I/O Control with the SX Microcontroller

Educational Tutorial for the SX University Program

Version 1.0

PARALLAX INC

Warranty

Parallax warrants its products against defects in materials and workmanship for a period of 90 days. If you discover a defect, Parallax will, at its option, repair, replace, or refund the purchase price. Simply call for a Return Merchandise Authorization (RMA) number, write the number on the outside of the box and send it back to Parallax. Please include your name, telephone number, shipping address, and a description of the problem. We will return your product, or its replacement, using the same shipping method used to ship the product to Parallax.

14-Day Money Back Guarantee

If, within 14 days of having received your product, you find that it does not suit your needs, you may return it for a full refund. Parallax will refund the purchase price of the product, excluding shipping / handling costs. This does not apply if the product has been altered or damaged.

Copyrights and Trademarks

Disclaimer of Liability

Parallax, Inc. is not responsible for special, incidental, or consequential damages resulting from any breach of warranty, or under any legal theory, including lost profits, downtime, goodwill, damage to or replacement of equipment or property, and any costs or recovering, reprogramming, or reproducing any data stored in or used with Parallax products. Parallax is also not responsible for any personal damage, including that to life and health, resulting from use of any of our products. You take full responsibility for your BASIC Stamp application, no matter how life-threatening it may be.

Internet Access

We maintain internet systems for your use. These may be used to obtain software, communicate with members of Parallax, and communicate with other customers. Access information is shown below:

E-mail:	sxtech@parallaxinc.com
Ftp:	ftp.parallaxinc.com - ftp.stampsinclass.com - ftp.sxtech.com
Web:	http://www.parallaxinc.com - http://www.stampsinclass.com - http://www.sxtech.com

Table of Contents

Forward

Forward: I/O Control with the SX Microcontroller

Introduction

One of the things that makes the Scenix SX microcontroller so powerful is its versatile I/O. Traditionally, microcontrollers have incorporated internal or external hardware for handling various I/O requirements. Particularly with internal hardware solutions, a different microcontroller must be selected to match each new design. Manufacturers have, in turn, come up with an increasingly large number of microcontroller packages. They do so in an attempt to fit their products into as many different designs as possible. The circuit designer ends up losing a degree of freedom when attempting to use these products. For example, when one chooses a package with one asynchronous I/O port and one A/D port, adding one more A/D line can be costly in terms of redesign time and hardware.

One thing that sets the SX apart from most microcontrollers is that it is fast enough to handle many forms of I/O in software instead of requiring special hardware. This allows the designer to simply change the SX program to meet the new design requirements. This is possible because of the SX chip's comparatively high processing speed. In future units, you'll see how to use this processing speed to create asynchronous serial ports, A/D ports, and more.

About This Course

In the first part of this course, *Introduction to Assembly Language Programming with the SX Microcontroller* covered the basics of SX processing and I/O. This second part covers more advanced I/O that can be implemented with the SX in software. These techniques are essential tools that can be used by embedded systems designers in an ever increasing variety of applications.

In addition to I/O, this course will highlight the use of macros. Although not strictly necessary, macros do make programs easier to write and understand. Macros can be used to make composite pseudo instructions consisting of multiple assembly instructions. Parameters can be used within a macro making it possible to customize the code it produces.

The SX chip's E2/Flash memory can be erased and reprogrammed more than 10,000 times. This allows students the luxury of trial and error with their assembly language programs. Coupled with the powerful SX-Key debugging tools, the SX Tech Tool Kit provides an ideal environment for learning and experimentation. The experiments in this course are best performed with the SX-Tech Tool Kit, available for on-line purchase at www.sxtech.com.

Forward - I/O Control with the SX Microcontroller

This text was written assuming the reader has already completed *Introduction to Assemble Language Programming with the SX Microcontroller.* The suggested background for this course is a familiarity with electric circuits and elementary digital electronics. Previous experience with a computer programming language is also important. Those with little or no electronics or computer programming background are urged to complete *What's a Microcontroller*? and *Basic Analog and Digital* before getting started with SX Tech. These introductory texts are part of the stamps in class curriculum, available at www.stampsinclass.com.

Unit I. Simple Hardware I/O Enhancements

Unit I from I/O Control with the SX Microcontroller

Introduction

The SX has a variety of built in, programmable I/O enhancements that can be used in place of certain external circuits. Options can be set to enable internal pull up resistors, configurable logic thresholds, and analog comparator functions. These features were first discussed in *Introduction to Assembly Language Programming with the SX Microcontroller*, chapter 6. Although these features can be used to reduce the overall parts count in many designs, they are for use with specific current and voltage limits. Three of the most common situations where the demands of a peripheral device exceed these limits are when:

- The device requires more current the SX I/O pin can supply.
- The device requires more than 5 V at its input.
- The device outputs above 5 V or below 0 V.

This chapter introduces some simple hardware solutions for these situations. These solutions can be used to make the SX light lamps, energize relays or coils, and control motors, or even pumps. Hardware solutions for RS232 voltages are also discussed because many applications make use of this standard, such as the serial port on a PC.

Specialized interfaces, such as liquid crystal display (LCD) drivers, or computer I/O ports, use a variety of different hardware connection schemes. Most of these devices also use one of several established communication protocols for exchanging data. These protocols are discussed, and an example of a hardware/software interface with a common parallel LCD is included. This will help introduce some basic I/O and register management techniques, setting the groundwork for methods used in later chapters.

Driving Loads

Compared to many chips, an SX I/O pin set to output can sink or source significant amounts of current (30 mA). This is plenty for driving an LED as well as most IC inputs. However, for many relays, lamps, and other loads, 30 mA is not nearly enough. Attempting to use an SX I/O pin to drive a high current load can damage the chip.

Fortunately, the SX chip's output capacity can be extended using simple external parts. Figures 1.1, 1.2 and 1.3 show three circuits that can be used to significantly boost the SX chip's output capacity. Figure 1.1 shows a circuit built around a common 2N2222 transistor. This circuit draws minimal current from the SX, but can sink nearly a half of an ampere when heat sinking is used on the transistor.

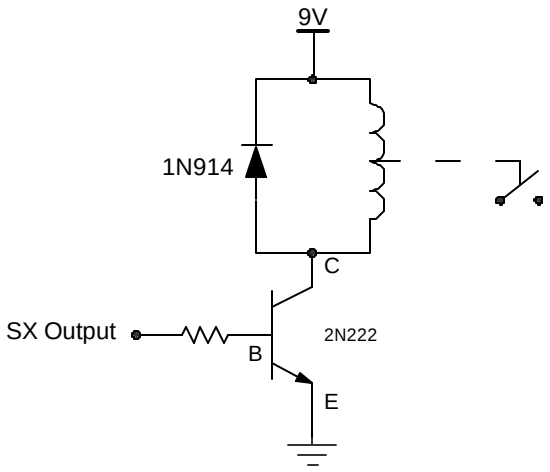


Figure 1.1 – Switching a high-current relay

This configuration is ideal for loads that require ground to be switched on and off. When the SX switches its output high, the voltage at the transistor's base, V_{BE}, rises to 0.7 V. The current through the resistor connected to the transistor's base is (5 – 0.7)/1000 = 4.3 mA. This is ample current to force the transistor into saturation without demanding too much current from the SX I/O pin.

Because the transistor is saturated, the collector will be in the neighborhood of 0.2 V above the emitter. For practical purposes, this is as good as ground. When the SX outputs 0 volts, or any voltage too low to bring V_{BE} above 0.7V, the transistor switches off. Although a very small amount of current is still conducted, it is insignificant as far as the coil is concerned.

TIP: Notice the diode across the relay coil in Figure 1.1. This is useful when driving inductive loads. When the current in any inductor changes, it can cause large voltage spikes, which can destroy the transistor. The diode shorts out negative voltage to prevent damage to the transistor. A relatively low inductance load, such as a light bulb, does not require the diode.

Most of the time, switching the ground lead of a load on and off works fine. However, some jobs require a positive voltage to be switched. For example, suppose an EPROM programmer requires a 14V supply to be switched on and off. The circuit in Figure 1.2 can be used for his application.

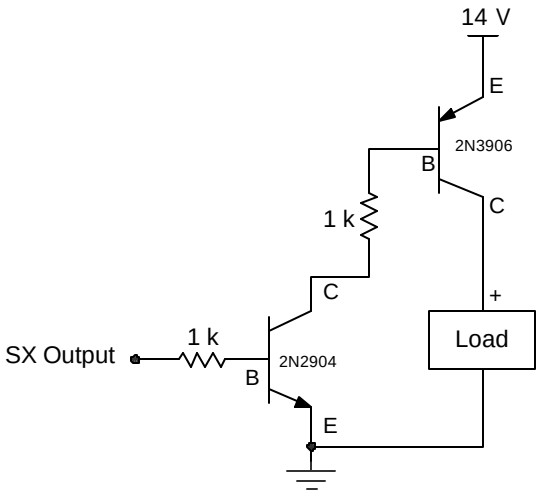


Figure 1.2 Circuit for switching a positive voltage.

The NPN transistor works as before, making a ground connection when the SX outputs a 1. This causes the voltage across the base of the PNP transistor to turn it on because the magnitude of V_{BE} will be greater than 0.7 V. As with the previous circuit, the magnitude of V_{CE} can be neglected.

The circuit in Figure 1.3 uses a power MOSFET. A MOSFET offers almost complete isolation between the processor and load. Modern MOSFETs can also handle relatively heavy current loads, and the device shown here can conduct up to 4A. Another MOSFET advantage is that it has a very low series resistance, in the neighborhood of 0.54 Ohms, when switched on.

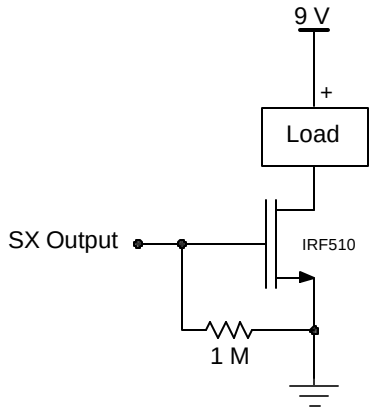


Figure 1.3 Using a MOSFET

The circuits just introduced will serve in a variety of situations, all of which are aimed at switching DC loads on and off. However, many designs call for something other than on/off values.

Analog I/O

Many practical sensors generate analog signals, and there are several strategies for reading analog values with a digital device like the SX. A common external hardware solution is to use specialized ICs that can convert analog to digital and vice versa. A device that converts numeric quantities to analog is called a Digital to Analog converter (DAC or D/A). The opposite function is performed by an Analog to Digital converter (ADC or A/D). These are available from many vendors with varying capabilities and price tags. SX software A/D and D/A solutions also exist, and will be introduced in Units 3, 5, and 6. In some cases, A/D conversion is overkill, because the voltage can be "trimmed" to a more appropriate level .

Analog Level Conversion

For an example of a trimming circuit, consider a battery monitor. Assume a battery's nominal voltage is 9 V, and the circuit will operate at voltages as low as 7.2 V. Your design goal is to detect when the voltage drops to 7.5 V, perhaps to light a low voltage indicator.

Using an A/D converter for this job would be a waste of money and resources. Taking advantage of an SX I/O pin's logic threshold is a much simpler, less expensive solution. When an SX I/O pin is set to CMOS input mode, it reads signals above 2.5 V as 1 and below 2.5 V as 0. A voltage divider can convert the 7.5V target voltage to 2.5V. A voltage divider is shown in Figure 1.4, and the voltage divider equation is given by:

$$V_o = V_i\left(\frac{R_2}{R_1 + R_2}\right) \qquad (1)$$

Resistor values should be selected to make V_o = 2.5 V when Vi = 7.5 V. A 10 and 20 KΩ resistor would do the job. However, the total current consumed will be 9/30000 or 300 µA. This is plenty of current to drive the SX inputs. The values could also be increased to 100K and 200K to reduce current consumption to 30 µA.

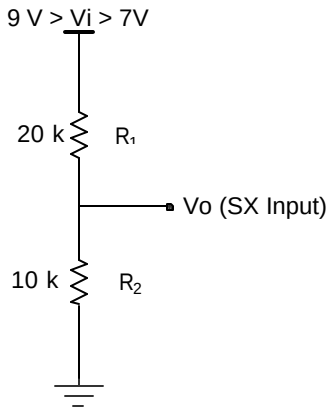


Figure 1.4 Detecting a low battery

When the battery is at full charge, the input pin will be 3 V, which is enough voltage for the SX to register a 1. At 7.5 V the pin drops to 2.5 V, which is right at the logic threshold. Any further drop is read by the SX as a zero. Compared to either software or hardware A/D conversion, this technique greatly simplifies both the programming and hardware used in the design.

Grouping Digital I/O – LCD Example

When using an individual SX I/O pin for switching and sensing, a single bit in a given port register is addressed. However, peripheral devices connected to microcontrollers have traditionally used parallel interfaces. These devices can be accommodated using the SX, but it's not necessarily the best use of the SX chip's limited number of I/O pins.

When reading and writing to parallel devices, each I/O port can be treated as a group of bits. For example, instead of treating **rb.1** through **rb.7** as individual bits, the **RB** register can be can be addressed as a group of 8-bits. In the 28 pin SX chip, **RA** is a 4-bit wide register, and **RB** and **RC** are each 8-bits wide. Keep in mind that if the data bus connected to the SX is not 4 or 8-bits, the program must be adapted to handle the data correctly.

Consider a typical liquid crystal display (LCD). Common LCDs use an on-board LCD driver IC such as the Hitachi HD44780 or a compatible device. Larger LCDs use a 44780 plus some additional Hitachi parts, but the programming turns out to be essentially the same. The 44780's datasheet is at the Hitachi Web site: semiconductor.hitachi.com/products/pdf/99rtd006d1.pdf.

LCD Hardware

The 14 pins on the LCD are likely arranged in the standard configuration given in Table 1.1.

Table 1.1: Pin Functions and Descriptions for Common LCDs with Hitachi or Compatible Driver

Pin	Function	Description	Pin	Function	Description	Pin	Function	Description
1	GND	Ground	2	+5	+ 5 V Power	3	C	Contrast voltage
4	RS	Reg. Select	5	R/W	Read/Write	6	E	Enable
7	DB0	Data Bit 0	8	DB1	Data Bit 1	9	DB2	Data Bit 2
10	DB3	Data Bit 1	11	DB4	Data Bit 4	12	DB5	Data Bit 5
13	DB6	Data Bit 6	14	DB7	Data Bit 7			

Some LCDs have 14-pin male single inline package (SIP) headers, and they can be plugged directly into a breadboard. Other LCDs have these pins arranged with a piece of ribbon cable that ends in a dual-row header. This isn't very handy for breadboarding. In this case, jumper wires can be used to connect the header pins/sockets to the breadboard. Figure 1.5 shows a connection diagram for operating a 14 pin LCD in 4-bit mode.

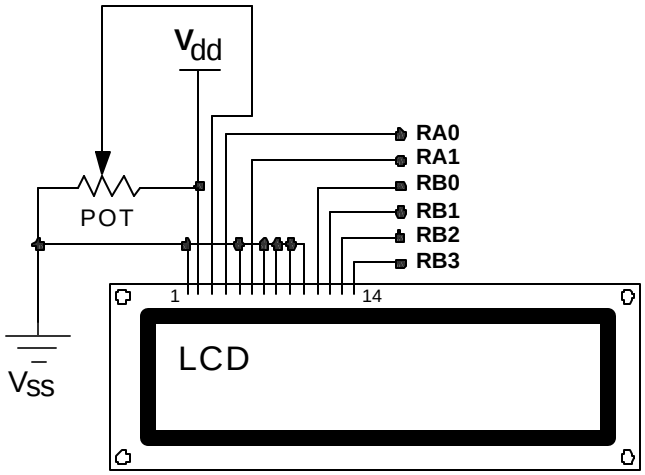


Figure 1.5 LCD Connection Diagram

Hitachi's data sheet shows a signal sequence that can be sent to the LCD to reset it and force it into 4-bit mode. Once in 4-bit mode, RS can be asserted, then ASCII characters can be sent. In 4-bit mode, the four most significant bits are sent first, and the lower four bits are sent second. RS is taken low for sending command codes. Each 4-bit transfer occurs when the E pin is pulsed.

If the LCD doesn't appear to work, try varying the contrast voltage on pin 3 of the LCD's 14-pin connector. Adjust the potentiometer connected to pin 3 until faint boxes or characters become visible. Note: Very few LCDs require negative voltages to set the contrast.

Program Listing – LCD Interface

Program Listing 1.1 is an LCD interface example using the techniques just discussed. The program displays a message you can change by changing the text in single quotes in the message routine.

```
             ; Program Listing 1.1
             ; 4-bit LCD driver by Al Williams

             device SX28L,turbo,stackx_optionx,oscxt5,bor42
             freq 4000000               ; Run at 4MHz to simplify timing.
             reset  start               ; Go to 'start' on reset.

             org     $0c
             dlyctr ds 1                ; Main delay counter.
             dlymultds 1                ; Delay multiplier.
             tmp           ds 1         ; Temp storage.
             work          ds 1         ; More temp storage.
             i             ds 1         ; Loop counter.

             ebit          equ ra.1     ; I/O: Enable and Register Select.

             rsbit         equ ra.0     ; Assumes DB4 to DB7 connect to RB.0-RB.3.

             org    0


ldelay       mov dlymult,#5             ; Long delay (5x256). Enter here if you want
delaym       clr dlyctr                 ; to set your own dlymult.

:delay       nop
             djnz dlyctr,:delay
             djnz dlymult,delaym
             ret

init         mov ra,#0                  ; Call to init the LCD.
             mov rb,#0                  ; Set all bits to zero.
             mov !rb,#%11110000         ; Set outputs.
             mov !ra,#%00
             call ldelay                ; Give LCD some time to catch up.
             mov rb,#$3                 ; Write a 3 out to the display 3 times.

             call pulsee
             call pulsee
             call pulsee

             mov rb,#$2                 ; Now go to 4-bit mode (twice).
             call pulsee
```

```
              call pulsee
              mov rb,#$8                ; Set 2-line mode (remove next 2 lines if
                                        ; display has 1 line).
              call pulsee

              mov w,#14                 ; Non blink cursor (use 15 for blinking).
              call lcdout
              mov w,#6                  ; Activate the cursor.
              call lcdout
              clear                     ; Clear the screen (init falls
                                        ; Into this routine).
              mov w,#1                  ; Send a command (clear falls
                                        ; Into this routine).
cmd           clrb rsbit
              call lcdout
              setb rsbit
              ret

lcdout        mov tmp,w                 ; Write to the LCD (4 bits at a time).

              mov work,w
              rr work                   ; Get top 4 bits first.
              rr work
              rr work
              rr work
              and work,#$F
              mov rb,work
              call pulsee
              mov w,tmp                 ; Then bottom 4 bits.
              and w,#$F
              mov rb,w
pulsee        setb ebit                 ; Pulse the E bit (lcdout falls into this).

              call ldelay
              clrb ebit
              ret

              ; Set the cursor to the specified pos note that all displays think that
              ; line 2 starts at pos 40 even if they don't have 40 characters.

setcursor     mov work,w
              mov w,#$80
              add w,work
              jmp cmd

              lookup mov w,i            ; Get a byte from the string to display.
              jmp pc+w
msg           retw 'Assembly Language I/O '
              retw 'with the SX-Key',13
              retw 'by Al Williams and Parallax',0

start         call init                 ; Here is the main program.
              call ldelay
```

```
              clr i                     ; Loop for each character.

ploop         call lookup
              ; exit if 0
              test w
              jz :loop
              inc i

              mov work,w                ; If 13 then go to line #2.
              cje work,#13,nl
              mov w,work

              call lcdout               ; Not 0 or 13 so print it.

              ; this delay gives a "teletype" effectcomment the following 2 lines
              ; for full speed.

              clr dlymult
              call delaym

              jmp ploop                 ; Keep going.

              ; This look waits for about 5 seconds or so and then starts the whole
              ; thing over.

:loop         mov tmp,#64
:loop1        clr dlymult
              call delaym
              djnz tmp,:loop1
              jmp    start

nl            mov w,#40                 ; Move to line 2.
              call setcursor
              jmp ploop
```

This program listing assumes no other part of the program uses ports A and B. If the pins not used by the LCD in a port are set to input, the program can write to the port bits, but no output occurs. On the other hand, if pins not used by the LCD are outputs, the command below will arbitrarily wipe out any output bits used by the other part of the program. This would lead to spurious outputs each time information is sent to the LCD. One solution is to read the output bits already in use before writing back to the port. In other words, instead of writing directly to a port using the command:

```
mov rb,bits
```

Substitute the code below:

```
and bits,#$F
mov w,rb
and w,#$F0
or w,bits
```

```
mov rb,w
```

In this example, only the four least significant bits in **RB** change. The hexadecimal value **#$F** is referred to as a mask. Masks are used with logic commands to force certain bits high or low within registers. For example, the first command:

```
and bits,#$F
```

forces the upper nibble (bits 4 through 7) in the bits register to zero while leaving the lower four bits unaffected. **RB** is then copied to the **w** register, followed by applying a mask that sets only the lower four bits in the **w** register to zero. The or command can then be used to copy the lower four bits in the bits variable into the **w** register. The contents of w can then be copied to **RB**. Although it seems like a roundabout way of doing things, it enables numeric control of groups of bits within a given I/O port.

About Serial Data

The LCD controller is a good example of a parallel interface with a peripheral device. The interface uses a total of six I/O pins, four for data and two for control. If an 18-pin SX is used, this would monopolize half of the twelve available I/O lines. Parallel interfaces that use too many I/O lines are a common problem among microcontrollers. Not surprisingly, a wide variety of devices that use serial protocols to communicate have been developed.

Serial communication can be done over a single wire, although two, three, and four-wire interfaces are also common. The protocols used can be broadly characterized as synchronous and asynchronous. A synchronous protocol uses some type of clock to synchronize the transmitter and receiver. Synchronous systems include Serial Peripheral Interface (SPI) and Inter-Integrated Circuit (IIC). In contrast, asynchronous protocols synchronize on some prearranged signal, typically a start bit. Common RS-232 ports, like those on the back of a PC, use asynchronous data transmission.

Synchronous Serial Data

Typical synchronous protocols use at least two lines, one for data and one for the clock signal. The receiver reads the data at the rising or falling edge of a clock pulses it sends to the transmitter. Often, the transmitting device clocks data in one pin and out another pin allowing an arbitrary number of devices to be daisy chained. Synchronous protocols allow high data rates but require multiple wires to work. Still many devices like A/D converters, EEPROMs, and other peripherals utilize this of protocol.

Asynchronous Serial Data

Asynchronous serial data is the more common of the two arrangements. The transmitter and receiver are set for the same transmission speed. The receiver then watches for a “start bit” and uses it to synchronize with the transmitter. As an example, suppose a serial data transmission consists of a start bit, 8 data bits, and one stop bit at 9600 bits per second (bps). To squeeze 9600 bits into a second, each bit can only be transmitted for:

$$T_{bit} = \frac{1}{9600\ \text{bps}} = 104\ \mu\text{s} \quad (2)$$

The transmitter and receiver must also be agreed on the signal that gets transmitted between bytes, the idle state. The start bit begins when the transmitter switches its signal out of the idle state. For example, if 1 is the idle state, as soon as the transmitter switches to 0, the 104 µs start bit has begun. When the receiver senses the start bit, it knows that 104 µs later the first bit of data will be transmitted. So, the receiver checks the state of the signal after 104 µs and records the value of the first bit. It repeats this sampling process eight more times, once for each of the eight data bits. The stop bit is somewhat of a misnomer since the state of the stop bit is the same as the line's idle state. The stop bit is actually the minimum idle time before the next byte can be transmitted. Modern systems often use 1 stop bit, that is, 1 bit period between bytes. Some older systems required 1.5 or even 2 stop bits.

RS-232C is by far the most common asynchronous serial protocol. Personal computer serial ports use this scheme. In fact, connecting a microcontroller to a PC is a common use for RS-232. Other devices, including specialized serial LCDs, PWM coprocessors, and PS/2 keyboard interfaces also use RS-232.

A typical RS-232 setup requires one line for each transmitter and one for each receiver. Some systems will share a single line for both transmitting and receiving. Additional lines used for flow control are also common. Flow control lines allow the receiver to send a signal that indicates when to send the next byte. Commonly referred to as handshaking, the receiver has to signal its willingness to receive before the transmitter can send.

RS-232 Practical Considerations

RS-232 is more than just an arrangement of bits. The standard also calls for particular connectors and voltage levels. This can be a problem for designs incorporating microcontrollers because the RS-232 signal varies between –12 V to transmit a 1 and +12 V to transmit a 0. Microcontrollers, of course, use the standard TTL/CMOS 0 and 5 V signals.

A variety of techniques can be used to convert from TTL to RS232 voltages and visa versa. Peripheral integrated circuits that make these conversions are often added to the design. The classic chips to do this are the 1488 and 1489 line drivers and receivers. However, these require a +/- 12V power supply, common in computers, but not so common is smaller electronic designs.

In many cases, the only reason to have +/-12V is for RS-232. In this case, the need for +/- 12 V can be eliminated all together with a MAX232 or MAX233 IC from Maxim. These clever chips convert TTL to RS232 using only a single 5V supply. The MAX232 and 233 generate their own 12 V supplies using internal "charge pumps". The actual voltage won't be exactly +/- 12 V, but it will be well within the RS-232 specification. The MAX232 uses a few external capacitors, but the MAX233 requires no external capacitors.

It is possible to connect a TTL output directly to an RS-232 input. It works most of the time, but it's only recommended for lab and prototyping situations, not for production designs. The only thing to keep in mind is that 5 V is interpreted as a 0 while 0 V is interpreted as a 1. An RS-232 output can also be connected to an SX

input, so long as a current limiting resistor is used. A 22 kΩ resistor, for example, can be placed in series between the RS-232 output and the SX input. The SX has internal diode protection that clamps voltages above 5 V and below 0 V. The resistor prevents possible circuit damage that can occur when these diodes conduct excessive current in an attempt to keep the voltage clamped. Keep in mind that the same logic inversion that occurs when sending serial RS232 data without a line driver also occurs when receiving without a line driver.

Summary

A variety of designs feature devices with voltage or current requirements that are higher than the SX chip can supply. External transistors can be selected to drive these loads, then the SX can be used to switch the transistors on and off. Input voltages can also exceed the 0 to 5 V range. For the sake of sensing when a voltage passes a particular threshold, a voltage divider can be used to trim the measured input so that it crosses an SX I/O pin's logic threshold.

When using the SX to communicate with a parallel device, such as the LCD with assembly code example introduced in this unit, masking may be necessary to make sure that outputs not used by the parallel device are unaffected. Serial devices are a common solution for reducing the overall number of microcontroller I/O pins dedicated to each peripheral device. Synchronous and asynchronous serial communication are the two most common timing schemes used for serial communication.

RS232 is a common standard for asynchronous serial communication, and it uses +/- 12 V. Although the SX can send TTL signals directly to an RS232 input and receive RS232 signals via a series resistor, this connection scheme is only recommended for experimentation. Specialized RS232 line driver, receiver, and transceiver ICs can be used for much more foolproof communication between SX and RS232 I/O.

Exercises

1. Which of the following is a characteristic of asynchronous communications?
(a) An external clock signal
(b) Bits take a variable amount of time
(c) Each byte begins with a start bit
(d) The transmitter sends 1, 1.5, or 2 bits at once

2. A sensor emits 0V when off and 3V when on. What techniques could you use to read it with an SX? (Select all that apply)
(a) Read the value directly with CMOS input thresholds
(b) Use a 2N2222 transistor to switch on when the signal is present
(c) Use a voltage divider with two resistors
(d) Use an external A/D converter

3. Which of the following is a characteristic of RS-232?
(a) RS-232 uses the same line for transmitting and receiving
(b) RS-232 does not require transmitter and receiver to agree on speed
(c) All bits in an RS-232 byte require the same amount of time to send
(d) Real-world RS-232 devices use positive and negative voltages to indicate 0s and 1s

Answers

1. (c) is the correct answer. Each byte begins with a start bit used to synchronize the receiver.
2. (a) and (b) are correct. Although you might argue that (d) would do the job, there is no need to measure the precise voltage of the sensor; only two voltages are required. Directly connecting the sensor to an SX pin would work, although the circuit will be more prone to noise errors than if you use method (b).
3. (d) is the correct answer. Although most bits require the same time to send, stop bits may be longer than 1 bit, so (c) is not correct.

Rev1.

Unit II. A Software UART – The Transmitter

Unit II from I/O Control the SX Microcontroller

Asynchronous serial data is very popular in the real world. Modems, terminals, mice, and printers can all use RS-232 ports to communicate with a variety of computers. Because of this popularity, single ICs that could handle RS-232 communications arrived on the scene even before microcontrollers became popular. These chips were called UARTs (for Universal Asynchronous Receiver and Transmitter).

In this course, you'll build a variety of software-only UARTs using the SX's speed to simulate a UART and still leave time for your actual program. In this unit, you'll examine the transmitter portion only. To avoid confusion, I'll continue to refer to a UART, even though in this unit the code only transmits.

Sometimes transmitting is all you need. For example, suppose you have a remote weather station that should send the temperature, wind speed, and wind direction to a remote receiver. This system may not require a receiver. It simply broadcasts its data to whoever is listening on the other end.

UART Transmission Logic

There are a few things you need to think about when designing a serial transmitter:

- What state is the line in while idle?
- How long should each bit last?
- How many bits are transmitted?
- Does the least-significant bit appear first or last?
- How long is the minimum idle between characters (the stop bit)?

For RS-232 many of these things can't change. For example, you send bits least-significant first. The baud rate corresponds to the number of bits per second, and therefore, the length of each bit is the reciprocal of the baud rate. So at 9600 baud, for example, each bit's period is 1/9600 or about 104 microseconds. The receiver and the transmitter agree on the minimum length of the stop bit and this is usually the same as the bit period.

The only remaining question then is what state is the line in while idle? This varies depending on the hardware design. If you are using an inverting line driver (like a MAX232), the line should be high when idle. If you are connecting directly to an RS-232 receiver (which, as mentioned earlier, is not always going to work) the line should be low for an idle. Below is an RS-232 transmission of an ASCII "A" character (%01000001). Notice the bits are inverted and the least-significant bit is first.

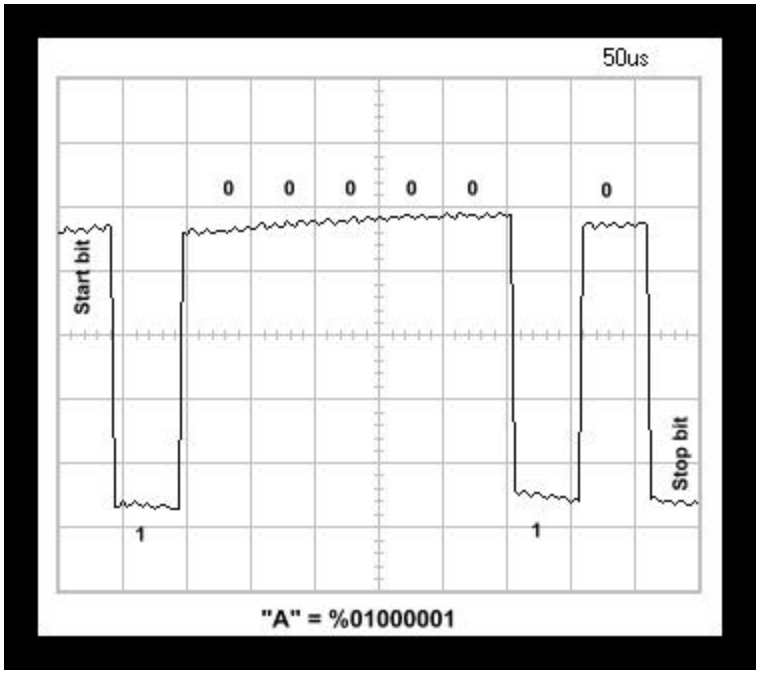


Figure II.1 – RS-232 Transmission

Creating the Code

SX Web sites have several UART routines. Actually, one of these implements 8 19.2K UARTs! Another example allows you to configure it to operate between 2400 and 230.4K baud.
That's a bit of overkill for this application. However, there is no shortage of examples to study.

One approach would be to use part of your ordinary program to directly manipulate the output port. This would work, but it would also tie up your program for the entire duration of the byte you wanted to send. It would also prevent you from sending characters while anything else was happening.

A better idea is to send the bits from within an interrupt service routine (ISR). You can set up a periodic interrupt that is faster than the bit rate and do all the work during the ISR. This makes even more sense when you consider that to receive serial data (the next logical step) you'll almost have to use interrupts unless you plan to do nothing but wait for the input's start bit.

You can find a simple UART transmitter in the section at the end of this unit entitled *The Transmitter Code*. This UART is fixed a rate of 19.2Kbaud (19,200 baud) and directly drives an RS-232 receiver with 8 bits and no stop bit.

When the main program wants to send a character, it calls **send_byte** with the character in the **w** register. This routine loads the character into the top 8 bits of the 16-bit transmit register (**tx_high** and **tx_low**). In reality, the code only uses 10 bits of the register since only **tx_low.7** and **tx_low.6** make any difference. The **send_byte** routine clears the top bit (bit 7) of **tx_low** – this corresponds to the start bit. The ISR will invert the bits, so a 0 will represent a high start bit.

Finally, **send_byte** sets the **tx_count** variable to 10. This is the bit count; 8 bits + 1 start bit + 1 stop bit. The routine, by the way, waits for **tx_count** to be zero to prevent overwriting an output byte in progress.

All the real work occurs in the interrupt routine. The first section examines **tx_count**. If this variable is zero, no transmission is pending, and there is no reason to do any further processing.

The second section simply decrements a counter (**tx_divide**) by 1 and if the counter is not zero, the ISR returns immediately. This has the effect of dividing the interrupt rate by 16. Of course, you could program the interrupt to occur once per bit period, but this method allows you to easily change the baud rate. For example, setting the division rate (**txdivisor**) to 32 will result in a 9600 baud speed. If you need 4800 baud you could set **txdivisor** to 64. You'll read more about baud rate calculations in the next section.

If it is time for a new bit, the ISR shifts the 16-bit transmit register to the right one place. Before it does this, it sets the carry bit. This will ensure that the final bit (or bits) will be high – just what you need for the stop bit (since the output is inverted). The output bit, represented by **tx_low.6**, is written out (inverted) to the I/O port. The **tx_count** variable, of course, is decremented. Shifting right means the least-significant bits go out first, as required by RS232.

Once the bit is written, the ISR is done, so it exits, scheduling itself to run again 163 clock cycles after the last interrupt. The main code spends most of its time waiting for **tx_count** to drop to zero (in the **send_byte**) routine so that it can send the next byte. Of course, a real program would probably have much more work to do while the ISR is sending data.

Calculating Baud Rates

Calculating the baud rate can sometimes seem like a black art, but with a little thought, it isn't too difficult. The SX, in this case, is running at 50Mhz, which corresponds to 1/50000000, or 20nS per clock cycle. The ISR will execute every 163 clock cycles or 3.26uS. Finally, the ISR only executes every 16 interrupts, so the code runs every 52.16uS. The desired baud rate is 19200 bits per second, which is 1/19200 or 52.08uS. The 52.16uS period is only off by 0.15% -- close enough for practical purposes.

Obviously, you can alter this equation to suit your needs. Suppose you want to run the SX at 10MHz instead of 50Mhz and work at 9600 baud? This lower clock frequency would reduce power consumption, but it will also require you to recalculate the interrupt rates.

Each clock cycle is 100nS. The total bit time is about 104.2uS. Dividing 104.2uS by 100nS tells you that each bit will require 1042 clock cycles. Of course, you can only program the timer with an 8 bit number, so you can't program the timer to directly interrupt every 1042 clock cycles.

If you select a timer rate of 50 cycles, the interrupt will occur every 5uS (handy later generating a real-time clock). The interrupt divisor can then be 21. This, of course, is not exactly correct (it should be 20.84). Is this too far off?

To determine this, reverse the calculations to find out the true bit time: 21 x 5uS is 105uS, and error of only 0.77%. This is well within the tolerance of any real-world device.

When selecting these values, you need to consider how many clock cycles your ISR requires to execute. In this example, the interrupt will occur every 50 clock cycles. If the ISR requires 50 clock cycles or more to execute, you'll have a problem. Even if the ISR approaches 50 clock cycles, you may not be able to use the numbers you calculate. Why? Suppose the ISR requires 40 cycles. This leaves only 10 cycles out of 50 to process your main program! So in 5uS, the ISR will use up 4uS, and the main code can execute for 1uS.

If you run into this problem, you can adjust the clock period up and the divisor value down. For example, 75 cycles in the last example results in a 7.5uS interrupt time. With a divisor value of 14 this leads to a 105uS bit period (off by less than 1%).

The simple transmitter code only requires 21 cycles (maximum) so in this case 50 cycles between interrupts is plenty. Also, most of the time the ISR only require 9 or 11 cycles so there is plenty of time left over for the main program.

Configuration

The program at the end of this unit simply transmits "ABC" repeatedly as fast as possible. The data bit is inverted so you can just directly connect the output pin (**RA.3**) to a PC's serial input. If you are using a DB9 connector, attach the DB9's pin 2 to the SX's **RA.3** pin. You'll also need to connect the DB9's pin 5 to a common ground (Vss) on your SX-Tech board.

What if you wanted to use a serial line driver (like a MAX232, for example)? You'd need to stop inverting the data output. The actual output operation occurs in this line of code (found just above the **noisr** label):

```
            movb  tx_pin,/tx_low.6        ; output next bit
```

The slash character indicates that the SX should invert the bit before writing it to **tx_pin**. You'll notice that near the top of the program, **tx_pin** is set to equal **ra.3**. This allows you to easily configure the program to use a different pin. Of course, if you change the port assignment, you'd need to change the initialization of the port registers too. For example, if you wanted to use **ra.0**, you'd also need to change the initialization code from:

```
reset_entry mov    ra,#%0000        ;init ra
            mov   !ra,#%0111
```

to:

```
reset_entry mov    ra,#%0000        ;init ra
            mov   !ra,#%1110
```

Of course if you wanted to use a pin on port B or C you'd have even more changes to make.

If you wanted to handle a line driver, you could remove the slash on the **movb** command so that it read:

```
            movb  tx_pin,tx_low.6         ; output next bit
```

Of course, you'd also want to change the initialization code to:

```
reset_entry mov    ra,#%1000        ;init ra
            mov   !ra,#%0111
```

Since the idle state of the line is high when using a driver.

Obviously, making changes involves a lot of trouble. This is where the SX Key's macro capabilities can be very handy.

For example, consider the inverted bit change. You could define a single symbol near the top of the program that controls the inversion:

```
linedriver  equ   0     ; 1 if using line driver
```

Then in the remainder of the code, you can use **IF** to selectively assemble different code. For example:

```
IF linedriver=0
      movb  tx_pin,/tx_low.6        ; output next bit
ELSE
      movb  tx_pin,tx_low.6         ; output next bit
ENDIF
```

Of course, you'd have to wrap each change with an **IF** statement. Keep in mind, this does not perform the logic at run time. It makes the comparison during assembly. This causes the assembler to only process one statement or the other. In this case, there is only one statement, but you can place as many statements as you like between the **IF** and the **ELSE** and the **ELSE** and the **ENDIF**. You don't have to use the **ELSE** statement if you don't want an alternative block of code. You can even nest one **IF** inside another:

```
IF someoption = 1
    mov w,#100
    IF anotheroption = 1
       mov avar,w
    ELSE
       mov bvar,w
    ENDIF
ENDIF
```

Another way to use **IF** is to use **IFDEF** and **IFNDEF**. Using these instead of **IF** allow you to test if a symbol is defined (or not defined in the case of **IFNDEF**).

Tip: You may have noticed that when a program sets a symbol value, it might use the **equ** directive, or it might use an equal sign (=). For example:

```
somevalue equ 100
```

or:

```
somevalue = 100
```

These statements do the same thing, with one important difference. Once you use **equ** you can't change the value of the symbol later. When you use the equal sign, you can decide to change the value later. For the purpose of these programs, **equ** is probably the best bet, but it doesn't make much difference. However, when you construct macros, you might want to change the value of the symbol as part of macro processing. Then you'd avoid using **equ**.

Testing The Transmitter

If you enter the code listed under *The Transmitter Code* at the end of this unit, you should be able to run it with the SX-Key's Run command. Connect **RA.3** to pin 2 of a DB9 connector and **Vss** to pin 5 of the connector. Then use a normal 9-pin serial cable to connect the DB9 connector to a free serial port on your PC. You should use a serial port that is not otherwise in use. Also, on most PCs, you can't use COM1 and COM3 or COM2 and COM4 at the same time.

You can use any terminal program to see the results. If you are using Microsoft Windows, you can use the Hyperterminal program. Simply create a new connection that uses the serial port you've used to connect to the SX. Make sure to select 19200 baud, 8 bits, 1 stop bit, no parity, and no handshaking, as in Figure 11.2.

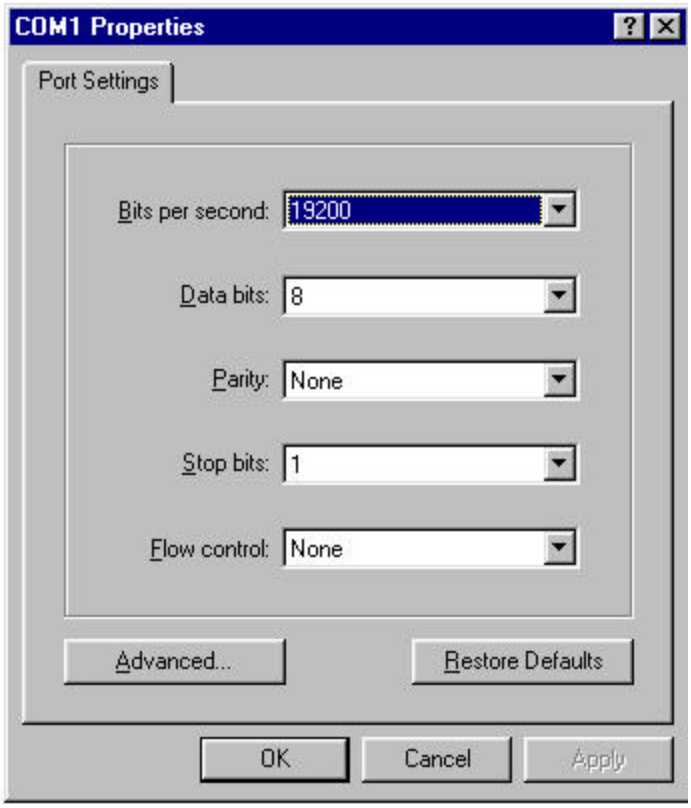


Figure 11.2. HyperTerminal Setup

You should observe the characters on the terminal window's screen. Troubleshooting serial problems is always tricky, but here are a few things to look for:

- If the terminal program complains that there is an error, you have no hope of anything working. You'll first need to find a free port, or close software using the port already.
- You should use a straight cable (or connect to DB9 pin 3 if the cable is crossed). You can determine if the cable is straight by measuring the pins with an ohmmeter.
- As mentioned before, the baud rate and other parameters must match exactly.
- Make sure the DB9's ground pin (pin 5) is connected to the same ground as the SX-Tech board.
- It is possible that the PC you are using will not accept RS-232 levels of 0 and 5V. If this is the case, try another PC if possible. You can also use a line driver like the Maxim MAX232. Virtually all modern desktop computers will work without a line driver. Laptops seem more questionable, but even then, most will work.

Debugging ISRs

Once you have the code running you might be tempted to use the SX's debugging capability. You can do this of course, but there are a few things you should know. First, the ISR will not work properly while debugging. After all, the whole premise that the serial transmitter operates on is that an interrupt will occur at a regular period. When you stop at a breakpoint, this upsets that assumption.

Of course, if you let the SX run at full speed under the debugger, the transmitter will work. Then you can't really peek into its execution very well. If you are trying to see what happens inside the ISR, the best idea is to place a breakpoint in the ISR code and let the processor run. Of course, the ISR's timing will be thrown off, but you can reliably see the flow of execution.

If you are stepping through non-interrupt code, don't be surprised if you suddenly find yourself inside the ISR (this happens when an interrupt occurs). If you don't want to step through each line of the ISR, simply place a breakpoint on the **RETIW** instruction and then step from there. Either way, the timing of the interrupt routine will be affected.

Summary

A serial transmitter, while useful in its own right, is only half of the story. While some devices only transmit, most will want to transmit and receive. In the next unit, you'll examine a case where transmitting data is sufficient. Later, you'll see how to handle serial data reception and then marry the two pieces to create a true software UART.

The Transmitter Code

```
; 19.2K RS232 transmitter
;
;
            device      sx28l,stackx_optionx
            device      oscxt5,turbo
            freq        50000000
            reset       reset_entry
;
;
; I/O definition
;
tx_pin            =     ra.3
;
;
; Variables
;
            org   8

temp        ds    1


            org   10h
serial            =     $

tx_high     ds    1
tx_low      ds    1
tx_count    ds    1
tx_divide   ds    1
txdivisor   equ   16   ; 16 periods per bit

            org   0
;
;
; Interrupt routine - UART
;
interrupt
            bank  serial
            test  tx_count                ; busy?
            jz noisr                      ; no byte being sent
```

```
            dec   tx_divide
            jnz   noisr
            mov   tx_divide,#txdivisor    ; ready for next
            stc                           ; ready stop bit
            rr    tx_high                 ; go to next bit
            rr    tx_low
            dec   tx_count                ; count-1
            movb  tx_pin,/tx_low.6        ; output next bit
noisr
            mov   w,#-163           ;interrupt every 163 clocks
            retiw
;

;***************
;
;
; Send byte via serial port
;
send_byte   bank  serial

:wait       test  tx_count          ;wait for not busy
            jnz   :wait

            mov   tx_high,w
            clrb  tx_low.7          ; set start bit
            mov   tx_count,#10      ;1 start + 8 data + 1 stop bit
            ret

reset_entry mov    ra,#%0000        ;init ra
            mov   !ra,#%0111

            clr   fsr               ;reset all ram banks
:loop       setb  fsr.4
            clr   ind
            ijnz  fsr,:loop
            mov   tx_divide,txdivisor
            mov   !option,#%10011111

; **** Your code goes here ****
xloop
            mov w,#'A'
            call send_byte
```

```
mov w,#'B'
call send_byte
mov w,#'C'
call send_byte
mov w,#13
call send_byte
mov w,#10
call send_byte
jmp xloop
```

Exercises

1. After you have the transmitter code working, alter it so that it operates at 20MHz and works at 9600 baud. Calculate the error your code will have compared to the ideal as a percentage.

2. Use equates to set the interrupt period so you can easily change it from its default value of -163.

3. Use equates and the **IF** directive to allow you to select the baud rate using a line like this:
```
baudrate        =       9600
```

Answers

1. There are many possible answers to this question. Changing the interrupt divisor from 16 to 13 would work (without changing the –163 in the ISR). This results in a bit period of 105.95uS, and error of about 1.4% -- a bit high but probably acceptable for most devices. Changing the –163 to –80 and setting the interrupt divisor to 26 results in 104uS, an error of less than 0.5%. Your answer should use an interrupt period high enough to allow processing and less than 255.

2. Simply add this line near the top of the file (after the **txdivisor** value is set is a good spot):
   ```
   isrperiod equ –163
   ```
   Then you also have to modify the line before the **iretw** statement to read:
   ```
   mov w,#isrperiod
   ```

3. There are several ways you could do this. Here is one example (assuming a 50MHz clock):
   ```
   baudrate          =     9600
   IF baudrate = 19200
   isrperiod   equ   -163
   txdivisor   equ   16
   ENDIF

   IF baudrate = 9600
   isrperiod   equ   -163
   txdivisor   equ   32
   ENDIF

   .
   .
   .
   ```


Rev1.

3

Unit III. Analog Input

Unit III from I/O Control the SX Microcontroller

The SX is, of course, a digital device. The classic way to interface an analog input to a digital device is to use an Analog to Digital converter (ADC or A/D). This is certainly possible with the SX. Many vendors make suitable ADCs that connect using some type of serial connection. There are also many ADCs that use parallel connections, but these take many pins and are usually less suitable for use with the SX.

However, because of the SX's speed and special features, you can perform analog input using just two resistors and a capacitor. Does that seem to good to be true? Well, there are some limitations to this technique, but in general you can make the SX read an analog voltage in this way.

The Simple ADC

Here is the circuitry required to form the simple ADC:

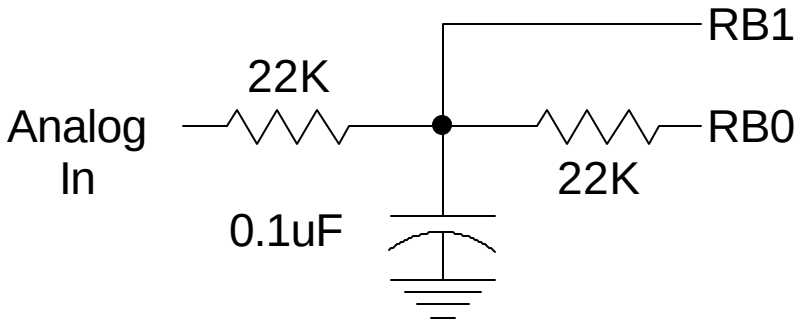


Figure III.1 The ADC

In this case, the input voltage is supplied by the potentiometer, which functions as a simple voltage divider. You could consider this technique one way to measure the position of a potentiometer, although it is really reading the voltage level developed at the junction of the resistors and the capacitor.

At first glance this doesn't seem likely to make an ADC. How does it work? The answer lies in two features of the SX. First, the SX can select a CMOS input threshold mode for input pins. In this mode, the input sees 2.5V as a 1 and anything below that to be a 0. The second feature this scheme relies on is sheer speed. In the schematic, RB0 is an output and RB1 is an input. The SX, via a periodic interrupt, modulates the output pin so that the input (RB1) hovers around the 2.5V threshold. Along the way the program counts how often the capacitor has charged

up past 2.5V and required a discharge. After 255 cycles, this count will be proportional to the voltage (as a percentage of 5V). So a 5V input will read 255 counts. A 2.5V input should read 128 counts.

Here is the basic logic written in pseudo code:

1. Read the input bit
2. Invert the input bit
3. Write the inverted input to the output
4. If the output was 0 (capacitor discharge), add 1 to the voltage count
5. Add 1 to the cycle count
6. If 256 cycles have elapsed (the count is 0), copy the result, set a flag, and zero the voltage count

The code does not explicitly repeat because it executes during a periodic interrupt (much as the UART did in the last unit).

If you take a minute to study the code in this easy-to-understand form, you can discern its operating principle. The processor tries to reverse the state of the input on each cycle. The number of discharge reversals is proportional to the input voltage. Consider the two extreme cases. If the input is stuck at 0V, the SX will never charge the capacitor, and will never need to discharge it. Therefore, the count should be 0. If the input is at 5V, the SX will never successfully discharge the capacitor and will try on each cycle leading to a count of 255. If the input is 2.5V, you'd expect it to alternate between charging and discharging leading to a count of 128 since the code will only count up on alternate cycles.

In real life, the result will not be the same each time. The last bit or two will tend to shift back and forth and small imprecisions in the circuit elements will create small variations in the result. Still, for such a simple circuit the accuracy isn't bad and the value is quite useful for many applications.

Writing the Code

Implementing the A/D in software isn't that hard once you have the idea. Of course, during initialization you must set one pin to an input and the other to output. You also have to set the input threshold to CMOS by manipulating the I/O port option register. Assuming you want to use RB0 and RB1 for the A/D (and you don't care about the rest of port B) you could use this code:

```
clr   rb                ;init rb
mov   !rb,#%00000010
mov   m,#$D             ;set cmos input levels
mov   !rb,#0
mov   m,#$F
```

You can find the complete code at the end of this unit. Setting the **m** register to $D allows you to set the threshold options. Clearing **!rb** sets the CMOS input level. Setting **m** back to $F is a good idea so you don't forget later in your program that the **!rb** register doesn't have its usual properties.

3

The interrupt routine follows the outline of the pseudo code:

```
      bank  analog

; shifting moves the input bit to the output bit
            mov   w,>>rb            ; read capacitor level
            not   w                 ; invert
            and   w,#%00000001      ; write to output
            mov   port_buff,w
            mov   rb,w              ; and update pins

            sb    port_buff.0
            incsz adc0_acc          ; if it was high, inc acc
            inc   adc0_acc
            dec   adc0_acc          ; inc/inc/dec prevents rollover
            inc   adc_count         ; done (8 bits)?
            jnz   adc_out
; Done so store result
            mov   adc0,adc0_acc
            setb  complete.0        ; set complete flag
; clear for next pass
            clr   adc0_acc
; standard UART transmit
      .
      .
      .
```

The interrupt routine continuously measures the input. When it completes 256 cycles (indicated by the **adc_count** variable) it sets the **complete** flag and copies the result (in **adc0_acc**) to **adc0**. This allows the interrupt routine to continue with the next calculation while the main program reads the previous value. Here is an excerpt from the main program:

```
:wait       jnb   complete.0,:wait   ; wait for data ready
            mov   w,adc0
            clrb  complete.0         ; set up to wait again
```

Mixing Interrupt Routines

The example program reads the analog input value and converts the raw hexadecimal value to 2 ASCII characters. It then uses the UART transmitter from the last unit to send this value to a PC. Each measurement ends with a carriage return. You can view the output with any terminal program (for example, Hyperterminal as used in the last unit). Of course, if you can write PC programs you could also write a custom program to post process and store the data.

This example uses the interrupt routine for analog conversion along with the UART transmitter routine. When you mix routines you have to consider several important factors:

1. Can the routines share an interrupt period?
2. Does either of the routines take a constant time to execute?
3. Does one or more routines need a precise period?
4. What is the total execution time of the two routines?

If you can adjust the routines to use the same interrupt period, you'll have less trouble. However, this isn't always possible. Sometimes you can set the interrupt period to a fast time and use counters to divide the time for the routines that need it. For example, suppose one interrupt routine needs to execute every 300uS and the other needs to execute every 500uS. You might consider setting the interrupt period to 100uS and use a counter to allow the first routine to execute on every third interrupt and the second routine to execute on every fifth interrupt.

The other concern is how precise do you need the timing for each routine? Suppose you set the interrupt to occur every 200uS. The first routine takes somewhere between 300nS and 700nS to execute. Then the second routine will not necessarily run every 200uS.

As an example, try an example using some numbers that are easier to work with (although unrealistic). Suppose your interrupt occurs every 10 seconds. Further suppose that routine A usually takes 1 second to execute. However, every third interrupt, routine A requires 3 seconds. Routine B always takes 1 second to execute. Finally, imagine that the first interrupt occurs when your mental stopwatch begins (T=0). Here is how your imaginary system would work:

T	Action	Elapsed Time
0	Routine A	N/A
1	Routine B	N/A
10	Routine A	10
11	Routine B	10
20	Routine A	10
23	Routine B	12
30	Routine A	10
31	Routine B	8
40	Routine A	10
41	Routine B	10

You can see that routine B will not run every 10 seconds as you'd expect. Since your program normally sees errors in the micro or nanosecond range, this may not be a problem. The program for this unit, for example, can easily tolerate a small error in the RS-232 bit rate. However, the A/D code is less accurate if the time period is inexact. That's why the A/D code appears first in the interrupt handler.

Sometimes you can write your code so that it takes a constant amount of time to execute. For example, consider this code:

```
      jz    intb
      inc   ctr1
intb
```

If the jump is not taken, this code requires 3 cycles to execute. If the jump is taken, it requires 4. You could compensate for this by rewriting the code:

```
      jz    intb
      inc   ctr1
      nop
intb
```

Now the code requires the same amount of time to execute no matter what. The **nop** instruction just wastes an instruction cycle. If you need to waste three cycles, you can save some space by using **jmp $+1**. This instruction effectively does nothing but wastes three cycles instead of just one.

If you need to write lots of **nop**s you can use the **REPT** directive. This is an instruction to the assembler that allows you to repeat a sequence of instructions. For example:

```
      REPT 10
      NOP
      ENDR
```

This inserts 10 **nop** instructions into your code. You can use the per cent character (%) to return the current repeat number (starting with 1). So to insert a table with the numbers 1 through 5 in it you could write:

```
table5
      dw 1
      dw 2
      dw 3
      dw 4
      dw 5
```

Or you could write:

```
table5      REPT  5
            dw %
            ENDR
```

If you wanted the numbers 0 to 4 instead, you'd use **dw %-1** in the middle of the **REPT** block.

The **REPT** block is one place where you have to be careful wit h labels. Suppose you wanted to repeat a 3 cycle **nop**. You might write:

```
      REPT 10
      jmp here
here
      ENDR
```

This makes sense, but it fails because it defines the **here** label 10 times. Even local labels won't work. Instead, use **$** to reference the current location:

```
      REPT 10
      jmp $+1
      ENDR
```

You could also use this form, but it isn't as elegant:

```
here        ; must be on a separate line
      REPT 10
      jmp here+%
      ENDR
```

Hex Conversion

The hex conversion routine might need a little study before it becomes clear. The **send_hex** routine stores the number in **number_low** so it can retrieve the value later. Notice this instruction:

```
mov   w,<>number_low          ;send first digit
```

This swaps the two four-bit halves of **number_low** and stores the result in **w**. So if the original number was $A1, **w** now contains $1A. The program then calls **:digit** which isolates the bottom four bits and converts it to ASCII (more on that routine later).

Once **:digit** is complete, the program reloads **w** from **number_low** and then just drops into the **:digit** routine. This is a special form of a technique known as the *hidden return*. It makes your code somewhat harder to read, but it saves valuable program space.

In your program, you can use the hidden return by spotting places where you have code that looks like this:

```
call b
ret
```

Since routine **b** must end in a **ret** instruction, you can replace these two lines with a single **jmp b** instruction. The hex conversion routine takes this idea one step further. By positioning the **b** routine at this spot in the program, you can eliminate both lines of code. Any other part of the program that calls **b** doesn't really care where it is located. Don't forget that the **SX** call instruction does require you to keep your subroutines in the first half of each page, however.

Table Lookup

The **:digit** routine uses the **iread** instruction to lookup the correct ASCII character. The **iread** instruction retrieves a value from the SX's program memory. The SX has enough memory space that a single byte can't address it all, so the **iread** instruction forms an address using the **M** register and the **w** register. So if you want to read location $200, you'd set **M** to 2 and **w** to 0. Of course, it is a good idea to restore **M** to its default value when you are done.

The **M** register is 4 bits wide, so you can form a 12-bit address. The resulting word is also 12-bits wide and **iread** returns the result in the **M** and **w** registers. In this case, the program is only interested in the byte result, so it discards what is in **M**.

The **iread** instruction is somewhat expensive (4 cycles in turbo mode). There is another way you can create a table – using the **retw** command. Suppose you want to construct a table that has the square of a number. You could write a subroutine like this:

```
lookup2     jmp   PC+W
            retw  0
            retw  1
            retw  4
            retw  9
```

You could extend this to any number of entries. Now when you call **lookup2**, the value in the **w** register causes a jump to the correct return statement. The assembler will also let you put the values together as in:

```
retw  0,1,4,9
```

A Word about Input Impedance

If you do some serious measurements with the A/D converter presented in this unit, you will find that the results may not match what you expect. The problem is that the input resistors set the circuit's input impedance, which is relatively low (for practical purposes, 11K – the value of both resistors in parallel). You can combat this somewhat with higher-value resistors, but at some point, it becomes too difficult to charge and discharge the capacitor, so accuracy suffers again.

If all you care about is measuring the position of a potentiometer or a relative voltage, you probably don't care. For serious work, however, you'd want to use an op-amp buffer. Any general-purpose op-amp (for example, a 741) could be connected as a non-inverting amplifier and would present a very high input impedance to the circuit. This would improve accuracy considerably. Just remember that most op-amp circuits require positive and negative voltages higher than the voltages they have to handle (for example, + and – 12V supplies are common).

Tip: In Unit VIII, you'll find out how to make another type of A/D converter that uses the SX's built-in comparator and, therefore, provides excellent input isolation. Of course, the SX only has a single comparator, so you can only use this technique for a single channel. The method in this unit could be replicated to provide multiple channels of input.

The Complete Code

```
; Simple A/D Converter
;
; Device
;
            device      sx28l,stackx_optionx
            device      oscxt5,turbo
            reset reset_entry
;
;
; Equates
;
tx_pin            =     ra.3
adc0_out_pin      =     rb.0
adc0_in_pin       =     rb.1
;
;
; Variables
;
            org   8

temp        ds    1
```

```
number_low  ds    1
complete    ds    1   ; bit 0 = 1 when complete
; holding for voltages
v0          ds    1


            org   10h
serial            =     $

tx_high     ds    1     ;tx
tx_low      ds    1
tx_count    ds    1
tx_divide   ds    1
txdivisor   =     16   ; 16 periods per bit

            org   30h
analog            =     $

port_buff   ds    1     ;buffer - used by all

adc0        ds    1     ;adc0
adc0_acc    ds    1

adc_count   ds    1     ; count for both ADCs


            org   0
;
;
; Interrupt routine – ADC + UART
;
interrupt
            bank  analog

; shifting moves the input bit to the output bit
            mov   w,>>rb            ; read capacitor level
            not   w                 ; invert
            and   w,#%00000001      ; write to output
            mov   port_buff,w
            mov   rb,w              ; and update pins

            sb    port_buff.0       ; adc0
```

```
            incsz adc0_acc          ; if it was high, inc acc
            inc   adc0_acc
            dec   adc0_acc          ; inc/inc/dec prevents rollover
            inc   adc_count         ; done (8 bits)?
            jnz   adc_out
; Done so store result
            mov   adc0,adc0_acc
            setb  complete.0        ; set complete flag
; clear for next pass
            clr   adc0_acc
; standard UART transmit
adc_out
            bank  serial
            dec   tx_divide
            jnz   noisr
            mov   tx_divide,#txdivisor   ; ready for next
            test  tx_count           ; busy?
            jz    noisr              ; no byte being sent
            stc                      ; ready stop bit
            rr    tx_high
            rr    tx_low
            dec   tx_count
            movb  tx_pin,/tx_low.6   ;output next bit
noisr
            mov   w,#-163            ;interrupt every 163 clocks
            retiw
;

; required to output HEX numbers
_hex        dw    '0123456789ABCDEF'
;
;
;***************
;* Subroutines *

; Send hex byte (2 digits)
;
```

```
send_hex
            mov   number_low,w            ; save W
            mov   w,<>number_low          ;send first digit
            call  :digit

            mov   w,number_low            ;send second digit

:digit      and   w,#$F                   ;read hex chr
            mov   temp,w
            mov   w,#_hex
            clc               ; just in case +c is enabled
            add   w,temp
            mov   m,#0
            iread             ; read from program mem!
            mov   m,#$F

; fall into send byte

;***************
;
;
; Send byte via serial port
;
send_byte   bank  serial

:wait       test  tx_count          ;wait for not busy
            jnz   :wait

            mov   tx_high,w
            clrb  tx_low.7          ; set start bit


            mov   tx_count,#10      ;1 start + 8 data + 1 stop bit
            ret
reset_entry mov    ra,#%1000        ;init ra
            mov   !ra,#%0111
            clr   rb                ;init rb
            mov   !rb,#%00000010
            mov   m,#$D             ;set cmos input levels
            mov   !rb,#0
            mov   m,#$F
```

```
            clr   fsr               ;reset all ram banks
:loop       setb  fsr.4
            clr   ind
            ijnz  fsr,:loop
            mov   tx_divide,txdivisor
            mov   !option,#%10011111

; **** Your code goes here ****
top                     ; main loop
            bank  analog
:wait       jnb   complete.0,:wait   ; wait for data ready
            mov   w,adc0
            clrb  complete.0     ; get ready to wait again
            call  send_hex       ; write out
            mov   w,#13          ; send cr
            call  send_byte
            jmp   top
```

Summary

Although the SX is primarily a digital device, its speed allows it to handle certain analog quantities. Under the right circumstances, employing techniques like this can save money by eliminating the need for an inventory of special processors or dedicated A/D chips.

Along with analog conversion, this unit explored the **REPT** directive and some interesting ways to handle table lookups. The programs are getting more complicated and you'll find directives like **REPT** more useful as you build more sophisticated programs.

Exercises

1. Add a second A/D channel using port B2 and B3. Have the program send both values then a carriage return.

2. Set the baudrate to 300 baud by changing its interrupt period to 10432 clocks, but keep the A/D running at the same rate (163 clock cycles).

3. Optional: If you are familiar with a PC programming language, write a program that reads the values from the program, calculates the voltage and displays it. The solution uses QBASIC under MSDOS.

Answers

1. You must modify the code in several places to accomplish this task. First, you must set the correct pattern of I/O pins during initialization:

```
        mov     !rb,#%00001010
```

You'll also have to add corresponding lines to the interrupt routine:

```
            mov   w,>>rb            ; read capacitor level
            not   w                 ; invert
            and   w,#%00000101      ; write to output
            mov   port_buff,w
            mov   rb,w              ; and update pins

            sb    port_buff.0       ; adc0
            incsz adc0_acc          ; if it was high, inc acc
            inc   adc0_acc
            dec   adc0_acc          ; inc/inc/dec prevents rollover
            sb    port_buff.2       ; adc1
            incsz adc1_acc          ; if it was high, inc acc
            inc   adc1_acc
            dec   adc1_acc          ; inc/inc/dec prevents rollover
            inc   adc_count         ; done (8 bits)?
            jnz   adc_out
; Done so store result
            mov   adc0,adc0_acc
            mov   adc1,adc1_acc
            setb  complete.0        ; set complete flag
; clear for next pass
            clr   adc0_acc
            clr   adc1_acc
```

The lines with underlines beneath them are changes to the existing code. Of course, you also have to define the **adc1_acc**, and **adc1** variables. Finally, you can modify the main program:

```
top                     ; main loop
            bank  analog
:wait       jnb   complete.0,:wait   ; wait for data ready
            mov   v1,adc1   ; hold temporary v1
            mov   w,adc0
            clrb  complete.0     ; get ready to wait again
```

```
            call  send_hex       ; write out
            mov   w,v1
            call  send_hex
            mov   w,#13          ; send cr
            call  send_byte
            jmp   top
```

It is important to store the value in a temporary (the new **v1** variable) so that the two values are from the same measurement time. Without this new variable, it would be possible for the channel 1 value to change while you were writing out the value for channel 0. In this example, it doesn't make much difference. In real life, you'd probably want the two values to correspond to each other.

2. The easiest way to accomplish this is to put a 64x divider in front of the UART code using a new variable:

```
adc_out
      inc   x64
      jnb   x64.6,noisr
      clr   x64
bank  serial
```

This allows the A/D code to continue running at a 163 clock cycle period, but effectively only runs the UART transmitter every 10432 clock cycles. Since 19200 baud is 64 times 300 baud, the **txdivisor** value need not change. If the question had asked to move to, for example, 9600 baud, you could simply adjust the **txdivisor** value, but in this case the speed difference was too great to be held in a single byte.

3. Your solution to this problem will vary depending on what languages you have at your disposal. The following program uses QBASIC (this Basic comes with many versions of MSDOS and Windows – you can also find it in the Windows Resource Kit). It assumes the SX is attached to COM1 and is operating at 300 baud.

```
' Simple program to read a voltage
DIM c AS STRING
DIM v AS STRING
DIM eu AS SINGLE
COM(1) ON
ON COM(1) GOSUB ComHandler  ' go here when characters available
start:
' open com1 no handshaking, 32k buffer
OPEN "COM1:300,n,8,1,CD0,CS0,DS0,OP0,RS,RB32768" FOR INPUT AS #1
top:
  WHILE INKEY$ = "": WEND
  END
```

```
ComHandler:
  c = INPUT$(1, 1)    ' read character
  IF ASC(c) = 13 THEN     ' end of packet?
    IF LEN(v) <> 2 THEN   ' not a full packet?
      v = ""
      RETURN
    ELSE
' got a full packet so interpret it
     eu = VAL("&H" + v) * 5 / 256
     PRINT eu
     v = ""
     RETURN
    END IF
  END IF
  v = v + c   ' build up packet
  RETURN
```

This program uses a special feature of QBasic that allows the **ComHandler** routine to gain control whenever serial data is available (similar to an interrupt). Note that QBasic is not fast enough to reliably handle high baud rates.

When a character arrives, the program assembles it into a packet (this program assumes 1 byte per packet). When a correctly formed packet arrives (2 characters followed by a carriage return), the program performs this calculation:

```
eu = VAL("&H" + v) * 5 / 256
```

Here the **eu** variable (short for engineering units) receives a floating point value that corresponds to the estimated input voltage. The **VAL** function converts a string to a number (the **&H** prefix tells QBasic this is a hexadecimal number). Each count from the SX is worth 5/256V (roughly 19.5mV).

Unit IV. A Software UART – The Receiver

Unit IV from I/O Control with the SX Microcontroller

4

In Units 2 and 3, you worked with a software serial transmitter. This is half of a UART (Universal Asynchronous Receiver Transmitter). The next obvious step is to design and build a receiver. The transmitter is somewhat simpler than a receiver. Why? Consider that when transmitting you don't have to synchronize with anyone else – it is the receiver's job to synchronize with you.

Receiving is a bit more difficult. Instead of generating pulses of a specific width, you have to measure pulses. This wouldn't be so hard, except you must synchronize with the transmitter's start bit. This leads to some special considerations that are not necessary for the transmitter.

Fast Enough?

Each bit in a 9600 baud data stream occupies 104uS. So if you sample an input every 104uS, you can detect each bit, right? No! The problem is that timing on both sides of the system are not precise. If you sample right at the leading or trailing edge of a start bit, you are in danger of looking at the very edges of the bits and you might read one a shade to early or too late.

Ideally, you'd find the rising edge of the start bit and then delay 52uS. This would be approximately in the center of the start bit. Now the code can safely sample every 104uS (a total delay of 156uS) with reasonable certainly that each bit will be stable. With interrupts you can wait for the start bit in this way, but the SX's interrupt structure makes it challenging to handle multiple interrupt sources. You'll eventually want to integrate the transmitter and the receiver (among other things) and it would be handy if you could use one periodic interrupt as a basis for both.

When you sample at a regular interval, the Nyquist sampling theorem rears its head. This staple of signal processing theory states (among other things) that you have to sample twice as fast as the fastest signal you want to measure. So to find a 104uS pulse, you'll need to measure the input at least every 52uS. Even this isn't enough if you are planning to delaying 52uS to center the timing. You might catch the center of the pulse – to be safe, you should sample much faster, say 26uS or less.

Basic Logic

The receiver will use several variables. The **rx_count** byte tracks the number of bits to read (including the stop bit). When the receiver is idle, this variable will be zero. Another byte, **rx_divide**, counts the number of interrupt periods that correspond to a bit. The received byte is in **rx_byte** and a single bit, **rx_bit**, is set when the byte is ready. The receiver's logic on each interrupt is:

1. Read the input bit
2. If no byte is in progress, check for a start bit
3. If a start bit is present, load **rx_count** with 9 and **rx_divide** with 1.5 bit periods
4. If a byte is in progress, decrement **rx_divide**; if not zero, exit
5. Reset **rx_divide** to 1 bit period
6. Decrement **rx_count**; if zero (indicating a stop bit) set the **rx_flag** bit; if not zero, shift **rx_byte** to the right and merge the sampled input bit from step 1 into the least-significant bit

Here is the complete ISR:

```
                bank    serial
    movb    c,/rx_pin           ;serial receive
                test    rx_count
                jnz     :rxbit              ;if not, :bit
                mov     w,#9                ;in case start, ready 9
                sc                          ;if start, set rx_count
                mov     rx_count,w
                mov     rx_divide,#baud15   ;ready 1.5 bit periods
:rxbit          djnz    rx_divide,rxdone    ;8th time through?
                mov     rx_divide,#baud
                dec     rx_count            ;last bit?
                sz                          ;if not, save bit
                rr      rx_byte
                snz                         ;if so, set flag
                setb    rx_flag
rxdone
```

This small bit of code performs the 6 steps (try and match each step with the corresponding code). Since the **rx_divide** counter is only really used once the receiver is synchronized, the code is searching for a start bit at the raw interrupt rate. If the ISR is using –163 as an argument to **iretw**, then this code searches for a start bit every 3.26uS. This is twice as fast as a 150 KBaud input signal and four times as fast as a 75 KBaud input.

If your main program wants to read a byte, it first tests **rx_flag**. Then it can read the byte. Of course, it must read characters fast enough to prevent character overruns. Here is a simple subroutine that reads a single character:

```
get_byte
            bank  serial
            jnb     rx_flag,$           ;wait till byte is received
            mov     byte,rx_byte        ;store byte (copy using W)
            clrb    rx_flag             ;reset the receive flag
            ret
```

Selecting the Baud Rate

For the code above to work, you need definitions for **baud** and **baud15**. These represent the number of interrupt cycles for a bit, and for 1.5 bits. If the interrupt period is 163 clock cycles at 50MHz, then each interrupt cycle is 3.26uS. For 9600 baud the bit period is about 104.2uS. Since 104.2/3.26 is 31.96 you could use a count of 32 and be close enough. The **baud15** symbol, of course would be 48.

One way to get the receiver working at 9600 baud would be to use the following statements:

```
baud        equ   32
baud15      equ   48
```

It would be clever to base **baud15** on **baud** so that it had to be correct:

```
baud        equ   32
baud15      equ   3*baud/2
```

You can do math like this as long as it uses all constants so the assembler can compute the result. In this case 3, 2, and **baud** all have known values during assembly. You have to be careful, because the assembler only deals with integer math. It also evaluates expressions from left to right (not the usual order of operations). So writing **3*baud/2** works but writing **3/2*baud** will not work. That's because the assembler computes 3/2 first and finds the result is 1! You can use parenthesis if you like to make the order clear:

```
baud15      equ   (3*baud)/2
```

It would be even better to select the baud rate in an intuitive way:

```
baudrate    equ   9600

IF baudrate = 9600
baud        equ   32
ENDIF

            IF baudrate = 19200
baud        equ   16
            ENDIF

baud15      equ   3*baud/2
```

Of course, you'd have to add **IF** cases for every baud rate you wanted to support. You might be tempted to write the entire calculation in the assembler. For example:

```
osc = 50_000_000  ‘ the assembler allows _ to separate numbers
icycle = 163
baudrate = 9600

baud = osc/(icycle * baudrate)
```

This is technically acceptable, but because of the integer math, the answer is not precise. The correct result for **baud** is 32 (because the real answer is 31.9). With integer math, the result is simply 31. This error will result in a baud rate of 9895, an error of 3%. This might be acceptable, but you can do better with 32 (about 0.15% error).

Buffering

Your program may have more to do than just process characters. It is often useful to store characters away in a buffer for later use. Usually such a buffer is a circular buffer. A circular buffer is constructed so you place characters in one end of the buffer and retrieve them from the other end. As long as you read the characters before the other end of the buffer catches up, the buffer can always accept more characters.

To implement a circular buffer, you'll decide on the total number of characters you can hold at once. You'll usually pick a power of two (16 is a handy number for the SX). You'll then use one pointer to point to the head of the buffer (where input characters go) and another to point to the tail of the buffer. Programs read characters from the tail. When the tail and the head are equal, the buffer is empty.

Each time you increment one of the pointers, you limit its value by anding it with, in this case, $F. This has the effect that the pointers wrap around. The head pointer moves in the sequence: 0, 1, 2, . . ., 14, 15, 0, 1, 2...

The head pointer always points to the next empty slot. Unless the buffer is empty, the tail points to the next character waiting to be read. If the head pointer is just behind the tail pointer, the buffer is full. That means with 16 bytes, the total number of characters you can store is really 15, since the full condition wastes one byte.

You could modify the ISR to store the character in such a circular buffer. Assume that **rx_byte** is in bank 0 (remember, bank 0 is available no matter what other bank is active). Also suppose that there is a **head** and **tail** variable in bank 0. An entire bank (any empty bank will do) will server as the 16-byte buffer.

You could replace the **setb rx_flag** statement in the ISR with a subroutine call. The call could look something like this:

```
      mov   fsr,#buffer
      add   fsr,head
      mov   ind,rx_byte
      inc   head
      and   head,#$F
      ret
```

Don't forget: the **ind** register really isn't a register at all. It contains the value of the memory location pointed to by **fsr**. This simple code doesn't check for overflow– if you overflow the buffer, you'll just lose characters. Don't forget that loading **fsr** changes the bank, so any statements that depend on a special bank will need to reload **fsr** or issue a **bank** command.

Now the **get_byte** routine looks different:

```
get_byte

            mov     w,head              ;wait till byte is received
            mov     w,tail-w
            jz      get_byte
            mov     fsr,#buffer
            add     fsr,tail
            mov     byte,ind
            inc     tail
            and     tail,#$F
            ret
```

This version of **get_byte** waits until the buffer contains at least one character and then loads it into the **byte** variable. Notice again that changing **fsr** changes the bank, so this code assumes **byte** is in bank 0.

A Simple Macro

In the ISR and **get_byte** there is code that increments a pointer and ands it with $F. This code is necessary to cause the pointers to wrap around from the end of the buffer back to the beginning. However, it is easy to forget to perform the and. This is a good place to use a macro. A macro is like a user-defined instruction. Consider this macro:

```
      circinc     macro 1
                  inc   \1
                  and   \1,#$F
                  endm
```

The first line names the macro. You'll use this name (**circinc**) to refer to the macro. The 1 at the end of the line signifies that the macro takes 1 parameter (or argument, if you prefer). The next two lines are straightforward assembly except for **\1** which signifies the parameter. The **endm** keyword ends the macro. So if you write:

```
      circinc     tail
```

The assembler generates:

```
inc   tail
and   tail,#$F
```

Of course, you can also write **circinc head** to do the same operation on the head variable. This is a very simple macro. You'll often see macros that are more complex. You can combine macros with repeat blocks, conditional assembly, and local labels to make very complicated pseudo instructions.

Connections

Good design practice dictates connecting the SX to an RS-232 transmitter via a buffer (for example, a Maxim MAX232 IC). However, you can take advantage of the SX's overvoltage protection diodes to prevent the +/-12V signals from damaging the SX. However, the diodes will short the transmitter to ground and could damage it, unless you use a series resistor. In practice, a 22K resistor between the RS-232 transmitter (pin 3 on a DB9 connector) and the SX pin will work fine.

Tip: If you elect to use a buffer IC, it will most likely invert the data. That means you'd have to change the UART code to sense an incoming 1 as a 0 and vice versa.

Summary

This unit shows the inner workings of a software UART receiver. In the exercises, you'll have a chance to implement this receiver and make it do something useful. Along the way you've learned about assembler math expressions and about simple macros.

The receiver gives the SX the ability to listen to a PC or other serial device. Obviously, the ultimate goal is to marry the receiver and the transmitter. For now, however, we'll only use one or the other.

Exercises

1. Consider these lines of code:

```
val   =     33
junk  =     1000/12*val
```

What is the value of **junk**?

a) 2.5
b) 2
c) 2739
d) 2750

2. In the last unit, you used a **rept** directive to generate a number of **nop** instructions. Encapsulate the **rept** inside a macro named **nop_n** that takes a single argument to indicate how many cycles to waste. Bonus: Can you make the macro use a combination of **jmp** and **nop** instructions? (Hint: You need the remainder from division operator //).

3. Hook LEDs in the usual way (using a 470 ohm resistor) to ports RA0 and RA1. Use a 22K resistor to connect pin 3 of a DB-9 connector to RB2. Be sure to ground pin 5 of the DB-9 to the common Vss pin on the SX-Tech board. Write a program so that when a PC sends an upper case A it lights the LED on RA0. Sending a lower case a turns the LED off. B and b can operate the LED on RA1.

4. Write a program that joins the serial transmitter and serial receiver together. For a main program, you can read characters from a PC, convert them to upper case, and echo them back to the PC all at 9600 baud. Hint: To shift a lower case “a” to an upper case “A”, clear bit 5. Be sure to test that the letter is really a lower case letter before making the change.

Answers

1. C is the correct answer.

2. The simple solution is:

```
nop_n       macro 1
            rept \1
            nop
            endr
            endm
```

To do the bonus part of this question, you had to perform a little math. The idea is to use **\1/3** to determine how many **jmp $+1** instructions are required and **\1//3** to determine how many **nop** instructions are necessary. However, it is possible that either of these numbers could be zero. Therefore each **rept** block is protected with an **IF** statement since **rept** does not accept zero as an argument.

```
nop_n       macro 1
            IF \1/3<>0
             rept \1/3
              jmp $+1
             endr
            ENDIF
            IF \1//3<>0
             rept \1//3
              nop
             endr
            ENDIF
            endm
```

Try using these macros and press Control+L in the SX-Key environment to see how the code expands for different cases.

3. There are several ways you could write this program. Here is one possible solution (assuming that a low on the output pin turns the LED on):

```
            device      sx28l,oscxt5,turbo,stackx_optionx
            reset start_point
            freq  50000000   ; 50 Mhz

BAUDRATE    EQU   9600  ; baud rate to stamp
; Port Assignment: Bit variables
;
```

```
rx_pin      EQU     rb.2                ; PC input
            org   8
; Head/tail pointer
head        ds    1
tail        ds    1
byte        ds    1                     ;temporary UART byte
rx_byte     ds    1                     ;buffer for incoming byte

            org     10h
serial          =       $               ;UART bank

rx_count        ds      1               ;number of bits remaining
rx_divide       ds      1               ;receive timing counter

IF BAUDRATE=9600
baud            =    32
baud15          =    48
ENDIF

int_period      = 163
bufmod            equ   $F


; circular buffer is at $50
            org   $50
scan        ds    1         ; buffer


            org   0
; Interrupt service routine
isr         bank    serial                  ;switch to serial register
bank

:receive
                movb    c,/rx_pin
                test    rx_count            ;waiting?
    jnz     :rxbit              ;if not,
                mov     w,#9                ;in case start, ready 9
                sc                          ;if start, set rx_count
                mov     rx_count,w
                mov     rx_divide,#baud15    ;ready 1.5 bit periods
```

```
:rxbit          djnz    rx_divide,rxdone     ;8th time through?
                mov     rx_divide,#baud
                dec     rx_count             ;last bit?
                sz                           ;if not, save bit
                rr      rx_byte
                snz                          ;if so, put in circbuff
                call    bufferin
rxdone

;interrupt every 'int_period' clocks
end_int     mov     w,#-int_period
            retiw                           ;exit interrupt


; put character in circular buffer
bufferin
      mov   fsr,#scan
      add   fsr,head
      mov   ind,rx_byte
      inc   head
      and   head,#bufmod
      ret

start_point
            mov     ra,#%0011           ;initialize port RA
            mov     !ra,#%0000              ;Set RA in/out directions
            mov   rb,#%00001010
            mov   !rb,#%11110101

            CLR     FSR               ;reset all ram starting at 08h
:zero_ram       SB      FSR.4         ;are we on low half of bank?
            SETB    FSR.3             ;If so, don't touch regs 0-7
            CLR     IND               ;clear using indirect addressing
            IJNZ    FSR,:zero_ram     ;repeat until done

            mov     !option,#%10011111      ;enable rtcc interrupt
            clr rb

; Here is where the action is!
mainloop
      call get_byte
      cje byte,#'A',Aon
```

```
      cje byte,#'a',Aoff
      cje byte,#'B',Bon
      cje byte,#'b',Boff
      jmp mainloop


Aon
      clrb ra.0
      jmp mainloop
Aoff
      setb ra.0
      jmp mainloop
Bon
      clrb ra.1
      jmp mainloop
Boff
      setb ra.1
      jmp mainloop



; Subroutine - Get byte via serial port
;
get_byte
            mov     w,head              ;wait till byte is received
            mov     w,tail-w
            jz      get_byte
            mov     fsr,#scan
            add     fsr,tail
            mov       byte,ind
            inc     tail
            and     tail,#$F
            ret
```

4. Again, there are many possible answers to this question. Here is one solution:

```
device      sx28l,oscxt5,turbo,stackx_optionx
reset start_point
freq  50000000   ; 50 Mhz
```

```
BAUDRATE        EQU     9600  ; baud rate to stamp
; Port Assignment: Bit variables
;
rx_pin          EQU     rb.2
tx_pin          EQU     rb.3

            org   8
; Head/tail pointer
head        ds    1
tail        ds    1
byte        ds      1                       ;temporary UART byte
rx_byte     ds      1                       ;buffer for incoming byte

            org     10h
serial          =       $                       ;UART bank
;
rx_count        ds      1               ;number of bits left
rx_divide       ds      1               ;receive timing counter
tx_high         ds      1                ;tx
tx_low          ds      1
tx_count        ds      1
tx_divide       ds      1

IF BAUDRATE=9600
txdivisor   =    32
baud        =    32
baud15      =    48
ENDIF

int_period      = 163
bufmod            equ   $F


; circular buffer is at $50
            org   $50
scan        ds    1         ; buffer


            org   0
```

```
; Interrupt service routine
isr         bank    serial                  ;switch to serial register
bank

:receive
                movb    c,/rx_pin             ;serial receive
                test    rx_count              ;waiting
                jnz     :rxbit                ; no?
                mov     w,#9                  ;in case start, ready 9
                sc                            ;if start, set rx_count
                mov     rx_count,w
                mov     rx_divide,#baud15     ;ready 1.5 bit periods
:rxbit          djnz    rx_divide,rxdone      ;8th time through?
                mov     rx_divide,#baud
                dec     rx_count              ;last bit?
                sz                            ;if not, save bit
                rr      rx_byte
                snz                           ;if so, set flag
                call    bufferin
rxdone
; transmitter
            bank  serial
            dec   tx_divide
            jnz     end_int
            mov   tx_divide,#txdivisor    ; ready for next
            test  tx_count                ;busy?
            jz end_int                    ; no byte being sent
            stc                           ; ready stop bit
            rr    tx_high
            rr    tx_low
            dec   tx_count
            movb  tx_pin,/tx_low.6        ;output next bit



end_int           mov     w,#-int_period
            retiw                           ;exit interrupt


; add to circular buffer
```

```
bufferin
      mov   fsr,#scan
      add   fsr,head
      mov   ind,rx_byte
      inc   head
      and   head,#bufmod
      ret

start_point
            mov   ra,#%0011           ;initialize port RA
            mov   !ra,#%0000          ;Set RA in/out directions
            mov   rb,#%11110111
            mov   !rb,#%11110111

            CLR     FSR                ;reset all ram starting at 08h
:zero_ram       SB      FSR.4          ;are we on low half of bank?
            SETB    FSR.3              ;If so, don't touch regs 0-7
            CLR     IND                ;clear using indirect addressing
            IJNZ    FSR,:zero_ram      ;repeat until done

            mov     !option,#%10011111 ;enable rtcc interrupt
            clr rb

; Here is where the action is!
mainloop
      call get_byte
      cjb byte,#'a',noshift
      cja byte,#'z',noshift
      clrb byte.5
noshift
      mov  w,byte
      call send_byte
      jmp mainloop



; Subroutine - Get byte via serial port
;
get_byte
            mov     w,head             ;wait till byte is received
            mov     w,tail-w
            jz      get_byte
```

```
            mov     fsr,#scan
            add     fsr,tail
            mov       byte,ind
            inc     tail
            and     tail,#$F
            ret

send_byte   bank  serial

:wait       test  tx_count          ;wait for not busy
            jnz   :wait

            mov   tx_high,w
            clrb    tx_low.7   ; set start bit


            mov   tx_count,#10      ;1 start + 8 data + 1 stop bit

            ret
```


Rev1.

Unit V. Pulse I/O

Unit V from I/O Control with the SX Microcontroller

5

When I was in high school I had a math teacher who used to say, "You have to use what you know to find out what you don't know." This is often the case with microcontrollers. Computers are very good at measuring certain things (like digital levels). Computers are not very good at measuring other things like analog quantities (at least without additional hardware).

So to paraphrase my math teacher, if you could convert something that is hard to measure into something that is easy to measure, you could more easily read it. Consider a potentiometer. Sure, you can read it using an A/D converter (see unit 3). However, what if you could connect the potentiometer so that the SX could measure time and determine the position? The SX is excellent at measuring time. All that you need is a circuit that will allow the potentiometer to control the width of a pulse. The SX can measure the pulse width and deduce the potentiometer's position.

What about other types of input? Many real-world sensors look like variable resistors. Ideally, you could treat them just like potentiometers and use the SX to read temperature, humidity, light intensity or any of the other things you can measure with a resistive sensor.

The same idea holds true for analog output. If you could convert time into voltage, you'd have a D/A (digital to analog) conversion scheme that the SX could handle. Converting back and forth between analog values and times requires a capacitor and the ability for the SX to create and measure pulses.

Capacitor Fundamentals

Capacitors have many uses in electronic circuits. For the purposes of this unit, we will use them as energy storage devices. Suppose you have a capacitor with one lead grounded. Initially, the capacitor has 0V across it. Then you apply 5V to the other lead of the capacitor via a resistor. At first, the capacitor looks like a dead short and the voltage across it remains 0V. But the capacitor charges so the voltage increases until the final voltage is practically 5V.

Of course, the capacitor doesn't charge instantaneously. It takes a finite amount of time for the capacitor's voltage to change from one value to another. The speed that the capacitor's voltage ramps up depends on the value of the resistor (*R*) and the value of the capacitor (*C*). The voltage *V* at time *t* with a 5V supply will be:

$$V = 5(1-e^{-t/RC})$$

So if R=100000 (100K ohms) and C = .00001 Farads (10uF), you'd find the voltage on the capacitor would look like this:

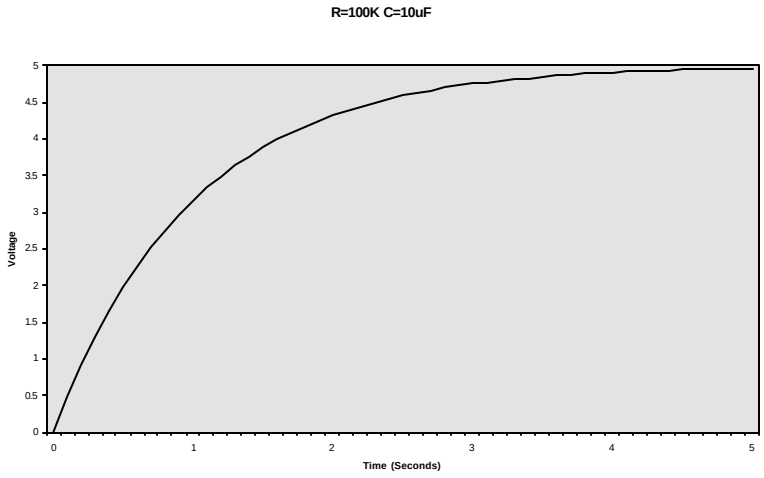


Figure V.1 – Capacitor Charging Curve

A good rule of thumb is that after *RC* seconds, the voltage will be 63% of the charging voltage. You can verify this on the above chart. The charging voltage is 5V so 63% is 3.15V. The curve is just above 3V at 1 second (100000 times .00001 is equal to 1).

Notice that changing the resistance value or the capacitor's value will change the amount of time it takes the curve to get to any particular voltage. Using the 63% rule, how long would it take to reach 3.15V if you doubled the resistance? The answer is 2 seconds. So by charging a capacitor you can convert a resistance to a time – just what the SX needs. Of course, you could use a fixed-value resistor and vary the capacitance, too. It works just as well either way.

The same thing happens if you charge the capacitor up and then discharge it through a resistor. It will take *RC* seconds to reach 37% of the initial voltage.

What can you do with this idea? Obviously you could read a potentiometer. Perhaps you want the SX to dim a light or control a motor speed as the user moves a knob. However, many sensors provide a resistive or capacitive reading. For example, a thermistor changes resistance in response to temperature. A strain gauge varies its resistance with weight. A cadmium-sulfide cell changes resistance in response to light. You could read any of these sensors using this technique.

Of course, theory and practice are often two different things. Real capacitors don't store energy perfectly. There is leakage resistance and other factors that can throw things off slightly. Most capacitors are temperature sensitive themselves. However, in practice these issues are not problems in most cases. Still, be aware that real-world capacitors are notorious for not matching their ideal characteristics.

Thresholds

To measure an unknown resistance, you can discharge the constant-value capacitor and compute how much time it takes to charge back to a logic 1 level. Alternately, you could charge the capacitor to 5V and compute how much time it takes to fall to a logic 0. This is an excellent place to use the SX's special I/O functions.

Each input pin on the SX has several control registers. You can use these control registers to set different options. One of these options is to use a CMOS input threshold. When this mode is active, any input over 0.5Vdd (nominally 2.5V) is considered a logic 1. If the CMOS mode is not set, the threshold voltage is about 1.4 to 1.5V. You can set each pin individually.

To set the threshold voltage for a port, you first set the **M** (mode) register to $D. Then you can store configuration bits in the **!ra**, **!rb**, and **!rc** registers. A zero in these registers makes the corresponding bit use the CMOS threshold. A one sets the pin for 1.4V (TTL) threshold. It is a good idea to set the **M** register back to the default value ($F) when you are finished. You could, in theory, use this feature to determine what part of the capacitor voltage curve you will detect.

In real life, however, neither choice is the best one. To see why, think about the types of signals an input pin normally sees. A typical logic signal moves from 0 to 5V very quickly (ideally, instantaneously although that isn't really possible). You think of these signals as "square" – the transitions are very steep. If you look at the above chart, you'll see that the capacitor's voltage is not steep at all. That means the circuit will slowly pass through the SX's threshold voltage. Right at the threshold, the SX may detect more than one change in the input's state. Power supply fluctuations and circuit noise can make a signal right at the threshold appear to be a 1 on one reading, a 0 on the next, and then later read to be a 1 again.

To combat this, it is common to use a special gate called a Schmitt trigger. This is simply a logic gate that reads a logic 1 when the input voltage rises above (approximately) 62% of Vdd (3.1V with a 5V supply). However, it will not read the pin as a logic 0, until the voltage falls below about 28% of Vdd (1.4V). This electronic inertia is known as hysteresis. Consider this table:

Time	Input voltage	Input state
0	0.0V	0
1	4.0V	0
2	4.5V	1
3	4.0V	1
4	2.0V	1
5	0.5V	0
6	2.0V	0
7	4.0V	0
8	4.5V	1

You can buy ICs that perform the Schmitt trigger function, but luckily, the SX already has them built in if you want them. To set Schmitt trigger mode, you set the **M** register to $C and then set the **!ra**, **!rb**, or **!rc** registers. Placing a zero in a bit makes the corresponding input a Schmitt trigger.

Measuring Time

The SX, of course, can keep time in a variety of ways. The trick is to select a method that provides adequate resolution for the task at hand without using such a high resolution that you'll need large counters to handle the time periods of interest. For example, suppose you have a 10K pot and a .1uF capacitor wired as shown:

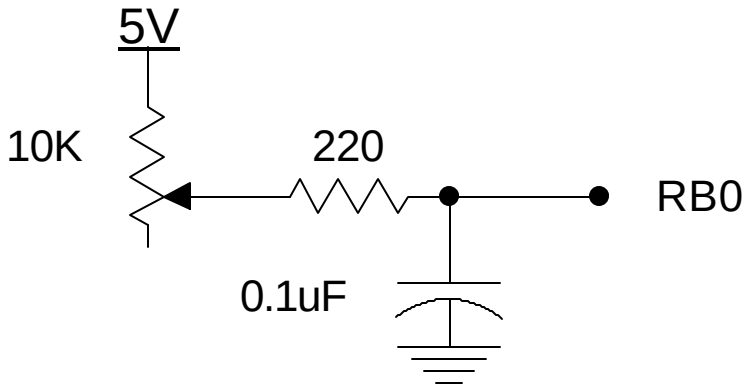


Figure V.2 – Reading a Potentiometer

The *RC* constant for this circuit is .001. That means that in 1mS, the capacitor will charge to about 3.15V. This is right around the threshold for a Schmitt trigger (3.1V). This sets an upper bound on the time you need to measure. Of course, the Schmitt level is not precise, and the components involved are not precise either. To be safe, you'd like to be able to measure at least 2mS.

There are many ways you could perform these measurements. A simple counter would work. However, if you write the following code:


```
loop  inc counter
      jnb loop
```

You'll find that the total execution time per loop is 5 clock cycles. At 50MHz that is only 100nS per count. You have to count to 20,000 to measure 2mS. That means you can't use a single byte counter. Two bytes can contain up to 65535 so you could write:

```
loop    jb done
        inc count0
        snz
        inc count1
        jmp loop
```

This takes 8 cycles per loop (ignoring the final loop) so each count represents 160nS. When **count0** overflows, the code increments **count1**. This forms a 16-bit counter.

Tip: Be sure to use **snz** and not **snc**. Using **inc** does not affect the carry flag. It does affect the zero flag.

This method leaves a little to be desired. The count will vary a bit because interrupts occur and steal cycles from the loop counter. You could disable interrupts, but that would affect the serial I/O code or any other ISRs that might be running.

A better way is to use the ISR to perform the timing for you. Suppose you made the ISR increment a 16-bit counter on each pass. You could use this counter to measure the number of interrupt periods that elapsed between two events. If you use the same ISR we've used throughout this course, you'd get a count every 3.26uS. A 2mS count would be around 613 or 614 – you'd still need two bytes for the counter.

This method is also somewhat inaccurate in practice. The serial transmitter and receiver code take a varying amount of time to execute. This can lead to small inaccuracies in the timing. However, for this purpose the timing is more than adequate.

Another idea would be to use the ISR to perform all the timing. Then the main program can simply read the count that the ISR generates. For the purposes of timing an RC network, any of these methods will work.

Program Details

Here is the basic way that the program will work:

1. Change **RB.0** to an output and pull it low
2. Pause a few ms to allow the capacitor to fully discharge
3. Restore **RB.0** to an input
4. Time how long it takes for **RB.0** to rise to a logic 1

The difference, of course, is how you measure the time. Here is a simple version:

```
read_rc
      clrb rb.0
      mov !rb,#%11110110      ; bit 0 to input
      call pause              ; discharge time
      mov dly,#$FF            ; reset timer
      mov dly1,#$FF
:zwait
      test dly                ; sync with ISR
      jnz :dwait
      mov !rb,#%11110111      ; back to input
captest
      jnb rb.0,captest
      mov vallow,dly
      mov valhigh,dly1
      ret
```

This requires a bit of support. Obviously, you need a **pause** routine. The exact time is not important, but it does need to be a long enough delay to allow the capacitor to fully discharge. The other part of the code that isn't clear here is how **dly** (and **dly1**) change. This, of course, is part of the ISR. The very first lines of the ISR are now:

```
      bank delaybank
      inc dly
      snz
      inc dly1
```

The **read_rc** code doesn't change banks, because the **pause** routine also uses **dly** and it sets the bank. The **pause** routine is just five calls to **pausems**. This routine delays about 1mS. Here is the code:

```
pausems
      bank delaybank
      mov dly1,#$FE
      mov dly,#$CD
```

```
:p1   mov w,dly1
      or w,dly
      jnz :p1
      ret
```

This bears some explanation. The routine takes advantage of the fact that the ISR will increment the 16-bit **dly** variable every 3.26uS. To pause 1mS (1000uS), the code needs to wait for 307 counts. Expressed in hex, 307 is $133. Rather than clear the **dly** variable and wait for $133, the code instead loads negative $133 and waits for the variable to reach 0 (a cleaner test). To negate $133 write it as binary, invert the bits and add 1. So:

%0000 0001 0011 0011 -> %1111 1110 1100 1100 + 1 = %1111 1110 1100 1101 = $FECD

Of course, other factors contribute, so the delay is not precise, but it doesn't need to be. Anything close to 1mS will be good enough in this case.

Pulse Output

It should be obvious that if you can measure precise times, you can also create pulses. You simply set an output bit's state, wait for a particular interval, and then reset the bit's state. In the next unit you'll see how a train of pulses combined with a capacitor can generate an analog output using a method known as pulse width modulation (PWM).

PWM is useful for other reasons as well. For example, you can control an LED or lamp's brightness. You can also use PWM to control the speed of a motor. Some external systems require pulses to operate. For example, servo motors (common in the radio control hobby) use a pulse to determine the shaft's position. These motors typically don't rotate 360 degrees. Instead they will move over a certain arc. With a narrow pulse, the motor will position the shaft to one extreme of the travel range. The wider the pulse, the further away the shaft moves (until it reaches the other extreme).

Summary

Converting an analog value like a resistance or capacitance into a measurable time is a powerful idea. With some additional circuitry you could even do the same thing with a voltage. For example, a 555 IC can generate pulses that vary in width depending on an applied voltage. There are also specific voltage to frequency ICs. An oscillator with a varactor in its resonator can also change frequency (and hence, pulse width) with an applied voltage.

Using sensors like thermistors or light-dependent resistors allows you to adapt this technique to make the SX read a variety of real-world parameters. Accepting this type of input is an essential component to creating control or data acquisition systems.

Exercises

1. Connect a 10K potentiometer and LED as in the diagram below. Write a program that allows you to test the threshold voltages for TTL, CMOS, and Schmitt trigger inputs by transferring the state of the input pin to the output LED. You can measure the input pin's voltage with a common voltmeter.

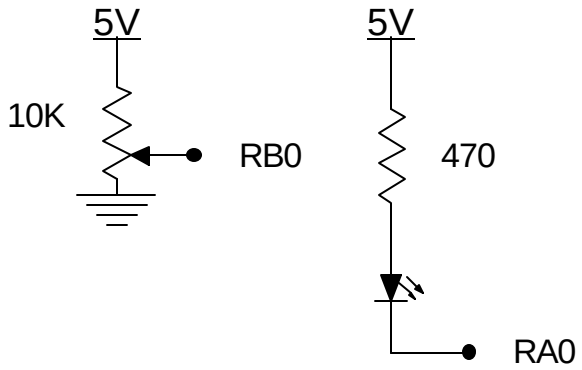


Figure V.3 – Threshold Test Circuit

2. Build the circuit shown earlier in this unit. Create a program that reads the 16-bit count that shows the potentiometer's position and verify your code's operation using the SX-Key debugger.

3. Modify the above program to display the result on an RS-232 terminal. Hint: Write a carriage return (13) and disable the terminal's auto linefeed mode (if any) to see a pleasing display.

Answers

1. Here is a possible solution:

```
            device      sx28l,oscxt5,turbo,stackx_optionx
            reset start_point
            freq  50000000   ; 50 Mhz

            org   0
start_point
            mov   ra,#%1111
            mov   !ra,#%1110


; set threshold here $C = Schmitt $D = CMOS
            mov m,#$C
            mov !rb,#%11111110
            mov m,#$F


; Here is where the action is!
mainloop
      movb ra.0,/rb.0
      jmp mainloop
```

Notice that in TTL or CMOS mode, the LED may light dimly. This is because without Schmitt trigger hysteresis, the SX is reading the pin as a 1 sometimes and a 0 at other times right at the threshold voltage.

2. See the answer for exercise 3. This is the same code but without the serial transmitter code.
3. There is no need for the serial receiver in this code although if you included it, there is no harm in it:

```
            device      sx28l,oscxt5,turbo,stackx_optionx
            reset start_point
            freq  50000000   ; 50 Mhz

BAUDRATE    EQU   9600  ; baud rate to stamp
; Port Assignment: Bit variables
;
tx_pin            EQU   rb.3
```

```
            org   8
; Head/tail pointer
byte            ds      1                       ;temporary UART byte
vallow          ds      1
valhigh         ds      1
number_low      ds      1
temp            ds      1

      watch dly,16,uhex


            org     10h
serial          =       $                       ;UART bank
;
tx_high     ds    1               ;tx
tx_low      ds    1
tx_count    ds    1
tx_divide   ds    1

IF BAUDRATE=9600
txdivisor   =    32
ENDIF

int_period      = 163


            org $30
delaybank   equ $
dly         ds    1
dly1        ds    1




            org   0
; Interrupt service routine
isr
            bank  delaybank
            inc   dly
            snz
            inc   dly1
```

```
            bank    serial

; transmitter
            bank  serial
            dec   tx_divide
            jnz     end_int
            mov   tx_divide,#txdivisor    ; ready for next
            test  tx_count                ;busy?
            jz end_int                    ; no byte being sent
            stc                           ; ready stop bit
            rr    tx_high
            rr    tx_low
            dec   tx_count
            movb  tx_pin,/tx_low.6        ;output next bit



end_int
            mov     w,#-int_period
            retiw                           ;exit interrupt



start_point
            mov     ra,#%0011           ;initialize port RA
            mov     !ra,#%0000              ;Set RA in/out directions
            mov   rb,#%11110111
            mov   !rb,#%11110111

            CLR     FSR               ;reset all ram starting at 08h
:zero_ram       SB      FSR.4         ;are we on low half of bank?
            SETB    FSR.3             ;If so, don't touch regs 0-7
            CLR     IND               ;clear using indirect addressing
            IJNZ    FSR,:zero_ram           ;repeat until done

            mov     !option,#%10011111      ;enable rtcc interrupt
            clr rb

; Set Schmitt trigger input
```

```
            mov m,#$C
            mov !rb,#%11111110
            mov m,#$F


; Here is where the action is!
mainloop
      call read_rc
      mov w,valhigh
      call send_hex
      mov w,vallow
      call send_hex
      mov w,#$D
      call send_byte
      jmp mainloop


read_rc
      clrb rb.0
      mov !rb,#%11110110  ; bit 0 to output
; pause a bit to let capacitor discharge
      call pause
      mov dly,#$FF
      mov dly1,#$FF
:zwait
      test dly   ; synchronize with ISR
      jnz :zwait
      mov !rb,#%11110111  ; back to input
captest
      jnb  rb.0,captest
      break
      mov vallow,dly
      mov valhigh,dly1
      ret


pause
:p1
      rept 5
      call pausems
      endr
```

```
      ret

; pause about 1mS
; (each int tick is 3.26uS
; 1000uS/3.26=307
; 307=$133 and -$133 = $FECD
pausems
      bank delaybank
      mov dly1,#$FE
      mov dly,#$CD
:p1   mov w,dly1
      or  w,dly
      jnz :p1
      ret


; required to output HEX numbers
_hex            dw      '0123456789ABCDEF'
;
;
;***************
;* Subroutines *

; Send hex byte (2 digits)
;
send_hex
                mov     number_low,w            ; save W
                mov     w,<>number_low          ;send first digit
                call    :digit

                mov     w,number_low            ;send second digit

:digit  and     w,#$F                   ;read hex chr
                mov     temp,w
                mov     w,#_hex
                clc
                add     w,temp
                mov     m,#0
                iread             ; read from program mem!
                mov     m,#$F
```

```
; fall into send byte

send_byte   bank  serial

:wait       test  tx_count          ;wait for not busy
            jnz   :wait

            mov   tx_high,w
            clrb    tx_low.7   ; set start bit


            mov   tx_count,#10      ;1 start + 8 data + 1 stop bit

            ret
```


Rev1.

Unit VI. PWM

Unit VI from I/O Control with the SX Microcontroller

In the last unit you looked at measuring pulse widths. Of course, if you can measure an interval, you can also create pulses. However, using pulses can have a few nuances that you should understand. In particular, you can use pulses, in combination with a handy capacitor, to generate a voltage from 0 to 5V– if you know all the right tricks.

PWM Theory

The most interesting use of pulses with a microcontroller is to use a string of pulses to generate an arbitrary analog voltage. These analog signals might be useful as control voltages or even audio outputs. Using pulses this way is known as Pulse Width Modulation (PWM).

To generate a voltage with PWM, you'll use our favorite energy storage device: the capacitor. The best way to understand the process is to look at the two extreme cases first. Suppose you have an SX output pin connected to a capacitor. If you bring the output pin low, the capacitor will discharge and it is easy to see that the capacitor's voltage will be 0V. Similarly, if you bring the output high, the voltage will charge the capacitor and you will soon have 5V across the capacitor.

What happens, however, if you bring the output pin high for 1mS and then low for 1mS and keep repeating this sequence? When the pin is high, the capacitor will charge up. When the pin is low, the capacitor will discharge. Since the 1mS time is the same for both conditions, the average voltage across the capacitor will be 2.5V (one half of the 5V output). If you keep the pin high for 1mS and then low for 4mS, the output will be 1V.

In general, the output voltage will be 5V times the percentage of time the pulse is high. In theory, it doesn't matter how long the pulses are, as long as the percentage is correct. If the high and low periods were 100uS and 400uS, the output would still be 1V. The percentage of time the signal is high is known as its duty cycle. In this example, the duty cycle is 20%.

Figure VI.1 shows a practical circuit. The resistor prevents excessive current draw from the SX.

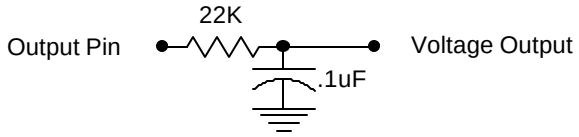


Figure VI.1 – PWM Output Circuit

Tip: Selecting a resistor and capacitor value can make a big difference in a PWM circuit. The smaller the capacitor, the quicker it will charge to the final desired value. On the other hand, smaller capacitors discharge quicker as well. The resistor, of course, also affects the timing. Lower values will reduce the amount of time required to charge the capacitor to its final value.

Practical Pulses

If you wanted to write a PWM output routine, you might be tempted to select a time period and divide it into, say, 100 slots. Then you could turn the output on, for the number of slots you wanted. For example, if each slot was 2uS and you wanted a 50% duty cycle, you'd turn the output high for 50 time periods (100uS) and then off for the next 50.

This would work, but it is less than optimal. Why? This scheme increases the amount of time it takes for the capacitor to charge and discharge. Ideally, the pulses should be as short as practical. One way to do this is to make the pulses proportional. For example, a 50% duty cycle with a 2uS timebase would have one 2uS high followed by a 2uS low. A 33% duty cycle would be 2uS on and 4uS off.

At first glance this would seem to be difficult to compute. However, a clever trick makes it quite simple. Suppose you use a byte to define 256 duty cycles. With this scheme, $FF is nearly 100%, $80 is 50% and, of course, 0 is 0%. Each unit is then roughly 0.4%.

Suppose you have an interrupt service routine that runs every 2uS and a duty cycle stored in the **pwm** variable. You can use an accumulator (**pwm_acc**) to easily handle the PWM algorithm. Here are the steps:

1) Set **pwm_acc** equal to **pwm_acc** plus **pwm**
2) If a carry results from the addition, set the output bit
3) If a carry did not result, clear the output bit.

The ISR is probably the simplest ISR you can imagine:

```
add pwm_acc,pwm
movb rb.0,c
mov w,#-100   ; every 2uS
retiw
```

Why does this work? Look at this table of values:

Time uS	duty=$FF		duty=$80	
	pwm_acc	output	pwm_acc	output
0	0	0	0	0
2	$FF	0	$80	0
4	$FE	1	$00	1
6	$FD	1	$80	0
8	$FC	1	$00	1

If you follow this sequence you'll see that this in fact works as promised. Of course, at a duty cycle of 1 (0.4%) you still have 2uS on and 511uS off, but this is the extreme case. Using a more straightforward algorithm results in this being the case for all values.

Limitations and Enhancements

There are several practical issues to consider with this type of circuit. First, the capacitor charges through a resistor. The larger the capacitor, the more time it takes to charge and discharge. On the other hand, holds it charge poorly as the PWM rate slows down.

If you really expect to draw any significant current from the PWM pin, you should consider using some sort of buffer amplifier (like an op-amp or an emitter follower amplifier). However, if you are drawing modest amounts of current (for example, a comparator or op-amp input) you can just use the PWM output directly.

You can also drive an LED using this type of PWM. You don't need a capacitor because your eye will integrate the flashes from the rapidly blinking LED. PWM (properly buffered) can also vary motor speeds.

In general, the faster the PWM rate, the smoother the PWM appears. With such a short ISR, you can easily reduce the rate by adjusting the ISR's period. For example, changing the ISR so that it loads **w** with 50 instead of 100 would drop the rate to 1uS. The entire ISR only requires 10 clock cycles, so you could reduce the number even further (as long as you don't add code to the ISR). Setting the ISR rate to 20, for example, drops the period to 400nS!

If you want finer-grain control, you could use larger PWM accumulators (and duty cycles). For example, a 10-bit set up would allow you to step the voltage about 0.1% per step (about 5mV). In this case you wouldn't use the carry bit to control the PWM, you'd use bit 9 of a 16-bit variable. Of course, at some point your step size will be smaller than the accuracy possible because of the component tolerances.

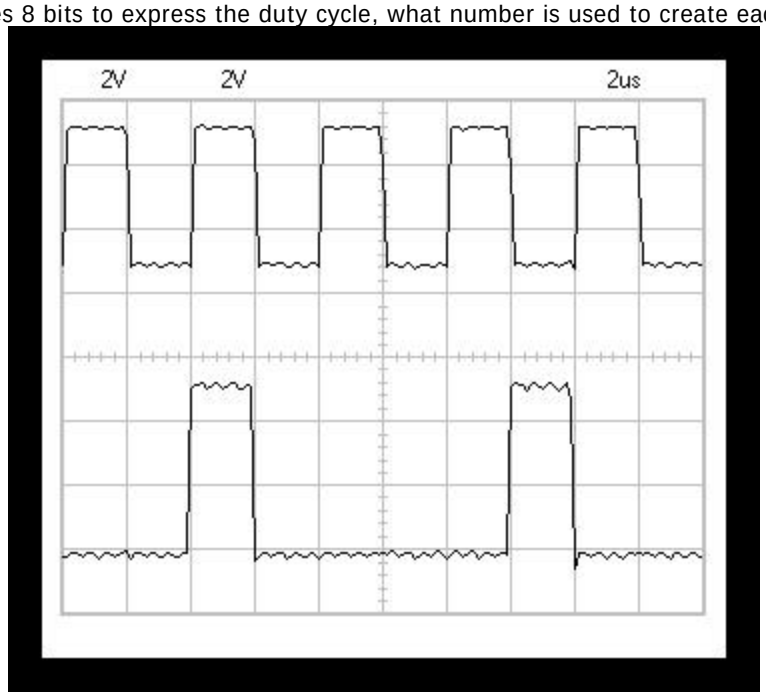
Summary

Generating pulses is both easy and extremely useful. Pulse trains can control motors, dim lights, and generate voltages with a minimum of external components.

PWM is not your only choice when it comes to analog output. There are readily available chips that will produce analog outputs. These D/A or DAC (Digital to Analog Converters) come in a bewildering array of styles and features. If you want to use a chip-level DAC, be sure to find one that accepts serial data so you conserve the SX's pins.

Exercises

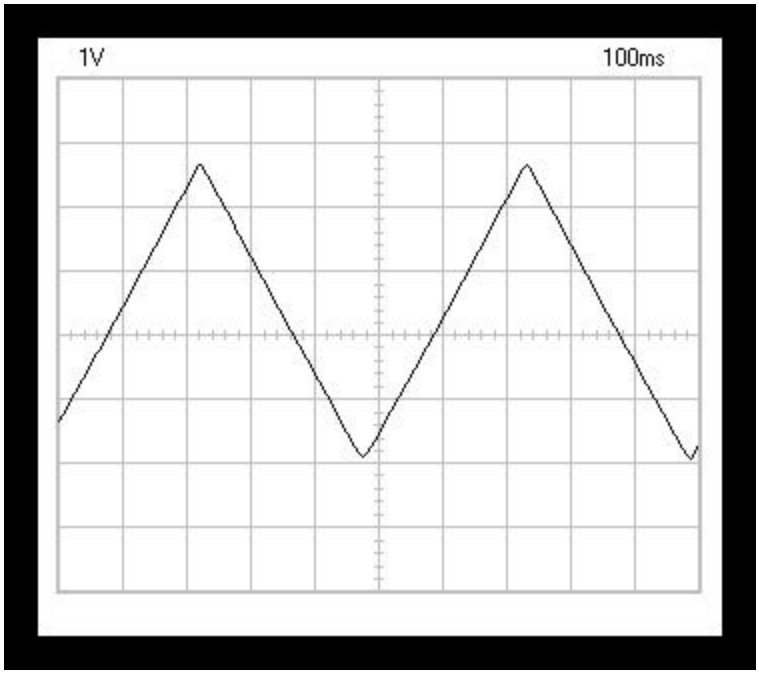
1. Below is a view of two PWM outputs. What is the duty cycle of each expressed as a percentage? If the PWM generator uses 8 bits to express the duty cycle, what number is used to create each output?



2. Set up a PWM circuit as shown and create code that varies the pwm duty cycle by 1 bit about every 250mS (or more). Using a voltmeter (or even better, an oscilloscope) verify that the change in voltage is near the expected 19.5mV. What would happen if you changed the pwm counter to use 9 bits instead of 8? Verify your answer.

3. Using your PWM circuit, devise a program that will find the input threshold voltage of another I/O pin automatically. You can do this by connecting the PWM output to another input and slowly ramping the output voltage until you find a 1 input. You can either verify your results with the debugger or with a voltmeter.

4. Look at the triangle waveform below. Can you simulate this with PWM? Write a program to generate this waveform. You can observe your results with an LED, or even better an oscilloscope, if available. Hint: The exact timing or voltage levels are not important.



Answers

1. The upper trace is high for 2uS of every 4uS and is therefore at 50% or duty cycle 128. The lower trace is high for 2uS of every 10uS – a 20% or 51 duty cycle.

2. Here is a possible 8 bit solution:

```
            device  sx28l,oscxt5
            device      turbo,stackx_optionx
            reset reset_entry
            freq  50_000_000


pwm_pin     =     rb.0

            org   8

temp        ds    1
pwm         ds    1                ;pwm0
pwm_acc           ds    1
dly         ds    1
dly1        ds    1


            org   0
;
;

;
interrupt
            inc   dly
            snz
            inc   dly1

            add   pwm_acc,pwm
            movb  pwm_pin,c

            mov     w,#-100
```

```
            retiw
;********
;* Main *
;********
;
;
; Reset entry
;
reset_entry

            mov    rb,#%00000000          ;init rb
            mov   !rb,#%11111110
            clr   fsr               ;reset all ram banks
:loop       setb  fsr.4
            clr   ind
            ijnz  fsr,:loop

            mov   !option,#%10011111      ;enable rtcc interrupt
;
;
;  - main loop
;

mainloop



            inc pwm
            call pause
            jmp mainloop


pause
:p0   mov temp,#250
:p1   call pausems
      djnz temp,:p1
      ret

; pause about 1mS
pausems
      mov dly1,#$FE
```

```
      mov dly,#$0C  ; FE0C = -500
:p1   mov w,dly1
      or  w,dly
      jnz :p1
      ret
```

To change the code to 9 bits, you'd change the ISR to look like this:

```
interrupt
            inc   dly
            snz
            inc   dly1

            add   pwm_acc,pwm
            addb  pwm_acc1,c
            add   pwm_acc1,pwm1
            movb  pwm_pin,pwm_acc1.1
            clrb  pwm_acc1.1

            mov     w,#-100
            retiw
```

Of course, you'll have to add the **pwm_acc1** and **pwm1** variables. Your main loop might look something like this:

```
            inc pwm
            snz
            inc pwm1

            call pause
            jmp mainloop
```

The expected voltage shift per step for 9 bits is 1/512V or about 2mV.

3. Here is a possible solution's main loop (this assumes an 8 bit PWM ISR):

```
mainloop

            call pause
            jb  rb.1,found
            inc pwm
            jnz mainloop
; hmmm... didn't find it
            jmp mainloop
```

```
found       break
            mov w,pwm
            jmp $   ; stop but let PWM continue
```

4. The length of the pause will determine the period of the triangle wave. Here is one possible way to generate the wave:

```
mainloop
            inc pwm
            jz  reverse
            call pause
            jmp mainloop

reverse     dec pwm
            jz mainloop
            call pause
            jmp reverse
```

Unit VII. A Practical Design - The SSIB

Unit VII from I/O Control the SX Microcontroller

One of the things the SX excels at is producing custom I/O devices for other microcontrollers. The SX is fast and inexpensive – it is well suited to the task of making dedicated peripheral devices. In this unit, you'll examine a serial communications buffer that uses an SX. This peripheral device can help other microcontrollers (like the Basic Stamp, for example) receive serial data from a PC or other device.

The Basic Stamp is a microcontroller (made by Parallax) that you program using Basic. These "Stamps" are perfect for quick and simple projects. Although Stamps excel at many jobs, they are inherently single-tasking. This single-tasking philosophy makes programming simpler, but it makes serial input tricky.

The Stamp has a perfectly capable command for reading serial data (the SERIN command). The problem is that the Stamp can't issue a SERIN command and do something else at the same time. If the Stamp is performing a task when serial data arrives, the data is lost.

To ameliorate this limitation, the Stamp can employ a handshaking signal. This output line signals the transmitting device when the Stamp is ready to accept serial data. This works well if the sending device can stop transmission. Unfortunately, this isn't always possible or desirable.

The best answer would be to insert a buffer between the sending device and the Stamp. The buffer would hold any incoming data until the Stamp program reads it. This is a perfect application for an SX. The high speed of the SX allows you to service many serial channels simultaneously with no chance of data loss. This particular design uses an SX18 – the project doesn't even use all the pins available, so there is no need for a 28-pin device. If you are working with the SX-Tech board, you can use a 28-pin device and just ignore the extra pins.

With any project, you should start with a design. Figure VII.1 shows the pin out for the buffer device (the Stamp Serial Input Buffer or SSIB). Notice that there are two input channels. The SSIB reads from these two channels and stores characters in a 16-byte buffer (each channel has its own buffer).

Each channel has an associated handshaking line. If the buffer for a channel fills up the SSIB deasserts the handshaking line and reasserts it when the buffer has more room. Of course, if you are sure the Stamp will empty the buffer faster than the device will fill it, you can ignore these handshaking lines.

On the Stamp side the SSIB uses 3 pins. One pin receives data from the SSIB. The other two pins act as handshake lines. If the Stamp asserts CHANA, the SSIB sends data from channel A to the Stamp. CHANB selects data from the B channel. If neither line is active the SSIB sends no data to the Stamp. Of course, if you are only using one channel you can connect 2 pins to the SSIB instead of 3.

In its default configuration, the SSIB uses 9600 baud communications on each channel. However, you can change a few configuration parameters to alter this for each port individually. See Table VII.1) for the available configuration options - you can change several parameters here including the polarity of each port.

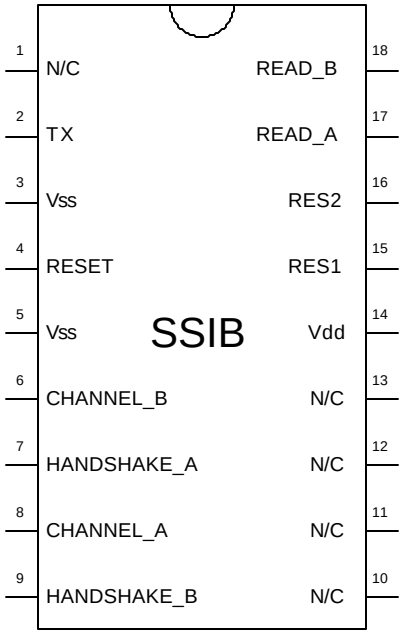


Figure VII.1 – The SSIB Pin Out (18-pin Version)

Inside the SSIB

The SSIB code (see the Listings at the end of this unit) takes advantage of the SX's high clock speed. Although the SX in use can clock up to 50MHz, this is overkill for this application. Even at 10MHz, there is plenty of time to do all the tasks required. Running more slowly allows the SX to draw less power. Remember, many processors divide their external clock, but the SX does not. So an SX running at 10MHz is comparable to some other processors running at 40MHz! A processor that divides by 4 would have to run at 200MHz to match a 50MHz SX. Almost all of the code executes in response to a high-speed periodic interrupt that occurs every 13uS.

The first thing the interrupt service routine (ISR) does is transmits any pending serial bits. Next, the serial receivers execute (first channel A, then channel B). Notice that the receivers are essentially copies of each other, but each receiver has private variables.

7

After servicing all 3 serial channels, the ISR turns its attention to managing the circular buffers for each channel. If a transmission is already in progress, the ISR simply exits. Otherwise, the ISR examines each channel's handshaking line. If the line is active, the code examines the corresponding circular buffer. If any characters are waiting, the ISR moves a waiting character into the transmit register so that on the next interrupt the character will be sent to the Stamp.

Parameter	Description	Default Value
XBAUDRATE	Baud rate to Stamp	19200
BAUDRATE_A	Baud rate to device A	9600
BAUDRATE_B	Baud rate to device B	9600
INVSEND	Use inverted mode to Stamp if 1	0
INVRCVA	Use inverted mode to device A if 1	0
INVRCVB	Use inverted mode to device B if 1	0
BUFFERLIM	Minimum free space before asserting handshake	2

Table VII.1 - SSIB Configuration

Compared to the ISR, the main code (beginning at the **start_point** label) is anticlimactic. Of course, the first few lines initialize the program, setting up the I/O pins and the periodic interrupt.

Once the chip is running, the main loop (at **mainloop**) simply waits for an incoming character, and moves it to the correct queue. The **enqueue** and **get_byte** routines (along with **enqueue1** and **get_byte1**) handle the mechanics of reading each byte and placing it in the circular buffer. Previous examples did the buffering in the ISR. However, with two channels, I decided to move the buffering to the main program (which has practically nothing to do anyway).

The queuing logic implements a 16-byte circular buffer that is more sophisticated than early versions you've examined. The tricky part of the code computes how much of the buffer is free. If this number is less than or equal to the **BUFFERLIM** constant, the SSIB turns off the inbound handshaking line for that channel. If the device in question can respond to handshake requests quickly, you could set **BUFFERLIM** to 1. However, many devices can still send a character or two before they respond to a handshake. In that case, you can set **BUFFERLIM** to a higher value.

Using the SSIB

Using the SSIB is easy with the Basic Stamp. You can find a summary of the SSIB's pins in Table VII.2. Figure VII.2 shows a sample test circuit. In this schematic, the Stamp at IC1 is receiving data

from the Stamp at IC2 (which stands in for two external devices). IC3 is the SSIB.

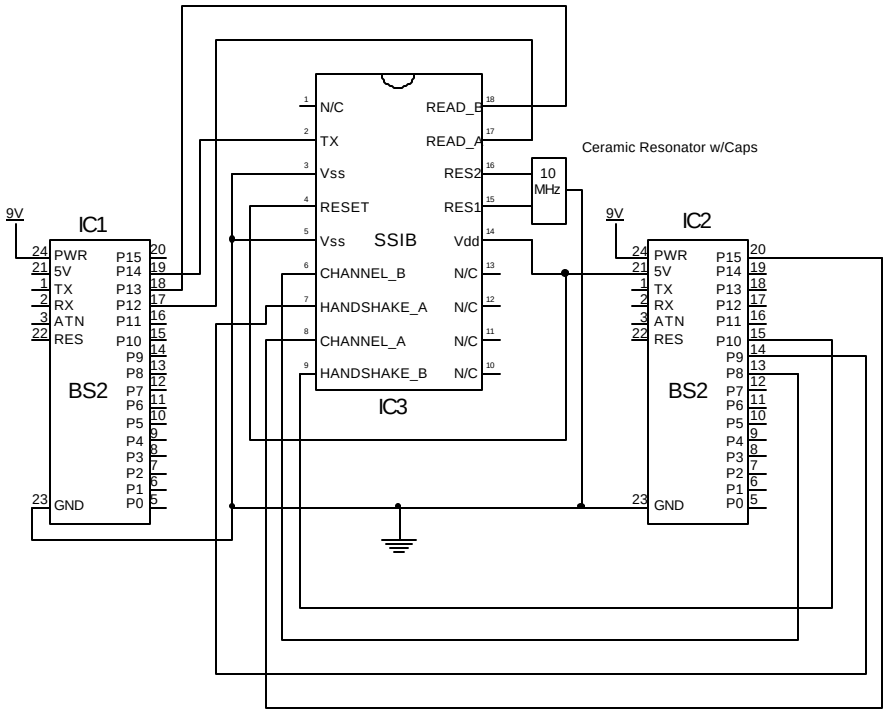


Figure VII.2 – Test Circuit for the SSIB

The device connected to the SSIB's RES1 and RES2 terminals is a 10MHz ceramic resonator with capacitors. This three-terminal device has a ground lead in the center. The other two terminals are interchangeable. If you are simply testing the circuit you can use the SX-Key or SX-Blitz to generate the 10MHz clock automatically (it senses the FREQ directive in the program). You could also use a 10MHz crystal with some extra capacitors, but a ceramic resonator is less expensive and just as good in this application. The SX data sheets show how to use a crystal if you want to try one.

The listing at the end of this unit shows the code that reads data from the SSIB. Instead of actually performing other processing, the program does simulated work in the form of a SLEEP statement. Notice that the Stamp reads data from the same pin regardless of which channel it wants to read.

However, the Stamp's SERIN command uses a different handshaking line to select the channel it wants. In this case, using pin 12 selects channel A and pin 13 selects channel B. Regardless, the Stamp reads the data from pin 14.

The simulator Stamp at IC2 (see the listings) just writes bytes out of each serial port periodically. Of course, the two Stamps won't be synchronized, so only the buffer allows this arrangement to work. If you set the first Stamp to read more often than the simulator writes, the buffer should never overflow. If you send bytes more often than you read, the SSIB buffers will fill. In this case, the SSIB will use the outbound handshaking lines to hold off the simulator.

Pin	Name	Function
1	N/C	Not connected
2	TX	Transmit data to Stamp
3	Vss	Ground
4	RESET	Pull low to reset; high for normal operation
5	Vss	Ground
6	CHANNEL_B	Input for channel B
7	HANDSHAKE_A	Optional handshake for device A
8	CHANNEL_A	Input for channel A
9	HANDSHAKE_B	Optional handshake for device B
10-13	N/C	Not connected
14	Vdd	+5V
15	RES1	Connection to 10MHz resonator
16	RES2	Connection to 10MHz resonator
17	READ_A	Signal to read from channel A
18	READ_B	Signal to read from channel B

Table VII.2 – SSIB Pinout

About Inverted Mode

The Stamp and the SSIB can perform serial I/O in standard mode, or in inverted mode. The mode selection affects the polarity of the signal line, of course, but it also changes the polarity of the handshaking lines. In standard mode, the handshake lines must go low to enable data transmission. This works well, because the SSIB has internal pull up resistors to hold the lines high in the absence of other input.

If you use inverted mode, be aware that the handshake lines will be enabled until the Stamp program or other device wakes up and explicitly inhibits transmission. This can cause problems when the Stamp misses some characters at the beginning or receives an erroneous byte right after resetting. A sleeping Stamp may also trigger data transmission since its I/O pins turn off every few seconds for a few milliseconds.

When you have a choice, use standard mode. You can set each channel independently. Another partial solution would be to use an extra Stamp pin to reset the SSIB (by pulling RESET low) after the Stamp program has control.

Customizing the Period

If you want to modify the timing used to generate the baud rates, you'll need to understand how the code handles different speeds. To ensure accuracy, the interrupt rate needs to be quite a bit faster than the period of a single bit. At 9600 baud, for example, a single bit is slightly longer than 104uS. You need to interrupt at least 4 times faster (26uS). Faster would be even better. If you don't interrupt quickly enough, you can miss a start bit. The Nyquist theorem says you must sample twice as fast, but to make sure you have enough time to work with a detected start bit, you'll want to go as fast as you can.

By default the SSIB runs at 10MHz. This causes the RTCC register to increment every 100nS. Causing an interrupt every 130 cycles makes the sampling rate 100nS*130 = 13uS; fast enough to 4x oversample a 19200 baud rate signal (52uS per bit).

The transmit code assumes that the baud rate divider will be a power of two. The define for baud9600, for example, is 3 indicating that the divisor for 9600 baud is 2 to the 3rd power, or 8. At 13uS per cycle, this works out to 104uS per bit – about 9615 baud. This is about 0.2% error – perfectly acceptable.

You might want to adjust the clock frequency to take advantage of an existing oscillator, operate at higher baud rates, or accommodate more channels. There are three things to consider:

1. The clock frequency
2. The interrupt period
3. The baud rate divider

Of these, the interrupt period is easiest to set incorrectly. Remember the RTCC keeps counting even after an interrupt occurs. The more often interrupts occur, the less time is available for the main

program. If you interrupt too frequently, the main code can't execute at all. As a practical consideration, you'll want to keep the interrupt period greater than about 80.

Suppose you wanted to use a 25MHz clock. This makes each RTCC count worth 40nS (1/25000000). If you want to sample a 9600 baud signal 8 times per bit, you need 13uS interrupts (as calculated above; this is the same as 13000nS). Therefore, the interrupt period is 13000/40 or 325. Unfortunately, it is difficult to program the single-byte RTCC register for 325 counts.

You might be able to work around this with prescaling or using a software prescaler. However, an easier method is simply to sample the signal more often. If you decide to check the bit 32 times instead of 8, you need roughly 3.3uS which requires an interrupt period of 3300/40 or about 82.

So to use a 25MHz clock, you can set the interrupt period to 82 and the baud rate number to 5 (2 to the 5^{th} power is 32). The actual time will be 40 * 82 * 32 = 104960nS or 104.96uS. Reversing the calculations, the actual bit period will be equivalent to 9527 baud; about 0.7% error. Using 81 shoots past the desired baud rate (9645 baud) but yields a smaller error (about 0.5%). In practice, either value will work.

Since the baud rate divisor number is a power of 2, it is easy to figure other baud rates. In the above example, since 5 sets 9600 baud, 4 will be 19200, 6 sets 4800, and 7 would be 2400. Since the divisor is a bit number, you can't exceed 7. To reach 1200 baud you'd need to change the clock or the interrupt period.

Further Experiments

Using this set up, you can try several other scenarios. For example, try setting the simulator to output at 2400 baud, but keep the Stamp channel at 9600. Then try reading one port at 9600 and the other at 2400.

You can change the periodic interrupt rate if you recalculate the baud rates. Just be careful to leave enough time in between interrupts to run the main program. Depending on the baud rates, clock speed, and interrupt period, you could accommodate more than just two input lines.

Summary

Why design chips like the SSIB? Creating functional modules allows designers that don't have your tools to still create powerful systems. With the low-cost of the SX chip there is no reason you can't add more than one to most designs. Even when designing with the SX, chips like the SSIB can let you distribute the workload among several processors for even more power.

The SSIB Code

```
; SSIB - by Al Williams, AWC http://www.al-williams.com/awce
; v2.0

; Use SX28L instead of sx18l for SX-Tech board
            device      sx18l,oscxt5,turbo,stackx_optionx
            reset start_point
            freq  10000000

; Port Assignment: Bit variables
;
int_period      EQU     130
XBAUDRATE   EQU   19200  ; baud rate to stamp
BAUDRATE_A  EQU   9600  ; Channel A baudrate
BAUDRATE_B  EQU   9600  ; Channel B baudrate
; Non inverted modes are best because
; the internal pull up resistors will stop all devices
; from talking, setting any of the below to 1
; makes the handshaking reverse which means
; devices are free to send until the SSIB and/or
; Stamp wakes up which may cause you problems
INVSEND     EQU   0     ; inverted/true to Stamp
INVRCVA     EQU   0     ; inverted/true to Chan A
INVRCVB     EQU   0     ; inverted/true to Chan B
BUFFERLIM   EQU   2     ; space free in buffer before h/s off

rx_pin      EQU   rb.2                  ;UART receive input
rx_pin1     EQU   rb.0
tx_pin      EQU   ra.3                  ;UART transmit output
enablepin   equ   ra.0
enablepin1  equ   ra.1
rxen_pin    equ   rb.1   ; handshake for buffer A
rxen_pin1   equ   rb.3   ; handshake for buffer B
;
            org   8
head        ds    1
head1       ds    1
tail        ds    1
```

```
tail1       ds    1
byte        ds    1
tmpvar      ds    1
flags       DS    1             ;program flags register
spare7            EQU   flags.7
rx_flag1          EQU   flags.6
rx_flag           EQU   flags.5     ;signals when byte is received
spare4            EQU   flags.4
spare3            EQU   flags.3
spare2            EQU   flags.2
spare1            EQU   flags.1
spare0            EQU   flags.0
            watch byte,8,uhex
            watch head,8,uhex
            watch tail,8,uhex
            watch rx_flag,1,uhex

            org     10h                 ;bank3 variables
serial          =       $               ;UART bank
;
tx_high         ds      1               ;hi byte to transmit
tx_low          ds      1               ;low byte to transmit
tx_count        ds      1               ;number of bits sent
tx_divide       ds      1               ;xmit timing (/16) counter
rx_count        ds      1               ;number of bits received
rx_divide       ds      1               ;receive timing counter
rx_byte         ds      1               ;buffer for incoming byte
rx_count1   ds    1
rx_divide1  ds    1
rx_byte1    ds    1

; baud rate bit #
baud2400 = 5
baud9600 = 3
baud19200 = 2
; above 19.2K may not be reliable
; without adjusting int speed (see text)

IF XBAUDRATE=2400
baud_bit        = baud2400                  ;for 2400 baud
```

```
start_delay     =       (1<<baud2400)+(1<<(baud2400-1))+1
ENDIF

IF BAUDRATE_A=2400
bauda       =     1<<baud2400
ENDIF

IF BAUDRATE_B=2400
baudb       =     1<<baud2400
ENDIF

IF XBAUDRATE=9600
baud_bit        = baud9600
start_delay     =       (1<<baud9600)+(1<<(baud9600-1))+1
ENDIF

IF BAUDRATE_A=9600
bauda            =    1<<baud9600
ENDIF

IF BAUDRATE_B=9600
baudb       = 1<<baud9600
ENDIF

IF XBAUDRATE=19200
baud_bit        = baud19200
start_delay     =       (1<<baud19200)+(1<<(baud19200-1))+1
ENDIF

IF BAUDRATE_A=19200
bauda       =     1<<baud19200
ENDIF

IF BAUDRATE_B=19200
baudb       =     1<<baud19200
ENDIF


; bit and a half for receiver alignment
baud15a           =     3*bauda/2
baud15b           =     3*baudb/2
```

```
            org   $50
scan        ds    1          ; buffer A
bufmod            equ   $F

            org $70         ; buffer B
scan1       ds    1
```

```
            org   0
isr         bank    serial
:transmit   clrb    tx_divide.baud_bit
            inc     tx_divide
            STZ
            SNB     tx_divide.baud_bit
            test    tx_count                ; are we sending?
            JZ      :receive                ; if not, go to :receive
            clc                             ; yes, ready stop bit
            rr      tx_high                 ; and shift to next bit
            rr      tx_low                  ;
            dec     tx_count                ; decrement bit counter
IF INVSEND
            movb  tx_pin,tx_low.6
ELSE
            movb    tx_pin,/tx_low.6        ; output next bit
ENDIF
;
:receive
IF INVRCVA
              movb      c,/rx_pin
ELSE
              movb    c,rx_pin            ;serial receive
ENDIF
                test    rx_count            ;waiting for stop bit?
                jnz     :rxbit              ;if not, :rxbit
                mov     w,#9                ;in case start, ready 9
                sc                          ;if start, set rx_count
```

```
                mov     rx_count,w
                mov     rx_divide,#baud15a  ;ready 1.5 bit periods
:rxbit          djnz    rx_divide,rxdone    ;8th time through?
                mov     rx_divide,#bauda
                dec     rx_count            ;last bit?
                sz                          ;if not, save bit
                rr      rx_byte
                snz                         ;if so, set flag
                setb    rx_flag
rxdone


:receive1
IF INVRCVB
               movb     c,/rx_pin1
ELSE
                movb    c,rx_pin1           ;serial receive (B)
ENDIF
                test    rx_count1           ;waiting for stop bit?
                jnz     :rxbit1             ;if not, :rxbit1
                mov     w,#9                ;in case start, ready 9
                sc                          ;if start, set rx_count
                mov     rx_count1,w
                mov     rx_divide1,#baud15b      ;ready 1.5 bit periods
:rxbit1         djnz    rx_divide1,rxdone1       ;8th time through?
                mov     rx_divide1,#baudb
                dec     rx_count1                ;last bit?
                sz                               ;if not, save bit
                rr      rx_byte1
                snz                              ;if so, set flag
                setb    rx_flag1
rxdone1


;
; check for circ buffer send
            test    tx_count
            jnz     end_int              ; busy?
            cje   head,tail,end_int1     ; nothing to send
; are we allowed to send?
IF INVSEND
```

```
            jnb   enablepin,end_int1
ELSE
            jb    enablepin,end_int1
ENDIF
            mov   fsr,tail
            add   fsr,#scan
            mov   w,ind
;send byte
            bank  serial
            not     w               ;ready bits (inverse logic)
            mov     tx_high,w       ; store data byte
            setb    tx_low.7        ; set up start bit
            mov     tx_count,#10    ;1 start + 8 data + 1 stop bit
            inc   tail
            and   tail,#bufmod      ; circularize
IF INVRCVA
            setb  rxen_pin
ELSE
            clrb  rxen_pin
ENDIF
; if transmitting why check alt channel?
            jmp end_int

end_int1
; are we allowed to send alt channel?
IF INVSEND
            jnb   enablepin1,end_int
ELSE
            jb    enablepin1,end_int
ENDIF
            mov   fsr,tail1
            add   fsr,#scan1
            mov   w,ind
;send byte
            bank    serial
            not     w               ;ready bits (inverse logic)
            mov     tx_high,w       ; store data byte
            setb    tx_low.7        ; set up start bit
            mov     tx_count,#10    ;1 start + 8 data + 1 stop bit
```

```
            inc   tail1
            and   tail1,#bufmod     ; circularize
IF INVRCVB
            setb  rxen_pin1
ElSE
            clrb  rxen_pin1
ENDIF


end_int     mov     w,#-int_period
            retiw                           ;exit interrupt


; ****** Main program begin

start_point
; want pull ups on all
      mode  $E
      mov   !ra,#0  ; pull ups on
      mov   !rb,#0  ; pull ups on
      mode  $F
IF INVSEND
      mov   ra,#%0011
ELSE
      mov   ra,#%1011             ;initialize port RA
ENDIF
      mov   !ra,#%0011             ;Set RA in/out directions
      mov   rb,#%00001010
      mov   !rb,#%00000101

warmboot
            CLR     FSR         ;reset all ram starting at 08h
:zero_ram   SB      FSR.4       ;are we on low half of bank?
            SETB    FSR.3       ;If so, don't touch regs 0-7
            CLR     IND         ;clear using indirect addressing
            IJNZ    FSR,:zero_ram  ;repeat until done

      mov     !option,#%10011111      ;enable rtcc interrupt

      clr rb
; ****
```

```
; Here is where the action is!
mainloop
      jnb   rx_flag,:t1
      call  get_byte     ; if char, copy to buffer
      call  enqueue
:t1
      jnb   rx_flag1,mainloop
      call    get_byte1 ; if char, copy to buffer
      call  enqueue1
      jmp   mainloop

enqueue
   ; check for buffer overrun!
       mov w,#1
       add w,head
       and w,#bufmod
       mov w,tail-w
       jz queuefull   ; if full too bad
       mov fsr,head
       add fsr,#scan
       mov ind,byte
       inc  head
       and  head,#bufmod  ; circular

; calculate buffer limit
  mov tmpvar,tail
  cjae tail,head,:normal
  add tmpvar,#16
:normal
  mov w,head
  sub tmpvar,w
  jz doret       ; buffer is empty?
  add tmpvar,#-BUFFERLIM
  jz :hshalt
  jc doret

:hshalt   ; buffer full so...


IF INVRCVA
```

```
      clrb rxen_pin
ELSE
      setb rxen_pin
ENDIF
doret
queuefull
       ret


enqueue1
   ; check for buffer overrun!
       mov w,#1
       add w,head1
       and w,#bufmod
       mov w,tail1-w
       jz queuefull1   ; if full too bad
       mov fsr,head1
       add fsr,#scan1
       mov ind,byte
       inc  head1
       and  head1,#bufmod  ; circular


; calculate buffer limit
  mov tmpvar,tail
  cjae tail,head,:normal
  add tmpvar,#16
:normal
  mov w,head
  sub tmpvar,w
  jz doret       ; buffer is empty?
  add tmpvar,#-BUFFERLIM
  jz :hshalt
  jc doret

:hshalt  ; buffer full...


IF INVRCVB
      clrb rxen_pin1
ELSE
```

```
      setb rxen_pin1
ENDIF
queuefull1
       ret


; Subroutine - Get byte via serial port
;
get_byte
            bank  serial
            jnb     rx_flag,$          ;wait till byte is received
            mov     byte,rx_byte       ;store byte (copy using W)
            clrb    rx_flag            ;reset the receive flag
            ret

get_byte1
            bank  serial
            jnb     rx_flag1,$        ;wait till byte is received
            mov     byte,rx_byte1     ;store byte (copy using W)
            clrb    rx_flag1          ;reset the receive flag
            ret
```

The SSIB Test Program

```
' Program to test SSIB
baudrate con 32

' Use the next 2 lines when using inv mode serial
' low 12
' low 13
' Use next 2 lines when using non inv mode serial
high 12
high 13
' Read starting numbers
debug "sync A "
serin 14\12,baudrate,[dec w3]
debug "B "
serin 14\13,baudrate,[dec w4]
debug "Complete",cr


top:
w3=w3+1   ' calculate expected next numbers
w4=w4-1
pause 1000   ' do some "work" (pause really)
' read numbers
serin 14\12,baudrate,[dec w1]
serin 14\13,baudrate,[dec w2]
debug "A:",dec w1,cr
debug "B:",dec w2,cr
' see if they met our expectations
if w1=w3 then testb
debug "Channel A mismatch. Expected ",dec w3, " got ", dec w1,cr
w3=w1
testb:
if w2=w4 then top
debug "Channel B mismatch. Expected ",dec w4, " got ", dec w2,cr
w4=w2
goto top
```

Simulated Serial Devices for the SSIB

```
' This program just writes out two
' data streams to test the SSIB
w1=0
w2=$FFFF
top:
serout 15\9,84,[dec w1,","]
serout 8\10,84,[dec w2,","]
w1=w1+1
w2=w2-1
pause 5
goto top
```


Exercises

1. If you wanted to add more serial channels to the SSIB, what points would you need to consider?
2. Devise a scheme to buffer 32 characters instead of 16. Show code to increment and decrement the pointer to the buffer.
3. Could you make the SSIB automatically detect the correct polarity of the input lines? What would be the plusses and minuses to doing this?

Answers

1. Adding another channel to the SSIB would require more program memory and data memory for the circular buffer. Of course, you'd also need addition I/O pins. However, the biggest limitation to adding another channel would be placing more code in the ISR. Remember, if the ISR's execution time exceeds the periodic interrupt rate, the code will not function properly. Also, as the ISR consumes more time it leaves less time for the remainder of the program. So if the ISR rate is, for example, 100uS and the ISR requires 80uS this leaves only 20uS for the remainder of the program.

   Of course, you can always move to a 28-pin device for more pins. The SSIB is not over taxing the part's memory. You could solve any potential ISR problems by increasing the part's speed so that you can execute more instructions in the same amount of time (of course, this increases current consumption).

2. Buffering 32 characters is somewhat complex because of the SX's banked architecture. Remember that the SX has 8 banks of 32 registers. However, the first 16 registers are the same in each bank. Of those 16 registers, 7 or 8 (depending on the device type) are reserved for system functions. The remaining 8 or 9 registers are usually used for variables that you have to frequently access so you can avoid bank switching.

   The current serial buffers are at addresses $50 and $70. If you try to grow these buffers arbitrarily you'll run into trouble. For example, $50 + $10 = $60, but $60 is really the **IND** register (the same as location $00).

   Suppose you decided to store the buffer for the first channel in two parts, one at $50 and one at $70 (you can move the other buffer to another address). When you increment the **head** or **tail** variable you'll have to take this into account:

```
      inc head
      cjne head,#$60,:nospan
      mov head,#$70
:nospan
      cjne head,#$80,:doneinc
      mov head,#$50
:doneinc
```

To decrement, you'd need this code:

```
      dec head
      cjne head,#$4F,:nospand
      mov head,#$7F
:nospand
      cjne head,#$6F,:doned
```

```
      mov head,#$50
:doned
```

3. Detecting the state of the line would require you to sense the input lines at some point when they were idle. For example, on reset you could read the serial input lines and assume they are idle. Then you could invert or not invert your inputs as appropriate. The problem is, what happens if the lines are not idle? You could erroneously sample a start bit, for example, and then you'd pick the wrong polarity.

   When designing a general-purpose component, you need to take great care that your devices will work under a variety of conditions. Therefore, this method is probably not appropriate since it could fail in certain cases that are likely to occur, at least for some users.

   A better idea would be to reserve an otherwise unused input pin and sense it on reset. The designer using your chip could then tie the input high or low to set the chip's polarity. This would be a must if you were not providing the source code with the part. Currently, the only way to change polarity is to recompile the source code. Some users won't be able to do this, and you may be unwilling to release your source code anyway.

Rev1.