Beginning Assembly Language for the SX Microcontroller

Version 2.0

PARALLAX INC

ISBN 1-928982-29-8

DISCLAIMER OF LIABILITY

WEB SITE AND DISCUSSION LISTS

The Parallax Inc. web site (www.parallax.com) has many downloads, products, customer applications and on-line ordering for the components used in this text. We also maintain several e-mail discussion lists for people interested in using Parallax products. These lists are accessible from www.parallax.com via the Support → Discussion Groups menu. These are the lists that we operate:

- BASIC Stamps – This list is widely utilized by engineers, hobbyists and students who share their BASIC Stamp projects and ask questions.
- Stamps in Class – Created for educators *and* students, subscribers discuss the use of the Stamps in Class curriculum in their courses. The list provides an opportunity for both students and educators to ask questions and get answers.
- Parallax Educators –Exclusively for educators and those who contribute to the development of Stamps in Class. Parallax created this group to obtain feedback on our curricula and to provide a forum for educators to develop and obtain Teacher's Guides.
- Parallax Translators – The purpose of this list is to provide a conduit between Parallax and those who translate our documentation to languages other than English. Parallax provides editable Word documents to our translating partners and attempts to time the translations to coordinate with our publications.
- Toddler Robot – A customer created this discussion list to discuss applications and programming of the Parallax Toddler robot.
- SX Tech – Discussion of programming the SX microcontroller with Parallax assembly language tools and 3rd party BASIC and C compilers.
- Javelin Stamp – Discussion of application and design using the Javelin Stamp, a Parallax module that is programmed using a subset of Sun Microsystems' Java® programming language.

ERRATA

While great effort is made to assure the accuracy of our texts, errors may still exist. If you find an error, please let us know by sending an email to editor@parallax.com. We continually strive to improve all of our educational materials and documentation, and frequently revise our texts. Occasionally, an errata sheet with a list of known errors and corrections for a given text will be posted to our web site, www.parallax.com. Please check the individual product page's free downloads for an errata file.

Table of Contents

Preface

Introduction

Very few people are interested in microprocessors (or even computers) *per se*. Instead, they are interested in specific tasks that the microprocessor can perform. You don't really want a microprocessor. You want a robot, or a data collection system, or an alarm system. However, building these exciting applications requires knowledge of hardware, software, and the microprocessor's capabilities.

The SX chip's Flash memory can be erased and reprogrammed more than 10,000 times. This allows users the luxury of trial and error with their assembly language programs. Coupled with the powerful and inexpensive 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.parallax.com.

This book is a compilation of two earlier books: *Introduction to Assembly Language Programming with the SX Microcontroller* and *Introduction to I/O Control with the SX Microcontroller* both by Al Williams. This new edition has been updated to use with the SX 28, and features improved formatting and graphics.

The first portion of this course will introduce you to the SX microcontroller's internal architecture as well as show some basic hardware and software concepts. Topics include: number systems, programming and debugging, flow control, math, basic I/O, interrupts, and virtual peripherals.

The second half of the book will cover more advanced I/O programming and expand on several advanced topics. Activities include RS-232 communication, pulse width modulation, an A/D converter, and a serial input buffer.

One of the things that makes the Ubicom 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.

Unit 1: Getting Started

Back in 1943, the chairman of IBM predicted that one day there would be a world-wide market for five computers. Today, computers are everywhere. Sure, there are PCs in many homes, but the real computer invasion isn't in the home PC. Instead, people buy computers in just about every electronic device they own. Today your television, your phone, your microwave oven, and your car all have computers (some have several computers).

These computers may not be as obviously powerful as your desktop PC, but they are designed to control the real-world. An integral part to designing electronic equipment today (for fun or for profit) is understanding how these devices work and how you can use them in your own creations.

Why use these microcontrollers? Often a microcontroller can replace a large number of other components. For example, consider a phone answering machine. Do you really need a microcontroller to do the job? No. If fact, many old fashioned answering machines did not use microcontrollers. Instead they had a circuit to detect a ringing phone. The ringing would activate a timer chip (or in a really old machine a timing cam on a motor). This timer would trip a relay that would take the phone off the hook. Then another timer would start the tape player that played the outgoing message. When the outgoing message finished (based on time, or sensing foil at the end of the tape), another timer would start a regular tape recorder for a preset time to record the call.

Instead of three timers, today's answering machine uses a microcontroller. With just a few external parts, the microcontroller can operate the entire system with ease. But there is much more. A microcontroller can also sense if someone is really talking on the other end of the line. It can accept Touch-Tone commands to allow remote control. It can even store and playback voice digitally instead of using tapes. Try making a sophisticated remote control without a microcontroller.

So our microcontroller phone machine is much more powerful than its ancestors. It also costs less. Microcontrollers are now quite inexpensive - even if you don't account for the number of parts it can replace. Fewer parts also make devices smaller, cheaper, and less prone to failure.

About This Course

This course is all about incorporating these powerful little computers - microcontrollers - into your own designs. Particularly, we will use the Ubicom SX microcontroller along with the SX-Key development system from Parallax. The SX is an inexpensive yet very powerful microcontroller. The SX-Key allows you to program the SX and also debug your programs in real-time. In the past, hardware like the SX-Key that allowed you to debug your program while the processor was in a real circuit was very expensive (thousands of dollars) and was only available to well-stocked labs. However, the SX-Key is quite affordable (only a few hundred dollars, depending on options).

To get the most out of this course, you should already be familiar with elementary hardware design. You should understand how LEDs work, for example, and understand basic electronic laws (like Ohm's law). This course will focus on designing programs to run the microcontroller and thereby control electronic circuits. Although you usually think of programs as software, when a program is inside a microcontroller it is often known as firmware - a cross between hardware and software.

The labs in this course are best performed with the SX-Tech Board available from Parallax. However, you can also wire up your own version of these circuits on any solderless breadboard (See Appendix B).

The SX chip is a very powerful chip, but is also useful as a learning tool. Unlike some microcontrollers, the SX uses electrically erasable memory to store programs. That means that you can write a program, try it, and then reprogram the chip immediately to run a different program (or a corrected version of the same program). This coupled with the powerful SX-Key tools provides an ideal environment for learning and experimentation.

Start at the Beginning

If you are not familiar with the way a computer operates internally, it can seem like black magic. It seems as though the little chips can do practically anything, no matter how complex. However, beneath this complexity is a surprise. The microcontroller operates very simply. This simplicity means that you - the programmer - have to take great pains to create these complex behaviors. Programming requires logical thought and attention to detail.

All programs operate by using a program, or a stored sequence of instructions. These instructions tell the computer what to do. When the computer first starts, it looks at these instructions in sequence. Some instructions read inputs. Others control outputs. Still other instructions do some sort of processing.

The Ubicom SX uses a Harvard-style architecture. This means that it has one area where it remembers instructions and another area where it remembers data (including inputs and outputs). This is a common architecture for microcontrollers (although some computers utilize a Von Neumann architecture where data and instructions mix together).

Suppose you started a new job at a factory that makes radios. The plant manager gave you the following instructions:

1. Put an empty crate at the base of the conveyor belt.
2. Flip the big red switch on to start the conveyor belt.

3. Watch for completed radios to come off the conveyor belt and into the crate. For each radio, click your handheld counter.
4. When the counter reaches 10, flip the switch again to stop the conveyor belt.
5. Move the crate and replace it with a new empty crate.
6. Reset the counter in your hand.
7. Go back to step 2.

This is exactly like a computer program. It is a sequence of steps. It has inputs (you deciding that a radio came off the conveyor belt). It has outputs (you flip the big red switch, for example). It also has processing in the form of counting and making decisions. In fact, this is just the kind of job a computer excels at.

Problem #1

There is a slight problem. Outside of Star Trek computers don't understand ordinary instructions like this. How do you instruct the computer to perform these steps? Every computer, from the smallest microcontroller to the largest supercomputer, stores its instructions in the form of numbers. Even worse, computers store these numbers using base 2 arithmetic (binary, a subject covered later in this unit). That means that a computer program looks like a series of 1's and 0's. This is called machine language, and is the basis of every computer program.

Of course, base 2 numbers are not easy for humans to understand, so people usually write the numbers in a more manageable system. However, even then it is hard to comprehend a program written only in numbers. For this reason, engineers typically use some more convenient method of expressing programs.

The most common way to program microcontrollers is using *assembly language*. This is a short hand method that allows abbreviations to stand in for the 1's and the 0's. You might use instructions like **ADD** or **JMP** (jump). In the old days you'd manually convert this shorthand into 1's and 0's, but today a special program known as an *assembler* does it for you. Of course, the microcontroller can't run this program, but your PC can. This is often called *cross assembling* - using one computer to assemble (convert from shorthand instructions to 1's and 0's) code for another computer. The short hand abbreviations, by the way, are known as *mnemonics*.

Many people find it daunting to program using these low-level instructions. Even though mnemonics are easy to read, they still represent the machine language, which is very simple. For example, the typical microcontroller can't directly multiply and divide numbers. Instead they calculate these operations using addition and subtraction. For this reason, some programmers turn to high level languages like BASIC or C - languages you might be familiar with from other computer systems.

If BASIC and C are available for microcontrollers, why use assembly language or machine code? The answer is efficiency. Microcontrollers generally have limited amounts of memory. Also, you often need them to perform as fast as possible. A program that uses a high level language will often consume more memory than a well-written assembly program. It may also run more slowly.

If you do use BASIC or C, you can count on the major portion of the language to run on your PC. This is similar to cross compiling. You write you C program on the PC and the PC converts your program into machine language. Parallax makes a successful line of products known as BASIC Stamp® microcontroller modules that use the PC to convert code into a quasi-machine language. The BASIC Stamp module then executes a program that interprets this quasi-machine language to perform the programming steps.

Different types of microcontrollers have different machine languages. However, most people find that if they learn one microcontroller's language, others are relatively easy to learn.

Problem #2

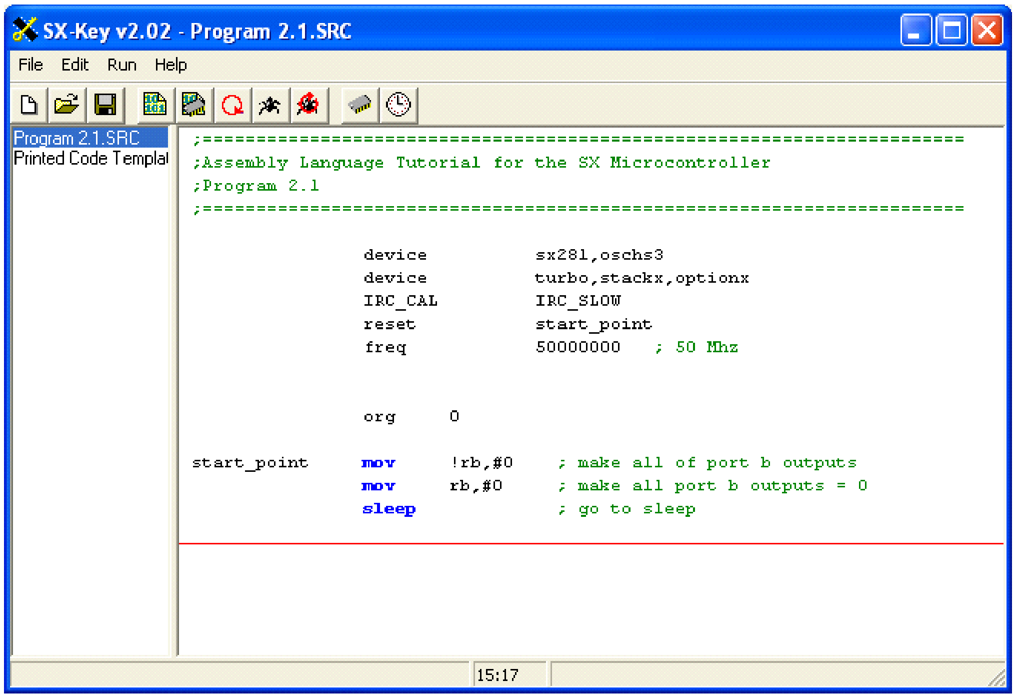
The next problem is what to do with the 1's and 0's once you have them. Somehow, you have to move these 1's and 0's into the computer. Older microprocessors used an external memory chip but modern processors have memory on board that you program with a special device known as a *programmer*. Some microcontrollers require ultraviolet light to erase the memory but the SX is instantly reprogrammable so you don't need to wait for a special light to erase the part.

In a Harvard architecture microcontroller, you can't change the program code while the microcontroller is running. Many microcontrollers can't even read data from their program storage while executing a program. However, the SX has a special feature that allows you to read data from the program's memory while running. This can be useful for storing constants, for example.

Watch Your Language

In this course, you'll use assembly language to program the SX. However, if you are familiar with BASIC or C you'll find parallel code examples to help you visualize the assembly code.

The Parallax BASIC Stamp module uses a particular variant of BASIC known as PBASIC. The BASIC code will mimic the BASIC Stamp module's language so you can apply the same concepts with the BASIC Stamp. There are several models of BASIC Stamp modules, and one of them, the BS2sx, has a SX microcontroller in it. However, you must program BASIC Stamp modules using PBASIC -- you can't use machine language. On the other hand, you might wonder why you'd want to use machine language if you could use BASIC. The truth is, BASIC is great, but some jobs require the speed and capabilities you can only get with machine code.



```
;======================================================================
;Assembly Language Tutorial for the SX Microcontroller
;Program 2.1
;======================================================================

                device          sx28l,oschs3
                device          turbo,stackx,optionx
                IRC_CAL         IRC_SLOW
                reset           start_point
                freq            50000000   ; 50 Mhz


                org     0

start_point     mov     !rb,#0     ; make all of port b outputs
                mov     rb,#0      ; make all port b outputs = 0
                sleep              ; go to sleep
```

Figure 1-1: The SX-Key Editor

The Working Environment

Figure 1-1 shows the main screen of the SX-Key software. Looks like a common text editor, and at this point it is. You can enter assembly language code in the window. When you want to test or run your program you can use the Run menu to check your syntax or program the SX chip. To just check your code for simple errors, use the Run|Assemble menu. You can also use Run|Run to execute the code (assuming you have the chip connected to the SX-Key hardware).

The assemble command only checks for simple syntax errors. Logic errors are up to you to find (with help from the debugger).

Is That It?

The real power of the SX-Key is not entering code. The impressive part is when your code doesn't work. Then you can use the Run|Debug command.

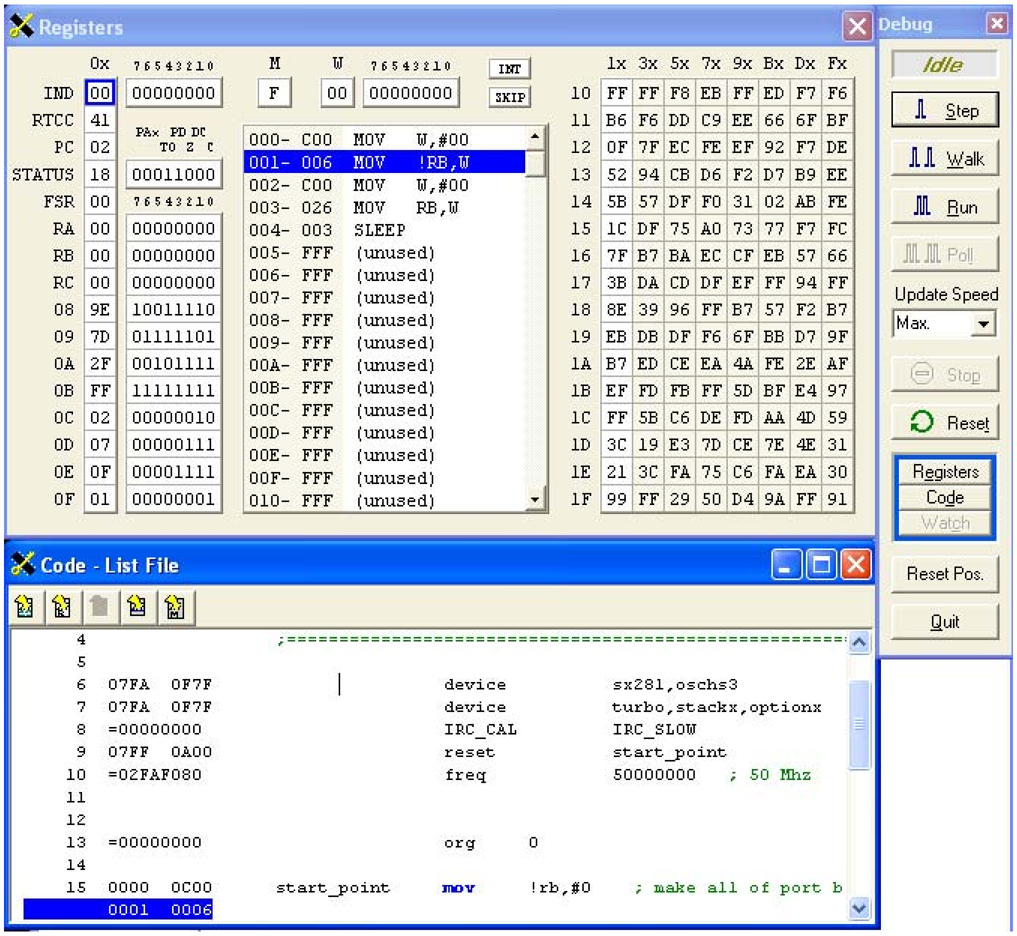
The debugger (see Figure 1-2) allows you to watch the SX execute your program one step at a time and examine its internal workings.

The Development Cycle

As you might imagine, such powerful tools greatly simplify programming. However you still need a plan. There is an old saying: "People don't plan to fail, they fail to plan." This is especially true when programming.

Earlier you read that programs read input, process it, and produce output. This is not a bad place to start when designing your software. Complex projects may require more rigorous design techniques, but many times this simple approach is enough. However, nearly every program (especially those for microcontrollers) will follow this model. Identifying your inputs, outputs, and processing is a solid first step towards realizing your design.

The next step depends on your background, experience, and personal preferences. You might start by making a list of instructions similar to the assembly line steps mentioned earlier. Some people prefer to draw the steps of their programs using boxes like a flowchart.



```
         0x   76543210      M     W   76543210
IND      00   00000000      F     00  00000000
RTCC     41   PAx PD DC
PC       02      TO Z C
STATUS   18   00011000
FSR      00   76543210
RA       00   00000000
RB       00   00000000
RC       00   00000000
08       9E   10011110
09       7D   01111101
0A       2F   00101111
0B       FF   11111111
0C       02   00000010
0D       07   00000111
0E       0F   00001111
0F       01   00000001
```

```
000- C00  MOV    W,#00
001- 006  MOV    !RB,W
002- C00  MOV    W,#00
003- 026  MOV    RB,W
004- 003  SLEEP
005- FFF  (unused)
006- FFF  (unused)
007- FFF  (unused)
008- FFF  (unused)
009- FFF  (unused)
00A- FFF  (unused)
00B- FFF  (unused)
00C- FFF  (unused)
00D- FFF  (unused)
00E- FFF  (unused)
00F- FFF  (unused)
010- FFF  (unused)
```

	1x	3x	5x	7x	9x	Bx	Dx	Fx
10	FF	FF	F8	EB	FF	ED	F7	F6
11	B6	F6	DD	C9	EE	66	6F	BF
12	0F	7F	EC	FE	EF	92	F7	DE
13	52	94	CB	D6	F2	D7	B9	EE
14	5B	57	DF	F0	31	02	AB	FE
15	1C	DF	75	A0	73	77	F7	FC
16	7F	B7	BA	EC	CF	EB	57	66
17	3B	DA	CD	DF	EF	FF	94	FF
18	8E	39	96	FF	B7	57	F2	B7
19	EB	DB	DF	F6	6F	BB	D7	9F
1A	B7	ED	CE	EA	4A	FE	2E	AF
1B	EF	FD	FB	FF	5D	BF	E4	97
1C	FF	5B	C6	DE	FD	AA	4D	59
1D	3C	19	E3	7D	CE	7E	4E	31
1E	21	3C	FA	75	C6	FA	EA	30
1F	99	FF	29	50	D4	9A	FF	91

```
                        ;=====================================================

07FA   0F7F                          device         sx28l,oschs3
07FA   0F7F                          device         turbo,stackx,optionx
=00000000                            IRC_CAL        IRC_SLOW
07FF   0A00                          reset          start_point
=02FAF080                            freq           50000000    ; 50 Mhz


=00000000                            org     0

0000   0C00     start_point          mov     !rb,#0     ; make all of port b
0001   0006
```

Figure 1-2: The SX-Key Debugger in Action

Once you have an idea of what your program will look like you can make your first pass at entering the program into the SX-Key editor using the assembly language instructions you'll learn in the following units. Your first attempt at running the program might work, but it isn't very likely. When things don't go as planned you'll turn to the debugger for a better understanding of your program's operating.

Even if your program works you may still want to use the debugger to study its operation. Sometimes you will see improvements you missed when thinking about the program in the abstract.

Number Systems

When normal people count they use base 10 or decimal. However, computers like to use binary or base 2. Programmers have to switch between the two and often use other systems as well.

When you say 138 (in decimal) you really mean:

$1\times100+3\times10+8\times1$

Decimal digits range from 0 to 9.

Binary numbers are similar, but they use only two digits: 0 and 1. The binary number 1001 is really:

$1\times8+0\times4+0\times2+1\times1 = 9$.

You can see how easy it is easy to convert from binary to decimal. Just remember that each digit is worth double what the digit to the right of it is worth.

Example:

10011110 = 2+4+8+16+128 = 158

Going the other way is a little more difficult. The trick is to determine which binary digit (known as a *bit*) is the largest necessary to represent the number. Consider the decimal number 122. The right-most bit in any binary number is always worth 1. The next bit is worth 2 then 4, 8, 16, 32, 64, 128, and so on.

Since 128 is bigger than 122, that bit can't be in the equivalent binary number. By convention, the right-most bit is considered bit 0 and the other bits are numbered sequentially from right to left. So the bit with the value of 128 is bit 7.

However, bit 6, with a value of 64, will have a 1 in the answer since 64 is less than 122. Since 122-64=58 you'll still have to account for this amount. The next bit's value is 32 and 32 is less than 58, bit 5 will also have a 1. The remainder is 58-32=26.

Bit 4 is worth 16 and so it will also be a 1 leaving 10. Bit 3 (8) will also contain a 1 leaving 2. Now consider bit 2. It has a value of 4 but this is greater than the remaining value and so it will contain a 0. The next bit is worth 2 so it will be a 1 and it leaves a remainder of 0. Therefore, all the bits to the left (in this case, only bit 0) will have a zero value.

So the answer is that 122 = 1111010. You can check your work by reversing the conversion. In other words:

$$1\times64+1\times32+1\times16+1\times8+0\times4+1\times2+0\times1 = 122.$$

It should be obvious, but you can add as many zeros as you like to the left of a binary number (or any number for that matter). So 1111010 and 01111010 and 0000000001111010 are all the same number.

Other Places, Other Bases

Since most people use decimal you have to use it sometimes. But many times it is easier to use other notations that are easier to convert to binary. The most common alternate base is *hexadecimal* or base 16.

Hexadecimal (commonly known as *hex*) uses 16 digits -- 0 to 9 and A-F. You can find the values in Table 1-1. Notice that to convert between binary and hex you can simply use the table. So F3 hex is 11110011 binary.

In hexadecimal each digit is worth 16 times more than the one before. So F3 hex is:

$$15\times16+3\times1=243$$

And 64 hex is:

$$6\times16+4\times1=100.$$

Table 1-1: Hexadecimal Digits

Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Many calculators, including the CALC program in Windows, can convert between bases automatically.

Say What You Mean

With these different ways of writing numbers, it is easy to get confused. Even the SX-Key assembler can't magically guess which number system you are using. That's why it is important to specify exactly what kind of number you are writing.

To specify the number system in use, you write numbers with special prefixes. A number that begins with a $, for instance, is a hex number. Binary numbers begin with a % character. Since decimal numbers are the most common numbers, they don't have a prefix.

Not all assemblers use this naming convention. For example, some assemblers use suffixes to indicate the number type. Others use different prefixes. However, the SX-Key assembler you will use in this course will use the prefixes as indicated.

Size Matters

Another concern with numbers is how many bits they occupy. The SX uses an 8-bit word size for data. This is often called a *byte*. The problem with bytes is that they can only hold numbers from 0-255. What if you need bigger numbers? Or negative numbers? Then you'll need to resort to special techniques found in Unit 6.

Remember, by convention, you number bits starting at the right-most bit. So the right-most bit is always bit 0. The left-most bit in a byte is bit 7. This is somewhat confusing because bit 7 is actually the eighth bit (because you started counting at 0 instead of 1).

Incidentally, although the SX uses an 8-bit word for data, its instructions are 12 bits wide. Since the Harvard architecture separates code and data, this isn't a problem, as it would appear to be.

The Hardware Connection

Of course, as nice as the SX-Key is, it is only a means to an end -- programming the actual SX chip!

The SX is an especially speedy processor. It can run at speeds up to 100 MHz and can execute most instructions in a single cycle (10 ns per instruction). In a real project, you must supply a crystal or a ceramic resonator for speeds greater than 4 MHz. However, when working with the SX-Key it provides the clock (you can change the clock speed using the Run|Clock menu).

The SX comes in an 18-pin package and a 28-pin variant. The 18-pin device has 12 I/O pins and the 28 pin device sports 20 I/O pins. Both devices have 2 K of program storage and about 136 bytes of data storage (although future devices may have different amounts of memory). There is also a surface mount-only, 20-pin device that is about the same as the 18-pin SX. When you write a 1 to an output pin, it generates (roughly) 5 V. If you write a 0 to the pin, it outputs 0 V. On input, the pins recognize voltages above a threshold (typically 1.4 V) as a 1 and below the threshold as a 0. You can make any pin an input or an output and you can even switch them during program execution.

Obviously, your choice of parts will often hinge on how many I/O pins you need. If you want to use, for example, 4 pins to drive an LCD display, and 8 pins to connect to a keypad, you won't have anything left over for other work if you use the 18-pin SX. However, for this course you may also be constrained by the experiment board you are using since it may only have a socket for one device or another.

You can find the hardware details of the SX in the official data sheet. However, you'll also read more about the SX hardware in the remaining units of this course.

Summary

The old saying goes: "The mightiest oak begins as a tiny acorn." In a similar vein, the simple functions of a microcontroller can build complex systems if you know how to use them.

To understand low-level computers like microcontrollers you have to speak their language -- or at least the shorthand assembly language and hex codes that most people use to represent the arcane machine language.

This unit -- by necessity -- covers these fundamentals. By now you should be itching to really use some hardware. You'll get your chance in the next unit.

Exercises

1. Convert the following numbers to decimal:
   (a) $27
   (b) %101110
   (c) $F1
   (d) $AA

2. Convert the following numbers to hexadecimal:
   (a) 100
   (b) 200
   (c) 17
   (d) %10110110
   (e) %1000001

3. Answer True or False to the following statements:
   (a) Programs consist of a series of steps.
   (b) All computers us a Harvard architecture.
   (c) A Harvard architecture computer uses separate memory for programs and data.

Answers

1. (a) 39 (b) 46; (c) 241; (d) 170

2. (a) $64; (b) $C8; (c) $11; (d) $B6; (e) $41

3. (a) True; (b) False; (c) True

Unit 2: Your First Program

By now, you are probably ready to jump in and start a project. Good, because that's exactly what you will do in this unit. You should have a PC running the SX-Key software connected to an SX-Tech board. If you don’t have an SX-Tech board you can use any other similar development board with some LEDs connected to port B so that they turn on when you output a 0 from the SX (see Figure 2-1). Connect an LED at least to two adjacent pins on the SX’s B port. If you are industrious, wire 8 LEDs, one to each pin on the B port.

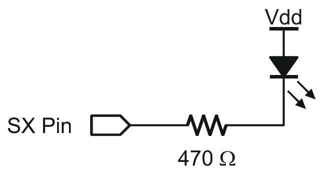


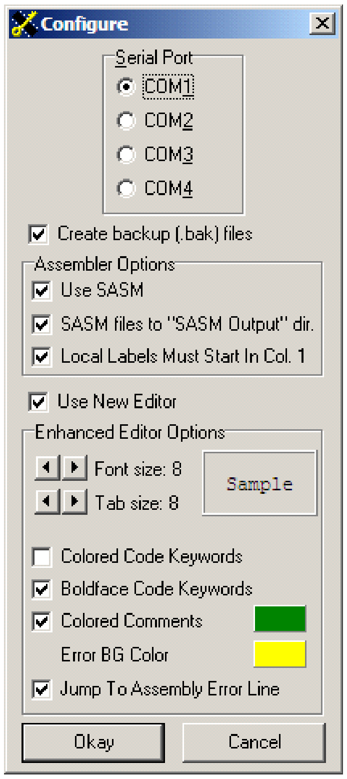
Figure 2-1: LED Circuit

To start with, you'll enter a program into the SX-Key editor, download it to an SX processor, and execute it. You'll see exactly what each part of the program means later in this unit. For now, just concentrate on getting familiar with the steps involved and your hardware setup.

First Step

If you haven't already, install the SX-Key software as instructed in the manual. The manual will also tell you how to start the program, and you should do so now. The first time you start the software it will show you a configuration screen Figure 2-2.

From this screen you must select the correct COM port that your computer is using for the SX-Key hardware. In addition, be sure to check Use SASM so that the SX-Key will use the newer SASM assembler. If you have already run the SX-Key software, you can check these options from the Run | Configure menu.



Configure

Serial Port
- (●) COM1
- ( ) COM2
- ( ) COM3
- ( ) COM4

[✓] Create backup (.bak) files

Assembler Options
- [✓] Use SASM
- [✓] SASM files to "SASM Output" dir.
- [✓] Local Labels Must Start In Col. 1

[✓] Use New Editor

Enhanced Editor Options
- Font size: 8
- Tab size: 8
- Sample
- [ ] Colored Code Keywords
- [✓] Boldface Code Keywords
- [✓] Colored Comments 🟩
- Error BG Color 🟨
- [✓] Jump To Assembly Error Line

Okay | Cancel

Figure 2-2: SX-Key Editor Configuration Screen

Once you dismiss the configure dialog (or you run the SX-Key software again), the initial screen is blank and you can enter your program (you can also, of course, load an existing program from disk).

What to enter? That's the problem! For now, enter the following simple program exactly as shown. Note that each line except the ones containing **start_point** is indented with a tab. This is a common practice in assembly language – placing labels (like **start_point**) in the first column, and placing commands to the right at least one tab.

```
;========================================================================
;Beginning Assembly Language for the SX Microcontroller
;Program 2.1
;========================================================================

               device         sx28l,oschs3
               device         turbo,stackx,optionx
               IRC_CAL        IRC_SLOW
               reset          start_point
               freq           50000000  ; 50 MHz


               org            0

start_point    mov            !rb,#0    ; make all of port b outputs
               mov            rb,#0     ; make all port b outputs = 0
               sleep                    ; go to sleep
```

The SX-Key software is not case sensitive, so it doesn't matter if you use upper case or lower case letter (you can change this behavior with the CASE directive, however). It is a good idea to save your program from time to time. If Windows freezes or crashes for some reason, you'll be glad you saved. Certainly you should save your work before you try to run the code on the SX.

When you finish entering the program, select Run|Assemble from the SX-Key menu. If you entered everything with out any mistakes, you'll see "Assembly Successful" In the status bar. Otherwise, you'll see a list of error messages (below the text window) and the cursor will jump to the first line containing an error. You can click on the error messages to show the corresponding source line. Fix the errors and try again. Keep in mind that sometimes one mistake will cause several errors. After you fix any obvious mistakes, you can always try to assemble your code again to see which errors you've actually fixed (and, in some cases, what new errors you've found or created).

At this point, the only thing the SX-Key software is doing is checking your program for syntax errors. It is still possible (and even likely) to make logical errors that the assembler can't catch. Think about a word processing program's spell checker. It can tell you if you spell 2 as "tew", but it can't warn you if you spelled it as "too" or "to". The assembler has the same problem. It can tell if you've made an obvious mistake, but it can't decide if you're program works as you expect it to operate.

Lock and Load

Once your program assembles correctly, you can download it to the SX chip. The most obvious way to do this is to use Run|Program. This assembles the program again and, if the assembly has no errors, loads the machine code to the SX chip. You can find more about the hardware setup in Appendix B. Of course, if you are using the SX-Tech board, you can also refer to its instructions for the hardware and software setup instructions

However, you may find it better to use the Run|Run menu item. This works just like the Program command, but it also starts the program running. If you've already used the Program command, you can just use the Run command again, or select Run|Clock to start running.

Either way you start running you should see the LEDs connected to port B light up. Not very exciting, but it is a start. At this point you know your hardware is working and your software is configured correctly. If things don't work right, here are a few things to consider:

Be sure the SX-Key is installed in the correct orientation. The markings on the SX-Tech board should match the legends on the SX-Key.
Be careful that the SX chip is installed correctly. On the SX-Tech board, pin 1 of the chip (designated by the dot on the IC package) is closest to the edge of the board.
If the LED on the SX-Tech board is not lit, or appears dim you may have a power supply problem.

Once you program the SX the chip retains the program until you reprogram it.

If you are guessing what the program is doing, you might wonder why the LEDs light up when the pins outputs a zero. This may seem counterintuitive, but it is a common practice. Although the SX can sink and source a considerable amount of current, many chips can sink more than they can source. Because of this, designers often wire LEDs and other loads so that they turn on with a 0 logic level.

So What?

On the face of it, this seems unimpressive. You can make LEDs light up with no microprocessor at all, right? So add the following line of code right before the line that has **sleep** in it:

```
mov    rb,#$AA   ; make every other port b output = 1
```

Now when you run the program, you'll see some lights on and some lights off. Is that correct behavior? After all, the program first turned all the lights on. Then it turned some of them off. Why can't you see all the lights turn on before some of them turn back off? The answer is that the SX chip is running each instruction in 20 ns! You'd have to have some pretty good eyes to see those LEDs light up for 20 ns.

However, if you could make the SX run instructions one at a time, you could see it. In fact, that is something the debugger can do. Before you dive into the debugger, however, let's take a look at what is happening inside this simple program.

Inside the Program

The easiest way to figure out what this simple program is doing is to examine it line by line. Along the way, you'll see some key concepts that you'll deal with in every program you write. The first two lines begin with the **device** keyword. This is not really an SX command. Instead, it is a directive to the assembler. Most keywords have some equivalent machine language value. However, directives don't generate machine code; they simply give the assembler instructions. In this case we want the assembler to know that we are writing a program where the target SX will be a SX28L and it will use the high-speed oscillator option (**oschs2**). The second line informs the assembler that we want to use several special modes that the SX supports. The assembler will use this information to burn the SX configuration fuses. These fuses control the chip's hardware settings and are not part of the actual program. The line that begins with **IRC_CAL** tells the SX-Key software to calibrate the SX's internal oscillator. However, we aren't using the oscillator, so this line tells the software to just set the oscillator to its slowest value.

The next line contains a **reset** directive. This informs the assembler where the program is to start executing. You might think that it would be logical for the program to start at the beginning, but you'll see later that this is not always the case. The name after the directive, **start_point**, is a user-defined label. This label can be any identifier you want and locates a spot in the program.

Labels and other identifiers can contain up to 32 characters. The first character must be a letter or an underscore. The other characters can be letters, underscores, or digits. You can't use reserved words (like sleep and reset) as an identifier.

The next line specifies the clock frequency in hertz. This doesn't really do anything for the SX chip, but it helps the debugger determine what clock frequency you want to use. If you don't specify a **freq** directive, the default is 50 MHz. You can also change the clock frequency for running programs using the Run|Clock menu. The assembler allows you to add underscores in any number like this to make it more readable. So you may see a similar line written like this:

```
        freq            50_000_000
```

The next line contains the final directive, **org** (which stands for *origin*). This directive instructs the assembler to begin generating code at a particular address. In this case, you want to start at the beginning so the **org** is 0.

The next 3 lines (or 4 if you've added the line of code that turns off some LEDs) are the actual program. The lines up to this point were simply directives to the assembler. The first program line starts with the **start_point** label. This is so the **reset** directive can refer to it. Notice that the label appears first on the line. The remainder of the line is the actual instructions for the microcontroller.

Registers

The data memory of the SX consists of a small number of byte-sized registers. Although there are well over 100 registers in the SX, your program can only work with 32 of them at a time. In a later unit, you'll learn about *banking* which allows you to get to all the registers, but for now, suffice it to say that there are 32 registers. Register $08 to $1F are available for you to store data. However, registers $00 to $07 are special because they control the SX chip as your program executes.

For example, register $05 corresponds to the SX's port A. When you read a value from register $05 (known to the assembler as the **ra** register), you are actually reading the digital signals present on port A's input pins. If you write to the **ra** register, you will alter the digital signals that appear on port A's output pins. You can also use $06 (**rb**) or $07 (**rc**).

This leads to another problem: How do you know which pins are inputs and which are outputs? Initially, all pins are set as inputs. However, your program can change this at any time by storing a special value into the port's direction register. To access the port's direction register you put an exclamation point in front of the register name. Writing a 0 to the direction register makes the corresponding bit an output. A 1 makes it an input.

Now the three lines of the program make more sense. The first line uses the **mov** (move) instruction. This instruction moves a zero into the port B direction register (**!rb**). Notice that the 0 has a # character in front of it. This marks it as a constant. Without this #, the instruction would move the contents of register 0 into **!rb**. You can add a base (or *radix*) specifier after the #, so #$FF is a hex constant and #%1011 is a binary constant.

The second line uses the same instruction, but now the destination register is **rb** instead of **!rb**. This writes the data out to the port. Since all the pins are outputs, each pin will now have a 0 V level. This causes the LED to light.

If you added the extra line of code, it writes $AA to the ports. This is the same as %10101010 so it alternates the LEDs. The final line, **sleep**, shuts the processor down in low power mode. You will rarely use this in a real program – at least, not in this way – most microcontrollers never just stop. Later, you'll see that you might want to sleep until some external event or time period wakes you up, but in this case the processor just sleeps forever – something almost unheard of for a microcontroller.

One other item you might notice in the program is the comments. These start with a semicolon and continue to the end of the line. You can use comments anywhere you want to make notes about the program's operation. This is a good idea in case anyone else has to read your work. It might even help you when you need to review your code 6 months down the road and you can't remember how things worked.

Another use for comments is to temporarily remove a line from your program. Just put a semicolon in front of the line you want to "delete" and later you can restore it by simply removing the semicolon.

If you were a PBASIC programmer, you might like to think of this program as similar to this:

```
DIRL = $FF
OUTL = $00
END
```

Notice that PBASIC uses a direction register just like the SX. However, the bit meaning is the opposite. In a BASIC Stamp program, direction bits of 0 set input pins, and a 1 sets the output pins. The SX is just the opposite.

Taken one piece at a time, this program isn't very complicated at all. However, there is an even better way to understand what it is doing: use the debugger.

Elementary Debugging

Once your program is running, you might like to try executing it with the debugger to see how it works. This will also give you practice using the debugger, something you are sure to need before long. To start, use the Run|Debug command. This is similar to the Run|Run command but it also loads a special debugging program into the SX chip. Normally, you don't know this program is present. However, you do have to have some free memory for the debugger or it won't work. In fact, the following requirements are necessary for debugging to work:

- No external clock (the SX-Key supplies the clock)
- Use the **RESET** directive
- No watchdog timer (covered later)
- 2 free instructions in the first bank of program memory
- 136 free instructions in the last bank of memory
- A **FREQ** directive, unless you want to run at 50 MHz, in which case **FREQ** is optional

After you press Run|Debug you'll see the usual programming windows. Then you'll see three windows open up. The Registers window contains the contents of the SX registers and a dump of the machine code you are executing. The Code window shows your source code (and the machine code to the left of that). Finally, the Debug window gives you a remote control to start and stop your program in a variety of ways.

In Figure 2-3 you'll notice that the Register window has the first 16 SX registers on the left-hand side of the screen. You'll notice the **RA**, **RB**, and **RC** registers, as well as the user registers $08 to $0F. The display is in hex, but directly to the right of each value is the same value in binary. The other registers (in hex only) are on the right-hand side of the Register window.

The center of the screen shows the machine language dump of your program. Notice that some instructions you write in your program actually generate more than one machine language instruction. For example, the line that reads:

```
mov    !rb,#0
```

Really generates:

```
mov    w,#0
mov    !rb,w
```

The debugger hides this from you in the code window, but you will notice that each line takes up more than one instruction. That's why in this program, the first line of code is at address 0, but the next line is at address 2. The multi-part instruction is consuming two words instead of the usual one. You can also see the instructions in the center portion of the Registers window.

Another peculiarity that appears in the Registers window is the first instruction of your program. You'll notice that although you instructed the assembler to start your program at address 0, the actual program starts at location $7FF (the top of memory). There is a single instruction at this address:

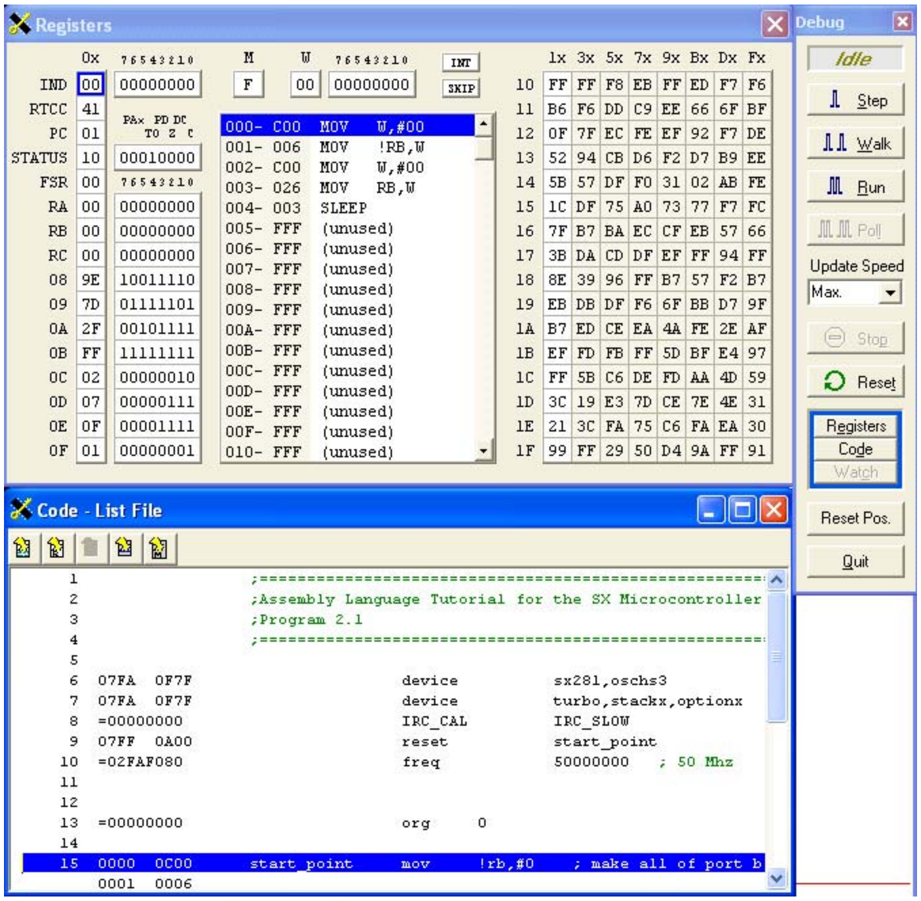
```
jmp    000
```

The program doesn't contain this instruction directly, but it is a result of using the **RESET** directive. The SX always starts execution at the highest program address, and this instruction (a jump) causes the processor to start executing the code at address 0. Notice that this instruction doesn't appear in the code window – that window simply shows the program as you entered it. The instructions in the Registers window shows the actual code that is inside the SX chip.

The **W** register (which appears near the top of the register window) is a special register often known as the *accumulator*. Practically all math operations occur in the **W** register.

There is no instruction to move a constant into the **!rb** register, so the assembler automatically used the **W** register. This can lead to program bugs if you don't keep it in mind. For example, consider this:

```
mov    w,#$AA
mov    !rb,#0
; Now w has 0 in it even though
; you think it has $AA in it!
```



```
        0x   76543210      M     W    76543210
   IND  00   00000000      F     00   00000000
  RTCC  41
    PC  01   PAx PD DC
               TO Z  C
STATUS  10   00010000
   FSR  00   76543210
    RA  00   00000000
    RB  00   00000000
    RC  00   00000000
    08  9E   10011110
    09  7D   01111101
    0A  2F   00101111
    0B  FF   11111111
    0C  02   00000010
    0D  07   00000111
    0E  0F   00001111
    0F  01   00000001
```

```
000- C00  MOV   W,#00
001- 006  MOV   !RB,W
002- C00  MOV   W,#00
003- 026  MOV   RB,W
004- 003  SLEEP
005- FFF  (unused)
006- FFF  (unused)
007- FFF  (unused)
008- FFF  (unused)
009- FFF  (unused)
00A- FFF  (unused)
00B- FFF  (unused)
00C- FFF  (unused)
00D- FFF  (unused)
00E- FFF  (unused)
00F- FFF  (unused)
010- FFF  (unused)
```

```
    1x 3x 5x 7x 9x Bx Dx Fx
10  FF FF F8 EB FF ED F7 F6
11  B6 F6 DD C9 EE 66 6F BF
12  0F 7F EC FE EF 92 F7 DE
13  52 94 CB D6 F2 D7 B9 EE
14  5B 57 DF F0 31 02 AB FE
15  1C DF 75 A0 73 77 F7 FC
16  7F B7 BA EC CF EB 57 66
17  3B DA CD DF EF FF 94 FF
18  8E 39 96 FF B7 57 F2 B7
19  EB DB DF F6 6F BB D7 9F
1A  B7 ED CE EA 4A FE 2E AF
1B  EF FD FB FF 5D BF E4 97
1C  FF 5B C6 DE FD AA 4D 59
1D  3C 19 E3 7D CE 7E 4E 31
1E  21 3C FA 75 C6 FA EA 30
1F  99 FF 29 50 D4 9A FF 91
```

```
                     ;=====================================================
                     ;Assembly Language Tutorial for the SX Microcontroller
                     ;Program 2.1
                     ;=====================================================

 07FA  0F7F                          device        sx281,oschs3
 07FA  0F7F                          device        turbo,stackx,optionx
 =00000000                           IRC_CAL       IRC_SLOW
 07FF  0A00                          reset         start_point
 =02FAF080                           freq          50000000   ; 50 Mhz


 =00000000                           org    0

 0000  0C00          start_point     mov      !rb,#0    ; make all of port b
 0001  0006
```

Figure 2-3: The Debugger Registers and Code Windows

The remote control has buttons that you can use to study your program:

- Hop – Executes one assembly language instruction (remember, this might be more than one machine instruction)
- Jog – Executes assembly instructions in slow motion letting you see the results as your program run slowly – press Stop to end Jog mode
- Step – Executes one machine language instruction
- Walk – Similar to Jog mode, but steps machine language instructions instead of assembly language instructions
- Run – Runs your program at full speed. The debugger can't examine registers until you press Poll or Stop
- Poll – This button only becomes active while running. It causes the debugger to freeze the processor momentarily, read the registers so you can view them, and resume program execution
- Stop – End a Jog, Walk, or Run command (only active when these commands are running)
- Reset – Starts the program over

As you step through your program, you'll see a highlight to indicate what instruction your program is executing. Also, registers that change value will appear in red. Other controls in the remote control include buttons to bring the other windows into view and a button to restore the other windows to their default positions. You can select from several update speeds (for the Walk and Jog commands). Of course, the Quit button exits the debugger.

Stopping the Debugger

This is a short program, so it is easy to step through it. However, this is not always the case. Many times, the area of your program you want to examine will be buried in the middle of a long program. Perhaps that piece of code only runs when an external event triggers it, or after a time delay. In this case, you'll want to set a *breakpoint*.

Simply put, a breakpoint is a stop sign in your program. When the SX tries to execute the line of code the breakpoint is on, the debugger takes control and the programs pauses execution. You can resume execution using the Debug remote control, either running the program or stepping through it.

The debugger supports one breakpoint at a time. To set a breakpoint, just click on the line you want to stop at (either in the Register or Code windows). The line will turn red. Now if you press Run (be sure to press Reset first if you've already run the program) the program execution will halt at the breakpoint. Setting a new break point clears any existing ones. If you want to clear all breakpoints, just click on the red line that already has a breakpoint.

You can also add a breakpoint in your assembly language program so that you'll always have a breakpoint set when you start debugging. You do this by adding a **BREAK** directive in your program like this:

```
mov    !rb,#0
break
mov    rb,#$FF
```

By the way, if you try to set a **BREAK** before a **sleep** instruction it won't work. If you have to do this, just use a **NOP** instruction after the break. **NOP** stands for no operation and the instruction does absolutely nothing but waste time. You may have to use this same trick when debugging code that loops to the same address using **jmp**.

Summary

So far you've read about four instructions, **mov**, **sleep**, **nop**, and **jmp**. There is more to learn about the **mov** instruction, but even then it is obvious you need more instructions to write any sort of useful programs. Still, even this small set of instructions allows us to control the output bits of the SX. In the next unit, you'll learn more about jumps and labels and build more functions into this simple program.

Exercises

1. Since each bit in the direction register stands for a different pin, it makes sense to specify the value for the direction register (and often for the port register itself) in binary. Rewrite the first example program in this unit to use binary numbers instead of hexadecimal numbers.

2. The **JMP** instruction transfers control to a different address. Can you replace the **SLEEP** instruction with a **JMP** back to the top of the program? Predict how this will affect the LEDs.

3. The problem with the program in this unit is that the LEDs change so fast, you can't see them without the debugger. Can you reduce the speed of the SX so you can visualize the LEDs when running without the debugger?

Answers

1. Change #0 to #%00000000 and #$AA to #%10101010

2. Change the **sleep** command to this:

```
            jmp   start_point
```

The LEDs now change rapidly over and over. You can't really see the lights change, but you'll notice that the lights that turn off appear somewhat dimmer than the ones that are on at all times.

3. Using Run|Clock, you can reduce the clock speed to 400 kHz. However, this is still not slow enough to see the LEDs change. Probably the best way to see what the program is doing is to use the Jog or Walk commands in the debugger.

Unit 3: Simple Flow Control

In the previous unit, you wrote and debugged a simple program. This program started at address 0, executed a simple set of instructions, and then went to sleep. While this was good to start with, it is clear that most microcontrollers don't execute a few commands and then stop – they run all the time, monitoring inputs and manipulating outputs.

In this unit, you'll extend the simple program from last time so that it does more interesting things. Along the way, you'll read about a few more simple SX instructions.

Running?

As you ran the last unit's program in the debugger, you may have notice that the **PC** register changed every time you executed a step. If you noticed a little more, you might have realized that the number in **PC** matched the address of the current machine language instruction. That's because **PC** is the *program counter*. This is a special register that tells the SX which instruction it will execute next.

Do you remember the first instruction you saw in the debugger? It was a **JMP** that the assembler automatically put in at the default reset address so that our program could start where we wanted it to. Of course, you can also write your own **JMP** instructions to control the flow of execution in your own program. This is like using a **goto** statement in BASIC or C.

In the last unit's exercises, you changed the **sleep** instruction to a **JMP** to cause the program to restart at the beginning instead of stopping. However, the obvious solution isn't as efficient as it could be. Here's the entire solution:

```
;=========================================================================
;Beginning Assembly Language for the SX Microcontroller
;Program 3.1
;=========================================================================

               device  sx28l,oschs3
               device  turbo,stackx,optionx
               IRC_CAL IRC_SLOW
               reset   start_point
               freq    50000000       ; 50 MHz


               org     0

start_point    mov     !rb,#0         ; make all of port b outputs
               mov     rb,#0          ; make all port b outputs = 0
               mov     rb,#$AA        ; change port b outputs
               jmp     start_point
```

What's wrong here? Nothing is actually wrong. However, the code as written keeps storing 0 in the direction register (**!rb**). There is no reason to do this. Once the direction register is set, there is no reason to keep setting it again. It doesn't hurt anything to reset it, but it wastes time that you could use to do something else.

The solution is simple. Just add another label to the line following **start_point**. Call it **again**. Then you can jump to **again** instead of **start_point**. So:

```
start_point  mov   !rb,#0       ; make all of port b outputs
again        mov   rb,#0        ; make all port b outputs = 0
             mov   rb,#$AA      ; change port b outputs
             jmp   again
```

Another way to make the program a bit more readable is to use the **CLR** instruction. The **CLR** instruction can set any normal register or the **W** register to 0. You can't use it with the **!rb** register though. This is also more efficient since using **MOV** to clear a normal register requires two instructions as opposed to a single instruction for **CLR** (remember, the **MOV** instruction may generate more than one instruction word, and in this case it generates two). Here is the code:

```
start_point  mov   !rb,#0       ; make all of port b outputs
again        clr   rb           ; make all port b outputs = 0
             mov   rb,#$AA      ; change port b outputs
             jmp   again
```

More Interesting?

To make the program more interesting, you'll need a few more instructions. Consider the **INC** (increment) instruction. The **INC** instruction adds 1 to a register. Since the port B pins look like a register (the **rb** register), you can increment it just like any other register.

Change your code to look like this:

```
start_point  mov   !rb,#0       ; make all of port b outputs
again        clr   rb
             inc   rb           ; change port b outputs
             jmp   again
```

In addition, change the **freq** line to read:

```
             freq  500000       ; 500 kHz
```

Normally, changing the frequency would also require changing the oschs3 clause in the DEVICE statement. However, in this case, the SX-Key provides the clock, so it is not necessary to change oschs3 to oscxt1.

What should this do? You'd like the program to cycle the lights in a binary pattern. So first all lights are on, then the LED on pin 0 turns off. Then it turns back on and the LED on pin 1 turns off. Just like counting in binary where on LEDs represent a 0.

That's what you'd like the code to do, but it won't work. Try it. When you run the code, the LEDs seem to stay on all the time. If you single step through the code, you'll see something a bit different. Use the debugger to determine what's wrong with the program (even if you've already figured it out) and then read the next section.

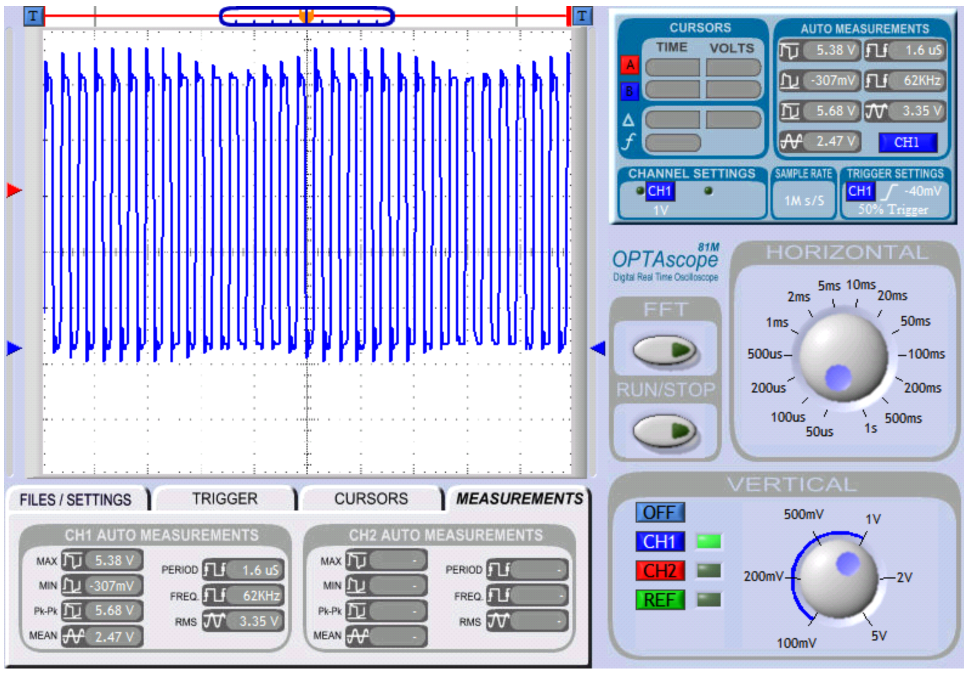
What's Wrong?

As you probably realized, the problem is that jumping to **again** makes the program reset the **rb** register to 0. If you didn't figure this out, go back and look at the program again. It is sometimes helpful to pretend you are the SX chip and execute the instructions in the order the processor does. In this case, you clear **rb**, increment it, and then immediately clear it again. So the increment has virtually no effect.

To fix this problem, move the **again** label to the next line like this:

```
start_point  mov   !rb,#0       ; make all of port b outputs
again        clr   rb
             inc   rb           ; change port b outputs
             jmp   again
```

Now the program works as you'd expect. If you have an oscilloscope, you might find it interesting to watch the port B pins. Bit 0 of port B will generate pulses of a certain width based on the system clock, as shown in Figure 3-1.



CH1 AUTO MEASUREMENTS

MAX	5.38 V	PERIOD	1.6 uS
MIN	-307mV	FREQ.	62KHz
Pk-Pk	5.68 V	RMS	3.35 V
MEAN	2.47 V		

CHANNEL SETTINGS: CH1 1V
SAMPLE RATE: 1M s/S
TRIGGER SETTINGS: CH1 -40mV 50% Trigger

Figure 3-1: Output Pulses

Bit 1 will emit pulses twice as long. Bit 2 will create pulses 4 times as long, and so on. Using the timings for each instruction provided in the SX data sheets, you can actually calculate these times. The **inc** instruction requires 1 clock cycle (2 μs at 50 MHz) and the **jmp** requires 3 cycles (6 μs at 50 MHz). So the pin will change every 8 μs. The total period is 16 μs, and the frequency should be 62.5 kHz. For practical purposes, you've created a square wave oscillator and a divider – all in software.

It is worth noting that the SX has two ways it can execute instructions: compatibility mode, and turbo mode. In compatibility mode, the SX requires more time to execute each instruction. For example, in compatibility mode, an **inc** requires 4 clock cycles. This make the SX compatible with programs written for other microcontrollers that may require a slower execution rate. All the programs in this tutorial use the **turbo** clause in the **device** statement, and therefore require about a quarter of the time to execute as they do in compatibility mode. For new programs you'll always want to enable turbo mode so you can get the best possible performance.

When programming microcontrollers, it is often necessary to compute the number of instructions that will execute so you can precisely set times. Sometimes you want to do this to set a time delay. Other times you'll be setting a frequency, as in this case. When you are fine-tuning your delays, you might find the **nop** instruction – an instruction that does nothing – useful. This instruction (**nop** stands for no operation) simply wastes 1 clock cycle (in turbo mode).

In PBASIC, by the way, you'd use a program similar to this:

```
DIRL=%11111111              'all outputs
OUTL=b1

DO
  b1 = b1+1
  OUTL = b1
LOOP
```

One thing to consider is what happens when some of the pins in port B are inputs (which they are not in this case). That could pose a problem since the increment instruction reads the port, increments the value it finds, and then writes the new value back to the port. When some pins are inputs, the instruction will read the input pins correctly and they will reflect the external stimulus placed on the SX chip. When you increment that, you may or may not get what you expect.

As an example, suppose that bit 7 was an input. When you write 0 to the port, that has no effect on bit 7. If port B's pin 7 has a logic low applied to it, the first **INC** instruction will work as you'd expect. It would read a 0 and write a 1 to the output. However, if the pin were high, the **INC** instruction would read a %10000000 and write %10000001. This probably wouldn't hurt anything, but there are cases where this is a problem. Always be wary of using instructions that read, modify, and write on I/O port registers.

Other Forms of JMP

The **jmp** instruction, by the way, has two other forms that you can use. First, you can use the **W** register (the accumulator) as the destination address. Just write:

```
        JMP    W
```

This is useful when you want to use a calculation to determine where to jump. The other form of **JMP** isn't a **JMP** at all. The **ADD** instruction allows you to add the **W** register to the **PC** register. This causes a jump over a certain number of instructions. Of course, the **ADD** instruction really just adds the **W** register to any other register. It just so happens that changing **PC** causes a jump. For example, consider this:

```
             CLR    8             ; clear register 8
             MOV    W,#2
             ADD    PC,W
             INC    8
             INC    8
             INC    8
             BREAK                ; what is in reg 8 now?
             INC    8
```

When the debugger hits the breakpoint, register 8 contains 1 because changing **PC** causes the first two **INC** instructions to not execute. The assembler allows you to write this instruction as **JMP PC+W** to make your program easier to read.

Since the 2 in this example is a constant, you really could use a regular JMP instruction to skip these two instructions. One way, of course, would be to label the target of the JMP. However, you can also use the special label $ which means the current address. So you could write jmp $+3 instead. Why +3 instead of +2? Since $ refers to the current address, you have to add 1 just to get to the next instruction. Adding 2 would only skip 1 instruction.

The real value to using ADD to perform a jump is when you compute the offset at run time. This allows you to create data and jump tables as you'll see later in this tutorial.

In this example, using 8 as a register number is confusing. Remember, it isn't a constant because it didn't start with a #. However, it is much nicer to name your variables in a meaningful way. The assembler provides you a couple of ways to do this that you'll read about in the next unit.

Of course, sometimes you want to jump only if some condition is true of false. For example, you might want to jump only when the user presses a button, or when a sensor reads a certain value. You'll find out how to do that in Unit 5.

Local Labels

One challenge when you are programming is coming up with new names for every label. The SX-Key assembler lets you create *local labels* that begin with a colon. These labels are only valid in between normal (or *global*) labels. Because the local labels are only valid within global labels, you can define the same label more than once without confusion. Consider this:

```
top          mov    w,#0   ; top is a global label
:loop       .              ; the first loop
            .
```

```
              .
                jmp    :loop  ; goes to first loop
ok              mov    w,#9   ; ok is a global label
:loop         .               ; the second loop
              .
              .
                jmp    :loop  ; jumps to second loop
```

You never have to use local labels. However, using them can make your life easier and your code more readable. The alternative is to generate unique labels for every address of interest in your program.

Another Way to INC

Sometimes you'd like to increment the value in a register, but you don't want to return the value to that register. In this case you can use a special form of the **mov** instruction:

```
                mov    w,++8
```

This leaves the result in the **W** register and does not change register 8. This allows you to use the register in other calculations without disturbing it.

In general, math operations always have these two forms. For example, the opposite of incrementing is decrementing (**dec**). This instruction subtracts one from a register. You can write it as:

```
                dec    8
```

or:

```
                mov    w,--8
```

The first form subtracts one from register 8 updating the value. The second form does the subtraction but leaves the result in **W** without changing the original value.

BASIC has no exact analog to inc and dec (other than x=x+1 or x=x-1). However, if you are a C or Java programmer, you can think of inc and dec as the ++ and – operators, respectively.

Stopping the Processor

In the early examples, the program used the **sleep** instruction to halt the processor. This might not seem very practical, but there are a few places where it can come in handy. For example, imagine a microcontroller that dials an emergency phone number. The signal to begin could be applying power to the circuit. The program would dial the number and then go to sleep, waiting for another power cycle to run again.

However, the main reason you'll use the **sleep** instruction is to put the processor in low-power mode until some external event occurs or some time period elapses. External events usually take the form of interrupts, a topic that will wait until Unit 7. However, you can wake up at a predetermined time by using the watchdog timer.
The main purpose of the watchdog timer is to reset the processor in the case of a malfunction. However, you can also use it as a timer to set a wake up time.

About the Watchdog

To enable the watchdog, add the **watchdog** setting to the **device** statements near the beginning of the program. Notice that turning on the watchdog will prevent the debugger from operating correctly, however. The idea behind the watchdog is that your program should use the **clr** instruction to zero the **!WDT** register periodically. This indicates that the program is working. If you fail to clear this register after a certain period of time, the processor resets.

The usual purpose of the watchdog timer is to reset the processor in case of a failure. It is usually best to have a single point in your program that clears the watchdog timer (!WDT). That way the chances of your program crashing and still clearing the timer are remote. If your program stops behaving correctly, the watchdog timer will restart it.

How long is that period? The SX has an internal oscillator for the watchdog that nominally runs at 14 kHz and the watchdog times out after 256 counts. So the timeout period is about 18 ms. So if you issue a **CLR !WDT** instruction at least once every 18 ms, you won't get a watchdog reset.

For timing purposes, this might not be long enough, however. The SX allows you to further scale the watchdog timer by setting bits in the **!OPTION** register. In particular, bit 3 of this register is set to 1 if you want to use the prescaler with the watchdog timer. Bits 2, 1, and 0 set the divide rate (see Table 3-1). The highest divide rate is 1:128 so the maximum time out is about 2.3 seconds.

Table 3-1: Watchdog Timer Prescale Values

Bit 2	Bit 1	Bit 0	Divide Rate
0	0	0	1:1
0	0	1	1:2
0	1	0	1:4
0	1	1	1:8
1	0	0	1:16
1	0	1	1:32
1	1	0	1:64
1	1	1	1:128

How can you set a single bit in a register? You can use **SETB** to set a bit to 1 and **CLRB** to clear a bit to 0. So to turn on the watchdog prescaler and set the divide rate to 1:32 you could write:

```
                setb    !option.3
                setb    !option.2
                clrb    !option.1
                setb    !option.0
```

The advantage to doing this is that you don't disturb the rest of the register. However, it is also possible to observe that the defaults for the remaining bits of the **!option** register should be 1's. So if you knew you wanted 1's in the other positions, you could write:

```
                mov     !option,#$FD
```

The **!option** register defaults to all 1's anyway, so if you want the maximum time out value, you don't need to do anything but enable the watchdog timer. Consider this program:

```
;=========================================================================
;Beginning Assembly Language for the SX Microcontroller
;Program 3.2
;=========================================================================
                device  sx28l,oschs3
                device  turbo,stackx,optionx, watchdog
                IRC_CAL IRC_SLOW
                reset   start_point
                freq    500000          ; 500 kHz


                org     0
start_point     mov     !rb,#0          ; make all of port b outputs
again           mov     w,#$FF
                xor     8,w             ; invert bits
                mov     rb,8
                sleep
```

This program will cause the LEDs to flicker so that you can actually see them. The only problem is that you don't know which LEDs will be on and which will be off initially. The program uses the **xor** instruction to exclusive-or the contents of register 8 with the constant $FF. You'll read more about **xor** in the next unit, but for now just realize that these two instructions will flip all the bits in register 8. That is to say that all 0s in register 8 will become 1s and all 1s will become 0s. You can, by the way, replace the **mov** and **xor** instructions with the **not** instruction which also flips the register bits and takes less time to execute. For now, however, leave the code as it is because the next unit will use the $FF constant to demonstrate some important ideas.

The last thing the program does is to store register 8's contents into port B. Since the code just flipped all the bits, all the LEDs that were on will turn off and all the ones that were off will turn on. Then the SX goes to sleep. However, since the watchdog is on (notice the **watchdog** clause in the second **device** line) the processor will reset in about 2.3 seconds. This will then flip the bits in register 8 again, reversing the state of the LEDs. Don't forget that you can't debug this program because it uses the watchdog. You'll have to use the Run | Run command to see the program work.

Earlier, you read that programs that use the watchdog must use **clr !wdt** to reset the timer. This program, however, doesn't clear the watchdog. Why? Because this program deliberately wants the watchdog timer to reset – that is how the program delays long enough for the LEDs to blink.

Of course, it would be nice to know that the reset was from the watchdog timer. You can do this by examining the bits in the **status** register. In particular, bit 4 will be 0 if the watchdog triggered a reset. If bit 3 is a 0, then a **sleep** instruction was active at the time. If you knew how to test these bits (a topic coming up shortly) you could initialize register 8 to a known value when a real reset occurred and not initialize it when a watchdog reset occurred.

Using the watchdog for timing is a bit unusual, but perfectly legitimate. In later units you'll find two other ways to make time delays: programmed loops and using the real time clock. These will be easier, because you'll be able to use the debugger when you employ these methods. Another advantage: when the processor resets, there is a brief time that all pins return to the input state until your program sets the direction register. The other methods for generating a time delay allow your program to stay in control of the processor at all times.

Summary

This unit covers a lot of instructions including:

- **jmp** – Jumps to a new program location
- **sleep** – Stops the processor
- **inc** – Adds 1 to a register (also use **mov w, ++r** to put result in **w**)
- **dec** – Subtracts 1 from a register (or use **mov w, --r**)
- **nop** – Does nothing for 1 clock cycle
- **setb** – Sets a bit in a register
- **clrb** – Clears a bit in a register
- **clr** – Sets a register, **w**, or the watchdog timer to zero
- **not** – Inverts bits in a register
- **xor** – Exclusive-ors the bits in a register (more in the next unit)
- **add** – Adds **w** to a register (more in the next unit)

You also read about the **PC** register, and parts of the **!option** and **status** register. In the next unit, you'll find out even more about arithmetic and variables, paving the way for more powerful programs.

Exercises

1. If you have access to an oscilloscope, add some **nop** instructions to the programs that blink the LEDs and examine the results.

2. Modify the watchdog program so that the LEDs blink at one half of the original rate (about 1.15 seconds).

3. What if you wanted to stop the watchdog LED program without using **sleep** and without triggering a watchdog reset? Modify the code so that it halts and does not reset. This will result in a steady pattern of LEDs lighting.

Answers

1. Here is a possible solution:

```
start_point  mov    !rb,#0       ; make all of port b outputs
again        clr    rb
             inc    rb           ; change port b outputs
             nop                 ; add more nops if you want
             jmp    again
```

2. To modify the rate of blinking, you'll change the watchdog timer prescaler value. One way to do this is to place **mov !option, #$FC** near the beginning of the program. You can also use **setb** and **clrb** to set and clear the individual bits in the **!option** register.

3. Replace the **sleep** instruction with:

```
halting      clr    !wdt
             jmp    halting
```

Unit 4: Variables and Math

The SX uses its registers as data storage. In the examples from previous units, we have simply referred to registers by their numbers. Remember, the first seven or eight registers (depending on the exact processor type) have special names (like **rb**, **status**, or **!option**) and functions.

The special names of these registers help you remember what they do. How can you use meaningful names for registers that your program uses for its own purposes?

Suppose you want to use register 8 as a variable in your program. There are several ways you can do so. First, you can set up an equate in one of two ways. Near the top of the program you could write:

```
Myvar         EQU    8
```

Or:

```
Myvar         =      8
```

Now you can replace all the occurrences of 8 with **Myvar**. You can use this method to define any constant even if it is not a register number. The assembler simply replaces every occurrence of **Myvar** with 8.

The other way to define a variable is by reserving space for it using the **DS** directive. The **DS** directive usually has a label in front of it, and has the number of bytes to reserve following it. So to replace the above equates with a **DS** directive you could write:

```
              org    8
Myvar         ds     1
```

The confusing part about this is that the **org** directive can refer to the data space or the program space, depending on the context. In this case, the 8 refers to the data memory. Before you start writing program steps, you'll want to write another **org** directive to set the beginning of your program (often location 0).

It is perfectly normal to specify several variables one after another. For example, consider this code that declares a byte variable named **Abyte** and two bytes named **Tbytes**:

```
              org    8
Abyte         ds     1
Tbytes        ds     2
```

When you use a variable name in your program, the name of a multi-byte variable refers to the first byte of the variable. So consider this statement:

```
              mov    w, Tbytes
```

This loads the first byte of the variable into **w**. On the other hand, look at this line:

```
              mov    w, Tbytes+1
```

This line of code will access the second byte. Is this any different than the following program snippet?

```
              org    8
Abyte         ds     1
Tbytes        ds     1
Tbyte1        ds     1
```

No. There is no difference except that using this form, you can use **Tbyte1** instead of **Tbytes+1**. Of course, you can still use **Tbytes+1**; the assembler does not care since either expression will result in a final value of 10 (decimal).

An Example

Remember the blinker programs in the last unit? Here it is again:

```
;=======================================================================
;Beginning Assembly Language for the SX Microcontroller
;Program 4.1
;=======================================================================
               device  sx28l,oschs3
               device  turbo,stackx,optionx, watchdog
               IRC_CAL IRC_SLOW
               reset   start_point
               freq    500000         ; 500 kHz


               org     0
start_point    mov     !rb,#0         ; make all of port b outputs
agn            mov     w,#$FF
               xor     8,w            ; invert bits
               mov     rb,8
               sleep
```

Here is the same program using symbolic variable names:

```
;=========================================================================
;Beginning Assembly Language for the SX Microcontroller
;Program 4.2
;=========================================================================
               device  sx28l,oschs3
               device  turbo,stackx,optionx, watchdog
               IRC_CAL IRC_SLOW
               reset   start_point
               freq    500000         ; 500 kHz


               org     8              ; data start
outval         ds      1

ledport        =       rb
flipmask       equ     $FF

               org     0              ; code start

start_point    mov     !ledport,#0    ; make all of port b outputs
                                      ; changed to use single
                                      ; instruction to xor with
                                      ; constant
agn             xor     outval,#flipmask
                mov     ledport,outval
                sleep
```

This program uses both = and **EQU**. This is often a matter of personal choice. However, once you define a symbol with **EQU** you can't change it later during assembly. Defining a symbol with = allows you to change it later. In this program, like most simple programs, the symbol values don't change at all, so you can use either method.

When you define a symbol for a constant (like **flipmask**) it still requires the # character to precede it. Without it, the assembler will think you are defining a register number.

Another way to use an equate is to define a name for a particular bit. You can specify bits in SX assembly language using a period after the name of the register and then the bit position. For example, the least-significant bit in register **rb** is **rb.0**. The most significant bit is **rb.7**. Using an equate you can define a meaningful name to a bit:

```
LEDpin         equu    rb.0
```

Using names for the registers and constants make the program much more readable. It also allows you to easily change things if you want. For example, it would be simple to change this program to blink LEDs on port A instead of B. It would also be no trouble to change the register from register 8 to another register, if you wanted to do so.

Assignment

In BASIC or C, you can assign one variable to another. The SX can do this too using the **mov** instruction. For example:

```
             org    8
byte1        ds     1
byte2        ds     1

             org    0
             mov    byte1,#$AA
             mov    byte2,byte1
```

This piece of code will put $AA in **byte1** and then put the contents of **byte1** into **byte2**.

The SX machine language does not really have an instruction that moves one register to another. That means the assembler generates a two-part instruction for the second mov instruction in this program. The two instructions are actually:

```
        mov     w,byte1
        mov     byte2,w
```

So this one line of code does destroy the w register. This can also lead to inefficiencies. For example, consider this:

```
        mov     byte2, byte1
        mov     byte3, byte1
```

This code unnecessarily loads the w register twice. A better way to do this would be:

```
        mov     byte2, byte1
        mov     byte3, w
```

Or:

```
        mov     w, byte1
        mov     byte2, w
        mov     byte3, w
```

Both of these take 3 instructions (instead of 4) and execute more quickly than the first example.

The only problem is with multi-byte variables. The SX only deals with bytes. That means that if you want to work with larger quantities, you'll have to break up the operations byte by byte. For example, you'd need two **mov** instructions to copy a two-byte variable to another two-byte variable.

For now, stick to bytes. However, bytes can only store numbers ranging from 0 to 255 (or –128 to 127). So if you need numbers larger than this, you'll have no choice but to resort to larger variables.

Performing Math

In the last unit, you saw that the **add** instruction can add the **w** register and another register. You can leave the result in **w** or in any register you like. You can also add a literal to a register, or add two registers together. However, these are two instruction sequences that destroy the **w** register in the process. Here are some examples:

```
              org    8
avar          ds     1
bvar          ds     1

              org    0
       .
       .
       .
              add    w,avar        ; w=w+avar
              add    avar,w        ; avar=w+avar
              add    avar,#10      ; avar=avar+10 (w destroyed)
              add    avar,bvar     ; avar=avar+bvar (w destroyed)
              add    bvar,avar     ; bvar=avar+bvar (w destroyed)
```

The byte-size of these operations can lead to a problem. What happens if the answer is larger than 8 bits? For example, if **w** contains $FF and you add **w** to a register that contains $10, what happens? The answer is that the SX truncates the result. However, to let you know that this has happened, it sets the carry flag (bit 0) in the **status** register. This is true regardless of the destination of the answer. Another bit in the **status** register (bit 2) is set whenever the answer is zero. You can use **status.0** and **status.2** to refer to the carry and zero flags, or to make your programs more readable you can use the symbolic names, **status.C** and **status.Z**.

Later in this unit, you'll learn how to examine these flag bits and use them to perform multi-byte math. You should be aware that not all operations affect these flag bits in the same way. For example, the **inc** and **dec** instructions (covered in the last unit) add or subtract 1 from a register. However, they do not set the carry flag. They do set the zero flag. The SX data sheet tells you which flags each instruction affects.

The opposite of adding, of course, is subtracting. The **sub** instruction can subtract **w** from any register. The result remains in the register. If you want to put the result in **w**, you can use this form of the **mov** instruction (where **R** is the register you want to use):

```
mov    w,R-w         ; w=R-w
```

You can also subtract two registers or a literal from a register. However, both of these are really two machine language instructions and destroy **W**. So:

```
sub    avar,W
sub    avar,#100     ; avar=avar-100 (w destroyed)
sub    avar,bvar     ; avar=avar-bvar (w destroyed)
```

The carry flag (bit 0 of **status**) has reversed meaning for **sub**. Suppose you subtract 100 from 30. The carry flag will be clear to indicate that the subtraction *underflowed*. However, if you subtract 30 from 100, carry will be set indicating that the subtraction yielded the correct result. Subtracting also affects the zero flag.

If you can add and subtract, you might wonder about multiplying and dividing. Simple microcontrollers like the SX can only add and subtract. However, using some techniques you'll see in the next unit, you can decompose multiplication and subtraction into multiple additions and subtractions.

Two's Compliment Numbers

If the carry flag is clear after subtraction, does that mean that the answer is incorrect? Not necessarily. Any microcontroller, including the SX, can handle negative numbers by using *two's compliment arithmetic*. The idea is simple. Treat the topmost bit (bit 7, in this case) as a sign bit. If the bit is 0, then the number is positive. If the bit is 1, then the number is negative. To represent a negative number, invert the magnitude of the number and add 1. Obviously, to find out the value of a negative number, you'd subtract 1, and invert it again.

Consider what happens if you subtract 60 from 40. The correct answer, of course, is negative 20. The SX, however, returns %11101100 ($EC). If you invert this number (%00010011) and add 1 (%00010100) you'll find the result is in fact 20. You can also make up new negative numbers. Suppose you want to add –5 to 10. First, find the binary representation of 5 (%00000101) and invert it (%11111010). Next add 1 to get %11111011 ($FB or 251). If you add 10 to 251, you get 261. But the SX does not get 261! It truncates the result to 5 (the bottom 8 bits of $105). Of course, 10 + -5 is 5, so the answer is correct.

These operations, by the way, are easy to perform on the SX. The **not** instruction will invert bits and **inc** or **dec** will add or subtract 1. So handling these negative numbers is not very difficult, even at run time.

The downside to two's compliment math? It limits the numbers you can represent. For a byte, the numbers between 0 and $7F represent 0 to 127 and the numbers from $80 to $FF represent –128 to –1. So although you usually think of a byte's limit as 255, when using signed math, the maximum number is really 127.

More Carry Tricks

Suppose you need larger numbers, say 0 to 999. You'll need to use more than 1 byte. A two-byte number can hold from 0 to 65535, plenty of room for this job. The problem is, how do you do math with these larger numbers.

The **addb** and **subb** instructions will add or subtract a bit – which could be the carry bit – from a register. Consider this simple program:

```
;========================================================================
;Beginning Assembly Language for the SX Microcontroller
;Program 4.3
;========================================================================
               device  sx28l,oschs3
               device  turbo,stackx,optionx
               IRC_CAL IRC_SLOW
               reset   start_point
               freq    500000         ; 500 kHz

               org     8              ; data start
counter        ds      2

               org     0              ; code start

start_point    clr     counter
               clr     counter+1      ; clear both bytes
again
                                      ; do a 16-bit add
               add     counter,#1
               addb    counter+1,status.0
               jmp     again
```

Here, the code is adding 1 to the 16-bit variable **counter**. It also adds the carry bit to the top 8 bits of the counter. Since the carry bit will only be set when the counter overflows, the count will be correct. You can do the same thing with subtraction by using **subb** instead of **addb**.

By using more registers and more **addb** or **subb** instructions, you can manipulate numbers of arbitrary size. A 24-bit number (3 bytes) can hold up to around 16 million. A 32-bit number (the same size the Pentium PC uses; 4 bytes) can hold numbers of around 4 billion in value.

Try It!

Enter the code above and step through it. You'll quickly get tired of watching register 8 cycle endlessly upwards. The Jog command helps, but it still takes a while to get to the interesting part of the code. This is a good time to learn a few extra features of the debugger. First you can click on the box for register 8 and change the value of the register. So if you plug in $FE (or %11111110) in the register 8 box, you'll be much closer to seeing the roll over! This works for all of the registers visible in the debugger.

Another annoyance is that you have to know that the counter variable is actually in registers 8 and 9. An easier way to observe the contents of memory is to use a **watch** directive. This is a statement in your program that tells the debugger to display a piece of memory with a name, and to format it so that it is meaningful. You specify the memory location, the size of the variable, and the format you want. For this program, try adding this line anywhere in your file:

```
            watch  counter,16,UDEC
```

This will show the 16-bit variable at location **counter** as an unsigned decimal number. You can find a list of all the format codes in Table 4-1.

Table 4-1: Watch Format Codes

Format Code	Appearance
UDEC	Unsigned decimal
SDEC	Signed decimal
UHEX	Unsigned hex
SHEX	Signed hex
UBIN	Unsigned binary
SBIN	Signed binary
PSTR	Fixed-length string of ASCII characters
ZSTR	String of ASCII characters terminated with a zero

ASCII (American Standard Code for Information Interchange) is a way to represent text characters as a 7 or 8 bit number. For example, in ASCII, an A is $41, a blank is $20, etc.).

A Few More Functions

You'll often use the carry bit for a variety of functions. Earlier in this tutorial, you read that you can use **setb** and **clrb** to set and reset a bit. Since the carry bit is just a bit in the **status** register, you can use these instructions to affect the carry.

However, this is a frequently used function, so the assembler provides other instructions to do it so you can type less. In particular, **clc** clears the carry (**clz** clears the zero flag) and **stc** sets the carry (**stz** sets the zero flag).

The real trick is to control your program's flow based on these flags. There are several ways to do this. First of all, the generic **jb** (jump on bit) instruction will execute a jump if the specified bit is set. So to jump to **lbl1** if the carry flag is set, you could write:

```
jb     status.0,lbl1       ; or jb status.C,lbl1
```

Of course, using **jb** you can specify any bit. However, the carry and zero bits are very common bits to test, so the assembler also allows you to use the **jc** and **jz** instructions to test for the carry or zero conditions. You can also use **jnb** (jump no bit) to jump when the bit is clear instead of set. For zero and carry, you can use **jnz** and **jnc**, respectively.

By performing a subtraction and then testing the carry and zero flags, you can easily write programs that can tell if one number is greater than, less than, or equal to another number. For example, suppose you wanted to know if variable **x** was greater than variable **y**:

```
mov    w,x
mov    w,y-w
jnc    x_greaterthan_y
```

This works because subtracting **x** from **y** will only be negative (that is, cause an underflow) if **x** is greater than **y**. Remember that carry is clear on an underflow when subtracting.

You might consider computing x-y and changing the jnc to jc. That would also work, but it would jump if x were greater than or equal to y. To see why, work out the case where x is equal to y. Of course, you can use jz to test for equality and jnz to test for inequalities. See Table 4-2 for a summary of possible results when subtracting two numbers.

Table 4-2: Results When Computing a-b

Carry	Zero	Meaning
0	0	a<b
X (don't care)	1	a=b
1	0	a>=b

Testing for equality with zero is a very common operation, so the assembler lets you write it in a special way. You can use **test**. The **test** instruction sets the zero flag based on any register (including the **w** register).

Another common function relating to zero testing is to increment or decrement a register and jump if the result is zero. You can use **djnz** (to decrement) or **ijnz** (to increment) for this purpose.

Here is another LED flasher that uses **djnz** to blink the LEDs a total of ten times:

```
;=======================================================================
;Beginning Assembly Language for the SX Microcontroller
;Program 4.4
;=======================================================================
               device  sx28l,oschs3
               device  turbo,stackx,optionx
               IRC_CAL IRC_SLOW
               reset   start_point
               freq    500000         ; 500 kHz

               org     8              ; data start
counter        ds      1
pattern        ds      1
               watch   counter,8,udec
               watch   pattern,8,ubin

               org     0              ; code start

start_point    mov     !rb,0          ; all outputs
               clr     rb             ; all low
               mov     counter,#10    ; 10 times
again
               mov     rb,pattern
               not     pattern
               djnz    counter,again
               sleep
```

Notice that the blinking code executes 10 times because the **counter** variable starts with 10, and reduces by 1 until it reaches zero. This is a powerful idea and often used in computer programs. Code like this is known as a *loop* because it executes in a loop as often as you need.

In BASIC or C, you'd do something like this with a **for** statement. In BASIC, for example, I might write a loop as:

```
FOR counter = 10 to 1 step -1
       ' Do the work
NEXT
```

Of course, you'd usually see this reversed, with **counter** ranging from 1 to 10. You could do this too, but it takes a few more assembly language instructions:

```
             inc    counter       ; assume counter was
                                  ; set to 0 at beginning
             mov    w,#10
             sub    w,counter-w
             jnz    again
```

In the next unit, you'll see a series of compare instructions that can perform this type of logic in one assembly language instruction (but they just write the same sort of code you see above).

Programmed Delays

Another important use of loops is in developing programmed delays. In the previous unit, you saw how to use the watchdog timer as a crude timing device. However, this is not the ideal way to generate a time delay. The watchdog timer makes it hard to debug your program since the SX-Key can't debug your code with the watchdog set. Also, the watchdog can't generate arbitrary delays, and you lose control of the program while waiting for the delay.

However, if you know your clock speed, and the number of cycles each instruction takes, you can compute loops that will cause the appropriate delay. For example, suppose you wanted to generate a 1 kHz tone. A 1 kHz tone cycles every 1 ms (1/1000 = .001) so to make a 1 kHz square wave, the SX needs to turn a pin on, wait for 500 µs (half of 1 ms), turn the pin off, wait another 500 µs, and then start over.

Assume you have a piezoelectric speaker connected to pin 7 of port B (a piezo speaker has a high-impedance and you can drive it directly from the SX's output pins). If you could toggle pin 7 at this rate, you'd hear a 1 kHz tone coming from the speaker.

The problem is that 500 µs is an eternity for the SX. At 50 MHz, each instruction cycle (in turbo mode) takes 20 ns. So to pause 500 µs you'll need 25000 instructions cycles! Consider this simple loop:

```
             clr    delay
wloop        djnz   delay,wloop
```

Studying the SX data sheet, you can find that the **djnz** instruction takes 4 cycles every time it has to jump, and 2 cycles if it doesn't have to jump. The **clr** instruction takes 1 cycle. So the total number of cycles in this loop is 256 * 4 + 3 or 1027, a far cry from the 25000 you need. Of course, you could use a 16-bit delay, but this is hard to calculate since the total time through the loop varies depending on the carry flag's status. Instead, it is usually simpler to place this loop inside another loop. Dividing 25000 by 1027 you'll find you need about 24 repeats of this loop to get to 25000. So:

```
              mov    delay1,#24
oloop         clr    delay
wloop         djnz   delay,wloop
              djnz   delay1,oloop
```

Of course 24 * 1027 = 24648, not exactly the right answer. However, the outer loop adds 95 cycles to the total loop (see if you can calculate that number). That brings the total delay to 24743 (a 1.02% error). For many purposes, this is not a problem. If you needed a more exact figure, you could reduce the number of cycles in the inner loop and increase the count in the outer loop until you get as close as necessary. You can also adjust the timing of the loops by adding **nop** instructions inside the loop to stretch it out.

Logical Functions

Since microcontrollers and other computers work with binary, it isn't surprising that they contain many operations designed to operate on the bits of word. Like other operations, these work on the **w** register and an arbitrary register with the result going to the register of your choice. You can also use a register and a constant, or two registers, but if you do, you will generate more than one machine language instruction and destroy the **w** register in the process. The main logical functions include **and**, **or**, and **xor**.

What do these functions do? They simply examine the two values you supply bit by bit and generate an output bit base on the corresponding input bits. Take **and**, for example. If you use **and** on %10101010 and %11110000, the result is %10100000. Why? Because **and** only outputs a 1 if both input bits are 1. The **or** instruction outputs a 1 if either input bit is 1. The **xor** instruction outputs a 1 if either input is a 1, but not if both inputs are a 1. You can find a summary of these operations in Table 4-3.

Table 4-3: Logical Instructions

Instruction	Truth Table			Move to W Form
	Input	Input	Output	
And	0	0	0	and w,R
	0	1	0	
	1	0	0	
	1	1	1	
Or	0	0	0	or w,R
	0	1	1	
	1	0	1	
	1	1	1	
Xor (exclusive or)	0	0	0	xor w,R
	0	1	1	
	1	0	1	
	1	1	0	
Not	0		1	mov w,/R
	1		0	
RL (rotate left)	n/a			mov w,<<R

You've already seen that you can use **not** to invert the bits in a register (including the **w** register). You can also rotate or shift bits left or right by using **rl** (left) and **rr** (right). Unlike the other logical instructions, these commands operate on a single register (or the **w** register in the case of **not**). When you shift a register left, each bit is replaced by the bit prior to it. So bit 7 gets the value of bit 6, bit 6 gets the value of bit 5, and so on. Bit 0 gets the value of the carry flag and the carry flag's value gets set to the original value of bit 7. Shifting right is the reverse process, where bit 7 gets the carry flag value, and bit 0 shifts into the carry flag.

When you shift left, you multiply the number by 2. Shifting right is the same as dividing by 2.

By combining shifts and addition you can perform many multiplications in an efficient way. For example, suppose you want to multiply a number by 10 (not an uncommon thing to do). One way would be to add the number to itself 10 times in a loop. While that would work, a more efficient way would be to realize that multiplying by 10 is the same as multiplying by 8 and then multiplying by 2. Since 8 and 2 are both powers of 2, you can do those multiplications using shifts.

Here is an example of both styles of multiplication:

```
;========================================================================
;Beginning Assembly Language for the SX Microcontroller
;Program 4.5
;========================================================================
               device  sx28l,oschs3
               device  turbo,stackx,optionx
               IRC_CAL IRC_SLOW
               reset   start_point
               freq    50000000       ; 50 MHz


               org     8              ; data start
value          ds      1
result         ds      1
result2        ds      1
counter        ds      1
               watch   value,8,udec
               watch   result,8,udec
               watch   result2,8,udec
val            =       21

               org     0              ; code start
start_point                           ; multiply by 10 2 different ways
               mov     value,#val
; first a loop
               mov     counter,#10
               clr     result
               mov     w,value
mloop          add     result,w
               djnz    counter,mloop
; ok answer is in result
               nop
               mov     value,#val
; now do shift add
               clc
               rl      value          ; value = value *2
               mov     result2,value
               clc
               rl      value
               clc
               rl      value          ; value = value *8
               add     result2,value
; same answer in result2

               sleep
```

Don't forget to clear the carry before rotating if you are using rotation for a multiply or divide. The carry bit shifts into the word which can throw off your results if you don't clear it first.

When debugging this program, don't forget that you can't directly set a breakpoint on a sleep instruction.

Of course, if you can't decompose your multiplication into something you can do with rotates, you'll have to look at the techniques covered in the next unit. Unfortunately, there is no easy way to combine divisions. You can divide by 2, 4, 8, or any power of two (by shifting right instead of left), but there isn't an easy way to divide by 10 or other arbitrary numbers.

Summary

Wow! This unit covers a lot of ground. You learned about **ADD**, **SUB**, **ADDB**, **SUBB**, lots of bit operations, and even some conditional jumps. Using these instructions you can do lots of different things including simple math, controlling the number of times a piece of code executes, and comparing numbers. These are the building blocks that allow your microcontroller to make decisions.

Remember in Unit 1 you read that a computer reads inputs, does processing, and generates outputs. The instructions in this chapter are the core that you will use to do the processing portions of your program.

Exercises

1. Change the counter program to use **inc** instead of **add**. Do you still need **addb**? If you do, which bit should you add?

2. Change the counter to use a 32-bit count instead of two bytes. Test your changes using the debugger.

3. Write the program that generates a 1 kHz tone on a speaker connected on pin 7 of port B. Note: don't hook a regular speaker directly to the SX output pins. Instead, use a piezoelectric speaker designed for direct IC drive. If possible, measure the output with an oscilloscope or frequency counter.

Answers

1. You can use **inc**, but remember that **inc** does not set the carry flag. However, it does set the zero flag. If you increment a number and get a zero, then it stands to reason that an overflow occurred. The correct code would look like this:

```
inc     counter
addb    counter+1,status.2   ; status.2 is zero flag
```

2. This is just a matter of changing the **ds** statement to reserve 4 bytes instead of 2 and adding two more **addb** instructions immediately following the one that is there:

```
add     counter,#1
addb    counter+1,status.0
addb    counter+2,status.0
addb    counter+3,status.0
```

Here is one possible solution:

```
;=========================================================================
;Beginning Assembly Language for the SX Microcontroller
;Program 4.6
;=========================================================================
               device  sx28l,oschs3
               device  turbo,stackx,optionx
               IRC_CAL IRC_SLOW
               reset   start_point
               freq    50000000        ; 50 MHz
               org     8               ; data start
delay          ds      1
delay1         ds      1
               org     0               ; code start
start_point
               mov     !rb,#$7F        ; speaker output only
loop           not     rb              ; toggle bits
               mov     delay1,#24
oloop          clr     delay
wloop          djnz    delay,wloop
               djnz    delay1,oloop
               jmp     loop
```

Unit 5: Advanced Flow Control

In the last unit, you learned how to control the flow of execution based on conditions. Instructions like **jz**, **jc**, and **djnz** allow you to jump when some condition is met. There are other ways that you can control the flow of your program, however, and you'll read about these in this unit. In addition, you'll read about ways to perform integer multiplication and division using several techniques.

Skipping

All of the jump instructions you read about in the last unit are not really machine language instructions. Instead, they are multi-instruction constructs that the assembler provides for your convenience. The SX actually only performs conditional tests as skips. The idea is to execute an instruction that, depending on the condition, will either execute the next instruction or skip it. It the next instruction is a **jmp** then you have an equivalent of the jump instructions you found in the last unit.

There are two things to consider here. First, the skipped instruction need not be a **jmp**. This can lead to faster, more efficient code in some cases. The second issue, however, is that skips only skip one machine language instruction. Many of the instructions you use are really composite instructions and they consist of more than one machine language instruction (see Table 5-1).

For example, some **mov** instructions require two words. Consider this bit of code:

```
skip
mov    8,#100
```

The **skip** instruction is supposed to cause the SX to skip the next instruction no matter what. However, it causes the program to skip the next machine language instruction. There is no machine language instruction that corresponds to a **mov** of a constant (or *literal*) to a register (other than **w**). So the assembler really generates:

```
skip
mov    w,#100
mov    8,w
```

The net result is that the program moves **w** – whatever happens to be in it – to register 8 without loading 100 into it first. Not what you expected. For this reason, you must be very careful when using skips.

You won't have much call to use the unconditional skip instruction. What you usually want is an instruction that skips on some condition. There are six skip instructions of this sort. The **sb** and **snb** instructions skip if a specified bit is set or clear. The assembler also provides special shorthand instructions for testing the carry (**sc** and **snc**), and the zero flag (**sz** and **snz**).

Table 5-1: Multi-word Instructions

Instruction	Words
ADD (without W)	2
ADDB	2
AND (without W)	2
CJA	4
CJAE	4
CJB	4
CJBE	4
CJE	4
CJNE	4
CSA	3
CSAE	3
CSB	3
CSBE	3
CSE	3
CSNE	3
DJNZ	2
IJNZ	2
JB	2
JC	2
JNB	2
JNC	2
JNZ	2
JZ	2
LCALL	1-4
LJMP	1-4
LSET	0-3
MOV (some forms)	2
MOVB	4
OR (without W)	2
RETW (with multiple values)	varies
SUB (without W)	2
SUBB	2
XOR (without W)	2

Comparing

Of course, a very common thing to do is to test two values and based on the result jump to some location. You saw this in the last unit done with a subtraction and a jump instruction. The assembler allows you to use special multi-instruction compares as a shorthand notation for doing this operation. You can find a list of these in Table 5-2. These instructions require three pieces of information: a register, a register or a constant, and a jump address.

Table 5-2: Compare Instructions

Instruction	Use	BASIC Equivalent	Skip Form
CJA A,B,LBL	Jump if above	if A>B then LBL	CSA
CJAE A,B,LBL	Jump if above or equal	If A>=B then LBL	CSAE
CJB A,B,LBL	Jump if below	If A<B then LBL	CSB
CJBE A,B,LBL	Jump if below or equal	If A<=B then LBL	CSBE
CJE A,B,LBL	Jump if equal	If A=B then LBL	CSE
CJNE A,B,LBL	Jump if not equal	If A<>B then LBL	CSNE

These compare instructions are very similar to a BASIC or C **IF** command. The only difference is that the comparison can only be between two variables or a variable and a constant. You'll find the equivalent BASIC syntax in Table 5-2.

You can also do a compare and skip the next instruction if the comparison is true. Just like any skip instruction, however, you have to be careful not to try to skip a multi-word instruction (see Table 5-1). Table 5-2 shows the skip instructions that correspond to different conditional jumps.

Using Call and Return

You'll often find yourself doing the same things several times in one program. For example, if you want to add two 16-bit numbers, it is a good bet that you need to do it in more than one place.

The SX knows that you will want to write code that you can reuse and so it provides **CALL** and **RET** instructions. These instructions implement the same sort of functions that **GOSUB** provides in BASIC (or functions in C).

In the previous unit, there is a program that generates a 1 kHz tone from a speaker connected to pin 7 of port B. But suppose you needed a program that did the following:

1. Make a 1 second beep on the speaker
2. Wait for you to push a button connected to port B, pin 0
3. Beep for 1 second again
4. Return to step #2

You can find the circuit required for this example in Figure 5-1. The code in the last unit that made the 1 kHz tone looks like this:

```
loop          not    rb            ; toggle bits
              mov    delay1,#24
oloop         clr    delay
wloop         djnz   delay,wloop
              djnz   delay1,oloop
              jmp    loop
```

Since each loop requires about 500 µs, you will need to execute the loop 2000 times to generate a 1 second tone. That simply requires another loop. However, it seems a waste to have to duplicate this code in two different parts of the program. That is where the **call** instruction is useful. You can make a *subroutine* out of the beep code and then call it from different parts of your program.

To create a subroutine, you simply assign the code a label. Other parts of your program will use this label (along with **call**) to execute the subroutine. When the subroutine code executes a **ret** (return) instruction, execution resumes with the instruction after the **call**. Consider the tone code rewritten as a subroutine:

```
beep          mov    second,#$DO   ; 2000 is $7D0
              mov    second+1,#$07

loop          not    rb            ; toggle bits
              mov    delay1,#24
oloop         clr    delay
wloop         djnz   delay,wloop
              djnz   delay1,oloop
; repeat 2000 times
              djnz   second,loop
              djnz   second+1,loop
              ret                  ; go back to wherever
```

Now the main part of the code can simply use **call beep** anywhere it wants a one second beep to occur. It is perfectly acceptable to have more than one entry point into the subroutine. For example, if you wanted to set the **second** variable in your main program, you could call **loop** instead of **beep** (although you'd probably want to give it a better name). You could also get a half beep like this:

```
              mov    second,#$E8   ; 3E8
              mov    second+1,#$03
              jmp    loop
```

Subroutines can call other subroutines, but the SX can only handle 8 levels of *nesting* subroutines. That is, if subroutine A calls subroutine B, and subroutine B calls subroutine C, and so on, the SX will get confused when subroutine H calls subroutine I.

This in no way limits the number of subroutines you can have in a program. It simply limits the number of subroutines you can have active at one time.

To help you understand the idea of nested subroutines and the limit on nesting, think about an elevator that can hold 8 people. Each time you execute a **call** instruction, you are putting someone else on the elevator. Each time a **ret** instruction (or a **retw** instruction; see below) executes, someone gets off the elevator. If you execute 8 **call** instructions in a row without returning, the elevator becomes full and you can't add any more people until someone gets out of the elevator. However, over the course of the day many people might ride the elevator (some more than once, even). As long as no more than 8 at a time ride, everything works.

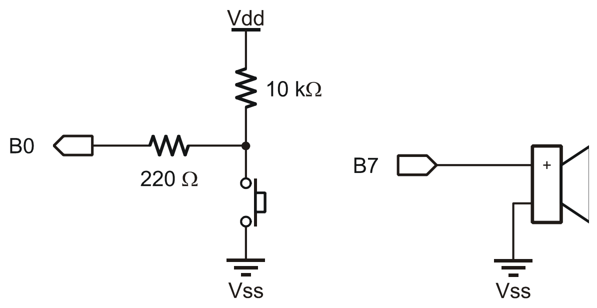


Figure 5-1: A Speaker and Switch Connected to the SX

Here is the tone program:

```
;=======================================================================
;Beginning Assembly Language for the SX Microcontroller
;Program 5.1
;=======================================================================
               device  sx28l,oschs3
               device  turbo,stackx,optionx
               IRC_CAL IRC_SLOW
               reset   start_point
               freq    50000000       ; 50 MHz

               org     8
```

```
second          ds      2               ; counter for 1 second tone
delay           ds      1
delay1          ds      1


                org     0
start_point     mov     !rb,#$7F        ; make speaker output
                call    beep
; wait for input button
bwait           jb      rb.0,bwait
                call    beep
                jmp     bwait

; subroutine

beep            mov     second,#$d0     ; 2000 is $7D0
                mov     second+1,#$07

loop            not     rb              ; toggle bits
                mov     delay1,#24
oloop           clr     delay
wloop           djnz    delay,wloop
                djnz    delay1,oloop
; repeat 2000 times
                djnz    second,loop
                djnz    second+1,loop
                ret                     ; go back to wherever
```

What if you wanted to use this subroutine in a program that already used labels like **oloop**, **loop**, and **wloop**? To prevent conflicts, try to use local labels (like **:oloop**, **:loop**, and **:wloop**) in your subroutines.

A few notes about this program are in order. For one thing, this is the first program in this tutorial that reads some input. The switch is connected in such a way that bit 0 of port B will read a 0 when you push the switch. The **jb** instruction tests for this – if the bit is a 1, it just loops to **bwait**.

Buttons are mechanical devices, and as such they exhibit *bounce*. That means that when you press the switch, the SX may see the switch open and close many times for a few microseconds until the switch firmly closes. The same thing happens when you release the switch – the button seems to turn on and off rapidly until it finally settles in the off position. In this program, this is no big deal because the tone forces a one second wait before the SX reads the switch again. However, if you were rapidly reading the button, you'd need to take this mechanical bounce into account.

If you run this program and hold the button down, the tone will continue until you release the button. That's because the program does not wait for you to release the button before continuing.

Often subroutines want to return some data (perhaps a status code) in the **w** register. To accommodate this common task, the SX provides the **retw** instruction. The **retw** instruction returns a constant in the **W** register. So:

```
             retw   #$FF
```

is the same as:

```
             mov    w,#$FF
             ret
```

Of course, **retw** is only a single instruction so it executes faster and requires less space.

Tables

One important use of **retw** is to generate tables. Suppose you wanted to find the square of a number between 0 and 10. You know that multiplication is difficult to do, so it makes sense to simply store the values in a table and read them out instead of doing the actual calculations. Here is a subroutine that does this:

```
; square a number from 0 to 10 in the W register
; return result in the W register
square       jmp    PC+W
             retw   #0
             retw   #1
             retw   #4
             retw   #9
             retw   #16
             retw   #25
             retw   #36
             retw   #49
             retw   #64
             retw   #81
             retw   #100
```

When the main program calls the **square** routine, it jumps to a different return instruction depending on the value in **W**. The **retw** instruction loads the correct value into **W** and returns to the caller. This is simple, efficient, and very fast. It is also so common, that the assembler lets you write multiple values on the same line. So you could replace the **square** routine with two lines of assembly:

```
square jmp    PC+W
              retw   0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100
```

The generated machine language code is exactly the same in either case, so there is no difference in using either method. It is a matter of personal preference.

Indirection

When you access the SX's registers, you need to know the address you want to use. Early in this tutorial, you used numeric addresses (like 8 or 9), but soon you saw the advantage to using symbolic names (like **status** or **counter**). However, sometimes you don't know the exact address you want to access when you are writing your program. For example, suppose you wanted to clear all the user memory in the SX. You could write:

```
              clr    8
              clr    9
              clr    10
              clr    11
              .
              .
              .
```

However, that seems wasteful. It would be nice if you could use a loop to *index* through the different registers. That is the purpose of the special **FSR** (File Select Register) and **IND** (Indirect) registers. The **IND** register is not an ordinary register. Instead it is an alias for another register somewhere in the SX. Where? Whichever register number is currently in **FSR**.

Here is a simple example:

```
R1            EQU    10            ; register 10 is R1
R2            EQU    11            ; register 11 is R2
              mov    R1,#100
              mov    R2,#200
              mov    FSR,#10       ; store address 10 in FSR
              mov    w,IND
; W now contains 100
              inc    FSR           ; go to next address
              mov    w,IND
; W now contains 200
              mov    FSR,#R1
              mov    w,IND
```

```
; W contains 100 again
             clr    IND           ; R1 is now 0!
```

Notice that you can write to **IND** as well as read from it. **IND** is a complete alias for whatever register number you store in **FSR**.

You'll usually want to load a constant number into FSR. In the previous example, for instance, if you used: mov FSR,R1 this would load the contents of R1 (100) into FSR – probably not what you meant. However, you can use the syntax mov FSR,#R1.

Here is a bit of code that will clear all the user registers (up to register $1F) in a loop:

```
             mov    FSR,#8
 :loop clr  ind
             inc    FSR
             jnb    FSR.5,:loop
```

This takes advantage of the fact that when **FSR** reaches $20 (that is, bit 5 is set for the first time) the looping is done. You could just as easily compare **FSR** with $20 or use some other scheme to break out of the loop.

This technique is not just for clearing memory. When programming, you'll often want an array of data (for example, the last 4 samples from a sensor, or the last 8 bytes read from a serial port). Using indirection is the way to efficiently code arrays, lists, and other data structures.

Math Functions

Armed with the ability to loop and test, you can tackle arbitrary multiplication and division problems with ease. A simple-minded approach to multiply, for example, 9 by 7 is to add 9 to itself 7 times. However, with a little knowledge of binary numbers, you can write a smarter algorithm.

Remember how you learned to multiply in grade school? You'd write your problem out and multiply the results digit by digit, moving to the left with each digit. Then you'd add all the partial results up to find the correct answer. The computer can do this too. As a bonus, the SX uses binary so each partial result can only be the original number shifted to the left some number of places or 0. Think about multiplying %1001 by %101 (9 by 5).

```
           1001
X           101
----------------
         001001
         000000
+        100100
----------------
         101101 = 32 + 8 + 4 + 1 = 45
```

Performing multiplication in this fashion is known as Booth's algorithm (an *algorithm* is just a fancy name for a set of program steps). Here is a bit of SX code that will multiply the byte in register **V1** by the byte in register **V2**:

```
             clr    V3            ; zero result
             mov    ctr,#8        ; 8 bits
mloop        rr     V2            ; load bit 0 of V2 into carry bit
             jnc    noadd         ; skip on no carry
             add    V3,V1         ; add to result
noadd        rl     V1            ; shift V1 over 1 place
             djnz   ctr,mloop     ; go 8 times
```

Of course, the result (**V3**) is a byte, so you can't multiply numbers that will require an answer larger than 255. You can easily extend this algorithm to handle more bits.

Division

You can use a similar algorithm to do division. If you remember your high-school math, dividing requires a *divisor*, a *dividend*, and produces a *quotient*. So when computing 20 divided by 5, 20 is the dividend and 5 is the divisor. The result, 4, is the quotient. Since 5 goes into 20 evenly, there is a *remainder* of 0.

When you perform division on paper, you reduce it to a series of subtractions. You also have to shift your position to keep track of what digit you are examining. The SX can do the same thing in binary. Since binary only has 1s and 0s, it is easy to tell if one number will "go into" another; simply see if the first number is smaller or equal to the second number.

1. Consider these program steps (or algorithm, if you prefer):
2. Set the quotient to 0
3. Shift the divisor to the left until the topmost bit is a 1
4. Remember how many shifts you performed in step 2 and add 1 to this count
5. Shift the quotient to the left (multiply by 2)

6. Compare the dividend and the divisor; if the dividend is greater than or equal to the divisor, subtract the divisor from the dividend and add 1 to the quotient
7. Shift the divisor to the right
8. Subtract 1 from the count and if not zero, return to step 4

Suppose you want to divide 20 by 5. After performing steps 1 to 3, you'll have a divisor of 160 and a count of 6. Table 5-3 is the looping part of the algorithm right after performing step 6:

Table 5-3: Looping Portion of Division Algorithm

Dividend	Divisor	Quotient	Counter	Comments
20	160	0	6	Shifted out 5 zeros; no subtraction
20	80	0	5	No subtraction
20	40	0	4	
0	20	1	3	Subtracted
0	10	2	2	
0	5	4	1	

What about a division with a remainder? If you replace 20 in Table 5-3 with, for example, 22 you'll find that the dividend column has a 2 in it after the subtraction. Since the divisor never goes below 2, the answer is the same. However, the dividend column winds up with the remainder (2).

Here is a simple division program written for the SX:

```
;=======================================================================
;Beginning Assembly Language for the SX Microcontroller
;Program 5.2
;=======================================================================
               device  sx28l,oschs3
               device  turbo,stackx,optionx
               IRC_CAL IRC_SLOW
               reset   start_point
               freq    50000000       ; 50 MHz

               org     8
dividend       ds      1
divisor        ds      1
quotient       ds      1
counter        ds      1
               watch   dividend,8,udec
               watch   divisor,8,udec
               watch   quotient,8,udec
               watch   counter,8,udec

               org     0
```

```
start_point
               mov     dividend,#20
               mov     divisor,#5
               call    divide
               break
               nop
               sleep

; subroutine

divide         clr     counter        ; assume not dividing by zero
               clc
:loop          rl      divisor
               inc     counter
               jnc     :loop
; restore divisor so top bit is 1
               rr      divisor
; counter has number of bits in quotient
               clr     quotient
:dloop
               test    counter
               jz      :done
               clc
               rl      quotient
               cjb     dividend,divisor,:dloop1
               sub     dividend,divisor
               inc     quotient
:dloop1
               dec     counter
               clc
               rr      divisor
               jmp     :dloop
:done
               ret                    ; go back to wherever
```

One thing this program does not do is test for divide by zero, which is an error. It would be simple to add a **test** instruction to set the zero flag if **divisor** was zero and jump to an error routine.

Summary

In this unit you've read about instructions that compare two values and make a decision based on the result. This type of flow control is crucial to implementing advanced multiplication and division algorithms (as well as for many other programming tasks). This unit also brought up subroutines (via the **call** and **ret** instructions) and ways to use subroutines to implement tables of constants. You can also create tables using the indirection registers (**fsr** and **ind**) that allow you to access registers without hard coding their addresses.

At this point in the tutorial, you have all the tools necessary to write some powerful programs. In the next three units you'll learn how to access all of the SX memory and how to further control the hardware. In addition, you'll work with interrupts and virtual peripherals.

Exercises

1. The example program in this unit beeps when the button is pressed for a short time. However, if the button remains depressed, the tone continues. Alter the program so that after the tone, the program waits until you release the button. Be sure to take steps to combat bounce.

2. Count the number of times the button is pressed. After 10 times, put the processor to sleep.

3. In earlier units, there is a blinker program that uses **sleep** and the watchdog timer to pause in between flashes. However, this precluded initializing the LEDs to a known state because the program could not tell the difference between the first reset and a reset after the **sleep** instruction timed out. Recall that the **status** register's bit 4 is 0 when a watchdog timeout occurs. Change the program to initialize port B to $AA in the event of a hard reset. The original program is below for your reference.

```
;=====================================================================
;Beginning Assembly Language for the SX Microcontroller
;Program 5.3
;=====================================================================
               device  sx28l,oschs3
               device  turbo,stackx,optionx
               IRC_CAL IRC_SLOW
               reset   start_point
               freq    500000         ; 500 kHz


               org     8
pattern        ds      1

               org     0
```

```
start_point    mov     !rb,#0          ; make all of port b outputs
               xor     pattern,#$FF
               mov     rb,pattern
               sleep
```

4. Connect buttons (as shown in Figure 5-1) to Port B pins 0, 1, 2, and 3. Connect a piezoelectric speaker to port B pin 7. Construct a program that plays a different tone for 500 ms each time you press a button. With more buttons, this could be the basis for a child's organ or a musical annunciator.

Answers

1. Here is the main code:

```
start_point   mov    !rb,#$7F      ; make speaker output
              call   beep
                                   ; wait for input button
bwait         jb     rb.0,bwait
                     call          beep
bwait1        jnb    rb.0,bwait1
                                   ; wait for bounce to complete
              clr    delay
:dwait        djnz   delay,:dwait
              jmp    bwait
```

The delay allows time for the button to quit bouncing – the time is arbitrary and might require adjustment depending on the kind of switch you use.

2. Here is an excerpt from the solution:

```
              org    8
second        ds     2             ; counter for 1 second tone
delay         ds     1
delay1        ds     1
presses       ds     1


              org    0
start_point   mov    !rb,#$7F      ; make speaker output
              call   beep
              clr    presses
                                   ; wait for input button
bwait         jb     rb.0,bwait
              call   beep
              inc    presses
              cje    presses,#10,halt
bwait1        jnb    rb.0,bwait1
                                   ; wait for bounce to complete
              clr    delay
:dwait        djnz   delay,:dwait
              jmp    bwait

halt          sleep
```

Of course, it would be just as legitimate to store 10 in the presses variable and decrement it. This would be somewhat more efficient because you could test the zero flag after decrementing the variable, thus saving a step.

3. The solution is to simply test for the bit 4 being clear:

```
;=========================================================================
;Beginning Assembly Language for the SX Microcontroller
;Program 5.4
;=========================================================================
               device  sx28l,oschs3
               device  turbo,stackx,optionx
               IRC_CAL IRC_SLOW
               reset   start_point
               freq    500000         ; 500 kHz

               org     8
pattern        ds      1

               org     0

start_point    mov     !rb,#0         ; make all of port b outputs
                                      ; check for real reset
               jnb     status.4,agn
               mov     pattern,#$AA
agn            xor     pattern,#$FF
               mov     rb,pattern
               sleep
```

You could make an argument for setting **pattern** to $55 instead of $AA since the very next instruction will invert the bits, but either way the result is acceptable.

4. There are several ways you could complete this exercise, depending on your personal preferences. The tricky part is realizing that since each tone takes a different amount of time, you have to adjust the number of cycles to get 500 ms. For example, a 1 kHz tone has 500 µs cycles, so you need 1000 cycles to get 500 ms. However, a 2 kHz tone has 250 µs cycles and therefore requires 2000 cycles to maintain the same duration. Here is one solution:

```
;=========================================================================
;Beginning Assembly Language for the SX Microcontroller
;Program 5.5
;=========================================================================
               device  sx28l,oschs3
               device  turbo,stackx,optionx
               IRC_CAL IRC_SLOW
               reset   start_point
               freq    50000000       ; 50 MHz
               org     8
```

```
second          ds      2               ; counter for 1 second tone
delay           ds      1
delay1          ds      1
tone            ds      1               ; tone constant
                org     0
start_point     mov      !rb,#$7F       ; make speaker output
; wait for input button
bwait           jnb     rb.0,bp0
                jnb     rb.1,bp1
                jnb     rb.2,bp2
                jb      rb.3,bwait
; tone 3
                mov     tone,#48
                mov     second,#$01
                mov     second+1,#$01

bp              call    beep
                jmp     bwait

bp2             mov     tone,#24
                mov     second,#$FD
                mov     second+1,#$01
                jmp     bp

bp1             mov     tone,#12
                mov     second,#$FA
                mov     second+1,#$03
                jmp     bp

bp0             mov     tone,#6
                mov     second,#$F4
                mov     second+1,#$07
                jmp     bp

; subroutine
beep
loop            not     rb              ; toggle bits
                mov     delay1,tone
oloop           clr     delay
wloop           djnz    delay,wloop
                djnz    delay1,oloop
                djnz    second,loop
                djnz    second+1,loop
                ret                     ; go back to wherever
```

Unit 6: Low-Level Programming

In the previous units you've written programs that do simple input and output. However, the SX has many powerful I/O features that you can use if you know how they work. Besides input and output capabilities, the SX has more program and data storage than previous programs have used. To access this extra memory, you'll need to understand a special technique called *banking*.

Port Control

The SX has three I/O ports: ports A, B, and C. Port A only has 4 pins. Ports B and C have 8 bits each. You can read and write the pins on a port by accessing the corresponding data register (**ra**, **rb**, or **rc**). You've also seen that you can change the direction of each bit by changing the control register for the port (**!ra**, **!rb**, or **!rc**).

However, the control register gives you many options in addition to the pin direction. Using the control register you can set other options including the threshold voltage for each pin and if the pin uses a Schmitt trigger input or a normal logic-level input. You can also elect to turn on an optional pull up resistor on each pin.

How can a single control register have this much capability? It can't. The trick is that the control register has multiple personalities determined by the **M** or mode register. By default, the **M** register (a 4-bit register) contains $F, which makes the control registers direction registers. When you write a 0 to the control register, it makes the corresponding bit an output, and a 1 makes the bit an input.

If you set the mode register to $E, for example, the control register selects which pins have pull up resistors connected internally. Each bit that is a zero will set a pull up resistor on. Pull up resistors prevent input pins from assuming random states if there is no external circuitry driving the pin. You can set pull up resistors on any of the three ports, by setting **M** to $E and then clearing the corresponding bit (or bits if you want to turn on more than one) in the **!ra**, **!rb**, or **!rc** registers.

You can use the **mov** instruction to load the **M** register with the contents of another register or a literal. You can also use the **mode** instruction to load a literal into **M**. Table 6-1 shows the effects of the control registers for different values of **M** (note that this table does not show settings that pertain to interrupts, a topic covered in the next unit).

Table 6-1: Mode Settings

Mode	!ra	!rb	!rc	SX Name
$F	Direction	Direction	Direction	TRIS_
$E	Pull up	Pull up	Pull up	PLP_
$D	Threshold	Threshold	Threshold	LVL_
$C	N/A	Schmitt	Schmitt	ST_

If you set a threshold bit to 0, the SX will read the input through a CMOS-compatible buffer. This buffer will treat levels below 30% of the supply voltage (say 1.5 V if the supply is 5 V) as a 0. Anything above 70% of the supply voltage (3.5 V) will be a 1. Voltages in between will result in an unpredictable bit, although practical experience shows that the threshold is about 50% of the supply voltage (but Ubicom does not specify this).

When the threshold bit is 1, the input uses a TTL-compatible buffer. Using a TTL compatible buffer treats a 0 as .8 V or less and a 1 as anything over 2 V. For most modern logic circuits, this is acceptable, but interfacing with certain devices may require one setting or the other. Also, when mixing analog circuitry with the processor, you might want to adjust the thresholds to read a particular voltage level.

Ports B and C can use a Schmitt trigger input if you set a zero into the Schmitt register. A Schmitt trigger uses different thresholds depending on the situation. Imagine you are trying to set the temperature of your swimming pool to a particular temperature (say 25 degrees Celsius). You turn on your water heater, and watch the thermometer. When the temperature gets to 25, you turn the heater off. However, the pool loses heat quickly so almost immediately, the temperature drops again and you turn on the heater again. Soon you are turning the heater on and off every few seconds, never able to attain 25 degrees for more than a split second.

A Schmitt trigger uses *hysteresis* to battle this sort of problem. The idea is that the Schmitt trigger will use one threshold to recognize 0 to 1 transitions and another threshold to identify 1 to 0 transitions. A Schmitt trigger might see a voltage rising from .8 to .9 V and output a logic 1 (5 V). However, it might require that the voltage drop below .5 V before returning to the zero state. This prevents a noisy or slow rising signal from causing multiple changes on the output. The SX's Schmitt triggers use 15% and 85% of the supply voltage as trip points. Once the signal rises above 85% of the supply voltage (4.25 V for a 5 V supply), the input reads a 1. It will continue to read a 1 until the input drops below 15% (.75 V).

This can be important when dealing with inputs from real-world sensors, or noisy inputs from long lines. You can also use it to "square up" a signal – for example, reading a digital input from a charging capacitor. Of course, using the Schmitt trigger option overrides the threshold settings for the pin.

Be sure you know the state of the **M** register before you use the control registers. A common mistake is to set the **M** register to some value other than $F, use the control register, and then later try to access the control register to set direction bits. The **M** register stays at the last value you set until a reset occurs..

Analog Capabilities

The SX has one more special capability on port B. Pin 1 and 2 of port B can function as an analog comparator. You can read the comparator's output in software and you can cause pin 0 of port B to reflect the comparator's output as well.

To enable the comparator, you simply set the **M** register to 8 and write a value to **!rb**. A value of $C0 will turn the comparator function off. To turn it on, write either $40 or $00 to **!rb**. If you use $00, the comparator will operate, and pin 0 will act as a comparator output. If you use $40, pin 0 will be free for normal I/O, but the comparator will still function (you'll have to read the result in software).

To read the state of the comparator, make sure **M** contains 8 and write to the **!rb** register. When you write to the comparator register (that is, **M** is equal to 8 and you perform a **mov** to **!rb**) the SX does a little trick behind your back. Instead of simply moving the data to the comparator register, it actually exchanges the **W** register with the comparator register. This is true even if you write:

```
mov     !rb,#0
```

Because this is really the same as writing:

```
mov     W,#0
mov     !rb,W
```

So after writing to the comparator register, the **W** register contains the previous contents. You should only examine bit 0, the comparator status bit, after you've already enabled the comparator with another instruction. If bit 0 is high, then the voltage on B2 is higher than the voltage on B1. If it is low, then the opposite condition is true.

Why would you want a comparator input? Maybe you want the SX to compare the voltage from a potentiometer and a thermocouple. Perhaps you want to divide down your battery voltage and compare it to a known reference so you can detect when your battery is low.

Register Banking

Earlier, you read that the SX has over 100 registers. That might seem odd, because the SX instruction set only has room for 5 bits of data to specify a register. So how can 5 bits refer to over 100 registers? The answer is banking.

Your program does have access to 136 memory locations (not including the special registers like **ind**, **fsr**, **ra**, etc.). However, it can only work with 32 registers at one time. The first 8 registers (register 0 to 7) are the special registers and you can always access them. The registers from 8 to 15 ($8 to $F) are also always accessible – the SX doesn't use them for anything, so you can do what you want with them. This accounts for 16 registers. The other 16 (registers $10-$1F) are available for you to use as you wish. However, there are really 8 sets of these registers. Which set of 16 you are using depends on the **FSR** register. So referring to register $10 may access a different memory location depending on the contents of **FSR**.

Conceptually, the SX memory map consists of 8 32-byte pages. That is, each page has 32 registers in it. However, the first 16 are always the same. The last 16 vary depending on the bank selected. Each register has its own address (although in the case of the shared registers, the actual reference is always between $10 and $1F). You can see this arrangement graphically in Table 6-2.

When you want to access a register, you have several choices. First, if you are using **FSR** anyway, just put the proper address into **FSR** before using **IND** to access the data. So if you want to access the last memory location, load **FSR** with $FF. Your other option is to set the top 3 bits of **FSR** before you access memory. The values you want to use are in the column headings of Table 6-2. You can store a value in **FSR**, of course, with a **mov** instruction. However, this destroys the entire register and it also requires two machine language instructions if you are using a literal value. Since most programs will want to load literals into **FSR**, there is a **bank** instruction. This instruction loads the top 3 bits of a literal into the top 3 bits of **FSR** with a single instruction. This is useful because you can just name the variable you want to access. For example:

```
              org    $FF
last          ds     1
              org    0
              bank   last
              mov    last,#0
```

Notice that although you specified $FF as the argument to **bank**, the actual instruction only uses the topmost three bits ($E).

Table 6-2: SX Memory Map

	FSR=$00	FSR=$20	FSR=$40	FSR=$60	FSR=$80	FSR=$A0	FSR=$C0	FSR=$E0
$00	IND	IND	IND	IND	IND	IND	IND	IND
$01	RTCC	RTCC	RTCC	RTCC	RTCC	RTCC	RTCC	RTCC
$02	PC	PC	PC	PC	PC	PC	PC	PC
$03	STATUS	STATUS	STATUS	STATUS	STATUS	STATUS	STATUS	STATUS
$04	FSR	FSR	FSR	FSR	FSR	FSR	FSR	FSR
$05	PORTA	PORTA	PORTA	PORTA	PORTA	PORTA	PORTA	PORTA
$06	PORTB	PORTB	PORTB	PORTB	PORTB	PORTB	PORTB	PORTB
$07	PORTC	PORTC	PORTC	PORTC	PORTC	PORTC	PORTC	PORTC
$08	8 registers addressable as $08-$0F, $38-$3F, $58-$5F, $78-$7F, $98-$9F, $B8-$BF, $D8-$DF, or $F8-$FF							
$09								
$0A								
$0B								
$0C								
$0D								
$0E								
$0F								
$10	$10	$30	$50	$70	$90	$B0	$D0	$F0
$11	$11	$31	$51	$71	$91	$B1	$D1	$F1
$12	$12	$32	$52	$72	$92	$B2	$D2	$F2
$13	$13	$33	$53	$73	$93	$B3	$D3	$F3
$14	$14	$34	$54	$74	$94	$B4	$D4	$F4
$15	$15	$35	$55	$75	$95	$B5	$D5	$F5
$16	$16	$36	$56	$76	$96	$B6	$D6	$F6
$17	$17	$37	$57	$77	$97	$B7	$D7	$F7
$18	$18	$38	$58	$78	$98	$B8	$D8	$F8
$19	$19	$39	$59	$79	$99	$B9	$D9	$F9
$1A	$1A	$3A	$5A	$7A	$9A	$BA	$DA	$FA
$1B	$1B	$3B	$5B	$7B	$9B	$BB	$DB	$FB
$1C	$1C	$3C	$5C	$7C	$9C	$BC	$DC	$FC
$1D	$1D	$3D	$5D	$7D	$9D	$BD	$DD	$FD
$1E	$1E	$3E	$5E	$7E	$9E	$BE	$DE	$FE
$1F	$1F	$3F	$5F	$7F	$9F	$BF	$DF	$FF

When debugging, the current bank of registers shows up in a bright highlight compared to the inaccessible banks in the debugging window.

If you organize your registers based on your usage of them, you can name your banks meaningfully. Not only does this make your code more readable, but it will often reduce the amount of switching necessary, as well. For example, suppose you reserve one bank of variables (bank $20) for math calculations and another for external communications (bank $40). You can define two symbols, **math** and **extcomm**, so you can write:

```
      bank     math  ; switch to math bank
```

Program Pages

Another place where the SX hides extra memory is in the program space. Although none of your programs have needed it so far, the SX has 4 pages of program memory, and each page contains 512 instructions (remember, instructions on the SX are not bytes). So you can use up to 2 K (2048) instructions.

However, using more than 512 instructions requires careful planning. Every jump instruction (except **jmp w**, **jmp pc+w**, and **ljmp**) only take 9 bits for an address. The extra bits required come from the top 3 bits of the **status** register. Instead of manually setting these bits, however, you can force the assembler to do it for you. Just put an "@" character before the address, like this:

```
             JMP    @FarAwayPlace
```

This actually produces the following instructions:

```
             PAGE   FarAwayPlace
             JMP    FarAwayPlace
```

The **page** instruction sets the status register bits to match the target address. Since using the @ sign requires extra space, you should only use it in cases where the target address resides in a different page.

To complicate things, calling a subroutine across page boundaries is even more difficult. The **call** instruction only takes 8 bits of address. The ninth bit is set to 0, and the remaining bits come from the **status** register just as with **jmp**. That means that a subroutine call can only occur to the first 256 instructions of a page.

This seems like a harsh restriction, but in reality, it is easy to overcome. If you can't organize your subroutines so that they are all in the first half of a page, just place a **jmp** to the subroutine (a single instruction) in the bottom half of the page, and call that instead. Don't forget that data tables (like the ones in Unit 5) are really subroutines so they have the same limitation – the **jmp** instruction that starts the table must be in the first half of the page so that other parts of the program can **call** into the table.

It is worth noting that the program counter is 11 bits long, but the **pc** register contains only the bottom 8 bits. There is no way to directly read the top 3 bits. The only access you have to these bits is when they are loaded from the top 2 or 3 bits of the **status** register.

When you call a subroutine in a different page, you need the processor to restore the full 11-bit address to the program counter. It is also handy to have it set the **status** register to the caller's page so that it can make more subroutine calls on its own page. That is the purpose of the **retp** instruction. It not only restores the full address so that the caller can continue executing, but it also sets the top 3 bits of the return address into the top 3 bits of the **status** register.

The ret instruction and the retp instruction take the same amount of space and execute at the same speed. If there is any chance you might call a subroutine from across page boundaries, use retp. The only exception would be if you wanted the subroutine to deliberately modify the top bits of status.

Reading Program Storage

In the previous unit you saw how to use **retw** to form tables in program memory. There is another way you can access program memory – the **iread** instruction. This instruction takes 4 cycles (unusual for an instruction that doesn't jump or skip). It takes the **M** register and the **W** register as an 11-bit address, reads the 12-bit word at that address, and loads it into the **M** and **W** registers.

How do you get arbitrary data into the program memory? Use **DW** as in:

```
             org    0
start_point  mov    m,#SomeData>>8   ; top part of address
             mov    w,#SomeData&$FF ; bottom part of address
             iread
             nop
             nop
             break
             nop
             sleep

SomeData     dw     $1A5
```

If you debug this program, the **W** register will contain $A5 and the **M** register will contain $1 at the breakpoint.

Be careful if you access the port control registers after executing **iread** since the **M** register will not contain what you expect and that alters the control register's function.

Summary

The techniques in this unit are not that useful for the simple programs you've written up to this point. But in real life, 24 bytes of data storage and 512 instructions only go so far. The key to success with large programs is to carefully plan and organize. If you can keep related variables in the same bank, you'll be much happier. Variables that you use in many parts of your program should be below $10 (the shared area). Of course, with only 8 bytes shared between banks, you have to be very frugal.

Organization for code is important too. Related routines on the same page do not need long jumps. You also need to be mindful of placing subroutines in the second half of any bank, since you won't be able to call them there.

If it seems odd that the SX has all these odd ways to access memory, remember that it is all in the name of compatibility. The SX is backward compatible with other processors that do not have so much memory. The price of having extra resources is extra complexity.

Exercises

1. Write a program to clear all 8 register banks. Be careful not to clear the first 8 registers (which are the special function registers like **pc** and **ind**). Also, don't clear the shared bank more than once. Can you make the clear loop a subroutine?

2. Use **org $200** to place the clearing subroutine in the above program in the first program bank. Single step through the execution.

3. Write a program to convert Celsius temperature to Fahrenheit, using a lookup table accessed with **iread**. Assume the input ranges from 0 to 29 degrees. The formula for conversion, by the way, is F=1.8C+32.

Answers

1. Here is one possible solution:

```
;========================================================================
;Beginning Assembly Language for the SX Microcontroller
;Program 6.1
;========================================================================
               device  sx28l,oschs3
               device  turbo,stackx,optionx
               IRC_CAL IRC_SLOW
               reset   start_point
               freq    50000000       ; 50 Mhz

               org     0
start_point    mov     fsr,#8         ; shared bank
               call    clear
               mov     fsr,#$10
zloop          call    clear
               add     fsr,#$11
               jnc     zloop
               nop
               break
               nop
               sleep

; subroutine clears from FSR until FSR AND $F is 0

clear          clr     ind
               inc     fsr
               mov     w,#$F
               and     w,fsr
               jnz     clear
               dec     fsr     ; back up
               ret
```

2. Moving the subroutine requires you to: 1) place **org $200** in front of the clear routine; 2) change each call to **clear** with one to **@clear**; and 3) change the **ret** instruction to a **retp**. Try performing each of these steps in sequence and debugging the code before making the next change.

3. Here is a simple implementation:

```
             org    8
tempm        ds     1                   ; place to hold M
value        ds     1                   ; value to convert


             org    0
start_point  mov    value,#11           ; 11 degrees C
             call   @convert
             nop
             break
             nop
             sleep
convert      mov    tempm,m
             mov    m,#table>>8
             mov    w,#table & $FF
             add    w,value
             iread
             ; don't need M
             mov    value,w
             mov    m,tempm  ; restore M
             ret

table        dw 32,34,36,37,39,41       ; 0-5
             dw 43,45,46,48,50          ; 6-10
             dw 52,54,55,57,59          ; 11-15
             dw 61,63,64,66,68          ; 16-20
             dw 70,72,73,75,77          ; 21-25
             dw 79,81,82,84             ; 26-29
```

Unit 7: Interrupts

One of the great strengths of modern computers is that they can do more than one thing at a time, right? With a Windows PC, you can surf the Web, work on an e-mail, and touch up a photo from your digital camera, all at the same time. This sounds great except for one thing: most computers (including your Windows PC) only do one thing at a time.

How is this possible? While it is true that most computers can only do one thing at a time, they can do one thing very rapidly. Modern operating systems allocate small chunks of time to each active task. In this way, each task appears to run at the same time. Also, modern computers can respond to external events – for example, a keystroke or a mouse movement. This also helps with the illusion that the computer is performing many tasks since the computer can handle events as they occur instead of waiting for them.

To get this sort of capability, a computer needs a way to track time and it also needs a way to stop what it is doing in favor of another task. The SX has two features that work together in this area: the *real time clock counter* (**RTCC** register) and *interrupts*. The **RTCC** register does just what its name implies: it increments on a precise predetermined interval regardless of what else the processor is doing. It can also increment in response to an external pulse input. Interrupts allow an external event or a time period to trigger a piece of your program. Whatever the SX was doing before the event is put on hold until the event code (an *interrupt service routine* or *isr*) completes.

In assembly language programming, interrupts have a reputation as being difficult to use. It is true that interrupts require careful planning. However, the SX has several features that make dealing with interrupts less troublesome than with many other similar processors.

What constitutes an event? One common event is when the **RTCC** register rolls over (that is, changes from $FF to $00). You can also configure interrupts to occur on rising or falling edges on any (or all) port B pins. To use interrupts, you must configure them first – by default no interrupts occur.

The Real Time Clock Counter

One of the most common sources of interrupts is when the **RTCC** register's value changes from $FF to $00. This indicates that 256 time periods have elapsed or 256 external events occurred. Using this interrupt, you can receive interrupts at a regular time interval which is useful for keeping time, measuring pulse widths, generating pulses, and other time-sensitive operations.

What causes the **RTCC** register to increment depends on bit 5 of the **!option** register (**RTS**). If this bit is 0, the counter increases with each instruction cycle. If the bit is 1, then **RTCC** increments each time it detects a pulse on the **RTCC** pin. By using the **RTE** bit (bit 4 of **!option**) you can determine if the counter responds to rising edges (0) or falling edges (1).

By default, the **RTCC** register increments on each instruction cycle or external event. At 50 MHz, then, the **RTCC** requires 20 ns * 256 = 5.12 µs to roll over when counting instruction cycles. This time is too short for most purposes (as you'll see shortly), so you'll often want to divide the clock cycle by some factor. You can do this by assigning the prescaler to **RTCC**. This is the same prescaler the watchdog timer uses, so you have to assign it to one use or the other. You can't scale the **RTCC** count and the watchdog timer at the same time.

To assign the prescaler to **RTCC**, clear bit 3 of the **!option** register (**PSA**). The last 3 bits in the **!option** register determine the division rate (see Table 7-1). The maximum ratio is 1:256 which at 50 MHz works out to 1.3 ms (.0013 s). Of course, if you are using a different clock frequency all of these times will be different as well. Obviously, if you are using an external source to drive the **RTCC** pin, the time between rollovers depends on the external source.

Notice that Table 7-1 does not contain a 1:1 setting. That is because a 1:1 setting is what you get when the prescaler is working for the watchdog timer.

Table 7-1: Prescaler Settings

PS2	PS1	PS0	Ratio	Roll overTime at 50 MHz
0	0	0	1:2	10.24 µs
0	0	1	1:4	20.48 µs
0	1	0	1:8	40.96 µs
0	1	1	1:16	81.92 µs
1	0	0	1:32	163.84 µs
1	0	1	1:64	327.67 µs
1	1	0	1:128	655.35 µs
1	1	1	1:256	1310.72 µs (1.31 ms)

RTCC Delays

Even without interrupts, the **RTCC** register can be useful. In previous units, programs used a programmed delay to pause for a particular interval. If the **RTCC** is incrementing with the instruction clock, you can use it to time your delays easily. Take a look at this subroutine:

```
; assume prescaler is 1:256
delay1_3ms    mov    rtcc,#1

; testing for zero is ok because the 256 prescaler is on
:wait         mov    w,rtcc
              jnz    :wait
              ret
```

The subroutine sets **rtcc** to 1 (which also, incidentally, clears the prescaler). It then waits for **rtcc** to equal zero. This will require 255 counts and each count requires 256 instruction cycles. Therefore, at 50 MHz, the total delay is 256*255*20 ns = 1.3 ms.

Don't forget that writing to **rtcc** clears the prescaler. This can lead to subtle side effects. For example, you might be tempted to use the **test** instruction to test the prescaler for a zero value. This won't work because using **test** is the same as moving a register into itself. While this does test for zero, it also clears the prescaler so that the **rtcc** register never increments.

Another pitfall is testing for equality. If the prescaler is not set, **rtcc** increments on each instruction cycle. Then it would be dangerous to test for a single value of the prescaler. Why? Because **rtcc** might assume that value while you are executing another instruction. For example, suppose the subroutine above loads **w** with $FF at the **:wait** label. With prescaling off, the next time through the loop the counter will be 3 – it was zero during the **jnz** instruction!

RTCC Interrupts

To enable **RTCC** rollover interrupts, clear the **RTI** bit (bit 6) in the **!option** register. Once this bit is clear, the processor will stop whatever it is doing when **RTCC** rolls over and execute the code starting at location 0. Of course, up until now, your program started at location 0, but that is only because the **reset** directive pointed there. You can start your program further up in memory to allow for interrupt processing.

When an interrupt occurs, the SX disables further interrupts. It also saves **status**, **fsr**, and **w**. The SX then clears the top 3 bits of the **status** register (these bits form the top portion of jump addresses) and jumps to address 0. All of this work is necessary so that the interrupt service routine (ISR) does not interfere with the execution of the main program. Once the ISR is finished, it uses the **reti** instruction to restore control to the main program. This also enables future interrupts.

Unlike many other processors, the SX automatically stores its context (the **w**, **fsr**, and **status** registers) in special temporary areas, not the stack. However, the chip does not service interrupts if they occur while still processing a previous interrupt.

Perhaps the simplest way to use the **rtcc** interrupt is to simulate a wider real time clock. Remember that even with the maximum prescaling in effect, **rtcc** rolls over every 1.3 ms or so (at 50 MHz). What if you wanted to delay 100 ms? Sure you could call the 1.3 ms delay nearly 100 times. But if you had a 16-bit **rtcc** register you could simply wait for the count to exceed 19531 (each count is worth about 5 µs when the prescaler is at 1:256).

Here is a simple 100 ms LED flasher based on these ideas:

```
;=========================================================================
;Beginning Assembly Language for the SX Microcontroller
;Program 7.1
;=========================================================================
               device  sx28l,oschs3
               device  turbo,stackx,optionx
               IRC_CAL IRC_SLOW
               reset   start_point
               freq    50000000       ; 50 MHz
               org     8
rtcc1          ds      1

               org     0
isr            inc     rtcc1          ; interrupt handler
               reti

start_point
               mov     !rb,#$80       ; 7 outputs, 1 input

; set RTCC to internal clock 1:256 ratio
               mov     !option,#$87
loop           xor     rb,#$FF
               call    delay100ms
               jmp     loop


delay100ms     clr     rtcc
               clr     rtcc1
:wait          mov     w,#$4c         ; $4c4b is 19531
               mov     w,rtcc1-w
               jnz     :wait
:wait0  mov            w,#$4b
```

```
               mov     w,rtcc-w
               jnz     :wait0
               ret
```

Periodic Interrupts

In the previous examples, the main program blinks an LED and controls the delay between flashes of the lamp. However, the real power to interrupts is allowing the ISR to perform a task, seemingly while the main routine is executing. Look at this program:

```
;=========================================================================
;Beginning Assembly Language for the SX Microcontroller
;Program 7.2
;=========================================================================
               device  sx28l,oschs3
               device  turbo,stackx,optionx
               IRC_CAL IRC_SLOW
               reset   start_point
               freq    50000000               ; 50 MHz
               org     8
rtcc1          ds      1

               org     0
isr            inc     rtcc1
               cjne    rtcc1,#$4D,iout        ; blink every $4D00 periods
               xor     rb,#$FF
; reset time
               clr     rtcc
               clr     rtcc1
iout
               reti

start_point
               mov     !rb,#$80               ; 7 outputs

; set RTCC to internal clock 1:256 ratio
               mov     !option,#$87
loop
               jmp     loop
```

The main program sets **!rb**, **!option**, and then does a simple **jmp** instruction to loop forever doing nothing. All the work occurs in the ISR. It is interesting to note that the ISR resets the **rtcc** register so that the interrupt will occur periodically. This isn't unusual when you want the interrupt to repeat at a regular interval.

There is one problem with this, however. A complex ISR may take a different amount of time to execute depending on the current situation. This can lead to timing errors intolerable in precise applications. For example, in the above piece of code, the **reti** instruction adds a slight delay to the total time although for this application it is negligible.

A better answer is to use the **retiw** instruction to end the ISR – especially if the prescaler is off. This instruction adds the **w** register's contents to **rtcc**. Say the processor is set so that **rtcc** will cause an interrupt when it rolls over and that the prescaler is assigned to the watchdog timer. Each count of the **rtcc** represents 20 ns (assuming, as always, a 50 MHz clock). When the interrupt begins the **rtcc** has already counted to 3. As the ISR continues, the **rtcc** continues to increase. To accurately set the time, you have to take into consideration how much time has already elapsed. Luckily, there is a simple answer – the **rtcc** register already has this information! If you subtract the number of cycles you want between each interrupt from the number of cycles already elapsed, you are left with the exact number of cycles required.

For example, say you want an interrupt to occur every 50 cycles (1 µs). You can simply use the following two lines of code at the end of your ISR:

```
mov    w,#-50
retiw
```

The only catch is that your ISR, including the 3 cycle interrupt latency, must not exceed 46 cycles. If it does, you'll either miss the next interrupt, or you will return to the main program only to have an interrupt occur immediately. Because of the interrupt latency you must always allow 3 cycles plus at least enough time for one instruction to execute in the main program – figure a total of 6 cycles. However, even then your main program will not execute very often – you should allow a more generous time slice between interrupts in most cases.

A Clock Example

A computer that knows what time it is can be very useful. You might want to count down a model rocket launch, or time stamp readings from a sensor. With an accurate interrupt it is easy to keep the time. The hard part is translating the rapid stream of interrupts into numbers more meaningful to humans. Here is a simple program that uses a 50 MHz clock to the **rtcc** register. The ISR adds –50 to **rtcc** so that it generates a periodic interrupt every 1µs. The ISR maintains two 16-bit counters to count microseconds and milliseconds.

Of course, every 1000 milliseconds constitutes a second, every 60 seconds is a minute, and 60 minutes make an hour. You could easily extend this to track days if you wanted to do so. The main program in this case doesn't do anything, but you could easily add whatever code you wanted.

This is a hard program to debug because single stepping it doesn't show the correct time. You can run the program at full speed in the debugger and press the Poll button to see the time change. You'll also see LEDs on port B blink and, if you connect a piezo speaker to one of the port B pins, you'll hear your SX clock ticking.

```
;========================================================================
;Beginning Assembly Language for the SX Microcontroller
;Program 7.3
;========================================================================
               device  sx28l,oschs3
               device  turbo,stackx,optionx
               IRC_CAL IRC_SLOW
               reset   start_point
               freq    50000000              ; 50 MHz
               org     8
microlow       ds      1
microhi        ds      1
millilow       ds      1
millihi        ds      1
seconds        ds      1
minutes        ds      1
hours          ds      1
               watch   hours,8,udec
               watch   minutes,8,udec
               watch   seconds,8,udec

               org     0
isr            inc     microlow
               snz
               inc     microhi
               cjne    microhi,#$03,iout     ; blink every $03e8 periods
               cjne    microlow,#$e8,iout
; 1000 uS already!
               clr     microlow
               clr     microhi
               inc     millilow
               snz
               inc     millihi
               cjne    millihi,#$03,iout
               cjne    millilow,#$e8,iout
; 1000 ms!
               clr     millihi
               clr     millilow
               xor     rb,#$FF               ; toggle LEDs
               inc     seconds
               cjne    seconds,#60,iout
; seconds roll over
               clr     seconds
               inc     minutes
               cjne    minutes,#60,iout
; minutes roll over
               clr     minutes
               inc     hours
               cjne    hours,#24,iout
; hour roll over
               clr     hours
; could track days if we wanted to

; reset time
iout
               mov     w,#-50                ; interrupt every 1uS
               retiw

start_point
               mov     !rb,#$00              ; all outputs
               clr     microhi
               clr     microlow
               clr     seconds
               clr     hours
               clr     minutes
; set RTCC to internal clock 1:1 ratio
               mov     !option,#$88          ; no prescale
loop
               jmp     loop
```

External Interrupts via RTCC

When you think of using the **RTCC** pin to monitor external events, you usually think of counting pulses. You can certainly do this, of course. When you set bit 4 of **!option** (the **RTS** bit), the pin monitors pulses and uses them to increment **RTCC**. If the **RTE** bit (bit 4 of **!option**) is clear, the count occurs on rising edges, otherwise the SX detects falling edges. The prescaler is still available, so you can divide the input down if you like.

However, what if you want a single external interrupt? At first glance, it would seem that you can't do this with **RTCC**. After all, even with the prescaler assigned to the watchdog timer, you still need 256 pulses to get a single interrupt, right?

While that seems true, there is a trick you can use to make **RTCC** simulate an external interrupt. Simply load the **RTCC** register with $FF. Assuming the prescaler is off and the **RTS** bit is set, the next input pulse will cause an interrupt. A simple but effective technique. Of course, the ISR will then need to reset **RTCC** to $FF before issuing a **reti** instruction so the interrupt will be "armed" for the next event.

Port B Multi Input Wakeup

In addition to the **RTCC** trick, you can configure any (or all) of port B's pins as external interrupts. Port B has two special registers that allow it to detect input edges. These register work at all times, not just when interrupts are enabled. That means you can detect input edges with or without using interrupts.

Like other special port registers, you access these by using **!rb** while the **M** register is set to a special value. If **M** is $A, you can select which edge each pin monitors. A 1 bit in this register makes the SX detect falling edges (that is, 1 to 0 transitions) on the corresponding pin. A 0 bit detects 0 to 1 transitions or rising edges. When the selected edge appears on a pin, the SX sets the corresponding bit in the multi-input wake up (MIWU) pending register (**!rb** with **M** = $9). The SX never clears this register. When your program writes the **W** register into **!rb** and **M** is $9, the SX actually swaps the two values. So you can read the pending bits and clear them at the same time.

This processing occurs at all times. Most programs just ignore this feature. However, you can use it to detect when an edge occurred even when you aren't using the port B interrupts. If you connect the circuit in Figure 7-1 to several port B pins, you can try this program:

```
;======================================================================
;Beginning Assembly Language for the SX Microcontroller
;Program 7.4
;======================================================================
               device  sx28l,oschs3
               device  turbo,stackx,optionx
               IRC_CAL IRC_SLOW
               reset   start_point
               freq    50000000              ; 50 MHz
               org     8
microlow       ds      1
microhi        ds      1
millilow       ds      1
millihi        ds      1
seconds        ds      1
minutes        ds      1
hours          ds      1
edges          ds      1

               watch   hours,8,udec
               watch   minutes,8,udec
               watch   seconds,8,udec

               org     0
isr            inc     microlow
               snz
               inc     microhi
               cjne    microhi,#$03,iout     ; blink every $03e8 periods
               cjne    microlow,#$e8,iout
; 1000 uS already!
               clr     microlow
               clr     microhi
               inc     millilow
               snz
               inc     millihi
               cjne    millihi,#$03,iout
               cjne    millilow,#$e8,iout
; 1000 ms!
               clr     millihi
               clr     millilow
               inc     seconds
               cjne    seconds,#60,iout
; seconds roll over
               clr     seconds
               inc     minutes
               cjne    minutes,#60,iout
; minutes roll over
               clr     minutes
               inc     hours
               cjne    hours,#24,iout
; hour roll over
               clr     hours
; could track days if we wanted to

; reset time
iout
               mov     w,#-50                 ; interrupt every 1uS
               retiw

start_point
               mov      !rb,#$FF
areset         clr     microhi
               clr     microlow
               clr     seconds
               clr     hours
               clr     minutes
; set RTCC to internal clock 1:1 ratio
               mov     !option,#$88           ; no prescale

; Turn on port B pull up resistors
               mode     $E
               mov     !rb,#$00
; set port B pin 0 to interrupt on falling edge
               mode    $A                     ; select edge
               mov     !rb,#$FF
               mode    $9                     ; enable interrupts
               mov     !rb,#%0                ; clear pending
; wait for 10 seconds
wait10         cjne    seconds,#10,wait10
               mov     !rb,#%0                ; read pending and clear
               mov     edges,w
; important: reset mode register
               mode    $F
               mov     !rb,#0                 ; set to outputs
; flip sense of edge bits
               not     edges
               mov     rb,edges
```

```
loop
; active wait so ticking will occur
               jmp     loop
```

This is more or less the same program as before, but it doesn't produce the blinking lights and ticking effect. Instead, it waits 10 seconds (easy to do with the clock interrupt routine) and then turns on lights that correspond to the buttons you pushed during that 10 seconds. This is trivially easy using the MIWU feature. Since the LEDs turn on when the port outputs a 0, the program uses the **not** instruction to invert the pending bits.

This program initially sets the direction register so that all port B pins are inputs. Then, after the pause, it sets all pins to outputs. An easy mistake to make here is to forget to set the **M** register back to $F before switching to outputs. The edge detection code changes the **M** register, and you must change it back to $F before accessing the direction register.

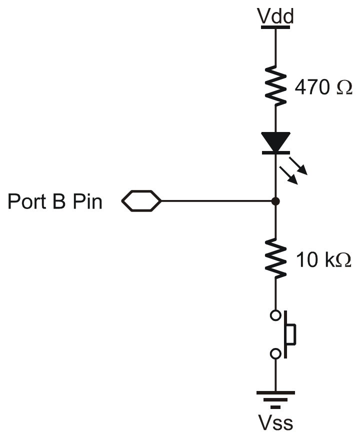


Figure 7-1: Switch/LED Circuit

Port B Interrupts

When the SX detects an edge, it can also generate an interrupt. You can select this behavior by clearing bits in the **!rb** register while **M** is equal to $B. When the SX detects an edge on the corresponding pin, it will generate an interrupt. It is up to the ISR to examine the pending register and clear it for further interrupts. This interrupt is exactly like an **rtcc** interrupt – it saves the SX context and starts at location 0.

It is possible to use port B interrupts and **rtcc** interrupts at the same time, but it can be tricky. For example, if a pulse occurs while the ISR executes, the SX will not generate interrupts after the ISR returns until a new event occurs. By the same token, if **rtcc** rolls over while the SX is processing a port B interrupt, you will miss the **rtcc** interrupt. In some cases, timing is not that critical, so losing a microsecond or two isn't that important. However, if you require solid time accuracy you should consider only dealing with one interrupt source (port B or **rtcc**) in one program.

If you need a real-time clock and edge detection, think about using the **rtcc** interrupt at a fast rate and simply examine the pending bits on each timer tick (this is often known as *polling*). For many applications, scanning the inputs quickly is good enough and this does not interfere with accurate timing of the **rtcc** interrupt.

It is also possible to use the port B interrupt to wake up after a **sleep** instruction. If a port B interrupt occurs after a **sleep** instruction, an interrupt does not occur. Instead, the processor resets with bit 3 of the **status** register clear and bit 4 set. Although port B interrupts will interrupt the SX's sleep, an **rtcc** interrupt will not. This is often used to put the processor to sleep (which conserves power) until a key is pressed, for example.

Summary

Interrupts need not be difficult to use. This is especially true of the SX because the chip takes care of many details for you. Interrupts are essential when you need to process inputs while doing something else, keep track of time, or generate precise outputs while doing other tasks.

Interrupts, coupled with the SX's high speed, form the basis for the virtual peripheral strategy discussed in the next unit. Although interrupt handling requires a bit of careful design, and can be difficult to debug, they are well worth the price.

Exercises

1. Write a program that uses a timer interrupt to track (at least) seconds. Normally, the program does nothing. However, when you press a button connected to pin 0 of port B, the program should flash an LED (or click a piezo speaker) every second until you push the button again. Pushing the button a third time should resume LED flashing and so on. Use the **rtcc** interrupt for timing and poll the switch in the main program.

2. Modify the above program so that the ISR samples the input switch using the MIWU capability but do not use the port B interrupts.

3. Modify the program again so that you use both interrupts; the **rtcc** and the port B interrupt.
4. Which of the three programs do you think uses the best approach?

Answers

1. The solution is straightforward. Notice you can't use the **sleep** instruction or else the program will just halt.

```
;=======================================================================
;Beginning Assembly Language for the SX Microcontroller
;Program 7.5
;=======================================================================
               device  sx28l,oschs3
               device  turbo,stackx,optionx
               IRC_CAL IRC_SLOW
               reset   start_point
               freq    50000000              ; 50 MHz
               org     8
microlow       ds      1
microhi        ds      1
millilow       ds      1
millihi        ds      1
seconds        ds      1
ticker         ds      1
tmp            ds      1


               org     0
isr            inc     microlow
               snz
               inc     microhi
               cjne    microhi,#$03,iout      ; blink every $03e8 periods
               cjne    microlow,#$e8,iout
; 1000 uS already!
               clr     microlow
               clr     microhi
               inc     millilow
               snz
               inc     millihi
               cjne    millihi,#$03,iout
               cjne    millilow,#$e8,iout
; 1000 ms!
               clr     millihi
               clr     millilow
               test    ticker
               jz      notick
               xor     rb,#$FF               ; toggle LEDs
notick         inc     seconds
iout
               mov     w,#-50                ; interrupt every 1uS
               retiw
```

```
start_point
               mov     !rb,#$01              ; 7 outputs, 1 in
areset         clr     microhi
               clr     microlow
               clr     seconds
               clr     ticker
; set RTCC to internal clock 1:1 ratio
               mov     !option,#$88          ; no prescale

loop
; active wait so ticking will occur
               jb      rb.0,loop
; button pushed
               not     ticker
; debounce delay (about 1 second)
milloop0       test    millihi               ; wait for millhi to go to 0
               jnz     milloop0
milloop1       test    millihi
               jz      milloop1              ; wait for nonzero
milloop test           millihi
               jz      milloop               ; wait for zero again
               jmp     loop
```

2. Compared to the last program, this one has a similar ISR, but a very different main program (all the work is in the ISR). Notice that the ISR changes the **M** register, so it has to save and restore it to ensure the main program's **M** register does not change (of course, in this case, the main program doesn't care, but that will not usually be the case). To protect against bounce, the code examines the edge pending register every 1 ms.

```
;=====================================================================
;Beginning Assembly Language for the SX Microcontroller
;Program 7.6
;=====================================================================
               device  sx28l,oschs3
               device  turbo,stackx,optionx
               IRC_CAL IRC_SLOW
               reset   start_point
               freq    50000000              ; 50 MHz
               org     8
microlow       ds      1
microhi        ds      1
millilow       ds      1
millihi        ds      1
seconds        ds      1
ticker         ds      1
tmp            ds      1


               org     0
isr
```

```
               inc     microlow
               snz
               inc     microhi
               cjne    microhi,#$03,iout      ; blink every $03e8 periods
               cjne    microlow,#$e8,iout
; 1000 uS already!
               clr     microlow
               clr     microhi

; check for key every 1ms
               mov     tmp,M                  ; save M register
               mode    $9
               clr     w
               mov     !rb,w                  ; exchange w and pending
               and     w,#1                   ; test low bit
               sz
               not     ticker                 ; invert ticker
               mov     M,tmp                  ; restore M

; roll millisecond

               inc     millilow
               snz
               inc     millihi
               cjne    millihi,#$03,iout
               cjne    millilow,#$e8,iout
; 1000 ms!
               clr     millihi
               clr     millilow
               test    ticker
               jz      notick
               xor     rb,#$FF                ; toggle LEDs
notick         inc     seconds
iout
               mov     w,#-50                 ; interrupt every 1uS
               retiw

start_point
               mov     !rb,#$01               ; 7 outputs
areset         clr     microhi
               clr     microlow
               clr     seconds
               clr     ticker
; set RTCC to internal clock 1:1 ratio
               mov     !option,#$88           ; no prescale
; set port B detect falling edge
               mode    $A                     ; select edge
               mov     !rb,#$FF

loop
               jmp     loop
```

3. This version is perhaps the least satisfactory of the three. It requires switches that don't bounce much since it is difficult to filter multiple interrupts caused by bouncing. Also, if an **rtcc** event occurs during processing for a switch closure, the time becomes inaccurate.

```
;=========================================================================
;Beginning Assembly Language for the SX Microcontroller
;Program 7.7
;=========================================================================
               device  sx28l,oschs3
               device  turbo,stackx,optionx
               IRC_CAL IRC_SLOW
               reset   start_point
               freq    50000000              ; 50 MHz
               org     8
microlow       ds      1
microhi        ds      1
millilow       ds      1
millihi        ds      1
seconds        ds      1
ticker         ds      1
tmp            ds      1


               org     0
isr
; check for pending key
               mov     tmp,M                 ; save M register
               mode    $9
               clr     w
               mov     !rb,w                 ; exchange w and pending
               and     w,#1                  ; test low bit
               jz      rtccisr
               not     ticker                ; invert ticker
               mov     M,tmp                 ; restore M
iret

rtccisr
               mov     M,tmp
               inc     microlow
               snz
               inc     microhi
               cjne    microhi,#$03,iout     ; blink every $03e8 periods
               cjne    microlow,#$e8,iout
; 1000 uS already!
               clr     microlow
               clr     microhi
```

```
; roll millisecond

               inc     millilow
               snz
               inc     millihi
               cjne    millihi,#$03,iout
               cjne    millilow,#$e8,iout
; 1000 ms!
               clr     millihi
               clr     millilow
               test    ticker
               jz      notick
               xor     rb,#$FF               ; toggle LEDs
notick         inc     seconds
iout
               mov     w,#-50                ; interrupt every 1uS
               retiw
start_point
               mov     !rb,#$01              ; 7 outputs
areset         clr     microhi
               clr     microlow
               clr     seconds
               clr     ticker
; set RTCC to internal clock 1:1 ratio
               mov     !option,#$88          ; no prescale
; set port B detect falling edge
               mode    $A                    ; select edge
               mov     !rb,#$FF
               mode    $B                    ; enable interrupt on pin 0
               mov     !rb,#$FE

loop
               jmp     loop
```

4. It is fairly clear that program #3 would require a great deal of work to make it robust. Mixing two interrupt sources is a risky business. Of the other two techniques, it boils down to personal taste. The code in #1 has more portions of the program in the main loop where they will be easier to debug. However, #2 is quite clean and keeps the processing out of the way of the main program (presumably, you'd be doing something in the main program).

Unit 8: Virtual Peripherals

Most (if not all) microcontrollers are valuable because they communicate with the outside world in some way. As a result, system designers spend a lot of time interfacing microcontrollers to the outside world. With old-fashioned processors, everything required additional electronic components. Want to read a voltage? Get an A/D (analog to digital) chip. What to talk to a PC? Get a UART (Universal Asynchronous Receiver and Transmitter) chip.

In recent years, microcontroller manufacturers have been integrating common peripheral chips directly into the microcontroller. This allows for simpler system design and conserves the controller's I/O capacity. The only problem is, no microcontroller can have every possible peripheral. For one project you might need a UART. The next project might require two A/D inputs. Still another project might require a single A/D but two UARTs. Obviously, no matter how clever the microcontroller designers are, you will never be able to have all peripherals built into the microcontroller.

Another problem with this approach is that you have to have different microcontrollers for different tasks. You can't take a microcontroller with a built-in A/D and use it in place of one that has a UART. This makes it complicated to control your inventory of microcontrollers. Ideally, you'd like to use the same part in all of your designs. At the least, you want the fewest number of different parts possible.

The SX address this problem via *Virtual Peripherals* or VPs. VPs take advantage of the SX's raw speed and interrupt capability to simulate traditional peripheral devices in software instead of hardware. This has many advantages:

- Use one part for all designs
- Add whatever devices you need for a particular project
- Modify devices to meet your needs – not usually possible in hardware

A VP is simply a code module (usually an interrupt service routine or ISR) that simulates an I/O device. You can download many VPs from the Parallax Web site. Other VPs may be available (for free or for a fee) from third parties. You can even write your own VPs for use in later projects or to sell to other programmers. Some VPs do require a few external components (usually a few resistors or capacitors). Others work completely in software.

Using a Virtual Peripheral

When you begin designing a project around the SX, you should first see if there are any standard VPs that would be of use to you. Here are a few of the more useful VPs that exist:

- DTMF Generation – Generates TouchTones

- FSK Detection – Receives frequency shift keying data
- FSK Generation – Generates frequency shift keying
- I2C – Interface with IIC-bus chips (one VP for slave, another for master)
- SPI – Interface with SPI-bus chips (one VP for slave, another for master)
- UART – Serial I/O (up to 230.4 Kbaud)
- Multi UART – 8 serial ports each running at 19.2 Kbaud
- LCD – Drives a standard Hitachi LCD module (one VP for 4 bit, another for 8 bit)
- LED – Drives seven segment LEDs
- PWM – A variety of VPs allow you to generate pulse width modulation, useful for generating voltages, controlling motor speeds and similar tasks
- ADC – You can actually use a few common parts to make an ADC almost completely in software
- Stepper Motor – Control stepper motors
- Timers – Common VPs can implement timers and real-time clocks
- Input – VPs exist that can debounce buttons and scan keypads

Be sure to check out the latest list on the Parallax Web site.

Once you select a VP, you need to integrate it into your program. You might be tempted to use more than one VP. You can do this (see below), but for now just pick one. As an example, suppose you wanted to build a circuit that would dial the Parallax telephone number using TouchTones over a piezo speaker connected to Port C pin 6.

If you look on the Parallax Web site, you'll see that there is a document file that describes the DTMF generation VP and source code to an example program. One problem is that the example program invariably does things you'd rather not do, so you have to cut and paste the pieces you want into your program.

The example program reads data from an RS-232 port and dials the number as instructed. For this example, you don't need the serial I/O VP. However, a quick examination of the example's ISR shows that it also contains PWM and timer VPs. Detailed examination reveals that both are necessary for the DTMF VP.

In addition to the ISR, you also have to get the variables that the routines use and several subroutines that help you access the VP's functions. The VP may also require specific initialization of port control registers, the **!option** register, or internal variables. In the end you may have to resort to a bit of trial and error unless you are prepared to fully comprehend what the program is doing.

Once you think you have everything you need, you might want to use the Run | Assemble command to see if you get any assembly errors. If you don't, then you probably have everything you need (although you may have extra things too if you are not careful).

Often, the VP does not use the same port assignments as you'd like to use. Usually you can interchange the pin numbers with no ill effects. However, be careful. If the VP is using, for example, port B's interrupt capabilities, you won't be able to move pins to port A or C which do not have interrupts. Usually the VP will have an equate near the top that sets the I/O definitions (**PWM_pin**, in this case). This is misleading, however. In addition to changing the equate, you also have to find all the places where the VP references the **ra**, **rb**, **rc**, **!ra**, **!rb**, or **!rc** registers and correct these lines as well.

With the VP in place, the main program is trivially simple:

```
; load digits
             clr    i
digloop      call   getdigit
             mov    byte,w
             cje    byte,#$FF,done
             call   @load_frequencies          ; VP routine
             call   @dial_it                   ; VP routine
inc          i
             mov    w,#20
             call   @delay_10n_ms
             jmp    digloop
done
             sleep
```

To dial again, reset the processor. The **load_frequencies**, **dial_it**, and **delay_10n_ms** routines are all part of the VP (and they reside on different pages which explains the at sign prefix). The **getdigit** routine is a simple lookup table that returns the phone number digits (you'll write this code in the exercises for this unit).

Mixing Virtual Peripherals

When you need to mix VPs, there are several areas you have to consider:

- At what frequency must the ISRs run?
- Port and variable conflicts
- Conflicting uses of the **!option** register
- Varying time paths through the ISR

Most of these issues are straightforward. Sometimes you can adjust parameters to resolve conflicts. For example, if you need a UART, you can adjust its timing so that it will work with other VPs that don't use the same frequency. Sometimes it is more difficult and requires significant effort to rewrite the VPs code.

Another issue is varying time paths through the ISR. Some VPs depend on an exact amount of time passing between interrupts. PWM generation, for instance, requires precise timing. If you merge a VP that requires an exact amount of time between interrupts with another VP, you should place the time-sensitive VP's interrupt code before the other VP's code. Reversing this order will upset the sensitive VP if the other VP's ISR does not always require the same time to execute. A few VPs use special techniques to ensure that they always require the same amount of time to execute, but most can take varying times depending on conditions.

Summary

Using VPs you can create powerful programs easily. However, it does take a bit of experience and effort to peel away the interesting parts of the VP examples and apply them to your program. The effort, however, is usually far less than it would take you to duplicate the VPs features in either hardware or software.

You can mix VPs if you are careful. However, blending together VPs can often be taxing as you try to make peace between conflicting requirements for each module.

Exercises

1. Download the DTMF generation VP and remove the portions that are unnecessary for building an auto dial program that automatically dials a phone number when it starts.

2. Move the DTMF output to Port C pin 6.

3. Add your own code to dial a number of your choice each time the processor resets. Put the processor to sleep after dialing. To hear the tones, you can connect a piezo speaker to the port. However, this will probably be too rough and too weak to really dial a phone. If you want to really dial the phone, add an RC filter (see the instructions in the VP documentation; you'll need a 600 Ω resistor and a capacitor around .2 µF). You can then use an amplified speaker or signal tracer to increase the volume to where it can really dial the phone.

Answers

(All). Here is the listing that satisfies the three problems in this unit (note some of the VP code is in the second program bank – the main program is in the middle of the listing):

```
;=======================================================================
;Beginning Assembly Language for the SX Microcontroller
;Program 8.1
;=======================================================================
               device  sx28l,stackx,optionx
               device  oschs3,turbo

               freq    50_000_000     ; default run speed = 50 MHz
               ID      'DIAL'

               reset   start          ; JUMP to start label on reset


;***********************************************************************
; Equates for common data comm frequencies
;***********************************************************************
f697_h         equ     $012           ; DTMF Frequency
f697_l         equ     $09d

f770_h         equ     $014           ; DTMF Frequency
f770_l         equ     $090

f852_h         equ     $016           ; DTMF Frequency
f852_l         equ     $0c0

f941_h         equ     $019           ; DTMF Frequency
f941_l         equ     $021

f1209_h        equ     $020           ; DTMF Frequency
f1209_l        equ     $049

f1336_h        equ     $023           ; DTMF Frequency
f1336_l        equ     $0ad

f1477_h        equ     $027           ; DTMF Frequency
f1477_l        equ     $071

f1633_h        equ     $02b           ; DTMF Frequency
f1633_l        equ     $09c


;***********************************************************************
; Pin Definitions
;***********************************************************************
```

```
;PWM_pin         equu     rb.7           ; DTMF output
PWM_pin         equ     rc.6            ; DTMF output

;*********************************************************************
;       Global Variables
;*********************************************************************
                org     $8              ; Global registers

flags           ds      1
dtmf_gen_en     equ     flags.1         ; Tells if DTMF output is enabled
timer_flag      equ     flags.2         ; Flags a rollover of the timers.
temp            ds      1               ; Temporary storage register
byte            ds      1               ; a byte
i               ds      1               ; loop counter

;*********************************************************************
;       Bank 0 Variables
;*********************************************************************
                org     $10

sin_gen_bank    =       $

freq_acc_high   ds      1               ;
; 16-bit accumulator which decides when to increment the sine wave
freq_acc_low    ds      1
freq_acc_high2  ds      1               ;
; 16-bit accumulator which decides when to increment the sine wave
freq_acc_low2   ds      1
freq_count_high         ds      1       ; freq_count = Frequency * 6.83671552
freq_count_low  ds      1               ; 16-bit counter
;decides which frequency for the sine wave

freq_count_high2        ds      1       ; freq_count = Frequency * 6.83671552
freq_count_low2         ds      1       ; 16-bit counter which
                                        ; decides which frequency
                                        ; for the sine wave

curr_sin        ds      1               ; The current value of the sin wave
sinvel          ds      1               ; The velocity of the sin wave

curr_sin2       ds      1               ; The current value of the sin wave
sinvel2         ds      1               ; The velocity of the sin wave

sin2_temp       ds      1               ; Used to do a temporary shift/add
register

PWM_bank        =       $

pwm0_acc        ds      1               ; PWM accumulator
pwm0            ds      1               ; current PWM output
```

```
;*********************************************************************
;       Bank 1 Variables
;*********************************************************************
               org     $30            ;bank3 variables
timers         =       $
timer_l        ds      1
timer_h        ds      1

;*********************************************************************
; Interrupt
;
; With a retiw value of -163 and an oscillator frequency of 50 MHz, this
; code runs every 3.26us.
;*********************************************************************
               org     0
;*********************************************************************
PWM_OUTPUT
; This outputs the current value of pwm0 to the PWM_pin.  This generates
; an analog voltage at PWM_pin after filtering
;*********************************************************************
               bank    PWM_bank
               add     pwm0_acc,pwm0  ; add the PWM output to the acc
               snc
               jmp     :carry         ; if there was no carry, then clear
                                      ; the PWM_pin
               clrb    PWM_pin
               jmp     PWM_out
:carry
               setb    PWM_pin        ; otherwise set the PWM_pin
PWM_out
;*********************************************************************
               jnb     dtmf_gen_en,sine_gen_out
               call    @sine_generator1
sine_gen_out


;*********************************************************************
do_timers
; The timer will tick at the interrupt rate (3.26us for 50 MHz.)  To set up
; the timers, move in FFFFh - (value that corresponds to the time.)
; Example:
; for 1ms = 1ms/3.26us = 306 dec = 132 hex so move in $FFFF - $0132 =
; $FECD
;*********************************************************************

               bank    timers         ; Switch to the timer bank
               mov     w,#1
               add     timer_l,w      ; add 1 to timer_l
               jnc     :timer_out     ; if it's not zero, then
               add     timer_h,w      ; don't increment timer_h
               snc
               setb   timer_flag
:timer_out
;*********************************************************************
:ISR_DONE
; This is the end of the interrupt service routine.
; Now load 163 into w and
; perform a retiw to interrupt 163 cycles from the start of this one.
; (3.26us@50 MHz)
;*********************************************************************
               break
; interrupt 163 cycles after this interrupt
               mov    w,#-163
               retiw                 ; return from the interrupt
;*********************************************************************

start          bank   sin_gen_bank   ; Program starts here on power up

;*********************************************************************
; Initialize ports and registers
;*********************************************************************
; use these values for a wave which is 90 degrees out of phase.
               mov    curr_sin,#-4
               mov    sinvel,#-8
; use these values for a wave which is 90 degrees out of phase.
               mov    curr_sin2,#-4
               mov    sinvel2,#-8
               call   @disable_o

               mov    !option,#%00011111     ; enable wreg and rtcc interrupt
               mov    !rc,#%10111111

               mov    m,#$D                  ; make cmos-level
               mov    !rc,#%10111111
               mov    m,#$F

; load digits
               clr    i
digloop        call   getdigit
               mov    byte,w
               cje    byte,#$FF,done
               call   @load_frequencies      ; load the frequency registers
               call   @dial_it               ; dial the number for 60ms
; and return.
               inc    i
               mov    w,#20
               call   @delay_10n_ms
               jmp    digloop
done
               sleep

; get i'th digit to dial
```

```
getdigit        mov     w,i
                jmp     PC+W
                retw    1,8,8,8,5,1,2,1,0,2,4,$FF




org     $200                          ; Start this code on page 1
;*********************************************************************
;       Miscellaneous subroutines
;*********************************************************************
delay_10n_ms
; This subroutine delays 'w'*10 milliseconds.
; This subroutine uses the TEMP register
; INPUT         w       -       # of milliseconds to delay for.
; OUTPUT        Returns after n milliseconds.
;*********************************************************************
                mov     temp,w
                bank    timers
:loop           clrb    timer_flag      ; This loop delays for 10ms
                mov     timer_h,#$0f4
                mov     timer_l,#$004
                jnb     timer_flag,$
                dec     temp            ; do it w-1 times.
                jnz     :loop
                clrb    timer_flag
                retp

;*********************************************************************
; Subroutine - Disable the outputs
; Load DC value into PWM and disable the output switch.
;*********************************************************************
disable_o
                bank    PWM_bank        ; input mode.
                mov     pwm0,#128       ; put 2.5V DC on PWM output pin
                retp

                org     $400            ; This table is on page 2.
; DTMF tone table
_0_             dw      f941_h,f941_l,f1336_h,f1336_l
_1_             dw      f697_h,f697_l,f1209_h,f1209_l
_2_             dw      f697_h,f697_l,f1336_h,f1336_l
_3_             dw      f697_h,f697_l,f1477_h,f1477_l
_4_             dw      f770_h,f770_l,f1209_h,f1209_l
_5_             dw      f770_h,f770_l,f1336_h,f1336_l
_6_             dw      f770_h,f770_l,f1477_h,f1477_l
_7_             dw      f852_h,f852_l,f1209_h,f1209_l
_8_             dw      f852_h,f852_l,f1336_h,f1336_l
_9_             dw      f852_h,f852_l,f1477_h,f1477_l
_star_          dw      f941_h,f941_l,f1209_h,f1209_l
_pound_         dw      f941_h,f941_l,f1477_h,f1477_l
```

```
               org     $600           ; These subroutines are on page 3.
;************************************************************************
; DTMF transmit functions/subroutines
;************************************************************************
;************************************************************************
load_frequencies
; This subroutine loads the frequencies using a table lookup approach.
; The index into the table is passed in the byte register.  The DTMF table
; must be in the range of $400 to $500.
;************************************************************************
               cje     byte,#$0FF,:end_load_it
               clc
               rl      byte
               rl      byte           ; multiply byte by 4 to get offset
               add     byte,#_0_      ; add in the offset of the first digit
               mov     temp,#4
               mov     fsr,#freq_count_high

:dtmf_load_loop
               mov     m,#4           ; mov 4 to m (table is in $400)
               mov     w,byte
               IREAD                  ; get the value from the table
               bank    sin_gen_bank   ; and load it into the frequency
               mov     indf,w         ; register
               inc     byte
               inc     fsr
               decsz   temp
               jmp     :dtmf_load_loop        ; when all 4 values have
                                              ; been loaded,
:end_load_it  retp                            ; return
;************************************************************************
dial_it ; This subroutine puts out whatever frequencies were loaded
        ; for 1000ms, and then stops outputting the frequencies.
;************************************************************************
               cje     byte,#$0FF,end_dial_it
               bank    sin_gen_bank
; use these values to start the wave at close to zero crossing.
               mov     curr_sin,#-4
               mov     sinvel,#-8
; use these values to start the wave at close to zero crossing.
               mov     curr_sin2,#-4
               mov     sinvel2,#-8
enable_o                                      ; enable the output
               mov     w,#3
               call    @delay_10n_ms          ; delay 30ms
               setb    dtmf_gen_en
               mov     w,#10
               call    @delay_10n_ms          ; delay 100ms
               clrb    dtmf_gen_en
               call    @disable_o             ; now disable the outputs
```

```
end_dial_it    retp
;*************************************************************************
sine_generator1                      ;(Part of interrupt service routine)
; This routine generates a synthetic sine wave with values ranging
; from -32 to 32.  Frequency is specified by the counter.  To set the
; frequency, put this value into the 16-bit freq_count register:
; freq_count = FREQUENCY * 6.83671552 (@50 MHz)
;*************************************************************************
               bank    sin_gen_bank
; advance sine at frequency
               add     freq_acc_low,freq_count_low;2
               jnc     :no_carry              ;2,4    ; if lower byte rolls over
               inc     freq_acc_high          ; carry over to upper byte
               jnz     :no_carry              ; if carry causes roll-over
; then add freq counter to accumulator (which should be zero,
; so move will work)
               mov     freq_acc_high,freq_count_high
                                                      ; and update sine wave
               jmp     :change_sin
:no_carry
; add the upper bytes of the accumulators
               add     freq_acc_high,freq_count_high
               jnc     :no_change
:change_sin

               mov     w,++sinvel             ;1      ; if the sine wave
               sb      curr_sin.7             ;1      ; is positive, decelerate
               mov     w,--sinvel             ;1      ; it.  otherwise,
                                                      ; accelerate it.
               mov     sinvel,w               ;1
               add     curr_sin,w             ;1      ; add the velocity to sin


:no_change

;*************************************************************************
sine_generator2                      ;(Part of interrupt service routine)
; This routine generates a synthetic sine wave with values ranging
; from -32 to 32.  Frequency is specified by the counter.  To set the
; frequency, put this value into the 16-bit freq_count register:
; freq_count = FREQUENCY * 6.83671552 (@50 MHz)
;*************************************************************************

;advance sine at frequency
               add     freq_acc_low2,freq_count_low2 ;2
               jnc     :no_carry                     ;2,4    ; if lower byte
                                                      ; rolls over
               inc     freq_acc_high2                ; carry over to upper byte
               jnz     :no_carry                     ; if carry causes
                                                      ; roll-over
; then add freq counter to accumulator (which should be zero,
               mov     freq_acc_high2,freq_count_high2
                                                      ; so move will work)
                                                      ; and update sine wave
               jmp     :change_sin
:no_carry
; add the upper bytes of the accumulators

               add     freq_acc_high2,freq_count_high2
               jnc     :no_change
:change_sin

               mov     w,++sinvel2    ;1              ; if the sine wave
               sb      curr_sin2.7    ;1              ; is positive,
                                                      ; decelerate it
               mov     w,--sinvel2    ;1              ; it.  Otherwise,
                                                      ; accelerate it.
               mov     sinvel2,w      ;1
               add     curr_sin2,w    ;1              ; add the velocity to sin


:no_change
               mov     pwm0,curr_sin2                 ; mov sin2 into pwm0
               mov     sin2_temp,w
; mov the high_frequency sin wave's current value
               clc                                    ; into a temporary
                                                      ; register

; divide temporary register by four by shifting right
               snb     sin2_temp.7
stc                                           ; (for result = (0.25)(sin2))
               rr      sin2_temp
               clc
               snb     sin2_temp.7
               stc
               mov     w,>>sin2_temp
; (1.25)(sin2) = sin2 + (0.25)(sin2)
               add     pwm0,w
; add the value of SIN into the PWM output
               add     pwm0,curr_sin
; for result = pwm0 = 1.25*sin2 + 1*sin
; put pwm0 in the middle of the output range (get rid of negative values)
               add     pwm0,#128
               retp                           ; return with page bits intact
```

Unit 9: Simple Hardware I/O Enhancements

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 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. The next few figures show three circuits that can be used to significantly boost the SX chip’s output capacity. Figure 9-1 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.

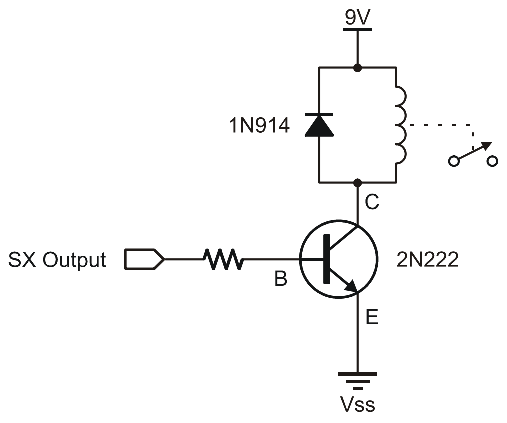


Figure 9-1: Switching a High-Current Relay

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 0.7 V, the transistor switches off. Although a very small amount of current is still conducted, it is insignificant as far as the coil is concerned.

Notice the diode across the relay coil in Figure 9-1. 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 14 V supply to be switched on and off. The circuit in Figure 9-2 can be used for his application.

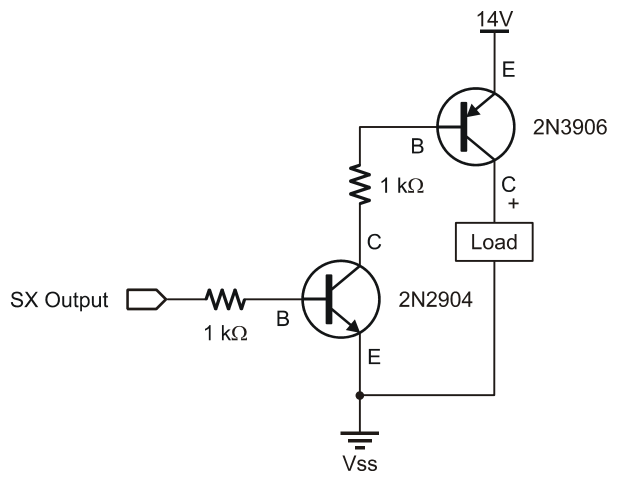


Figure 9-2: Circuit for Switching a Positive Voltage.

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. When the NPN transistor does not conduct, minimal base current will flow in the PNP transistor's base circuit, and therefore, virtually no collector current will flow (that is, the load will not receive current).

The circuit in Figure 9-3 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 Ω, when switched on.

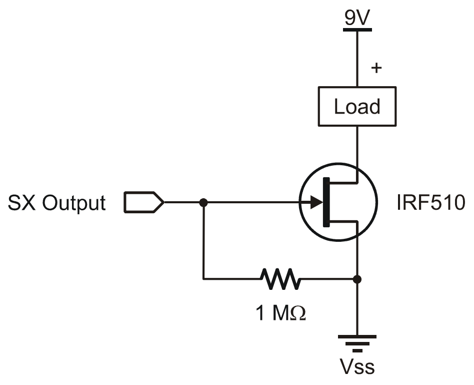


Figure 9-3: Using a MOSFET

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 you'll read more about them in Units 11, 13, and 14. 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 7.5 V target voltage to 2.5 V. A voltage divider is shown in Figure 9-4, and the voltage divider equation is given by:

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

Resistor values should be selected to make V_o = 2.5 V when Vi = 7.5 V. A 10 kΩ 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 resistor values could also be increased to 100 kΩ and 200 kΩ to reduce current consumption to 30 μA.

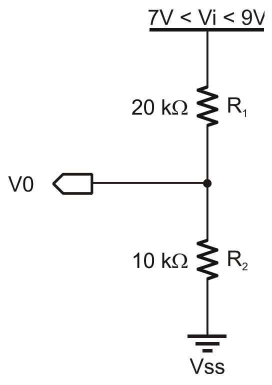


Figure 9-4: Detecting a Low Battery

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 SX28 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 (now Renesas) HD44780 or a compatible device. Larger LCDs use a 44780 plus some additional parts, but the programming turns out to be essentially the same.

LCD Hardware

The 14 pins on the LCD are likely arranged in the standard configuration given in Table 9-1.

Table 9-1: Pin Functions and Descriptions for Common LCDs with Hitachi or Compatible Driver

Pin	Function	Description	Pin	Function	Description
1	GND	Ground	8	DB1	Data Bit 1
2	+5	+ 5 V Power	9	DB2	Data Bit 2
3	C	Contrast voltage	10	DB3	Data Bit 1
4	RS	Reg. Select	11	DB4	Data Bit 4
5	R/W	Read/Write	12	DB5	Data Bit 5
6	E	Enable	13	DB6	Data Bit 6
7	DB0	Data Bit 0	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 9-5 shows a connection diagram for operating a 14 pin LCD in 4-bit mode.

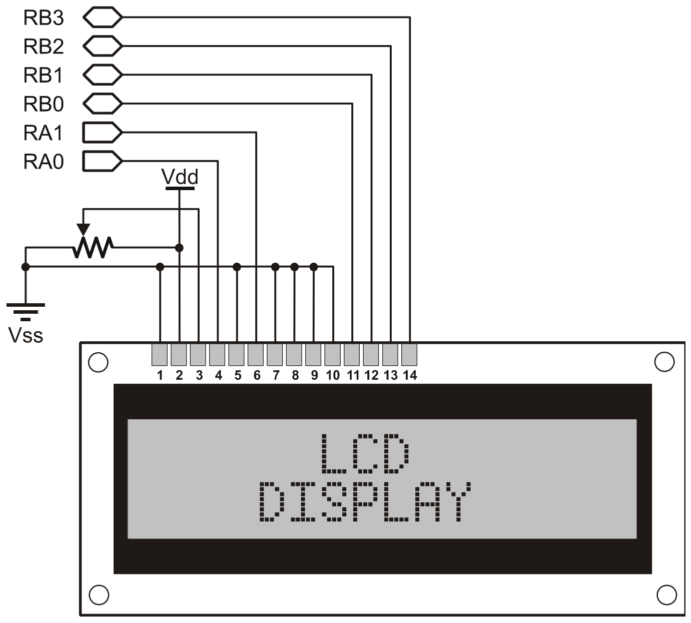


Figure 9-5: LCD Connection Diagram

The LCD 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, followed by the lower four bits. RS is brought low when sending command codes. Each 4-bit transfer occurs when the program pulses the E pin.

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: A very few LCDs require negative voltages to set the contrast. If you encounter one of these displays, it may appear dead until you provide a negative contrast voltage. Fortunately, these displays are not very common.

Program Listing – LCD Interface

The next program is for an LCD interface, using the techniques just discussed. The program displays a message you can change by changing the text in single quotes in the **msg** routine.

```
;=========================================================================
;Beginning Assembly Language for the SX Microcontroller
;Program 9.1
;4-bit LCD driver by Al Williams
;=========================================================================

               device  SX28L,turbo,stackx,optionx,oscxt1,bor42
               freq    4000000        ; Run at 4 MHz to simplify timing.
               reset   start          ; Go to 'start' on reset.

               org      $0c
dlyctr         ds      1              ; Main delay counter.
dlymult        ds      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.
```

9

```
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 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, writing to the entire port will arbitrarily wipe out any output bits used by the other part of the program. This would lead to spurious outputs each time the program sends information 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. For a 28-pin SX this monopolizes nearly a third of the 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 type 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:

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

The transmitter and receiver must also agree 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. (Real systems often sample several times during the interval to improve noise rejection, but that's not important to this discussion.) 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 +/- 12 V power supply, common in computers, but not so common is smaller electronic designs.

In many cases, the only reason to have +/-12 V is for RS-232. If this is true, 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 5 V 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, and 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 communications 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 the 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 0 V when off and 3 V 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.

Unit 10: A Software UART – The Transmitter

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 can 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 connecting to the TTL side of 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 when idle. Figure 10-1 is an RS-232 transmission of an ASCII "A" character (%01000001). Notice the bits are inverted and the least-significant bit is first.

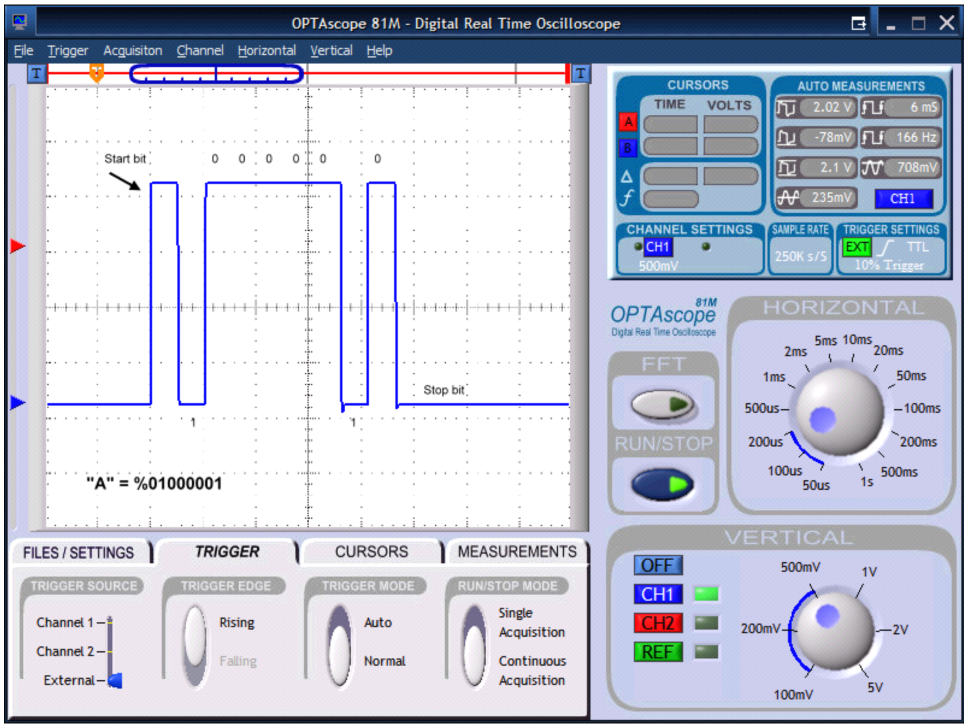


Figure 10-1: RS-232 Transmission

Creating the Code

The Parallax Web site contains several UART routines. Actually, one of these implements 8 19.2 K UARTs! Another example allows you to configure the UART to operate between 2400 and 230.4 K 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 at 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. Also, when receiving characters you'll need multiple interrupts per bit time anyway, as you'll see in a later unit.

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 50 Mhz, which corresponds to 1/50000000, or 20 ns per clock cycle. The ISR will execute every 163 clock cycles or 3.26 µs. Finally, the ISR only executes every 16 interrupts, so the code runs every 52.16 µs. The desired baud rate is 19200 bits per second, which is 1/19200 or 52.08 µs. The 52.16 µs 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 10 MHz instead of 50 MHz 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 in this case is 100 ns. The total bit time is about 104.2 µs. Dividing 104.2 µs by 100 ns 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 5 µs (handy for 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 5 µs is 105 µs, an 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 5 µs, the ISR will use up 4 µs, and the main code can execute for 1 µs.

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 7.5 µs interrupt time. With a divisor value of 14 this leads to a 105 µs 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. Remember that 21 cycles is the worst case. 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
```

Naturally 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
```

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    equu    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 that this does not perform the logic at run time. It makes the comparison during the assembly process. 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    var,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**).

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:

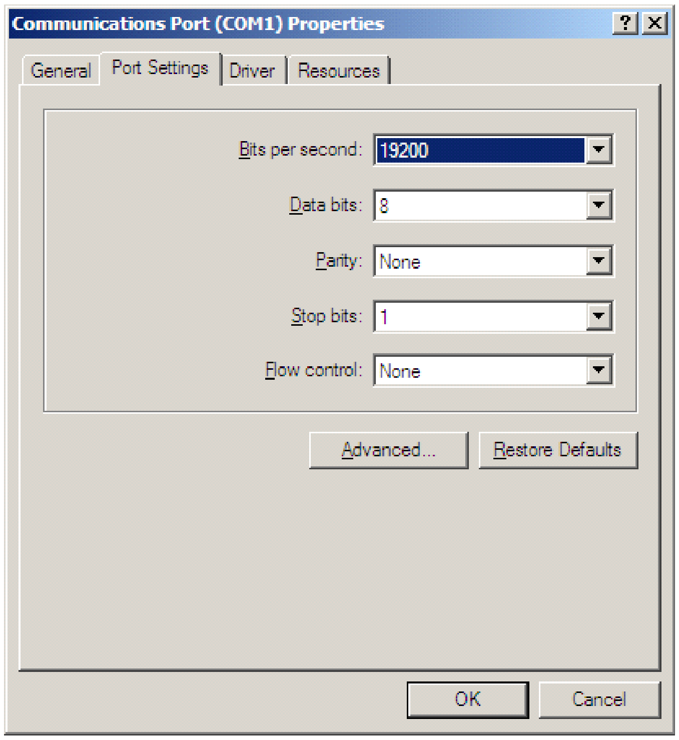
```
somevalue    =      100
```

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 many 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 10-2.



Bits per second: 19200
Data bits: 8
Parity: None
Stop bits: 1
Flow control: None

Figure 10-2: 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. A straight cable connects pin 2 on one side to pin 2 on the other side (and the same for pin 3). A crossed cable will connect pin 2 on one side with pin 3 on the other side (and vice versa).
- 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 5 V. 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, many 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.

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

```
;=======================================================================
;Beginning Assembly Language for the SX Microcontroller
;Program 10.1
;19.2K RS232 transmitter
;=======================================================================
               device  sx28l,oschs3
               device  turbo,stackx,optionx
               IRC_CAL IRC_SLOW
               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
```

10

```
                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 20 MHz 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 105.95 µs, 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 104 µs, 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 50 MHz clock):

```
baudrate      =      9600
       IF     baudrate = 19200
isrperiod     equ    -163
txdivisor     equ    16
       ENDIF

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

       .
       .
```

Unit 11: Analog Input

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

Figure 11-1 is the circuitry required to form the simple ADC:

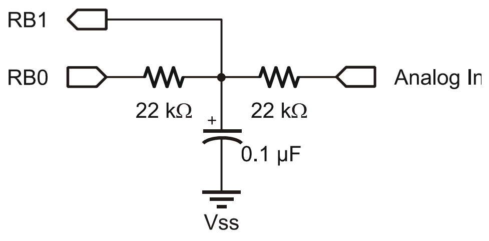


Figure 11-1: The Simple ADC Circuit

You can use a potentiometer to provide the analog input, or use a variable power supply. If you use a potentiometer, simply wire it as a voltage divider – place +5 V at one end, ground at the other end, and connect the center (the wiper) to the analog input pin. 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 2.5 V 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 2.5 V threshold. Along the way the program counts how often the capacitor has charged up past 2.5 V and required a discharge. After 255 cycles, this count will be proportional to the voltage (as a percentage of 5 V). So a 5 V input will read 255 counts. A 2.5 V input should read 128 counts.

Here is the basic logic written in pseudo code:

- Read the input bit
- Invert the input bit
- Write the inverted input to the output
- If the output was 0 (capacitor discharge), add 1 to the voltage count
- Add 1 to the cycle count
- If 256 cycles have elapsed (the cycle 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 0 V, 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 5 V, the SX will never successfully discharge the capacitor and will try on each cycle leading to a count of 255. If the input is 2.5 V, 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 (this is not the case on an SX48/52, which requires a $10 and $1F, respectively). 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.

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/decprevents 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:

- Can the routines share an interrupt period?
- Does either of the routines take a constant time to execute?
- Does one or more routines need a precise period?
- 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 300 µs and the other needs to execute every 500 µs. You might consider setting the interrupt period to 100 µs 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 200 µs. The first routine takes somewhere between 300 ns and 700 ns to execute. Then the second routine will not necessarily run every 200 µs.

As an example, try 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). Table 11-1 how your imaginary system would work:

Table 11-1: Interrupts

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 (not in seconds), 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 with 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.

If you want to document this hidden return, you can put the missing instructions in as a comment. For example, you might write:

```
        jmp     b       ; call b
        ;ret
```

Or, if you've deleted everything, you could write:

```
        ; call    b
        ;ret
b       mov     x,100.
```

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 between 0 and 3. 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 less than 255. 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, 11 kΩ – 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 or a 324) 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 many op-amp circuits require positive and negative voltages higher than the voltages they have to handle (for example, + and – 12 V supplies are common).

The Complete A/D Converter Code

```
;=======================================================================
;Beginning Assembly Language for the SX Microcontroller
;Program 11.1
;Simple A/D Converter
;=======================================================================
; Device
;
               device  sx28l,oschs3
               device  turbo,stackx,optionx
               IRC_CAL IRC_SLOW
               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

(All). 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 variable (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.

1. 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.

2. 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 5/256 V (roughly 19.5m V).

Unit 12: A Software UART – The Receiver

In Units 10 and 11, 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 104 µs. So if you sample an input every 104 µs, 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 too early or too late.

Ideally, you'd find the rising edge of the start bit and then delay 52 µs. This would be approximately in the center of the start bit. Now the code can safely sample every 104 µs (a total delay of 156 µs for the first bit) 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 104 µs pulse, you'll need to measure the input at least every 52 µs. Even this isn't enough if you are planning to delay 52 µs to center the timing. You might catch the center of the pulse and then skid past the end after the delay. To be safe, you should sample much faster, say 26 µs 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 3.26 µs. This is twice as fast as a 150 KBaud input signal and four times as fast as a 75 K Baud 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 50 MHz, then each interrupt cycle is 3.26 µs. For 9600 baud the bit period is about 104.2 µs. 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          equu   32
baud15        equ    48
```

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

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

You can do math like this as long as the computation 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  = 1    9200
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 since you have access to 16 registers in each bank). 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 code to place characters in the buffer. The changed program 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 in **get_byte** there is code that increments a pointer and masks it with $F (that is, it applies the logical **and** function to the pointer and $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** command every time you increment the pointer. 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 +/- 12 V 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 22 kΩ resistor between the RS-232 transmitter (pin 3 on a DB9 connector) and the SX pin will work fine.

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 Ω resistor) to ports RA0 and RA1. Use a 22 kΩ 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):

```
;======================================================================
;Beginning Assembly Language for the SX Microcontroller
;Program 12.1
;======================================================================
              device  sx28l,oschs3
              device  turbo,stackx,optionx
              IRC_CAL IRC_SLOW
```

```
               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:

```
;========================================================================
;Beginning Assembly Language for the SX Microcontroller
;Program 12.2
;========================================================================
               device  sx28l,oschs3
               device  turbo,stackx,optionx
               IRC_CAL IRC_SLOW
               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
```

Unit 13: Pulse I/O

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 11). 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 0 V across it. Then you apply 5 V 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 0 V. But the capacitor charges so the voltage increases until the final voltage is practically 5 V.

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 5 V supply will be:

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

So if R=100000 (100 kΩ) and C = .00001 Farads (10 μF), you'd find the voltage on the capacitor would look like Figure 13-1.

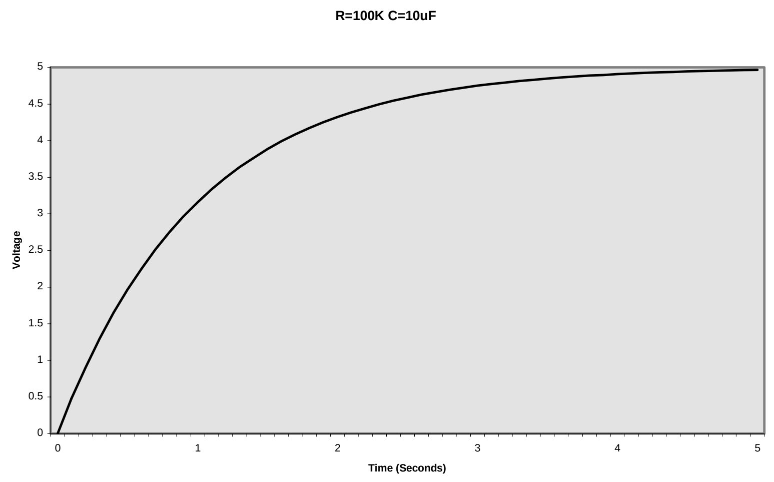


Figure 13-1: Capacitor Charging Curve

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 5 V so 63% is 3.15 V. The curve is just above 3 V 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 3.15 V 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 5 V 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 0.5 Vdd (nominally 2.5 V) is considered a logic 1. If the CMOS mode is not set, the threshold voltage is about 1.4 to 1.5 V. You can set each pin individually.

To set the threshold voltage for a port, you first set the **M** (mode) register to $D ($10 on the SX48/52). Then you can store configuration bits in the **!ra**, **!rb**, and **!rc** registers. A zero in these registers makes the corresponding bit use the CMOS threshold. A one sets the pin for 1.4 V (TTL) threshold. It is a good idea to set the **M** register back to the default value ($F, or $1F on the SX48/52) 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 5 V 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 (3.1 V with a 5 V supply). However, it will not read the pin as a logic 0, until the voltage falls below about 28% of Vdd (1.4 V). This electronic inertia is known as hysteresis. Consider Table 13-1:

Table 13-1: Hysteresis Example

Time	Input voltage	Input state
0	0.0 V	0
1	3.0 V	0
2	3.5 V	1
3	3.0 V	1
4	2.0 V	1
5	0.5 V	0
6	2.0 V	0
7	3.0 V	0
8	3.5 V	1

You can buy ICs that perform the Schmitt trigger function, but luckily, the SX already has these triggers built in if you want them. To set Schmitt trigger mode, you set the **M** register to $C ($1C on the SX48/52) 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 10 kΩ potentiometer and a .1 μF capacitor wired as shown in Figure 13-2.

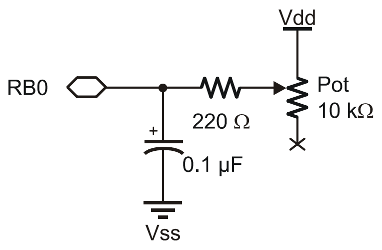


Figure 13-2: Reading a Potentiometer

The *RC* constant for this circuit is .001. That means that in 1 ms, the capacitor will charge to about 3.15 V. This is right around the threshold for a Schmitt trigger (3.1 V). 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 2 ms.

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    oop
```

You'll find that the total execution time per loop is 5 clock cycles. At 50 MHz that is only 100 ns per count. You have to count to 20,000 to measure 2 ms. 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 160 ns. When **count0** overflows, the code increments **count1**. This forms a 16-bit counter.

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 3.26 µs. A 2 ms 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:

Change **RB.0** to an output and pull it low
Pause a few ms to allow the capacitor to fully discharge
Restore **RB.0** to an input
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**. The **pausems** routine delays about 1 ms. 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 3.26 µs. To pause 1 ms (1000 µs), 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 1 ms 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 radio control hobbies) 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 ICs that convert voltage to frequency. 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 10 kΩ potentiometer and LED as in Figure 13-3. 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.

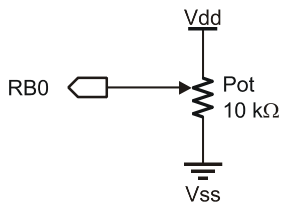


Figure 13-3: Threshold Test Circuit

2. Build the circuit shown in Figure 13-2. 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:

```
;=========================================================================
;Beginning Assembly Language for the SX Microcontroller
;Program 13.1
;=========================================================================

               device  sx28l,oschs3
               device  turbo,stackx,optionx
               IRC_CAL IRC_SLOW
               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:

```
;=========================================================================
;Beginning Assembly Language for the SX Microcontroller
;Program 13.2
;=========================================================================

               device  sx28l,oschs3
               device  turbo,stackx,optionx
               IRC_CAL IRC_SLOW
               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



               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

               watch   dly,16,uhex



               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
```

Unit 14: Pusle Width Modulation

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 5 V – 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 0 V. Similarly, if you bring the output high, the voltage will charge the capacitor and you will soon have 5 V across the capacitor.

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

In general, the output voltage will be 5 V 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 100 µs and 400 µs, the output would still be 1 V. The percentage of time the signal is high is known as its duty cycle. In this example, the duty cycle is 20%.

Figure 14-1 shows a practical circuit. The resistor prevents excessive current draw from the SX.

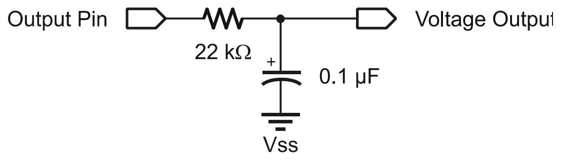


Figure 14-1: PWM Output Circuit

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 2 µs and you wanted a 50% duty cycle, you'd turn the output high for 50 time periods (100 µs) 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 2 µs timebase would have one 2 µs high followed by a 2 µs low. A 33% duty cycle would be 2 µs on and 4 µs 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 2 µs 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:

- Set **pwm_acc** equal to **pwm_acc** plus **pwm**
- If a carry results from the addition, set the output bit
- 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 the values in Table 14-1:

Table 14-1: PWM Accumulator

Time µS	pwm(duty)=$FF		pwm(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 2 µs on and 511 µs 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 1 µs. 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 400 ns!

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 5 mV). 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. Figure 14-2 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?

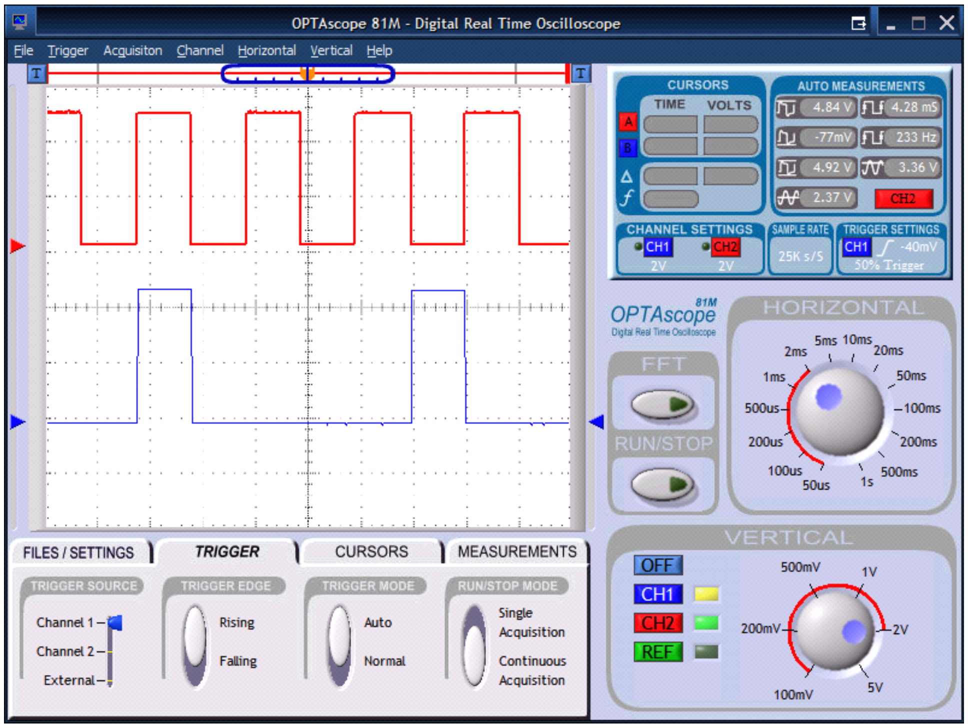


Figure 14-2: Two PWM Output Signals

2. Set up a PWM circuit as shown and create code that varies the pwm duty cycle by 1 bit about every 250 ms (or more). Using a voltmeter (or even better, an oscilloscope) verify that the change in voltage is near the expected 19.5 mV. 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 in Figure 14-3. 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.

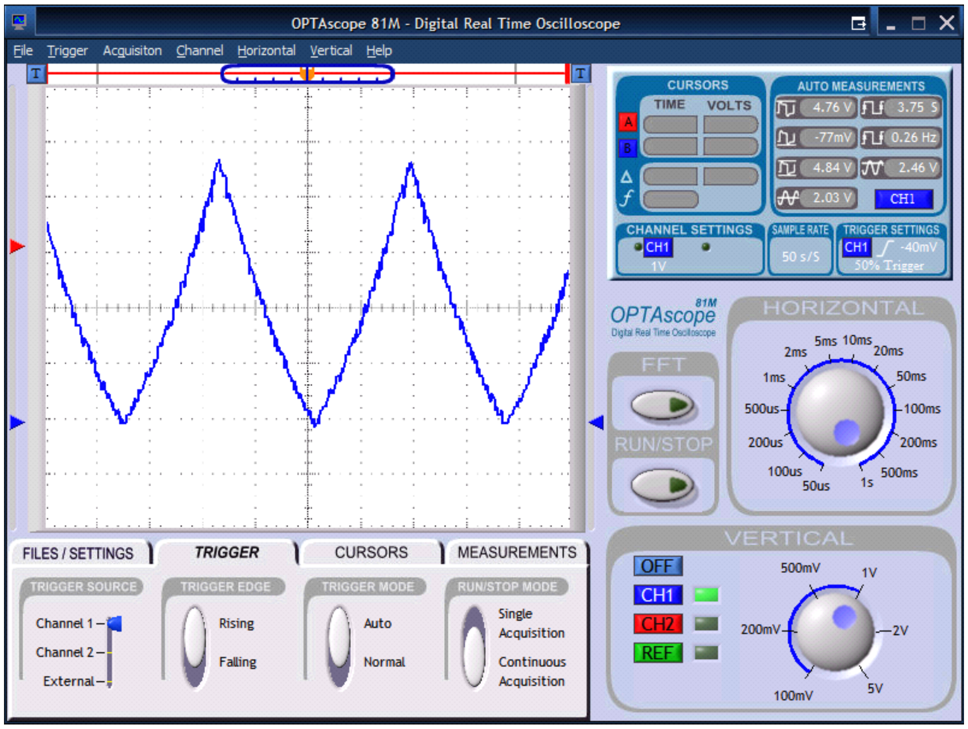


Figure 14-3: Triangle Wave

Answers

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

2. Here is a possible 8 bit solution:

```
;=========================================================================
;Beginning Assembly Language for the SX Microcontroller
;Program 14.1
;=========================================================================

               device  sx28l,oschs3
               device  turbo,stackx,optionx
               IRC_CAL IRC_SLOW
               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    ainloop
```

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

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
```

14

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 15: A Practical Design - The SSIB

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 BASIC Stamp modules, for example) receive serial data from a PC or other device.

Parallax's BASIC Stamp modules have a microcontroller that you program using BASIC. These BASIC Stamp modules are perfect for quick and simple projects. Although BASIC Stamp modules excel at many jobs, they are inherently single-tasking. This single-tasking philosophy makes programming simpler, but it makes serial input tricky.

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

To ameliorate this limitation, BASIC Stamp modules can employ a handshaking signal. This output line signals the transmitting device when the BASIC 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 BASIC Stamp module. The buffer would hold any incoming data until the BASIC 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 SX28 – the project doesn't even use all the pins available, and just ignore the extra pins.

With any project, you should start with a design. Figure 15-1 shows the pin out for the buffer device (the BASIC 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 BASIC Stamp will empty the buffer faster than the device will fill it, you can ignore these handshaking lines.

On the BASIC Stamp side the SSIB uses 3 pins. One pin receives data from the SSIB. The other two pins act as handshake lines. If the BASIC Stamp asserts CHANA, the SSIB sends data from channel A to the BASIC Stamp. CHANB selects data from the B channel. If neither line is active the SSIB sends no data to the BASIC 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 15-1 for the available configuration options - you can change several parameters here including the polarity of each port.

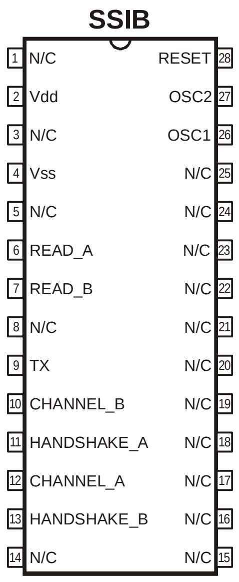


Figure 15-1: The SSIB Pin-Out

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 50 MHz, this is overkill for this application. Even at 10 MHz, 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 when in turbo mode. So an SX running at 10 MHz is comparable to some other processors running at 40 MHz! A processor that divides by 4 would have to run at 200 MHz to match a 50 MHz SX. Almost all of the code executes in response to a high-speed periodic interrupt that occurs every 13 µs.

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.

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 BASIC Stamp.

Table 15-1: SSIB Configuration

Parameter	Description	Default Value
XBAUDRATE	Baud rate to BASIC Stamp	19200
BAUDRATE_A	Baud rate to device A	9600
BAUDRATE_B	Baud rate to device B	9600
INVSEND	Use inverted mode to BASIC 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

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 15 -2.
Figure 15-2 shows a sample test circuit. In this schematic, the BASIC Stamp at IC1 is receiving data from the BASIC Stamp at IC2 (which stands in for two external devices). IC3 is the SSIB.

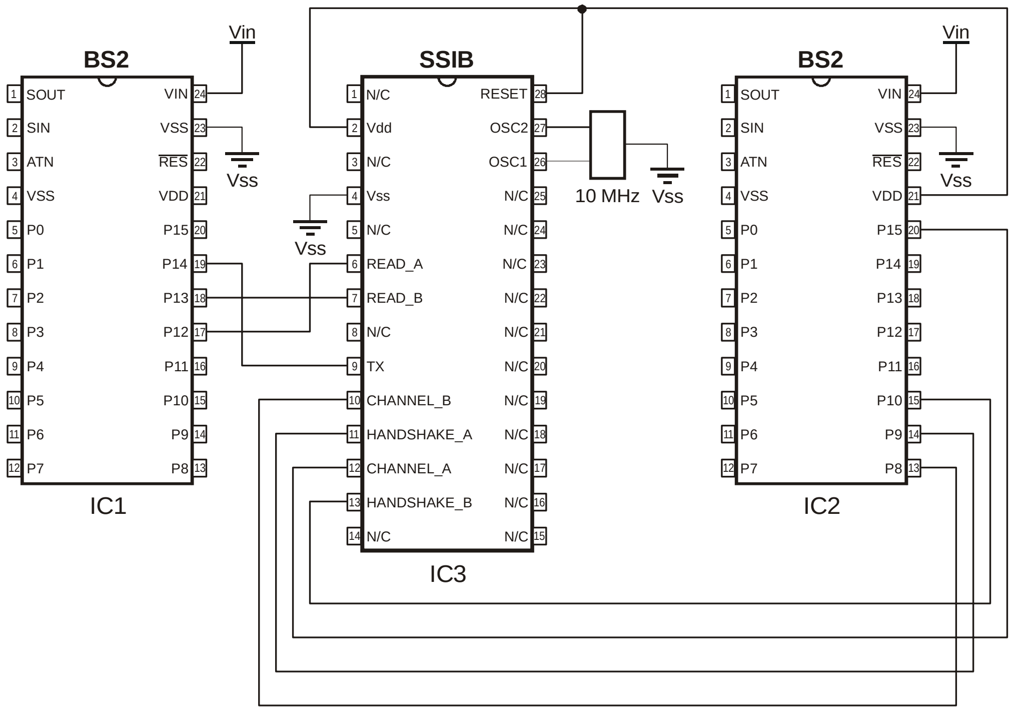


Figure 15-2: Test Circuit for the SSIB

The device connected to the SSIB's RES1 and RES2 terminals is a 10 MHz 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 10 MHz clock automatically (it senses the **FREQ** directive in the program). You could also use a 10 MHz 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 BASIC Stamp reads data from the same pin regardless of which channel it wants to read. However, the BASIC Stamp program'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 BASIC Stamp reads the data from pin 14.

The simulator BASIC Stamp at IC2 (see the listings) just writes bytes out of each serial port periodically. Of course, the two BASIC Stamp modules won't be synchronized, so only the buffer allows this arrangement to work. If you set the first BASIC Stamp module 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.

Table 15 -2: SSIB Pinout

Pin	Name	Function
1	N/C	Not connected
2	Vdd	+5 V
3	N/C	Not connected
4	Vss	Ground
5	N/C	Not connected
6	READ_A	Signal to read from channel A
7	READ_B	Signal to read from channel B
8	N/C	Not connected
9	TX	Transmit data to BASIC Stamp
10	CHANNEL_B	Input for channel B
11	HANDSHAKE_A	Optional handshake for device A
12	CHANNEL_A	Input for channel A
13	HANDSHAKE_B	Optional handshake for device B
14-25	N/C	Not connected
26	OSC2	Connection to 10 MHz resonator
27	OSC2	Connection to 10 MHz resonator
28	RESET	Pull low to reset; high for normal operation

About Inverted Mode

The BASIC 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 BASIC Stamp program or other device wakes up and explicitly inhibits transmission. This can cause problems when the BASIC Stamp misses some characters at the beginning or receives an erroneous byte right after resetting. A sleeping BASIC Stamp module 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 BASIC Stamp pin to reset the SSIB (by pulling **RESET** low) after the BASIC 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 104 µs. You need to interrupt at least 4 times faster (26 µs). 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 10 MHz. This causes the RTCC register to increment every 100 ns. Causing an interrupt every 130 cycles makes the sampling rate 100 ns*130 = 13 µs; fast enough to 4x oversample a 19200 baud rate signal (52 µs 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 13 µs per cycle, this works out to 104 µs 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 25 MHz clock. This makes each RTCC count worth 40 ns (1/25000000). If you want to sample a 9600 baud signal 8 times per bit, you need 13 µs interrupts (as calculated above; this is the same as 13000 ns). 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 3.3 µs which requires an interrupt period of 3300/40 or about 82.

So to use a 25 MHz clock, you can set the interrupt period to 82 and the baud rate number to 5 (2 to the 5th power is 32). The actual time will be 40 * 82 * 32 = 104960 ns or 104.96 µs. 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 BASIC 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 and skills 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

```
;=======================================================================
;Beginning Assembly Language for the SX Microcontroller
;Program 15.1
;SSIB - by Al Williams, AWC http://www.al-williams.com/awce
;v2.0
;=======================================================================

               device  sx28l,oschs3
               device  turbo,stackx,optionx
               IRC_CAL IRC_SLOW
               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
; BASIC Stamp wakes up which may cause you problems

INVSEND        EQU     0     ; inverted/true to BASIC 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
```

15

```
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?
```

15

```
               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
               mov     !rc,#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

```
' Beginning Assembly Language for the SX Microcontroller
' TestSSIB.bs2
' BASIC Stamp program to test SSIB

'{$STAMP BS2}
'{$PBASIC 2.5}


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

```
' Beginning Assembly Language for the SX Microcontroller
' DataStreamForSSIB.bs2

' This BASIC Stamp program just writes out two
' data streams to test the SSIB

'{$STAMP BS2}
'{$PBASIC 2.5}


W1=0
W2=$FFFF

DO
  SEROUT 15\9,84,[DEC W1,","]
  SEROUT 8\10,84,[DEC W2,","]
  W1=W1+1
  W2=W2-1
  PAUSE 5
LOOP
```

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, 100 µS and the ISR requires 80 µs this leaves only 20 µs for the remainder of the program.

Of course, with the 28-pin device, there are enough pins for more ports. 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.

Appendix A: Instruction Summary

Processor Control

Instruction	Words	Turbo Cycles	Description
BANK x	1	1	Sets current register bank
MODE x	1	1	Sets I/O mode
NOP	1	1	No operation
PAGE	1	1	Sets current code page
SLEEP	1	1	Puts processor in low power sleep mode

Flow Control

Instruction	Words	Turbo Cycles	Description
CALL	1	3	Call subroutine
CJA	4	4,6	Compare jump above
CJAE	4	4,6	Compare jump above or equal
CJB	4	4,6	Compare jump below
CJBE	4	4,6	Compare jump below or equal
CJE	4	4,6	Compare jump equal
CJNE	4	4,6	Compare jump not equal
CSA	3	3,4	Compare skip above
CSAE	3	3,4	Compare skip above or equal
CSB	3	3,4	Compare skip below
CSBE	3	3,4	Compare skip below or equal
CSE	3	3,4	Compare skip equal
CSNE	3	3,4	Compare skip not equal
DECSZ	1	1,2	Decrement skip zero
DJNZ	2	2,4	Decrement jump not zero
INCSZ	1	1,2	Increment skip zero
IJNZ	2	2,4	Increment jump not zero
JB	2	2,4	Jump if bit set
JC	2	2,4	Jump if carry set
JMP	1	3	Jump
JNB	2	2,4	Jump if bit not set
JNC	2	2,4	Jump if no carry
JNZ	2	2,4	Jump if no zero
JZ	2	2,4	Jump if zero
MOVSZ	1	1,2	Move (with optional inc/dec) skip on zero
RET	1	3	Return from subroutine
RETP	1	3	Return across page
RETW	1	3	Return literal
SKIP	1	2	Skip next instruction
SNB	1	1,2	Skip if bit clear
SNC	1	1,2	Skip if no carry
SNZ	1	1,2	Skip if not zero

Math and Logic

Instruction	Words	Turbo Cycles	Description
ADD	1	1	Add (register + W or W + register)
ADD	2	2	Add (register + register or literal)
ADDB	2	2	Add bit
AND	1	1	And (register and W, W and register, W and literal)
AND	2	2	And (register and literal or register and register)
DEC	1	1	Decrement
INC	1	1	Increment
NOT	1	1	Invert
OR	1	1	Or (register and W or W and register or W and literal)
RL	1	1	Rotate left
RR	1	1	Rotate right
SUB	1	1	Subtract W from register
SUB	2	2	Subtract register from register or literal from register
XOR	1	1	Exclusive Or register and W or W and register
XOR	2	2	Exclusive Or register and register or register and literal

Interrupt Handling

Instruction	Words	Turbo Cycles	Description
RETI	1	3	Return from interrupt
RETIW	1	3	Return from interrupt and add W to rtcc

Bit Manipulation

Instruction	Words	Turbo Cycles	Description
CLC	1	1	Clear carry
CLRB	1	1	Clear bit
CLZ	1	1	Clear zero
MOVB	4	4	Move bit
SETB	1	1	Set bit
STC	1	1	Set carry
STZ	1	1	Set zero

Move/Clear/Test

Instruction	Words	Turbo Cycles	Description
CLR	1	1	Clear register, W, or WDT
MOV	1	1	Move W to register, register to W, literal to W
MOV	2	2	Move register to register or literal to register
TEST	1	1	Test W or register, set flags

Miscellaneous

Instruction	Words	Turbo Cycles	Description
IREAD	1	4	Reads program memory
LCALL	1-4	3-6	Obsolete
LJMP	1-4	3-6	Obsolete
LSET	0-3	0-3	Obsolete

Appendix B: Hardware

The projects in this tutorial are simple to build using common components. For the maximum flexibility, you'll want to use a solderless breadboard. If you use the Parallax SX-Tech board you can simply connect the circuits to the integrated breadboard.

You can also use your own breadboard if you like. The SX chip simply requires a regulated 5 volt supply (a bench supply will work fine) and a connection to the SX-Key programmer. If you are using an SX-Blitz, or you want to operate the circuit without the SX-Key, you'll also need a 50 MHz ceramic resonator (Murata CST50.00MXW040 or equivalent). Some of the circuits use slower clocks, and you'd need a resonator, crystal, or external oscillator if you didn't want to change the speed of the example circuit.

To successfully complete the tutorial exercises, you only need a few common parts:

- LEDs (or 5 V LEDs with integrated resistors)
- 470 Ω resistors (if not using 5 V LEDs)
- Push button switches
- Non-critical pull up resistors (10 kΩ to 22 kΩ, 1/4W or 1/8W)
- A piezo electric speaker

Common Circuit

All the circuits require the SX to be connected to the programmer and the chip's support circuitry. Again, if you are using an SX-Tech board this is already done. If you are using the SX-Key, you only need to connect the chip to 5 V, ground, and the SX-Key. You can use an existing 5 V power supply if you have one (make sure it is regulated). If you want to build a simple 5 V supply, look at Figure B-1. This supply will handle about 100mA as shown, or can handle over 1A if you use a 7805 with a heat sink in place of the 78L05 specified. You can use an ordinary wall transformer to supply the unregulated DC input.

To ensure proper operation, you should also connect the MCLR pin to 5 V either directly or through a pull up resistor. If you use a pull up resistor you'll be able to short the MCLR pin to ground to reset the processor. For the ultimate convenience you could use a push button switch to make the ground connection.

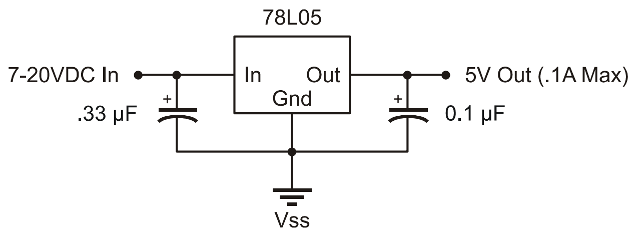


Figure B-1: A Simple 5 V Supply

To connect the programmer, you can use pins with .1 inch spacing. You usually buy these in strips that you can snap to the correct length with a pair of pliers or even your fingers. Insert one end into your breadboard and the SX-Key (or SX-Blitz) will plug into the other side. If one side of the pins is too short, you can usually slide the plastic insulator with a pair of pliers so that the pins on each side are of equal length. Table B-1 shows the pin connections necessary.

Table B-1: SX28 Pin Connections

5 V	Ground	OSC1	OSC2	MCLR
15,16	5,6	18	17	4

I/O Circuits

Most of the projects in the tutorial require some input or output. The I/O usually takes the form of an LED, a push button, both an LED and a push button, or a piezo speaker. Figure B-2 shows the common LED hookup. If you are using 5 V LEDs, you don't need the resistor as it is built into the LED. Notice that the LED is polarized; refer to the LEDs specifications to identify which lead is which. With the LED wired as shown, you must bring the SX pin low to light the LED.

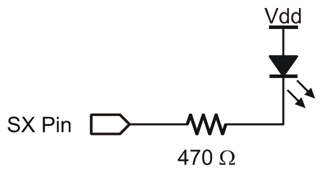


Figure B-2: An LED Circuit

In Unit 5, some exercises use a push button and a piezo speaker for I/O as in Figure B-3. The 10 kΩ resistor's value is not overly critical. Anything from 10 kΩ to 22 kΩ (or even more) should work fine. If a project calls for more switches, you can duplicate the switch portion of the circuit for other pins. Just use a pull up resistor on the pin and connect the switch to ground.

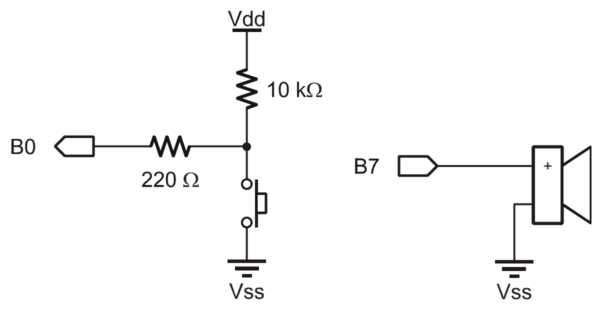


Figure B-3: A Speaker and Switch Circuit

Don’t connect an ordinary speaker directly to the SX pin as the load presented by such a speaker may damage the SX chip. Most ICs, including the SX, can directly drive a piezo speaker.

Final Projects

The first project is a TouchTone-style phone dialer. For demonstration purposes, you can hear the tones in a piezo speaker (although they may be quite low – you may have to put your ear right up to the speaker). If you want to really dial a phone, you’ll need two things: a filter and an amplifier.

The Ubicom notes on the DTMF generation VP specifies the component values for the low pass filter. This filter prevents high-frequency noise (an unavoidable byproduct of using PWM to generate tones) from entering the phone lines. Connect a 620 Ω resistor to the SX output pin and a .22 μF capacitor from the other side of the resistor to ground (the Ubicom data calls for 600 Ω resistor and .2 μF capacitors, but these values are close enough and easy to obtain). This will make the tones even weaker than before, however. Some sort of amplification is necessary if you plan to feed the tones into the phone. You can use any sort of amplified speaker, signal tracer, or build a small amplifier from an LM386 chip (see Figure B-4) and drive an ordinary 8 Ω speaker.

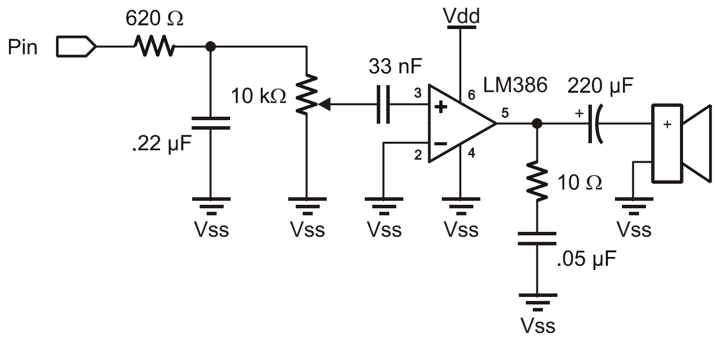


Figure B-4 : A Phone Dialer

If you wish to perform these next projects, you'll need a parallel LCD, a 10 kΩ potentiometer, and a way to connect a PC serial port to a BASIC Stamp module. One way to do this is with a DB9 cable that has a female end to connect to the PC and bare wires on the other end. You can also solder a DB9 connector to wires and plug a standard cable into it. In either case, the wires would plug into your solderless breadboard.

If you want to experiment with proper RS232 communications – which is usually not strictly necessary – you also need a MAX232 (along with the associated capacitors) or a MAX233 (which requires no capacitors). The MAX232A requires 4 or 5 0.1 µF capacitors, while the regular MAX232 requires 4 or 5 1µF capacitors.

If you do wish to connect a MAX232, you need the circuit shown in Figure B-5. Note that the capacitor between Vcc (pin 16) and ground is a decoupling capacitor and may not be necessary if your 5 V power already contains decoupling capacitors to handle other circuitry. Obviously, if you aren't using polarized capacitors, you can disregard the plus signs on the schematic.

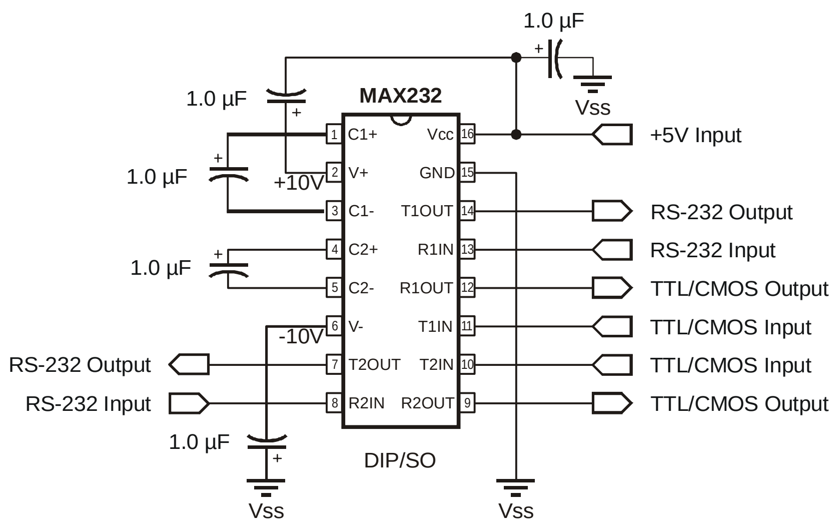


Figure B-5: MAX232 Circuit

The MAX233 doesn't require external capacitors to operate (although decoupling capacitors are always a good idea). However, the chip is a bit more expensive (usually more expensive than the 4 capacitors you can eliminate). You can see an example schematic in Figure B-6.

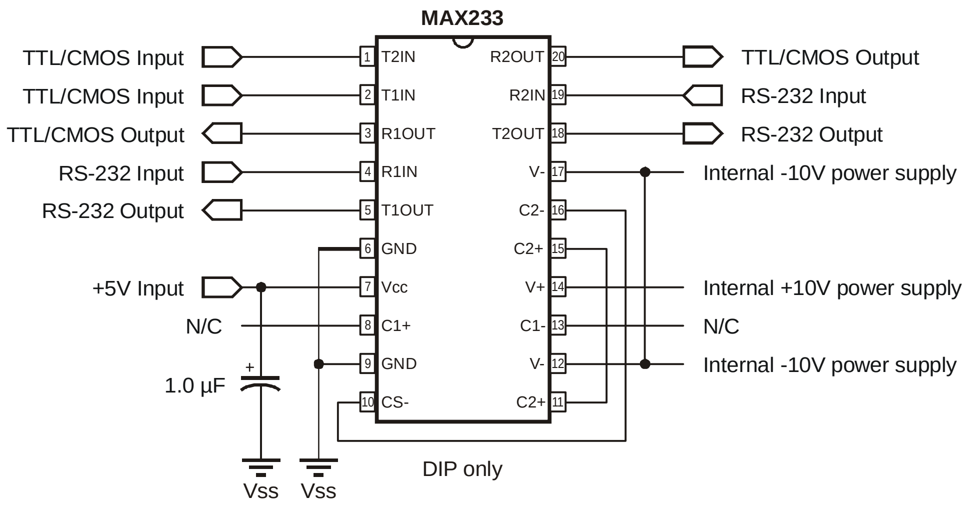


Figure B-6: MAX233 Circuit