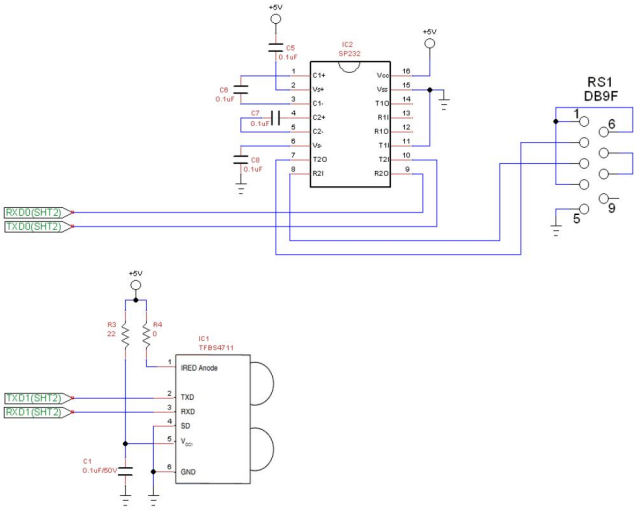
The Serial Communications Interface (SCI)

Your 9S12 micro contains SCI modules for asynchronous serial communications. You will use one of the SCI modules to communicate with the PC, although the SCI module may communicate with any other compatible UART (Universal asynchronous receiver-transmitter). If you are directly interfacing to another UART, you will likely not use RS-232, but TTL levels instead.

In order to connect to a PC the TTL-level signals (0-5V) from the SCI must be converted to RS-232 levels (~±10V). Your board contains a chip dedicated to this task, and the signals are brought to a standard RS-232 DB9 connector on the top edge of your board. RS-232 permits much longer distances between devices and has *some* resistance to signal interference. NOTE: The use of RS-232 does not change the timing of the UART signal, it just uses a different signaling scheme.

Your board has an IrDA transceiver as well on `SCI1`. We may not get to use this port in this course, but IrDA is good for ~1m and uses infrared light as the physical layer. Using infrared light provides *extreme* electrical isolation between the two communicating devices.



While RS-232 supports additional signaling options, we will be using only three wires: ground, transmit data, and receive data.

In asynchronous communications, the transmitter may begin a data send operation to the receiver at any time. Once started, a complete block of data (known as a data character) must be completely transmitted. The delay between data characters may be any length. Transmission of the individual bits in the data character is driven by a local clock. The transmitter and receiver must operate independent clocks that are approximately equal in rate to correctly exchange data. The term 'asynchronous' refers to the fact that the clocks on the two devices are independent (there is no synchronizing clock signal), and communication can be initiated at any time.

Because we are reading and writing bytes in serial communications, the SCI module acts as a parallel-to-serial and serial-to-parallel converter.

The RS-232 protocol allows for a wide range of signaling characteristics. Here are a few:

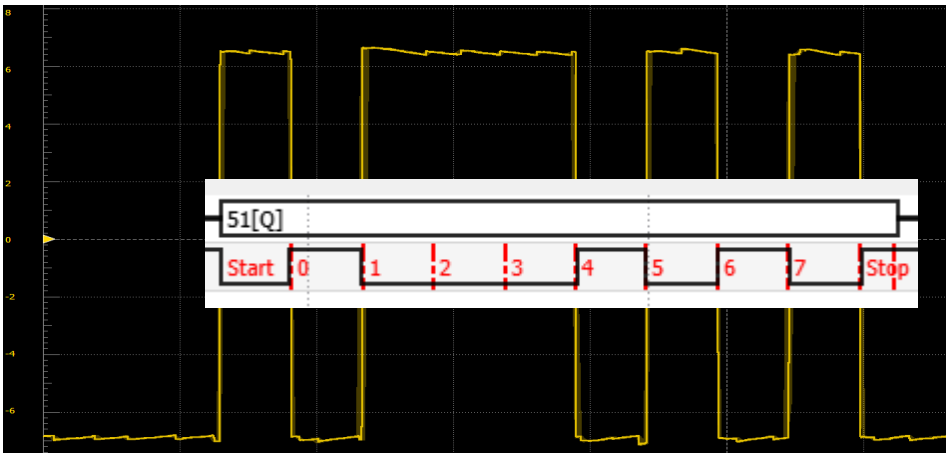
- 'Standard' transmission rates from 75-115200 BAUD* (these are specified speeds only, like 9600, not just any value in that range).
- Data may be sent as 7-bit standard ASCII characters, 8-bit extended ASCII characters, or binary data (note: this is interpretation, data is data).
- Simple error checking via parity is possible.
- The minimum time between characters (stop bits) may be adjusted.
- Handshaking and flow control are possible.

The SCI on our board also supports a 9-bit mode, but we won't use this, as it is non-standard, and is more complex to operate.

We will use the very common `8N1` configuration. This means one start bit, eight bits of data, no parity, and one stop bit. The start bit signals the start of communications. The eight bits that follow are the data payload, and the stop bit is used to set a minimum time between characters. The stop bit was once used to ensure that the receiver had time to process the received character, but this is not normally a consideration for modern equipment.

*BAUD is a pseudo acronym that effectively means "Bits per Second". You may find references that describe it as "Bits of Actual Usable Data", but this is confusing, as the start bit, stop bit(s), and optional parity bit are not part of the data payload. The actual signal is 8 of 10 bits transmitted in the 8N1 format.

Note: when converted to RS-232, the UART TTL signal is inverted, bipolar, and non-return-to-zero!



The yellow trace in the background is the RS-232 signal viewed on a scope (±7V).

The image imposed on top is from the protocol analyzer in the AD2 (0-5V).

Data is sent least significant bit first, so the data appears 'backwards' from how we normally view it.

When the signal is idle, the RS-232 level is negative with respect to ground. This is known as 'MARK'.

When the signal is active, the RS-232 level is positive with respect to ground. This is known as 'SPACE'.

By not using common (ground) for any valid signal (non-return-to-zero), RS-232 is easier to troubleshoot: if a signal is at ground, it is not connected correctly!

Setting the BAUD rate

The SCI module uses a clock 16 times the BAUD rate for sampling. To reserve clock for this, the bus rate is inherently divided by 16, then by the 13-bit `SCIBD` register. For example, if you wanted a BAUD rate of 9600, you would divide `20E6 / 16 / 9600 = 130.20833`. You can't put a fraction into the integral register, but 130 is 99.84% of ideal.

For some rates, the denominator can get pretty big, which is really bad in integer division. For example, with the fastest 'standard' rate of `115200`, we run into a problem:

`20E6 / 16 / 115200 = 10` in the integer realm, but that value was truncated from `10.85`, so a value of `11` would provide a more accurate BAUD rate. You can still use integer division to calculate the value for the SCIBD register, you just need to provide rounding that will push the value up the next whole number if truncation would lose a fraction over `0.5`. This is achieved by multiplying the operands by `10`, adding `5` to the result, then dividing the overall result by `10`. Doing this adds a 'half' to what would be the digit to the right of the decimal point. When truncation occurs, if the original value was less than `5` then there is no change; if the value was `5` or more, the added `5` 'rounds' it to the next digit prior to truncation.

Consider the `115200` BAUD example from above (all calculated with integer types):

`20E6 / 16 / 115200 = 10` (original integer value)

```
20E6 * 10 / 16 = 12500000
```

```
12500000 / 115200 = 108
```

```
108 + 5 = 113
```

`113 / 10 = 11` (with rounding implemented)

Using this method will provide more accurate BAUD rate values.

As it turns out, that oversampling permits some tolerance in the BAUD rate, and if the actual rate is within ~2% of ideal, the communications should function.

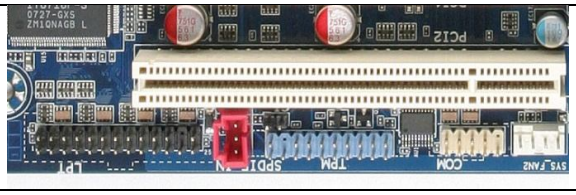
Remember that whatever value you put in the BAUD register must be limited to 13 bits, so the max divisor is `8191`. Using a divisor of zero will disable the BAUD generator.

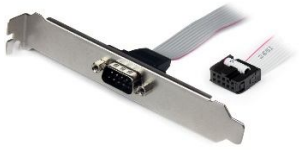
Most communications programs will expect one of several fixed BAUD rates. The following is a list of 'standard' BAUD rates, where ones in red are particularly common:

75, 110, 300, 600, 1200, 2400, 4800, 9600, 14400, 19200, 38400, 57600, 115200.

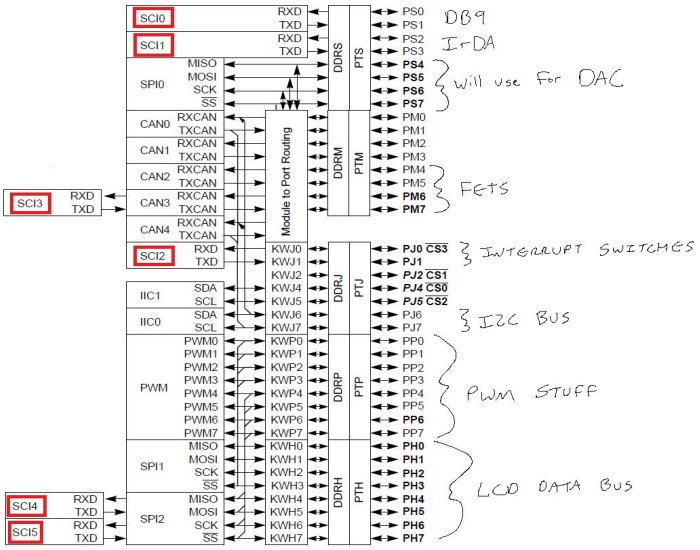
If your computer is not equipped with an actual serial port, you may need to use a USB to RS-232 adaptor cable. Some of these cables will not work with all rates (generally the very low ones).

If you are using a PC, there is generally a serial port available, but it might only be pinned out on the motherboard, and you would need to procure a cable to bring it to a connector on one of the expansion slots. Laptops, generally, do not have serial ports available – you must use a converter cable from USB.





There are several SCI modules on our chip, and the port you are using needs to be distinguished when using the registers:



The naming convention is to use the module number in the register name:

```
SCI0BD = 130;
```

`SCI0` is the SCI module that is connected to the DB9 on your board. `SCI1` is the SCI module that is connected to the IrDA module (infrared data). The remaining SCI ports should not be used, as these pins are allocated for other devices or modules (by the design of the board).

Once the BAUD rate is set (both devices must be set to the same BAUD rate), you must enable the SCI. Like other modules, this unit is powered off out of reset.

The `SCICR2` register is used to configure parts of the SCI. You are looking to turn on the transmitter and receiver, so the `TE` and `RE` bits of `SCICR2` need to be set to 1:

```
SCI0CR2 = 0b00001100;
```

11.3.2.6 SCI Control Register 2 (SCICR2)

	7	6	5	4	3	2	1	0
R	TIE	TCIE	RIE	ILIE	TE	RE	RWU	SBK
W								
Reset	0	0	0	0	0	0	0	0

Figure 11-9. SCI Control Register 2 (SCICR2)

Read: Anytime

Write: Anytime

Table 11-9. SCICR2 Field Descriptions

Field	Description
7 TIE	**Transmitter Interrupt Enable Bit** — TIE enables the transmit data register empty flag, TDRE, to generate interrupt requests. 0 TDRE interrupt requests disabled 1 TDRE interrupt requests enabled
6 TCIE	**Transmission Complete Interrupt Enable Bit** — TCIE enables the transmission complete flag, TC, to generate interrupt requests. 0 TC interrupt requests disabled 1 TC interrupt requests enabled
5 RIE	**Receiver Full Interrupt Enable Bit** — RIE enables the receive data register full flag, RDRF, or the overrun flag, OR, to generate interrupt requests. 0 RDRF and OR interrupt requests disabled 1 RDRF and OR interrupt requests enabled
4 ILIE	**Idle Line Interrupt Enable Bit** — ILIE enables the idle line flag, IDLE, to generate interrupt requests. 0 IDLE interrupt requests disabled 1 IDLE interrupt requests enabled
3 TE	**Transmitter Enable Bit** — TE enables the SCI transmitter and configures the TXD pin as being controlled by the SCI. The TE bit can be used to queue an idle preamble. 0 Transmitter disabled 1 Transmitter enabled
2 RE	**Receiver Enable Bit** — RE enables the SCI receiver. 0 Receiver disabled 1 Receiver enabled
1 RWU	**Receiver Wakeup Bit** — Standby state 0 Normal operation. 1 RWU enables the wakeup function and inhibits further receiver interrupt requests. Normally, hardware wakes the receiver by automatically clearing RWU.
0 SBK	**Send Break Bit** — Toggling SBK sends one break character (10 or 11 logic 0s, respectively 13 or 14 logics 0s if BRK13 is set). Toggling implies clearing the SBK bit before the break character has finished transmitting. As long as SBK is set, the transmitter continues to send complete break characters (10 or 11 bits, respectively 13 or 14 bits). 0 No break characters 1 Transmit break characters

NOTE: The SCI can generate interrupts for four different conditions. Eventually, we will use interrupts to manage processing of received data. Initially, you will poll for received data, to get a basic idea about how data is sent and received.

Much of the status of the SCI module is revealed through the `SCISR1` register:

	7	6	5	4	3	2	1	0
R	TDRE	TC	RDRF	IDLE	OR	NF	FE	PF
W								
Reset	1	1	0	0	0	0	0	0

= Unimplemented or Reserved

Figure 11-10. SCI Status Register 1 (SCISR1)

The two flags that are of principal interest are `TDRE` and `RDRF`. If `TDRE` is a 1 when the module is ready to accept a byte for transmission. You will check to ensure this flag is set before you attempt to send a byte.

The `RDRF` flag indicates that a new byte has been received. Initially you will use polling to check this bit to see if a byte has arrived for processing. If the flag is set, reading the flag then reading the byte from `SCIDRL` will clear it, making the module ready for another byte reception.

The following code demonstrates how to configure `SCI0` for 9600 BAUD communication. It will attempt to send out bytes as fast as possible, and display ASCII codes on the segs if a byte is received:

```
void main(void)
{
  // main entry point
  _DISABLE_COP();
  EnableInterrupts;

  /*****************************************************************/
  // initializations
  /*****************************************************************/
  PLL_To20MHz();
  SWL_Init();
  Segs_Init();

  // do SCI startups
  SCI0BD = 130; // 20E6 / (9600 * 16) // 11.3.2.1

  SCI0CR2 = 0b00001100; // turn on TX/RX // 11.3.2.6

  /*****************************************************************/
  // main program loop
  /*****************************************************************/
  for (;;)
  {
    // if the transmitter buffer is empty, load a new byte to send (TX)
    if (SCI0SR1_TDRE /*&& SWL_Transition (SWL_CTR)*/)
      SCI0DRL = rand() % 26 + 'A';

    // if a byte has been received, pull it!
    if (SCI0SR1_RDRF)
      Segs_8H (2, SCI0DRL);
  }
}
```

The main loop will run very quickly, as the micro is very fast relative to 9600 BAUD. In fact, a fun activity to try would be to figure out how many iterations the main loop runs for each character transmission (maybe add that code and put it on the second segs line).

Seeing the characters sent and received may be done at the UART level with the AD2, and at RS-232 levels with a terminal program.

Both the AD2 and terminal programs will allow you to send and receive characters, but you may also write programs in C# that use serial communications.

You will need to build up a library for the SCI that is capable of initializing the SCI, blocking send, non-blocking send, blocking receive, non-blocking receive, and possibly interrupt management:

```
// sci 0 - normal mode ******************************
// set baud, returns actual baud
unsigned long sci0_Init (unsigned long ulBusClock, unsigned long ulBaudRate);

// read a byte, non-blocking
// returns 1 if byte read, 0 if not
int sci0_read (unsigned char * pData);

// blocking byte read
// waits for a byte to arrive and returns it
unsigned char sci0_bread (void);

// send a byte over SCI (blocking)
void sci0_txByte (unsigned char data);

// send a null-terminated string over SCI
void sci0_txStr (char const * straddr);

// receive a string from the SCI
// up to buffer size-1 (string always NULL terminated)
// number of characters is BufferSize minus one for null
// once user enters the max characters, null terminate and return
// if user enters 'enter ('\r')' before-hand, return with current entry (null terminated)
// echo valid characters (non-enter) back to the terminal
// return -1 on any error, otherwise string length
int sci0_rxStr (char * const pTarget, int BufferSize);

// set/clear interrupt flags for SCI0
void sci0_SetIntFlag (unsigned char flags);
void sci0_ClrIntFlag (unsigned char flags);
// sci 0 - normal mode ******************************
```

These topics will be covered in class with demonstrations.

Using SCI Interrupts

The most useful interrupt for the SCI is arguably the *Receiver Full Interrupt*. This interrupt will occur when a byte has been received and is ready to be pulled from the receive buffer. Because we typically don't know when a byte will arrive, and one could *never* arrive, this interrupt can save a lot of polling. Additionally, if we want to use the micro for CPU intensive work while performing communications, particularly higher-speed communications, the interrupt model can make the code more responsive, less prone to missing data, and more efficient.

The SCI module routes all interrupt events to a single ISR, so if you request more than one interrupt source, you must determine the source of the interrupt in the ISR by flag checking. If you have only one interrupt cause enabled, you may skip this step.

Interrupts for the SCI are managed through the `SCIxCR2` register, as shown above. To enable an interrupt when data is fully received, you would set `RIE` to `1`.

```
// setup interrupt for RDRF
SCI0CR2_RIE = 1;
```

In doing so, you have committed yourself to dealing with this interrupt. You will need a suitable ISR:

```
interrupt VectorNumber_Vsci0 void ISR_SCI0 (void)
{
    // single read to capture flags
    unsigned char status = SCI0SR1;

    // if you've done more than one interrupt on this device
    //  you need to identify the interrupts, otherwise, clear the
    //  one and only one you asked for...

    // TDRE: cleared by reading SCI0SR1 w/B7 set, then write to SCI0DRL
    // RDRF: cleared by reading SCI0SR1 w/B5 set, then reading from SCI0DRL

    // check SCI0SR1 for RDRF, this does the int clearing operation (use lib function)
    if (status & SCI0SR1_RDRF_MASK)
    {
      // retrieve byte by reading from SCI0DRL (use library method)
    }

    // other flags may still be set (if requested), so continue checking other int sources
    if (status & SCI0SR1_TC_MASK)
    {
      // send next byte by writing to SCI0DRL (use library method)
    }
}
```

Normally you will only have `RIE` enabled, so the ISR is simpler than shown above. Section 11.3.2.7 in Big Pink discusses flag clearing for each interrupt flag in the `SCIxSR1`:

5 RDRF	**Receive Data Register Full Flag —** RDRF is set when the data in the receive shift register transfers to the SCI data register. Clear RDRF by reading SCI status register 1 (SCISR1) with RDRF set and then reading SCI data register low (SCIDRL). 0 Data not available in SCI data register 1 Received data available in SCI data register

For RIE and any other interrupts you end up using, you must adhere strictly to the flag clearing mechanism. Your library functions should be written to automatically clear the flags through normal behavior.

The following program, for example, will display received characters on the segs using interrupts:

```
void main(void)
{
  // main entry point
  _DISABLE_COP();
  EnableInterrupts;

  PLL_To20MHz();
  Segs_Init();

  // start SCI at 38400
  (void)sci0_Init(20E6, 38400);

  // setup interrupt for RDRF
  SCI0CR2_RIE = 1;

  for (;;)
  {
    asm wai;
  }
}

interrupt VectorNumber_Vsci0 void ISR_SCI0 (void)
{
  // only one source of interrupt! RIE (RDRF), no need to check flags!
  // your blocking read will test RDRF and read data
  // flag is cleared via bread function!
  unsigned char data = sci0_bread();
  Segs_8H (0, data);
}
```

In the case of a single interrupt source, the ISR is much simpler to write! Most of the time this is what you will be doing, but you should be prepared for more complex operation of the SCI.

How you handle the data in the ISR is worthy of note. Remember that you don't want the ISR to be long in execution, as it will suspend subsequent interrupts. At high data rates, this could mean the loss of data. Ideally the ISR will simply put the received character into a buffer (queue or similar) for processing, and the main code would deal with it, but this is beyond the scope of this course.

You will instead store the received data in a location visible to the main program code and will use a flag to indicate availability. As a result, the data processing work you will do will be relatively simple.