Practical SX/B

(No assembly required!)

by

Jon Williams

PARALLAX INC
Publishing

WARRANTY

Parallax Inc. warrants its products against defects in materials and workmanship for a period of 90 days from receipt of product. If you discover a defect, Parallax Inc. will, at its option, repair or replace the merchandise, or refund the purchase price. Before returning the product to Parallax, call for a Return Merchandise Authorization (RMA) number. Write the RMA number on the outside of the box used to return the merchandise to Parallax. Please enclose the following along with the returned merchandise: your name, telephone number, shipping address, and a description of the problem. Parallax will return your product or its replacement using the same shipping method used to ship the product to Parallax.

14-DAY MONEY BACK GUARANTEE

If, within 14 days of having received your product, you find that it does not suit your needs, you may return it for a full refund. Parallax Inc. will refund the purchase price of the product, excluding shipping/handling costs. This guarantee is void if the product has been altered or damaged. See the Warranty section above for instructions on returning a product to Parallax.

COPYRIGHTS AND TRADEMARKS

DISCLAIMER OF LIABILITY

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

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. 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.

SUPPORTED HARDWARE, FIRMWARE AND SOFTWARE

Hardware	SX-Key IDE Software	SX/B Language
SX28	Version 3.2.3	Version 1.51.03

Dedication

To my son, Sabu: It is my great pleasure to be your dad, and I hope I make you proud in everything that I do.

Acknowledgements

First and foremost, there would be no SX/B without the tireless efforts of Ken Gracey. When Ubicom wanted to abandon their SX microcontroller line, Ken stepped up and said, "Fine, we'll take it." Since that time Ken has guided Parallax to provide more and better low-cost (and no-cost!) tools for those of us that love working with the SX so much. Thanks, Ken, for your friendship and belief in me, and for all the great times we have working together.

The heart of SX/B beats in the chest of engineer, Terry Hitt (aka Bean). I'm pretty sure that Terry never expected his little compiler – which started as a personal project – to become such a tremendous success and be enjoyed by so many people. Thank you, Terry, for all your hard work and the uncountable hours you've poured into SX/B and supporting SX users. I am especially grateful for your infinite patience with my questions, comments, and suggestions as we've worked together to grow and improve SX/B.

There are two others that all of us in the SX community owe a debt of gratitude to: Peter Montgomery who does a great job maintaining the SX-Key IDE for Parallax, and Günther Daubach who created and maintains the free SXSim simulator program. Both Peter and Günther are active participants the Parallax user forums (forums.parallax.com) and are incredibly generous in sharing their knowledge of electronics and programming the SX. Thanks, gentlemen, I appreciate your friendship and all the things you've taught me.

Table of Contents

Preface

The back-story: It all started back in the summer of 2004 when Ken Gracey, my boss at the time, called and asked me what I thought about Parallax providing a free BASIC compiler for the SX microcontroller line. Of course I thought it was a fantastic idea – assembly language has not, in the past, come easily to me. Since my days with the Timex-Sinclair 1000 I just seem to live, breathe, and think in BASIC. This explains my affection for the BASIC Stamp microcontroller and why I've built so many personal and professional projects with it.

The third, and key, member of the original SX/B team is Terry Hitt. Terry is an electronics engineer and consultant who had started experimenting with compiler designs to improve his own work-flow for clients. The three of us knocked our heads together, designing SX/B to be very familiar to BASIC Stamp users, and at the same time provide a bridge to those that wanted to learn more about SX programming by seeing well-crafted assembly output.

I can tell you from being on the inside it was a lot of hard work. Ken provided Terry and me with the resources we needed, Terry wrote the compiler to conform to the syntax specifications we'd all agreed upon, and I spent my time testing, debugging, and creating the online help file.

The hard work paid off – handsomely – even if I do say so myself. We released the first version of SX/B in November of 2004 and it was an immediate hit. Why wouldn't it be? – by downloading the updated version of the SX-Key IDE Parallax customers now had a completely free BASIC compiler for their SX projects.

SX/B is definitely a case of getting more than you pay for. It is a quality product that in the rights hands (that would be yours) allows the programmer to be incredibly productive. I know this from first-hand experience.

Fast-forward to the summer of 2006: I am now an independent consultant living and working in Los Angeles when I get a call from a local company that needs someone to design and build a specialized piece of test equipment for solenoids. After meeting with the client we come to an agreement that the project should take about two weeks. Then it happened: the client's parting handshake lasted a little longer than it should have as he said to me, "Jon, do you think you could come up with something quick-and-dirty by Monday?" Those that know me know how much I loathe the phrase "quick-and-dirty" because it is my opinion that those projects are never quick and are always far dirtier than anyone imagines.

Well, I was wrong in this case. I told the client that I couldn't lay out a custom PCB or do the digital displays we'd discussed in that short amount of time time, but I might be able to come up with something that would allow him to get started. He agreed.

Thankfully, I had an SX28 Proto Board on my desk to which I added a dual-channel ADC, a couple potentiometers, and an L293D driver for the coils. Using straight SX/B I was able to write a program to read the pots and drive the coil at a user-settable frequency and duty cycle; a digital input even allowed for the device to reverse coil current on alternate cycles. It worked perfectly the first time we tested it, in fact, it worked so well that the rest of the contract was cancelled!

For me there are two lessons: 1) SX/B is a great tool and can help anyone produce quality embedded programs in very little time and, 2) I should collect more fees up front when I know I'm going to be using the SX and SX/B!

All kidding aside, SX/B is a wonderful tool for the embedded BASIC programmer at any level, and while this book's title is a bit on the cheeky side (re: No Assembly Required), what you'll learn in these pages is that SX/B can be as flexible as you need it to be, and that includes full interrupt support and the ability to embed assembly language should you choose – all this from a compiler that costs you nothing but a few minutes to download and install it.

I'm going to operate on the assumption that you're here deliberately, that you understand the impact microcontrollers have had on our world, and that you want to be a part of all that. Excellent, you've come to the right place. My most important admonition to you is that if you're serious about learning the SX, then spend time experimenting – don't wait for a "real" project to show up before you get your feet wet. I am quite certain than if you give it a chance, you'll find the SX and SX/B will bring you hours of fun and real productivity in your embedded control projects.

Part I:Getting Started

1: Introducing SX/B

If you're new to embedded programming, or your experience is limited to interpreter-based devices like the BASIC Stamp microcontroller, you may in fact be wondering what SX/B is, exactly. SX/B is a specialized version of the BASIC language *and* a compiler for the SX microcontroller line.

BASIC (*Beginner's All Purpose Instruction Code*) is a computer language that has been around for a long time – since 1964! Compilers have been around for a long time, too, you see, a compiler is a language translator. In the case of embedded controllers, the compiler output is usually the assembly language for the target device.

So, as suggested above, SX/B has a dual definition: 1) It is a variant of the popular BASIC programming language that has been designed for embedded applications running on the SX microcontroller and, 2) It is a compiler that translates high-level SX/B code to SX assembly code (SASM variant).

Perhaps you've heard the term *machine language;* this is what actually runs on the target device. *Assembly language* is a human-friendly version of machine language. In most cases there is a one-for-one relationship between assembly language and machine language, though there are a few cases where an assembly language mnemonic will produce more than one machine instruction.

Have a look at this bit of machine language:

```
0C01
01ED
```

Can you determine the purpose of this code? Don't worry, you're not alone, which is why assemblers – programs that translate assembly language to machine language – were developed.

Here's the same code in assembly language:

```
ADD $0D, #1
```

This is certainly more useful than the machine code version, yet not altogether obvious to the inexperienced programmer. In this example, 1 is added to the value currently stored at memory location $0D. Assemblers allow the programmer to alias (rename) memory locations, so what you'll actually see is something like this:

```
ADD counter, #1
```

This style makes the program much easier to understand and maintain. Still, even assembly language is not as easy for most programmers to learn and code with as a high-level language like BASIC. The example above is written in BASIC as follows:

```
counter = counter + 1
```

I think you'll agree that the BASIC code is much easier for the beginner to comprehend.

The SX/B compiler, then, will convert your BASIC program to the appropriate assembly code which the SX-Key IDE will assemble into the machine code output for the SX microcontroller.

SX/B differs from many other compilers in that it was designed to facilitate the learning of assembly language, so it does what I call *compile in place*. This means that every BASIC command is directly translated to the corresponding assembly language code as it appears in the listing. In fact, the assembly output retains the original SX/B source as comments so that you can see how SX/B commands are translated. The assembly output from the SX/B output looks like this:

```
0C01 ADD counter, #1         ;  counter = counter + 1
01ED
```

That the original BASIC source is included in the assembly language output becomes especially useful as your programming skills advance and you desire to add assembly segments to your SX/B programs; you can simply start with the assembly output from an existing SX/B command and modify it as required. This is particularly true of complex commands as is illustrated here.

```
0CFB    MOV FSR, #__TRISB          ;  PULSOUT Servo, 150
0024
005F    MODE $0F
0400    CLRB IND.0
0200    MOV  !RB, IND
0006
0018    BANK $00
0C01    XOR  RB, #%00000001
```

```
01A6
0C05    MOV  __PARAM3, #5
002A
0CAB    MOV  __PARAM4, #171
002B
02EB    DJNZ __PARAM4, @$
0010 0A44
02EA    DJNZ __PARAM3, @$-3
0010 0A44
0000    NOP
0000    NOP
0000    NOP
0C01    XOR  RB, #%00000001
01A6
```

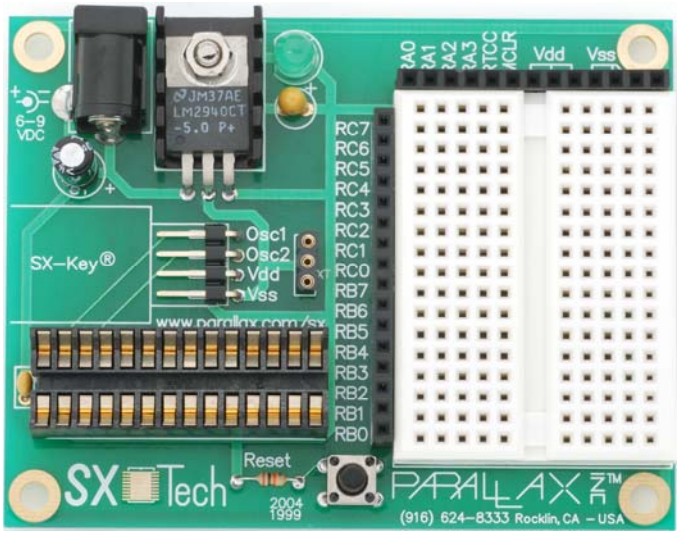
Now, don't let the listing above unnerve you – you may happily program the SX in SX/B for the rest of your days without ever having a look at the assembly output. Just remember that it is there for you to learn from if and when you decide to take that step.

2: SX Development Tools

While you can learn about programming with SX/B by reading a book, to actually learn the language will require that you write and execute real programs on an SX microcontroller, and usually with a bit of additional circuitry (e.g., LEDs, etc.). Thankfully, Parallax gives the experimenter a range of choices when working with the SX and associated circuitry.

SX Programming Boards

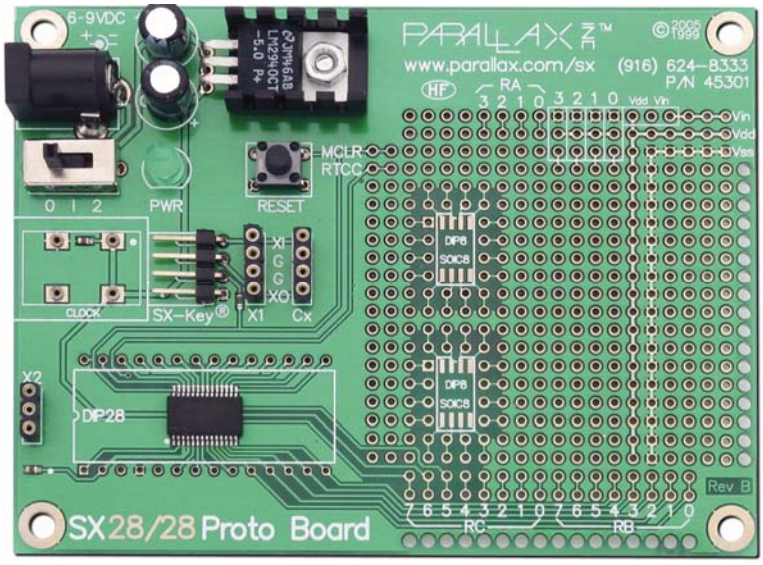
The SX Tech Board (#45205). This student-friendly board is somewhat similar to Parallax's famous Board of Education® platform. The SX Tech board employs an LIF (low insertion force) socket for the SX28AC/DP allowing you to use the board as a programmer for an SX28 that you intend to move to another device. A socket is also provided for a ceramic resonator so that you may experiment with running the SX at different speeds. All of the SX28 pins are made available to you through SIP headers, and a small breadboard allows you to construct circuits that attach to the SX28.



SX Tech Board

Figure 2.1

The SX28 Proto Board (#45302). If you're really in a budget pinch your friends at Parallax have an answer for you in the SX28 Proto Board. As with the SX Tech Board, the SX28 Proto board has a regulated 5-volt power supply and a socket for a ceramic resonator. In addition to the resonator socket, the SX28 Proto Board has a four-pin socket for a TTL oscillator. About half the board is configured as a through-hole prototyping area that includes pads for two 8-pin SOIC devices. You should, though, attach SIP socket headers and a small breadboard for your experimenting stages. Note that this board uses an SX28AC/SS that soldered in place, so while you can test SX28 programs with this board, you cannot move the chip to another project as with the SX Tech Board.

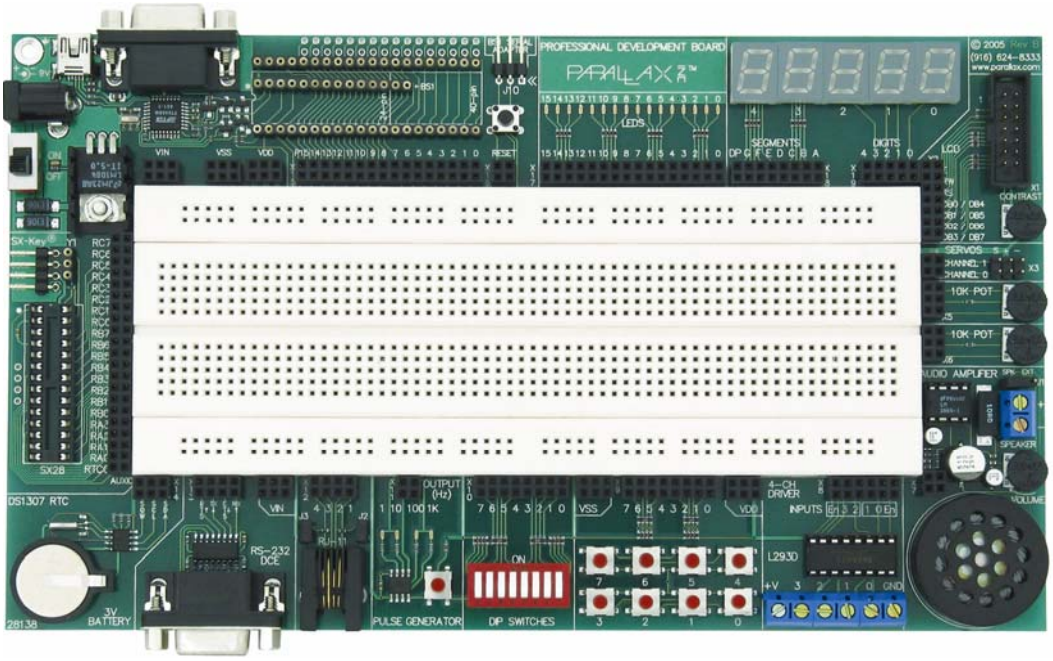


SX28 Proto Board

Figure 2.2

The Professional Development Board (#28138). If your budget allows then this is absolutely the best way to go in my opinion. In addition to a beefy power supply, the PDB includes a socket and programming connection for the SX28AC/DP, as well as sockets for all BASIC Stamp® and Javelin Stamp modules.

The PDB comes pre-equipped with LEDs (single and seven-segment), an LCD connection, servo ports, potentiometers, an audio amplifier, an L293 push-pull driver chip, push buttons, DIP switches, a pulse generator, an RJ-11 connection that can be used for 1-Wire® or X10® interfaces, an RS-232 interface and DB-9 connector, and a DS1307 Real Time Clock with battery backup. All of these components surround a large, solderless breadboard. The PDB has long been my platform of choice, and I use it when developing my own projects, as well as those for my clients.



Parallax Professional Development Board

Figure 2.3

Roll Your Own. If you have a well-stocked lab and are really the diehard DIY type, you can certainly create your own development board for the SX. See Appendix (Pending!) for a suitable schematic.

SX Programming Tools

Regardless of your development platform you will need a programming device to move code from the IDE into the SX chip. Your choices are the SX-Key and the SX-Blitz. The SX-Key gives you full programming and in-circuit debugging capabilities; this tool will generally be selected by professionals. The SX-Blitz is a great student tool; with it you can program the SX, but there are no facilities for debugging with this device, hence the simpler circuit and lower cost. For the most part, this book will assume that you're using the SX-Blitz.

SX-Key® Rev F

Figure 2.4

SX-Blitz USB

Figure 2.5

If you desire the capabilities of the SX-Key but don't have a standard serial port available on your computer, the SX-Key can be converted to USB with Parallax's USB to Serial RS-232 Adapter (#28030).

3: Quick Start - Success in Under 30 Minutes

There is nothing that inspires success like success, and you're about to succeed in programming the SX in under 30 minutes. Nothing fancy, mind you, but when you've succeeded here you'll know that you've got the pieces in place to move forward and become a master SX/B programmer.

To run the SX-Key IDE and program SX chips you'll need a Windows® PC. For best performance you should be running Windows XP with the latest service packages installed. See the Parallax web site (www.parallax.com) for the latest recommendations on PC specifications for Parallax development tools.

Step 1: Install the SX-Key IDE

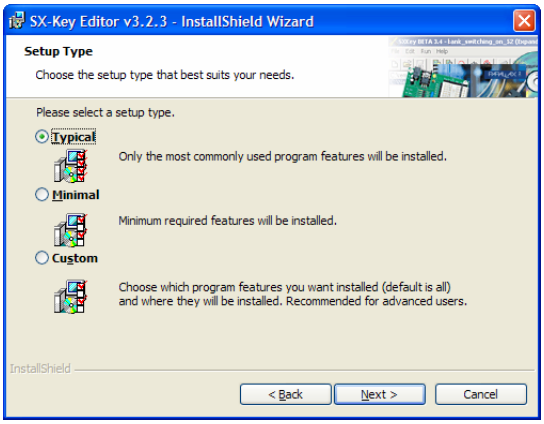
Even if you ordered product from Parallax and received a CD, I suggest that you navigate to www.parallax.com's downloads section and obtain the latest version of the SX-Key IDE. Parallax continuously updates and improves their tools based on customer feedback. CDs take time to produce, so the version of the SX-Key IDE on your CD could very well be outdated.

The SX-Key installer is very typical of Windows application installation programs. When first run, you'll see a standard "wizard" interface as shown in Figure 3.1. Click the **Next** button to start the actual installation.

SX-Key Editor Installer

Figure 3.1

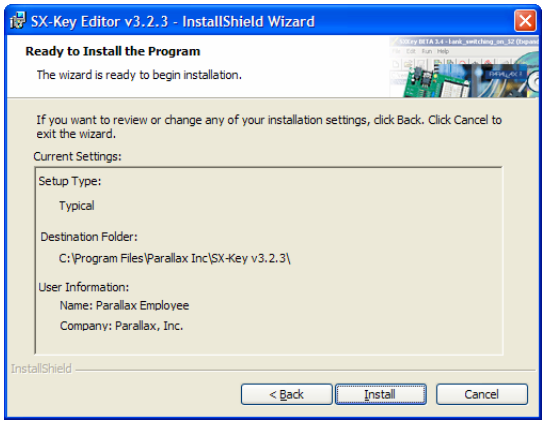
The next page of the installation wizard will prompt you for the installation type. The SX-Key IDE is pretty lean and doesn't use a lot of disk space, so you should always select **Typical**. Click the **Next** button to move on.



Select the **Typical** Setup

Figure 3.2

The next page gives you an overview of the installation details.



Installation Details

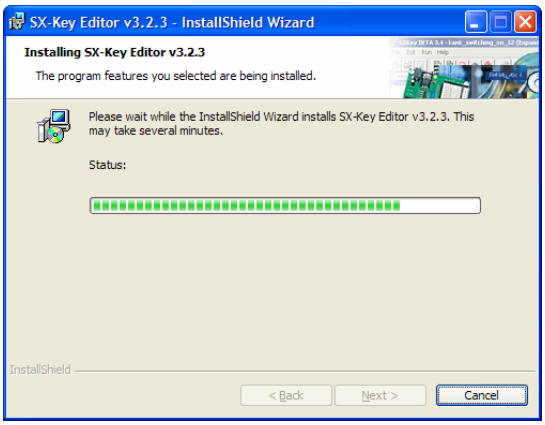
Figure 3.3

Note in particular the installation location on your system, it should look something like:

C:\Program Files\Parallax Inc\SX-Key v3.2.3\

The actual version number when you install may differ from what is illustrated above. The point is that you will need to know this location later when you want to update the SX/B compiler files or install the **SXSim** simulator program.

If you find some problem with the installation details click the **Back** button and fix it, otherwise click on **Install** and the SX-Key IDE software will be installed on your system. As stated earlier, it's not a big program and will only take a couple minutes to complete the installation process.



Installation Status Bar

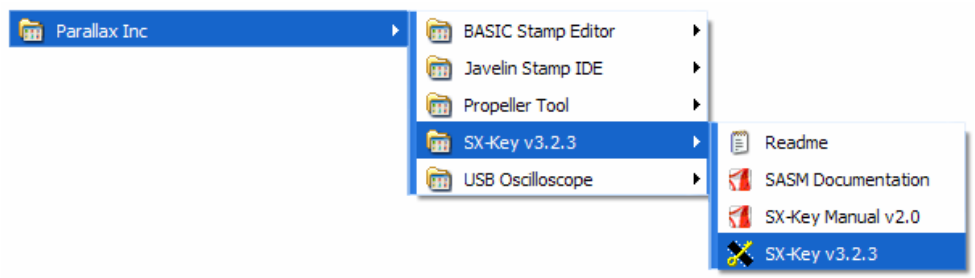
Figure 3.4

When the installation is complete you'll get one final page; click **Finish**.

Installation Complete

Figure 3.5

Now click on the Windows **Start** button, then **All Programs**, then the **Parallax Inc** group, then the **SX-Key** vX.Y.Z group, and then finally on the **SX-Key** vX.Y.Z program icon (Note that vX.Y.Z denotes the version number of the SX-Key software you just installed).

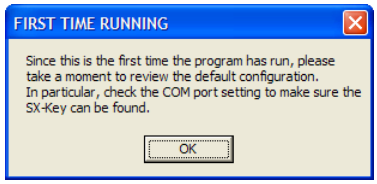


Launching the SX-Key Editor from the Start Button

Figure 3.6

To make starting the SX-Key IDE easier, right-click on the SX-Key vX.Y.Z program icon and then select **Pin to Start menu**. Doing this will allow you to start the SX-Key IDE with just two mouse clicks (Start \ SX-Key vX.Y.Z).

The first time the SX-Key IDE runs you will be prompted to do a bit of setup:



First Time Running

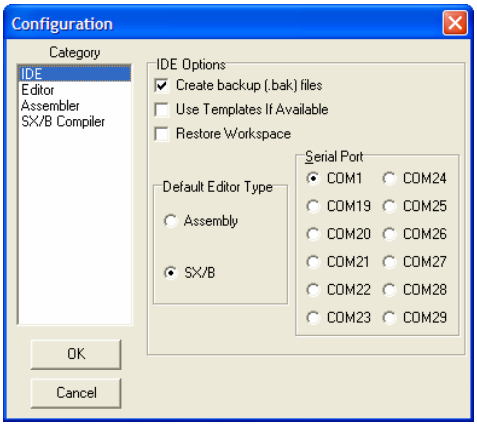
Figure 3.7

After clicking **OK** on the dialog above you'll be presented with the SX-Key IDE Configuration dialog that has four sections: IDE, Editor, Assembler, and SX/B Compiler.

The following screens show the suggested starting options for learning SX/B. Note that the list of serial ports on the IDE page is created at the time the dialog is opened, and will only show serial ports that are available on the system at that time – not all of them may be valid for SX programming. Be sure to select a known standard serial or USB port.

☑ Select **SX/B** as the **Default Editor Type**.

☑ Select the **COM** Port number you are most likely to use for SX programming.



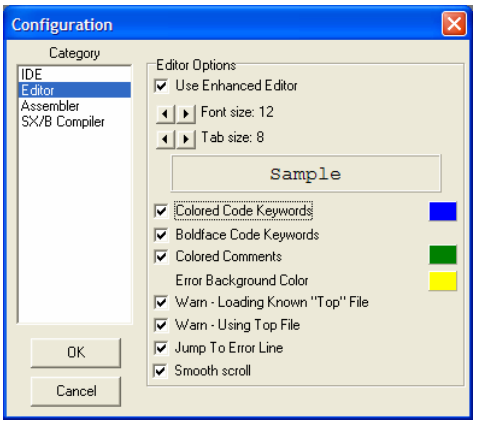
IDE Settings

Figure 3.8

Now click on **Editor** in the **Category** list.

☑ Set the **Font Size** to 12 or something comfortable for your screen.

☑ Check the **Colored Code Keywords** box.



Editor Settings

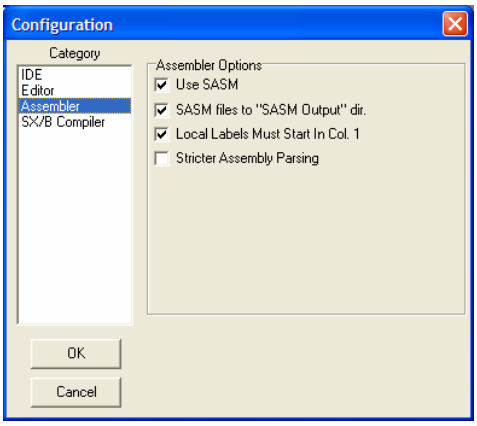
Figure 3.9

Next, click on the **Assembler** in the **Category** list.

☑ Check the **Use SASM** box.

☑ Check the **SASM files to "SASM Output" dir.** box.

☑ Check the **Local Labels Must Start in Col. 1** box.

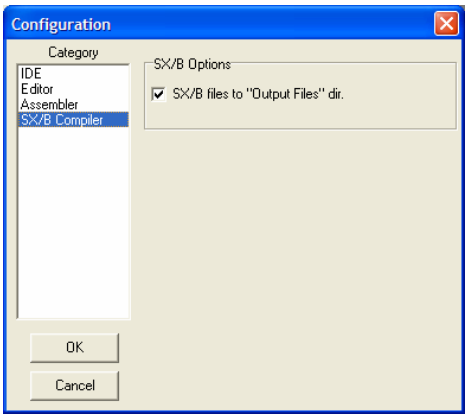


Assembler Settings

Figure 3.10

Finally, click on **SX/B Compiler** in the **Category** list.

☑ Check the **SX/B files to "Output Files" dir.** box.



SX/B Compiler Settings

Figure 3.11

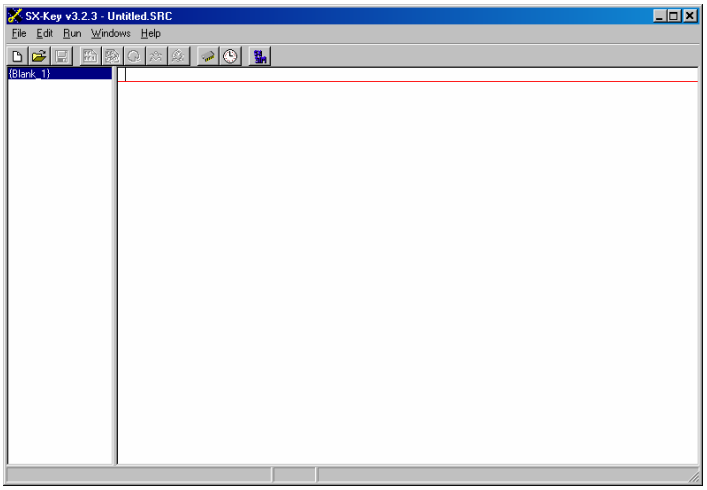
When you've reviewed and are satisfied with the Configuration settings, click **OK**. Note that you can come back and change these dialogs after the installation has been completed.

With the configuration setup complete you'll be prompted to accept a disclaimer as shown below. By using the SX-Key IDE you agree not to hold Parallax and/or Peter Montgomery ("the developers") responsible for any problems that arise from the use of the software, SX-Key programming tool, or the SX chip.

Disclaimer Dialog

Figure 3.12

After clicking **I Accept** on the disclaimer dialog (the program will not run until you do) the SX-Key IDE will open as shown in Figure 3.13.



The SX-Key IDE Main Window

Figure 3.13

Before you start programming in SX/B, however, you should check to make sure you've got the latest version of the compiler installed. This may seem confusing at first since the

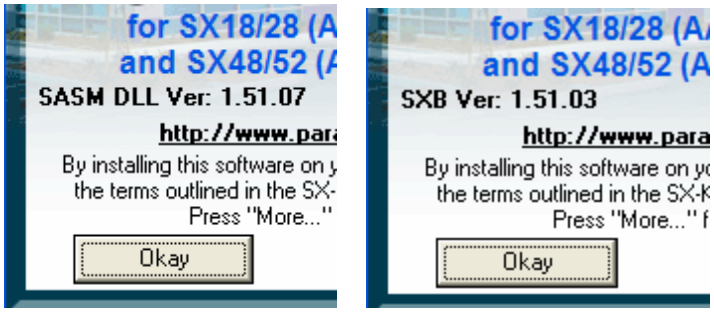
compiler is installed with the SX-Key IDE, but the fact is that the compiler is actually a separate component. You'll find that Parallax is very responsive to customer input, and having the SX/B compiler work as a separate module allows the compiler to be updated without having to modify the IDE. This architecture also allows Parallax to add other language modules (e.g., a C compiler) to the IDE if they should so choose.

Click on the **Help** menu, and then click **About**. You will see the SX-Key IDE About dialog as shown in Figure 3.14.

About Dialog (IDE Version)

Figure 3.14

Move the mouse cursor over the IDE version number, and then click on it. When you do you'll see that line change to the SASM DLL version number. Click one more time to display the SX/B compiler version, as shown in Figure 3.15 (right). Make a note of it.



Versions: SASM , left; SX/B, right.

Figure 3.15

Open your web browser and navigate to forums.parallax.com. When there, click on the SX Microcontroller forum and look for a "sticky" thread at the top that holds the latest version of the SX/B compiler. Click on this post and if the latest compiler is newer than the one you have, download the ZIP archive to your PC.

When you open the archive you will find four files:

- SXB.EXE
- RESERVED.TXT
- INVALID.TXT
- WHATS NEW X_YY_ZZ.TXT

The first three files are critical to the update; the "What's New" file is simply information for your reference and it doesn't hurt to keep a copy of it convenient. Extract these files to the SX/B compiler folder; the path will be something like:

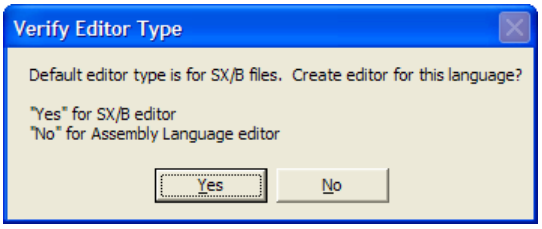
```
C:\Program Files\Parallax Inc\SX-Key vX.Y.Z\Compilers\SXB
```

(Again, "vX.Y.Z" refers to the version number of the SX-Key software that you just installed.) When the files have been replaced you can restart the SX-Key IDE and you're ready to write your first program.

Step 2: Ready to Code

Start the SX-Key program and create a blank SX/B file by clicking **File → New (SX/B)** from the main menu. Alternately, you can click on the New toolbar button: 

...and then select **Yes** when prompted by the **Verify Editor Type** dialog:



Verify Editor Type

Figure 3.16

If after following the steps above you end up with what looks like a blank program template, simply remove it by right-clicking in the program window, then clicking on Select all. With everything selected, press the Delete key. The use of templates will be covered in Chapter 13:

In the world of PC programming there is the ubiquitous "Hello, World!" program that sends a message to the screen; this is almost always presented as the introduction to a new language. Embedded microcontrollers usually don't have screens or fancy displays attached to them, but LEDs are very commonplace and used in the vast majority of microcontroller projects that require some kind of visual output. So, in the realm of the microcontroller, "Hello, World!" is realized by blinking a standard LED.

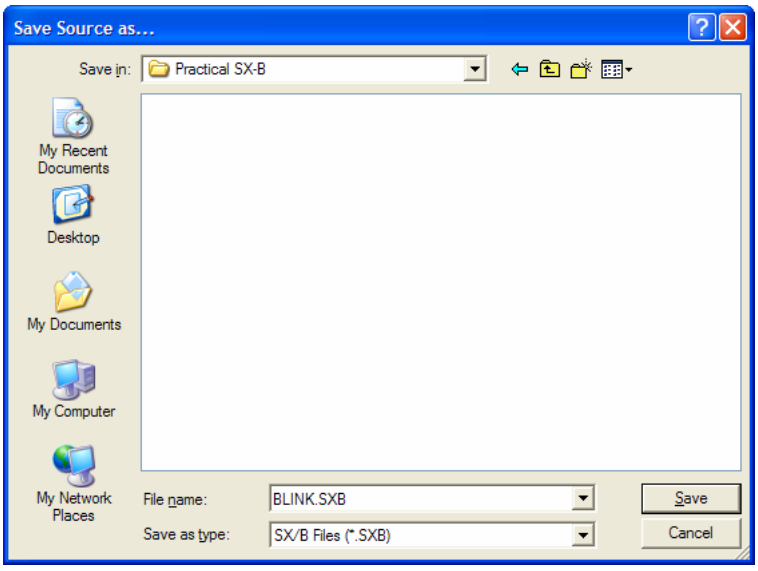
Don't scoff at the apparent simplicity – if you can get the program entered, saved, compiled, programmed into the SX, and running on your target hardware as expected, then you've completed all the steps required for any embedded program; the rest is simply code.

Here's your first SX/B program:

```
' BLINK1.SXB

DEVICE          SX28, OSC4MHZ, TURBO, STACKX, OPTIONX
FREQ            4_000_000

Led             PIN     RA.0


PROGRAM Start

Start:
  HIGH Led
  PAUSE 500
  LOW Led
  PAUSE 500
  GOTO Start
```

Enter the program exactly as shown above. When you're done, click the **Save** toolbar button: 

...or click **File → Save** from the main menu. Point the **Save Source** as dialog to a convenient location for your files (usually in the **My Documents** folder) and save the file as `BLINK.SXB`. Note that the SX-Key IDE will not compile or assemble code that hasn't been saved, so this is a necessary first step with every program. As the adage goes, *save early, save often*.



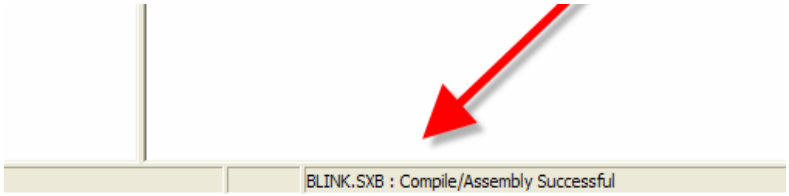
Save Source As...

Figure 3.17

When the program is saved, compile it by clicking the **Compile** toolbar button: 
...or or click **Run → Compile** from the main menu. If there are no errors the IDE status line will read:

"BLINK.SXB: Compile/Assembly Successful"

You should see this message in the bottom of the IDE window:



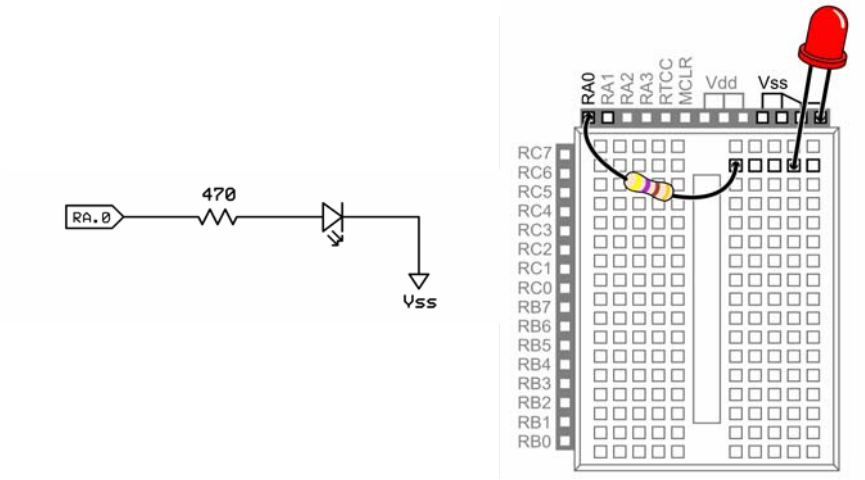
Compile/Assembly Successful

Figure 3.18

If you see this message, fantastic! If you don't, don't panic, it's probably something simple. Double-check the listing for errors and try again. One possibility that is common, especially with new programmers, is substituting "O" (the letter) for zero (the number), and vice versa. Once you get the program to compile/assemble successfully, it's time to download it into an SX28.

Step 3: LED Circuit

Assuming that you're not using a PDB, you'll need two components to complete the SX28 "Hello, World!" circuit: a 470-ohm (yellow-violet-brown) resistor, and a common LED. The schematic for the circuit and how it might look connected on an SX Tech Board are shown below.



LED Circuit Schematic and Wiring Diagram

Figure 3.19

If you're lucky enough to have a PDB, simply run a length of 22-gauge solid hook-up wire from the RA.0 socket to one of the individual LEDs at the top of the board.

Connect your SX-Blitz or Key to your PC, using the port you specified for programming earlier, and then connect it to your development system noting the pin orientation on the Blitz/Key and the board. If you happen to connect the Blitz/Key upside down you may damage it, so do be careful.

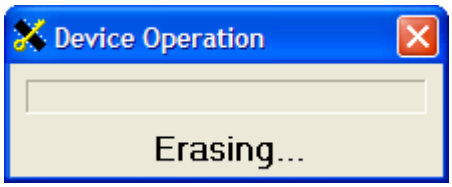
Be very careful when connecting the SX-Key to your development system; the safest way is without power applied to the circuit. If the SX-Key is misconnected when power is applied damage can occur.

Now connect a suitable DC power supply to the development board; 7.5 vdc @ 1 amp is suggested. The SX requires a bit of current for programming, so don't try to use a small (0.5 amp or less) power supply. With the development board powered up it's time for the final step.

Getting back to the SX-Key program, click the Program toolbar button: 

...or click **Run → Program** from the main menu.

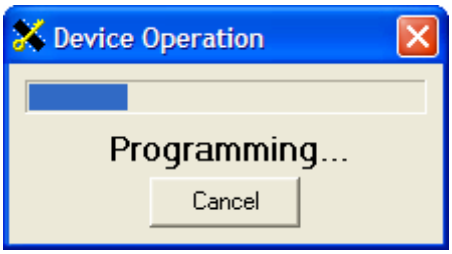
Since you've previously tested the program for syntax by compiling and assembling it, you should immediately see the **Device Operation: Erasing** dialog:



Erasing...

Figure 3.20

The Erasing dialog won't be onscreen long. What comes next is the Device Operation: Programming dialog, as shown in Figure 3.21.



Programming...

Figure 3.21

As you can see, a progress bar displays the current state of the programming process. When it is complete and the dialog disappears, have a look at your development board, you should have a blinking LED.

If you do, celebrate! If you don't – don't panic, it's probably something simple. Check these things:

- ☑ Does the development board have power?

- ☑ Is the circuit correctly connected? (Check the LED orientation.)
- ☑ Is the SX-Key/Blitz connected to the correct serial or USB port?

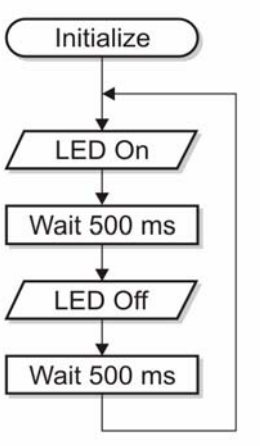
Once the offending issue has been located and removed, download the program and join the celebration – you're an SX/B programmer. Enjoy your success, and then when you're ready, turn the page and have a look under the hood of the LED blinker to see just how it works.

4: The LED Blinker Program - How it Works

While realize that blinking LEDs are not particularly glamorous, they are extraordinarily useful. Oftentimes a circuit will require a simple annunciator and the blinking LED is a great way to convey information to the user. How? Well, the rate at which the LED blinks, for example, is a quick visual indication of status: a slow rate might convey "everything okay" while a quick, somewhat urgent rate might convey, "emergency – deal with it now."

As you move forward there will be several experiments that show just how versatile a single LED can be. Before you construct other programs, though, work your way through the deconstruction of this one so you know exactly how it does what it does.

Many programmers start a project with a graphic device called a flowchart. These can be very useful tools in the design process, and I'm a big fan of flowcharting on a dry-erase board with colored markers. Here's what the `BLINK.SXB` program looks like in flowchart form:



The Blinker Program in Flowchart Form

Figure 4.1

The flowchart should immediately make sense based on the listing that you've already typed in and run. What should be pointed out, however, is that in `BLINK.SXB` the **Initialize** section of the flowchart is actually handled "behind the scenes" by SX/B.

In computer programming the term *initialize* is generally used to indicate the setup a known state; in embedded programming this is especially true for the user memory (RAM) and the

I/O pins. The fact is that many elements of the SX RAM power up in an unknown state and it is the programmer's responsibility to initialize them. Again, with the SX/B that is taken care of for us.

Advanced programmers will be happy to know that the automatic initialization of the SX RAM and I/O pins may be disabled if desired. See the `NOSTARTUP` option of the `PROGRAM` directive in the SX/B Online Help.

Line by Line

Okay, let's breakdown the program, line by line, and learn what it does:

```
' BLINK.SXB
```

This line is a *comment*, which gets ignored by the compiler as it is information intended for the programmer, not for the machine. A comment may start at the beginning of a line as shown above, or at the end of a line of source code, like this:

```
  HIGH Led                ' turn LED on
```

Comments always start with an apostrophe and all characters that follow to the end of that line are considered part of the comment. I tend to favor a coding style that minimizes comments – my time at the keyboard should produce work, not notes. As we move forward I will show you a programming style that will allow others to understand, maintain, or modify your programs without excessive commenting on your part.

```
DEVICE          SX28, OSC4MHZ, TURBO, STACKX, OPTIONX
```

The `DEVICE` directive tells the SX/B compiler and the SX-Key IDE what kind of chip you're programming, its clock source, and some specifics on how its code runs. While the SX can – in assembly language – be programmed in what is called "compatibility mode" (which uses a 4x clock cycle), the SX/B compiler is designed for a `TURBO` mode (1x clock) so a 20 MHz clock source gives you 20 million instructions per second (MIPS)! In "compatibility mode" the same clock would result in only five million instructions per second – clearly the advantage is `TURBO` mode. The `STACKX` and `OPTIONX` settings are also required, but only when using the SX18, SX20, or SX28. When programming the SX48 or SX52 the `TURBO`, `STACKX`, and `OPTIONX` settings are assumed and the IDE will generate an "Obsolete keyword" warning if these options are used.

The DEVICE line in this program, then, specifies that you're using an SX28. Note that packaging doesn't matter to SX/B or the SX-Key IDE; the chip can be in DP or SS form; it is still considered an SX28.

The setting OSC4MHZ tells the SX-Key IDE that the chip will use its internal 4 MHz oscillator to drive the chip. This is great for stand-alone projects where timing isn't critical, but it will not be suitable for timing-sensitive applications (those that use serial communications, for example). Other internal oscillator settings are OSC1MHZ, OSC128KHZ and OSC32KHZ, the latter being an excellent choice for low-speed, low power applications.

```
FREQ            4_000_000
```

The FREQ directive is used by the SX/B compiler when generating timing-specific sections of assembly code, as it does with PAUSE. This directive is also used by the SX-Key when an external clock source is specified. The SX-Key has an onboard clock generator that can drive the SX at the specified frequency. If you're using the SX-Blitz and specify an external clock source you'll need to connect that clock source (e.g., a ceramic resonator) before your program will run.

You can make large numbers easier to read by using an underscore character to separate groups of digits, for example: 4_000_000 is much easier to read than 4000000. This can be used with any of the numeric formats, and is especially useful for large decimal numbers, or for separating bits in a binary value, for example: %1001_0000.

```
Led             PIN     RA.0
```

This is the first line that has to do with your application. What you're doing with the PIN definition is renaming (often called aliasing) an I/O pin to make the program easier to read, debug, and maintain. Wow, all that from one definition? Yes.

Professional programmers consider it very bad form to embed the hardware pin names (e.g., RA.0) into working code. The reason is simple: computer code, whether it runs on your SX micro or a Cray super-computer, is a living, breathing, evolving entity that is usually changing over the design cycle. What happens, say, when you decide to move from an SX28 to and SX48 to get more pins and the new chip requires a change in I/O layout? Well, if you'd started by using the actual pin name through your program you'd have to go through and find all occurrences of the name and change them. Sure, you can use global search-and-replace but your task is simplified by having the name in one location, and the program will

be much easier for others to understand and maintain without you adding a lot of extra comments (something most programmers are loathe to do, anyway).

You can make your programming life simpler from the outset by using pin aliases instead of their actual hardware names.

```
PROGRAM Start
```

The PROGRAM directive serves two purposes: 1) it specifies the label at which your program actually begins (Start in this case), and 2) is specifies the location of the internally generated initialization code.

```
Start:
```

This is a program label, which is simply a place marker in the code. As with other programming languages, labels should begin in column one of the source file and be terminated with a colon. In SX/B, you must have a label that matches the specification in the PROGRAM directive. Note that when using a program label in code (or in the PROGRAM directive) the colon is not included. Labels may be up to 32 characters long, may include letters, numbers, and the underscore character, but they must not begin with a number.

```
Check_Level:
```

...is a valid label, while

```
1_Check:
```

...is not. Labels tend to be easier to read if the words within the label are separated by an underscore character, and new words begin with an uppercase letter.

```
  HIGH Led
```

At last, an instruction that does some real work. HIGH is a compound instruction that does two things: 1) it puts the specified pin into output mode, and 2) it connects the pin's output drive to Vdd (usually 5 volts), making it "high." With an active-high LED circuit (anode side connected to pin, cathode side connected to ground) the HIGH instruction will cause the LED to light.

```
  PAUSE 500
```

Even when the SX micro runs at the relatively "slow" speed of four megahertz, each instruction takes only 250 nanoseconds – that's 250/1,000,000ths of a second! Yes, it actually does take more than one machine instruction to execute `HIGH`, but still, you need to slow down the program in order to see the LED blink with your eyes.

`PAUSE` is designed to do just that; it holds the program right where it is (like the pause button on a DVD player) for a duration specified in milliseconds, usually from one to 65,535. In `BLINK.SXB` a period of 500 milliseconds will cause the `PAUSE` instruction to hold the LED on or off for one half second.

```
  LOW Led
```

`LOW` is the complement of `HIGH`. Like `HIGH`, the `LOW` instruction sets the specified pin to output mode, but it connects the pin's output driver to Vss (ground), causing the output to be a zero volts, or "low." With an active-high LED circuit the `LOW` instruction will cause the LED to extinguish. Next, another `PAUSE` command keeps the LED off for 500 ms, making the LED's on-time equal to the off-time.

```
  GOTO Start
```

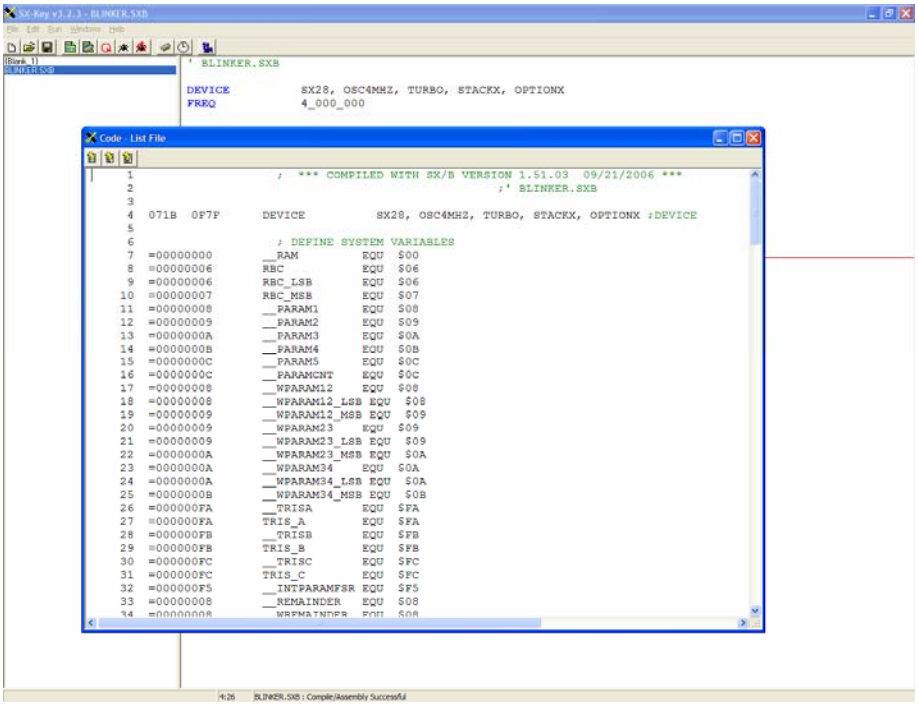
Finally, after the LED has been on a bit, then off a bit, you want the process to repeat. `GOTO` redirects the program to the specified label, creating what is called an *unconditional jump*. It is called unconditional because it always happens.

Some programmers consider the use of `GOTO` bad form because it can lead to what is called "spaghetti code." The creation of spaghetti code has to do with the programmer, not the programming language or its keywords. I believe the use of `GOTO` is fine, and as you'll see in later chapters there are other facilities in SX/B for both unconditional and conditional jumps.

A Look Under the Hood

Now that you know how each of the elements of `BLINK.SXB` work, have a look at the assembly output created by the SX/B compiler to see just how much work it does for you. First, make sure that you have compiled the program by clicking the **Compile** toolbar button (you may also click **Run → Compile** from the main menu, or use the keyboard shortcut **Ctrl+A**).

Now click on **Run → View List** (use the keyboard shortcut **Ctrl+L**). You should get a new window that floats above the IDE that contains the assembly listing of the `BLINK.SXB` program.



Viewing the Assembly Listing of the BLINK.SXB Program

Figure 4.2

Go ahead and scroll through this listing. Don't be nervous, most of the time you'll never look at the assembly output but it's good to know that it's there for your review and edification. When you get to the point where you want to learn assembly language, looking at the assembly listing from a working SX/B program is a great place to learn some great tricks and techniques.

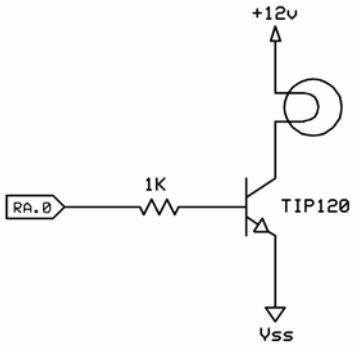
As you examine the assembly listing you'll see that there's actually quite a bit of code between the `PROGRAM Start` directive and the `Start` label; this is the "behind the scenes" initialization code that was discussed earlier. This bit of code clears the RAM space,

initializes the I/O pins (usually to inputs unless otherwise instructed in the `PIN` declaration), and can handle other initialization chores that we'll examine later.

As you can see, the assembly listing contains your SX/B program instructions. Isn't it interesting that a single-line instruction like `HIGH Led` expands out to seven lines of assembly code? This is the reason programmers use compilers – it saves time. The benefit of using the SX/B compiler is that you get the ease of BASIC programming with the blazing execution speed of SX assembly.

5: Beyond the Basic Blink

With a firm grip on simple LED blinking it's time to explore additional strategies. Keep in mind that when you can control an LED you can control just about anything, usually with some very simple interface circuitry. To control an incandescent lamp, for example, a transistor circuit can be used in place of the LED and resistor as shown below.



Transistor Circuit for an Incandescent Lamp

Figure 5.1

You probably notice that the on- and off-time delays for the blinker program are the same. When this is the case, the program can be simplified like this:

```
' BLINK2.SXB

DEVICE          SX28, OSC4MHZ, TURBO, STACKX, OPTIONX
FREQ            4_000_000
Led             PIN     RA.0

PROGRAM Start

Start:
  DO
    TOGGLE Led
    PAUSE 15
  LOOP
```

Update the blink program and save it as a new file (use **File → Save As**) called `BLINK2.SXB`. When the program is run it will behave exactly like the first version, albeit with a bit less code.

In the first version of the program **GOTO** was used to create an infinite loop. In this version, a more modern programming style is employed via **DO–LOOP**. The advantage of **DO–LOOP** is that you don't have to create a label as with **GOTO** (remember, the `Start` label in this program is required by the **PROGRAM** directive).

In this form, **DO–LOOP** is unconditional. So, how does it work? Again, it's the compiler doing some behind-the-scenes work for you. The assembly listing will show that the compiler created an internal label for the jump. **DO–LOOP** is especially appropriate for small loops like this where `DO` and `LOOP` can be easily matched when reading a listing. As your programs grow and you have larger sections of code within the loop, **GOTO** may be a better choice to make the program easier for yourself and others to follow.

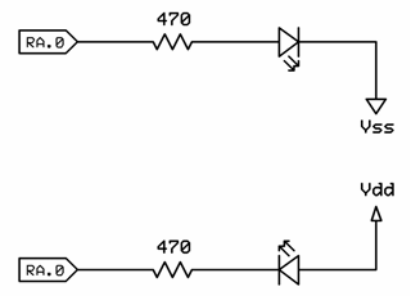
The LED state is now controlled with **TOGGLE**. As with **HIGH** and **LOW**, **TOGGLE** sets the specified pin to output mode, but then it inverts the state of the pin; if the pin was off, it gets turned on, and vice-versa. And now when you want to modify the LED flash rate there is but one **PAUSE** value to change.

Experiment with the blink rate by adjusting the `PAUSE` and re-running the program. Notice that at `PAUSE` values below 20 the blinking process is no longer visible due to the human eye's persistence of vision and, in fact, the LED appears just a bit dim. Fast switching of the LED, called *pulse width modulation*, can be used to control LED brightness.

As you've seen, **HIGH** and **LOW** are very easy to use, but there is a bit of a trap. Most newcomers consider "high" to mean "on." In the real world, however, this is not always the case. Have a look at the two LED circuits in Figure 5.2.

Both are valid. The top circuit will light the LED when the SX output is high – this is called an *active-high* circuit. The bottom circuit works as well, but it will light the LED when the SX output is low, so it is called an *active-low* circuit.

Note: The experiments in this book will use active-high circuits to maintain compatibility with the built-in circuitry of the Professional Development Board.



LED Schematics:
Active-high LED (top)
Active-low LED (bottom)
Figure 5.2

Since circuit designs can vary you should not assume that a high output will turn something on and, in fact, it's a good idea to design your programs to be hardware agnostic. With very little effort you can update the original blinker program to be hardware agnostic and virtually self-commenting.

```
' BLINK3.SXB

DEVICE          SX28, OSC4MHZ, TURBO, STACKX, OPTIONX
FREQ            4_000_000

Led             PIN     RA.0 OUTPUT

IsOn            CON     1
IsOff           CON     0


PROGRAM Start

Start:
  DO
    Led = IsOn
    PAUSE 500
    Led = IsOff
    PAUSE 500
  LOOP
```

Update the program as shown above and save as `BLINK3.SXB`. Notice the change to the **`PIN`** declaration: the **`OUTPUT`** modifier has been added after the pin name. The **`OUTPUT`** modifier will cause the compiler to place additional code into the initialization section that sets this pin to output mode. When a pin is in output mode you can make it high by writing a "1" to the pin, or low by writing "0" to it.

That said, you should not embed 1 or 0 into your programs, instead, rename these values so that the program becomes self-commenting and easier to maintain. This can be

accomplished with the **CON** (*constant*) declaration. The **CON** declaration allows you to create a name for a numeric value, character, or even string of characters. Constant values cannot be changed during the run of a program, hence the term, *constant*. These are all valid **CON** declarations:

```
RoomTemp        CON     72          ' a numeric value
TempMode        CON     "F"         ' a character
Baud            CON     "N9600"     ' a string of characters
```

When the program is compiled and assembled the constant name is replaced by its value from the **CON** declaration. Using **CON** declarations will make your listings easier to read, and also much easier to maintain, especially when the same value is used in several places in your program.

Back to BLINK3.SXB, there is no doubt now as to the purpose of these lines of code:

```
    Led = IsOn
    Led = IsOff
```

This is good example of self-commenting code, and it is clear that no additional comments are required here. Through the judicious use of PIN and CON declarations, your programs can be easier to write and maintain, and immediately useful to others without them having to sift through a lot of comments. Please trust that you'd rather be typing working code than comments.

Go ahead and have a look at the assembly listing (use the **Ctrl + L** keyboard shortcut) to see how the SX/B compiler handles:

```
   Led = IsOn
```

What you should see is that this compiles to just a single line of assembly code:

```
  SETB Led
```

Remember that when using **HIGH** it took seven lines of assembly code to turn the LED on or off. For blinking LEDs program execution speed is of little consequence, but in other applications, where speed is paramount, this strategy will be important.

The assembly mnemonic SETB stands for "Set Bit." This instruction writes a 1 to the specified bit variable (an I/O pin in this case). The complement of SETB is CLRB which writes a 0 to the specified bit.

Blinks with Meaning

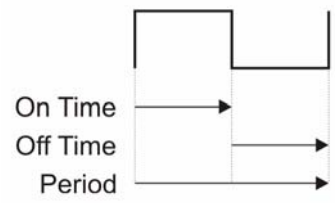
In hardware designs that use an LED annunciator, there are three common strategies employed to convey information through the blink: rate, duty cycle, and coded blinking.

When rate style blinking is employed, the on- and off-times are the same as demonstrated in BLINK2.SXB. Still, **TOGGLE** actually sets the pin to output mode every time it's executed and this is no longer required when you use the **OUTPUT** modifier in the **PIN** declaration. When a pin is an output, its state can be inverted with the ~ operator.

```
' BLINK4.SXB

DEVICE          SX28, OSC4MHZ, TURBO, STACKX, OPTIONX
FREQ            4_000_000

Led             PIN     RA.0 OUTPUT

IsOn            CON     1
IsOff           CON     0


PROGRAM Start

Start:
  DO
    Led = ~Led
    PAUSE 500
  LOOP
```

Update the blinker program and save as BLINK4.SXB. As you can see, with two fewer lines of code the LED still blinks, and by changing the PAUSE to a smaller value it blinks at a higher rate, conveying a sense of urgency.

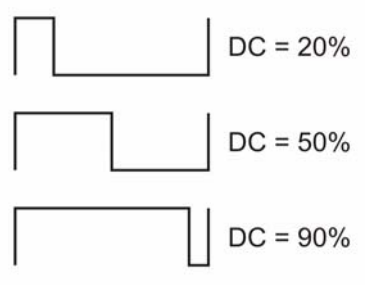
Another way to express information through the LED blink is with *duty cycle*. Duty cycle is the percentage of the LED's on-time versus the overall period, which is the sum of the on- and off-time values as shown below.



50% Duty Cycle

Figure 5.3

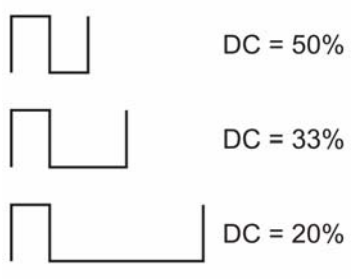
When on- and off-times are equal the duty cycle is 50% as the on-time is half the duration of the period. By changing the on-time and keeping the same period, you can change the duty.



Changing the On-time Changes the Duty Cycle

Figure 5.4

You can also change the duty cycle by keeping the on-time constant and changing the off-time, which changes the overall period.



Changing the Off-time Changes the Period

Figure 5.5

Reopen `BLINK3.SXB` and modify the `PAUSE` values to change the duty cycle. Note that when using small `PAUSE` values (5 to 15), the apparent brightness of the LED can be affected by changing the duty cycle.

When working with period signals like the blinking LED control you will hear the term, *frequency*. Frequency, expressed in Hertz (Hz), is the number of cycles per second of a periodic signal. Frequency can be calculated from a known period:

```
Frequency = 1 / Period
```

In `BLINK4.SXB` the period is 0.2 seconds (0.1 second on plus 0.1 second off) so the frequency is 5 Hz.

As you've seen, rate and duty cycle control are useful ways to convey specific meaning with a blinking LED. Another method is with coded blinking. You could, for example, have a piece of automation that once used an LED to indicate "okay" and "not okay" that now needs to be updated to provide additional information from that same LED. Rate and duty cycle can sometimes be misinterpreted when used for indicators, so coded blinking is a good choice.

Update the LED blinker as shown below and save as `BLINK5.SXB`.

```
' BLINK5.SXB

DEVICE          SX28, OSC4MHZ, TURBO, STACKX, OPTIONX
FREQ            4_000_000

Led             PIN     RA.0 OUTPUT

IsOn            CON     1
IsOff           CON     0


PROGRAM Start

Start:
  DO
    Led = IsOn
    PAUSE 250
    Led = IsOff
    PAUSE 250
    Led = IsOn
    PAUSE 250
    Led = IsOff
    PAUSE 250
    Led = IsOn
    PAUSE 250
    Led = IsOff
    PAUSE 250
    PAUSE 1000
  LOOP
```

When you run the program you'll see that the LED blinks three times, followed by a longer off period. The value of this technique should be immediately obvious, but the code is not quite as clean as it could be.

Thus far you have used **GOTO** *Label* and **DO–LOOP** to control program flow. In both cases, the loop was infinite, i.e., it ran forever. This coded blinker could benefit from a controlled loop and that is accomplished with the FOR–NEXT structure as shown in the updated listing below.

```
' BLINK5a.SXB

DEVICE          SX28, OSC4MHZ, TURBO, STACKX, OPTIONX
FREQ            4_000_000

Led             PIN     RA.0 OUTPUT

IsOn            CON     1
IsOff           CON     0

blinks          VAR     Byte


PROGRAM Start

Start:
  DO
    FOR blinks = 1 TO 3
      Led = IsOn
      PAUSE 250
      Led = IsOff
      PAUSE 250
    NEXT
    PAUSE 1000
  LOOP
```

The **FOR–NEXT** structure takes the following form:

```
  FOR controlValue = StartValue TO EndValue
    ' program statements
  NEXT
```

As you can see, **FOR–NEXT** requires a control value, and this must be a *variable*. A variable is a value that can change, and as shown above is defined with the VAR declaration. SX/B provides support for three variable types: *Bit*, *Byte*, and *Word*. The table below shows the range of values for each variable type.

Bit: 0 to 1
Byte: 0 to 255 (eight bits)
Word: 0 to 65,535 (16 bits)

When defining a variable you should always be mindful of the value range it will hold; using a variable type that exceeds the actual program requirements is wasteful of valuable memory space.

The only native variable types in the SX microcontroller are Bit and Byte. The Word variable type is synthesized by the compiler.

Those coming from a background in BASIC Stamp programming may be wondering about the Nib variable type. Nib is a non-native type that is synthesized in the BASIC Stamp. In the Code Library you'll find a function to extract a specified Nib from a value.

When the program encounters a **`FOR–NEXT`** loop *StartValue* (which can be a constant or variable) is copied to the control variable. The contents of the loop are executed and at **`NEXT`** the value of the control variable in incremented and compared to *EndValue* (which can also be a variable or constant). When the control variable is less than or equal to the *EndValue* the loop will continue to run. As soon as the control variable exceeds the *EndValue* the loop will terminate and the program will continue at the line that follows **`NEXT`**.

Experiment with the *EndValue* setting in `BLINK5a.SXB`. At what point does this value become unwieldy at providing useful information?

Can you rewrite the program to use LED toggling instead of the on and off methods shown above? Yes, of course you can, but you'll want to make sure that the `FOR–NEXT` loop always runs an even number of cycles so that the LED is off before the long delay.

6: Decision-Making and Program Flow

To this point your programs have run in fixed loops; most real-world applications, however, will need to make decisions that change the flow of the program. In all variants of the BASIC programming language, the **IF–THEN** construct is used as the primary decision maker.

In its simplest form, **IF–THEN** assumes this structure:

```
IF Condition THEN Label
```

… where *Condition* is a Boolean expression that will evaluate as True or False. When *Condition* evaluates as True the program will jump to *Label*, otherwise it will fall through to the line that follows `IF-THEN`. Some experienced programmers may think this form of **IF–THEN** as odd, having perhaps been accustomed to the more modern form:

```
IF Condition THEN
  ' program statements
ELSE
  ' other program statements
ENDIF
```

SX/B supports both forms of **IF–THEN**, though the first more closely matches the actual structure of SX assembly code. Both forms are valid and have their place in your applications.

The **IF–THEN** *Condition* element is used to compare a variable to another value; the second value may be a variable or may be a constant. Several comparison operators are available for **IF–THEN**:

- = Equal
- <> Not Equal
- > Greater Than
- >= Greater Than or Equal
- < Less Than
- <= Less Than or Equal

SX/B allows just one comparison per line, so checking for a variable to be within (or excluded from) a given range requires two **IF–THEN** statements, like this:

```
IF variable >= LowLimit THEN
  IF variable <= HighLimit THEN
    ' variable is in range
  ENDIF
ENDIF
```

Modify the blinker as shown below and save it as `BLINK6.SXB`. Can you determine the program's behavior before running it?

```
' BLINK6.SXB

DEVICE          SX28, OSC4MHZ, TURBO, STACKX, OPTIONX
FREQ            4_000_000

Led             PIN     RA.0 OUTPUT

IsOn            CON     1
IsOff           CON     0

code            VAR     Byte
blinks          VAR     Byte


PROGRAM Start

Start:
  code = 3

Main:
  IF code = 0 THEN Start
  FOR blinks = 1 TO code
    Led = IsOn
    PAUSE 250
    Led = IsOff
    PAUSE 250
  NEXT
  PAUSE 1000
  GOTO Start
```

If you deduced that, essentially, nothing would happen then you're right – as far as the LED is concerned. Still, the program is in fact running. With the value of *code* set to zero, the **IF–THEN** condition evaluates as True and the program is redirected back to Start. A non-

zero value in *code* causes the **IF–THEN** condition to evaluate as False and the program drops through to the **FOR–NEXT** loop that causes the LED to flash.

As suggested earlier, SX/B supports modern **IF–THEN** and **IF–THEN–ELSE** constructs as well, as is demonstrated by this version of the program.

```
' BLINK6a.SXB

DEVICE          SX28, OSC4MHZ, TURBO, STACKX, OPTIONX
FREQ            4_000_000

Led             PIN     RA.0 OUTPUT
IsOn            CON     1
IsOff           CON     0
code            VAR     Byte
blinks          VAR     Byte

PROGRAM Start

Start:
  code = 4

Main:
  IF code = 0 THEN Start
  IF code < 4 THEN
    FOR blinks = 1 TO code
      Led = IsOn
      PAUSE 250
      Led = IsOff
      PAUSE 250
    NEXT
    PAUSE 1000
  ELSE
    Led = IsOn
    PAUSE 100
    Led = IsOff
    PAUSE 100
  ENDIF
  GOTO Start
```

Can you tell how the program will behave before you run it? Do try. This version is certainly more interesting and closer to what one might find in a control application that uses a single LED annunciator.

The early form of **IF–THEN**, while seemingly simple, actually lets one write code that is quite easy to follow and this is very important for the long-term maintenance of your

programs. The following program, which controls the LED via duty cycle, demonstrates this fact.

```
' BLINK7.SXB

DEVICE          SX28, OSC4MHZ, TURBO, STACKX, OPTIONX
FREQ            4_000_000

Led             PIN     RA.0 OUTPUT

IsOn            CON     1
IsOff           CON     0

code            VAR     Byte


PROGRAM Start

Start:
  code = "Q"

Main:
  IF code = "Q" THEN Quick_Blink
  IF code = "M" THEN Med_Blink
  IF code = "L" THEN Long_Blink
  GOTO Start

Quick_Blink:
  Led = IsOn
  PAUSE 100
  Led = IsOff
  PAUSE 900
  GOTO Start

Med_Blink:
  Led = IsOn
  PAUSE 500
  Led = IsOff
  PAUSE 500
  GOTO Start

Long_Blink:
  Led = IsOn
  PAUSE 800
  Led = IsOff
  PAUSE 200
  GOTO Start
```

When the list of options is short and of non-contiguous values, as is the case above, there are other strategies available to the SX/B programmer. One such option is **ON–GOTO**, as demonstrated here (only the section at `Main` is shown):

```
Main:
  ON code = "Q", "M", "L" GOTO Quick_Blink, Med_Blink, Long_Blink
  GOTO Start
```

Internally, the SX/B generates the same output as with the previous **IF–THEN** version; the advantage here is fewer lines of SX/B source code. That said, this form may not be quite as obvious as the **IF–THEN** form, and is only suitable for short lists of options as SX/B statements are restricted to a single line.

Finally, when the control value is contiguous, (e.g., 0, 1, 2, 3, etc.) the code above can be simplified to:

```
Main:
  ON code GOTO Quick_Blink, Med_Blink, Long_Blink
  GOTO Start
```

This form of **ON–GOTO** expects the control variable to be from zero to N, when N is the number of times in the labels list minus one. SX/B also includes the **BRANCH** instruction for compatibility with the BASIC Stamp and other BASIC-language embedded controllers. The following instruction using `BRANCH` is behaves the same as **ON–GOTO** above:

```
Main:
  BRANCH code, Quick_Blink, Med_Blink, Long_Blink
  GOTO Start
```

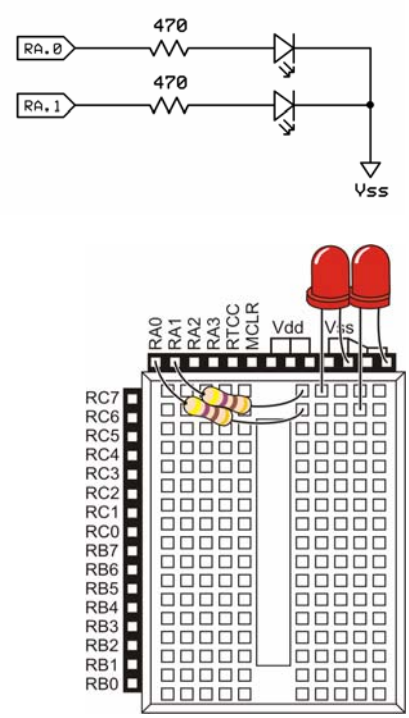
Experiment with the control structures demonstrated in this chapter. Use primary and secondary control values to add interest and variety to your LED blinker.

7: Controlling Multiple LEDs

As your projects grow in sophistication you will, undoubtedly, have need to control two or more LEDs. You might suppose that in situations where independent timing of each LED is required that the program will require extensive use of **`IF-THEN`** constructs and timing variables for each LED. Of course there will be programs where this is the required solution, but there is in fact a simpler way to control two more LEDs independently with a single variable – if the application allows for an implicit timing relationship between the LEDs.

This programming trick takes advantage of the SX's ability to copy any bit to any other bit, as well as the normal bit pattern of an increasing binary value.

Add a second LED to your project, connected to pin RA.1.



Two-LED Schematic and Wiring Diagram

Figure 7.1

Enter and save the following program as `BLINK8.SXB`.

```
' BLINK8.SXB

DEVICE          SX28, OSC4MHZ, TURBO, STACKX, OPTIONX
FREQ            4_000_000

Led0            PIN     RA.0 OUTPUT
Led1            PIN     RA.1 OUTPUT

L0              CON     0
L1              CON     2
ctrl            VAR     Byte            ' timing control

PROGRAM Start

Start:
  DO
    Led0 = ctrl.L0
    Led1 = ctrl.L1
    PAUSE 250
    INC ctrl
  LOOP
```

When you run the program you'll see that Led0 flashes quickly while Led1 flashes more slowly, though both with a 50% duty cycle (i.e., the on- and off-times are equal). This happens because the ever changing bits of the variable *ctrl* are being copied to the LEDs. Table 7.1 shows the bit pattern of `ctrl` at the beginning of the program.

Bit Pattern	Value of *ctrl*
%00000000	(ctrl = 0)
%00000001	(ctrl = 1)
%00000010	(ctrl = 2)
%00000011	(ctrl = 3)
%00000100	(ctrl = 4)
%00000101	(ctrl = 5)
%00000110	(ctrl = 6)
%00000111	(ctrl = 7)
%00001000	(ctrl = 8)

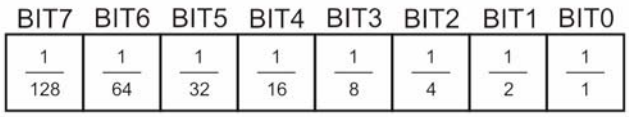
Table 7.1
The Bit Pattern for the *ctrl* Variable

Note that BIT0 (right-most column), changes every other count whereas BIT2 (second column from right) changes every fourth count. Since the LED pins have been set as outputs by the `PIN` declaration, copying a bit from the timing control variable allows for direct control.

You may use any bit (0..7) from a [byte] variable using the *dot notation* shown in the program. SX/B does not allow the position to be expressed as a variable value, you must use a constant (aliased or embedded as shown).

SX/B allows dot notation of bit position with word variables as well; the legal range for bits with a word is zero (LSB) to 15 (MSB).

The following chart shows the timing relationship between each of the bits in the control variable.



Timing Chart

Figure 7.2

As you can see, using BIT0 causes the connected LED to change every cycle through the loop, using BIT1 causes the LED to change every second cycle (half the base rate), BIT2 every fourth cycle (one fourth the base rate), etc.

When using this strategy the only thing that can be set is the base timing rate with the loop delay, the relationship of all other bits to BIT0 is fixed. Note, too, that the program takes advantage of `INC` which is a shortcut for:

```
value = value + 1
```

When `INC` is applied to a variable at its maximum value (255 for bytes, 65535 for words) the variable will *roll over* to zero.

SX/B as well as SX assembly include the DEC instruction which is the complement to INC and is the same as:

```
value = value - 1
```

When DEC is applied to a variable at its minimum value it will *roll under* to its maximum (255 for bytes, 65535 for words).

In those applications where the implicit timing between the bits in a single control variable will not work, independent timing control – while maintaining a 50% duty cycle – is possible by using a control variable (timing multiplier) for each output. Enter and save the following program as BLINK9.SXB.

```
' BLINK9.SXB

DEVICE          SX28, OSC4MHZ, TURBO, STACKX, OPTIONX
FREQ            4_000_000

Led0            PIN     RA.0 OUTPUT
Led1            PIN     RA.1 OUTPUT

TLed0           CON     1           ' multiplier for LED 0
TLed1           CON     7           ' multiplier for LED 1

ctrl0           VAR     Byte        ' timing control for LED 0
ctrl1           VAR     Byte        ' timing control for LED 1

PROGRAM Start

Start:
  IF ctrl0 = 0 THEN                 ' timer expired?
    Led0 = ~Led0                    ' invert LED state
    ctrl0 = TLed0                   ' reload timer
  ELSE
    DEC ctrl0                       ' count down
  ENDIF

  IF ctrl1 = 0 THEN
    Led1 = ~Led1
    ctrl1 = TLed1
  ELSE
    DEC ctrl1
  ENDIF

  PAUSE 100
  GOTO Start
```

As this program uses independent timing multiplier variables for each LED it affords a great deal more flexibility that the BLINK8.SXB, though it requires additional code for each. LED control is based on the value currently loaded in the control variable. When this variable is zero the LED state gets inverted and the variable is reloaded with the timing multiplier for the LED. On those passes through the program loop when the control variable is not zero the value is simply decremented.

While this program is more flexible than the former, the LED duty cycle is still fixed at 50% and this may not always be desirable. Should your application require a variable duty cycle for each LED, you may be inclined to think that a second control variable is required for each LED. Indeed, this is not the case as demonstrated in the updated version, BLINK9a.SXB, shown below.

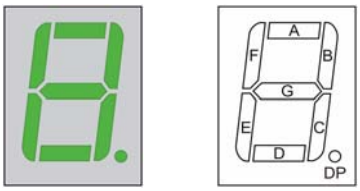
```
' BLINK9a.SXB

DEVICE          SX28, OSC4MHZ, TURBO, STACKX, OPTIONX
FREQ            4_000_000

Led0            PIN     RA.0 OUTPUT
Led1            PIN     RA.1 OUTPUT

IsOn            CON     1
IsOff           CON     0

TLed0           CON     1           '  on multiplier for LED 0
TLed0Off        CON     5           ' off multiplier for LED 0
TLed1           CON     7           '  on multiplier for LED 1
TLed1Off        CON     3           ' off multiplier for LED 1

ctrl0           VAR     Byte        ' timing control for LED 0
ctrl1           VAR     Byte        ' timing control for LED 1


PROGRAM Start

Start:
  IF ctrl0 = 0 THEN                 ' timer expired?
    IF Led0 = IsOff THEN            ' is LED off?
      Led0 = IsOn                   '   then turn on and
      ctrl0 = TLed0                 '   load on multiplier
    ELSE
      Led0 = IsOff                  '   else turn off and
      ctrl0 = TLed0Off              '   load off multiplier
    ENDIF
  ELSE
    DEC ctrl0                       ' count down
```

```
  ENDIF

  IF ctrl1 = 0 THEN
    IF Led1 = IsOff THEN
      Led1 = IsOn
      ctrl1 = TLed1
    ELSE
      Led1 = IsOff
      ctrl1 = TLed1Off
    ENDIF
  ELSE
    DEC ctrl1
  ENDIF

  PAUSE 100
  GOTO Start
```

Up to this point, the programs have set the LED state. As demonstrated in `BLINK9a.SXB` the current LED state can in fact be read by the program as with any other bit variable. So, upon expiration of the associated control variable the state of the LED is tested and the new state and appropriate timing multiplier is loaded accordingly. By using separate timing multipliers for the on- an off-states of the LED, the duty cycle may be independently set for each resulting in very sophisticated output patterns with simple code.

8: 7-Segment LED Displays

Multiple LEDs become particularly useful when they take a thin, rectilinear shape and are physically arranged and packaged as in the illustration below:



7-Segment Display

Figure 8.1

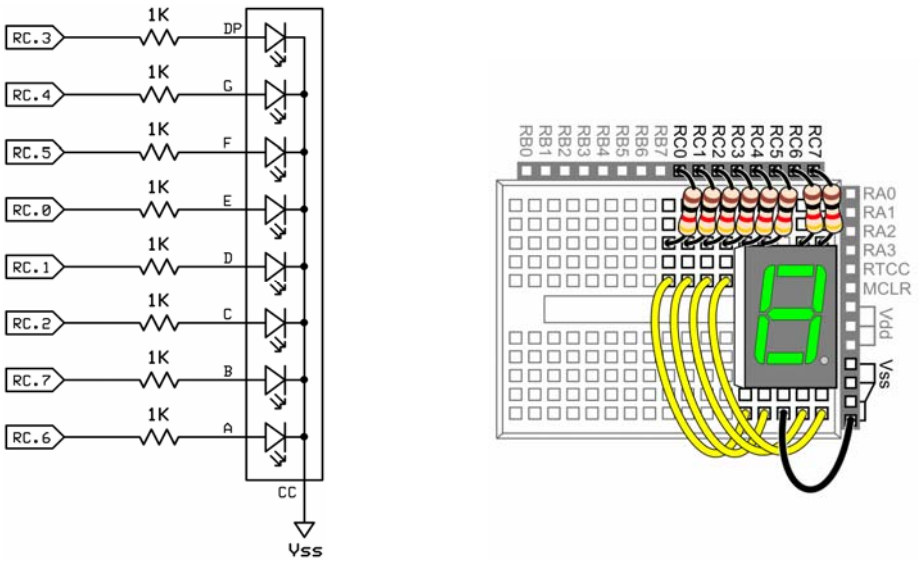
No doubt you've see such displays any number of times in any number of applications, from alarm clocks to cable TV decoders to smart toasters. Seven segment displays, as they're called, have been used since the early days of the space program and still find favor for their low cost, ease of use, as well as their exceptional variety of size and color options.

While called seven segment displays, the modules usually include an eighth LED to form a decimal point as shown in the illustration above. The displays come in two basic varieties: common-cathode and common-anode. We'll use a common-cathode display to simplify programming. A common cathode display will have nine active pins: eight for the segment LEDs, the ninth that is the cathode (Vss) connection to all LEDs in the module (Note: Some modules will have multiple common pins).

Each of the LEDs in the display is assigned a letter code as illustrated; segment A is the topmost segment (12 o'clock position) with the others being assigned in order as one moves clockwise around the display, ending in segment G in the center, or ending with the decimal point when part of the module.

Connect a seven segment display to an SX28 using the schematic in Figure 8.2. Note that the pin assignments, while seemingly random, actually facilitate the connection of the display when using the SX Tech Board.

When using the Parallax Professional Development Board (PDB) connect the segments as shown, and then connect a single digit control line to Vss. Current-limiting resistors are built into the PDB and do not need to be added externally.



7-Segment LED Display Schematic and Wiring Diagram

Figure 8.2

In earlier programs the `PIN` directive was used with a single I/O point and, as you'll soon see, can also be used with an I/O port (e.g., RA, RB, RC, RD, RE). Enter the following program and save it as `SEGS7.SXB`.

```
' SEGS7.SXB

DEVICE          SX28, OSC4MHZ, TURBO, STACKX, OPTIONX
FREQ            4_000_000

Segments        PIN     RC      OUTPUT


PROGRAM Start

Start:
  '           bafg.cde
  Segments = %10000100                  ' 1
  END
```

Notice the assignment of `Segments`; the output assignment includes all the bits of the RC port, i.e., RC.0 through RC.7. All eight bits are set to outputs by the `PIN` directive. In those

rare cases when you need to set the I/O direction manually, you can do so with the TRIS register for associated port, for example:

```
TRIS_C = %00000000
```

Using the TRIS register in this style is useful when a port has a mixture of input and outputs, and using the multiple `PIN` directives to define them could be tedious.

BASIC Stamp programmers should note that in SX/B, a 0 in the associated bit of the TRIS register puts the pin into output mode; a 1 in the TRIS register puts the pin into *input* mode. This is easy to remember in that a zero looks like the letter "O" (for output), and a one looks like the letter "I" (for input).

After initialization, there is but one active line in the program which activates the LEDs connected to RC.7 (segment B) and RC.2 (segment C) which causes a "1" to be displayed.

After running the program, experiment on your own with the bit pattern assigned to the segments. Can you create additional digits and alpha characters?

Since the static display of a single digit or character could be had without the use of a microcontroller, update the program as follows:

```
' SEGS7a.SXB

DEVICE          SX28, OSC4MHZ, TURBO, STACKX, OPTIONX
FREQ            4_000_000

Segments        PIN     RC      OUTPUT


PROGRAM Start

Start:
  '           bafg.cde
  Segments = %10000100                  ' 1
  PAUSE 1000
  Segments = %11010011                  ' 2
  PAUSE 1000
  Segments = %11010110                  ' 3
  PAUSE 1000
```

```
  GOTO Start
```

This is certainly more interesting than the former version, yet it lacks the flexibility to assign the display as may be required by the program. Update the program again and save as SEGS7b.SXB.

```
' SEGS7b.SXB

DEVICE          SX28, OSC4MHZ, TURBO, STACKX, OPTIONX
FREQ            4_000_000

Segments        PIN     RC      OUTPUT

idx             VAR     Byte


PROGRAM Start

Start:
  FOR idx = 0 TO 9
    READ Dig_Map + idx, Segments
    PAUSE 1000
  NEXT
  GOTO Start


Dig_Map:
  '      bafg.cde
  DATA  %11100111                       ' 0
  DATA  %10000100                       ' 1
  DATA  %11010011                       ' 2
  DATA  %11010110                       ' 3
  DATA  %10110100                       ' 4
  DATA  %01110110                       ' 5
  DATA  %01110111                       ' 6
  DATA  %11000100                       ' 7
  DATA  %11110111                       ' 8
  DATA  %11110110                       ' 9
```

Like most versions of BASIC, SX/B supports a DATA statement to store information for later use. In this version of the program the DATA statement is used to store the bit patterns of the ten decimal digits. Note the use of a label, Dig_Map, ahead of the DATA statements. This is important for the use of information stored with DATA.

The information stored by `DATA` is permanent, somewhat like a constant, but it is not aliased like a constant value. Instead, a value can be retrieved from a set of `DATA` statements knowing the storage address (hence the use of the `Dig_Map` label). And while elements from a `DATA` table can be retrieved during the execution of the program, they cannot be modified (except, of course, at the source code level before compilation).

A value is retrieved from a `DATA` table with `READ` which takes three parameters: the *base address* of the `DATA` table, an *offset index* for the desired element, and an output variable. The base address is provided by the label used to define the table; the offset index is the zero-based position of the element within the table. In this program, the output variable is the segment control pins.

As you can see, then, `FOR-NEXT` is used to iterate idx through the values zero to nine and point to the segment patterns in the table defined at `Dig_Map`. You must be careful not to allow the table offset index to extend beyond the bounds of the table otherwise undefined data will be placed on the outputs.

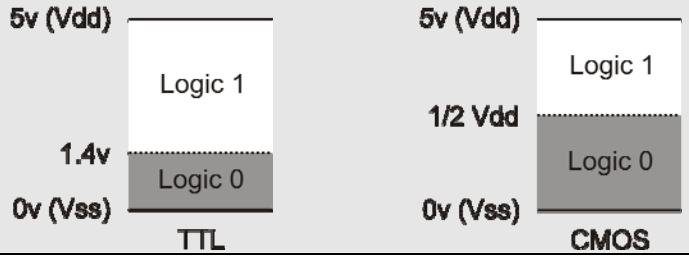
Update the program to display hex digits, 0 through F. Hint: Use lowercase letters for digits B and D to prevent confusion with eight and zero.

9: Digital Input

As you have no doubt become more acutely aware of the number of seven-segment displays enriching your life, you probably noticed that they are nearly always accompanied by some sort of button; to set the time on the alarm clock, for example, or change the channel on the cable television decoder box. These are but a few examples of where push-buttons get used without much notice. Did you "surf" the Internet today? Perhaps as you did you were able to detect the resolute "click" of the pushbutton in your computer's mouse.

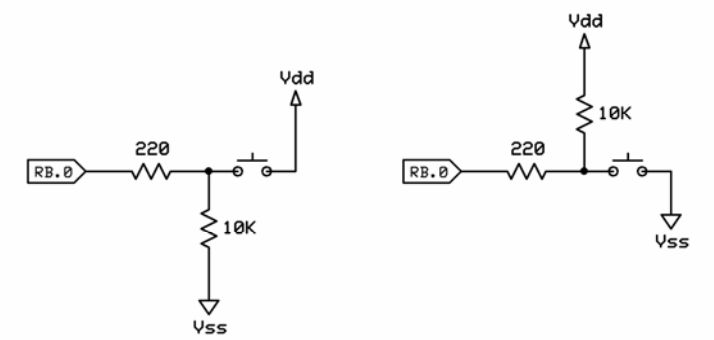
You can connect a push-button or switch to any of the SX's I/O pins. When set to input mode – the default mode after a reset – the current state of a pin can be determined by examining the associated port bit. The bit value, 0 or 1, will be determined by the voltage level present on the input pin, which will usually be zero volts (input bit is 0) or five volts (input bit is 1).

Input pins have a voltage threshold that determines the set-point at which an input transitions from 0-to-1 and back; in fact, this threshold may be configured. The standard set-point, called TTL level, is at 1.4 volts. The optional set-point, called CMOS level, is at one-half Vss – 2.5 volts in most applications.



Digital inputs are very sensitive and can in fact be adversely affected by static electricity near the SX microcontroller. For this reason, when a pin is to be used as a digital input it is best to "pull" the pin to the "off" state with a resistor, usually 10 kΩ, though 4.7 kΩ resistors are also a popular choice. The resistor will hold the I/O pin at the off state until an actual input is present (e.g., a button gets pressed).

The schematics below show the connections of a normally-open push-button in both the active-high (input bit is 1 when pressed) and active-low (input bit is 0 when pressed) modes.



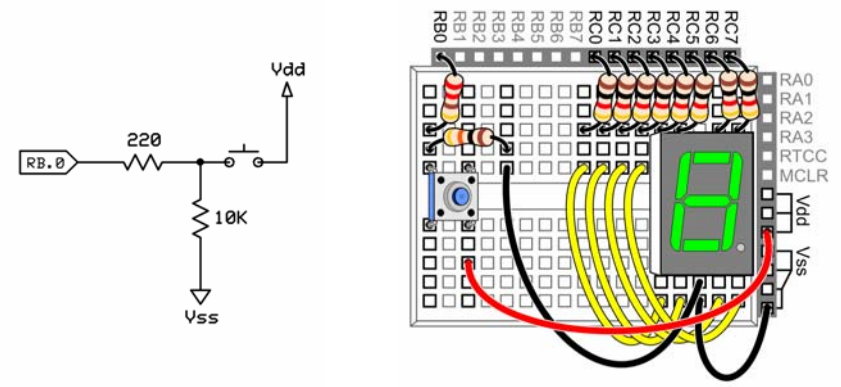
Push-button Schematics: Active-high (left); Active Low (right)

Figure 9.1

DO NOT REMOVE THE PULL-UP OR PULL-DOWN RESISTOR FROM THE CIRCUIT, NOR TRY TO CONNECT MORE THAN ONE I/O PIN TO A SINGLE PULL-UP/PULL-DOWN RESISTOR.

The addition of a small series resistor is also recommended to protect the SX microcontroller from a possible programming error. If the input pin were to be inadvertently made an output and set the opposite level of the expected input state, a button press could create a short circuit that does damage to the I/O pin if there is nothing to limit the current flow through it; the 220-ohm resistor provides this protection.

Connect the active-high button circuit above to the seven-segment display experiment as shown in Figure 9.2, and then enter and save the following program as `PBUTTON.SXB`.



Push-button Circuit Add to the Seven-Segment LED Display Circuit

Figure 9.2

```
' PBUTTON.SXB

DEVICE          SX28, OSC4MHZ, TURBO, STACKX, OPTIONX
FREQ            4_000_000


PButton         PIN     RB.0    INPUT
Segments        PIN     RC      OUTPUT


PROGRAM Start

Start:
  IF PButton = 1 THEN                   ' check input
    Segments = %10000100                ' "1"
  ELSE
    Segments = %11100111                ' "0"
  ENDIF
  GOTO Start
```

When you run the program you'll see that it displays the state of the push-button input pin, RB.0. Note that in the listing, the `PIN` declaration for the push-button uses the `INPUT` modifier; while not strictly necessary in this example, it does add clarity to the listing.

With a bit of modification and the addition of a *counter* variable, the program can be modified to as to make it useful. Modify and save the program as `PBCOUNT1.SXB`.

```
' PBCOUNT1.SXB

DEVICE          SX28, OSC4MHZ, TURBO, STACKX, OPTIONX
FREQ            4_000_000


PButton         PIN     RB.0    INPUT
Segments        PIN     RC      OUTPUT

IsPressed       CON     1               ' for active-high input
IsNotPressed    CON     0

counter         VAR     Byte


PROGRAM Start

Start:
  READ Dig_Map + counter, Segments      ' show counter
  IF PButton = IsPressed THEN           ' check input
    INC counter                         ' update counter
    IF counter = 10 THEN                ' past limit?
      counter = 0                       ' yes, reset
    ENDIF
    PAUSE 250                           ' minimum display time
  ENDIF
  GOTO Start

Dig_Map:
  '      bafg.cde
  DATA  %11100111                       ' 0
  DATA  %10000100                       ' 1
  DATA  %11010011                       ' 2
  DATA  %11010110                       ' 3
  DATA  %10110100                       ' 4
  DATA  %01110110                       ' 5
  DATA  %01110111                       ' 6
  DATA  %11000100                       ' 7
  DATA  %11110111                       ' 8
  DATA  %11110110                       ' 9
```

Run the program and experiment with pressing the button. You'll see that very short presses result in a single-digit change, while holding the button results in a constantly-incrementing count. When it is not desirable to have a running count on a "stuck" button, a small update can be made that allows just one change per button press-and-release cycle.

Modify the core of the program as follows and save the new version as `PBCOUNT2.SXB`.

```
Start:
  READ Dig_Map + counter, Segments      ' show counter
  IF PButton = IsPressed THEN           ' check input
    INC counter                         ' update counter
    IF counter = 10 THEN                ' past limit?
      counter = 0                       ' yes, reset
    ENDIF
    DO
      PAUSE 50
    LOOP UNTIL PButton = IsNotPressed   ' force button release
  ENDIF
  GOTO Start
```

What's new is the modification of the `DO-LOOP` with the [optional] `UNTIL` condition. In this configuration, the statements enclosed in `DO-LOOP` will execute until the condition expressed after `UNTIL` is satisfied. In this version of the program the `DO-LOOP` will continue until the button is not pressed. As this is part of the main loop, the user is forced to press and release the button before seeing the updated count.

Another option of `DO-LOOP` is with `WHILE`, as shown below:

```
  DO
    PAUSE 50
  LOOP WHILE PButton = IsPressed      ' force button release
```

Can you see that this loop is functionally equivalent to that used in the program above?

In the last two examples of `DO-LOOP` the condition test was placed at the end; this configuration forces the contents of the loop to run at least one time. In some cases it is desirable to test the condition before executing the loop contents, as shown in the examples below:

```
  DO UNTIL PButton = IsNotPressed     ' force button release
    PAUSE 50
  LOOP
```

and…

```
  DO WHILE PButton = IsPressed        ' force button release
    PAUSE 50
  LOOP
```

Ready for your first "real" project? It starts on the next page...

10: Application- A Digital Die

With the circuits used in the previous chapters and a small bit of code, you have what it takes to create simple yet full-blown application: a single-digit, digital die. By using a low clock frequency to reduce power consumption, this program is perfectly suited for battery power.

The program DIGI-DIE.SXB is a little longer than what you're accustomed to at this point, so I'll break it down and explain it section-by-section (for full listing see page 71).

```
DEVICE          SX28, OSC1MHZ, TURBO, STACKX, OPTIONX, BOR42
FREQ            1_000_000
```

The header is straightforward, specifying an internal 1 MHz resonator which is plenty fast enough for this little program. Something new has been added: the BOR42 setting. This enables the brownout monitor circuit and sets the low-voltage threshold to 4.2 volts. Should the incoming voltage drop below 4.2 volts the SX go into reset to prevent errant program behavior.

Brownout refers to a condition where the incoming power sags (drops) below its nominal value; this condition is often caused by weak batteries (in battery-powered circuits) or demands on the power supply that exceed its capability.

```
PButton         PIN     RB.0 INPUT
Segments        PIN     RC   OUTPUT

IsPressed       CON     1               ' for active-high input
IsNotPressed    CON     0

pntr            VAR     Byte            ' animation pointer
seed            VAR     Byte            ' random value
die             VAR     Byte            ' die value for this roll
```

The next section holds out pin, constant, and variable definitions. Note that PButton is defined as an input. This is not explicitly necessary as all pins are defined as inputs unless otherwise changed to outs; still the INPUT specification does not harm and can add a bit of clarity for others that review your listings.

```
PROGRAM Start

Start:
  DO
    READ Bug + pntr, Segments           ' update animation
    INC pntr                            ' update pointer
    IF pntr = 6 THEN                    ' reached max position?
      pntr = 0                          '   yes, reset
    ENDIF
    PAUSE 75                            ' set bug speed
    RANDOM seed                         ' tumble random value
  LOOP WHILE PButton = IsNotPressed     ' wait for button
```

The front end of the program runs in a conditional `DO–LOOP` structure and serves two purposes: 1) It creates a simple animation (rotation) in the 7-segment LED to get the user's attention and, 2) It tumbles a variable called *seed* using the `RANDOM` function – this is akin to shaking the die.

The animation is accomplished by moving a pattern from a `DATA` table (at label, `Bug`) into the display. In this case it's very simple, as we just light a single segment in the order A-B-C-D-E-F before starting over. The variable called *pntr* is use as an index into the pattern table. A short `PAUSE` is used to control the speed of the animation. Be careful about making this value too small or too large; if to small, the display will be a blur; if too large, the value of *seed* won't get tumbled very much.

The `RANDOM` function in SX/B is like that of other embedded controllers: it's actually pseudo-random. What this means is that it actually produces a sequence of seemingly random numbers; but given the same input value, the output will be the same. By adding a "human touch," that is, a human has to press the button, true randomness is achieved as it would be impossible for a human to know or keep track of the pseudo-random output values and press the button at just the right moment.

```
  Segments = %00000000                  ' clear display
  DO
  LOOP UNTIL PButton = IsNotPressed     ' force button release
```

Once the user presses the button, the display is cleared to confirm the button press. It's good idea to force the button release before moving to the next section. You probably see this behavior all the time without noticing; the next time you use your computer's mouse, check to see if the selected behavior does not actually take place until the button is released.

```
die = seed // 6                         ' extract value, 0 to 5
INC die                                 ' fix to 1 to 6
READ Dig_Map + die, Segments            ' display digit
PAUSE 1000                              ' minimum display time
```

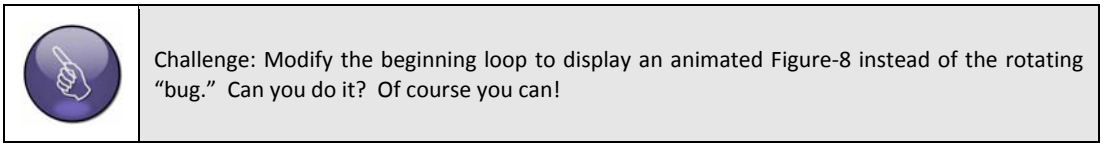
And now the program can create and display the *die* value. Remember that a die has a value from one to six, but the value of *seed* is something between zero and 255. To create the proper value, the modulus operator (`//`) is used. The modulus operator returns the remainder of an integer division which means that the result will always be something between zero and the divisor minus one.

By using a divisor of six the value of *die* is initially set to something between zero and five (based, of course, on the current value of *seed*). The next line increments (adds one to) *die* to get it into the correct value range, and with this value the digit pattern can be read from a `DATA` table (at label, `Dig_Map`) and moved into the display. A one-second `PAUSE` ensures a minimum display duration of the current *die* value.

```
DO
LOOP UNTIL PButton = IsPressed          ' wait for press

DO
LOOP UNTIL PButton = IsNotPressed       ' wait for release

GOTO Start
```

The end of the program waits for the user to press and release the button and then jumps back to the top to start all over.

And there you have it, your first real SX/B application. Now, before you get too excited and move on, why not try to modify the program?



Challenge: Modify the beginning loop to display an animated Figure-8 instead of the rotating "bug." Can you do it? Of course you can!

The complete listing for `DIGI-DIE.SXB` is given below.

```
' DIGI-DIE.SXB

DEVICE          SX28, OSC1MHZ, TURBO, STACKX, OPTIONX, BOR42
FREQ            1_000_000


PButton         PIN     RB.0 INPUT
Segments        PIN     RC   OUTPUT

IsPressed       CON     1               ' for active-high input
IsNotPressed    CON     0

pntr            VAR     Byte            ' animation pointer
seed            VAR     Byte            ' random value
die             VAR     Byte            ' die value for this roll


PROGRAM Start

Start:
  DO
    READ Bug + pntr, Segments           ' update animation
    INC pntr                            ' update pointer
    IF pntr = 6 THEN                    ' reached max position?
      pntr = 0                          '   yes, reset
    ENDIF
    RANDOM seed                         ' tumble random value
    PAUSE 75                            ' set bug speed
  LOOP WHILE PButton = IsNotPressed     ' wait for button

  Segments = %00000000                  ' clear display
  DO
  LOOP UNTIL PButton = IsNotPressed     ' force button release

  die = seed // 6                       ' extract value, 0 to 5
  INC die                               ' fix to 1 to 6
  READ Dig_Map + die, Segments          ' display digit
  PAUSE 1000                            ' minimum display time

  DO
  LOOP UNTIL PButton = IsPressed        ' wait for press

  DO
  LOOP UNTIL PButton = IsNotPressed     ' wait for release

  GOTO Start
```

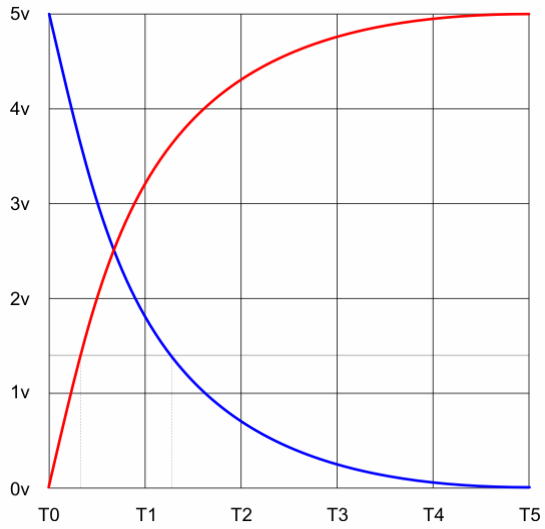
```
Bug:
  '      bafg.cde
  DATA  %01000000                       ' tumbling animation
  DATA  %10000000
  DATA  %00000100
  DATA  %00000010
  DATA  %00000001
  DATA  %00100000


Dig_Map:
  '      bafg.cde
  DATA  %11100111                       ' 0
  DATA  %10000100                       ' 1
  DATA  %11010011                       ' 2
  DATA  %11010110                       ' 3
  DATA  %10110100                       ' 4
  DATA  %01110110                       ' 5
  DATA  %01110111                       ' 6
  DATA  %11000100                       ' 7
  DATA  %11110111                       ' 8
  DATA  %11110110                       ' 9
```

11: Simple Analog Input - Reading a Potentiometer

Thus far, all of our experiments have been purely digital in nature, that is, an input or output was either fully on or fully off. Yet just as between white and black there are many shades of gray, in the analog world in which we live there are an infinite number of levels between ground (zero volts) and five volts.

That said, the SX is a digital device, so how do we deal with analog inputs? Well, with a few simple components, the understanding of RC circuits, and a bit of code, we can deal with one form of analog input: the position of a potentiometer. Let's have a quick look at the behavior of RC circuits.

Figure 11.1 shows the characteristic charge (beginning at 0 V) and discharge (beginning at 5 V) curves of an RC circuit. The physics of RC circuits dictate that a resistor will fully charge or fully discharge in five time-constants, where a time-constant, expressed in seconds, is equal to R × C; with R expressed in ohms and C in farads.



Characteristic RC Circuit Charge and Discharge Curves

Figure 11.1

If a circuit uses a 10 k Ω resistor and a 0.1 µF capacitor, the TC will be one millisecond:

TC = 10,000 Ω × 0.0000001 F = 0.001 s

That it takes a specific, predictable period for the capacitor to charge or discharge is what allows a purely digital device like the SX to measure the RC value. You see, every pin has a threshold voltage where the input switches from 0-to-1 (rising) or 1-to-0 (falling). With the SX chip that voltage is normally set to 1.4 volts (TTL level), but can be changed by the user to ½ Vdd if desired (CMOS level). For these experiments we'll leave the I/O pin in TTL mode.

In the graph above you'll find a line at 1.4 volts; this is the TTL switching threshold. If you look carefully you'll see that the charge curve crosses this line after about 0.3 TCs, and that the discharge curve crosses after about 1.3 TCs. It makes sense that it should take longer to go from 5 volts to 1.4 volts (3.6-volt span) than from 0 volts to 1.4 volts (1.4-volt span). In order to get the greatest resolution from an RC measurement, then, it's best to measure the discharge of the RC circuit.

The charge time of a capacitor, to voltage V_{final}, can be calculated with this equation:

$$T_{charge} = -RC \times \ln(1 - (V_{final} / V_{applied}))$$

Using a 10K resistor and a 0.1 µF capacitor, the charge time from zero volts to the TTL threshold of 1.4 volts works out to:

$$T_{charge} = -0.001 \times \ln(1 - (1.4 / 5.0))$$
$$T_{charge} = -0.001 \times \ln(1 - (0.28))$$
$$T_{charge} = -0.001 \times \ln(0.72)$$
$$T_{charge} = -0.001 \times -0.328$$
$$T_{charge} = 0.328 \text{ milliseconds}$$

The discharge time of a capacitor, from voltage $V_{initial}$ to voltage V_{final} can be calculated with this equation:

$$T_{discharge} = -RC \times \ln(V_{final} / V_{initial}))$$

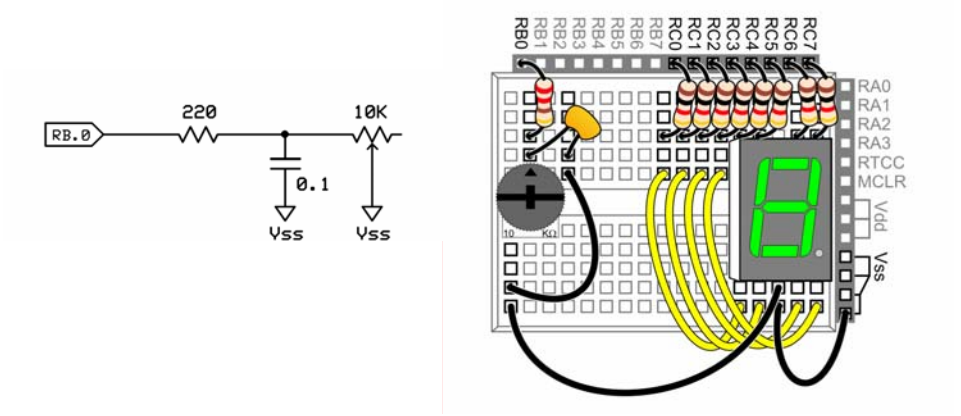
Using a 10K resistor and a 0.1 µF capacitor, the discharge time from five volts to the TTL threshold of 1.4 volts works out to:

$$T_{discharge} = -0.001 \times \ln(1.4 / 5.0)$$
$$T_{discharge} = -0.001 \times \ln(0.28)$$
$$T_{discharge} = -0.001 \times -1.272$$
$$T_{discharge} = 1.272 \text{ milliseconds}$$

Measuring the RC discharge time with the SX chip requires these steps:

1. Make the I/O pin an output and high to charge the capacitor.
2. Hold the program long enough to allow the capacitor to fully charge (5 × TC).
3. Make the I/O pin an input to allow the capacitor to discharge.
4. Keep track of the time elapse until the I/O pin switches from 1 to 0.

Step four is the most involved, and SX/B has a built-in function specifically for this purpose: `RCTIME`. Add the following circuit to the seven-segment LED project:



Potentiometer Circuit added to 7-Segment LCD Display Circuit

Figure 11.2

As you can see, this circuit actually has two resistors, and there is a very good reason: the 220-ohm fixed resistor is designed to protect the SX's I/O pin. In step 1, above, the pin is made an output and high. Assuming no 220-ohm resistor, what would happen if the potentiometer was at its minimum (0 ohm) position and the program was delayed? The pin would short directly to Vss and could be damaged. By inserting a 220-ohm resistor – as we did with the basic button input circuit – we can protect the SX by limiting the current through the I/O pin.

With this circuit the capacitor charge time will be: $5 \times 220\ \Omega \times 0.0000001\ \text{F} = 0.00011$ s, or 110 microseconds. I suggest you double this to 200 microseconds to account for component

tolerances. It's true that 200 microseconds is a very short duration, but SX/B can handle it with its `PAUSEUS` (pause in microseconds) instruction, as shown in the listing that follows.

The following program requires the SX-Key to display the `RCTIME` value using the debug output window. If you're using the SX-Blitz, read the program explanation and then move to the next section that demonstrates moving the `RCTIME` value to a seven-segment display.

```
' RCTIME1.SXB
' -- requires SX-Key for debug mode (use Run or Poll)

DEVICE          SX28, OSC4MHZ, TURBO, STACKX, OPTIONX
FREQ            4_000_000

PotPin          PIN     RB.0

potVal          VAR     Byte


PROGRAM Start

Start:
  DO
    HIGH PotPin                         ' charge the cap
    PAUSEUS 20
    RCTIME PotPin, 1, potVal, 3         ' read pot, 6 uS units

    WATCH potVal                        ' display potVal
    BREAK                               ' update Debug window

  LOOP
```

The definitions section of the program should make sense to you now so we'll focus on the main loop at `Start`. The first line sets the control pin high to charge the capacitor; remember that the `HIGH` instruction automatically sets the specified I/O pin to the output mode.

As with other versions of embedded BASIC, SX/B includes the `INPUT` and `OUTPUT` instructions. These instructions are rarely required, however, and do not need to be used in conjunction with most complex instructions (e.g., RCTIME). To do so will not cause harm, but it will consume code space.

As explained above, the capacitor will charge in about 110 microseconds so a 200 microsecond delay using `PAUSEUS` provides ample time for the capacitor to charge. The real work is done by `RCTIME` which is complex, multi-stage instruction.

`RCTIME` starts by setting the specified pin to input mode as this will allow the capacitor to discharge to Vss through the potentiometer. Next, the output variable is cleared (to zero) and a timer is started. At the end of the timer the output variable is incremented and the I/O pin is tested to see if it has crossed the input voltage threshold or the output variable has rolled-over to zero. If neither event has taken place the timer-test-increment sequence is repeated.

In the `RCTIME` function syntax the value that follows the I/O pin is the pin level when `RCTIME` is active; 1 when using discharge mode, 0 when using charge mode. The default timing unit is two microseconds, but this is not always adequate for a given RC circuit and the size of the output variable.

In this program, *potVal* is a byte, so its range is zero to 255. What we need to do is ensure that at the maximum potentiometer setting the output value will fit into a byte. Here's how we do that: we start by taking our time-constant, one millisecond, and multiplying it by the time required for the RC circuit to go from fully charged to 1.4 volts, a factor of 1.27 (see the info box, above).

So, it should take about 1.27 milliseconds for the capacitor to discharge to the input threshold level. We divide 255 into this value and if the result is exactly two microseconds or less, we can use `RCTIME` without the optional resolution parameter.

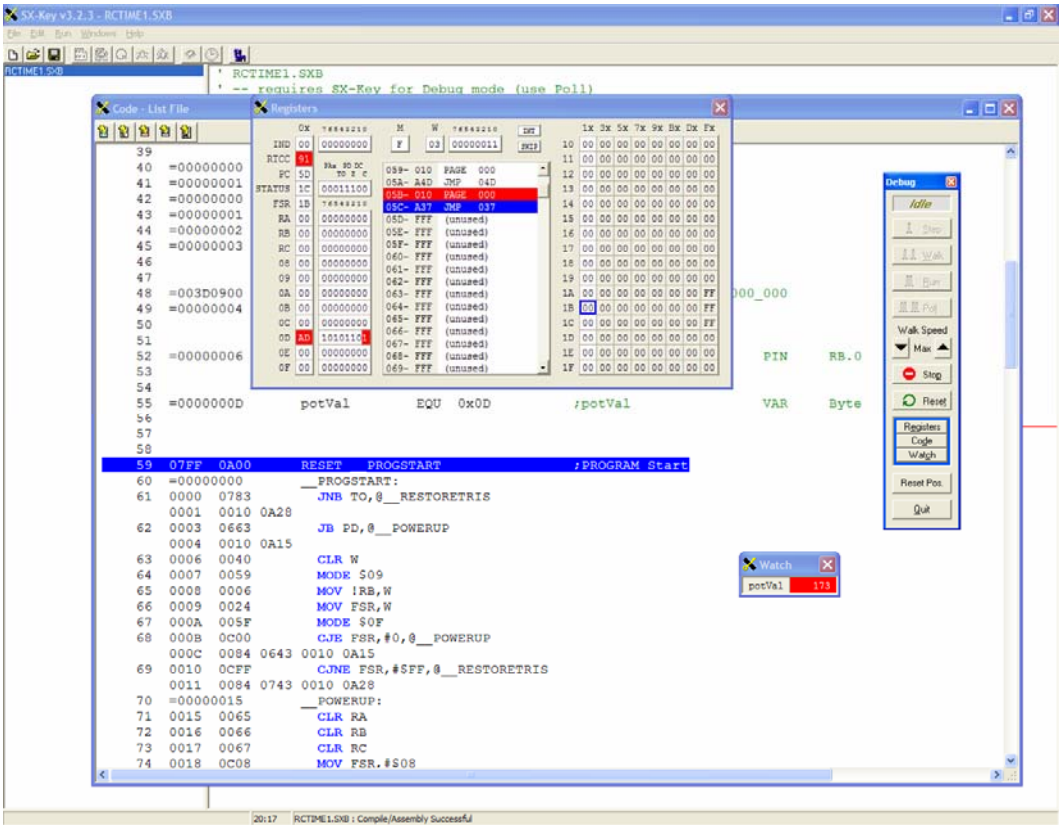
That's not the case here, however. Here's the math:

0.00127 s / 255 = 0.00000498 (4.98 microseconds)

As this value is greater than two microseconds we must divide it by two and round up to the nearest integer. The result is three in the resolution parameter which sets the internal timer for `RCTIME` to six microseconds. With this resolution the theoretical maximum output for this circuit using `RCTIME` is 212 (1.27 milliseconds divided by six microseconds).

How can we see this value as the program operates? Without additional hardware (e.g., serial LCD display), the easiest thing we can do is use the debug capability of the SX-Key IDE, but we can only do this when using the SX-Key (the IDE will not enter debug mode

when using the SX-Blitz). The SX-Key differs from the SX-Blitz in that it can generate the required clock frequency and display the contents of the SX's memory as the program runs.



The SX-Key Debugging Tool Windows

Figure 11.3

To facilitate viewing the value we have inserted two lines into the program:

```
    WATCH potVal                        ' display potVal
    BREAK                               ' update Debug window
```

The `WATCH` directive moves the target variable to its own window so that it's easy to see. The `BREAK` directive stops the program at that point so that we can see the results.

After you've entered and saved the program, compile it and enter debug mode by clicking the Debug toolbar button:

...or click *Run* → *Debug* from the main menu. If there are no errors the program will be downloaded and several new windows will appear on your screen as shown in Figure 11.3.

In the Debug Control Panel, click the *Run* button. The program will run and the value in the Watch window will probably change and be displayed on a red background. Red is used to indicate that the value has changed since the last break. Turn the potentiometer to a new position and click *Run* again.

If you want to see the program in [near] real-time, click the *Poll* button. This runs the program, refreshes the Watch window and display at `BREAK`, and then restarts the program. In polling mode you can watch *potVal* change as you turn the potentiometer.

Display without Debug

Okay… what if you don't have an SX-Key (you're using the SX-Blitz) and can't run the IDE in debug mode – how do you see the value of the potentiometer without resorting to a fancy external display (that you haven't learned to use yet!)?

The answer is simple, if not immediately obvious: we can use the seven-segment display from Chapter 8. Wait a minute, there's only one digit. No problem, we'll display the three digit value, one digit at a time, with a short break in between so we can differentiate one reading from the next. We'll be using a technique similar to the coded LED pulsing, but this time we're displaying three digits in sequence.

Modify the listing as shown below.

```
' RCTIME2.SXB
' -- display in (coded) DEC3 format with 7-segment LED

DEVICE          SX28, OSC4MHZ, TURBO, STACKX, OPTIONX
FREQ            4_000_000

PotPin          PIN     RB.0
Segments        PIN     RC OUTPUT

potVal          VAR     Byte
digit           VAR     Byte
PROGRAM Start

Start:
```

```
  HIGH PotPin                           ' charge the cap
  PAUSEUS 200
  RCTIME PotPin, 1, potVal, 3           ' read pot, 6 uS units

Show_100s:
  digit = potVal / 100                  ' get 100s digit
  potVal = __REMAINDER                  ' save 10s and 1s
  READ Dig_Map + digit, Segments        ' display digit
  PAUSE 350
  Segments = %00000000                  ' blank
  PAUSE 100

Show_10s:
  digit = potVal / 10                   ' get 10s digit
  potVal = __REMAINDER                  ' save 1s
  READ Dig_Map + digit, Segments
  PAUSE 350
  Segments = %00000000
  PAUSE 100

Show_1s:
  digit = potVal                        ' get 1s digit
  READ Dig_Map + digit, Segments
  PAUSE 350
  Segments = %00000000
  PAUSE 500                             ' delay between readings
  GOTO Start

Dig_Map:
  '      bafg.cde
  DATA  %11100111                       ' 0
  DATA  %10000100                       ' 1
  DATA  %11010011                       ' 2
  DATA  %11010110                       ' 3
  DATA  %10110100                       ' 4
  DATA  %01110110                       ' 5
  DATA  %01110111                       ' 6
  DATA  %11000100                       ' 7
  DATA  %11110111                       ' 8
  DATA  %11110110                       ' 9
```

After reading the RC circuit we extract the hundreds digit from *potVal* by dividing by 100. SX/B does only integer division and anything left over is put into an internal variable called *__REMAINDER*; at this point *__REMAINDER* will hold the tens and ones values so we copy that back into *potVal* for the sections that follow.

The value of *digit* is used as an index into the seven-segment table and `READ` is used to update the LED display. We hold the display with `PAUSE` and then clear it briefly. The clearing step is very important as it lets us differentiate each digit.

After the blanking period the process is repeated for the tens and the ones, but we don't have to do any dividing to get the ones because the remainder of a division by 10 will always be something between zero and nine.

The final blanking space is extended to 500 milliseconds. This provides a clear indication that what's being displayed is a new value.

Update the program to display the potentiometer value in hexadecimal format. Hint: you only need two digits, and the divisor is 16. For those with a little programming experience, see if you can find a way to show the number without using division.

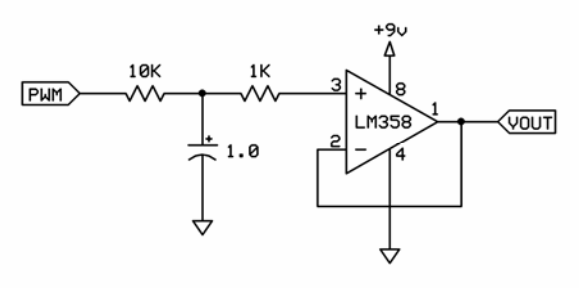
12: Simple Analog Output - Modulating an LED

The last experiment demonstrated that with a small amount of code and simple support components, a digital device like the SX can deal with the world of analog input (one form, we'll see others later). The same holds true for analog output, specifically an analog voltage from a digital output pin. This is accomplished with the `PWM` instruction.

Since the SX/B `PWM` instruction is designed to charge an RC circuit it behaves differently than you might expect. If we look at the raw output of the instruction we'll see what sometimes looks like a random stream of 1's and 0's. What's happening, though, is that the ratio of 1's to 0's in the stream – when averaged over the Duration – matches the ratio of Duty / 255. Since the full output is five volts we can calculate Duty to provide a specific output voltage with PWM using this formula:

$$\text{Duty} = (\text{Volts} / 5.0) \times 255$$

Figure 12.1 shows a simple circuit that allows the SX to provide a useful output voltage (0 to 5) when using the PWM instruction.



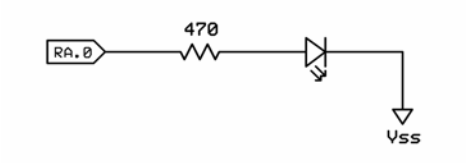
LM358 Op-Am Schematic

Figure 12.1

In order to get 5.0 volts from the LM358, its supply voltage should be at least 6.5 volts. Refer to the LM358 datasheet for details.

What really counts here is the RC structure connected to the output pin. By holding a charge the modulated output from `PWM` is converted to a DC level. The op-amp simply buffers the RC components so that they are not swamped by an applied load.

We can in fact demonstrate the `PWM` instruction with a much simpler circuit – one that allows us to use built-in test equipment.



LED Circuit

Figure 12.2

Have you guessed what test equipment will be used? Our eyes, of course! You see, when the `PWM` instruction is applied to an LED, the modulation of the LED is so fast that our eyes will integrate (through persistence of vision) the pulses into an apparent brightness level. If we put `PWM` into a loop we can modulate the brightness; instead of blinking the LED appears to slowly pulsate.

```
' PWM.SXB

DEVICE          SX28, OSC4MHZ, TURBO, STACKX, OPTIONX
FREQ            4_000_000

Led             PIN     RB.0

duty            VAR     Byte


PROGRAM Start

Start:
  DO
    FOR duty = 0 TO 254                    ' off to bright
      PWM Led, duty, 2                     ' modulate the LED
    NEXT
    FOR duty = 255 TO 1 STEP -1               ' bright to off
      PWM Led, duty, 2
    NEXT
    PAUSE 50
  LOOP
```

As you can see there are two FOR-NEXT loops that handle the brightening and dimming phases of the modulation, and that the maximum output is at 255 (this is the 100% point where the output stream contains only 1's). For the dimming loop we're forced to use a STEP value of negative 1 so that the loop will count backward. Without a negative STEP the FOR-NEXT loop will terminate prematurely because the starting value is greater than the terminating value. If your LED appears to go from dark slowly to bright, then immediately back to dark, you've left out the STEP parameter of the second FOR-NEXT loop. This is easy to do, and something you'll want to watch for when using FOR-NEXT to count backward.

It's important to understand that the SX/B PWM instruction works differently from the PWM process used by DC motor controllers. Motor control circuits typically use a fixed frequency and vary the duty cycle of the output to change the motor speed. The PWM instruction, on the other hand, does not have a perfectly-fixed period within a single on-off cycle of the output pin. The output, however, when averaged over the time specified in the Duration parameter, does attain the specified duty cycle.

Internally, a byte-sized accumulator is kept in which the Duty value is constantly added during the course of the Duration. When this accumulator rolls-over (exceeds 255), the output is made high, otherwise it is made low. If the Duty was set to 115 (45%), for example, we would see the following values in the accumulator at the start of the instruction.

Accumulator Value	PWM Pin
0	Low
15	Low
230	Low
345(89)	High(overflow)
204	Low
319(63)	High(overflow)

Table 12.1
Accumulator Values

Even with this short list you can see that the output is not symmetrical, and yet if you continue the list it will average out to a duty-cycle of 45% over time. If you watch the raw output on an oscilloscope the wave-form will appear to bounce-around a bit; this is due to the asymmetrical roll-over behavior of the accumulator. This style of PWM may not be the best for controlling motor speed, but it works very well for generating an analog voltage and is extremely code efficient.

If you need to use the op-amp circuit above to create a "stiff" voltage, be sure to set the Duration parameter to match the RC charge time. In the circuit above the RC time-constant is 10 milliseconds (10,000 × 0.000001 = 0.01), so the Duration parameter of `PWM` should be set to 10 or greater to ensure that the capacitor will be fully changed when the Duty parameter is set to 255 (100%). Remember, it takes five time constants to charge a capacitor to the applied voltage.

Finally, even with the op-amp buffer, there will be some leakage from the capacitor so the `PWM` instruction will need to be called periodically to hold the output at the desired level.

Part II:Practical Programming

13: Using a Template and Programming with Style

As you study the habits of professional programmers, those individuals who make a living by writing code day-in and day-out, you will find that they tend to be very organized, especially when it comes to their source code listings. The reason is simple: why expend creative energy on formatting and programming style when that energy is best targeted at the problem to be solved?

Note that I used the term "creative energy" here. I used this term deliberately because programming is both art and science. What it comes down to is that I sincerely believe you will be well-served by creating or adopting a programming style and then just sticking with it. In this section I'm going to share my template and my style guidelines. Now, I cannot claim to have created all of this myself; this is the result of many years of programming various machines and in a variety of languages, and mostly from a lot of great input from many highly-capable sources.

An interesting note is that I can usually recognize my own code where it has been incorporated into another's program, because my programming style is so consistent. I view this as a good thing, as my programming habits tend to follow the way I think so when I do find my code elsewhere I can usually figure out what is going on without a lot of stress. Adopt the template and style as-is, or modify it so that it better fits you and your needs; the key is to make things comfortable so that you can simply focus on the logic of your program, not the mechanical execution of entering it. Let's have a look at what I consider a nice SX/B programming template:

```
' ============================================================================
'
'   File......
'   Purpose...
'   Author....
'   E-mail....
'   Started...
'   Updated...
'
' ============================================================================


' ----------------------------------------------------------------------------
' Program Description
' ----------------------------------------------------------------------------
```

```
' -----------------------------------------------------------------------------
' Conditional Compilation Symbols
' -----------------------------------------------------------------------------


' -----------------------------------------------------------------------------
' Device Settings
' -----------------------------------------------------------------------------

DEVICE          SX28, OSCXT2, TURBO, STACKX, OPTIONX, BOR42
FREQ            20_000_000
ID              "Name"


' -----------------------------------------------------------------------------
' I/O Pins
' -----------------------------------------------------------------------------


' -----------------------------------------------------------------------------
' Constants
' -----------------------------------------------------------------------------


' -----------------------------------------------------------------------------
' Variables
' -----------------------------------------------------------------------------


' =============================================================================
' INTERRUPT
' =============================================================================

' RETURNINT


' =============================================================================
  PROGRAM Start
' =============================================================================


' -----------------------------------------------------------------------------
' Subroutine / Function Declarations
' -----------------------------------------------------------------------------
```

```
' -----------------------------------------------------------------------
' Program Code
' -----------------------------------------------------------------------

Start:


Main:

  GOTO Main

' -----------------------------------------------------------------------
' Subroutine / Function Code
' -----------------------------------------------------------------------


' -----------------------------------------------------------------------


' -----------------------------------------------------------------------
' User Data
' -----------------------------------------------------------------------
```

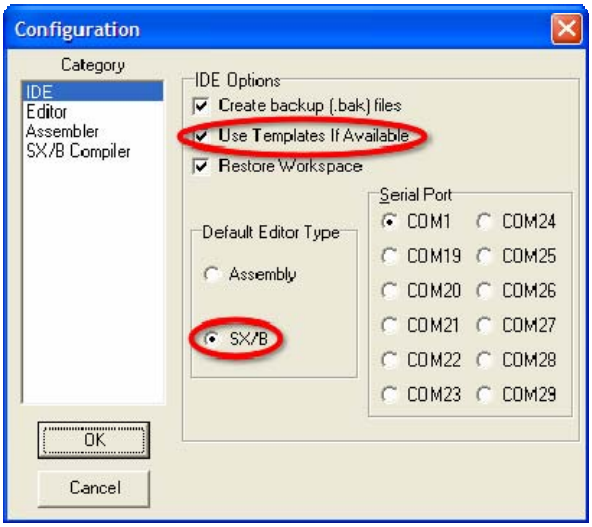
What you should immediately notice is that everything is sectionalized, providing a bit of roadmap to program construction. Where the order of things does matter, that is already taken care of (e.g., the **INTERRUPT** section must appear before other code). Of course, we won't and certainly don't have to use every section, but for a template it is best to be all-inclusive and then remove those sections that are not required when the program has been completed and is ready for distribution. Or, if you expect a program will evolve, you can simply leave them in place as they have no affect on the compiled output.

Another nice thing about using a template is that we don't have to remember mundane details like the elements of the program header, those items in the **DEVICE** directive. And do note that the **INTERRUPT** section is there, but "commented out." We do this because an empty interrupt still interrupts the program and may cause the program to behave in ways we don't want or expect. When we do need to have an interrupt in our program, however, the template has it for us, in the correct place, and all we have to do is remove the comment markers.

The great news is that the SX/Key IDE supports templates so we don't have to open the file and save it to a new name; most of this process is automated. Simply take the template file (above) and copy it to the **Template** folder of the IDE. On my system, that's located at:

C:\Program Files\Parallax Inc\SX-Key v3.2.3\Templates

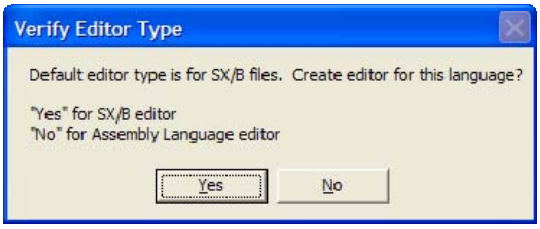
To enable the template you'll need to go back into the Configuration dialog by clicking **Run → Configure** on the menu. In the **Configuration/IDE** page, click on **Use Templates If Available** and make sure the **Default Editor Type** is set to **SX/B**.



- Select **Use Templates If Available**
- Select **SX/B** as the **Default Editor Type**

Figure 13.1

Click on **OK** to exit the dialog. Now, when you click on **File → New** your template will be loaded and ready to use. If you click on the **New** toolbar button you'll get the following dialog to verify the editor type:



Verify Editor Type Dialog

Figure 13.2

Just click on **Yes** and your SX/B template will be loaded into the IDE.

A Matter of Style

As I mentioned earlier, programming style is a very big thing in the professional world. In fact, many organizations write entire manuals on programming style so that they can use contract programmers and still end up with code that gracefully integrates into the code written by their staff.

I don't think we need a whole book, but a few guidelines will make your listings easier to read, and therefore easier to debug. Yes, we would all like to think that we write perfect programs 100% of the time, but that is just not the case – nor will it ever be. It is my contention, then, that a neat, orderly style will make tracking down "bugs" (human errors) in your programs a whole lot easier. Why do I think this way? Because most of the problems I've created myself were not errors in logic, they were typos created out of laziness and not sticking to my own guidelines.

For new programmers some of the terms used below may seem a bit foreign – don't worry about that. The chapters that follow will explain everything (like subroutines, functions, etc.) in a great deal of detail.

I ask you to consider these things when writing your SX/B programs:

1. Do It Correctly the First Time

Many programmers, especially new ones, fall into the "I'll knock it out now and fix it later." trap. Invariably, the "fix it later" part never happens and then sloppy code makes its way into production projects. If you don't have time to do it correctly now, when will you find time to do it again?

Start clean and you'll be less likely to introduce errors into your programs. And if errors do pop up, clean and organized formatting will make them easier to find and fix.

2. Be Organized and Consistent

Using a standard template (see above) will help you organize your programs and establish a consistent presentation. The SX-Key IDE allows you to specify a template for the **File → New (SX/B)** menu option.

3. Use Meaningful Names

Be verbose when naming constants, variables and program labels. The compiler will allow names up to 32 characters long. Using meaningful names will reduce the number of comments and make your programs easier to read, debug, and maintain.

4. Naming I/O Pins

Begin I/O pin names with an uppercase letter and use mixed case, with uppercase letters at the beginning of new words within the name. For example,

```
HeaterCtrl      PIN     RA.0
```

Since connections don't change during the program run, I/O pins are named like constants (#5) using mixed case, beginning with an uppercase letter. Resist the temptation to use SX pin names (e.g., RB.7) in the body of a program as this can lead to errors when making circuit changes.

5. Naming Constants

Begin constant names with an uppercase letter and use mixed case, with uppercase letters at the beginning of new words within the name.

```
AlarmCode       CON     25
```

6. Naming Variables

Begin variable names with a lowercase letter and use mixed case, with uppercase letters at the beginning of new words within the name.

```
waterLevel      VAR     Byte
tally           VAR     Word
```

7. Variable Type Declarations

SX/B supports word, byte, byte array, and bit variables. To define bit variables, the byte or word that holds them must be defined first.

```
sysCount        VAR     Word
alarms          VAR     Byte
overTemp        VAR     alarms.0
underTemp       VAR     alarms.1
clock           VAR     Byte(3)
```

8. General Program Labels

Begin program labels with an uppercase letter, separate words within the label with an underscore character, and begin new words with a number or uppercase letter. Labels should be preceded by at least one blank line, begin in column one, and must be terminated with a colon. Note that program labels are not followed with a colon when used as part of a command, such as with **GOTO**, below.

```
Blink_Led:
  Leds = %00000001
  DO
    PAUSE 100
    Leds = Leds << 1
  LOOP UNTIL Leds = %00000000
  GOTO Blink_Led
```

9. SX/B Keywords

All SX/B language keywords, including **CON**, **VAR**, and **SUB** should be uppercase. The SX-Key IDE does syntax highlighting, but does not change case so this is the responsibility of the programmer.

```
Main:
  DO
    HIGH AlarmLed
    WAIT_MS 100
    LOW AlarmLed
    WAIT_MS 100
  LOOP
```

10. Declare Subroutines and Functions

Declared subroutines were introduced in SX/B version 1.2, and as of version 1.5, SX/B supports functions as well. Functions allow a routine to return byte or word values, while subroutines are limited to returning a single byte. Good programming practice suggests using a function when the normal behavior is to return a value.

Declared subroutines and functions benefit the programmer in two ways: 1) the compiler creates a jump table that allows the subroutine code to be placed anywhere in the program space and, 2) the compiler does a syntax check on the subroutine call to ensure that the proper number of parameters are being passed.

```
DELAY_MS        SUB     1, 2
GET_TEMP        FUNC    2
```

When using a declared subroutine, the use of **GOSUB** is not required.

```
Main:
  DO
    HIGH AlarmLed
    DELAY_MS 100
    LOW AlarmLed
    DELAY_MS 100
  LOOP
```

Note that when a subroutine or function does not require any parameters to be passed to it you should declare zero parameters, like this:

```
SOUND_ALARM     SUB     0
RX_BYTE         FUNC    1, 0
```

In the RX_BYTE function declaration, above, there are no parameters passed but one byte is returned.

11. Subroutine and Function Labels

Subroutines and functions are reusable sections of code that go a long way to conserving SX program space and making our listings easier to read and manage. For subroutine and function labels use all uppercase letters, separating words within the label with an underscore character. Note that in use, subroutine and function labels will be preceded with **SUB** or **FUNC** and do not require a colon; the code block will be terminated with **ENDSUB** or **ENDFUNC**. Using this format makes your SX/B programs compatible with planned updates of SX/B, so it's best to adopt this style now.

```
' Use: DELAY_MS ms
' -- 'ms' is delay in milliseconds, 1 - 65535

SUB DELAY_MS
  IF __PARAMCNT = 1 THEN
    tmpW1 = __PARAM1                            ' save byte value
  ELSE
    tmpW1 = __WPARAM12                          ' save word value
  ENDIF
  PAUSE tmpW1
  ENDSUB

' Use: result = RX_BYTE
' -- returns one byte from serial input

FUNC RX_BYTE
  SERIN RX, Baud, tmpB1           ' rx one byte
```

```
  RETURN tmpB1
  ENDFUNC
```

As shown above, it is good practice to document the subroutine with usage requirements, especially when optional parameter(s) may be used.

12. Indent Nested Code

Nesting blocks of code improves readability and helps reduce the introduction of errors. Indenting each level with two spaces is recommended to make the code readable without taking up too much space on any line when you have several levels of indenting.

```
' Use: LCD_OUT [ aByte | string | label ]
' -- "aByte" is single-byte constant or variable
' -- "string" is an embedded literal string
' -- "label" is DATA statement label for stored z-String

SUB LCD_OUT
..tmpB1 = __PARAM1
..IF __PARAMCNT = 2 THEN
....tmpW1 = __WPARAM12
....DO
......READINC tmpW1, tmpB1
......IF tmpB1 = 0 THEN EXIT
......SEROUT LcdTx, LcdBaud, tmpB1
....LOOP
..ELSE
....SEROUT LcdTx, LcdBaud, tmpB1
..ENDIF
..ENDSUB
```

Note: The dots are used to illustrate the level of nesting and are not a part of the code.

14: Divide, Conquer, and Rule!

While the chapter title sounds like the advice given by some great general to a commander about to be sent into battle, this is in fact my admonition to you regarding your SX/B programs. It's easy for us to feel overwhelmed by the scope of a big project, and the best way to get past that feeling is to use the *elephant rule*:

Q: How does one eat an elephant?
A: One bite at a time.

When you consider it, a large program is nothing more than a collection of related lines of code, and when we're especially practical there will be groups of lines within the program that will serve a specific purpose and that can (and should) be reused. We call these groups *subroutines* and *functions*. The use of subroutines and functions allows us to divide a big program into smaller, more manageable modules.

If you have any programming background at all you've probably heard the terms but may not be clear on what distinguishes one from the other. The difference is subtle: subroutines do some process; functions do some process and return one or more values. The truth is that subroutines can return a single value, but we don't often employ this ability, and it is good form to define a module that can return one or more values as a function.

The use of subroutines and functions serves two major purposes in SX/B: 1) they allow us to extend and customize the language to our needs and, 2) they allow us to encapsulate SX/B instructions that generate a lot of assembly code into a single location, thus saving the program space requirements of our application.

Let's look at an example using **PAUSE**. Now, **PAUSE** doesn't generate a lot of assembly code but it is used so frequently within embedded applications that we can save program space by encapsulating it in a subroutine. It would be nice, too, if the subroutine could work with byte or word values. One of the great things about SX/B is that it includes a mechanism for passing values (called parameters) to subroutines, and the compiler will even check our code to ensure that we're using the subroutine correctly.

Declaring and Using Subroutines

Everything starts with the declaration. For subroutines the declaration syntax is as follows:

```
NAME    SUB     Min_Bytes {, Max_Bytes }
```

The declaration starts with the name of the subroutine; per convention the subroutine name will be in all caps and with an underscore to separate words within the name. This is followed by the **SUB** identifier and the parameter count list. The parameter count list tells how many bytes, if any, will be passed to the subroutine. Not all subroutines will require parameters and when they don't we should define the *Min_Bytes* value as zero. If a subroutine can handle a variable number of parameter bytes, the first value in the list is the minimum number of bytes expected, the second is the maximum.

I tend to use the name DELAY_MS for the subroutine that encapsulates **PAUSE**. And here's how that subroutine is declared in an SX/B program:

```
DELAY_MS        SUB     1, 2                    ' delay in milliseconds
```

In the declaration for DELAY_MS we can see that the subroutine expects at least one byte and can accept two. If we try using it with zero or more than two bytes, the compiler will flag an error as we've violated our own definition of the subroutine's interface.

The mechanism for moving values from our main program to the subroutine is through a set of reserved variables. In the global RAM space (see Chapter 15: Using and Managing Variable Space in SX/B) there are five bytes of RAM used for parameter passing to and from subroutines and functions. Groups of two bytes can be combined to form word values as shown in the table below:

<table>
<tr><th>Byte</th><th colspan="2">Word</th></tr>
<tr><td>__PARAM1</td><td rowspan="2">__WPARAM12</td><td></td></tr>
<tr><td>__PARAM2</td><td rowspan="2">__WPARAM23</td></tr>
<tr><td>__PARAM3</td><td rowspan="2">__WPARAM34</td></tr>
<tr><td>__PARAM4</td><td></td></tr>
<tr><td>__PARAMCNT/__PARAM5</td><td></td><td></td></tr>
</table>

Table 14.1
Parameter-passing RAM

As you can see, there are four parameter bytes (which can be grouped into words) and a byte that passes the parameter byte count to the subroutine or function. For those subroutines and functions that use a fixed number of parameters the *__PARAMCNT* byte may be used as *__PARAM5* if needed.

Let's work through the DELAY_MS subroutine before exploring the intricate details of mixing parameter types. Here's the listing:

```
' Use: DELAY_MS ms
' -- 'ms' is delay in milliseconds, 1 - 65535

SUB DELAY_MS
  IF __PARAMCNT = 1 THEN
    tmpW1 = __PARAM1                          ' save byte value
  ELSE
    tmpW1 = __WPARAM12                        ' save word value
  ENDIF
  PAUSE tmpW1
  ENDSUB
```

The subroutine starts with the **SUB** definition and the name from the previous declaration, DELAY_MS. Note that the subroutine name is not followed by a colon as is the case with standard program labels. The first operational line of code examines *__PARAMCNT* to see how many bytes were passed; if one byte was passed then *__PARAM1* is copied into *tmpW1*; if two bytes were passed (indicating a word value) then *__WPARAM12* is copied into *tmpW1*. We don't have to worry about moving the values into *__PARAM1* or *__WPARAM12*, the compiler handles that part for us – we just need to know that a byte will show up in *__PARAM1* and a word in *__WPARAM12*.

SX/B uses the *__PARAMx* and *__WPARAMx* variables internally so the first thing a subroutine or function should do is copy any passed values to other variables. Failing to copy passed values may result in their corruption by the use of an SX/B instruction.

Once we've captured the value that was passed the rest is easy; we simply use that value with **PAUSE**. Again, the reason for doing this is that **PAUSE** is compiled into a single location and this saves space versus using **PAUSE** at multiple locations in our program.

Once the subroutine is declared and coded its use is straightforward – just like a new keyword:

```
  DELAY_MS 100
```

...or:

```
  DELAY_MS 2500
```

In the first example the value of 100 will be put into *__PARAM1* and *__PARAMCNT* will be set to one before jumping to `MS_DELAY` because a byte will hold that value. In the second example our value exceeds the limits of a single byte so the value of 2500 will be put into *__WPARAM12* (which is constructed from *__PARAM1* and *__PARAM2*) and *__PARAMCNT* will be set to two before calling `MS_DELAY`. When `MS_DELAY` is complete and the **ENDSUB** instruction is executed the program will return to the line that follows the call to the subroutine.

Declaring and Using Functions

A function is very much like a subroutine except that by its design it will return one or more bytes to the caller. The syntax for a function declaration is:

```
Name    FUNC    Return_Bytes {, Min_Bytes {, Max_Bytes }}
```

You can see that with functions there may be up to three values in the parameters definition list. The first is how many bytes will be returned to the caller. The second is the minimum number of bytes, if any, that will be passed to the function, the final, if used, is the maximum number of bytes that will be passed to the function.

It should be clear that the parameters list for subroutines and functions specify the number of bytes that are passed, not the number of values. If, for example a subroutine or function requires a byte and a word to be passed to it, the minimum parameter count value will be three.

Let's have a look at a simple function that will be useful in many embedded applications

```
RX_BYTE         FUNC    1, 0, 1                 ' receive a byte
```

By this declaration you can see that the function called `RX_BYTE` will return one byte to the caller and may or may not be passed a single byte parameter. The purpose of `RX_BYTE` will be to receive a byte from a serial device like a PC, modem, or even another processor – perhaps a master controller in a multi-processor system. The SX/B function that receives a serial byte is called **SERIN**. **SERIN** is a complex function and generates a lot of assembly code relative to other SX/B instructions, so it makes good sense to package it into a custom function. And by doing this we can even modify the **SERIN** return value if desired.

Here's the custom RX_BYTE function:

```
' Use: theByte = RX_BYTE { ConvertToUpper }
' -- converts "a".."z" to "A".."Z" if "ConvertToUpper" bit 0 is 1

FUNC RX_BYTE
  IF __PARAMCNT = 1 THEN                        ' option specified
    tmpB2 = __PARAM1                            ' yes, save it
  ELSE
    tmpB2 = 0                                   ' no, set to default
  ENDIF
  SERIN RX, Baud, tmpB1                         ' receive a serial byte
  IF tmpB2.0 = 1 THEN                           ' convert to uppercase?
    IF tmpB1 >= "a" THEN                        ' lowercase?
      IF tmpB1 <= "z" THEN
        tmpB1.5 = 0                             ' ...yes, make uppercase
      ENDIF
    ENDIF
  ENDIF
  RETURN tmpB1
  ENDFUNC
```

As you'd expect a function begins with the **FUNC** declaration followed by the function name, RX_BYTE (again, without a trailing colon). And as we did with the subroutine earlier, the first order of business is to check for parameters that have been passed. With RX_BYTE you may or may not get a parameter. If one is passed that value is copied into *tmpB2*, otherwise *tmpB2* is cleared to zero.

The **SERIN** keyword is used just as it would be in any other program, specifying the receive pin, a baud rate/mode constant, and a return variable for the value, in this case *tmpB1* is used for the return value. The reason that a parameter could be passed to RX_BYTE is that this will allow the function to convert a lowercase letter to uppercase if desired. This is very useful in command processor applications where the master device may send a command as uppercase or lowercase (this is especially true when the "master" is a terminal program being run by a human that doesn't make the same distinction between "a" and "A" that a microcontroller does). By converting letters to uppercase the command processing code of the application is simplified.

After the serial byte has been received the code checks to see if bit 0 of *tmpB2* (which holds the parameter, or zero if none) is 1. If it is, then the received byte is checked to see if it is a lowercase letter. If both conditions are met the letter is converted to uppercase by clearing bit 5 of its [ASCII] value. Finally, **RETURN** is used to send the value of *tmpB1* back to the caller.

To use RX_BYTE without case conversion we would do this:

```
  cmd = RX_BYTE
```

… where *cmd* is a byte variable. To convert *cmd* to uppercase when a letter is received we would modify the call like this:

```
  cmd = RX_BYTE 1
```

What's wrong with this line of code? Nothing, from a technical standpoint. It will in fact work, but it's not very friendly or elegant. We can fix that with a constant definition:

```
Uppercase       CON     1
```

And now the call to RX_BYTE can be changed to something far more user friendly:

```
  cmd = RX_BYTE Uppercase
```

Creating Cooperative Subroutines and Functions

In the course of many applications you will have the opportunity to call a subroutine or function from another and you'll want to design them such that they can cooperate with each other in the manner in which they use variables.

In the example above we saw how the RX_BYTE function was created to put a wrapper around SX/B's **SERIN** function. The complement to **SERIN** is of course called **SEROUT** and it is used to transmit a serial byte to another device. Like **SERIN**, the **SEROUT** instruction generates a lot of assembly code and should be wrapped in a custom subroutine to conserve program space.

Here's the declaration for TX_BYTE which is used to transmit a serial byte (on a known I/O pin and fixed baud rate).

```
TX_BYTE         SUB     1                       ' transmit a byte
```

And here's the subroutine code that corresponds to it:

```
' Use: TX_BYTE theByte

SUB TX_BYTE
  tmpB1 = __PARAM1
  SEROUT TX, Baud, tmpB1
  ENDSUB
```

This subroutine is very simple; the parameter passed to `TX_BYTE` is copied into *tmpB1* and that variable is used with SX/B's **SEROUT** instruction. We can get double-duty out of this subroutine by defining another routine to transmit strings of characters. For this we will add a new subroutine declaration to the program:

```
TX_STR          SUB     2                       ' transmit a z-string
```

The reason that this subroutine requires two bytes is that we will actually pass an address pointer to the beginning of the string. This can come in two forms: an inline (embedded) string or in a **DATA** statement (see chapter XX for more details on **DATA** and **WDATA**). The inline use would look like this:

```
  TX_STR "SX/B Rocks!"
```

This code will compile to the string of characters between the quotes and be followed by a zero, hence the term *z-string* (zero-terminated string); the address of the inline string is passed to `TX_STR`. As you'll see in just a moment the zero terminator is critical to the operation of the `TX_STR` subroutine.

To put a string in a **DATA** statement you would do this:

```
Msg:
  DATA  "SX/B Rocks!", 0
```

Note the trailing zero – again, this is critical and not added automatically as is the case with embedded strings. To use this string the call to `TX_STR` is as follows:

```
  TX_STR Msg
```

The compiler resolves the label, `Msg`, to an address value in the SX's program space. The advantage of the second method is that it saves program space in the SX if the same string is transmitted from different locations in the code. Okay, let's look at the code for sending a string:

```
' Use: TX_STR [String | Label]
' -- pass embedded string literal or DATA label (zString at label)

SUB TX_STR
  tmpW1 = __WPARAM12                            ' get address of string
  DO
    READINC tmpW1, tmpB1                        ' read a character
    IF tmpB1 = 0 THEN EXIT                      ' if 0, string complete
    TX_BYTE tmpB1                               ' send the byte
  LOOP
  ENDSUB
```

With a two-byte (word) address being passed to the subroutine the compiler will put this value into *__WPARAM12*; our subroutine will copy this into *tmpW1* for use in our code. The subroutine enters a loop using **READINC** to pull a character from memory at the address now specified in *tmpW1*. After the read operation the value of *tmpW1* is automatically incremented which causes it to point to the next character.

The current character is now in *tmpB1* and compared against zero; if it is zero we have reached the end of the string and the loop is terminated with **EXIT**. If not zero, the byte is passed to TX_BYTE in *tmpB1* and the loop continues to process the next character. Remember that TX_BYTE is going to capture the parameter sent to it and place it in *tmpB1* for transmission via **SEROUT**.

What's really happening, then, is that the character is moving from *tmpB1* to *__PARAM1* and then back into *tmpB1*. There is no elegant way around this small redundancy and at SX speeds it will not affect the operation of the program. By designing two cooperative subroutines like this we get the greatest flexibility while using the smallest amount of variable space.

Mixing Parameter Values

There will come a point when an application requires multiple values be passed to a subroutine or function and that those values can be either bytes or words. There is no problem in doing this, yet there is one caveat: you must know the design of your subroutine or function so that when you pass values of mixed types you pass them in the correct order.

Let's use multiplication as an example. The * operator seems somewhat innocuous and yet generates a fair amount of assembly code, so let's package this operator into a custom function called MULT. For the greatest flexibility the function should allow two bytes, a byte and a word, or two words to be multiplied. It's that second option that is the tricky part: we need to know the order in which mixed-type values are passed. I tend to favor the *small value first* approach so the when a byte and word are to be multiplied, the byte will be the first parameter.

Since the MULT function will return a word value the declaration is as follows:

```
MULT            FUNC    2, 2, 4                 ' multiply two values
```

To handle the options available to the MULT function we'll use a complex **IF-THEN** structure to move the various parameters to the correct working variables.

```
' Use: result = MULT value1, value2

FUNC MULT
  IF __PARAMCNT = 2 THEN                ' two bytes?
    tmpW1 = __PARAM1
    tmpW2 = __PARAM2
  ELSEIF __PARAMCNT = 3 THEN            ' word and byte?
    tmpW1 = __WPARAM12
    tmpW2 = __PARAM3
  ELSE
    tmpW1 = __WPARAM12                  ' two words
    tmpW2 = __WPARAM34
  ENDIF
  tmpW1 = tmpW1 * tmpW2
  RETURN tmpW1
  ENDFUNC
```

When a word and byte are passed, the ***__PARAMCNT*** variable will be set to three. The `MULT` function expects mixed values to be passed as word and then byte. If we pass the byte and then the word, ***__PARAMCNT*** will still be set to three but the results will not be correct due to the misalignment of values when moved into tmpW1 and tmpW2.

You may have noticed in some of the listings above that there is a "Use" example in comments associated with the code. This practice will help you (and others using your code) to keep things straight. Remember, the ***__PARAMCNT*** variable indicates the number of bytes being passed to a subroutine or function, but it does not determine what those bytes mean – your program does. Here are the possibilities:

***__PARAMCNT* = 1**

A. One byte in ***__PARAM1***

***__PARAMCNT* = 2**

A. First byte in ***__PARAM1***
Second byte in ***__PARAM2***

B. One word in __WPARAM12

***__PARAMCNT* = 3**

A. First byte in *__PARAM1*
   Second byte in *__PARAM2*
   Third byte in *__PARAM3*

B. One byte in *__PARAM1*
   One word in *__WPARAM23*

C. One word in *__WPARAM12* ← Recommended for mixed types
   One byte in *__PARAM3*

__PARAMCNT = 4

A. First byte in *__PARAM1*
   Second byte in *__PARAM2*
   Third byte in *__PARAM3*
   Fourth byte in *__PARAM4*

B. First byte in *__PARAM1*
   Second byte in *__PARAM2*
   One word in *__WPARM34*

C. First byte in *__PARAM1*
   One word in *__WPARM23*
   Second byte in *__PARAM4*

D. One word in *__WPARM12* ← Recommended for mixed types
   First byte in *__PARAM3*
   Second byte in *__PARAM4*

E. First word in *__WPARAM12*
   Second word in *__WPARAM34*

I think the list above illustrates the importance of designing your subroutines and functions carefully when they are using mixed-type parameters. A quick comment associated with your subroutines and functions code can save a lot of troubleshooting.

Returning Multiple Values from a Function

We've seen how a subroutine or function can receive multiple, even mixed-type, values. What if we want a function to return multiple values – is this possible? Yes, it is; a function can return up to four bytes and as with receiving multiple parameters, returning multiple bytes, especially as mixed-type values, requires a bit of thought.

A function that returns two bytes could be returning a word or two individual bytes; again, that is based on our design – the compiler makes no assumption of what the bytes mean. When returning a word we would do something like this:

```
RETURN tmpW1
```

… and in the main listing we'd use something like this to get that word:

```
wResult = MY_FUNC
```

But what if we want the function to return two individual (and unrelated) byte values? The function's **RETURN** instruction would be modified like this:

```
RETURN tmpB1, tmpB2
```

In this case the first byte will be returned in *__PARAM1* and the second in *__PARAM2*. Of course, the call to the function must now be modified as well:

```
bResult1 = MY_FUNC                          ' get the first byte
bResult2 = __PARAM2                         ' get the second byte
```

Note that two (or more) lines are required if the function is returning more than one value. If the second line (above) is not used, then the value in *__PARAM2* will likely be lost as this variable (as with all the *__PARAM* variables) is used internally by SX/B instructions. So, make sure you save the results of a function immediately after the call to prevent losing what was returned. By using multiple comma-delimited values with **RETURN**, your function can return up to five bytes (*__PARAM1* through *__PARAM5*) to the caller.

Let's look at one more example just so that this is clear. With this line:

```
RETURN tmpW1, tmpB1, tmpB2
```

… the calling code should look something like:

```
myWordVal = MY_FUNC
myByteVal1 = __PARAM3
myByteVal2 = __PARAM4
```

Does this make sense? – it should. The word value is returned in *__WPARAM12* which is built from *__PARAM1* and *__PARAM2*, hence the first byte in the list is returned in *__PARAM3*.

Review

Subroutines and functions allow us to extend SX/B as we desire and to save precious code space that would otherwise be [redundantly] used by the compile-in-place nature of SX/B. SX/B affords a great deal of flexibility in passing values back and forth between the main program and custom subroutines and functions; it's simply a matter for us to design, declare, and code those routines. You'll get lots of practical guidance using subroutines and functions in the chapters that follow.

15: Using and Managing Variable Space in SX/B

When compared to a microcontroller like the Basic Stamp, the SX has a lot of RAM (variable) space; 136 bytes in the SX28 and 262 bytes in the SX48. That said, not all of the RAM space is available for user variables. Another challenge at first is that the RAM space is not contiguously addressed, it is in fact split into "banks" of 16 bytes each; the bank concept for RAM is not unique to the SX and is used in many other small microcontrollers. The SX28 has seven banks; the SX48 has 16 banks.

For simple programs with minimal variable requirements we will not be bothered by the RAM banking. The SX's RAM, when managed by SX/B, works out like this:

Device	General RAM	Arrays	Max Array Size
SX20	20	6x16 + 1x5 + 1x4	16
SX28	19	6x16 + 1x5 + 1x4	16
SX48/52	17	223	223

Table 15.1
Parameter-passing RAM

This table deserves a bit of explanation. When using an SX28, for example, we have 19 bytes of program memory that we never have to worry about. In the general RAM space we can define bytes and words, and define bits within a byte or word. Here's an example defining each of these types:

```
flags           VAR     Byte
hiTemp          VAR     flags.0
loTemp          VAR     flags.1

tmpSetting      VAR     Word
```

These definitions would occupy three bytes of the general RAM space: one byte for the *flags* variable, two bytes for the *tmpSetting* variable. When defining words like this the compiler also provides aliases to the low- and high-bytes of the word, in this case those aliases would be *tmpSetting_LSB* and *tmpSetting_MSB*.

It may at first seem odd that the bigger processors have smaller general RAM spaces. The reason for this is that the SX's architecture maps the output ports into the general RAM space; the SX28 has one more port (RC) than the SX20, and the SX48/52 has two more ports (RD and RE) than the SX28.

It's not a frequent occurrence, but you can actually run out of general (bank 0) RAM space. When you do the compiler will complain with the following error:

"VARIABLE EXCEEDS AVAILABLE RAM"

Of course, we haven't run completely out of RAM, we've just exceeded the limits of the general RAM space. We can use the variable space in bank 1 and higher but to do so in SX/B means we have to declare those variables as elements of an array of bytes – SX/B does not currently support arrays of bits or words. In the table above you can see that the SX28 will handle up to six arrays of 16 elements, plus one array of five elements, plus one array of four elements.

As we covered in earlier chapters, SX/B allows the *aliasing* (renaming) of variables so in an advanced application we might see something like this:

```
serial          VAR     Byte (16)
txCount         VAR     serial(0)
txDivide        VAR     serial(1)
txLo            VAR     serial(2)
txHi            VAR     serial(3)
rxCount         VAR     serial(4)
rxDivide        VAR     serial(5)
rxByte          VAR     serial(6)
baud1x0         VAR     serial(7)
baud1x5         VAR     serial(8)
```

When using these definitions, manipulating the variable called *txCount* is actually manipulating element zero of the serial array. When an array is declared as above, it is automatically placed in the first available RAM bank above bank zero.

Before moving on let's look at a diagram of the SX RAM banking scheme in Figure 15.1. The SX20/28 memory consists of a global RAM bank of special function registers (for program and port control), and eight, 16-byte banks of general-purpose RAM. The SX48 memory consists of the global RAM bank and 16, 16-byte banks of general-purpose RAM.

In this diagram, ***n*** is seven for the SX20/28, and 15 for the SX48/52. On the SX20 ports C, D, and E are not implemented so those bytes are available for general use. On the SX28 ports D and E are not implemented so those bytes are available for general use. The SX/B compiler reserves five bytes (as *__PARAM1* through *__PARAM5*) of the global bank immediately after the port variables for internal use.

	Global		Bank 0	Bank 1	Bank 2		Bank n
$00 –	IND	$10 –	$0	$0	$0		$0
$01 –	RTCC	$11 –	$1	$1	$1	. . .	$1
$02 –	PC	$12 –	$2	$2	$2		$2
$03 –	Status	$13 –	$3	$3	$3		$3
$04 –	FSR	$14 –	$4	$4	$4		$4
$05 –	Port A	$15 –	$5	$5	$5		$5
$06 –	Port B	$16 –	$6	$6	$6		$6
* $07 –	Port C	$17 –	$7	$7	$7		$7
* $08 –	Port D	$18 –	$8	$8	$8		$8
* $09 –	Port E	$19 –	$9	$9	$9		$9
$0A –	$A	$1A –	$A	$A	$A		$A
$0B –	$B	$1B –	$B	$B	$B		$B
$0C –	$C	$1C –	$C	$C	$C		$C
$0D –	$D	$1E –	$D	$D	$D		$D
$0E –	$E	$1E –	$E	$E	$E		$E
$0F –	$F	$1F –	$F	$F	$F		$F

SX RAM Banking Scheme

Figure 15.1

Note that SX/B reserves several bytes in bank 7 on the SX20 and SX28 for TRIS "shadow registers" and saving internal variables during interrupts; advanced programmers need to exercise caution before manually mapping variables into the SX's RAM space. When defining arrays, bank selection is handled by the compiler.

The general RAM space used by SX/B is the memory available in the global bank after the port variables, and in bank 0. When arrays are defined they are mapped into bank 1 and higher as space is available. Note that the SX20 and SX28 are limited to arrays of 16 elements each. Given no other definitions, these arrays:

```
arrayN1         VAR     Byte (12)
arrayN2         VAR     Byte (12)
```

… would be mapped into banks 1 and 2, respectively, when used in an SX20 or SX28. If the total size of both arrays was 16 or fewer bytes they could be mapped into a single bank; this process is handled during the compilation stage, so despite what looks like scary complexities at first, we need not be concerned about the actual bank mapping of our arrays. SX/B takes care of the nitty-gritty details.

SX/B Variable Usage Considerations

As suggested earlier the general RAM space (global space and bank 0) is easiest on the code generated by SX/B. In fact, many instructions reset the SX's bank pointer (called the FSR) after an instruction to make sure the SX is pointing to the general RAM by default.

For complete details on SX RAM space utilization and access refer to Appendix E in the *SX-Key/Blitz Development System Manual*.

It should be understood that the only native variable types in the SX are the byte and bit, and a bit is not really a stand-alone element, it is, essentially, an element of a bit-array that we call a byte.

The word variable type is synthesized in code. In SX/B the only place that word variables can be assigned is in the general space. And for those coming from Parallax's Basic Stamp 2 family (or any of the multitude of copy-cats) the nib variable type is also synthesized and does not exist in SX/B.

One of the great joys I get from being a programmer is programming my way out of problems. The rest of this chapter will be devoted to utility functions that let us get the most from the SX's abundant RAM space.

Bit Access in Bytes and Words

A byte really is just an array of eight bits, but there is no mechanism in SX/B – or assembly for that matter – to provide a variable index for a bit within a byte. We spent a lot of time on subroutines and functions in Chapter 14 so let's put that education to work.

We'll start by creating a set of functions that allow us to manipulate byte-sized or word-sized bit arrays. Start with these declarations:

```
SET_BIT         FUNC    2, 2, 3                 ' set a bit
CLR_BIT         FUNC    2, 2, 3                 ' clear a bit
PUT_BIT         FUNC    2, 3, 4                 ' put bit into value
GET_BIT         FUNC    1, 2, 3                 ' get bit from value
```

The first three functions return two bytes so that they're compatible with word variables. Don't worry if your output variable for one of the functions is a byte, the SX/B compiler

knows to use the LSB of a word variable that is being moved to a byte. As we saw in the last chapter, we can use SX/B's *__PARAMCNT* variable to determine whether a byte or word was passed to the function.

The first function, SET_BIT, is designed to let us write a 1 to any bit position within a byte or word. The modified value will be returned (as a word). The reason for this design is that it provides the greatest flexibility in use as the function result can be written back to the variable that was passed, or to a separate variable altogether – like these examples:

```
  val1 = SET_BIT val1, 3                      ' change val1
  val2 = SET_BIT val1, 0                      ' change val2; val1 unchanged

And now for the function listing:

' Use: result = SET_BIT value, pos
' -- sets bit of "value" specified by "pos" to 1

FUNC SET_BIT
  IF __PARAMCNT = 2 THEN
    tmpW1 = __PARAM1                          ' save byte value
    tmpB1 = __PARAM2                          ' save position
  ELSE
    tmpW1 = __WPARAM12                        ' save word value
    tmpB1 = __PARAM3                          ' save position
  ENDIF
  tmpW2 = 1 << tmpB1                          ' convert position to mask
  tmpW1 = tmpW1 | tmpW2                       ' set selected bit
  RETURN tmpW1                                ' return to caller
  ENDFUNC
```

This function starts by saving the value passed to it in *tmpW1* (a word) and the bit position to set in *tmpB1*, using the *__PARAMCNT* variable to determine how the values were passed to the function. The position value *tmpB1* is converted to a word-sized bit mask for the position by using the left-shift operator.

Using the left-shift operator is an efficient way to create a mask; here are a few values to illustrate how it works:

```
1 << 0  = %0000000000000001
1 << 1  = %0000000000000010
1 << 8  = %0000000100000000
1 << 15 = %1000000000000000
```

When using the left-shift operator (<<) the shift value indicates how many zeroes will end up on the right of the value being shifted.

The value and position bit mask are ORed together to set the selected bit position (one OR anything is one) and then the modified value is returned to the caller.

The second function, CLR_BIT, is the complement of SET_BIT in that it allows us to clear the specified bit within a byte or word value.

```
' Use: result = CLR_BIT value, pos
' -- clears bit of "value" specified by "pos" to 0

FUNC CLR_BIT
  IF __PARAMCNT = 2 THEN
    tmpW1 = __PARAM1                          ' save byte value
    tmpB1 = __PARAM2                          ' save position
  ELSE
    tmpW1 = __WPARAM12                        ' save word value
    tmpB1 = __PARAM3                          ' save position
  ENDIF
  tmpW2 = 1 << tmpB1                          ' convert position to mask
  tmpW2 = ~tmpW2                              ' invert mask
  tmpW1 = tmpW1 & tmpW2                       ' clear selected bit
  RETURN tmpW1                                ' return to caller
  ENDFUNC
```

As you can see, this function starts out identically to SET_BIT. The change comes after the creation of the bit mask where the bits in the mask value are inverted (0 to 1, 1 to 0). Now the mask is ANDed with the target value, and as the position bit is zero from the inversion of the mask, this bit will be cleared (zero AND anything is zero).

For those programs where we want to write a variable value to a given position within a byte or word, the PUT_BIT function is useful. This is, essentially, a combination of SET_BIT and CLR_BIT that uses the bit value passed to the function to determine how the mask is applied.

```
' Use: result = PUT_BIT value, pos, bitVal
' -- sets bit of "value" specified by "pos" to bitVal.0

FUNC PUT_BIT
  IF __PARAMCNT = 3 THEN
    tmpW1 = __PARAM1                          ' save byte value
    tmpB1 = __PARAM2                          ' save position
    tmpB2 = __PARAM3                          ' save bit value
  ELSE
    tmpW1 = __WPARAM12                        ' save word value
    tmpB1 = __PARAM3                          ' save position
    tmpB2 = __PARAM4                          ' save bit value
  ENDIF
  tmpW2 = 1 << tmpB1                          ' convert position to mask
```

```
  IF tmpB2.0 = 0 THEN                           ' 0?
    tmpW2 = ~tmpW2                              ' yes, invert mask
    tmpW1 = tmpW1 & tmpW2                       ' clear selected bit
  ELSE
    tmpW1 = tmpW1 | tmpW2                       ' no, set selected bit
  ENDIF
  RETURN tmpW1                                  ' return to caller
  ENDFUNC
```

Note that the bit value must be passed as a byte and will be saved in *tmpB2*. When the function is ready to check this value it tests bit 0; if this bit is zero then the specified position bit is cleared, otherwise the position bit is set.

The final function for dealing with bit arrays is called `GET_BIT`. This function will retrieve a specific bit value from a byte or word.

```
' Use: result = GET_BIT value, pos
' -- returns 0 or 1 based on value.pos

FUNC GET_BIT
  IF __PARAMCNT = 2 THEN                        ' byte value
    tmpW1 = __PARAM1                            ' save value
    tmpB1 = __PARAM2                            ' save position
  ELSE
    tmpW1 = __WPARAM12                          ' word value
    tmpB1 = __PARAM3
  ENDIF
  tmpW2 = 1 << tmpB1                            ' convert position to mask
  tmpW1 = tmpW1 & tmpW2                         ' clear other bits
  IF tmpW1 > 0 THEN                             ' if result not zero
    tmpB1 = 1                                   '   return 1
  ELSE
    tmpB1 = 0
  ENDIF
  RETURN tmpB1                                  ' return to caller
  ENDFUNC
```

Like the others, this function creates a mask and applies it to the target value. If the specified bit was set, the result of the AND operation will be equal to the mask value, otherwise the result will be zero. To keep the function friendly we'll reassign the output byte to 1 if the bit was set, hence the bit value ends up bit 0 of the returned byte.

For BASIC Stamp 2 programmers the `PUT_BIT` and `GET_BIT` functions described above can be used in place of the **`.LOWBIT(idx)`** variable modifier.

Nib Access in Bytes and Words

You know by now that the nib, like the word, is a synthesized data type and not native to the SX. That said, using nib values can be convenient and there will be times, especially when dealing with BCD (*binary coded decimal*) numbers that nib placement or extraction is useful.

As above, we can use SX/B to create a set of routines to take care of nib access for those programs that need this feature. With two functions we can set and get a specified nib within another byte or word value.

```
PUT_NIB         FUNC    2, 3, 4                 ' sets specified nib
GET_NIB         FUNC    1, 2, 3                 ' returns specified nib
```

The `PUT_NIB` function will return an updated value if the position specified is valid, that is between 0 and 3, inclusive, as these are the only valid nibs in a word. Likewise, `GET_NIB` will return the specified nib if the position specified is valid, otherwise the result will be zero. Let's look at the code for `PUT_NIB`:

```
' Use: result = PUT_NIB value, newNib, pos
' -- "value" returned in "result" if "pos" not valid

FUNC PUT_NIB
  IF __PARAMCNT = 3 THEN                        ' byte passed
    tmpW1 = __PARAM1                            ' save value
    tmpB1 = __PARAM2 & $0F                      ' save newNib (four bits)
    tmpB2 = __PARAM3                            ' save position
  ELSE                                          ' word passed
    tmpW1 = __WPARAM12
    tmpB1 = __PARAM3 & $0F
    tmpB2 = __PARAM4
  ENDIF
  IF tmpB2 < 4 THEN                             ' validate position
    tmpB2 = tmpB2 << 2                          ' x4 for nibs
    tmpW2 = %1111 << tmpB2                      ' create mask
    tmpW2 = ~tmpW2                              ' invert for ANDing
    tmpW1 = tmpW1 & tmpW2                       ' clear previous nib
    tmpW2 = tmpB1 << tmpB2                      ' shift newNib into
position
    tmpW1 = tmpW1 | tmpW2                       ' put into result
  ENDIF
  RETURN tmpW1
  ENDFUNC
```

After saving the parameters passed to it, PUT_NIB checks the position value – if this is out of range (0..3) there is no need for additional processing. Assuming the position value is good, the nib position is converted to a bit shift value by multiplying it by four. Note, though, that the multiplication operator is not used, instead, the shift-left operator is used here. There are times when shifting is the better choice than multiplying or dividing.

When multiplying or dividing by powers of two (2, 4, 8, 16, 32…), the shift operators are far more efficient than * or /. Instead of:

```
result = result * 4
```

use:

```
result = result << 2
```

The left-shift operator (<<) is used for multiplying, the right shift operator (>>) for dividing.

Using the newly adjusted *tmpB2* a mask is created in *tmpW2*. The first line sets ones in the bits of the target nib, the second line inverts the bits of the mask so that the AND operator can be used to clear the bits of the target nibble.

The new nibble value is now shifted into the appropriate position and placed into the result by using OR. The updated value, in *tmpW1*, is returned to the caller.

Now let's see how we can extract a nibble value from a byte or word.

```
' Use: result = GET_NIB value, pos
' -- returns 0 if "pos" is not valid

FUNC GET_NIB
  IF __PARAMCNT = 2 THEN                        ' byte passed
    tmpW1 = __PARAM1                            ' save value
    tmpB1 = __PARAM2                            ' save position
  ELSE                                          ' word passed
    tmpW1 = __WPARAM12
    tmpB1 = __PARAM3
  ENDIF
  IF tmpB1 < 4 THEN                             ' valid
    tmpB1 = tmpB1 << 2                          ' x4 for nibs
    tmpW1 = tmpW1 >> tmpB1                      ' move target nib to _LSB
  ELSE
    tmpW1_LSB = 0                               ' not valid, result is zero
  ENDIF
  tmpW1_LSB = tmpW1_LSB & $0F                   ' strip off high nibble
  RETURN tmpW1_LSB                              ' return result to caller
  ENDFUNC
```

For the GET_NIB function we need the value and the nibble position. With those parameters saved we'll check the position to ensure that it's valid, if not the low byte of *tmpW1* is cleared to zero. For a valid position the position is converted to a bit shift value as above, and then the input value is shifted right; this puts the target nibble into the low nib of tmpW1_LSB. The high nibble of tmpW1_LSB is stripped off by ANDing it with $0F and the final result is returned to the calling program. Even though a byte is returned, it will always be in the range of $0 to $F.

You might think that we would extend our function library with PUT_BYTE and GET_BYTE but as indicated in the first part of this chapter, a word variable is actually created from two bytes. Instead of creating a whole function to set a given byte within a word we can use our knowledge of SX/B variables to do it directly:

Let's say we want to set the high byte of result to 0 – here's how easy that is:

```
  result_MSB = 0
```

For review, this declaration:

```
result          VAR     Word
```

… actually generates the following definitions:

```
result          EQU     0x0E
result_LSB      EQU     result
result_MSB      EQU     result+1
```

Note that to use a word you must declare a word – you cannot manually synthesize words by making equivalent declarations in high-level SX/B.

Review

The SX has a lot of RAM space and in most applications we won't ever have to think about assignments. In those rare cases where our simple variable requirements exceed the amount of general RAM we can overcome this problem by creating arrays and using aliasing to give the array elements useful names; SX/B will take care of the rest for us. When bit or nib access is required the subroutines and functions in this chapter should be handy.

Yes, there are additional memory tricks (like arrays of more than 16 bytes on an SX28) that we can use to get the most of the SX's ample RAM space, and those are best learned in application, so let's get to it!