MPSD

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)	POINTS (2)	TA init.
Grade for performance verification (50% max.)			
Grade for answers to TA's questions (20% max.)			
Grade for documentation and appearance (30% 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

- Read Sections 11.1 to 11.4 from *Software and Hardware Engineering* by Cady & Sibigtroth.
- Write a C program that is free from syntax errors (i.e., it should assemble without error messages and produce a .S19 file).

1. INTRODUCTION TO THE ASYNCHRONOUS SERIAL COMMUNICATIONS INTERFACE (SCI)

The 68HC12 has two on-board asynchronous serial communications interfaces, SCI0 and SCI1, and an additional synchronous serial peripheral interface, SPI. With an additional external chip provided on the EVB to permit the required ±12 volt swing, these ports may be configured as standard RS-232 serial ports. As a universal asynchronous receiver-transmitter (UART), each port has 10 registers for status, control and data transfers. In the basic polled configuration without interrupts, a subset of 6 registers is used. These are (the 'n' appearing in the register names is either a '0' or a '1' depending on the port referenced and the _H12… name is that used by the DBUG.H file):

SCnDRL (_H12SCnDRL): Data Register Low - character sent or received {A Data Register High is available for serial configurations using an uncommon character length of 9 bits.}

SCnCR2 (_H12SCnCR2): Control Register 2 - TE & RE enable transmit & receive

SCnCR1 (_H12SCnCR1): Control Register 1 - advanced features as well as parity, bit length, and stop bits setting

SCnBDH & SCnBDL (_H12SCnBD): SCIn Baud Rate Control - a double register for setting the baud rate {**NOTE: The Introl compiler only recognizes a single 2-byte wide register named _H12SCnBD.**}

SCnSR1 (_H12SCnSR1): SCIn Status Register 1 - TDRE & RDRF indicate data has been transmitted or data has been received

In more sophisticated configurations, interrupts may be generated when data is received or ready to be transmitted, transmission, framing or overrun errors detected, self-checking modes may be configured,

and auto wakeup sequences employed.

2. ASYNCHRONOUS SERIAL PORT SETUP

Chapter 11 of the text has a very detailed explanation of the SCI 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 parity mode must be selected, the number of data bits chosen, and whether 1 or 2 stop bits are to be used - 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. Table 11-2 in the text show what value to use in the double register SCnBDH:SCnBDL for the desired system baud rate. Remember that the 68HC12 EVBs have an M-clock of 8 MHz and setting must be taken from that column.

The text has Example 11-1 at the end of section 11.3 showing a working configuration. Although it is in assembler, it is easy to understand how the registers must be configured and used to set up, transmit, and receive data. When all else fails, you can always use D-Bug12 to look at (but not change) the registers for SCI0, the working port used by the monitor program to communicate with the user terminal.

Transmitting a character through a port involves a simple two step procedure. First the Transmit Data Register Empty Flag must be checked to see that it is set. If not, the program must wait and keep checking until the bit is high. Then the data to be transmitted must be loaded into the Data Register. Receiving data is a similar two step process. The Receive Data Register Full Flag must be checked. If it is cleared, this indicates that no data is available and the program may go on to something else or decide to wait for something to appear by repeatedly checking the flag. A set flag indicates that data is available and may be read from the Data Register. Note that the same data register is used for transmitting as well as receiving data.

3. SYNCHRONOUS SERIAL PORT INTERFACE (SPI) SETUP

Synchronous serial communication between processors is possible using the Synchronous Serial Peripheral Interface (SPI) port on the 68HC12 EVB. 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. Figure 11-6 in the text shows the signals between two SPI devices and also demonstrates the mechanism where, as data from one device is clocked out of its shift register, data from the second device is simultaneously clocked into the register. The SPI on the 68HC12 is capable of rates up to 4 Mbits/sec. Although 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. Longer wires may limit the maximum speed. Synchronous serial devices are not as well standardized as asynchronous RS-232 and are therefore less common. To get around the lack of

extra synchronous devices, this exercise will initially use a single 68HC12 and have it communicate with itself through a loop-back connection from the MOSI to the MISO on the protoboard bus. Next, a 68HC11 EVB will be configured as a compatible SPI slave device with which the 68HC12 can communicate.

Using the SP0CR1 (_H12SP0CR1) register, configure the SPI port for polled use without interrupts, SPI bits 4-7 for dedicated operation, no wired-OR, master mode, shift clock low when not shifting, serial data sampled on the rising edge of SCK, slave select output, and most significant bit first. Configure the SP0CR2 (_H12SP0CR2) register for active pull-up, no reduced drive capability, and normal two-wire mode. Select a clock frequency of 1 MHz with the SP0BR (_H12SP0BR) baud rate register. Finally give the SPI control over the output lines by setting the Port S data direction bits in DDRS to $E0.

A character written to the SP0DR (_H12SP0DR) data register will be transmitted back into the processor's data register, but first the processor must enable the slave by clearing $\overline{SS}$ (PS7 in PORTS). The SPIF flag in the SP0SR (_H12SP0SR) status register needs to be read to indicate when a character has been received in the data register. Note again that the same register is used for transmitting and receiving. Upon completion of transmission $\overline{SS}$ 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 $\overline{SS}$ is high otherwise the slave will not be able to reply with its own data byte.

4. PROGRAMMING ASSIGNMENT

PART I

The first programming assignment is to write a procedure that will have the EVB monitor both SCI ports continuously. Whenever it detects a character coming in one port, it should echo it back out that port and also send it out the other port. On your PC you will have two terminal windows open using either ProComm Plus or HyperTerminal. One window will communicate through COM1: and the other through COMn:. (Any PC serial port COM3: through COM7: may be used. Different values are assigned to the USB-mapped communication ports, depending on when the adapter was plugged into the PC. You must check the currently assigned value so that the second terminal window can be configured to talk to the proper port. This is done by right clicking on the *My Computer* icon and selecting the *Properties* menu item. Select *Device Manager* in the window and scroll down to the *Universal serial bus controller*. Expand the list if necessary and note which port number has been assigned to the USB to serial adapter. Alternatively, in ProComm Plus, when the correct active port has been selected, the transmit/receive indicators at the bottom right of the window will turn red.) COM1: will be connected to SCI0 (the normal monitor port) and COMn: to SCI1. For convenience, leave SCI0 in its default configuration of 9600 baud and N-8-1 (no parity, 8 data bits, 1 stop bit). You may also use the built-in D-Bug12 functions DB12->putchar() and DB12->getchar() to communicate with the terminal on COM1:. To check to see if

a character is available on SCI0 without locking the program into a state where it will wait for a character requires the interrogation of the RDRF flag in SC0SR1. getchar() will put the program in a loop that can't be broken until a character is received.

SCI1 will be configured for 28800 baud and N-8-1. You will need to configure it manually as well as send and receive characters using the status, control, and data registers. Remember you must match the terminal programs setup parameters to the port's configuration. Note that the PC cards in the Sun Ultra10s support only COM1: directly. COMn: 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 either terminal should display a brief message on both screens and halt the program.

PART II

This program duplicates the functionality of the program in Part I but allows the SCI ports to generate interrupts when characters are received and has ISRs handle the job of echoing the received characters to the two displays. Interrupt 10 is assigned to SCI1 and 11 to SCI0. The assignment of interrupts to ISRs follows the same convention as was used in the Interrupts and Timer ISRs lab exercise. Note that each interrupt (10 or 11) is shared by five possible causes on the port. See Table 11-4 in the text for the list of causes. This means that the ISRs must interrogate the SCnSR1 status register to find the particular cause of a generated interrupt and take appropriate action for each separate cause. The main objective of this exercise is to handle the Receive Data Register Full case but you may choose to write code to handle the other four cases.

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 SCI1 ports together using the 2 bus signals TxD1, RxD1 and a ground reference. Remember transmit on one processor must be connected to receive on the other processor. Now any character typed on the SCI0 terminal on either processor will show up on the other processor's terminal. Instead of using a TxD/RxD switching (crossing over) RS-232 cable, a set of wires connecting PS2, PS3, and GROUND on J9 between the two EVBs may be used.

PART III

Write a simple C program to set up and communicate through the 68HC12's SPI. 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 +5 volts, the input character should be $00 or $FF 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, with a program to send and receive synchronous serial data provided for your use. The HC11 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 buffer 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 class WebCT site 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> character ($7F). 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 $FF character (about 100 character total). The master must dummy characters to the slave until the $FF is received in order to obtain the entire message.

When SPISLAVE is executed on the HC11 EVB (L T <SPISLAVE.S19>; `G 6000`) it displays on the console hardware configuration data for wiring up the HC11 to the HC12 EVB. 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 HC12.

DETAILS: The 68HC11 EVBs in the studio are configured with an SPI port similar to that on the 68HC12 EVBs. 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. 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, and the status register, SPSR, are identical to their corresponding SP0DR and SP0SR registers on the 68HC12. Bits 2 through 7 of the control register also correspond directly to the same bits in SP0CR1 on the 68HC12. The last 2 SPCR bits are used to set the clock rate using a similar scheme as on the 68HC12, but using the 2 MHz E clock divided down by a scale factor set by SPR1 and SPR0 using the following table. Note the only shift mode on the 68HC11 is MSB first.

SPR1	SPR0	
0	0	2
0	1	4
1	0	16
1	1	32

The short C program SPISLAVE written on 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: 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 $\overline{SS}$ line (PORTS bit 7). The SPISLAVE.C program will only be able to send the ASCII string if the master clears $\overline{SS}$ 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 $\overline{SS}$ 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 mater program (incrementing an integer to 100) after reading the data register will give the slower slave time to catch up, resynchronize and write date to its own data register in preparation for the transfer (swapping) of the next byte. Following these 6 steps in the master program will permit successful data transmission to the slave and vice versa.

```
// SPISLAVE.C
// RPK
// September 25, 2001
// May 23, 2003 (revised)
//
// Compile, link & hex convert: CC6811 SPISLAVE
//
// Synchronous Serial Port slave device for the 68HC11 EVB
// for use as a test device for a Synchronous Serial Port Master
// implemented on the 68HC12 EVB.
//
// Hardware connections:
// 68HC11 BUS        68HC12 BUS
// MISO (pin 22) output TO MISO (J9 pin 43) input
// MOSI (pin 23) input  TO MOSI (J9 pin 44) output
// SCK  (pin 24) input  TO SCK  (J9 pin 41) output
// SS*  (pin 25) input  TO SS*  (J9 pin 42) output
// VSS  (pin  1) (GND)  TO VSS  (J9 pin 59) (GND)


#include<HC11A1.h>
#include<introl.h>
#include<stdio.h>
#include<stdlib.h>


void main()
{
   char char0, char1;
   unsigned char l;

   _H11DDRD = 0x04;     //0x3C or 0x04
   _H11SPCR = 0x40;     //0x40 for slave mode, 0x50 for master mode
   printf("\033[2J");
// for(l=0;l<28;l++)printf("\n")

   printf("\nHC11 SPI Slave mode initialization: %x %x \n\r", &_H11SPCR, _H11SPCR);
   printf("   Hardware connections:\r");
   printf("68HC11 BUS              68HC12 BUS\r");
   printf("MISO (pin 22) output TO MISO (J9 pin 43) input\r");
   printf("MOSI (pin 23) input  TO MOSI (J9 pin 44) output\r");
   printf("SCK  (pin 24) input  TO SCK  (J9 pin 41) output\r");
   printf("SS*  (pin 25) input  TO SS*  (J9 pin 42) output\r");
   printf("VSS  (pin  1) (GND)  TO VSS  (J9 pin 59) (GND)\r");

   printf("\nThis program echoes characters received on the SPI port\r");
   printf("to the 68HC11 console and back to the sender through the SPI\r");
   printf("on the next transmission cycle from the master device.\r\n");
   printf("Restart this program after the SPI master if encountering problems.\r");

   while(1)
   {
      while(_H11SPSR < 0x80);    //wait for character
      char0 = _H11SPDR;          //read character
      _H11SPDR = char0;          //echo character back to sender
// This doesn't do anything if SS* is not released
      putchar(char0);            //print on console
      while(_H11SPSR < 0x80);    //wait for transfer
      char1 = _H11SPDR;          //dummy read
//    printf(" '%c'  %x(hex) received\n\r",char0,char0);
      if(char0 == '\177')        //if <del> is received, then
      {
```

```
// A stumbling point here is that the slave can't load the data register
// for sending out data unless the master releases the SS* output line tied
// to the slave's SS* input. Otherwise any value that is loaded is ignored.
// This prevents the transfer of new data from the slave to the master.
// Only the previous byte already there is transferred back to the master.
         for(l = 32;l<127;l++)
         {
            _H11SPDR = l;
//          putchar(l);
            while(_H11SPSR < 0x80);
            char0 = _H11SPDR;    //read character
         }

         _H11SPDR = '\r';        // \r <return>
//       putchar('\r');
         while(_H11SPSR < 0x80);
         char1 = _H11SPDR;       //dummy read
         _H11SPDR = '\n';        // \n <new line>
//       putchar('\n');
         while(_H11SPSR < 0x80);
         char1 = _H11SPDR;       //dummy read
         _H11SPDR = '\007';      // \007 <bel>
//       putchar('\007');
         while(_H11SPSR < 0x80);
         char1 = _H11SPDR;       //dummy read
         _H11SPDR = '\377';      // \377 <rubout>
         while(_H11SPSR < 0x80);
         char1 = _H11SPDR;       //dummy read

      }
   }
}
```