**MPS Serial Communication Lab Exercise**

**Asynchronous & Synchronous Serial Communications Interface**

Student's name & ID (1): \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Partner's name & ID (2): \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Your Section number & TA's name \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Notes:**

You must work on this assignment with your partner. Hand in a printer copy of your software listings for the team. Hand in a neat copy of your circuit schematics for the team.

These will be returned to you so that they may be used for reference.

------------------------------- do not write below this line -----------------------------

|  |  |  |  |
| --- | --- | --- | --- |
|  | POINTS (1) (2) | | TA init. |
| Grade for performance verification (50% max.) | | | |
| Part 1 (10% max.) |  | |  |
| Part 2 (15% max.) |  | |  |
| Part 3 (25% max.) |  | |  |
| Enhancement (10% max) |  | |  |
| Grade for answers to TA's questions (20% max.) |  |  |  |
| Grade for documentation and appearance (20% max.) |  | |  |
|  |  |  | TOTAL |

Grader's signature: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Date: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Asynchronous & Synchronous Serial Communications Interface**

**GOAL**

By doing this lab assignment, you will learn to program and use:

1. The Asynchronous Serial Communications Ports and the Synchronous Serial Peripheral Interface.

2. Serial communications among multiple processors.

**PREPARATION**

• References: *769 Reference Manual (Register Map).pdf*

Ch. 34 (UART), 35 (SPI)

*Mastering STM32*

Ch. 8 (UART; skip 8.2.1 and 8.3.1-8.3.2), 15 (SPI; skip 15.3)

*769 Description of HAL Drivers.pdf*

Ch. 66 (UART), 62 (SPI)

(Reference only; alternatively, “stm32f7xx\_hal\_uart.h/.c” and

“stm32f7xx\_hal\_spi.h/.c” contain more or less the same info in their

comments)

*769 Datasheet (Alternate Functions etc).pdf*

Ch. 4 (Table 13 only), Ch. 5.3.1 (Table 17 only), Ch. 5.3.7 (Table 39 only)

(These tables should help with clock rate determination)

**UNIVERSAL ASYNCHRONOUS RECEIVER TRANSMITTER (UART)**

1. **Introduction to the Asynchronous Serial Communications Port**

The DISCO has several on-board universal synchronous/asynchronous receiver/transmitter (USART) interfaces and an additional synchronous serial peripheral interface (SPI). The DISCO board has one built-in virtual UART communications channel over USB which USART1 is configured to use. The USARTs can be configured through a hardware abstraction layer (HAL) structure called UART\_HandleTypeDef, the fields of which are listed below.

typedef struct {

USART\_TypeDef \*Instance; /\* UART registers base address \*/

UART\_InitTypeDef Init; /\* UART communication parameters \*/

...

} UART\_HandleTypeDef;

The UART\_InitTypeDef is another struct containing the following parameters, which set up how the UART operates to exchange data:

x

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;

In more sophisticated configurations, interrupts may be generated when data is received or ready to be transmitted and when transmission, framing, or overrun errors are detected. Self-checking modes may also be configured and auto wakeup sequences may be employed, as well.

1. **Asynchronous Serial Port Setup**

In the basic mode, a three-step set-up sequence is all that is necessary. First, the port must be enabled for transmitting and receiving. Second, the number of data/stop/parity bits chosen - the standard RS-232 parameters. Finally, the baud rate must be selected. The chip is extremely flexible in allowing the selection of all standard rates, as well as custom rates. Then, this initialization struct (the aforementioned UART\_HandleTypeDef) is passed to the function HAL\_UART\_Init(), which actually sets up the UART. Note that HAL\_UART\_Init() calls back to HAL\_UART\_MspInit(), which initializes the ports and pins of the device for reading and writing to/from the UART communications channel. You can see the full HAL\_UART\_Init() function in the file “stm32f7xx\_hal\_uart.c” in the stm32lib folder.

Transmitting and receiving data is a relatively simple procedure requiring HAL\_UART\_Transmit and HAL\_UART\_Recieve. The setup used for the USB serial connection can be seen in uart.c/uart.h; it provides a good example of how a simple UART instance is configured using an STM32.

*Note regarding HALs & uart.c:*

Worth driving home is that HALs aim to make it easier to configure complex hardware like modern chips, and to also allow code written for one chip to work on others. Using some kind of HAL is how the majority of microprocessors are programmed for these days, which is why uart.c does not have any raw register manipulations. Also STM32 does not officially support NOT using the HAL and its functions—hello.c in Lab 1 was created to show what the HAL actually does, and Lab 2 was to get a feel for low-level operations. With the complexity of this chip, not using the HAL for “bigger” protocols like USB and Ethernet would be extremely tedious (but nonetheless possible for the masochistic).

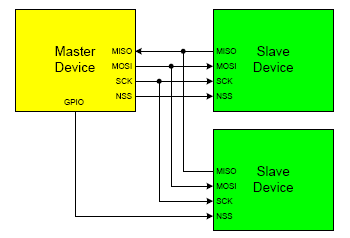
There is also an even higher level of abstraction, “BSP” (Board Support Package), which abstracts away STM32 board-specific functions. For example, BSP\_LED\_TOGGLE(LED1) is specific to the STM32F769I-DISCO to toggle the LED named “LED1.” For further discussion and a full low-level breakdown of BSP, HAL, & CMSIS[[1]](#footnote-1) see hello.c from Lab 1.

In this course, we will be concerning ourselves primarily with the HAL layer, but if you want to go lower and use CMSIS-level code, go for it. Please avoid using BSP functions, however, since we cannot accurately assess whether you know what is going on or not when they are used. (*Hint: for every BSP function, there is a way to do it with the HAL and with CMSIS!*)

**SYNCHRONOUS SERIAL PORT**

1. **Synchronous Serial Port Interface (SPI) Setup**

Synchronous serial communication between processors is possible using a Synchronous Serial Peripheral Interface (SPI) port on the 769 chip. Synchronous, or separately clocked, serial connections can communicate at much higher rates than standard RS-232 data rates. They also use master/slave configurations between devices where the single master provides the clocking signal to all slave devices. The figure below shows the signals between two SPI devices. The figure also demonstrates the mechanism where, as data from one device is clocked out of its shift register, data from a second device is simultaneously clocked into the register.



SPI is designed as a short-range, on-board bus specification for distances of less than a foot. It may be used between separate systems for higher speed communication when the wires are kept as short as possible, as longer wires may limit the maximum speed. Synchronous serial devices are not as well standardized as asynchronous RS-232 devices, and are therefore less common. To get around the lack of extra synchronous devices, this exercise will initially use a single STM and have it communicate with itself through a loop-back connection from MOSI to MISO on the protoboard bus. Next, a 68HC11 EVB will be configured as a compatible SPI slave device with which the STM can communicate.

In the four-wire SPI mode, a character written to the MOSI data register will be transmitted back into the processor's data register, but first the processor must enable the slave by clearing NSS. Note that the same register is used for transmitting and receiving. Upon completion of transmission NSS must be released (set to 1) to allow the slave to write to its data register. The master should pause briefly after each transmission (increment an integer to 100) permitting the slower slave to read the register and write new data while NSS is high—otherwise the slave will not be able to reply with its own data byte.

1. **Steps for SPI Transmission**
2. Configure SPI clock rate, wire mode (duplex), polarity, and sampling edge (phase). Set the STM as master. Enable SPI, set the relevant pins (as push-pull), and disable CRC. Poll the registers.
3. For a READ
   * 1. Wait for RX to be enabled
     2. Read bytes from data register
     3. Set the TXFIFO\_THRESHOLD to indicate 8bit instead of 16bit
     4. Make sure FRLVL is empty and the status register is not BSY (busy)
4. For a WRITE
5. Enable slave select (on configured GPIO pin)
6. Wait for transmit to be enabled
7. Set the SPI Data register (it’s 16 bit so be careful when writing your last byte!)
8. Make sure FTLVL is empty and the status register is not BSY

**PROGRAMMING TASKS**

**PART I**

The first programming assignment is to write a procedure that will have the program monitor two serial ports continuously. The DISCO board comes with 5 onboard serial ports, of which we will mainly be using two: USART1 and USART6.[[2]](#footnote-2) USART1 has been routed over USB via “uart.c/uart.h.” In order to add a second serial port connected to USART6, a USB-Serial adapter needs to be obtained from the lab. These adapters have three relevant pins: TX, RX , and GND. Connect the green wire (TX) on the serial adapter to the RX pin of USART6 (D0). Connect the white wire (RX) on the serial adapter into the output of an inverter, and connect the inverter input to the TX pin of USART6 (D1). Connect the black wire (Ground) to a ground on the board (marked GND). Also connect the Ground for USART6 to the other grounds (marked GND). The red wire (VCC) does not need to be connected. An image of the USB-Serial adapter is included below.



**USB-Serial Converter**

Whenever the program detects a character coming in from either of the onboard serial ports, it should echo it back to both serial ports. On your PC you will have two terminal windows open using either SecureCRT or PuTTY (Windows) or Picocom or Screen (Mac, Linux). Different COM numbers are assigned to the USB-mapped communication ports, depending on when the adapters were plugged into the PC. You must check the currently assigned port so that both terminal windows can be configured to talk to the proper ports.

As a reminder, in Windows, this is done by right clicking on the *My Computer* iconand selecting the *Manage* menu item. Select *Device Manager* in the window and scroll down to *Ports (COM & LPT)*. Expand the list if necessary and note which port number has been assigned to the USB to serial adapters.

On Macs, the device is simply “/dev/tty.usbmodem[something]” or “/dev/tty.usbserial[something].”

In Linux the device is “/dev/ttyUSB[something].”

For convenience, we will always be using the onboard serial port (USART1) with 115200 baud and N-8-1 (no parity, 8 data bits, 1 stop bit).

USART6 must be configured for 9600 baud and N-8-1. You will need to configure it manually as well as send and receive characters using the HAL initialization functions as described above. Remember you must match the terminal program’s setup parameters to the port's configuration, otherwise you will either get nothing or garbage output. Note that COMn and is created by using a USB to Serial converter and software that allows the USB port and hardware to emulate a second serial port on the PC.

Write a program to poll both ports continuously and then echo any character received to both ports so the received character will show up on both displays. An <ESC> key pressed on the onboard serial port (USART1) should display a brief message on both screens and halt the program. Note that getchar() cannot be used for polling because the function will just wait for USART1 to receive a character.

*Part 1 Helpful Hints:*

1. Test your hardware wiring first. If USART1 is disabled all functions using it must be disabled to keep your program from locking up. Remember to set the terminal baud rate correctly for each UART.
2. If the incorrect baud rate is configured, the terminal will not output characters or the terminal will output seemingly random characters.
3. If GND, RXD, TXD are disconnected or if RXD and TXD are switched on the DB-9, the terminal will not output characters.
4. If numerous random characters are repeatedly sent and echoed to the terminals, ensure that the grounds on the UART and the protoboard are tied together.
5. The code does not need to be downloaded every time after halting; only when software changes are made. The reset button on the board will restart the program.
6. Now may be a good time to look inside those “uart.c/.h” and “init.c/.h” files if you haven’t yet…

**PART II**

This program duplicates the functionality of the program in Part I but allows the serial ports to generate interrupts when characters are received and has ISRs handle the job of echoing the received characters to the two displays. This is done using interrupts. Instead of using HAL\_UART\_Receive, as you did for the first part of this lab, you will now be using HAL\_UART\_Receive\_IT to receive in a non-blocking, interrupt-based way. HAL\_UART\_Receive\_IT frees the main process to do other things while it runs, and only alerts your code (via callback) when it has received data. As a result, HAL\_UART\_Receive\_IT does not require a timeout parameter, but syntax is otherwise identical to HAL\_UART\_Receive.

The ISR USART1\_IRQHandler is assigned to USART1 (USB) and USART6\_IRQHandler to USART6. Each of these two ISRs should call HAL\_UART\_IRQHandler(UART\_HandleTypeDef \*huart) with a pointer to the appropriate UART (either USB\_UART or the other UART) as a parameter. Note that each ISR is shared by several causes on the port including a receive or transmit, and these details are handled by HAL\_UART\_IRQHandler() (for more information on the details of how this works, see chapter 8.4 in *Mastering STM32*). This means that HAL\_UART\_IRQHandler will generate calls to several different callback functions, each of which has a different functionality. The main objective of this exercise is to handle the receive callback, HAL\_UART\_RxCpltCallback(UART\_HandleTypeDef \*UartHandle), but you may choose to write code to handle other cases, including framing and parity errors.

The receive callback, HAL\_UART\_RxCpltCallback, is called whenever a receive operation is complete (i.e. whenever the number of bytes specified to HAL\_UART\_Receive\_IT have been received). This callback receives a handle which points to the UART configuration that generated the interrupt, and can be used to determine the UART from which the character originated. Otherwise, this callback acts just like a normal ISR that would trigger upon the reception of data.

When you have completed the program and verified its operation, you will need to find another group with a working version of Part II and connect your two serial ports together via the Arduino TX & RX signals and a common ground. Remember transmit on one processor must be connected to receive on the other processor. Now any character typed on the USB terminal on either processor will show up on the other processor's terminal.

*Part 2 Helpful Hint:*

1. Use the following code to handle the receive interrupts:

// Handle USB/UART Interrupts with HAL

void USART1\_IRQHandler(void) {

HAL\_UART\_IRQHandler(&USB\_UART);

}

void USART6\_IRQHandler(void) {

HAL\_UART\_IRQHandler(&Second\_UART);

}

1. You can use HAL\_NVIC\_EnableIRQ to enable interrupts. Also, don’t forget HAL\_UART\_RxCpltCallback!

**PART III**

Write a simple C program to set up and communicate through the STM’s SPI (**please use SPI2**). Initially the port will only talk to itself through a wire connecting MOSI to MISO. Any character transmitted will be the same character received while the loop-back wire is present. If the input to MISO is held to ground or +3.3 volts,[[3]](#footnote-3) the input character should be 0x00 or 0xFF respectively after a transmission is received (*Why?*). Use your program to verify all three cases.

Write your program so that it uses ANSI escape sequences to split the terminal window in half with the top half used to display characters typed locally at the keyboard to be sent out the SPI port and the bottom half to display characters received by the SPI serial port. It is not necessary to scroll the top half and bottom half separately as the number of lines hits the limit of half the screen, but doing so will enhance your grade.

After verifying operation with the simple loop-back connection, the next part of the procedure is to incorporate a second EVB, a 68HC11, as a slave SPI device. A program called SPISLAVE is available for your use on the course web page to run on the HC11, allowing it to send and receive synchronous serial data in slave mode. The 68HC11 EVB with the provided software will permit you to verify your program with more interesting test strings under more realistic operating conditions.

When running with the loop-back wire, SPI clock frequencies of 1 to 4 MHz should work, but when connecting to the slower HC11 EVB through the EVB bus protection circuits to the protoboard, these may need to be reduced to 1 MHz (the HC11’s maximum speed) or less. The combination of loose unshielded wires, ribbon cables, and bus buffers in both directions make this change necessary. Additionally, with the direct loop-back wire transmitted data will be received instantly, while when going through a slave device, the reply will be delayed by one character transmission due to the added second shift register in the slave. Your software will need to accommodate this circumstance.

SPISLAVE.C is a program for the HC11 EVB that will configure its synchronous SPI serial port to run as a slave device. The S19 file is available on the course website under Course Handouts and may also be in the HC11 directory under CStudio on the studio PCs. This program is designed to accept characters clocked in from an SPI master device on one transfer and echo back the same character on the following transfer. The slave will display the transmitted character on its console when it has been received. This can be used as an aid in debugging. An additional feature on the slave may be implemented in the downloaded S19 version. This function will transmit a message from the slave to the master when the master sends a <DEL> (Backspace) character (0x7F). Upon receipt of a <DEL>, the slave still echoes it back on the following transfer as with all characters, but follows this with a sequence of printable ASCII characters in a string sent to the master ending with a 0xFF character (about 100 character total). The master must transmit dummy characters to the slave until the 0xFF is received in order to obtain the entire message.

Use the following commands to download the program to the HC11 EVB through a ProComm Plus connection directly to the serial port (0) at 9600 baud on the EVB. Make sure the ProComm terminal is in **ANSI BBS** mode and the Xfer protocol is **ASCII** or **RAW ASCII** (lower left corner, 2nd line up).

Hit <Enter> once or twice

L T<Enter> (click on the **Send File** icon at the top and then select the SPISLAVE.S19 file)

G 6000<Enter> (begin execution)

If HyperTerminal is used instead of ProComm Plus, set **Connect using:** to COM1, and under **Configure…** make sure to use 9600 Bits per second, 8 Data bits, no Parity (None), 1 Stop bit, and Xon/Xoff for Flow control. Then after the L T<Enter> command, select **Transfer → Send Text File…** to download the desired S19 file.

If you are feeling adventurous and want to try using a Unix system (namely a Mac or Linux machine) to connect and send the files, use picocom in Terminal:

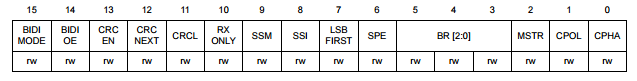
picocom /dev/tty[serial device] -b [baudrate] --receive-cmd "ascii-xfr -r -v" --send-cmd "ascii-xfr -s -v"

From there, CTRL + A then CTRL + S sends text files. Only use this if you feel comfortable with Unix command line and understand that the TAs and professor might not be able to help with issues from using this method!

When SPISLAVE is executed on the HC11 EVB, it displays on its console hardware configuration data for wiring up the HC11 to an 8051. You will need to identify the correct pins on the STM32 (*Hint: The SPI pin names—MOSI, MISO, NSS, SCK—are the same*). If SPISLAVE does not seem to be functioning correctly, try restarting it (reloading it is not necessary after a simple reset) **after** starting your SPI master program on the STM32.

***DETAILS (STM32)***

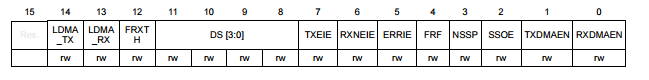
**SPI CONTROL REGISTER 1 (SPIx\_CR1)**



|  |  |
| --- | --- |
| **REGISTER** | **DESCRIPTION** |
| BIDIMODE | Bidirectional data mode enable.  0: 2-line unidirectional mode selection  1: 1-line bidirectional mode selected |
| BIDIOE | Output enable in bidirectional mode.  0: Output disabled (receive only mode)  1: Output enabled (transmit only mode) |
| CRCEN | Hardware CRC Calculation Enable, 1: Enabled |
| CRCNEXT | Transmit CRC Next  0: Next transmit value is from Tx Buffer  1: Next transmit value is from Tx CRC register |
| CRCL | CRC Length  0: 8-bit CRC Length  1: 16-bit CRC Length |
| RXONLY | Receive only mode enabled.  0: Full duplex (Transmit and Receive)  1: Output disabled (receive-only mode) |
| SSM | Software Slave Management  0: Software slave management disabled  1: Software slave management enabled |
| SSI | Internal Slave Select  0:Software slave management disabled  1: Software slave management enabled |
| LSBFIRST | Frame format.  0: Data is transmitted/received with the MSB First  1: Data is transmitted/received with the LSB First |
| SPE | SPI Enable, 1: Enabled |
| BR[2:0] | Baud rate control (see SPI manual) |
| MSTR | Master Selection  0: Slave Configuration  1: Master Configuration |
| CPOL | Clock Polarity  0: CK to 0 when idle  1: CK to 1 when idle |
| CPHA | Clock Phase  0: First clock transition is the first data capture edge  1: The second clock transition is the first data capture edge |

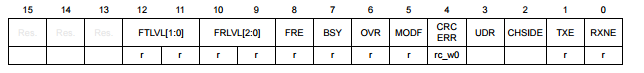
***See Ch. 35 in* 769 Reference Manual (Register Map).pdf *for full list of registers***

**SPI CONTROL REGISTER 2 (SPIx\_CR2)**



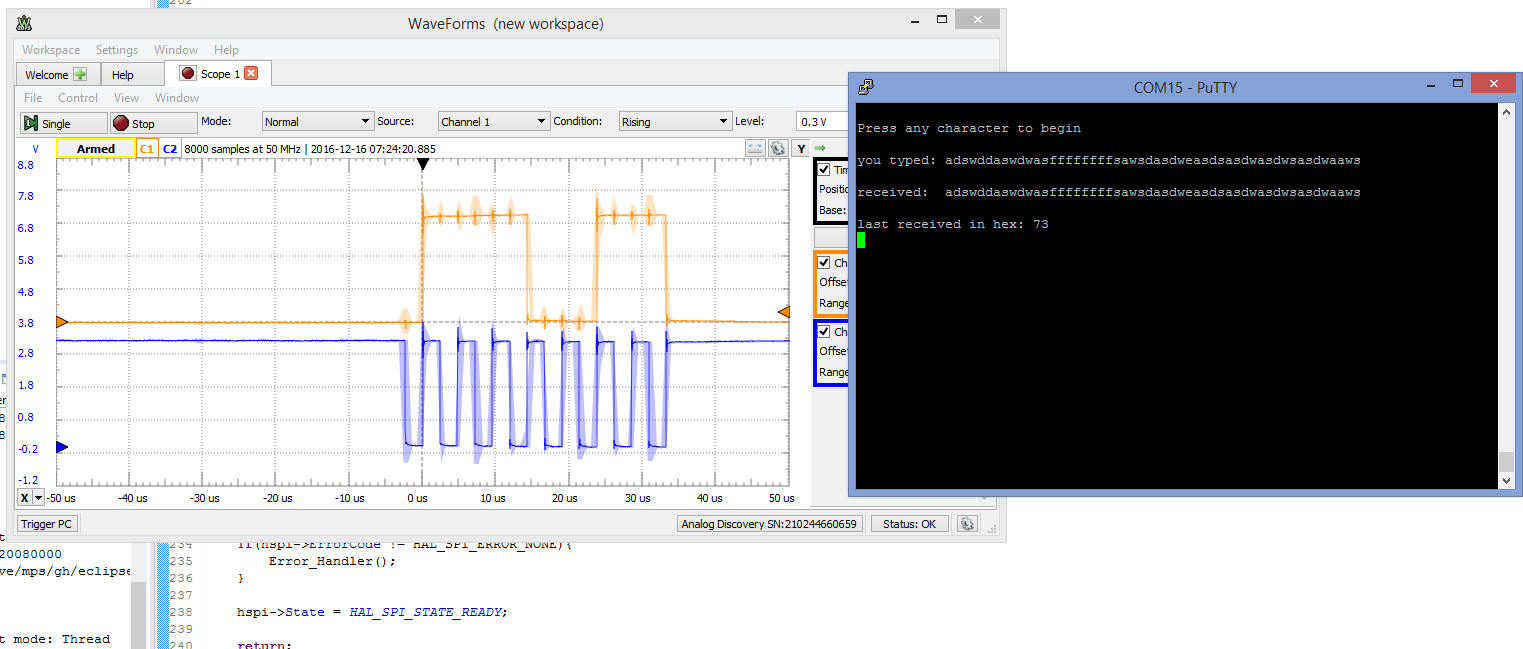
***See Ch. 35 in* 769 Reference Manual (Register Map).pdf *for full list of registers***

**SPI STATUS REGISTER (SPIx\_SR)**



|  |  |
| --- | --- |
| **REGISTER** | **DESCRIPTION** |
| FTLVL[1:0] | FIFO Transmission Level  00: FIFO Empty  01: ¼ FIFO  10: ½ FIFO  11: FIFO Full |
| FRLVL[2:0] | FIFO Reception Level |
| BSY | 0: SPI not busy  1: SPI is busy in communication or Tx buffer is not empty |
| TXE | Transmit Buffer Empty |
| RXNE | Receive Buffer Not Empty |

***See Ch. 35 in* 769 Reference Manual (Register Map).pdf *for full list of registers***



**Example SPI Output on Oscilloscope and Terminal**

***DETAILS (68HC11 EVB)***

The source code for SPISLAVE is available on the course web page. For those interested in how it works, here is a little bit of background information on how the registers are used to configure the SPI operations.

The 68HC11 EVBs in the studio are configured with an SPI port similar to that on the STM32, on a register level. It is an older design with a few less features, but can be used as a device with which to communicate through a synchronous master/slave serial configuration. The HC11 has three registers associated with its SPI port. They are:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |  |
| $1028 | SPIE | SPE | DWOM | MSTR | CPOL | CPHA | SPR1 | SPR0 | SPCR |
| RESET = | 0 | 0 | 0 | 0 | 0 | 1 | U | U |  |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |  |
| $102A | SPD7 | SPD6 | SPD5 | SPD4 | SPD3 | SPD2 | SPD1 | SPD0 | SPDR |
|  |  | (Double | buffered | in, | single | buffered | out ) |  |  |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |  |
| $1029 | SPIF | WCOL |  | MODF |  |  |  |  | SPSR |
| RESET = | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |  |

The data register, SPDR, is the same as the STM32 data register, SPIx\_DR (well, smaller). The status register, SPSR, and the control register, SPCR, provide similar configuration, control and status bits corresponding to the STM32 registers SPIx\_SR, SPIx\_CR1, and SPIx\_CR2. The HC11 SPI has far fewer clock rates available than the STM32, and even the old C8051. Bits 1 & 0 of SPCR are used to set the clock rate using a scheme show below, with the HC11’s 2 MHz E clock divided down by a scale factor set by SPR1 and SPR0 using the following table.

|  |  |  |
| --- | --- | --- |
| SPR1 | SPR0 | E-Clock Divide-by |
| 0  0  1  1 | 0  1  0  1 | 2  4  16  32 |

Bits 2 through 7 of the SPCR control register correspond to the following bits: CPHA (clock phase), CPOL (clock polarity), MSTR (master enable), DWOM (Wired Or Mode – set output to push/pull or open drain in another SFR), SPE (SPI enable/disable feature to save power), and SPIE (SPI Interrupt Enable). The 3 bits in the SPSR register are SPIF (SPI transfer complete flag), WCOL (write collision), & MODF (mode fault). After configuration, SPIF is the main bit checked during operation to see when transmission is complete and the SPI is ready to accept the next byte to transmit. Note that the only shift mode on the 68HC11 is MSB first, unlike the STM32’s option to use MSB or LSB first.

The short C program SPISLAVE written for the 68HC11 can read in characters sent to the SPI port and echo them to both the terminal and back out the SPI to the source. Here again, the terminal would be a second HyperTerminal or ProComm window on the PC communicating with the 68HC11 EVB through COMn: (or /dev/[whatever] in Unix) with the standard 9600 8-N-1 configuration. When a <DEL> code ($7F) is received, the program should send the complete set of printable ASCII characters from <space> ($20) to ‘~’ ($7E) followed by a <RETURN> ($0D), <LINE FEED> ($0A), <BEL> ($07) and ($FF).

There is an important item to note about the SPI implementation on the EVBs: The SPI slave device is not able to load any characters into its SPI data register unless the master releases the  line. The SPISLAVE.C program will only be able to send the ASCII string if the master clears  just prior to loading its data register (to initiate a transmission), waits for the SPIF flag in the status register to go high, and then sets  again. Reading the data register clears the status register SPIF in preparation for another transmission and gets the received byte. A short delay in the master program (incrementing an integer to 100) after reading the data register will give the slower slave time to catch up, resynchronize and write data to its own data register in preparation for the transfer (swapping) of the next byte. Keeping this in mind when writing the master program will permit successful data transmission to the slave and vice versa. Though, the STM32 does have some functionality up its sleeve to make this easier… :)

**Possible lab enhancements:**

Add LED indicators. All 4 GPIO-connected LEDs are user-accessible via HAL & CMSIS.

Add automatic parity checking to UART communications

Try to use Framing Errors to automatically figure out baud rate and adjust it

Connect up a 2nd 68HC11 SPI slave device and use a port output pin as its NSS signal;   
 connect the two slaves in either a daisy-chain or in parallel (use NSS in parallel to disable one or   
 the other slaves and prevent both outputs from driving each other)

Determine the maximum SPI clock frequency for reliable communication with the 68HC11 slave or   
 the maximum baud rate for UART communication with the ProComm terminal

Other possibilities…

1. CMSIS (Cortex Microcontroller Software Interface Standard) is a lower-level general ARM abstraction layer that is consistent across ARM chips, not just STM32 varieties. This is actually what Labs 1 & 2 were using, and it is basically a standardized naming convention for ARM registers. Learn CMSIS and you can program *any* Cortex-M MCU!

   For context, Cortex-M devices include iPhone Motion Coprocessors, JTAG debuggers, drones, and other embedded platforms. The Cortex-A series, of which Cortex-M is a subset, contains the CPUs in most smartphones, portable game consoles (e.g. all portable/hybrid Nintendo devices ever made), Wi-Fi/Bluetooth peripherals, car computers, and very many more devices. [↑](#footnote-ref-1)
2. The STM32F769NI chip itself, however, supports up to 8 (4x USART & 4x UART), which is bonkers. On the DISCO, we can only access USART1, UART4, UART5, USART6, & UART7 through clever usage of alternate functions on the Arduino pins. In a similar fashion, we can access 2 of the chip’s 6 SPI buses (SPI2 & SPI5) via alternate functions on the Arduino pins. [↑](#footnote-ref-2)
3. For the sake of having it written somewhere: All of the digital Arduino pins (D0-D15) are 5V tolerant. All the analog pins, EXCEPT A1, can be operated as digital pins with 5V tolerant inputs, but they default to analog mode on reset. **When in analog mode, all analog pins (A0-A5) are 3.3V only**. A1 is 3.3V no matter what, as it is connected directly to ADC1. That stated, please only use 3.3V max, as that is what the DISCO board is primarily rated for (and it will resist wiring accidents!). [↑](#footnote-ref-3)