initialize our properties through the HAL_SPI_Init equation. In this way, the initialization of the SPI is completed.

Then we need the HAL_SPI_MspInit equation to set the GPIO interface required by SPI. By consulting the User manual, we can know that B14 and B15 are MISO and MOSI respectively. Finally, enable SPI GPIO port clocks, set HAL GPIO init structure's values for each through HAL_NVIC_EnableIRQ equation.

Then comes the focus of the main function, first reading the keyboard input from the terminal using the UART we used in the previous task. Next, an equation called HAL_SPI_TransmitReceive will be used.

```
HAL_SPI_TransmitReceive(&SPI_2, (uint8_t*) wordss, (uint8_t*) wordsr, 1, 1000);
```

This equation has three inputs, SPI_HandleTypeDef, pTxData, pRxData, Size and Timeout. The specific function of this equation is to transmit the data of pTxData and store the data received from the slave to pRxData. Then we will interact with the data through the HAL_SPI_TransmitReceive equation, and finally print the contents of the reception data buffer, and all the functions of this task are realized.

Results and Analysis

The program would echo a message as expected.

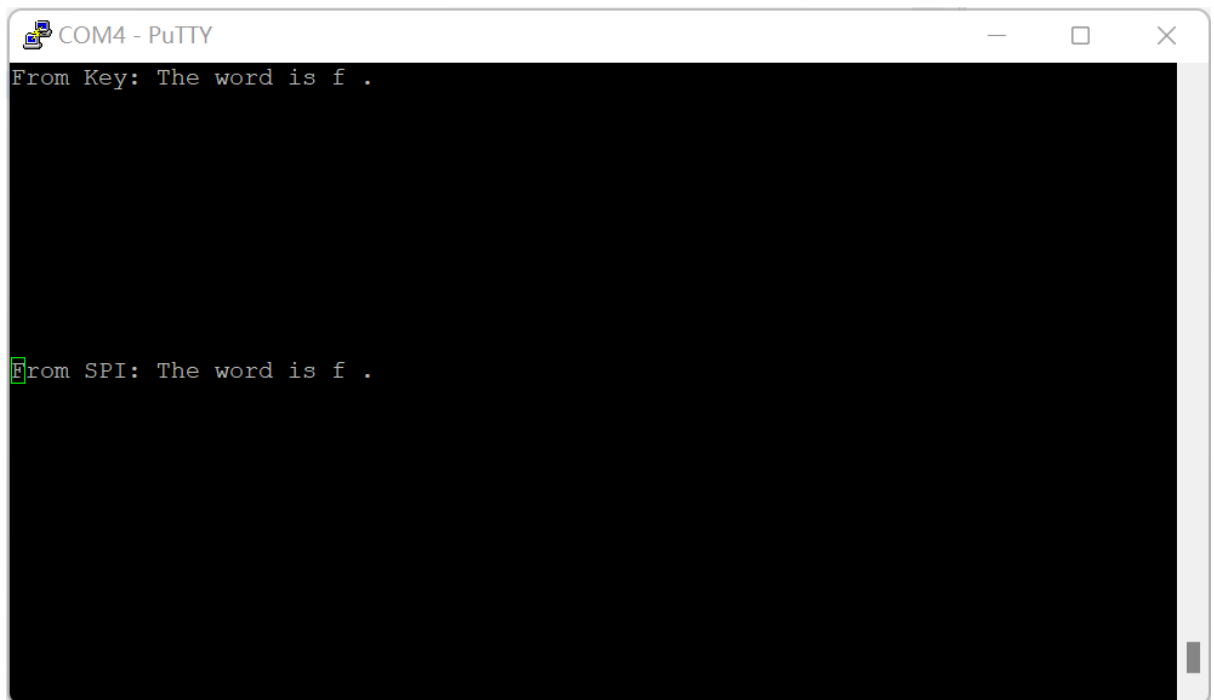


Figure 9: normal running

Task 4: Connect to STM32-Based SPI Device

Introduction and High-Level Description

Task 4 uses two different microprocessor boards to communicate with each other by using SPI buses. The STM32F769NI is the master board, and users can control the board by using a terminal. While a new Nucleo-F303RE board is a slave board. There is output on its terminal, but the user should not control it.

In Task 4, the program should have a menu, allowing the user to choose various functions. 2. Receive terminal characters from the peripheral device using SPI. The received characters should be printed on the bottom half of the controller's terminal. 3. Read the peripheral's firmware version upon startup. 4. 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. 5. Clear or reset the peripheral terminal. 6. Change and read the device ID of the peripheral. All these functions should be running in the master board and controlled in the terminal.

The slave board should be intalled a firmware that is provided here.¹ This firmware should be installed by ST-LINK Utility. After installation, rebooting the board does not erase the firmware. This firmware is specified for Task 4, which stores and prints out the character in the terminal, and prints out raw temperature.

The wiring of this task is much. From the master board, Chip Select(CS/NSS, D10) should be connected to slave Chip Select(A2). The two SYSCLK (both D13) are connected. The two MISO(both D12) are connected. The two MOSI(both D11) are connected. The grounds are connected. This task use Analog Discovery to verify the SPI bus operation. All the connections above are linked to the Analog Discovery by digital logic lines. Also, there is a Debug line(D7) on the slave board, which does not transfer message, but it is helpful for debugging.

¹ https://sites.ecse.rpi.edu/courses/static/ECSE-4790/Labs/Lab03/LAB-03-STaTS_Firmware-F303RE.bin

Low Level Description

SPI Bus is discussed in Task 3. For initialization of the SPI Bus, most of the parameters are the same, but the BaudRatePrescaler should be 256, because the CPU frequency of the master board is 216MHz, so it should be divided into a smaller number for the bus to communicate.

The most essential code used in this task is HAL_SPI_TransmitReceive function. The whole process for the SPI Bus to transmit messages is the following steps: 1. Pull Chip Select low. 2. Wait about 10 us. 3. transmit the reference(&) to the register number. 4. Wait about 10 us. 5. transmit or receiver the pointer to the data. 6. Wait about 10 us. 7. Pull Chip Select high. Indicated by the following code.

```
HAL_GPIO_WritePin (GPIOA, GPIO_PIN_11, GPIO_PIN_RESET); // pull CS low
for(ii = 0; ii < 160; ii++){ // wait 10us
    int a = 0;
    HAL_SPI_TransmitReceive(&SPI_2, (uint8_t *)&a, (uint8_t *)output, 1,
1000); // choose register 0
    for(ii = 0; ii < 160; ii++){ // wait 10us
        int b = 4;
        HAL_SPI_TransmitReceive(&SPI_2, (uint8_t *)&b, (uint8_t *)output, 1,
1000); // transmit 4 to register 0, receive data from bus to output
        for(ii = 0; ii < 160; ii++){ // wait 10us
            HAL_GPIO_WritePin (GPIOA, GPIO_PIN_11, GPIO_PIN_SET); // pull CS high
```

For the slave board, there are in total 10 registers to use. The register are previously programmed by the firmware, so the SPI Bus can just transfer the needed information to the slave board and the slave board can automatically return the requested value. There are 8 registers that will be used. They will be discussed in the sub-tasks that are used.

Receive terminal characters

For receiving terminal characters, STS_REG (Reg#: 1) and CH_BUF (Reg#: 5) will be used. Bit 5 (NCHBF0) and 6 (NCHBF1) of STS_REG are the buffers that indicate how many characters are in the terminal receiving buffer (CH_BUF) awaiting transmission (maximum 3 characters). CH_BUF is a FIFO

buffer that holds up to three characters. Writing a value to CH_BUF will result in the ASCII value being transmitted to the connected terminal. To avoid writing to the terminal when reading from the buffer, write a value of 0x00.

The program should be always checking if the NCHBF0 and NCHBF1 are 1, by using the following.

```
if (((check & 0b01100000) == 0b01100000) || ((check & 0b01000000) == 0b01000000) || ((check & 0b00100000) == 0b00100000))
```

If the buffer has characters in it, the program should receive the data from CH_BUF and print it out.

Read the peripheral's firmware

For reading the peripheral's firmware, V_MAJ (Reg#: 7) and V_MIN (Reg#: 8) will be used. V_MAJ and V_MIN together form the full version number of the firmware installed on the STaTS device; where the firmware version is [V_MAJ].[V_MIN]. The program should choose each register, and receive the data to a container. After both registers are chosen and received, the full firmware version can be printed out.

Temperature measurement

To read a temperature measurement, CTL_REG (Reg#: 0), STS_REG (Reg#: 1), TMP_LO (Reg#: 3) and TMP_HI (Reg#: 4) are used. Bit 1 (RDTMP) of CTL_REG requests a new temperature measurement to be taken. Bit 3 (TRDY) of STS_REG is 1 if a temperature measurement is ready for reading. This bit is automatically cleared when TMP_HI is read. TMP_LO and TMP_HI store the low and high byte of the temperature measurement. The value of the temperature is a 12-bit number, right justified. The full 12-bit number is the averaged values of the raw ADC measurements of the temperature sensor.

For this sub-task, the program should first set RDTMP to high to start a new temperature measurement. Then it should go into a loop, wait until TRDY is 1. After that the program should choose each of the two temperature register, receive the data to containers.

To calculate the raw and exact temperature, the data should be modified. The raw temperature should be 12-bit, the data from the TMP_LO are the lower 8 bits, but the higher 4 bits are from TMP_HI, which can be calculated like this.

```
tempraw = (temphigh << 8) + templow;
```

The equation to convert the raw temperature to temperature in Celcius is $357.6 - 0.187 \text{tempraw}$. It should be reminded that the calculated temperature is double, so in the printf function, %f should be used to print. If the IDE warns “*The float formatting support (-u_printf_float) is not enabled from linker flags*”, “-u_printf_float” should be added here.

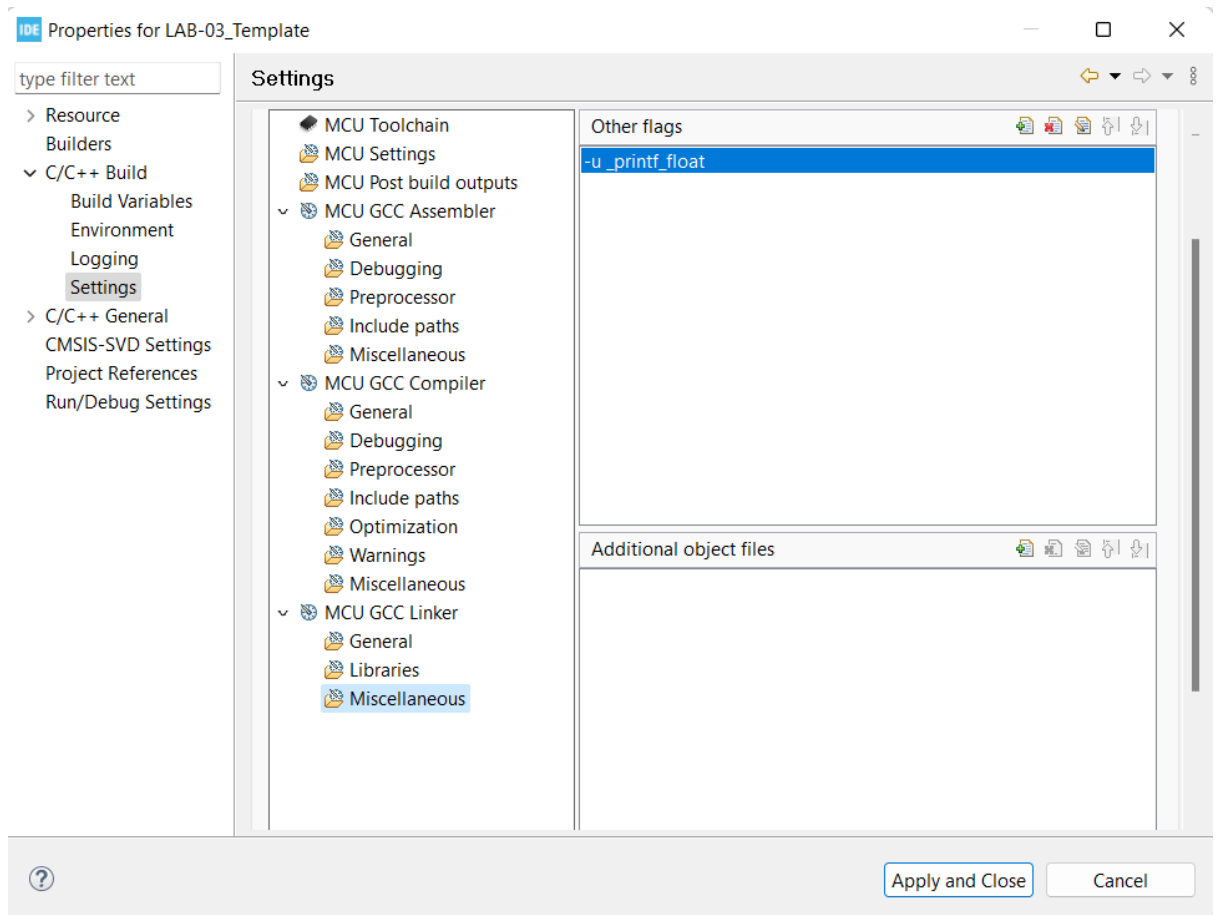


Figure 10: Environment set of float in IDE

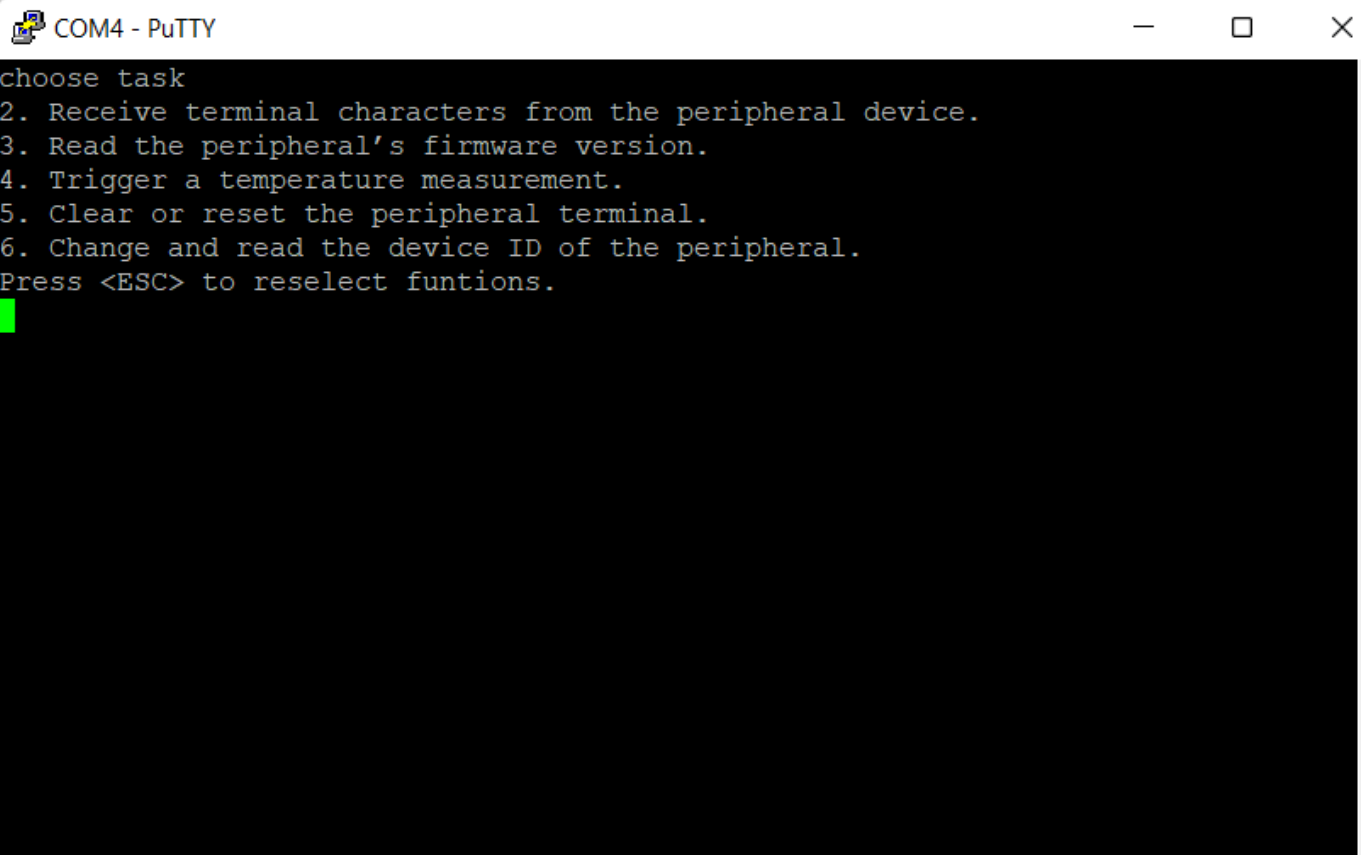
Clear peripheral terminal

To clear the peripheral terminal, CTL_REG (Reg#: 0) will be used. Bit 2 (TRMCLR) or Bit 3 (TRMRST) of the CTL_REG can clear the terminal driven by the STaTS device. The difference is that TRMCLR does not reset the text attributes, but TRMRST does. In this sub-task, the program does not change the text attributes, so either of these two bits can be high.

Change and read the Device ID

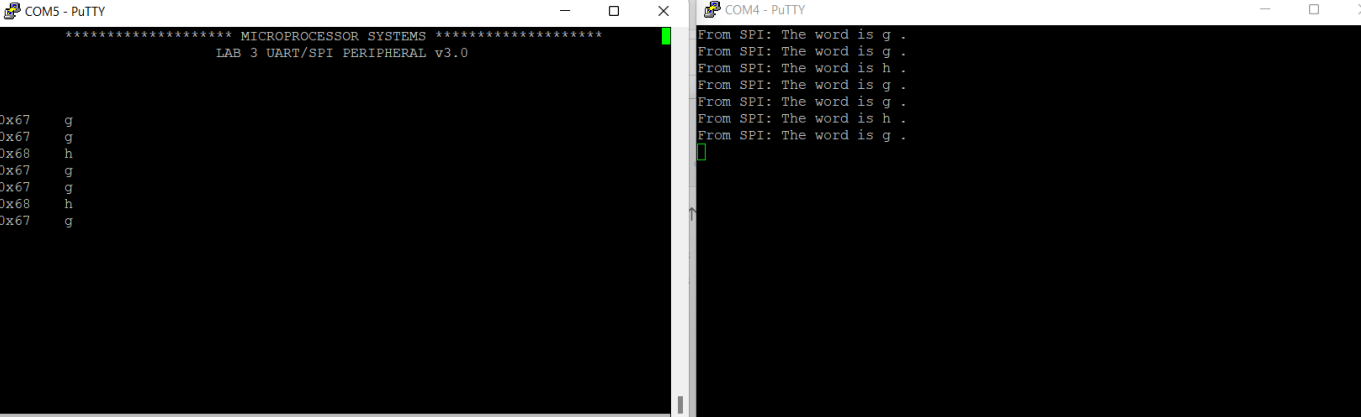
For changing and reading the device ID of the peripheral, CTL_REG (Reg#: 0) and DEVID (Reg# 9) will be used. Bit 7 (ULKDID) unlocks the Device ID for writing. It will be re-locked again after the next full SPI register+data word transfer. DEVID is a Read/Write register that will reset value 0xB6. Changes to the DEVID register do not survive power cycles or system resets. To read the device ID, ULKDID should not be written to 1, so the register can be read only. For this sub-task, the program is designed to change the Device ID to 0x04.

Results and Analysis



```
COM4 - PuTTY
choose task
2. Receive terminal characters from the peripheral device.
3. Read the peripheral's firmware version.
4. Trigger a temperature measurement.
5. Clear or reset the peripheral terminal.
6. Change and read the device ID of the peripheral.
Press <ESC> to reselect funtions.
```

Figure 11: Task 4 sub-tasks menu



```
COM5 - PuTTY
***** MICROPROCESSOR SYSTEMS *****
LAB 3 UART/SPI PERIPHERAL v3.0

0x67 g
0x67 g
0x68 h
0x67 g
0x67 g
0x68 h
0x67 g
0x00

COM4 - PuTTY
From SPI: The word is g .
From SPI: The word is g .
From SPI: The word is h .
From SPI: The word is g .
From SPI: The word is g .
From SPI: The word is h .
From SPI: The word is g .
```

Figure 12: Receive terminal characters from slave to master

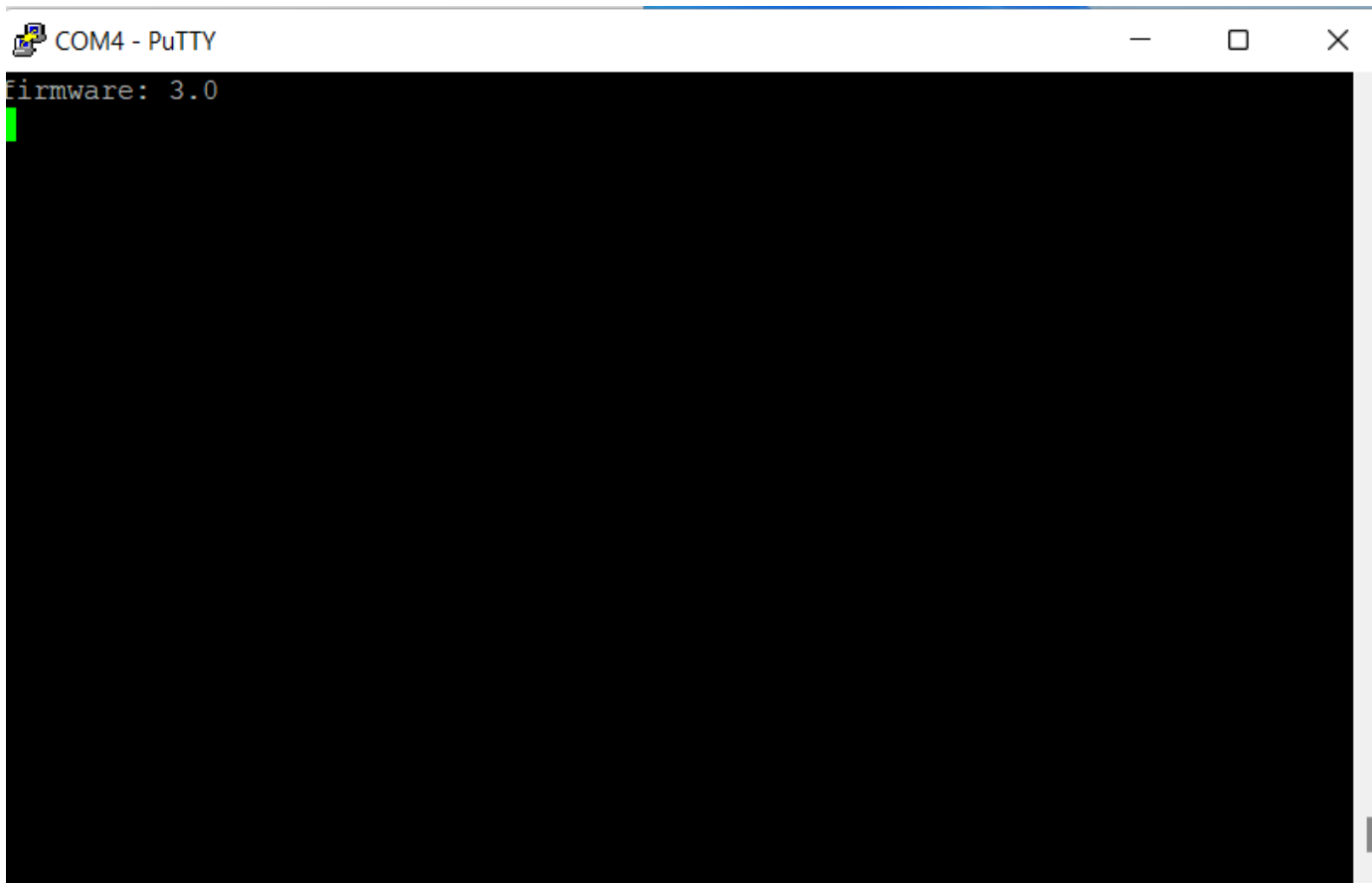


Figure 13: Read the peripheral's firmware

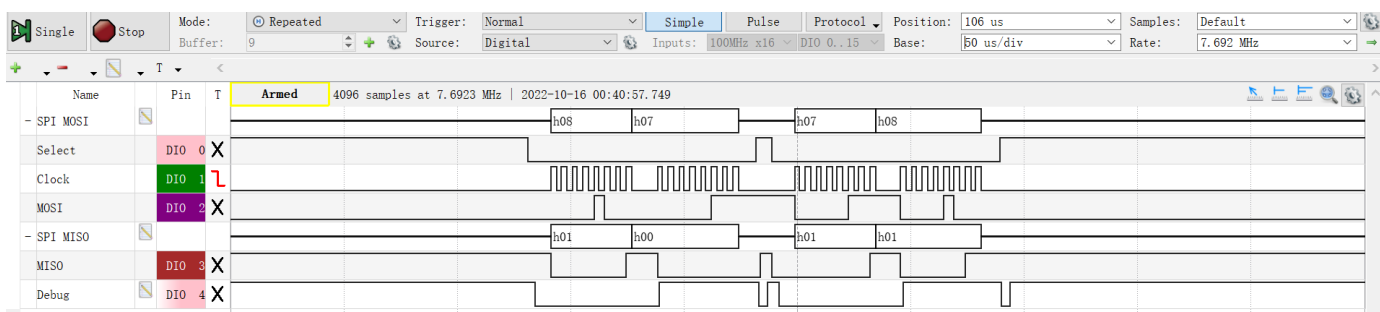


Figure 14: Firmware waveform

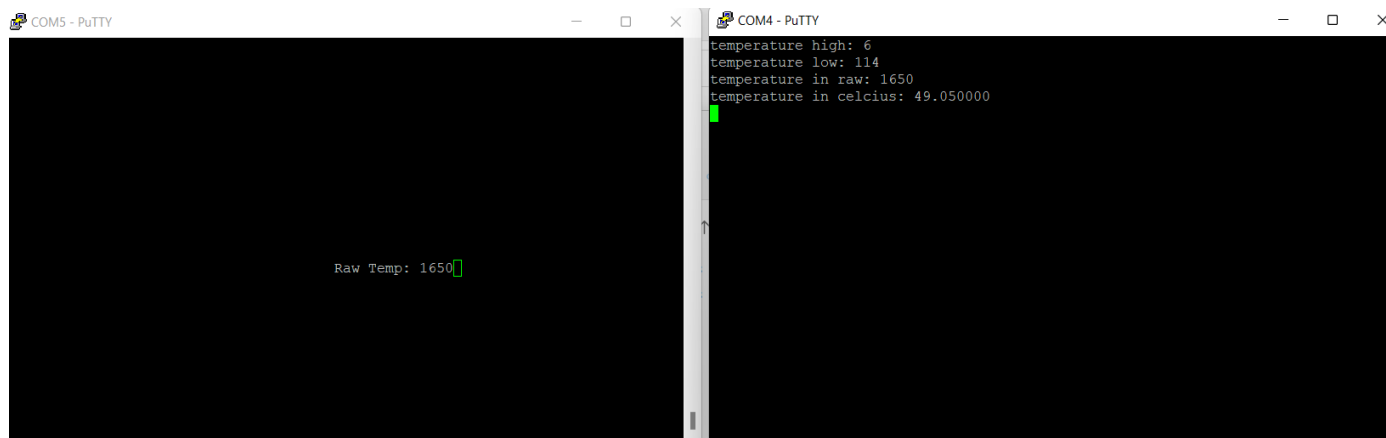


Figure 15: Temperature measurement

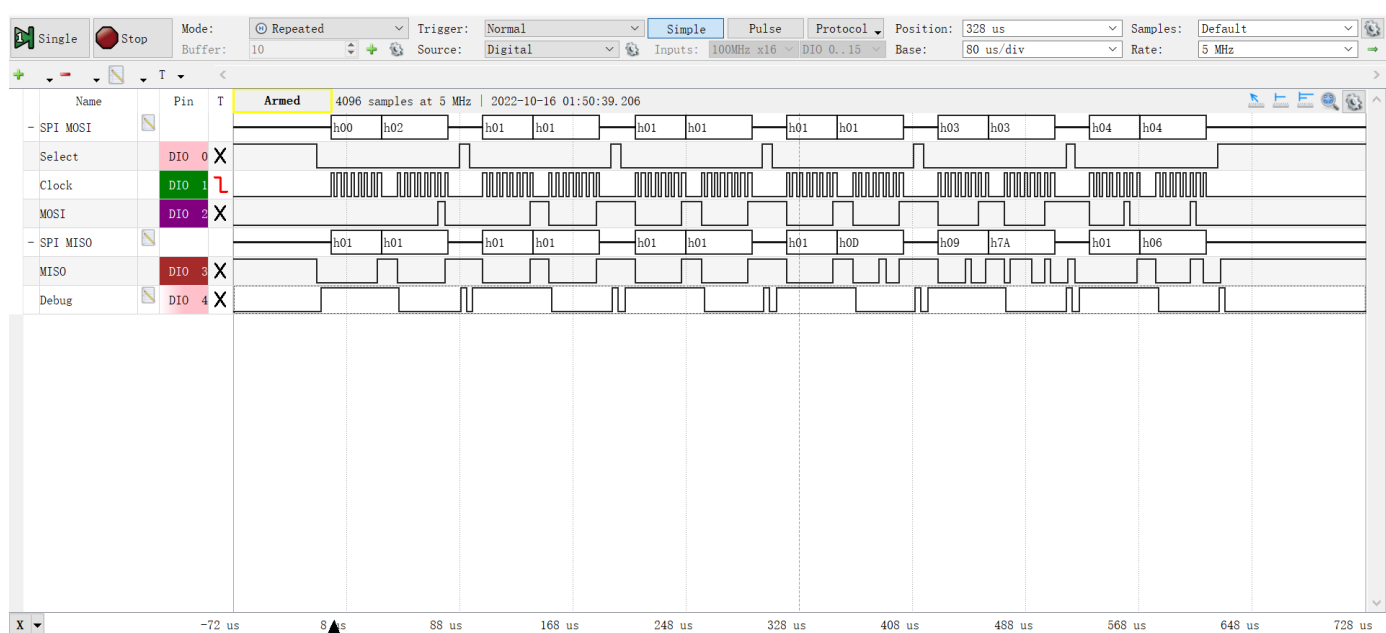


Figure 16 : Temperature waveform

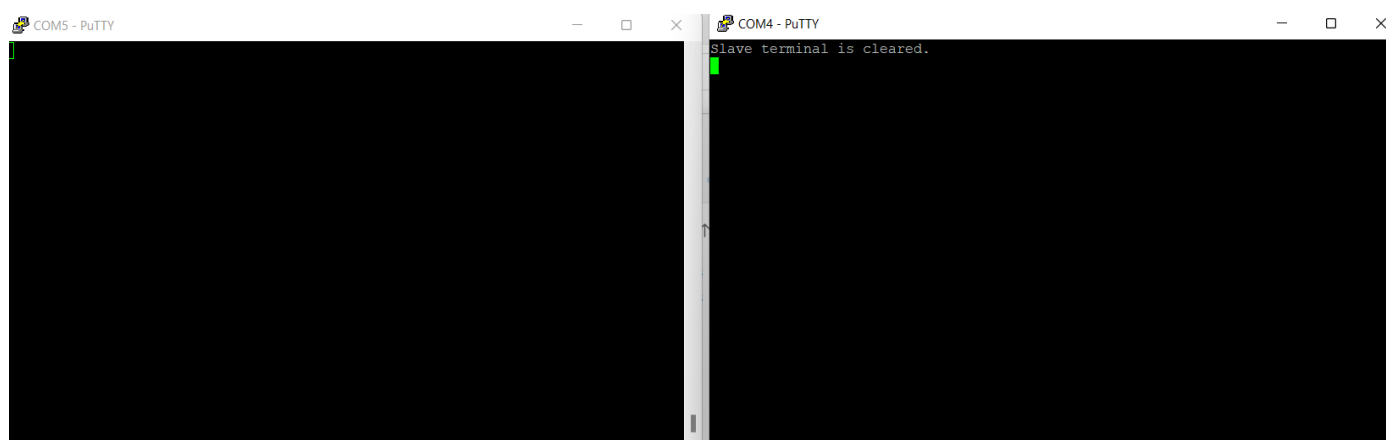


Figure 17: Clear peripheral terminal

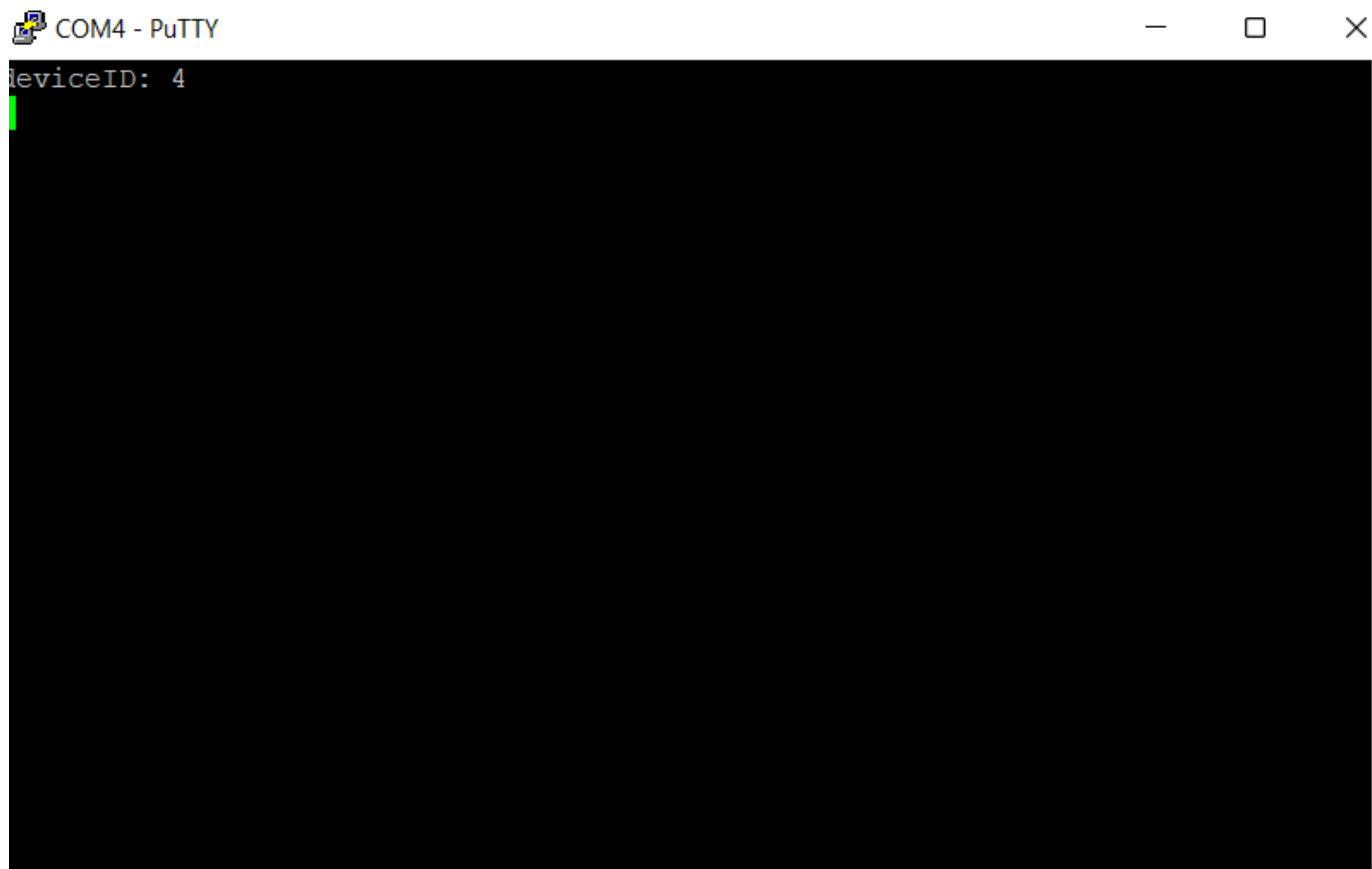


Figure 18: Change and read the Device ID

The results are as expected. Most of the waveforms are the same, so only two representative waveforms are shown. In the temperature waveform (Figure 16), the second, third and fourth period is the program waiting for the temperature to be taken, that is Reg 1 Bit 3 is high.

There is also a problem that happened in the waveform. For some unknown reason, Bit 7 of the slave register cannot last long enough at high, so the master board cannot successfully read data that is larger than 127. This may be caused by clock frequency, but there is no evidence about it.

Conclusions

Through this experiment, we learned how to transfer data between two boards. In the first part we learned how to use UART, and in the second part we learned how to use the SPI bus. The main difference between UART and SPI is that UART is asynchronous but SPI is synchronous. Here are some discussions of some real-world applications of these topics.

UART

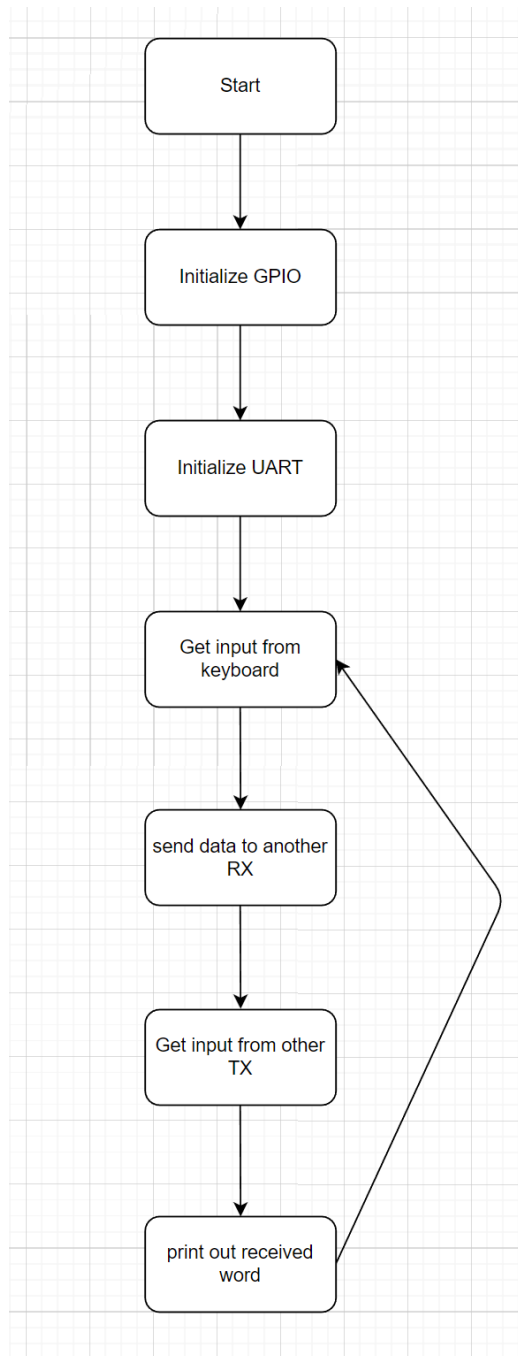
UART is one of the most simple and most commonly used Serial Communication techniques. Today, UART is being used in many applications like GPS Receivers, Bluetooth Modules, GSM and GPRS Modems, Wireless Communication Systems, RFID based applications etc.

SPI

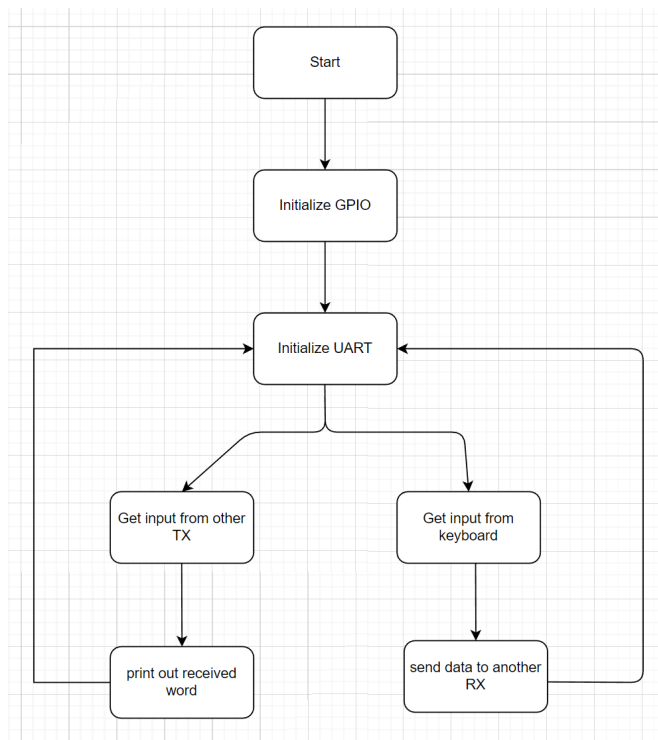
Chip or FPGA based designs sometimes use SPI to communicate between internal components; on-chip real estate can be as costly as its on-board cousin. SPI is used to talk to a variety of peripherals, such as temperature sensors, ADC, touchscreens, video game controllers.

Appendix

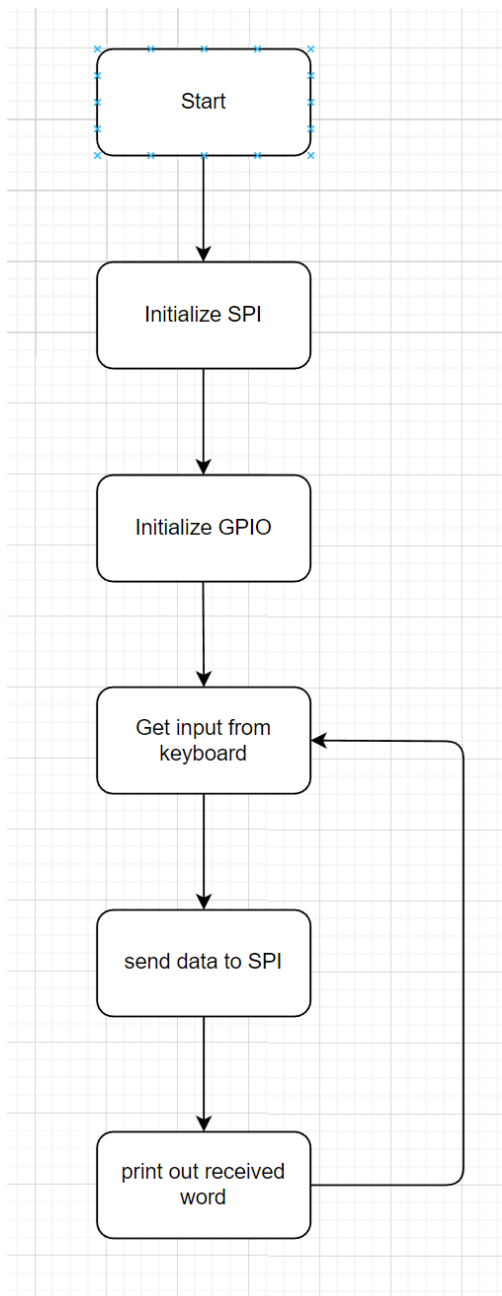
[1] Task 1 flowchart.



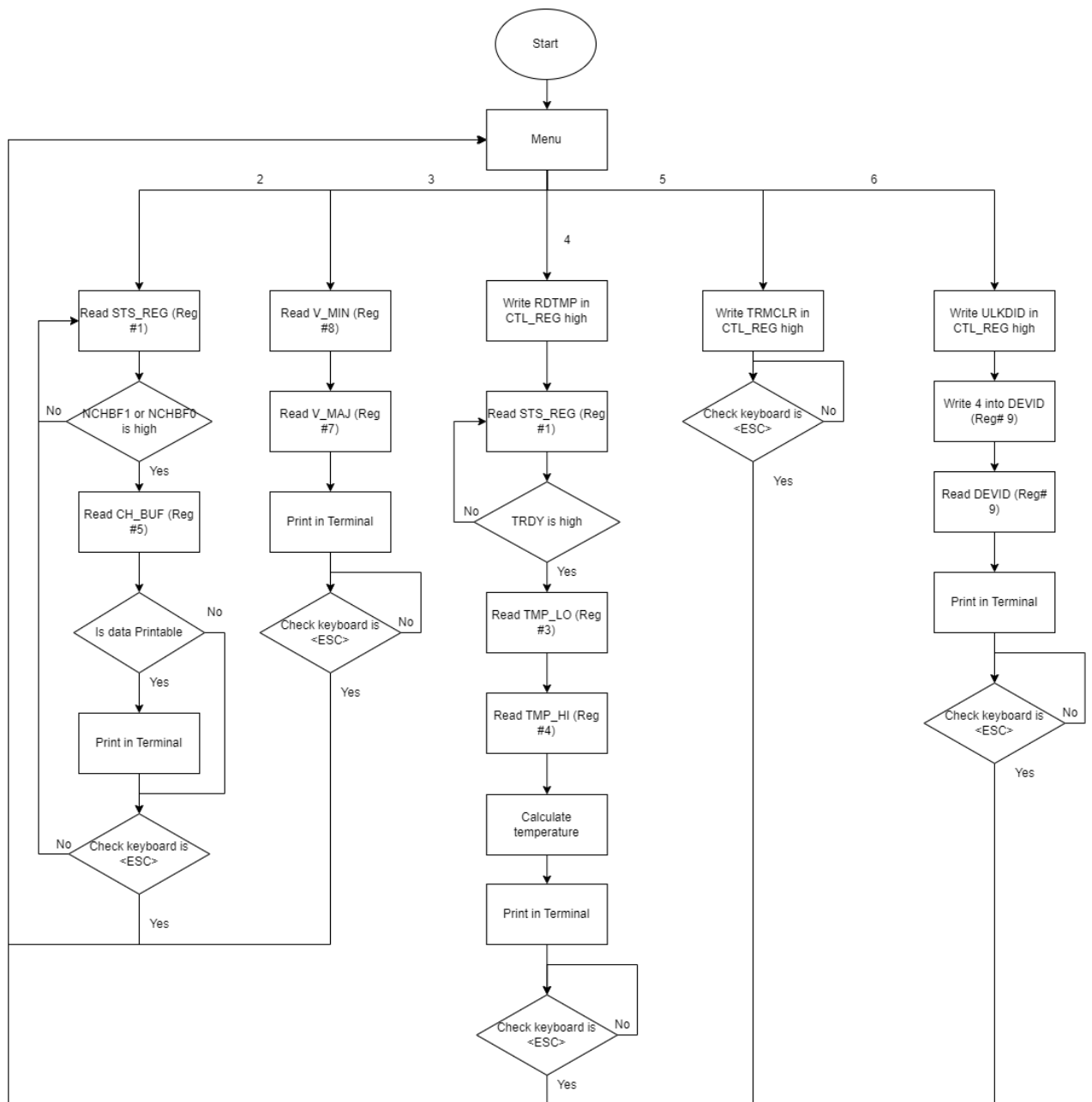
[2] Task 2 flowchart.



[3] Task 3 flowchart.



[3] Task 4 flowchart.



MPS**Lab03: Serial Communication****Microprocessor Systems Lab 3****Checkoff and Grade Sheet****Partner 1 Name:** Weiheng Zhuang**Partner 2 Name:** ZiYu Zhu

Grade Component	Max.	Points Awarded	TA Init.s	Date
Performance Verification: Task 1	5 %	05	TAC	10.13.22
Task 2	10 %	10	TAC	10.13.22
Task 3	5 %	05	TAC	10.13.22
Task 4	20 %	20	TAC	10.13.22
Task 5 [Depth]	10 %			