SCI Communications

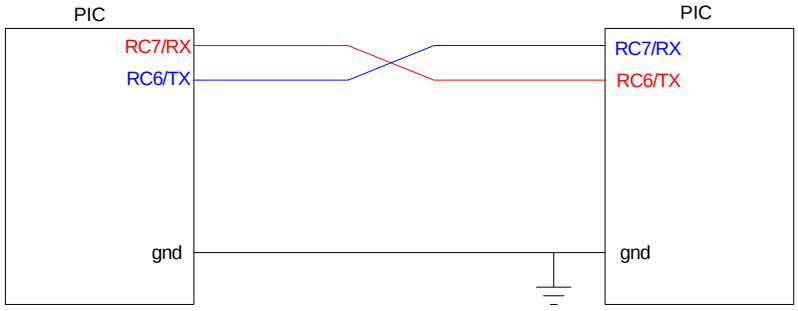
Goal:

- Send and receive serial data using SCI protocol.
- Send data to a PC via the RS232 serial port.

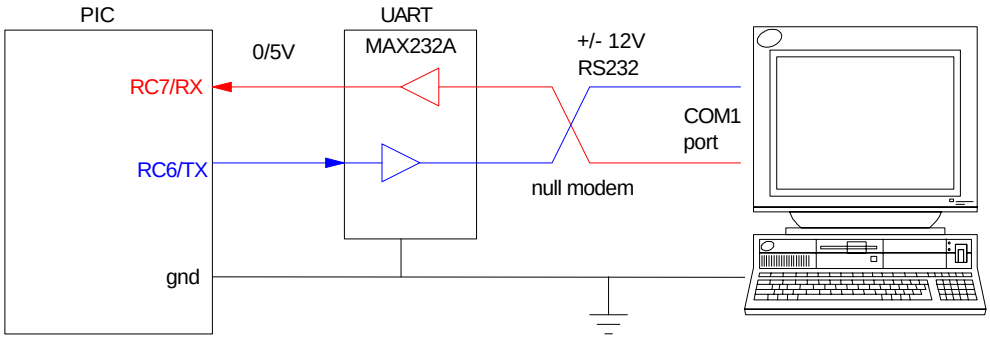
Why:

- Debugging code.
  - Sending data to a PC is a useful way to debug code, similar to the LCD display, or collect data.
  - Hyperterm lets you see and save data that comes in on the COM1 or COM2 port.
- Efficiency. You can send an unlimited amount of data using 3 lines in full duplix mode:
  - RC7/RX receive: data being sent to the PIC
  - RC6/TX transmit: data being sent from the PIC
  - ground common ground (not needed if launching a wave)
- or 2-lines in 1/2 duplix mode:
  - Data transmit / recieve data
  - Ground common ground (not needed if launghing a wave)

Hardware Connection:



PIC to PIC communication via SCI communications

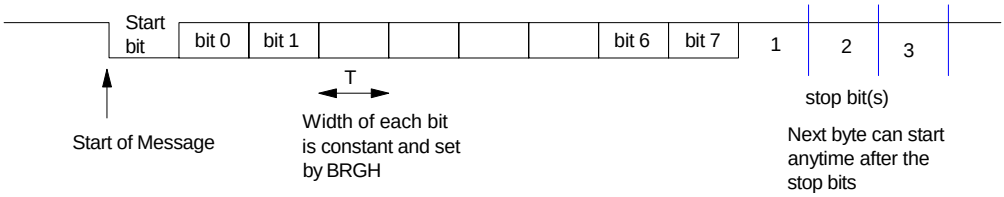


PIC to PC communication via SCI communications. Note that a UART is required to convert PIC voltage levels (0/5V) to RS232 voltage levels (±12V).

Also note that you need to swith the TX/RX lines between devices. This is what a null-modem does on a dB9 connector.

Timing:

A generic SCI message looks like the following:



The data is sent and recieved as

- A start bit (high to low transition)
- 8 data bits - least sgnificant bit first
- 0, 1, or 2 stop bits

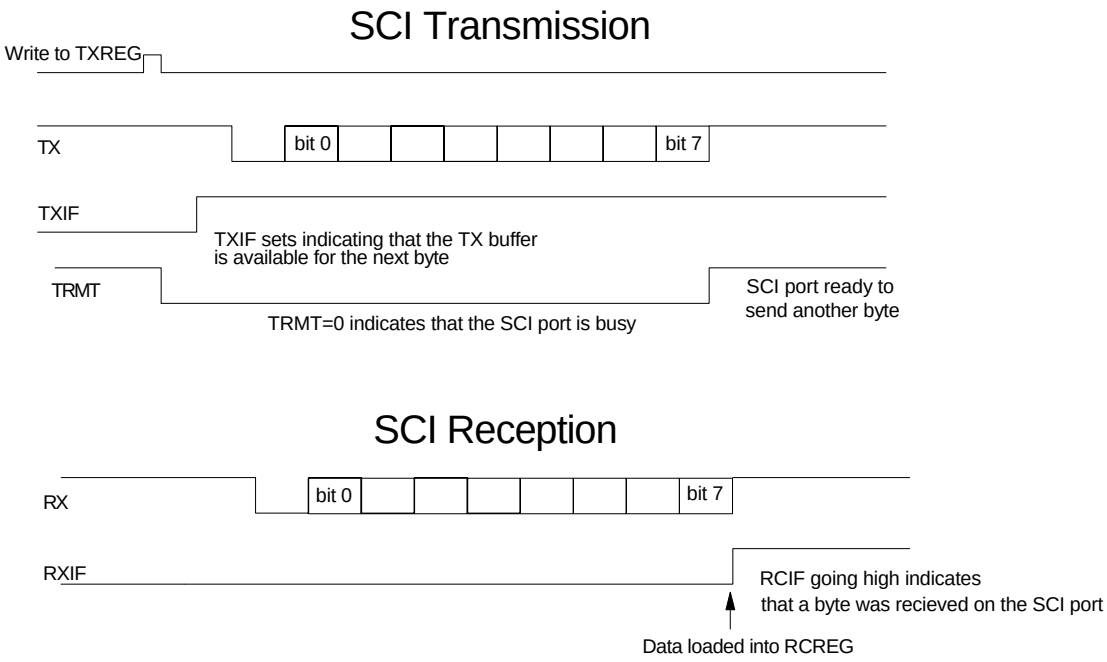
The start bits signals that a message is coming. It is needed since the first data bit may be the same as the default state of the data line. Since there is no clock, the reciever wouldn't know that anything has changed and that data is incoming.

If you use the built-in SCI port, the timing for reading and writing the bits is automatic. For transmission:

- The hadware adds the start and stop bits
- The hardware sets up the timing for each bit.
- All you need to do in software is
  - Check that the SCI port is free (wait until TRMT=1)
  - Start the data transmission (write to TXREG)

For reception

- The hardware detects the start bit automatically
- The hardware sets up the timing for reading in each bit
- The hardware actually reads each bit three times and does best-of-three voting to reduce errors
- The hardware then signals the software when 8-bits have beed read in by setting RCIF
- Once a bye has been recieved, the data can be read from RCREG.



note: Transmission and reception are not synchronized. Either can happen at any time relative to the other.

How: Software:

1. Set up PORTC as follows:

TRISC (address 0x__ - Bank __)

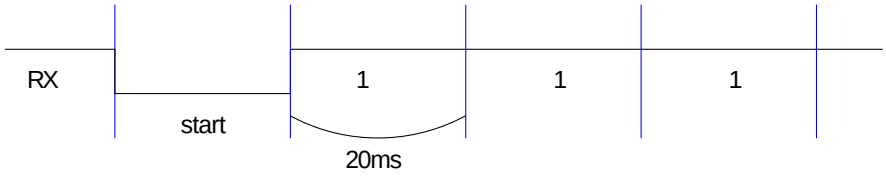
Bit	7	6	5	4	3	2	1	0
name	RX	TX	-	-	-	-	-	-
value	1	1	x	x	x	x	x	x

note: Both transmit and receive are set up as input.

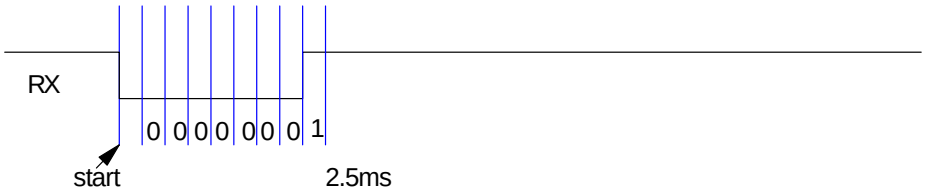
- When TXEN=1 (transmit enable), you override TRISC and make RC6 an output.
- When TXEN=0 (transmit disabled), RC6 returns to high-impedance. This allows someone else to drive the data line.

2. Set the baud rate. There is no clock, so the two devices **must** know how long each bit is. For example, suppose you recieve the following data:

If you think the data was transmitted at 50 baud (for one bit being 20ms), you'd read this as 0xFF



If you think the daa was transmitted at 4k baud (for one bit every 2.5ms), you'd read this as 0x80 (remember: LSB first)



Moral: The transmitter and reciever must be set to the same baud rate or else the data won't get through correctly.

The baud rate is set by bits SYNF, BRGH, and SPBRG

TABLE 10-1: BAUD RATE FORMULA (F_{OSC} = 10,000,00)		
SYNC	BRG16 = 1 BRGH = 0	BRG16 = 1 BRGH = 1
0	Baud Rate = $F_{OSC}/(16(X+1))$	Baud Rate= $F_{OSC}/(4\ (X+1))$

X = SPBRGH : SPBRG (0 .. 65,535)

Some common settings for a 20MHz crystal follow:

Baud Rate	SPBRG	BRGH	BRG16	SYNC	Error (%)
2,400	255	0	1	0	-1.70%
4,800	129	0	1	0	-0.16%
9,600	255	1	1	0	-1.70%
19,200	129	1	1	0	-0.16%
38,400	64	1	1	0	-0.16%
57,600	42	1	1	0	-0.95%
115,200	21	1	1	0	+1.44%

Transmit Status Register: TXSTA (address 0x98 - Bank 1)

Bit	7	6	5	4	3	2	1	0
Name	CSRC	TX9	TXEN	SYNC	-	BRGH	TRMT	TX9D
Write Vaue (for set-up)	1	0	1	0	-	1/0	-	-
Read Value	-	-	-	-	-	-	0/1	bit 9

bit 7: CSRC: Clock Source Select bit

- 1 = Clock generated by PIC
- 0 = Clock from external source

bit 6: TX9: 9-bit Transmit Enable bit

- 1 = Selects 9-bit transmission
- 0 = Selects 8-bit transmission

bit 5: TXEN: Transmit Enable bit

- 1 = Transmit enabled
- 0 = Transmit disabled

bit 4: SYNC: USART Mode Select bit

- 1 = Synchronous mode
- 0 = Asynchronous mode

bit 3: Unimplemented: Read as '0'

bit 2: BRGH: High Baud Rate Select bit

- 1 = High speed
- 0 = Low speed

bit 1: TRMT: Transmit Shift Register Status bit

- 1 = TSR empty
- 0 = TSR full

bit 0: TX9D: 9th bit of transmit data. Can be parity bit.

RCSTA: RECEIVE STATUS AND CONTROL REGISTER (ADDRESS 18h)

Bit	7	6	5	4	3	2	1	0
Name	SPEN	RX9	SREN	CREN	ADDEN	FERR	OERR	RX9D
Write Vaue (for set-up)	1	0	1	1	0	-	-	-
Read Value (reception)	-	-	-	-	-	1/0	1/0	1/0

bit 7: SPEN: Serial Port Enable bit

- 1 = Serial port enabled (Configures RC7/RX/DT and RC6/TX/CK pins as serial port pins)
- 0 = Serial port disabled

bit 6: RX9: 9-bit Receive Enable bit

- 1 = Selects 9-bit reception
- 0 = Selects 8-bit reception

bit 5: SREN: Single Receive Enable bit (This bit is cleared after reception is complete.)

- 1 = Enables single receive
- 0 = Disables single receive

bit 4: CREN: Continuous Receive Enable bit

- 1 = Enables continuous receive
- 0 = Disables continuous receive

bit 3: ADDEN: Address Detect Enable bit

Asynchronous mode 9-bit (RX9 = 1)

- 1 = Enables address detection, enable interrupt and load of the receive burffer when RSR<8> is set
- 0 = Disables address detection, all bytes are received, and ninth bit can be used as parity bit

bit 2: FERR: Framing Error bit

- 1 = Framing error (Can be updated by reading RCREG register and receive next valid byte)
- 0 = No framing error

bit 1: OERR: Overrun Error bit

- 1 = Overrun error (Can be cleared by clearing bit CREN)
- 0 = No overrun error

bit 0: RX9D: 9th bit of received data (Can be parity bit)SSPEN = 1 to enable the SPI port.

Sample Routines: SCI Data Transmission

Program #1: Initialize the SCI port to send and recieve data at 9600 baud:

```
void SCI_Init(void)
{
   TRISC = TRISC | 0xC0;
   TXIE = 0;
   RCIE = 0;
   BRGH = 1;
   BRG16 = 1;
   SYNC = 0;
   SPBRG = 255;
   TXSTA = 0x22;
   RCSTA = 0x90;
   }
```

Program #2: Send the message 'Hello' to the SCI port at 9600 baud.

- You can see this on the PC if you load Hyperterminal, set it to 9600 baud, 8 data bits, 1 stop bit, no handshaking.
- You will also need to use a UART to convert the 0/5V signals to +/- 12V.

```
// Global Variables
const unsighed char MSG[6] = "Hello";

// Subroutine Declarations
#include  <pic.h>

// Subroutines
#include        "function.c"
#include        "sci_init.c"    // see above

// main
{
    unsigned char i;
    SCI_Init();

    for (i=0; i<5; i++) {
       while(!TRMT);  TXREG = MSG[i];
       }
    }
```

Program #3: Read the A/D input and send its reading to a PC once every second:

```
// Global Variables

// Subroutine Declarations
#include  <pic.h>
#include  <stdio.h>

// Subroutines
#include        "function.c"
#include        "sci_init.c"    // see above
#include        "a2d.c"


// main
{
    SCI_Init();

    while(1) {

// read the A/D input.  Data = 0..1023

       Data = A2D_Read(0);

// Send Data to the PC via the SCI port in hexadecimal format.

       while(!TRMT);  TXREG = ascii(Data >> 12);
       while(!TRMT);  TXREG = ascii(Data >> 8);
       while(!TRMT);  TXREG = ascii(Data >> 4);
       while(!TRMT);  TXREG = ascii(Data);

// end with a carriage return <13>, line feed <10>

       while(!TRMT);  TXREG = 13;
       while(!TRMT);  TXREG = 10;

// and repeat every second.

       Wait_ms(1000);

       }
    }
```

Sample Routines: SCI Data Reception

Write a routine which

- Reads in data from the PC
- Echos back each character as you type it in
- Saves the message in a buffer, and
- Looks for a carriage return <13> to terminate the message.

Since the data could arrive at any time

- Use interrupts to save the data.
- Use a stack to save the data as it comes in.
- Echo back what you read on the SCI port. If you connect to a PC, you should see Display the data on an LCD display.

```
// Global Variables

bank1 unsigned char MSG[20];              // the data coming in
bank1 unsigned char MSG_LENGTH;           // message length
bank1 unsigned char MSG_FLAG;             // set when <cr> seen


// Interrupt Service Routine
void interrupt IntServe(void) @ 0x10

   if (RCIF) {

      TEMP = RCREG;
      while (!TRMT);  TXREG = TEMP;

      if (TEMP > 20) MSG[MSG_LENGTH++] = TEMP;
      if (MSG_LENGTH > 19) MSG_LENGTH = 19;

      if (TEMP == 13) MSG_FLAG = 1;
         }

      RCIF = 0;
      }
   }
```

The main routine now

- Looks at MSG_FLAG. If set, a message is waiting in the buffer.
- You can access the message by looking at MSG[i] for i=0..MSG_LENGTH
- Once you receive the message, clear MSG_FLAG and clear MSG_LENGTH.