SX-Key/Blitz Development System Manual

Version 1.1

Warranty

Parallax warrants its products against defects in materials and workmanship for a period of 90 days. If you discover a defect, Parallax will, at its option, repair, replace, or refund the purchase price. Simply call our sales department for an RMA number, write it on the label and return the product with a description of the problem. We will return your product, or its replacement, using the same shipping method used to ship the product to Parallax (for instance, if you ship your product via overnight express, we will do the same).

This warranty does not apply if the product has been modified or damaged by accident, abuse, or misuse.

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 refund. Parallax will refund the purchase price of the product, excluding shipping / handling costs. This does not apply if the product has been altered or damaged.

Copyrights and Trademarks

Disclaimer of Liability

Parallax, Inc. is not responsible for special, incidental, or consequential damages resulting from any breach of warranty, or under any legal theory, including lost profits, downtime, goodwill, damage to or replacement of equipment or property, and any costs of recovering, reprogramming, or reproducing any data stored in or used with Parallax products.

Internet Access

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

Web:	http://www.parallaxinc.com
	http://www.sxtech.com
Ftp:	ftp.parallaxinc.com
	ftp.sxtech.com
General e-mail:	info@parallaxinc.com
Tech. e-mail:	sxtech@parallaxinc.com

Internet SX Discussion List

We maintain an SX discussion list for people interested in SXs. Many people subscribe to the list, and all questions and answers to the list are distributed to all subscribers. It's a fun, fast, and free way to discuss SX issues. To subscribe to the SX list, send email to *majordomo@parallaxinc.com* and write *SUBSCRIBE SX* in the body of the message.

This manual is valid with the following software and firmware versions:

Assembler:

SXKey28L.exe software version 1.07
SXKey52.exe software version 1.12

Firmware:

SX-Key rev. F

The information herein will usually apply to newer versions but may not apply to older versions. New software can be obtained free on our ftp and web site (ftp.parallaxinc.com, www.parallaxinc.com). If you have any questions about what you need to upgrade your product, please contact Parallax.

Contents

PREFIX 5

PACKING LIST...5
- SX Tech Tool Kit 5
- SX Tech University Kit 6
- SX-Blitz..6

INTRODUCTION TO THE SX-KEY HARDWARE 7

QUICK START GUIDE..9

QUICK START INTRODUCTION...9
CONNECTING AND DOWNLOADING (SX TECH BOARD)...............................9
CONNECTING AND DOWNLOADING (SX-KEY DEMO BOARD).......................11
- Configuring HyperTerminal (SX-Key Demo Board)...................................13

COMMUNICATING WITH THE VIRTUAL PERIPHERALS IN THE SX CHIP..............13
CONNECTING AND DOWNLOADING (SX-KEY QUICKPROTO BOARD) 14

INSTALLATION OF THE SX-KEY INTERFACE..17

WHAT'S NEW 19

FOR THOSE FAMILIAR WITH PIC® MICROCONTROLLERS:..............................19
- Similarities:..19
- Differences: 19

FOR THOSE FAMILIAR WITH THE SUPER PARALLAX ASSEMBLER: 21
- Similarities:..21
- Differences: 21

INTRODUCTION TO THE SX-KEY INTERFACE...23

THE MENUS 24
- The File menu: 24
- The Edit menu:...25
- The Run menu: 25
- The Help menu:..26

THE WINDOWS..26
- Print Window 26
- Find/Replace Window 27
- Debug Windows..27
- Debugging:...32
- Device Window:..33
- Configure Window:...35

Contents

SX-KEY ASSEMBLER....................39

COMMENTS....................40
ASSEMBLER DIRECTIVES....................40
SYMBOLS....................55
LABELS....................56
EXPRESSIONS....................57
DATA TYPES....................59

SX SPECIAL FEATURES AND CODING TIPS....................61

INTRODUCTION....................61
PORT CONFIGURATION AND USAGE....................61
Port Direction....................62
Pull-up Resistors....................63
Logic Level....................64
Schmitt-Trigger....................65
Edge Detection....................66
Wake-Up (Interrupt) on Edge Detect....................68
Comparator....................70
ALL ABOUT INTERRUPTS....................72
RTCC Rollover Interrupts....................74
CREATING TABLES....................77
8-bit Data Tables....................77
8-bit or 12-bit Data Tables....................78
DEALING WITH CODE PAGES....................79
Branching Across Pages....................79
Calling Across Pages with Jump Tables....................80

APPENDIX A: SX FEATURES....................83

INTRODUCTION....................83
CPU FEATURES....................83
PERIPHERAL AND I/O FEATURES....................83

APPENDIX B: SX INSTRUCTION SET....................85

PARALLAX INSTRUCTION SET SUMMARY....................85
SINGLE-WORD INSTRUCTIONS....................86
MULTI-WORD INSTRUCTIONS....................87
INSTRUCTION SET QUICK REFERENCE....................88

APPENDIX C: SX-KEY RESERVED WORDS....................113

APPENDIX D: SX-KEY ERROR MESSAGES....................115

SX-KEY ERROR MESSAGES........115

APPENDIX E: SX-KEY DEMO/QUICKPROTO BOARD........121

SX TECH BOARD FEATURES........121
Features........122
CONNECTING AND DOWNLOADING........122
SX-KEY DEMO BOARD FEATURES........124
Features........125
VIRTUAL PERIPHERAL SOURCE CODE EXAMPLE........125
CONNECTING AND DOWNLOADING........125
COMMUNICATION........126
CONFIGURING THE VIRTUAL PERIPHERALS........126
SX-KEY DEMO BOARD SCHEMATIC........127
SX-KEY QUICKPROTO BOARD FEATURES........128
Features........129
CONNECTING AND DOWNLOADING........129
SX-KEY QUICKPROTO BOARD SCHEMATIC........130
PINOUT INFORMATION AND DESCRIPTIONS (NOT TO SCALE OR PROPORTION)........131

APPENDIX F: SX DATA SHEET........131

ARCHITECTURE........132
INSTRUCTION PIPELINE........132
Read-Modify-Write Considerations........133
REGISTER MAP STRUCTURE........134
*Port C is available as general-purpose RAM in the SX18.........134
* Bank 0 is available only when using semi-direct addressing.........134
SPECIAL FUNCTION REGISTERS........135
IND - The Indirect Register ($00)........135
RTCC Real Time Clock/Counter and WREG ($01)........135
PC - Program Counter ($02)........136
STATUS Register ($03)........136
FSR - File Select Register ($04)........138
Direct Addressing........139
Indirect Addressing........141
The Bank Instruction........143
The Jump Instruction........143
Jumping Across Pages........144
Call Instruction........144
Calling Across Pages........145
Returning from a subroutine........146
The Stack........146

Contents

The Push 146
The Pop 147
Stack Overflow 147
Stack Underflow 147
Returns 148
PORT CONFIGURATION REGISTERS 148
Port A Registers 148
Port B Registers 149
Port C Registers 152
CONTROL REGISTERS 154
Mode 154
Option 156
Fuses 156
INTERRUPTS (SX18/28) 157
Description 157
The Specifics 157
RTCC Interrupt 158
RB0-RB7 Interrupt 158
PERIPHERALS 158
Oscillator Driver 158
XT, LP, and HS Mode 158
External RC Mode 160
Internal RC Mode 161

INDEX 163

Thank you for purchasing the Parallax SX-Key®/Blitz development system. We have done our best to produce a full-featured, yet easy to use development system for the SX microcontrollers. The result is the SX-Key and the SX-Blitz; very tiny, full-featured SX development tools with a Windows® 95/98/NT4 interface. We hope you will find this system as enjoyable to use as we do.

This manual is written for the latest available SX chips as of August, 1999. This includes the SX18/28 chips with a date code of AB9921AA or later, and SX48/52 chips with a date code of 9901 or later**. Chips with older date codes are not supported by this manual.** The SX microcontroller line is likely to continue to evolve in the future so it is important to download the latest errata sheet from http://www.parallaxinc.com/sx/sxerrata.htm.

Packing List

There are two SX-Key development packages and one SX-Blitz package available: the SX Tech Tool Kit, the SX Tech University Kit and the SX-Blitz. The packing lists for each of these is listed below:

SX Tech Tool Kit

- (1) SX-Key (rev. F) Programmer/Debugger Device
- (1) SX-Key/Blitz Development System software
- (1) SX-Key/Blitz manual (this manual)
- (1) SX Tech Board
- (1) Power Supply (except for kits shipped to Europe)
- (1) 9-pin serial cable
- (2) 28-pin SX chips (SX28AC/DP)
- (1) Murata 50 MHz resonator

Prefix

SX Tech University Kit

- (1) SX-Key (rev. F) Programmer / Debugger Device
- (1) SX-Key / Blitz Development System software
- (1) SX-Key / Blitz manual on diskette or CD
- (1) SX Tech Board
- (2) 28-pin SX chips (SX28AC / DP)
- (1) Murata 50 MHz resonator

SX-Blitz

- (1) SX-Blitz Programmer Device
- (1) SX-Key / Blitz Development System software
- (1) SX-Key / Blitz manual on diskette or CD
- (2) 28-pin SX chips (SX28AC / DP)

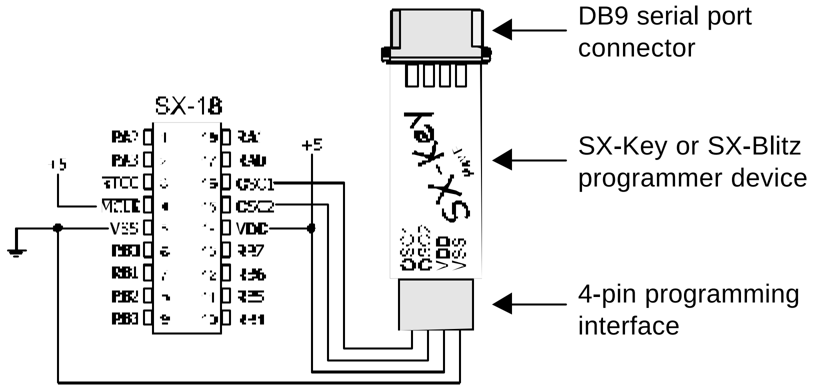
If any items are missing, please let us know.

Additionally, there are two other development boards that support the SX 18 and 28 parts: SX-Key Quick Proto Board and the SX-Key Demo Board, respectively. See Appendix E for more information.

1: Introduction to the SX-Key/Blitz Hardware

The SX-Key / Blitz hardware consists of the programmer unit, a four-pin programming interface and a standard, female serial port connector (DB9). The serial port connector should be plugged into an available standard, straight-through serial cable on an IBM-compatible PC. The four-pin connector on the SX-Key / Blitz board should be connected to four pins (VSS, VDD, OSC2 and OSC1) of the SX chip itself (see Figure 1.1).

Figure 1.1: SX-Key / Blitz Hardware Connection.


DB9 serial port connector
SX-Key or SX-Blitz programmer device
4-pin programming interface
SX-18
+5
+5
RA2
RA3
RTCC
MCLR
VSS
RB0
RB1
RB2
RB3
RA1
RA0
OSC1
OSC2
VDD
RB7
RB6
RB5
RB4

The SX-Key / Blitz is powered by the target circuit's power supply and programming and debugging takes place over the oscillator pins. The power supply to the SX-Key / Blitz must be +5 vdc. If an external crystal, resonator or RC circuit is used, the SX-Key / Blitz can usually remain connected to the SX chip for programming purposes, without affecting the operation of the circuit. When debugging, the SX chip must not have an external clock source since the SX-Key's internal programmable oscillator must be used. ***The SX-Blitz can only program SX chips, it can not debug them.***

Each SX microcontroller contains the necessary debugger hooks required to perform SX in-circuit debugging. No other supporting chips are necessary for the debugging process. During debugging, the SX-Key provides the oscillator signal to drive the SX microcontroller until such time that a breakpoint is hit or a single step or stop mode is initiated. NOTE: Figure 1.1 shows all the connections necessary to program, debug and run the SX microcontroller. An external resonator or crystal should be connected to the OSC1 and OSC2 pins to run the

SX if the SX-Blitz is used, or if the SX-Key's internal clock oscillator is not used.

The SX-Blitz is designed to be a lower-cost device for programming the SX chips only (no debugging features are available). The SX-Blitz and SX-Key use the same interface software for programming, however, debugging features will not work with the SX-Blitz. NOTE: Since the SX-Blitz and SX-Key function almost identically, they will be referred to as the SX-Key/Blitz, except where there are distinct differences.

At the time of this writing, TQFP and PQFP sockets for the SX48/52 parts are available from:

YAMAICHI Electronics USA, Inc.
WWW: http://www.yeu.com
Part# IC51-0484-806 (48-pin TQFP)
Part# IC51-0524-1612 (52-pin PQFP)

2: Quick Start Guide

Quick Start Introduction

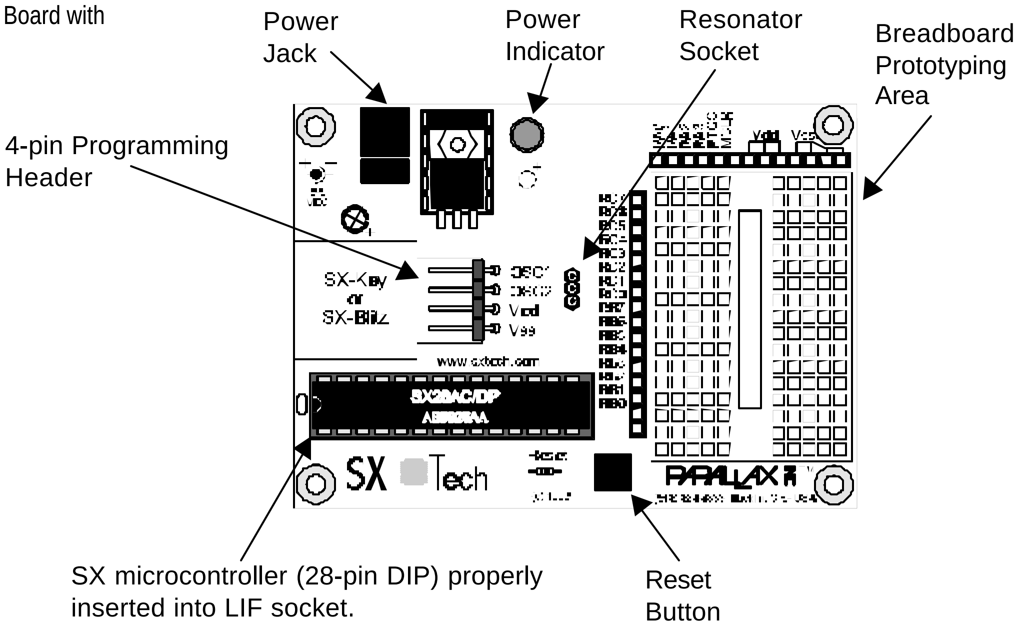
This chapter is a quick start guide to connecting the SX-Key/Blitz and programming the SX microcontroller. Without even knowing how the SX-Key/Blitz and the SX chip function, you should be able to obtain satisfactory results from the steps that follow.

This guide is split into three sections, one covers the SX Tech Board, one covers the SX-Key Demo Board and the last covers the SX-Key QuickProto board. Please proceed to the section concerning the board you received (if any).

Connecting and Downloading (SX Tech Board)

In order to get familiar with how the SX-Key/Blitz Development System works, we'll use the SX Tech Board to program and run a 28-pin SX chip. **This quick start guide section will not work with the SX-Key Demo Board or the QuickProto board. Please see the appropriate section for more information.**

Figure 2.1: SX Tech Board with SX chip inserted.



Keep in mind that the SX Tech Board is not a programmer; rather the SX-Key/Blitz is the programmer/debugger device while the SX Tech board is a type of prototyping board. Follow these steps to connect and download a program:

1) Plug an SX28AC/DP into the 28-pin LIF socket on the SX Tech board as shown in Figure 2.1. Make sure it is oriented so that the half-moon notch in the chip faces away from the "Reset" button.

2) Connect the SX-Key/Blitz to a serial cable, and the serial cable to an available serial port (Com: port) on the PC.

3) Connect the SX-Key/Blitz to the 4-pin programming header with the VSS, VDD, OSC2 and OSC1 indicators lining up with the same indicators on the board.

4) Insert one end of a 470 ohm resister into the RC7 socket (next to the upper left side of the breadboard). Insert the other end of the resister into any hole in the breadboard.

5) Insert the shortest leg of an LED into the breadboard hole that is closest (horizontally) to the resister leg. Insert the other leg of the LED into one of the VDD sockets (next to the top side of the breadboard).

6) Plug the power supply into the SX Tech board and into an available wall outlet. (The power indicator should light up).

7) Insert the SX-Key/Blitz Development System diskette into a floppy drive and use Windows Explorer (or the Run item on the Start menu) to locate and run the sxkey28l.exe program from the diskette. (The SX-Key window should appear).

8) In the SX-Key window, pull down the File menu and select Open (or press CTRL-O). In the browser window that appears, select and open the led28.src file. (The led28.src source code should appear in the SX-Key code window).

9) If the serial port (Com: port) you are using is not Com 1, pull down the Run menu, select Configure… and select the proper port from the Configuration window.

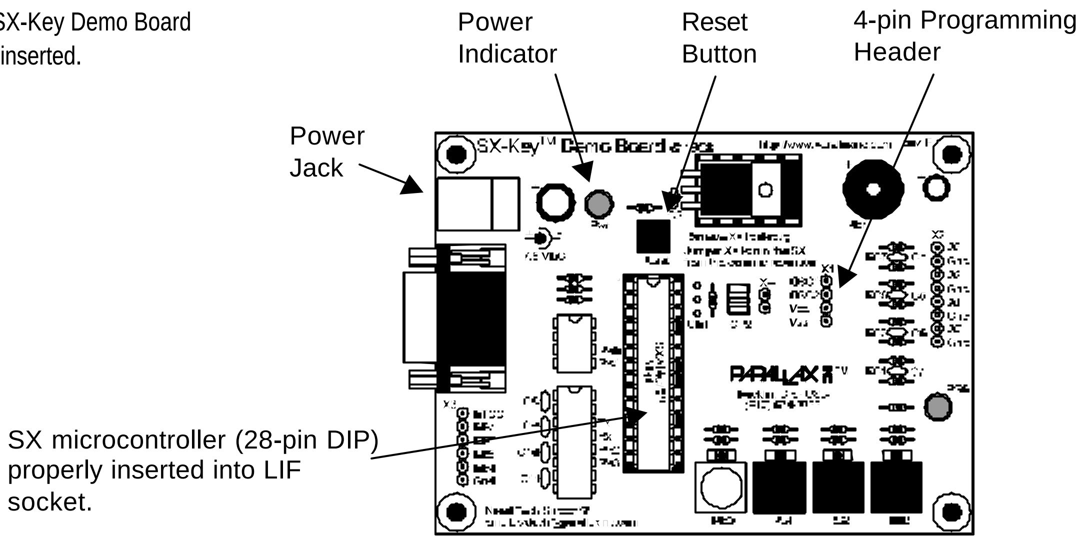
10) Pull down the Run menu and select Run (or press CTRL-R). (The SX-Key software should assemble the code and begin the programming process).

Congratulations! You have just programmed the SX microcontroller with the SX-Key/Blitz Development System. The program in the SX microcontroller should start running. The LED should flash on and off (if wired correctly).

Connecting and Downloading (SX-Key Demo Board)

In order to get familiar with how the SX-Key/Blitz Development System works, we'll use the SX-Key Demo Board to program and run a 28-pin SX chip. **This quick start guide section will not work with the SX Tech Board or SX-Key QuickProto board. Please see the appropriate section for more information.**

Figure 2.2: SX-Key Demo Board with SX chip inserted.



Keep in mind that the SX-Key Demo Board is not a programmer; rather the SX-Key / Blitz is the programmer / emulator device while the demo board is a type of prototyping board. Follow these steps to connect and download a program:

1) Plug an SX28AC / DP into the 28-pin LIF socket on the demo board as shown in Figure 2.2. Make sure it is oriented so that the half-moon notch in the chip faces the "Reset" button.

2) Connect the SX-Key / Blitz to a serial cable, and the serial cable to an available serial port (Com: port) on the PC.

3) Connect the SX-Key / Blitz to the 4-pin programming header (labeled X1) with the VSS, VDD, OSC2 and OSC1 indicators lining up with the same indicators on the board. Remove the X4 strap if it is installed.

4) Plug the power supply into the demo board and into an available wall outlet. (The power indicator should light up).

5) Insert the SX-Key / Blitz Development System diskette into a floppy drive and use Windows Explorer (or the Run item on the Start menu) to locate and run the sxkey28l.exe program from the diskette. (The SX-Key window should appear).

6) In the SX-Key window, pull down the File menu and select Open (or press CTRL-O). In the browser window that appears, select and open the sxdemo.src file. (The sxdemo.src source code should appear in the SX-Key code window).

7) If the serial port (Com: port) you are using is not Com 1, pull down the Run menu, select Configure… and select the proper port from the Configuration window.

8) Pull down the Run menu and select Run (or press CTRL-R). (The SX-Key software should assemble the code and begin the programming process).

Congratulations! You have just programmed the SX microcontroller with the SX-Key/Blitz Development System. The SX Virtual Peripheral demonstration program should start running. The piezo speaker will make a clicking noise and the yellow LED should flash on and off. To communicate with and configure the SX Virtual Peripherals, continue with the steps below.

Configuring HyperTerminal (SX-Key Demo Board)

9) Disconnect the serial cable from the SX-Key/Blitz and connect the serial cable to the demo board directly. NOTE: There is no need to unplug the power.

10) Open the HyperTerminal program. If you are unfamiliar with this program, click on the Start button, select the Run item and type: hypertrm in the Open: field. Note: there is only one "e" in the actual program name. (The HyperTerminal window should appear).

11) On the Connection Description window, type: ComPort in the Name: field and click on the OK button. (The Phone Number window should appear).

12) In the "Connect Using:" field, select one of the "Direct to Com…" items. **Make sure to select the same Com port number that you used to program the SX in steps 1 through 8.** Click OK when done. (The COM Properties window should appear).

13) Select 19200 bits per second, 8 data bits, no parity, 1 stop bit and no flow control. Click the OK button when done.

Communicating with the Virtual Peripherals in the SX chip (SX-Key Demo Board)

Press and release the Reset button on the SX-Key Demo Board. You should hear a clicking noise, the yellow LED should flash and "SX Virtual Peripheral Demo" should appear on your screen. The SX chip is now communicating with HyperTerminal at 19.2K baud over the serial cable.

Your cursor should appear next to a “>” prompt. Now you can type in simple commands to change the behavior of the virtual peripherals. For example, type: “T6” and press the Enter key. The yellow LED should start blinking faster. Type: “F9” and press the Enter key to increase the frequency on the piezo speaker. (Type: “F0” to stop the speaker). For more information on the SX-Key Demo Board or for a listing of available Virtual Peripheral demo commands, see Appendix E.

Connecting and Downloading (SX-Key QuickProto Board)

In order to get familiar with how the SX-Key/Blitz Development System works, we’ll use the SX-Key QuickProto Board to program and run an 18-pin SX chip. **This quick start guide section will not work with the SX Tech and SX-Key Demo board. Please see the previous sections for more information.**

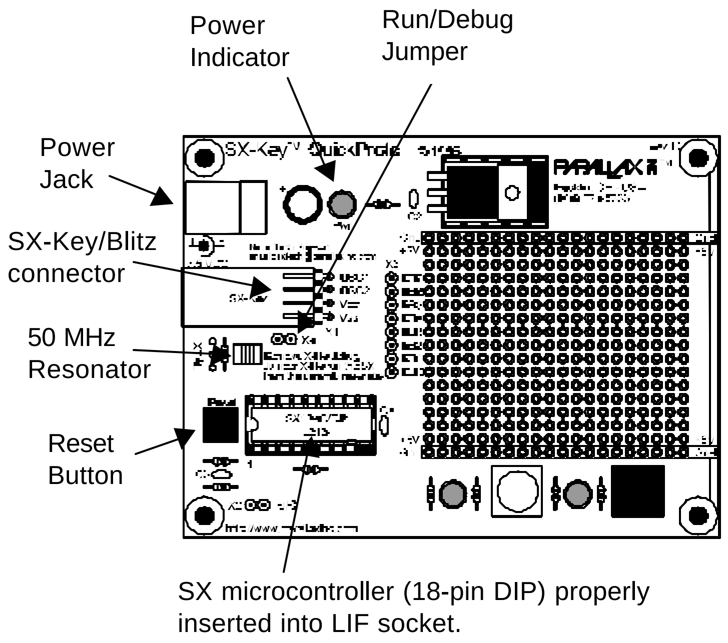


Figure 2.3: SX-Key QuickProto Board with SX chip inserted.

Keep in mind that the SX-Key QuickProto Board is not a programmer; rather the SX-Key/Blitz is the programmer/debugger device while the

QuickProto board is a type of prototyping board. Follow these steps to connect and download a program:

11) Plug an SX18AC/DP into the 18-pin LIF socket on the QuickProto board as shown in Figure 2.3. Make sure it is oriented so that the half-moon notch in the chip faces the “Reset” button.

12) Connect the SX-Key/Blitz to a serial cable, and the serial cable to an available serial port (Com: port) on the PC.

13) Connect the SX-Key/Blitz to the 4-pin programming header (labeled X1) with the VSS, VDD, OSC2 and OSC1 indicators lining up with the same indicators on the board. Remove the X4 strap if it is installed.

14) Plug the power supply into the QuickProto board and into an available wall outlet. (The power indicator should light up).

15) Insert the SX-Key/Blitz Development System diskette into a floppy drive and use Windows Explorer (or the Run item on the Start menu) to locate and run the sxkey28l.exe program from the diskette. (The SX-Key window should appear).

16) In the SX-Key window, pull down the File menu and select Open (or press CTRL-O). In the browser window that appears, select and open the led18.src file. (The led18.src source code should appear in the SX-Key code window).

17) If the serial port (Com: port) you are using is not Com 1, pull down the Run menu, select Configure… and select the proper port from the Configuration window.

18) Pull down the Run menu and select Run (or press CTRL-R). (The SX-Key software should assemble the code and begin the programming process).

Congratulations! You have just programmed the SX microcontroller with the SX-Key/Blitz Development System. The program in the SX microcontroller should start running. One of the yellow LEDs should flash on and off.

3: Installation of the SX-Key/Blitz Interface

The SX-Key/Blitz Interface consists of the integrated editor, programmer, and debugger software. The following system requirements are a minimum for using the SX-Key/Blitz Interface:

- 80486 (or higher) IBM or compatible PC;
- Windows 95/98/NT4 operating system;
- 16 Mb of RAM;
- 1 Mb of available hard drive space;
- 3 ½ inch floppy drive;
- 1 available serial port.

(Note: though it is suggested that the SX-Key/Blitz interface be installed on your hard drive, it is not required. The interface program may be run right off the SX-Key/Blitz Development System diskette).

To install the SX-Key/Blitz Interface:

1. Insert the SX-Key/Blitz Development System diskette in an available floppy drive.
2. Run the Windows Explorer program.
3. Create a directory on the hard drive in which you would like to install the software.
4. Copy all files from the floppy drive to the newly created directory.
5. Close Windows Explorer.

To use the SX-Key/Blitz Interface:

1) Run the appropriate program from the directory it was installed into in the procedure above. Use sxkey28l.exe for SX18/28 chips with a date code of AB9921AA or later. Use sxkey52.exe for SX48/52 chips.

4: What's New

For those familiar with PIC® microcontrollers:

The SX microcontroller has many similarities with Microchip's PIC16C5x series of microcontrollers. The similarities and differences are highlighted here. This is not a complete list. Please refer to the SX Datasheet for more information.

Similarities:

- **Pinout**: The SX-18 and SX-28 microcontrollers have the same pin configuration as the corresponding PIC16C5x devices; although physical package size might differ slightly. SX chips with 48 and 52-pins are also available.
- **RAM Registers:** The RAM registers and RAM banks are identical (except within the SX48/52).
- **Program Space:** The same amount of program space is available (512 x 12, 1024 x 12 and 2048 x 12). SX48/52 chips contain 4096 x 12 words of program space.
- **Opcodes:** The same opcodes are supported allowing object code developed for PIC16C5x devices to be programmed into the SX with no changes.
- **Instruction Execution**: The same instruction execution speed is available, in compatibility mode, at up to 20 MHz.

Differences:

- **Programming Pins**: The SX may be programmed using the OSC1 and OSC2 pins. This is the method used by the SX-Key/Blitz Development System.
- **Program Space**: The program memory is E^2Flash capable of being reprogrammed up to 10,000 times.
- **Debugging:** Each SX chip contains the necessary debugger hooks to allow in-circuit debugging. This is achieved with the SX-Key over the oscillator pins. No bondout chips are required and no I/O pin conflicts occur during debugging.
- **Opcodes:** Ten new opcodes are included to allow easy bank and page switching, interrupts with context save/restore, and more.

- **Instruction Execution:** By selecting the Turbo fuse, the SX chip will execute one instruction per clock, except branches which take three. This option gives you 4 times the speed of Compatibility mode. Additionally, the oscillator of the SX can be driven at up to 75 Mhz (on selected chips). Combined with the Turbo fuse, execution speeds of 75 MIPS can be realized. Faster versions of the SX chip are planned by Scenix.
- **Fuses:** New fuses have been added for functions such as: turbo mode, stack extend, option extend, add/sub with carry and input syncing.
- **Stack:** The SX includes a stack extend fuse to expand the stack from the PIC-compatible 2-levels to 8-levels deep.
- **Interrupts:** Interrupt capability is included with automatic context save/restore to and from built-in shadow registers.
- **Oscillators:** An internal RC oscillator is included that can run at speeds of 31 Khz to 4 Mhz. External oscillators can drive the chip at up to 75 Mhz. Note: SX with 4-digit date codes require a resister across OSC1 and OSC2. Contact Parallax, Inc for more information.
- **I/O Pins:** All I/O pins can source and sink 30 mA. All I/O pins have built-in, configurable pull-up resistors. RB, RC, RD and RE input pins are selectable as Schmitt Trigger.

4: What's New

For those familiar with the Super Parallax Assembler:

The SX-Key/Blitz assembler has many similarities with Parallax's Super Parallax Assembler. The similarities and differences are highlighted here.

Similarities:

- **Mnemonics:** Most mnemonic instructions are the same as in the Super Parallax Assembler. Most code already developed for PIC16C5x devices will load and assemble in the SX-Key/Blitz.
- **Language Syntax:** The SX-Key/Blitz Assembler uses a similar syntax as that of the Super Parallax Assembler.
- **Directives and Equates:** Directives are similar and backward compatible equates exist within the SX-Key/Blitz Assembler.

Differences:

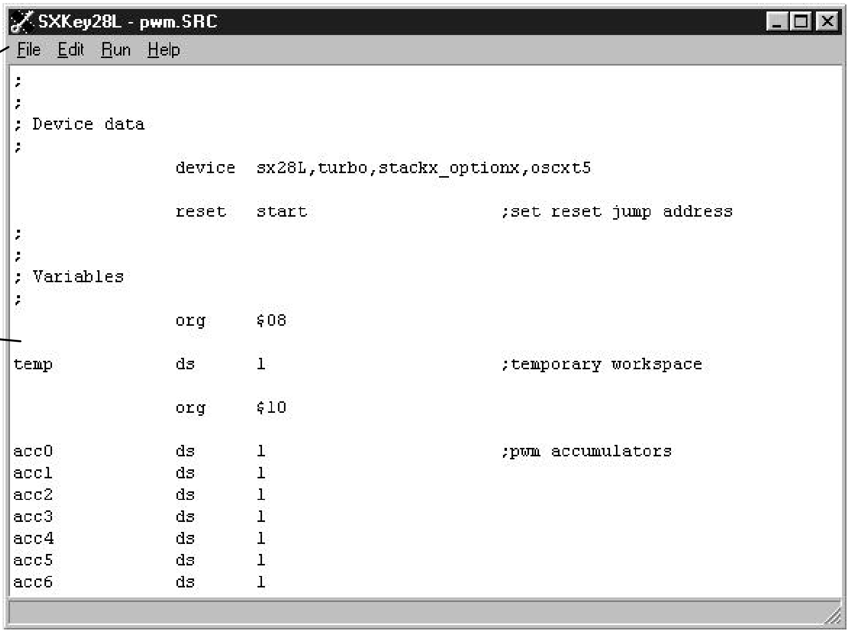
- **Environment:** The SX-Key/Blitz Assembler is an integrated editor, programmer, emulator and debugger interface that runs in Windows 95/98/NT4.
- **New Mnemonics:** Several new mnemonics are available to support new opcodes in the SX chips. The new mnemonics are: 1) RET, 2) RETP, 3) RETI, 4) RETIW, 5) PAGE, 6) BANK, 7) IREAD, 8) MOV M,W, 9) MOV W,M and 10) MOV M,#lit. See Appendix B for more information.
- **Language Syntax:** Some language syntax has changed, although many of the older formats are supported for backward compatibility. For example, it is suggested that you use the $xxxx format, instead of 0xxxxh, to specify hexadecimal numbers.
- **Equates:** Many new dynamic equates are provided for ease of use of the SX microcontroller.
- **@ sign:** The @ sign may be used in all address references to indicate an absolute, flat-architecture, address. The

assembler will handle page swapping for you by inserting the PAGE instruction before your address reference.

- **Device Settings:** All device setting symbols have been updated to describe the SX device more thoroughly. The settings that affect binary fuses indicate an "option active" state. Leaving the setting out of the device line indicates the option is inactive. See Chapter 6 for more information.

5: Introduction to the SX-Key/Blitz Interface

The SX-Key / Blitz interface is an integrated editor, programmer, and debugger. All the functions of the SX-Key and the SX-Blitz are available through this single software interface. Parallax, Inc. has made this interface as simple as possible to ease its use and to help eliminate possible confusion.



```
;
;
; Device data
;
                device  sx28L,turbo,stackx_optionx,oscxt5

                reset   start                   ;set reset jump address
;
;
; Variables
;
                org     $08

temp            ds      1                       ;temporary workspace

                org     $10

acc0            ds      1                       ;pwm accumulators
acc1            ds      1
acc2            ds      1
acc3            ds      1
acc4            ds      1
acc5            ds      1
acc6            ds      1
```

Figure 5.1: The SX-Key/Blitz integrated editor, programmer and debugger.

Throughout the rest of this manual, the SX-Key / Blitz interface will be referred to as the SX editor, or more simply, the editor.

The SX editor (see Figure 5.1, above) consists of one window containing a small menu at the top and a large editor window below. The editor window is where your SX source code may be entered and edited. Standard Windows editing shortcut keys (listed in Table 5.1, below) may be used in addition to commands in the Edit menu to manipulate the source code.

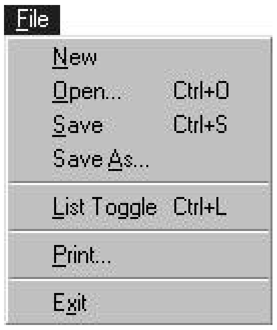
Function Name	Shortcut Keys	Function Description
Copy	Ctrl-C	Copies selected text to the clipboard.
Cut	Ctrl-X	Cuts selected text to the clipboard.
Paste	Ctrl-V	Pastes clipboard contents.
Page Up	Pgup	Moves editor window one page up.
Page Down	Pgdn	Move editor window one page down.
Tab	Tab	Moves cursor to the next tab position. One tab position is set for every 8 characters.
Text Bigger	Ctrl-CrsrUp	Increase size of editor text.
Text Smaller	Ctrl-CrsrDn	Decrease size of editor text.

Table 5.1: Keyboard shortcuts for common editing commands.

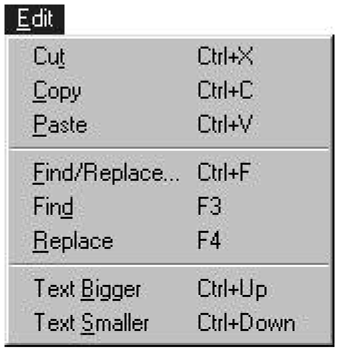
The Menus

The SX editor menu bar contains four menus: File, Edit, Run and Help. These menus and their associated menu items are each defined below.

The File menu:

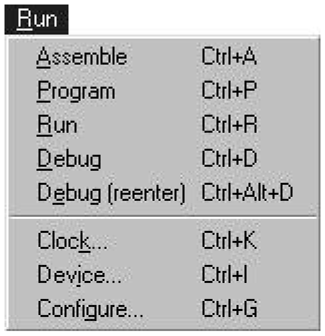


New:	removes any text in the editor window. Use this item to start a new source code editing session.
Open:	opens a browse window to locate and load source code files.
Save:	saves the current source code file.
Save As:	opens a Save As dialog box to save the current source code with a designated name.
List Toggle:	Toggles the editor window between source code view and list view.
Print:	opens a print dialog box to print the current source code.
Exit:	terminates the SX-Key / Blitz editor.



The Edit menu:

Cut:	cuts the selected text from the editor window and stores it in the clipboard.
Copy:	copies the selected text from the editor window and stores it in the clipboard.
Paste:	pastes the text from the clipboard into the editor window starting at the current cursor location.
Find/Replace:	opens the Find and Replace dialog box.
Find:	finds the next occurrence of desired text.
Replace:	replaces the selected text with desired text.
Text Bigger:	makes editor text larger.
Text Smaller:	makes editor text smaller.



The Run menu:

Assemble:	assembles the code.
Program:	assembles the source code and programs the SX microcontroller (if assembly was successful).
Run:	assembles the source code, programs the SX and generates a clock signal.
Debug:	assembles the source code, programs the SX, generates a clock signal and initiates debug mode. (Not used on the SX-Blitz).
Debug (reenter):	assembles the source code, assumes device already programmed with Debug, generates a clock signal and enters debug mode. (Not used on the SX-Blitz).
Clock...:	opens the clock control dialog box to allow modification of the clock activity and frequency. (Not used on the SX-Blitz).
Device...:	opens the device dialog box to allow modification of the SX microcontroller parameters.
Configure...:	opens the configuration dialog box to allow modification of the SX-Key / Blitz programming parameters.

The Help menu:

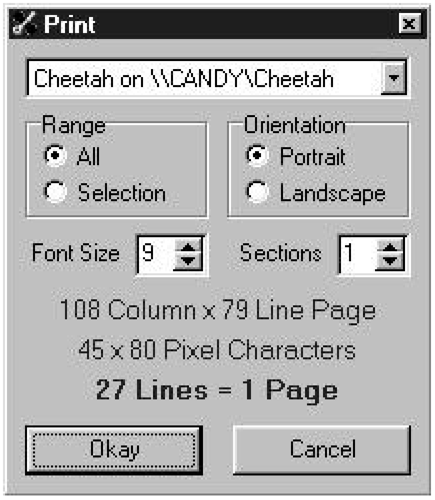
Contents: displays symbolic debugging information.
About: displays the SX-Key/Blitz Development System information box.

The Windows

Many menu items open up a separate window for further configuration or monitoring. These windows are described below.

Print Window

The Print window is accessed via the Print... item on the File menu. This window allows configuration of code for printing.



Cheetah on \\CANDY\Cheetah

Range: [✓ All] Selection

Orientation: [✓ Portrait] Landscape

Font Size 9

Sections 1

108 Column x 79 Line Page

45 x 80 Pixel Characters

27 Lines = 1 Page

Figure 5.2: The Print window.

With the Print window, a printer, print range, orientation, font size and number of sections can be selected. The number of sections determines the number of vertical sections of code that will appear on the printed page. This allows multiple pages of code to fit on the same page.

Find/Replace Window

The Find / Replace window is accessed via the Find / Replace… item on the Edit menu. This window allows searching for and optionally replacing text within the source code.

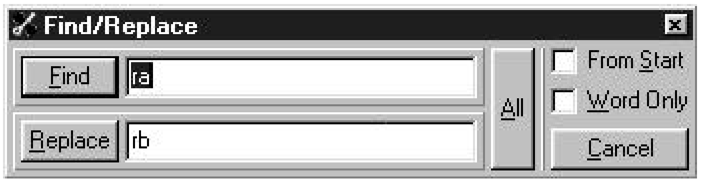


Figure 5.3: The Find/Replace window.

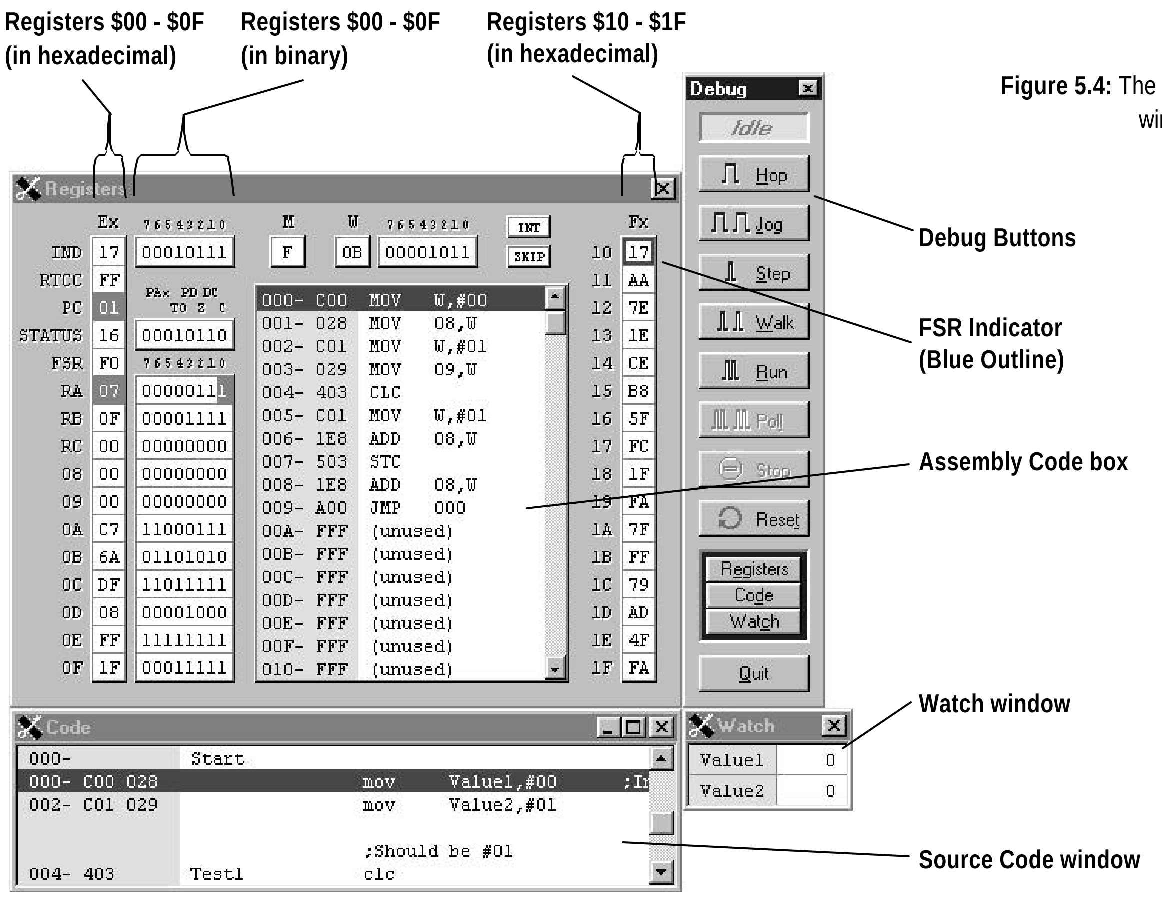
Enter the text to search for in the Find field and, if necessary, the text to replace it with in the Replace field. Select From Start to search from the start of code (deselecting this item searches from current cursor position) and Word Only to match entire words only (deselecting this item causes matching full words and portions of words).

Clicking the Find button finds the next occurrence of desired text. Clicking the Replace button replaces the selected text with the desired text. Clicking All finds and replaces all occurrences of desired text.

Once initiated, subsequent "finds" can be invoked by pressing the F3 key and "replaces" by pressing the F4 key.

Debug Windows

The Debug windows (Registers, Debug, Code, and Watch) are accessed via the Debug and Debug (reenter) items on the Run menu, or by pressing Ctrl-D or Ctrl-Alt-D. These windows display the status of the SX chip and are the interactive control panels for debugging.



	Ex	76543210
IND	17	00010111
RTCC	FF	
PC	01	
STATUS	16	00010110
FSR	F0	
RA	07	00000111
RB	0F	00001111
RC	00	00000000
08	00	00000000
09	00	00000000
0A	C7	11000111
0B	6A	01101010
0C	DF	11011111
0D	08	00001000
0E	FF	11111111
0F	1F	00011111

PAx PD DC TO Z C

M	W	76543210
F	0B	00001011

```
000- C00  MOV   W,#00
001- 028  MOV   08,W
002- C01  MOV   W,#01
003- 029  MOV   09,W
004- 403  CLC
005- C01  MOV   W,#01
006- 1E8  ADD   08,W
007- 503  STC
008- 1E8  ADD   08,W
009- A00  JMP   000
00A- FFF  (unused)
00B- FFF  (unused)
00C- FFF  (unused)
00D- FFF  (unused)
00E- FFF  (unused)
00F- FFF  (unused)
010- FFF  (unused)
```

	Fx
10	17
11	AA
12	7E
13	1E
14	CE
15	B8
16	5F
17	FC
18	1F
19	FA
1A	7F
1B	FF
1C	79
1D	AD
1E	4F
1F	FA

```
000-           Start
000- C00 028                   mov     Value1,#00        ;In
002- C01 029                   mov     Value2,#01

                               ;Should be #01
004- 403       Test1           clc
```

Value1	0
Value2	0

Figure 5.4: The Debug windows.

The Registers window contains all the data and describes the current state of the SX chip. The leftmost column within the Registers window displays the hexadecimal contents of registers $00 through $0F in the currently selected RAM bank. Registers $10 through $1F of all other banks are shown in the columns on the far right of the window. The bank offsets are labeled at the top of each column and the current bank is highlighted in white. These columns will expand or contract to fit the number of RAM banks available in the SX. In this example, one bank is shown.

The blue outline, shown here on the $10 register, indicates the location that the FSR (file select register) is currently pointing at.

The second column, just to the right of the first sixteen registers, displays a binary representation of some of the registers, namely IND, Status, RA, RB, RC and 08 through 0F.

At the top center of the Debug window is the contents of the M register (hexadecimal) and the W register (both hexadecimal and binary) and the Interrupt and Skip flags. The Interrupt and Skip flags turn blue when set and white when cleared.

The assembly code box lists several contiguous instructions at once. The first three digits on each line is the hexadecimal address, followed by the opcode and finally the assembly mnemonic and operand(s). This window normally shows the active section of code, around which the program counter (PC) points, however, the scroll bar allows movement of the window's field of view to any section of code.

The Debug window contains buttons for debug functions. Each button has an associated shortcut key, as described in Table 5.2, below.

- The Hop button will execute one source code instruction (may be multiple machine instructions) and update all registers.
- The Jog button will quickly execute one source code instruction after another, updating the display automatically and continuing until the Stop button is pressed, or a breakpoint is encountered.
- The Step button will execute one machine instruction and update all registers.
- The Walk button will quickly execute one machine instruction after another, updating the display automatically and continuing until the Stop button is pressed, or a breakpoint is encountered.
- The Run button will initiate a full-speed execution of the program, and will continue until a breakpoint is hit or the Poll or Stop buttons are pressed. The displayed registers will not update until the Poll button is pressed or execution is stopped.
- The Poll button operates in one of two modes. If a breakpoint exists, the Poll button runs the code at full speed halting execution

at the break just long enough to update the display and then continues running. If a breakpoint is not set, the Poll button can be pressed only during run mode to get an instant update of the register displays.

- The Stop button halts execution of a jog, walk, run or poll operation and updates the display.
- The Reset button returns the SX chip to its initial state and sets the program counter (PC) to the location containing the reset vector.
- The Registers, Code and Watch buttons bring the associated window into view if they were hidden. (The Debug window is always on top).
- The Quit button closes the Debug windows and exits debug mode.

Table 5.2: Summary of debug buttons and shortcut keys.

Button	Shortcut Keys	Function
Hop	Alt-H	Executes one source code instruction.
Jog	Alt-J	Executes multiple source instructions, one at a time.
Step	Alt-S	Executes one machine instruction.
Walk	Alt-W	Executes multiple machine instructions, one at a time.
Run	Alt-R	Executes instructions in full speed.
Poll	Alt-L	Updates display then continues execution. Can be used synchronously or asynchronously.
Stop	Alt-P	Halts execution of a walk or run operation.
Reset	Alt-T	Resets the SX chip.
Registers	Alt-E	Brings Registers window into view.
Code	Alt-D	Brings Code window into view.
Watch	Alt-C	Brings Watch window into view.
Quit	Alt-Q	Closes the Debug Windows and exits debug mode.

Color Coded Display

The Debug windows are highly active and interactive displays. Every time the display is updated (after a hop, jog, step, walk, run or poll operation), each register that was modified since the previous update is highlighted in red. This provides a clear indication of what the last instruction accomplished. Similarly, each bit that was changed is marked in red within all registers shown in binary. Additionally, the assembly code box and Code window highlights the instruction pointed to by the program counter (PC) in blue, and a breakpoint in red.

Modifying registers during debugging

Any register, bits within registers, or flags can be modified using the mouse and keyboard (see Table 5.3 for a summary of the editing keys). To modify a register (in hexadecimal), first click on it or use the tab and cursor keys to move the focus to that register. (The focus is indicated by a blinking, black highlight within the register). Next, type in the new hexadecimal value on the keyboard and press the enter, space, backspace or arrow keys to write the value to the register. The new value will appear in the selected register, highlighted in red to indicate a change.

To change a bit or flag in the binary registers, simply click the mouse on the appropriate bit. The bit will toggle to the opposite state and will be highlighted in red to indicate a change. Click on the INT or SKIP flags to toggle their state. The INT and SKIP flags, unlike registers, do not indicate a change with a red highlight. Instead, a blue color indicates the flag is set, while a white color indicates the flag is cleared.

If a register's contents are changed by accident, press the ESC (escape) key to restore its previous value.

The Watch window displays the contents of selected registers in a user-defined format. The values in the Watch window can be modified using the same methods described above. In addition, the numerical values in the Watch window can be modified in any format (binary, hexadecimal or decimal) regardless of the displayed format. Simply precede the input value with a %, $, or nothing, respectively. String values can only be modified by entering new strings. See the Watch Directive section in Chapter 6 for information of defining watches.

Table 5.3: Debug window navigation and editing keys.

Key	Function
TAB	Move focus to new control block and ignore any changes to previous register.
Cursor Keys	Move focus to new register within control and write any changes to previous register.
Space	Same as cursor down.
Backspace	Same as cursor up.
Enter	Write changes to register.
ESC	Changes register value to previous value, if it has been changed by the user. This will not work if the enter, space, backspace or cursor keys have been pressed first.

Breakpoints and the current instruction

The assembly code box and Code window displays a breakpoint as a red highlighted line and the next instruction to be executed as a blue highlighted line.

The breakpoint can be set to a new line by clicking the mouse button once on the desired line. Clicking the mouse button again will remove the breakpoint. Only one breakpoint can be set at a time.

Setting the Program Counter

The next instruction to execute can be set to a new line by double-clicking the mouse button on the desired line. Additionally, the program counter (PC) register's contents can be manually modified via the keyboard to set the next instruction to execute.

If a breakpoint and the program counter should both be on the same line, it will become multicolored. The first third of the line will be highlighted in red (to indicate the breakpoint) and the last two thirds of the line will be highlighted in blue (to indicate the next line to execute).

Debugging:

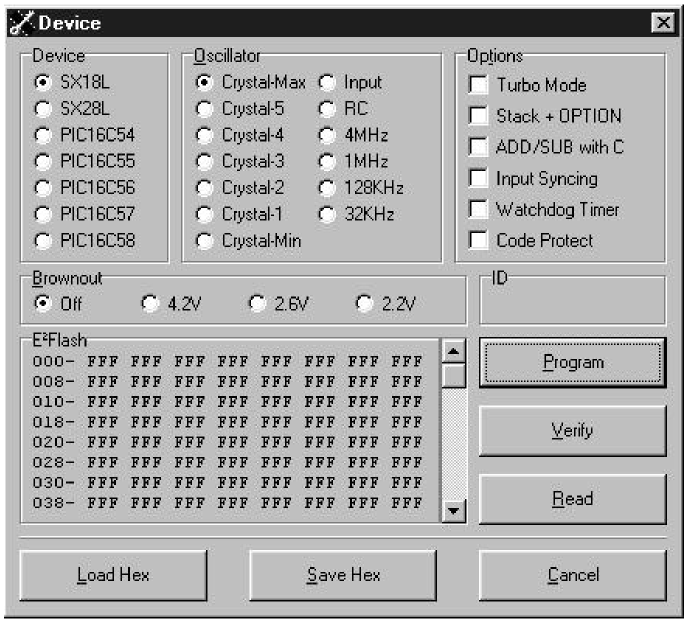
The following is required to use the debug features:

- SX-Key Rev. E (or greater),
- SX chip date code 9825 or later,
- No external clock source connected to the SX chip. This includes oscillator packs, crystals, resonators and RC circuits.

Source code to be debugged must include the RESET directive (see Chapter 6), must have WATCHDOG set to off, and must have 2 free words in the first page of code and 136 free words near the end of the last page of code (from 177 to 1FE, 377 to 3FE, 577 to 5FE, 777 to 7FE, 977 to 9FE, B77 to BFE, D77 to DFE, F77 to FFE), depending on the number of E^{2}Flash pages. If an oscillator frequency of other than 50 MHz (default) is desired, the source code should contain a FREQ directive (see Chapter 6) stating the frequency. **NOTE: On some machines, it is necessary to close background software (graphics, screen savers, etc.), for proper operation of the DEBUG windows.**

Device Window:

The Device window is accessed via the Dev<u>i</u>ce… item on the <u>R</u>un menu, or by pressing Ctrl-I. This window allows you to specify all the device settings for the SX microcontroller you are using, program and verify or read the contents of the device and load or save object files for the SX.



Device: [✓ SX18L] SX28L PIC16C54 PIC16C55 PIC16C56 PIC16C57 PIC16C58

Oscillator: [✓ Crystal-Max] Crystal-5 Crystal-4 Crystal-3 Crystal-2 Crystal-1 Crystal-Min Input RC 4MHz 1MHz 128KHz 32KHz

Options: Turbo Mode, Stack + OPTION, ADD/SUB with C, Input Syncing, Watchdog Timer, Code Protect

Brownout: [✓ Off] 4.2V 2.6V 2.2V

ID

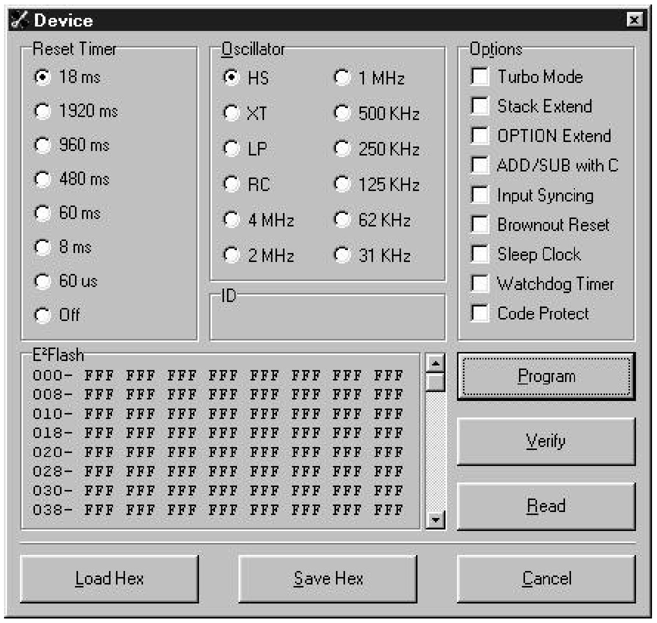
E²Flash

```
000- FFF FFF FFF FFF FFF FFF FFF FFF
008- FFF FFF FFF FFF FFF FFF FFF FFF
010- FFF FFF FFF FFF FFF FFF FFF FFF
018- FFF FFF FFF FFF FFF FFF FFF FFF
020- FFF FFF FFF FFF FFF FFF FFF FFF
028- FFF FFF FFF FFF FFF FFF FFF FFF
030- FFF FFF FFF FFF FFF FFF FFF FFF
038- FFF FFF FFF FFF FFF FFF FFF FFF
```

Figure 5.5: The SX18/28 Device window.

All data shown in this window reflect the settings specified in the DEVICE line(s) of the source code, the settings in the loaded object code or the settings read out of the device itself. You may modify these settings manually and program or reprogram the chip, however, those modifications will not be reflected in the source code.

The Device section (SXKEY28L.EXE) should be set to the SX or the PIC microcontroller that most closely resembles the desired device options. The Reset Timer section (SXKEY52.EXE) should be set to the desired reset delay. The reset delay is the amount of time the SX48/52 waits after a reset condition before executing the first program instruction.



Device

Reset Timer: [✓ 18 ms] 1920 ms 960 ms 480 ms 60 ms 8 ms 60 us Off

Oscillator: [✓ HS] 1 MHz XT 500 KHz LP 250 KHz RC 125 KHz 4 MHz 62 KHz 2 MHz 31 KHz

Options: ☐ Turbo Mode ☐ Stack Extend ☐ OPTION Extend ☐ ADD/SUB with C ☐ Input Syncing ☐ Brownout Reset ☐ Sleep Clock ☐ Watchdog Timer ☐ Code Protect

ID

E²Flash

```
000- FFF FFF FFF FFF FFF FFF FFF FFF
008- FFF FFF FFF FFF FFF FFF FFF FFF
010- FFF FFF FFF FFF FFF FFF FFF FFF
018- FFF FFF FFF FFF FFF FFF FFF FFF
020- FFF FFF FFF FFF FFF FFF FFF FFF
028- FFF FFF FFF FFF FFF FFF FFF FFF
030- FFF FFF FFF FFF FFF FFF FFF FFF
038- FFF FFF FFF FFF FFF FFF FFF FFF
```

Figure 5.6: The SX48/52 Device window.

The Oscillator, Options and Brownout sections specify the configuration of the SX fuses for options like turbo mode, code protect, oscillator type and brown-out reset detection. The oscillator options are different for the SX18/28 and SX48/52. For the SX18/28, the RC, Input and Crystal-Min through Crystal-Max options specify external clock sources and the 32 KHz to 4 MHz options specify internal clock frequency. All options are described in Table 5.4. For the SX48/52 microcontrollers, the HS, XT, LP and RC options specify external clock sources and the 31 KHz – 4 MHz options specify the internal clock frequency. All options are described in Table 5.5.

The ID and E^{2}Flash sections display the values contained in the ID and E^{2}Flash memory (the program memory) respectively.

Programming

The Program button initiates programming the SX chip with the assembled source code, or the object code loaded into the Device window.

Table 5.4: SX18/28 Oscillator Settings.

Setting	Description
Crystal-Min to Crystal-Max	Specifies oscillator drive capacity. Crystal-Min drives the crystal/resonator very lightly; used for low frequencies. Crystal-Max drives the crystal/resonator very hard; used for high frequencies.
Input	Specifies no drive out of OSC2; used when connecting a clock-oscillator pack to OSC1.
RC	Specifies special drive for resistor-capacitor clock circuits.
32 KHz – 4 MHz	Specifies internal clock at indicated frequency.

Table 5.5: SX48/52 Oscillator Settings.

Setting	Description
LP – HS	Specifies external crystal/resonator clock source.
RC	Specifies special drive for resistor-capacitor clock circuits.
32 KHz – 4 MHz	Specifies internal clock at indicated frequency. NOTE: Use IRC Calibration setting of 4 MHz (in Configure window) when programming to ensure greater internal clock frequency accuracy.

Reading and Verifying

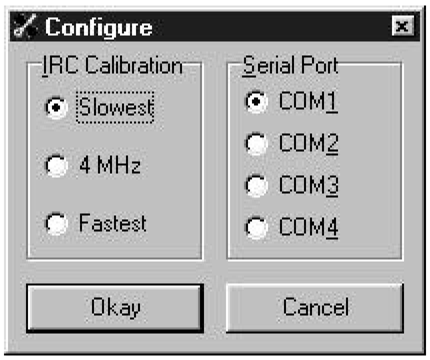
Use the Verify and Read buttons to verify the code in the SX against that shown in the Device window or to simply read the SX's code into the Device window. These options are valuable should the code in an SX chip be questionable or unknown. Note that verifies will fail and reads will not reveal the true code or fuse settings if the SX chip was programmed with the code-protect fuse on. The ID field will always read properly, however.

Loading and Saving Hex Files

The Load Hex and Save Hex buttons may be used to load or save assembled object files. If an object file is desired for a particular source program, simply load the source into the SX-Key editor, assemble it, open the Device window and press the Save Hex button. To program SX chips with an object file, use the Load Hex button on the Device window to load that file and then press the Program button.

Configure Window:

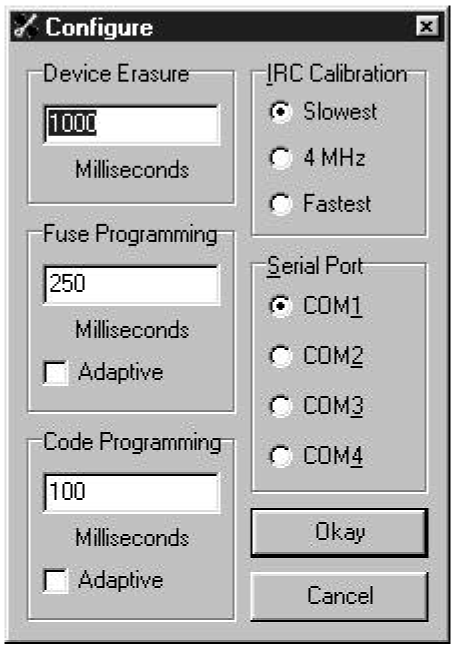
The Configure window is accessed via the Configure… item on the Run menu. This window allows modification of the parameters guiding the programming functions of the SX-Key.



IRC Calibration: [✓ Slowest] 4 MHz Fastest

Serial Port: [✓ COM1] COM2 COM3 COM4

Figure 5.7: The SX18/28 Configure window.



Device Erasure: 1000 Milliseconds

Fuse Programming: 250 Milliseconds ☐ Adaptive

Code Programming: 100 Milliseconds ☐ Adaptive

IRC Calibration: [✓ Slowest] 4 MHz Fastest

Serial Port: [✓ COM1] COM2 COM3 COM4

Figure 5.8: The SX48/52 Configure window.

You may specify which serial port the SX-Key is connected to in the Serial Port section.

The Device Erasure parameter (SXKEY52.EXE) specifies how long, in milliseconds, the SX-Key should perform the erase cycle on the SX48/52 chip.

The Fuse Programming and Code Program parameters (SXKEY52.EXE) specify how long, in milliseconds, the SX-Key should perform the program cycle on each fuse and EEPROM location, respectively.

All time values can be specified up to 5000 ms (5 seconds). The default settings are the suggested settings.

The Adaptive check box (SXKEY52.EXE) can be used to increase programming speed by allowing the SX-Key/Blitz to automatically determine the appropriate program time for each location. When checked, the programming parameter changes from milliseconds to overpulses and indicates the desired number of extra pulses to give each location after the location is first programmed.

IRC Calibration

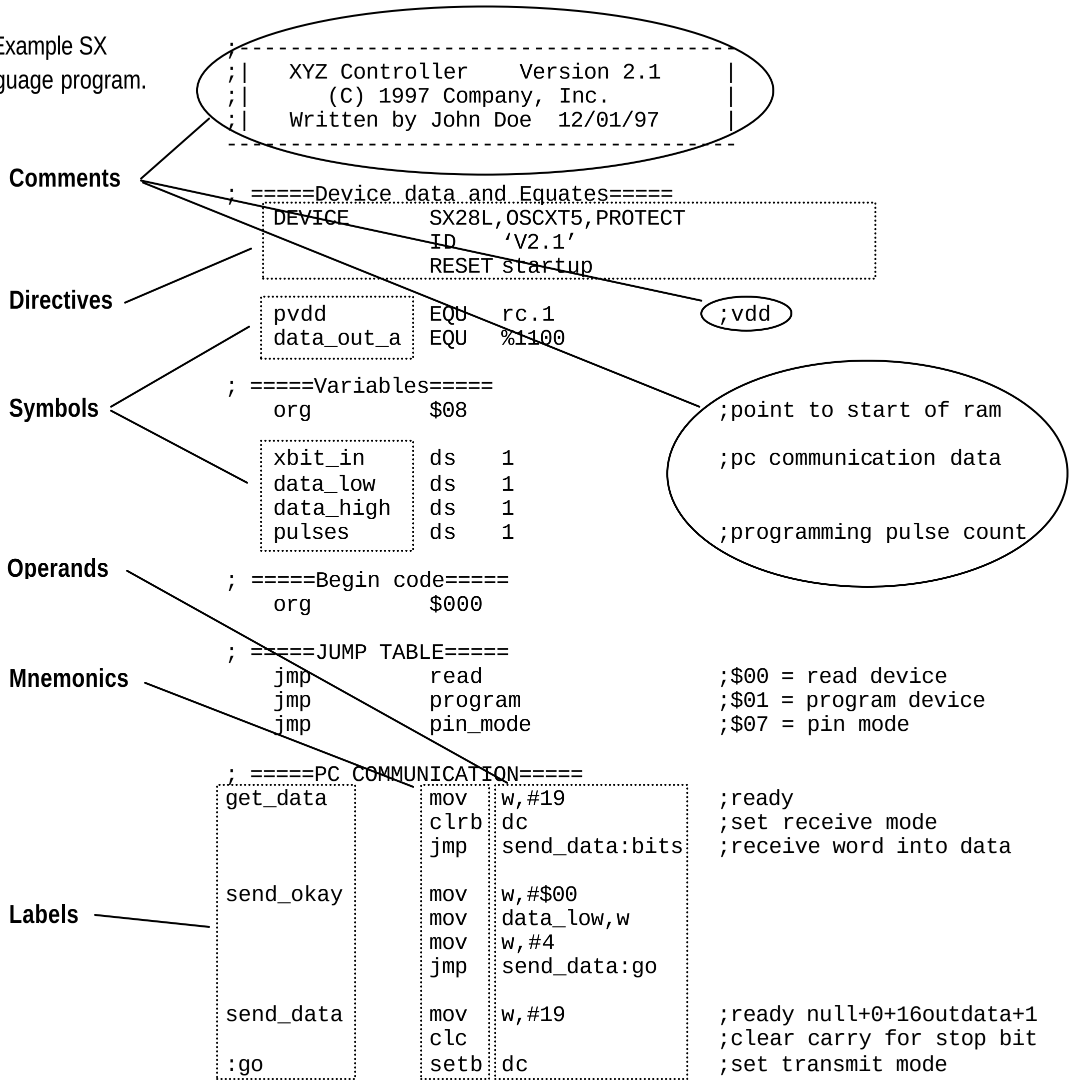
The IRC Calibration section allows setting the special trim feature of the SX-Key. The internal oscillator in the SX chip is based on an RC circuit. Unfortunately, RC circuits suffer timing variances due to component tolerances, temperature and other factors. Because of this, the 4 MHz internal oscillator in the SX chip may vary from frequencies way below the rated 4 MHz to frequencies way above it. This nature may make it difficult to use the internal oscillator for anything other than mundane timing activities.

The SX chip has 3 trim bits to help tune the internal oscillator and the SX-Key can automatically tune it during programming. Selecting 4 MHz in the IRC Calibration section will cause the SX-Key to tune the internal oscillator of every SX chip, during programming, to a value as close as possible to 4 MHz. The actual frequency chosen is displayed after calibration. Selecting the Slowest or Fastest options simply results in the SX-Key setting the trim bits to the slowest or fastest internal oscillator operation possible. Note that the execution speed of the SX program (when using the internal oscillator) is determined by the base oscillation speed, the divide-by ratio (set by the 31 KHz to 4MHz options) and the state of the Turbo fuse.

If not using the internal oscillator, select the Slowest or Fastest options from the IRC Calibration box to speed up the programming process.

A typical SX assembly language program (see Listing 6.1) contains comments, directives, symbols, labels, expressions and mnemonic instructions. This chapter discusses these source code components and demonstrates their uses.

Listing 6.1: Example SX assembly language program.



```
;----------------------------------------
;|   XYZ Controller    Version 2.1      |
;|      (C) 1997 Company, Inc.          |
;|   Written by John Doe  12/01/97      |
----------------------------------------

; =====Device data and Equates=====
    DEVICE      SX28L,OSCXT5,PROTECT
                ID    ‘V2.1’
                RESET startup

    pvdd        EQU   rc.1              ;vdd
    data_out_a  EQU   %1100

; =====Variables=====
    org         $08                     ;point to start of ram

    xbit_in     ds    1                 ;pc communication data
    data_low    ds    1
    data_high   ds    1
    pulses      ds    1                 ;programming pulse count

; =====Begin code=====
    org         $000

; =====JUMP TABLE=====
    jmp         read                    ;$00 = read device
    jmp         program                 ;$01 = program device
    jmp         pin_mode                ;$07 = pin mode

; =====PC COMMUNICATION=====
get_data        mov   w,#19             ;ready
                clrb  dc                ;set receive mode
                jmp   send_data:bits    ;receive word into data

send_okay       mov   w,#$00
                mov   data_low,w
                mov   w,#4
                jmp   send_data:go

send_data       mov   w,#19             ;ready null+0+16outdata+1
                clc                     ;clear carry for stop bit
:go             setb  dc                ;set transmit mode
```

Comments

Comments are optional messages usually used to document the source code. They are ignored by the assembler and may be placed almost anywhere in the program. A comment must be preceded by a semicolon (;). The following demonstrates two examples of comments.

```
;this program controls the GPX513v driver chip
mov        counter, 120        ;initialize loop counter
```

Notice that a comment can be placed on the same line as an instruction (see the second line above). Since the assembler ignores everything that appears to the right of a semicolon, a comment may only appear on its own line or to the right of an instruction.

For debugging purposes, lines of code can be hidden from the assembler, or commented out, simply by inserting a semicolon before the first character of the line.

Assembler Directives

Assembler directives are special instructions to the assembler to help define symbols, set device options or indicate how the code should be assembled. Directives may appear within the source code, like assembly instructions, however, since they are instructions to the assembler and not the SX microcontroller, they are not actually assembled into the final machine language program. Table 6.1 describes available SX-Key Assembler directives.

Table 6.1: SX-Key Assembler Directives.

Directive	Description	Syntax
DEVICE	Sets SX device options. These directives must precede all other directives and instructions.	DEVICE setting {,setting…}
FREQ	Specifies the clock frequency, in Hz, to be generated by the SX-Key during debugging.	FREQ n
ID	Assigns a value to the 8-byte ID word in the SX. The text argument may be up to 8 characters and should be in apostrophes.	ID 'text'
ORG	Specifies the starting RAM or EEPROM location of the code that follows.	ORG value
RESET	Specifies the starting location of the program in the SX's memory.	RESET label
EQU	Assigns a value to a symbol.	symbol EQU value
=	Assigns or reassigns a value to a symbol. This directive is used to create and manipulate assemble-time variables.	symbol = value
DS	Increments the memory pointer ($) by value. Used to reserve RAM and E^2Flash during assembly.	symbol DS value
DW	Defines words in EEPROM.	DW data {,data…}

IF
{ELSE}
ENDIF
	Conditional assembly and alternate conditional assembly block.	IF condition
		{ELSE}
		ENDIF

IFDEF / IFNDEF
{ELSE}
ENDIF
	Conditional assembly and alternate conditional assembly block based upon symbol definitions.	IF{N}DEF symbol
		{ELSE}
		ENDIF

REPT ENDR	Repeat block of code a specified number of times.	REPT count ENDR

MACRO
{EXITM}
ENDM
	Defines a macro.	label MACRO {value}
		{EXITM}
		ENDM

EXPAND / NOEXPAND	Specifies that the following macro instructions should be, or should not be, expanded in the list file.	{NO}EXPAND
CASE / NOCASE	Specifies that the following instructions should be, or should not be, case sensitive.	{NO}CASE
BREAK	Defines a run-time breakpoint.	BREAK
WATCH	Defines a symbol to watch during debugging.	WATCH addr, count, format
ERROR	Defines an assemble-time error.	ERROR 'error text'
END	Marks the end of the source code	END

Device Directive

The device directive is perhaps the most important directive to appear in source code. The device directive specifies the number of pins, EEPROM pages, RAM banks, oscillator type and more. The various symbols for specifying these options are listed in Table 6.2.

The device directive, if supplied, must be the first directive in the code and must appear before the first instruction. Multiple device lines can be used to accommodate many parameters as long as no conflicting parameters are given. The syntax of the device directive is:

```
DEVICE          setting {,setting…}
```

The following device lines tell the assembler that the SX chip to be programmed is an SX 18-pin chip (with 4 EEPROM pages and 8 RAM banks), will use a high speed oscillator that requires drive level 5, will initiate brown-out at 4.2 volts, runs in turbo mode, and is code protected.

```
DEVICE          SX18L, OSCXT5
DEVICE          BOR42, TURBO, PROTECT
```

If a setting is not specified, the default is assumed. In this example, the device will also be set for 2-level stack, 6-bit option register, carry bit ignored, no input synching and no watchdog timer, since the opposing settings were not specified. Similarly, if the source code contains no device directive, the defaults will be used for all possible settings. See Table 6.2 for the defaults.

Table 6.2: Device Directive Settings.

Setting	Description	Default
SX18L SX28L	Specifies the device type, 18 or 28-pin (not used with SX48/52).	SX18L
OSCXTMAX OSCXT5 OSCXT4 OSCXT3 OSCXT2 OSCXT1 OSCXTMIN OSCRC INPUT	External crystal/resonator – Max drive External crystal/resonator – drive level 5 External crystal/resonator – drive level 4 External crystal/resonator – drive level 3 External crystal/resonator – drive level 2 External crystal/resonator – drive level 1 External crystal/resonator – Min drive External RC circuit Crystal pack on OSC1; OSC2 disconnected	OSCXT5
OSCHS OSCXT OSCLP OSCRC	Specifies external crystal / resonator Specifies external crystal / resonator Specifies external crystal / resonator Specifies external RC circuit	OSCHS
DRT18MS DRT1920MS DRT960MS DRT480MS DRT60MS DRT8MS DRT60US DRTOFF	Device reset timer waits 18 ms Device reset timer waits 1,920 ms Device reset timer waits 960 ms Device reset timer waits 480 ms Device reset timer waits 60 ms Device reset timer waits 8 ms Device reset timer waits 60 us Device reset timer is off	DRT18MS
BOR42 BOR26 BOR22	Brownout to trigger at < 4.2 volts Brownout to trigger at < 2.6 volts Brownout to trigger at < 2.2 volts	no brownout
BROWNOUT	Specifies brownout to trigger at 4.2 volts	no brownout
TURBO	Specifies turbo mode (1:1 execution)	compatible mode (1:4 execution)
STACKX_OPTIONX	Stack is extended to 8 levels and Option register is extended to 8 bits	2 levels/6 bits
STACKX	Stack is extended to 8 levels	2 levels
OPTIONX	Option register is extended to 8 bits	6 bits
CARRYX	ADD and SUB instructions use Carry flag as input*	Carry flag ignored
SYNC	Input Syncing enabled	Input Syncing disabled
WATCHDOG	Watchdog Timer enabled	Watchdog disabled
PROTECT	Code Protect enabled	Code Protect disabled
SLEEPCLOCK	Clock is enabled during sleep	No clock during sleep

* Many instructions are adversely affected by the carry flag when CARRYX is specified. See Appendix B for more information.

- Shaded areas indicate SX48/52-only directives.

For backward compatibility with the Super Parallax Assembler, a number of other symbols are accepted for device settings. Table 6.3 lists those device setting symbols. It is recommended that all code developed for use with the SX microcontroller use the new symbols as defined in Table 6.2, above.

Table 6.3: Backward Compatible Device Directive Settings.

Setting	Description	Translation	Default
PIC16C54 PIC16C55 PIC16C56 PIC16C57 PIC16C58	Specifies the number of pins, pages and banks for the device.	’54 emulation using SX18L ’55 emulation using SX28L ’56 emulation using SX18L ’57 emulation using SX28L ’56 emulation using SX18L	SX18L
HS_OSC XT_OSC LP_OSC RC_OSC	external crystal / res. external crystal / res. external crystal / res. external RC circuit	OSCXTMAX OSCXT3 OSCXTMIN OSCRC	HS_OSC
WDT_OFF WDT_ON	Watchdog Timer setting	(no equivalent) WATCHDOG	WTD_OFF
PROTECT_OFF PROTECT_ON	Code Protect setting	(no equivalent) PROTECT	PROTECT_OFF

FREQ Directive

The FREQ (frequency) directive is used to set the frequency (in Hz) of the SX-Key’s internal programmable oscillator to be used during debugging. The syntax for the FREQ directive is:

```
        FREQ            frequency
```

Note that *frequency* can be any number from 400000 to 110000000. Additionally, underscore characters can be used to help make the number more readable, as in 50_000_000 which is 50 MHz.

ID Directive

The ID (identification) directive is used to write up to eight bytes of text into the ID word of the SX chip. This is used to record a version number or other unique identification for the code. This ID word can be read out of the SX chip at any time, regardless of the code protect setting. The line below will write *GPXv2.1* into the ID word:

```
        ID              ‘GPXv2.1’
```

ORG Directive

The ORG (origin) directive tells the assembler the starting location to use for the following instructions. The syntax for the ORG directive is:

```
        ORG             location
```

Note that the ORG directive does not dictate whether the location is in RAM or EEPROM. The assembler simply sets the location pointer as desired and the instructions or directives following the ORG will be processed in relation to this pointer. The ORG directive is used to place data and instructions at specific locations in RAM and E^2Flash.

Reset Directive

The RESET directive specifies the starting address of the code to be executed when a reset condition occurs. The assembler places a 'Jump to Location' instruction at the last location in memory to facilitate this. The syntax of the reset directive is:

```
RESET           location
```

The *location* argument must reside within the first page memory.

EQU Directive

The EQU (equate) directive defines symbols for constants. See the section entitled "Symbols" for further details.

= Directive

The = (equal) directive defines symbols for constants or assemble-time variables. This directive is similar to EQU except that any symbols created with the = directive can be reassigned new values during assemble-time with additional = directives. See the section entitled "Symbols" for further details.

DS Directive

The DS (define space) directive increments the location pointer during assembly. This may be used to cause the assembler to arrange sequential RAM assignments (arrays). For example:

```
ORG             $08
Array           DS   3
Other           DS   2
```

defines 3 bytes, starting at location 8, for the *Array* symbol. *Array+1* is the second element of the array and *array+2* is the third element. The *Other* symbol's first and second elements are placed starting with location $0B. Note that no code is generated in the example above.

DW Directive

The DW (define word) directive defines 12-bit words in EEPROM. This is used to store a data table in the SX EEPROM space. For example:

```
        DW              $FFF, $009, $1A0
```

stores the values $FFF, $009 and $1A0 into EEPROM starting at the current location. See Chapter 7 for more information on using tables.

IF..ELSE Directive

The IF...ELSE directive is used to create conditional assembly blocks. A conditional assembly block is source code that is assembled only if the specified condition is true; otherwise, the code block is ignored by the assembler. Conditional assembly allows for easy code customization for multiple applications. For example, it might be necessary to produce a number of related products all based upon the same main source code but each having a small portion of unique code. The syntax for the IF...ELSE directive is:

```
        IF              condition
                        codeblock
        {ELSE
                        codeblock}
        ENDIF
```

Note that the ELSE block is optional and the ENDIF is required to end the conditional block. The comparison operators for the condition argument are listed in Table 6.4 below.

Table 6.4: Comparison Operators.

Symbol	Operation
=	Equal
<>	Not equal
<	Less than
>	Greater than
=<	Equal or less than
=>	Equal or greater than

The following example demonstrates the use of the IF...ELSE directive.

```
Delay   EQU 10
Mode    EQU 1

IF  Delay >=9
    mov  $08, #5
    add  $09, #%011
ENDIF
```

```
IF  Mode = 0
    mov  $0A, #$1B
ELSE
    mov  $0A, #$1C
ENDIF
```

The condition for an IF...ELSE directive can contain expressions and multiple conditional statements. Two or more conditional statements may be specified by appending them together with the conditional operators NOT, AND, OR and XOR. For example:

```
IF  mode=1 AND delay>9
```

would assemble the code block following it if *mode* equals 1 and *delay* is greater than 9 at assemble time. If the statement or expression evaluates to anything other than zero (0), the condition is true; otherwise, the condition is false.

IF{N}DEF..ELSE Directive

The IFDEF…ELSE (if defined) and IFNDEF…ELSE (if not defined) directives are very similar to the IF…ELSE directive. The difference is they assemble or prevent assembly of code blocks based on whether a symbol is defined or not. For example:

```
DriverOn        EQU         1

IFDEF    DriverOn
{some code block here}
ENDIF
```

would assemble the code block in the IFDEF statement because the *DriverOn* symbol was defined. If *DriverOn* was not defined (ie: line number 1, above, was commented out) the code block would be ignored.

Repeat Directive

The REPT (repeat) directive is used to indicate that a block of code is to be repeated a specified number of times during assembly. The syntax for the REPT directive is:

```
REPT     count
codeblock
ENDR
```

Note that *count* must be greater than 0 and ENDR is required to end the repeat block. For example:

```
REPT    3
add     $0A, #$01
ENDR
```

would result in the source code being expanded to:

```
add     $0A, #$01
add     $0A, #$01
add     $0A, #$01
```

during the assembly of the code.

Within a repeat block, the percent sign (%) alone may be used to refer to the current iteration (1–n) of the block during assembly. For example:

```
REPT    3
add     $0A,#%
ENDR
```

would result in:

```
add     $0A, #1
add     $0A, #2
add     $0A, #3
```

during the assembly of the code. In other words, during assembly, the first time through the repeat block, the % symbol is equal to 1, the second time through it is equal to 2, etc.

Macro Directive

The MACRO directive defines blocks of code that can be reused in multiple places. This is helpful when source code contains numerous sections with similar routines. A macro allows the definition of this routine only once and it can be added to code in multiple places simply by referencing its name. The syntax for the macro directive is:

```
label           MACRO       {argcount}
                codeblock
                {EXITM}
                ENDM
```

Note that *argcount* and EXITM are optional and ENDM is required to end the macro block. *Argcount*, when provided, specifies the exact number of arguments required by the macro and must be in the range 0 to 64. An error will occur during assembly if the macro is invoked with more or less than the required number of arguments. If the *argcount* argument is left out, the macro can accept any number of arguments. To invoke a macro within the code, simply reference it's name followed by the desired arguments. The following example defines a macro, called Add15, which adds the argument + 15 to registers $08 through $0A.

```
Add15           MACRO       1
                mov  $08,#\1+15
                mov  $09,#\1+15
                mov  $0A,#\1+15
                ENDM
```

Within a macro, arguments can be referenced by their index value preceded with a backslash (\), as in the code above. The first argument is \1. The second argument would be \2, etc. If the macro were called with the value 50 as it's first argument, then every occurrence of \1 within the macro would be replaced with 50 at the time of assembly. The reference \0 returns the *number* of arguments passed to the macro. This is useful if a macro has an unspecified number of valid arguments. In fact, inside the macro, a REPT block could be used to cycle through the arguments by using \%.

To use this macro, simply call it by name along with it's required argument at the place in the code where it is needed. For example:

```
            add             ra,#2
            jc              Overflow
            Add15           21
```

would result in the code being expanded, during assembly, to the following:

```
            add             ra,#2
            jc              Overflow
            mov             $08,#36
            mov             $09,#36
            mov             $0A,#36
```

If a macro call requires multiple arguments, specify them in the calling line separated by commas (,).

The expansion of macros can be controlled during list generation with the EXPAND and NOEXPAND directives, see below. The default is to expand all macros within a list file.

The optional EXITM directive can be used to prematurely end macros. This is useful inside an IF…ELSE conditional block within a macro to end the macro at various points based on certain conditions.

Macros may call other macros as well as themselves. This means recursive macro calls are possible.

Expand and noexpand Directives

The EXPAND and NOEXPAND directives specify how to handle macro calls for the purposes of list generation. If a list file is needed that has no macro mneumonics expanded, simply place the noexpand directive above the first macro call. The expand and noexpand directives can be used as often as desired and will only affect the code below them. For example, if source code referenced two macros, M1 and M2, and a list file was needed with only the M1 macro expanded, the expand / noexpand directives might be used as follows:

```
{macro definitions and other code appears above this point}
M1              {arguments here}

NOEXPAND
M2              {arguments here}
EXPAND

M1              {arguments here}
```

The above code will result in a list file with the first and last macro calls expanded and the second macro call unexpanded. Note that expand is the default so the expand directive was not used above the first macro call. Additionally, the list file will always show addresses and assembly within a macro regardless of the use of expand and noexpand directives.

CASE and NOCASE Directives

The CASE and NOCASE directives specify how to handle the character case (upper or lower) of symbols in source code. The CASE directive will cause all the code below it, up to a NOCASE directive, to be case sensitive. The NOCASE directive will cause all code below it, up to a CASE directive, to be case insensitive. The default is case insensitive. The CASE and NOCASE directives can be used as often as desired and will only affect the code below them.

Using case sensitivity will allow symbols with the same name, but different character cases, to be treated as different symbols. For example:

```
CASE
temp    EQU $01
Temp    EQU $02
```

The above code would assemble properly and would have two distinct symbols, *temp* and *Temp.*

NOTE: All directives, instructions and reserved words must be specified in upper case when CASE is active. Using case sensitive mode can be very tricky, can easily lead to wasted time spent debugging, and is not recommended.

Break Directive

The BREAK directive causes a breakpoint to be set at the first line of executable code immediately following it. This is used to set and save a breakpoint in the source code in order to avoid the need to manually set a breakpoint in the Debug window. The syntax of the break directive is:

```
BREAK
breakpoint code
```

where *breakpoint code* is the desired line of source code to break on. The BREAK directive is ignored during any operation other than Debug. Only one breakpoint can be defined at a time.

Watch Directive

The WATCH directive allows the definition of format for viewing and modifying variables at runtime during debug mode. The variable's bit address, number of bits or bytes to view, and display format may be specified. The syntax for the WATCH directive is:

```
WATCH symbol{.bit}, count, format
```

The *symbol* argument can be a symbol name or register address and can optionally specify a bit address within the symbol. If no bit address is specified, bit 0 is assumed. The *count* argument indicates the number of bits (1-32) or bytes (1-16) to include in the displayed value. When using the FSTR or ZSTR format, the *count* is the number of bytes while in all other formats the *count* is the number of bits. Up to 32 WATCH directives can be specified. Table 6.5 lists available format settings for the WATCH directive.

Table 6.5: Format Settings for the WATCH directive.

Format	Operation
UDEC	Displays value in unsigned decimal format
SDEC	Displays value in signed decimal format
UHEX	Displays value in unsigned hexadecimal format
SHEX	Displays value in signed hexadecimal format
UBIN	Displays value in unsigned binary format
SBIN	Displays value in signed binary format
FSTR	Displays values in fixed-length string format
ZSTR	Displays values in zero-terminated format with maximum size

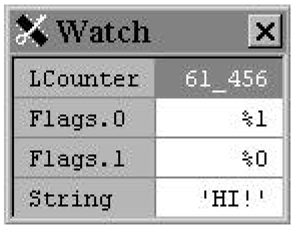
The WATCH directive may be specified anywhere in the source code; however, it is suggested that it be specified near the top, where symbols are defined. When code containing one or more WATCH directives is assembled and programmed using Debug mode, a Watch window appears, along with the other debugging windows, to display the results. The Watch window display is updated at the same time the Registers and Code windows are updated. Typically, WATCH directives are used in conjunction with the BREAK directive, however, asynchronous breaks and polls (with the Poll button) may be used to update the display as well.

The following code snippet is an example of multiple WATCH directives and their output:

```
            LCounter    EQU      $08
            Hcounter    EQU      $09
            Flags       EQU      $0A
            String      EQU      $0B    ;(3 chars, $0B - $0D)
            WATCH       LCounter, 16, UDEC
            WATCH       Flags.0, 1, UBIN
            WATCH       Flags.1, 1, UBIN
            WATCH       String, 3, FSTR

Start       ;Start of main routine
            MOV    LCounter, #$10
            MOV    HCounter, #$F0
            MOV    Flags, #%01
            MOV    String,#'H'
            MOV    String+1, #'I'
            MOV    String+2, #'!'
```

This code snippit would result in a Watch window similar to below:


Watch

LCounter	61_456
Flags.0	%1
Flags.1	%0
String	'HI!'

Figure 6.1: The Watch window.

The first variable in the Watch window is created by the WATCH directive on line 5 of the source code. It tells the SX-Key to display a value starting at bit 0 of LCounter (since bit 0 is assumed if no bit address is specified) containing 16 bits and formatted as an unsigned decimal number. Note that although LCounter is only an 8-bit register, the WATCH directive can span multiple registers (from low-byte to high-byte) in order to construct up to a 32-bit value. In this case, LCounter ($08) is the lower 8 bits and HCounter ($09) is the upper 8 bits of the displayed value (61,456 or $F010).

The second and third variables in the Watch window are created by the WATCH directives on lines 6 and 7 of the source code. Both are single bit values, shown in unsigned binary format. The first of these corresponds to Flags' bit 0 (Flags.0) and the second corresponds to Flags' bit 1 (Flags.1) as specified by the *symbol* arguments.

The fourth variable in the Watch window is created by the last WATCH directive. It tells the SX-Key to display a fixed-length string (FSTR) of 3 bytes starting with the String symbol. In this case, 'HI!' is displayed.

If the ZSTR format is used, the Watch window will display all bytes up to a byte equal to zero, or up to the maximum length (specified by the *count* argument).

The WATCH directive can not span register banks. If a WATCH directive indicates to use a number of bits or bytes that would take the value across a bank's boundary, such as WATCH $1F, 16, UDEC, the WATCH directive wraps around to the first location in that bank. This situation should be avoided, as the value displayed may not be useful.

The values in the Watch window can be modified just like registers in the Registers window. See Modifying Registers During Debugging in Chapter 5 for more information.

Error Directive

The ERROR directive causes an error message to be displayed in the status bar at assemble-time. This is usually used within a conditional block to halt assembly if a certain condition is not met. The syntax of the error directive is:

```
ERROR          'errormessage'
```

The *errormessage* argument can contain static text messages only. For example, the following line, if reached during assembly, will halt assembly and display *Illegal Argument* on the screen.

```
ERROR          'Illegal Argument'
```

Symbols

Symbols are descriptive names for 32-bit numeric values. Symbol names can consist of up to 32 alphanumeric and underscore (_) characters and must start with a letter or underscore. Symbols for constants are usually defined near the start of assembly source code using the equal directive EQU. For example:

```
loop_count   EQU     10
index        EQU     $08
```

defines the symbol *loop_count* to be equal to the number 10, and the symbol *index* to be equal to the hexadecimal number 08. During assembly, everywhere the symbol *loop_count* appears in the code, the number 10 will be used. Similarly, the hexadecimal number 08 will be used in place of the symbol *index* during assembly.

Symbols are a great way to define names for registers and constants that would otherwise appear in many places of a program as just non-descriptive numbers. Assembly code that makes ample use of symbols is easier to read and debug and requires fewer comments.

The = directive also assigns values to symbols, but unlike EQU, can reassign values to those symbols. This directive is useful in macros or repeat blocks to create assemble-time variables. For example:

```
count           =    0
ORG             $20
REPT            5
                DW   count+1*2
                count = count + 3
ENDR
```

stores the values 2, 8, 14, 20 and 26 in memory locations $20 through $24 at assemble-time.

Labels

Labels are descriptive names that are given to sections of code. Similar to symbol names, label names can consist of up to 32 alphanumeric and underscore (_) characters and must start with a letter or underscore. Labels are used to direct code execution to specific routines, and are defined simply by specifying the label name before the routine. For example:

```
main      mov     index, #15
          jmp     main
```

defines the label *main* to point at the first line of code. Later in the program (the second line in this example) to continue execution back at the first line, the jump command is used with the argument *main*. This is usually read as, “jump to main.”

There are three types of labels: global, local, and macro. A global label is one that is unique for the entire length of the code. No two global labels can exist with the same name. A local label is one that is unique for only a portion of the code and must begin with a colon (:). A local label can have the same name as one or more other local labels in a program but they must be separated by at least one global or macro label. A macro label is the name of a macro block (see the MACRO directive, above) and is unique. No macro label can exist with the same name as another macro, global or local label. The following code demonstrates global and local labels.

```
main      mov    loop_count, #15    ;initialize loop_count
:loop     mov    $09, #100          ;set some other register
          djnz   loop_count, :loop  ;decrement loop_count,
                                    ;jump to :loop if not zero
continue  move   loop_count, #50    ;set loop_count to 50
:loop     djnz   loop_count, :loop  ;decrement loop_count,
                                    ;jump to :loop if not zero
          jmp    main ;start over
```

The example above contains two global labels, *main* and *continue*, and two local labels, both named *loop*. The area between the *main* and *continue* global labels is where a local label can exist. Since the djnz instruction in line 3 references the *:loop* label, and line 3 is between the global labels *main* and *continue*, it will only jump to the *:loop* label at line 2 and not the *:loop* label at line 6.

Local and global labels are also allowed within a macro. It is suggested that global labels not be used within a macro, however, as that would prevent the macro from being called more than once.

Expressions

Expressions may be used within the arguments of instructions and directives to calculate values at assemble time. The use of expressions helps build more maintainable, easier to understand code. For example, if a program uses values that are all related to the same base number, it makes sense to include an expression crafted from that relationship. If a symbol *N* is defined as being equal to the base number 2, then N*2+1 and N*3 can be used to derive values related to it; 5 and 6 in this case. At a later time, it might become necessary to adjust the base value to 3 and since expressions were used to derive the related values, only the symbol *N* needs to be modified.

All expressions are evaluated at assemble-time only; they are never reevaluated during run-time. This means symbols for unknowns, such as registers (which change at runtime), can not be included inside expressions.

All constants and symbols within expressions may be 32-bit numbers and the result of the expression is a 32-bit number. Since the SX is only an 8-bit processor, care must be taken to extract the appropriate portion of the result for any particular operation. Table 6.5 and 6.6 describes the available unary and binary operators for use within expressions.

Table 6.5: Unary Operators.

Symbol	Operation
| |	Absolute Value
-	Negative
~	Not
\	Macro assignment

Table 6.6: Binary Operators.

Symbol	Operation
+	Addition
-	Subtraction
*	Multiplication
/	Division
/ /	Modulus
&	Logical AND
|	Logical OR
^	Logical XOR
<<	Shift Left
>>	Shift Right
><	Reverse bits
.	Bit address
( )	Subexpression

Expressions are evaluated strictly from left-to-right, except when parenthesis are present. The following code demonstrates some simple expression usage.

```
loop_count    equ 5*3
table         ds   5+(2*3)
mov           loop_count+10/5, #%01101
```

The second expression, above, will evaluate to 11; the expression within parentheses is evaluated first. The third expression, loop_count+10/5, will evaluate to 5; 15 + 10 = 25 / 5 = 5.

Data Types

The SX-Key/Blitz Assembler understands four data types: decimal, binary, hexidecimal and ASCII. Table 6.7 describes these data types and their syntax.

Table 6.7: Data Types.

Data Type	Syntax	Example
Decimal	xxxx	1250
Binary	%xxxx	%01101010
Hexadecimal	$xxxx	$1AC6
ASCII	‘x’	‘S’

Note: For compatibility with the Super Parallax Assembler, the notation xxxxb and xxxxh is accepted for the binary and hexadecimal data types, respectively. It is suggested that all new SX assembly code use the notation listed in Table 6.7.

Any numbers that evaluate to a result greater than 32-bits wide will cause an error at assemble time. The largest decimal number allowed is 4_294_967_295. The largest hexadecimal number allowed is $FFFF_FFFF. The largest binary number allowed is %1111_1111_1111_1111_1111_1111_1111_1111.

7: SX Special Features and Coding Tips

Introduction

The SX chip offers many configurable features. This chapter explains how to use many of these features, offers coding tips and demonstrates most topics with sample code. Many of the code snippits can be found in the Help_Code directory of the SX-Key/Blitz Development System diskette. All examples are written for the SX18/28 chips and may need modification for SX48/52 chips.

Port Configuration and Usage

There are many configuration options for each of the ports on the SX chip as shown in Table 7.1. The following sections explain how to use the various port configuration options.

Table 7.1: Port configuration options.

	Port A	Port B	Port C	Port D	Port E
Input/Output	X	X	X	X	X
Pull-Ups	X	X	X	X	X
CMOS/TTL	X	X	X	X	X
Schmitt-Trigger		X	X	X	X
Edge-Interrupts		X			
Comparator		Three Pins			

To set these functions, a special form of the MOV instruction, called the port configuration instruction, is used to modify the port configuration registers. The syntax of this instruction is:

```
        MOV             !port, src
```

By default, the port configuration instruction writes to the port direction registers, called the tristate registers. To write to other registers, the MODE register must be preset with a specific value. Table 7.2 lists these values. See Appendix F for more information.

MODE	Port A	Port B	Port C	Port D	Port E
$0F	TRIS_A	TRIS_B	TRIS_C	TRIS_D	TRIS_E
$0E	PLP_A	PLP_B	PLP_C	PLP_D	PLP_E
$0D	LVL_A	LVL_B	LVL_C	LVL_D	LVL_E
$0C		ST_B	ST_C	ST_D	ST_E
$0B		WKEN_B			
$0A		WKED_B			
$09		Swap W with WKPEN_B			
$08		Swap W with COMP_B			
$07 - $00					

NOTE: More options exist for the SX48/52 parts. See Appendix F for details.

Table 7.2: Mode value definitions.

Port Direction

Each of the I/O pins in each of the ports can be configured to an input or output direction by writing to the appropriate tristate register (TRIS_A, TRIS_B, TRIS_C, TRIS_D and TRIS_E). The default I/O pin direction is input. I/O pin direction configuration is usually done once, near the start of code, however, the pin directions can be changed multiple times at any place in the code.

To configure the direction of the I/O pins to inputs or outputs:

1) Set the MODE register to $0F (the default value at startup).
2) Use the port configuration instruction to set the individual directions of each I/O pin within each port. A high bit (1) sets the corresponding pin to input mode and a low bit (0) sets the pin to output mode.

The following code snippet demonstrates this:
(See Help_Code\Port_Direction.src on the diskette for full example)

```
;Direction Configuration
MODE   $0F              ;Set Mode to allow Direction configuration
MOV    !ra,#%0000       ;Port A bits 0-3 to output
MOV    !rb,#%11110000   ;Port B bits 4-7 to input, bits 0-3 output
MOV    !rc,#%00001111   ;Port C bits 4-7 to output, bits 0-3 input
```

If the logic-level of output pins are expected to begin at a certain state (0 or 1), care should be taken to set the output latch appropriately before setting the pin's direction to output. Failing to do so may result in a momentary glitch on the pin during initialization. For example, if all output pins were expected to begin in a low state (0), insert the following lines above the previous code snippet:

```
;Set output pins low
MOV     ra, #%0000          ;Port A bits 0-3 low
MOV     rb, #%00000000  ;Port B bits 0-7 low
MOV     rc, #%00000000  ;Port C bits 0-7 low
```

Pull-up Resistors

Every I/O pin has optional internal pull-up resisters that can be configured by writing to the appropriate pull-up register (PLP_A, PLP_B, PLP_C, PLP_D and PLP_E). By configuring pull-up resisters on input pins, the SX chip can be connected directly to open/drain circuitry without the need for external pull-up resisters. The internal pull-up resisters are disabled by default. Pull-up resisters can be activated for all pins, regardless of pin direction but really matter only when the associated pin is set to input mode.

To configure the I/O pins to have internal pull-up resisters:

1) Set the MODE register to $0E (the value for pull-up register configuration).
2) Use the port configuration instruction to set the individual pull-up state of each I/O pin within each port. A high bit (1) disables the pull-up for the corresponding pin and a low bit (0) enables the pull-up resister for a pin.
3) Set I/O pin directions as necessary.

The following code snippet demonstrates this:
(See Help_Code\Pull-Up.src on the diskette for full example)

```
;Pull-Up Resistor Configuration
MODE   $0E            ;Set Mode for Pull-Up Resistor config.
MOV    !ra,#%0000     ;Port A bits 0-3 to pull-ups
MOV    !rb,#%11110000 ;Port B bits 4-7 normal, bits 0-3 pull-ups
MOV    !rc,#%00001111 ;Port C bits 4-7 pull-ups, bits 0-3 normal
MODE   $0F            ;Set Mode to allow Direction configuration
MOV    !ra,#%1111     ;Port A bits 0-3 to input
MOV    !rb,#%00001111 ;Port B bits 4-7 to output, bits 0-3 input
MOV    !rc,#%11110000 ;Port C bits 4-7 to input, bits 0-3 output
```

Logic Level

Every I/O pin has selectable logic level control that determines the voltage threshold for a logic level 0 or 1. The default logic level for all I/O pins is TTL but can be modified by writing to the appropriate logic-level register (LVL_A, LVL_B, LVL_C, LVL_D and LVL_E). The logic level can be configured for all pins, regardless of pin direction, but really matters only when the associated pin is set to input mode. By configuring logic levels on input pins, the SX chip can be sensitive to both TTL and CMOS logic thresholds. Figure 7.1 demonstrates the difference between TTL and CMOS logic levels.

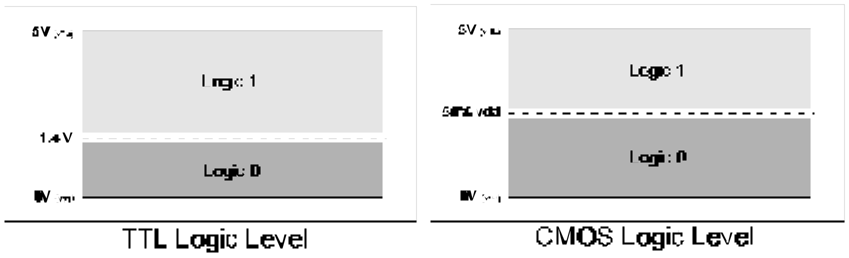


Figure 7.1: TTL and CMOS Logic Levels

The logic threshold for TTL is 1.4 volts; a voltage below 1.4 is considered to be a logic 0, while a voltage above is considered to be a logic 1. The logic threshold for CMOS is 50% of vdd, a voltage below ½vdd is considered to be a logic 0, while a voltage above ½vdd is considered to be a logic 1.

To configure the I/O pins to use CMOS- or TTL-level logic:

1) Set the MODE register to $0D (the value for logic-level register configuration).
2) Use the port configuration instruction to set the individual logic-level state of each I/O pin within each port. A high bit (1) sets the corresponding pin to TTL-level logic and a low bit (0) sets it to CMOS-level logic.
3) Set I/O pin directions as necessary.

The following code snippet demonstrates this:
(See Help_Code\Logic_Level.src on the diskette for full example)

```
;Logic Level Configuration
MODE   $0D            ;Set Mode to Logic Level configuration
MOV    !ra,#%0000     ;Port A bits 0-3 to CMOS
MOV    !rb,#%11110000 ;Port B bits 4-7 to TTL, bits 0-3 CMOS
MOV    !rc,#%00001111 ;Port C bits 4-7 to CMOS, bits 0-3 TTL
MODE   $0F            ;Set Mode to allow Direction configuration
MOV    !ra,#%1100     ;Port A bits 0-1 to output, bits 2-3 input
MOV    !rb,#%10110011 ;Port B bits 2,3,6 to output, all others input
MOV    !rc,#%11011110 ;Port C bits 0,5 to output, all others input
```

Schmitt-Trigger

Every I/O pin in port B through port E can be set to normal or Schmitt-Trigger input. This can be configured by writing to the appropriate Schmitt-Trigger register (ST_B, ST_C, ST_D and ST_E). The I/O pins are set to normal input mode by default. Schmitt-Trigger mode can be activated for all pins, regardless of pin direction but really matter only when the associated pin is set to input mode. By configuring Schmitt-Trigger mode on input pins, the SX chip can be less sensitive to minor noise on the input pins. Figure 7.2 details the characteristics of Schmitt-Trigger inputs.

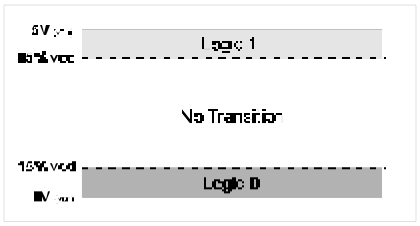


Figure 7.2: Schmitt-Trigger Input Levels.

Schmitt-Trigger inputs are conditioned with a large area of hysteresis. The threshold for a logic 0 is 15% of vdd and the threshold for a logic 1 is 85% of vdd. The input pin defaults to an unknown state until the initial voltage crosses a logic 0 or logic 1 boundary. A voltage must cross above 85% of vdd to be interpreted as a logic 1 and must cross below 15% of vdd to be interpreted as a logic 0. If the voltage transitions somewhere between the two thresholds, the interpreted logic state remains the same as the previous state.

To configure the I/O pins to Schmitt-Trigger input:

1) Set the MODE register to $0C (the value for Schmitt-Trigger register configuration).
2) Use the port configuration instruction to set the individual Schmitt-Trigger state of each I/O pin within each port. A high bit (1) sets the corresponding pin to normal and a low bit (0) sets it to Schmitt-Trigger.
3) Set I/O pin directions as necessary.

The following code snippet demonstrates this:
(See Help_Code\Schmitt-Trigger.src on the diskette for full example)

```
;Schmitt-Trigger Configuration
        MODE   $0C               ;Set Mode to Schmitt Trigger configuration
        MOV    !rb,#%11110000    ;Port B bits 4-7 to normal, bits 0-3 to S.T.
        MOV    !rc,#%00001111    ;Port C bits 4-7 to S.T., bits 0-3 normal
        MODE   $0F               ;Set Mode to allow Direction configuration
        MOV    !rb,#%10110011    ;Port B bits 2,3,6 to output, all others input
        MOV    !rc,#%11011110    ;Port C bits 0,5 to output, all others input
```

Edge Detection

Every I/O pin in port B can be set to detect logic level transitions (rising edge or falling edge). This can be configured by writing to the Edge Selection register (WKED_B) and detected by monitoring the Pending register (WKPEN_B). The I/O pins are set to detect falling edge transitions by default. By configuring edge detection on input pins, the SX chip can set the pin's associated bit in the Pending register when the desired edge arrives. The Pending register bits will never be cleared by the SX alone; the running program is responsible for doing so. This means, if a desired edge is detected, the flag indicating this will remain set until the program has time to attend to it. This feature can be used by the SX program for signals that need attention, but not necessarily immediately.

To configure the I/O pins for edge detection:

1) Set the MODE register to $0A (the value for Edge Detect register configuration).
2) Use the port configuration instruction to set the individual edge to detect on each I/O pin. A high bit (1) sets the corresponding pin to falling-edge detection and a low bit (0) sets it to rising-edge detection.
3) Set I/O pin directions as necessary.

The following code snippet demonstrates this:
(See Help_Code\Edge_Detection.src on the diskette for full example)

```
        ;Edge Detection Configuration
Start
        MODE    $0A             ;Set Mode to allow Edge configuration
        MOV     !rb,#%11111111  ;Port B bits 0-7 to falling edge
        MODE    $0F             ;Set Mode to allow Direction configuration
        MOV     !rb,#%11111111  ;Port B bits 0-7 to input

Main
        MODE    $09             ;Set Mode for Pending register check
        MOV     !rb,#%00000000  ;This line moves WKPND_B to W and
                                ;writes all zeros to WKPND_B
        JMP     Main            ;At this point, W should be checked for
                                ;high bits, indicating a falling edge occurred
```

The following are points to remember with edge detection:

- The edge detection feature is always enabled and the Pending register is always updated even if the SX program does not configure or use it.
- It is up to the SX program to clear the bits of the Pending register when detection of a future transition is desired. The MOV !rb,#%00000000 instruction effectively clears all bits of the Pending register at the same time that it stores the current edge detection status in W.
- If the SX program is designed to handle only one edge detection event at a time (on two or more pins), it will be necessary to get the status (as shown above), clear only the bit being attended to and move the modified status back to the Pending register.
- An edge detection event will not wake up the SX chip from a SLEEP mode unless the Wake-Up Enable mode is also set. See below for more information.

Wake-Up (Interrupt) on Edge Detection

Every I/O pin in port B can be set to cause an interrupt upon logic level transitions (rising edge or falling edge). By configuring interrupts on input pins, the SX chip can respond to signal changes in a quick and deterministic fashion. In addition, an interrupt of this sort will wake up the SX chip from a SLEEP state. This can be configured by writing to the Edge Selection register (WKED_B) and the Wake-Up Enable register (WKEN_B) and detected by monitoring the Pending register (WKPEN_B) in the interrupt routine. The I/O pins have interrupts disabled and are set to detect falling edge transitions by default.

As with edge selection, the Pending register bits will never be cleared by the SX alone; the running program is responsible for doing so. This means if a desired edge is detected, the interrupt will occur and the flag indicating this will remain set until the program clears it. Additional transitions on that pin will not cause interrupts until the associated bit in the Pending register is cleared.

To configure the I/O pins for wake-up (interrupt) edge detection:

1) Set I/O pin edge detection as desired. (See Edge Detection, above, for more information).
2) Set the MODE register to $0B (the value for Wake-Up Enable register configuration).
3) Use the port configuration instruction to enable the individual pins for wake-up interrupts. A high bit (1) disables interrupts and a low bit (0) enables interrupts.
4) Set I/O pin directions as necessary.
5) Clear the Pending register to enable new interrupts.

The following code snippet demonstrates this:
(See Help_Code\Wake-Up_Enable.src on the diskette for full example)

```
        RESET  Start

Interrupt                         ;Interrupt routine (must be at address $0)
        ORG    $0
        MODE   $09                ;Set Mode for Pending register
        MOV    !rb,%00000000      ;Clear Pending/get current status in W
        RETI                      ;rest of interrupt routine goes here

        ;Wake-Up Edge Detection Configuration
Start
        MODE   $0A                ;Set Mode to allow Edge configuration
        MOV    !rb,#%11111111     ;Port B bits 0-7 to falling edge
        MODE   $0B                ;Set Mode to allow Wake-Up configuration
        MOV    !rb#%11110000      ;Port B bits 4-7 to normal, 0-3 to Wake-Up
        MODE   $0F                ;Set Mode to allow Direction configuration
        MOV    !rb,#%11111111     ;Port B bits 0-7 to input
        MODE   $09                ;Set Mode for Pending register
        MOV    !rb,%00000000      ;Clear register to allow new interrupts
Main
        NOP                       ;rest of main routine goes here
        JMP    Main
```

The following are points to remember with Wake-Up Interrupts:

- The interrupt routine must be located starting at address $0 in the SX program.
- It is up to the SX program to clear the bits of the Pending register when future interrupts on that pin are desired. This should normally be done as part of the interrupt routine. The MOV !rb,#%00000000 instruction effectively clears all bits of the Pending register at the same time that it stores the current edge detection status in W.
- The SX chip will activate the interrupt routine exactly 5 clock cycles in Turbo mode or exactly 10 clock cycles in Compatible mode after a Wake-Up Edge Detection event occurs. This deterministic feature allows for nearly jitter-free interrupt response. Latency may vary by as much as +1 instruction cycle when interrupting on external asynchronous events, thus a high clock speed may be necessary to lessen the effects.
- If multiple interrupt pins are required, the SX chip may not be able to properly process them in certain situations. See Interrupts, below, for more information.
- An edge-detection interrupt event will wake up the SX chip from a SLEEP mode.

See Interrupts, below, for more information.

Comparator

I/O pins 0 through 2 in port B can be set for comparator operation. This can be configured by writing to the EN and OE bits of the Comparator register (CMP_B) and monitored by reading the RES bit. The comparator mode is disabled by default. Comparator mode can be activated for all three pins, regardless of pin direction, but really matters only when pin 1 and 2 are set to input mode (pin 0 can optionally be set to output the comparative result). By configuring Comparator mode, the SX chip can quickly determine logical differences between two signals and even indicate those differences for external circuitry.

When Comparator mode is activated, the RES bit in the Comparator register indicates the result of the compare. A high bit (1) indicates the voltage on pin 2 is higher than that of pin 1, a low bit (0) indicates the voltage on pin 2 is lower than that of pin 1. If the OE bit (Output Enable) of the Comparator register is cleared, output pin 0 of port B reflects the state of the RES bit.

To configure port B I/O pins 0 though 2 for Comparator mode:

1) Set the MODE register to $08 (the value for Comparator register configuration).
2) Use the port configuration instruction to enable the Comparator and, optionally, the result output on pin 0.
3) Set I/O pin directions appropriately.

The following code snippet demonstrates this:
(See Help_Code\Comparator.src on the diskette for full example)

```
        ;Comparator Configuration
        MODE   $08              ;Set Mode to Comparator configuration
        MOV    !rb,#%00000000   ;Enable comparator and result output
        MODE   $0F              ;Set Mode to allow Direction configuration
        MOV    !rb,#%11111110   ;Port B bits 1-7  to input, bit 0 to input
Main
        MODE   $08
        MOV    !rb,#$00
        JMP    Main             ;Here, bit 0 of W holds result of compare
```

The following are points to remember with Comparator mode:

- Port B I/O pins 1 and 2 are the comparator inputs and I/O pin 0 is, optionally, the comparator result output.
- Port B I/O pin 0 may be used as a normal I/O pin by setting the OE bit of the Comparator register.
- The comparator is independent of the clock source and thus will operate even if the SX chip is halted or in SLEEP mode. To avoid spurious current draw during SLEEP mode, disable the comparator.

SX Special Features and Coding Tips

All About Interrupts

The SX18/28 chip allows for up to nine sources of interrupts; eight external and one internal. The SX48/52 chip allows for up to 17 sources of interrupts; 10 external and one internal. Any or all of the port B I/O pins can be configured as external interrupts. See Wake-Up (Interrupt) on Edge Detection, above, for information on configuring external interrupts. The internal interrupt can be configured to occur upon a rollover condition within the Real Time Clock Counter (RTCC) register. A special return-from-interrupt command may also be used to adjust the value of the RTCC to cause interrupts to occur at a specific time interval. See RTCC Rollover Interrupts, below, for more information.

In addition to the interrupts supported by the SX18/28, the SX48/52 pins RC.0 and RC.1 can be configured as external interrupts and six internal different interrupts can be configured for the Timer 1 and Timer 2 overflow and R1/R2 counter comparison registers.

These interrupt options can be very powerful features but can also cause havoc if not configured or understood properly. If using interrupts of any kind is desired, the following items should be reviewed.

- **Interrupt Vector:** The interrupt vector in the SX chip points to address $0 and is not configurable. The interrupt routine must reside at location $0 to be properly executed upon an interrupt event.

- **Auto Interrupt Disable:** As soon as an interrupt occurs, additional interrupts are automatically ignored by the SX chip until the interrupt routine is completed. This prevents the interrupt routine from being interrupted and prevents the loss of return vector data. This is also one of the most important considerations when working with interrupts; you can not immediately (without jitter) process more than one interrupt at a time. **Note: Should additional interrupts occur, the SX chip does not automatically queue up interrupts for future processing. See Interrupt Queuing, below, for more information.**

- **Latency Delays:** When an interrupt occurs, there is a latency delay before the interrupt routine is actually activated. For the internal RTCC rollover, this latency is exactly 3 clock cycles in Turbo mode and 8 clock cycles in Compatible mode. For the external interrupts, the latency delay is exactly 5 clock cycles in Turbo mode and 10 clock cycles in Compatible mode. Latency may vary by as much as +1 instruction cycle when interrupting on external asynchronous events, thus a high clock speed may be necessary to lessen the effects.

- **Interrupt Routine Size:** Normally it is a requirement for an application to process every interrupt without missing any. To ensure this happens, the longest path through the interrupt routine must take less time than the shortest possible delay between interrupts.

- **Interrupt Queuing:** If an external interrupt occurs during the interrupt routine, the pending register will be updated but the trigger will be ignored unless interrupts had first been turned off at the beginning of the routine and turned on again at the end. This also requires that the new interrupt doesn't occur before interrupts are turned off in the interrupt routine. If there is a possibility of extra interrupts occurring before they can be disabled, the SX will miss those interrupt triggers.

- **Multiple Interrupts:** Using more than one interrupt, such as multiple external interrupts or both RTCC and external interrupts, can result in missed or, at best, jittery interrupt handling should one occur during the processing of another.

- **Clearing Pending Bits:** When handling external interrupts, the interrupt routine should clear at least one pending register bit. The bit that is cleared should represent the interrupt being handled in order for the next interrupt to trigger.

- **Debugging Interrupts:** The SX chip may act strangely while debugging code that contains interrupts. The SX chip may or may not enter the RTCC interrupt routine (and will never enter a MIWU interrupt) while using the Hop, Jog, Step or Walk functions. This is due to the SX chip giving higher priority to the SX-Key than its internal interrupt flags. **If interrupt code needs to be debugged or verified, place a BREAK directive, or a breakpoint, in an appropriate place within the interrupt routine and use the Run or Poll functions.**

RTCC Rollover Interrupts

The SX chip can be set to cause an interrupt upon rollover of the Real Time Clock Counter (RTCC). By configuring an interrupt on RTCC rollover, the SX chip can perform an operation at a predefined time interval in a deterministic fashion. This can be configured by setting the STACKX_OPTIONX fuse (in the DEVICE directive) and writing to the RTI, RTS and RTE bits of the Option register (OPTION). The RTCC rollover interrupt is disabled by default.

To configure the RTCC rollover interrupt:

1) Set the STACKX_OPTIONX fuse in the DEVICE line.
2) Write to the RTI, RTS and RTE bits of the OPTION register to enable RTCC interrupts. For RTI, a high bit (1) disables RTCC rollover interrupts and a low bit (0) enables RTCC rollover interrupts. For RTS, a high bit (1) selects incrementing RTCC on internal clock cycle and a low bit (0) increments RTCC on the RTCC pin transitions. For RTE, a high bit (1) selects incrementing on low-to-high transition and a low bit (0) increments on a high-to-low transistion.

The following code snippet demonstrates this:
(See Help_Code\RTCC_Interrupt.src on the diskette for full example)

```
        DEVICE STACKX_OPTIONX
        RESET  Start

Interrupt                       ;Interrupt routine (must be at address $0)
        ORG    $0
        RETI                    ;rest of interrupt routine goes here

        ;RTCC Rollover Interrupt Configuration
Start
        MOV    !OPTION,#%10011111    ;Enable RTCC rollover interrupt
                                     ;RTCC inc on clock, no prescale
```

```
Main
        NOP                     ;rest of main routine goes here
        JMP     Main
```

The above code will cause the interrupt routine to be executed once every 256 clock cycles (when RTCC rolls over from 255 to 0). A different return-from-interrupt command called RETIW can be used, however, to customize the time interval (cycle interval) in which the interrupt executes. RETIW, like RETI, causes a return from the interrupt routine. RETIW has the additional effect of adding the contents of W to the RTCC register upon return. By moving a negative number into W just before executing an RETIW, the RTCC will be backed-off by the designated number of cycles. This method also has the benefit of compensating for the number of cycles spent in the interrupt routine.

For example, if the interrupt routine should be executed once every 50 cycles, use the following two lines of code in place of the RETI command in the listing above:

```
        MOV     W,#-50
        RETIW
```

Of course, for this to work properly the interrupt routine must take 46 cycles or less (see below for cycle bandwidth calculation). Even if the interrupt routine contained multiple paths of execution, due to compare-jump instructions, and each path consumed a different number of clock cycles, the interrupt would still execute once every 50 cycles. Table 7.3 demonstrates the effects on the RTCC if the interrupt routine contained two possible paths of execution (path 1 is 28 cycles and path 2 is 15 cycles):

Table 7.3: Example RTCC values during interrupts.

Event	Path 1 (28 cycles)	Path 2 (15 cycles)
1) RTCC rolls over	RTCC = 0	RTCC = 0
2) 3 cycles required to enter interrupt routine	RTCC = 3	RTCC = 3
3) Interrupt routine executes	RTCC = 31	RTCC = 18
4) –50 (206 in twos-compliment) is added to RTCC	RTCC = 237	RTCC = 224
5) RTCC rolls over again in exactly 19 additional cycles (Path 1) or 32 additional cycles (Path 2)	Total Cycles = 31 + 19 = 50	Total Cycles = 18 + 32 = 50

By adjusting the value in W before the RETIW command, various amounts of cycle bandwidth will be allocated to the main routines. Normally this won't be a problem but care should be taken to ensure that the RTCC doesn't rollover too often, causing little or no cycle time to be allocated to the main routines. If the RTCC adjustment value is too small for the size of the interrupt routine, the main routine may eventually hang up, may not execute at all, or the interrupt routine will miss the rollover and only execute every 256 cycles. Use the following equation as a general rule-of-thumb when determining the minimum adjustment value:

Minimum RTCC Adjustment Value = **-**(max cycles for interrupt + 6)

The *6* in this equation accounts for the number of cycles required to enter the interrupt routine (3 cycles) plus the number of additional cycles needed to complete the longest command (3 cycles extra to finish an IREAD). If the IREAD command is not used in the main program, a value of **-**(maximum cycles for interrupt + 4) is the minimum, allowing for only a single instruction in the main routine to be executed between interrupts.

As an example, the following interrupt routine takes 4 cycles to execute:

```
Interrupt
        MOV     W,#-8           ;1 cycle
        RETIW                   ;3 cycles
```

The adjustment value of –8, which is –(max cycles for interrupt + 4), will cause the interrupt routine to execute every 8 cycles and will only allow one single-cycle or one three-cycle instruction in the main routine to execute between interrupts. If the main routine contained an IREAD command, however, the main routine would execute one instruction between interrupts until it reached the IREAD, at which point it would get eternally stuck, and only the interrupt would continue. As another example, if the adjustment value was –7, this would be too small an adjustment and would cause the interrupt routine to execute, but no instructions in the main routine would execute at all.

The following are points to remember with Wake-Up Interrupts:

- The interrupt routine must be located starting at address $0 in the SX program.
- The interrupt routine should take a maximum of 6 cycles less than the desired cycle time slot. (ie: if the interrupt should execute once every 20 cycles, it needs to be less than 15 cycles in size).
- The SX chip will activate the interrupt routine exactly 3 clock cycles (Turbo) or 8 clock cycles (Compatible) after an RTCC rollover event occurs. This deterministic feature allows for jitter-free interrupt response.
- An RTCC rollover interrupt event will not occur during SLEEP mode and thus can not wake up the SX chip from a SLEEP mode.

Creating Tables

Data Tables

Tables of 8-bit or 12-bit data may be stored in the unused program space of the SX chip. Tables of data may be necessary in cases where a set of data can not be calculated by an equation, or will take too long to calculate. There are two methods available to store data tables in the SX chip.

If only 8-bit data is required, the RETW method may be used to create the data table. This method uses a set of RETW commands which each hold an 8-bit data value as their operand. The table is preceded by a JMP PC+W command. By moving an index value to W and then CALLing the first line of the table, which is the JMP command, W is added to the program counter and upon the next clock cycle, the proper RETW command is executed. RETW simply moves its operand to W and then returns to the line after the CALL.

To create an 8-bit data table with the RETW command:

1) Set the table's location, and insert a label and a JMP PC+W command at the start of the table.
2) Add as many RETW commands as necessary.
3) When data is needed from the table, move the index value of the desired item to W and CALL the table. Upon returning, the 8-bit value is in W.

The following code snippet demonstrates this:
(See Help_Code\RETW_Table.src on the diskette for full example)

```
            RESET   Initialize

            Idx     EQU     $08         ;Define index symbol

            ;8-bit data table
            ORG     $0
Table       JMP     PC+W                ;Jump into the table
            RETW    'ABCDEFG'           ;Store text
            RETW    10,100,255,0        ;Store numbers
Initialize
            MOV     Idx,#$FF            ;Reset table index

Main        INC     Idx                 ;Increment table index
            MOV     W,Idx
            CALL    Table               ;Retrieve data
```

If 8-bit or 12-bit data is required, the DW method may be used to create the data table. This method uses a set of DW (Define Word) directives each of which place a 12-bit data value in program memory. By moving a 12-bit index value to M and W (upper 4 bits of address in M) and executing an IREAD command, M and W are replaced with the 12-bit data value.

To create an 8-bit or 12-bit data table with the DW directive:

1) Set the table's location and insert a label.
2) Add as many DW directives as necessary.
3) When data is needed from the table, move the index value of the desired item (in relation to the label) to M and W and execute an IREAD. Upon returning, the 12-bit value is in M and W.

The following code snippet demonstrates this:
(See Help_Code\IREAD_Table.src on the diskette for full example)

```
            RESET  Initialize

            Idx    EQU   $08          ;Define index symbol
            ;8-bit data table
            ORG    $0
Table       DW     'ABCDEFG'          ;Store text
            DW     10,300,4095,0      ;Store numbers

Initialize
            MOV    Idx,#Table         ;Reset table index

Main        MOV    M,#Table>>8        ;upper 4-bits of table address
            MOV    W,Idx              ;lower 8-bits of table index
            IREAD                     ;Retrieve data
            {use the data}
            INC    Idx                ;Increment table index
            CJNE   Idx,#11,Main       ;Continue
```

Both table methods shown above will only access a maximum of 256 elements, however, the DW method can easily be modified to access every possible address. If speed is desired, the DW method, above, is 4 cycles shorter per element than the RETW method. With clever modifications, both methods could be used for more than 8 or 12-bit2.

Dealing with Code Pages

Branching Across Pages

The SX chip's program memory is organized into pages of 512 words each. If a program won't fit within a single page, special care must be taken when branching across page boundaries to avoid misdirected jumps.

All instructions that perform a jump, except JMP W, JMP PC+W and LJMP, use a 9-bit address as the operand. This limits the jump range to the current page only (512 words). To jump across a page boundary, the page select bits (the upper bits of the Status register) must first be set to the appropriate page before executing the jump. The SX assembler provides a convenient method of doing this, as shown below.

To jump across page boundaries:

1) Specify the jump command (JMP, JB, CJE, etc) with the page-set option (the @ sign preceding the address).

The following code snippet demonstrates this:
(See Help_Code\Page_Jumping.src on the diskette for full example)

```
Start   ORG   $0          ;This routine is in page 0
        JMP   @Routine    ;Jump to proper page
        JMP   Start

Routine ORG   $200        ;This routine is in page 1
        JMP   @Start      ;Jump back to proper page
```

The @ symbol preceding the address causes the SX editor to insert a PAGE instruction just before the JMP to set the page select bits appropriately. The second JMP, in line three, does not require an @ symbol since the destination address is within the current page. If the @ was left out of line two, the SX would jump to address $000 instead of $200. See Jumping Across Pages in Appendix F for more information.

Calling Across Pages with Jump Tables

Calling subroutines can pose even more boundary problems than jumping across pages. The CALL instruction uses only an 8-bit address as the operand (the 9th bit of the address is always cleared). This limits the calling destination to the first 256 words of the current page.

Because it is sometimes impossible to organize all subroutines within such a tight space, a common practice is to make use of a subroutine jump table. The jump table consists of a list of JMP commands to various subroutines and is located within the first 256 words of the page. The CALL instructions can simply call the proper location within the jump table and code execution jumps to the appropriate subroutine, even if it exists in different pages or above the 256 word barrier.

To call subroutines across page boundaries:

1) Design a jump table with the page-set option (the @ sign preceding the addresses).
2) Place the subroutines in any desired location being sure to end them with RETP.
3) Call the subroutine's alias-name in the jump table.

The following code snippet demonstrates this:
(See Help_Code\Page_Calling.src on the diskette for full example)

```
        ORG     $0              ;This routine is in page 0
        ;Define the subroutine jump-table
Sub1    JMP     @_Sub1          ;Set page and jump
Sub2    JMP     @_Sub2

        ;Start of main routines
Start   CALL    @Sub1           ;Call the Jump Table
        JMP     @Continue

        ORG     $200            ;This routine is in page 1
Continue
        CALL    @Sub2           ;Call the Jump Table
        JMP     @Start

        ORG     $400            ;This routine is in page 2
_Sub1   ;Subroutine 1 code goes here
        RETP                    ;Return and reset page

_Sub2   ;Subroutine 2 code goes here
        RETP                    ;Return and reset page
```

The first CALL in the Start routine calls the *Sub1* address in the jump table. The JMP command at *Sub1* then jumps to the *_Sub1* subroutine (in page 2) which eventually returns to the line following the CALL. The RETP command used to return from the subroutine resets the page select bits to the page of the calling routine (exactly as intended).

The @ symbol preceding the addresses causes the SX editor to insert a PAGE instruction just before the JMP and CALL commands to set the page select bits appropriately. The first CALL, in the Start routine, would function the same without an @ symbol, as shown above, since the destination address is within the current page. See Calling Across Pages in Appendix F for more information.

Appendix A: SX Features

Introduction

The SX chip is a fully static CMOS MPU conservatively rated for DC to 50 MHz operation. The SX provides 2K words of on-chip EE/Flash program memory (4K words in SX48/52). In Turbo mode, all instructions are single cycle except program branches, which take three cycles, and IREAD, which takes four cycles.

CPU Features

- Single cycle instruction execution (20 ns cycle time @ 50 MHz)
- DC to 50 MHz operation (75 MHz on selected chips)
- User selectable clock options: (Internal R/C, External R/C, resonator, crystal oscillator or crystal-oscillator pack)
- Internal R/C oscillator (31 KHz to 4 MHz, +/- 8% accuracy)
- 43 instructions (designed for compatibility with the PIC16C5x series; 10 new instructions for improved code efficiency)
- 2048 x 12-bits (4096 x 12-bits in SX48/52) EE/Flash program memory rated for 10,000 rewrite cycles
- Up to 137 bytes (262 bytes in SX48/52) of directly, or indirectly, addressable RAM
- Selectable 8-level hardware stack
- Fixed interrupt response time: 60 ns internal, 100 ns external
- Hardware context save/restore of PC, W, STATUS, and FSR on interrupt.
- Multi-Input Wake-Up (MIWU) on 8 pins
- In-system programming via OSC pins
- Single-step and breakpoint debugging via OSC2 pin
- Analog comparator (RB0 out, RB1 in-, RB2 in+)
- Built-in brown-out detector (On/Off, 4.2V) (4.2, 2.6, 2.2, Off in SX48/52)
- W mappable into RTCC space for flexibility
- Nine sources of interrupts (17 in SX48/52)
- 1998 UL compliance and fast lookup provided through run-time readable code

Peripheral and I/O Features

- Every pin programmable as input or output

SX Features

- Inputs are each TTL or CMOS level selectable
- All pins include selectable internal pull-ups (~20 kΩ to V_{DD})
- RB, RC, RD and RE inputs each selectable as Schmitt Trigger
- All outputs capable of sinking / sourcing 30 mA
- Symmetrical drive on RA outputs (same Vdrop + / -)
- Two 16-bit timers count clock cycles, external events and generate interrupts and external signals.

Parallax Instruction Set Summary

This chart lists the general syntax for the entire Parallax SX instruction set.
{} = optional, [|] = choice of one of many.

ADD	dest, src
ADDB	dest, {/} src_bit
AND	dest, src
BANK	dest
CALL	addr8
CJA	op1, op2, addr
CJAE	op1, op2, addr
CJB	op1, op2, addr
CJBE	op1, op2, addr
CJE	op1, op2, addr
CJNE	op1, op2, addr
CLC	
CLR	dest
CLRB	dest_bit
CLZ	
CSA	op1, op2
CSAE	op1, op2
CSB	op1, op2
CSBE	op1, op2
CSE	op1, op2
CSNE	op1, op2
DEC	dest
DECSZ	dest
DJNZ	dest, addr
IJNZ	dest, addr
INC	dest
INCSZ	dest
IREAD	
JB	op_bit, addr
JC	addr
JMP	addr
JNB	op_bit, addr
JNC	addr
JNZ	addr
JZ	addr
LCALL	addr11
LJMP	addr11
LSET	addr11
MODE	lit
MOV	{!}dest, {!,/,--,++,<<,>>,<>} src {-dest}
MOVB	dest_bit, {/}src_bit
MOVSZ	dest, [-- | ++]src
NOP	
NOT	dest
OR	dest, src
PAGE	addr12
RET	
RETI	
RETIW	
RETP	
RETW	lit{,lit…}
RL	dest
RR	dest
SB	src_bit
SC	
SETB	dest_bit
SKIP	
SLEEP	
SNB	src_bit
SNC	
SNZ	
STC	
STZ	
SUB	dest, src
SUBB	dest, {/} src_bit
SWAP	dest
SZ	
TEST	dest
XOR	dest, src

SX Instruction Set

Single-Word Instructions

All instructions in this list consume only one word of SX EEPROM space.

ADD	fr, W	MOV	!OPTION, W
ADD	W, fr	MOV	!port, W
AND	fr, W	MOVSZ	W, ++fr
AND	W, fr	MOVSZ	W, --fr
AND	W, #literal	NOP	
BANK	fr	NOT	fr
CALL	addr	NOT	W
CLC		OR	fr, W
CLR	fr	OR	W, fr
CLR	W	OR	W, #literal
CLR	!WDT	PAGE	addr
CLRB	op.bit	RET	
CLZ		RETI	
DEC	fr	RETIW	
DECSZ	fr	RETP	
INC	fr	RL	fr
INCSZ	fr	RR	fr
IREAD		SB	op.bit
JMP	addr	SC	
JMP	W	SETB	op.bit
JMP	PC+W	SKIP	
MODE	literal	SLEEP	
MOV	fr, W	SNB	op.bit
MOV	W, fr	SNC	
MOV	W, /fr	SNZ	
MOV	W, fr-W	STC	
MOV	W, ++fr	STZ	
MOV	W, --fr	SUB	fr, W
MOV	W, <<fr	SWAP	fr
MOV	W, >>fr	SZ	
MOV	W, <>fr	TEST	fr
MOV	W, #literal	TEST	W
MOV	W, M	XOR	fr, W
MOV	M, #literal	XOR	W, fr
MOV	M, W	XOR	W, #literal

Multi-Word Instructions

All instructions in this list may consume more than one word of SX EEPROM space.

ADD	fr, #literal	DJNZ	fr, addr
ADD	fr1, fr2	IJNZ	fr, addr
ADDB	fr, op.bit	JB	op.bit, addr
ADDB	fr, /op.bit	JC	addr
AND	fr, #literal	JNB	op.bit, addr
AND	fr1, fr2	JNC	addr
CJA	fr, #literal, addr	JNZ	addr
CJA	fr1, fr2, addr	JZ	addr
CJAE	fr, #literal, addr	LCALL**	addr
CJAE	fr1, fr2, addr	LJMP**	addr
CJB	fr, #literal, addr	LSET*	addr
CJB	fr1, fr2, addr	MOV	fr, #literal
CJBE	fr, #literal, addr	MOV	fr1, fr2
CJBE	fr1, fr2, addr	MOV	fr, M
CJE	fr, #literal, addr	MOV	M, fr
CJE	fr1, fr2, addr	MOV	!OPTION, fr
CJNE	fr, #literal, addr	MOV	!OPTION, #literal
CJNE	fr1, fr2, addr	MOV	!port, fr
CSA	fr, #literal	MOV	!port, #literal
CSA	fr1, fr2	MOVB	op.bit1, op.bit2
CSAE	fr, #literal	MOVB	op.bit1, /op.bit2
CSAE	fr1, fr2	OR	fr, #literal
CSB	fr, #literal	OR	fr1, fr2
CSB	fr1, fr2	RETW***	
CSBE	fr, #literal	SUB	fr1, #literal
CSBE	fr1, fr2	SUB	fr1, fr2
CSE	fr, #literal	SUBB	fr, op.bit
CSE	fr1, fr2	SUBB	fr, /op.bit
CSNE	fr, #literal	XOR	fr, #literal
CSNE	fr1, fr2	XOR	fr1, fr2

* zero to three words. ** one to four words. *** one or more words.

Instruction Set Quick Reference

This chart is a quick reference to command syntax, size, cycle time and CARRYX sensitivity.
{} = optional, [|] = choice of one of many, ( ) = non-Turbo cycles, #,# = no branch / branch cycles.

Command		Words	Cycles	Command		Words	Cycles
ADD	fr, W	1	1(4)	MOV	fr, [#literal | fr2 | M]	2	2(8)
ADD	fr, [#literal | fr2]	2	2(8)	MOV	W, {/|++|--|<<|>>|<>}fr	1	1(4)
ADD	W, fr	1	1(4)	MOV*	W, fr-W	1	1(4)
ADDB	fr, {/} op.bit	2	2(8)	MOV	W, [#literal | M]	1	1(4)
AND	fr, W	1	1(4)	MOV	M, fr	2	2(8)
AND	fr, [#literal | fr2]	2	2(8)	MOV	M, [#literal | W]	1	1(4)
AND	W, [#literal | fr]	1	1(4)	MOV	!OPTION, [fr | #literal]	2	2(8)
BANK	fr	1	1(4)	MOV	!OPTION, W	1	1(4)
CALL	addr	1	3(8)	MOV	!port, [fr | #literal]	2	2(8)
CLC		1	1(4)	MOV	!port, W	1	1(4)
CLR	[fr | W | !WDT]	1	1(4)	MOVB	op.bit1, {/}op.bit2	4	4(16)
CLRB	op.bit	1	1(4)	MOVSZ	W, [++ | --]fr	1	1,2(4,8)
CLZ		1	1(4)	NOP		1	1(4)
CJA*	fr, [#literal | fr2], addr	4	4,6(16,20)	NOT	[fr | W]	1	1(4)
CJAE*	fr, [#literal | fr2], addr	4	4,6(16,20)	OR	fr, W	1	1(4)
CJB*	fr, [#literal | fr2], addr	4	4,6(16,20)	OR	fr, [#literal | fr2]	2	2(8)
CJBE*	fr, [#literal | fr2], addr	4	4,6(16,20)	OR	W, [fr | #literal]	1	1(4)
CJE*	fr, [#literal | fr2], addr	4	4,6(16,20)	PAGE	addr	1	1(4)
CJNE*	fr, [#literal | fr2], addr	4	4,6(16,20)	RET		1	3(8)
CSA*	fr, [#literal | fr2]	3	3,4(12,16)	RETI		1	3(8)
CSAE*	fr, [#literal | fr2]	3	3,4(12,16)	RETIW		1	3(8)
CSB*	fr, [#literal | fr2]	3	3,4(12,16)	RETP		1	3(8)
CSBE*	fr, [#literal | fr2]	3	3,4(12,16)	RETW		1+	3(8)
CSE*	fr, [#literal | fr2]	3	3,4(12,16)	RL	fr	1	1(4)
CSNE*	fr, [#literal | fr2]	3	3,4(12,16)	RR	fr	1	1(4)
DEC	fr	1	1(4)	SB	op.bit	1	1,2(4,8)
DECSZ	fr	1	1,2(4,8)	SC		1	1,2(4,8)
DJNZ	fr, addr	2	2,4(8,12)	SETB	op.bit	1	1(4)
INC	fr	1	1(4)	SKIP		1	2(8)
INCSZ	fr	1	1,2(4,8)	SLEEP		1	1(4)
IJNZ	fr, addr	2	2,4(8,12)	SNB	op.bit	1	1,2(4,8)
IREAD		1	4(16)	SNC		1	1,2(4,8)
JB	op.bit, addr	2	2,4(8,12)	SNZ		1	1,2(4,8)
JC	addr	2	2,4(8,12)	STC		1	1(4)
JMP	[addr | W]	1	3(8)	STZ		1	1(4)
JMP*	PC+W	1	3(8)	SUB*	fr1, [#literal | fr2]	2	2(8)
JNB	op.bit, addr	2	2,4(8,12)	SUB*	fr, W	1	1(4)
JNC	addr	2	2,4(8,12)	SUBB	fr, {/}op.bit	2	2(8)
JNZ	addr	2	2,4(8,12)	SWAP	fr	1	1(4)
JZ	addr	2	2,4(8,12)	SZ		1	1,2(4,8)
LCALL	addr	1-4	3-6(8-20)	TEST	[fr | W]	1	1(4)
LJMP	addr	1-4	3-6(8-20)	XOR	fr, [#literal | fr2]	2	2(8)
LSET	addr	0-3	0-3(0-12)	XOR	fr, W	1	1(4)
MODE	literal	1	1(4)	XOR	W, [fr | #literal]	1	1(4)
MOV	fr, W	1	1(4)				

* adversely affected by carry flag when CARRYX specified. See instruction syntax for more information.

Appendix B: SX Instruction Set

The columns of each instruction definition table in this appendix contain important information about the instructions behavior, size and structure.

The Command column lists all the available forms of the given command. The operands in lower-case letters indicate a symbol or value should be inserted in their place. The operands in upper-case letters should be entered exactly as seen. For example, the following form of the MOV command:

```
MOV !OPTION, #literal
```

should be entered into code (assuming $A5 is the desired literal) as follows:

```
MOV !OPTION, #$A5
```

The operands are described in Table B.1, below.

Table B.1: Symbols in the command column.

Symbol	Definition
addr	An address symbol or value
fr	A file register
#literal	A literal value ('#' must precede value)
M	The mode register
op.bit	The specified bit of the specified operand
!OPTION	The option register
PC	The program counter
!port	The specified I/O port data direction reg.
W	The working register
!WDT	The watchdog register

The Words column indicates the number of 12-bit EEPROM words consumed by the instruction.

The Cycles column indicates the number of clock cycles the given instruction will take. The number outside of parenthesis is the number of cycles in Turbo mode. The number inside parenthesis is the number of cycles in non-Turbo mode (compatibility mode).

The Affects column indicates the flags and registers the given instruction affects. These flags and registers are described in Table B.2, below.

Symbol	Definition
C	The carry flag
DC	The digit-carry flag
FSR	The file select register
fr	The file register
M	The mode register
op.bit	The specified bit of the specified operand
OPT	The options register
PC	The program counter
PD	The power-down flag
!port	The specified I/O port data direction reg.
TO	The time-out flag
W	The working register
Z	The zero flag

Table B.2: Symbols in the affects column.

The Coding column indicates how the given command will assemble, including binary values and mnemonic instructions. Some commands assemble into multiple, simpler commands. Table B.3 describes the binary symbols.

Symbol	Definition
b	Bit address
f	File register address
K	Constant

Table B.3: Symbols in the coding column.

Any instruction that performs a skip will only skip one instruction word. To avoid strange results, care must be taken to make sure that a single word instruction immediately follows the skipping instruction.

Many instructions are adversely affected by the carry flag when CARRYX (Add/Sub with C) is specified. Be careful to follow the special notes within the instruction descriptions concerning this.

In addition to the next single instruction, PAGE instructions are automatically skipped after any skip instruction.

ADD dest, src — Add src into dest

Command		Words	Cycles	Affects	Coding
1) ADD	fr, W	1	1 (4)	fr,C,DC,Z	`0001 111f ffff ADD fr,W`

2) ADD	fr, #literal	2	2 (8)	fr,W,C,DC,Z	`1100 kkkk kkkk MOV W,#lit`
					`0001 111f ffff ADD fr,W`

3) ADD	fr1, fr2	2	2 (8)	fr,W,C,DC,Z	`0010 000f ffff MOV W,fr2`
					`0001 111f ffff ADD fr1,W`

4) ADD	W, fr	1	1 (8)	W,C,DC,Z	`0001 110f ffff ADD W,fr`

Operation: Src is added into dest. C will be set if an overflow occurs; otherwise, C will be cleared. DC will be set if an overflow occurs in the lower nibble; otherwise, DC will be cleared. Z will be set if the result is 0; otherwise, Z will be cleared. W is left holding the source value in command #2 and #3. **If CARRYX is specified, c is added to result. Insert a CLC before the first Add on a register to avoid strange results.**

ADDB dest, src_bit — Add src_bit into dest

Command		Words	Cycles	Affects	Coding

1) ADDB	fr, op.bit	2	2 (8)	fr,Z	`0110 bbbf ffff SNB op.bit`
					`0010 101f ffff INC fr`

2) ADDB	fr, /op.bit	2	2 (8)	fr,Z	`0111 bbbf ffff SB  op.bit`
					`0010 101f ffff INC fr`

Operation: Src_bit is added into dest. If fr is incremented, Z will be set if the result is 0; otherwise, Z will be cleared. This instruction is useful for adding the carry into the upper byte of a double-byte sum after the lower byte has been computed.

AND dest, src — AND src into dest

Command		Words	Cycles	Affects	Coding
1) AND	fr, W	1	1 (4)	fr,Z	`0001 011f ffff AND fr,W`

2) AND	fr, #literal	2	2 (8)	fr,W,Z	`1100 kkkk kkkk MOV W,#lit`
					`0001 011f ffff AND fr,W`

3) AND	fr1, fr2	2	2 (8)	fr1,W,Z	`0010 000f ffff MOV W,fr2`
					`0001 011f ffff AND fr1,W`

4) AND	W, fr	1	1 (4)	W,Z	`0001 010f ffff AND W,fr`
5) AND	W, #literal	1	1 (4)	W,Z	`1110 kkkk kkkk AND W,#lit`

Operation: Src is AND'd into dest. Z will be set if the result is 0; otherwise, Z will be cleared. W is left holding the source value in command #2 and #3.

BANK dest — Set bank select bits

Command	Words	Cycles	Affects	Coding
1) BANK fr	1	1 (4)	FSR	`0000 0001 1fff BANK fr`

Operation: writes file registers bits 7 through 5 into file select register bits 7 through 5 in preparation for a RAM access across a bank boundary. The full 8-bit register address must be used as the destination. **On the SX48/52, bit 4 is cleared to 0 which will select only even numbered banks. To select an odd numbered bank, on the SX48/52, set bit 4 with a SETB FSR.4 after the BANK instruction.**

CALL addr8 — Call subroutine with 8-bit address

Command	Words	Cycles	Affects	Coding
1) CALL addr8	1	3 (8)	PC	`1001 kkkk kkkk CALL addr8`

Operation: The next instruction address is pushed onto the stack and addr8 is moved to the program counter. The ninth bit of the program counter will be cleared to 0. Therefore, calls are only allowed to the first half of any 512-word page, although the CALL instruction can be anywhere.

CJA op1, op2, addr — Compare op1 to op2 and jump if above

Command	Words	Cycles	Affects	Coding

1) CJA fr, #literal, addr	4	4 or 6 (jump) (16 or 20)	W,C,DC,Z	`1100 kkkk kkkk MOV W,#lit^$FF`
				`0001 110f ffff ADD W,fr`
				`0110 0000 0011 SNC`
				`101k kkkk kkkk JMP addr`

2) CJA fr1, fr2, addr	4	4 or 6 (jump) (16 or 20)	W,C,DC,Z	`0010 000f ffff MOV W,fr1`
				`0000 100f ffff MOV W,fr2-W`
				`0111 0000 0011 SC`
				`101k kkkk kkkk JMP addr`

Operation: op1 is compared to op2. If op1 is greater than op2, a jump to addr is executed. W is left holding the result of op1 + ~op2 in command #1 and op2 - op1 in command #2. **If CARRYX is specified, c affects result. Insert a CLC before command #1 and an STC before command #2 to avoid strange results.**

CJAE op1, op2, addr Compare op1 to op2 and jump if above or equal

Command	Words	Cycles	Affects	Coding

1) CJAE fr, #literal, addr	4	4 or 6 (jump) (16 or 20)	W,C,DC,Z	`1100 kkkk kkkk MOV W,#lit`
				`0000 100f ffff MOV W,fr-W`
				`0110 0000 0011 SNC`
				`101k kkkk kkkk JMP addr`

2) CJAE fr1, fr2, addr	4	4 or 6 (jump) (16 or 20)	W,C,DC,Z	`0010 000f ffff MOV W,fr2`
				`0000 100f ffff MOV W,fr1-W`
				`0110 0000 0011 SNC`
				`101k kkkk kkkk JMP addr`

Operation: op1 is compared to op2. If op1 is greater than or equal to op2, a jump to addr is executed. W is left holding the result of op2 - op1. **If CARRYX is specified, c affects result. Insert an STC before command to avoid strange results.**

CJB op1, op2, addr Compare op1 to op2 and jump if below

Command	Words	Cycles	Affects	Coding

1) CJB fr, #literal, addr	4	4 or 6 (jump) (16 or 20)	W,C,DC,Z	`1100 kkkk kkkk MOV W,#lit`
				`0000 100f ffff MOV W,fr-W`
				`0111 0000 0011 SC`
				`101k kkkk kkkk JMP addr`

2) CJB fr1, fr2, addr	4	4 or 6 (jump) (16 or 20)	W,C,DC,Z	`0010 000f ffff MOV W,fr2`
				`0000 100f ffff MOV W,fr1-W`
				`0111 0000 0011 SC`
				`101k kkkk kkkk JMP addr`

Operation: op1 is compared to op2. If op1 is less than op2, a jump to addr is executed. W is left holding the result of op2 - op1. **If CARRYX is specified, c affects result. Insert an STC before command to avoid strange results.**

CJBE op1, op2, addr — Compare op1 to op2 and jump if below or equal

Command	Words	Cycles	Affects	Coding

1) CJBE fr, #literal, addr	4	4 or 6 (jump)
		(16 or 20)
			W,C,DC,Z	`1100 kkkk kkkk MOV W,#lit^$FF`
				`0001 110f ffff ADD W,fr`
				`0111 0000 0011 SC`
				`101k kkkk kkkk JMP addr`

2) CJBE fr1, fr2, addr	4	4 or 6 (jump)
		(16 or 20)
			W,C,DC,Z	`0011 000f ffff MOV W,fr1`
				`0000 100f ffff MOV W,fr2-W`
				`0110 0000 0011 SNC`
				`101k kkkk kkkk JMP addr`

Operation: op1 is compared to op2. If op1 is less than or equal to op2, a jump to addr is executed. W is left holding the result of ~op2 + op1 in command #1 and op2 - op1 in command #2. **If CARRYX is specified, c affects result. Insert a CLC before command #1 and an STC before command #2 to avoid strange results.**

CJE op1, op2, addr — Compare op1 to op2 and jump if equal

Command	Words	Cycles	Affects	Coding

1) CJE fr, #literal, addr	4	4 or 6 (jump)
		(16 or 20)
			W,C,DC,Z	`1100 kkkk kkkk MOV W,#lit`
				`0000 100f ffff MOV W,fr-W`
				`0110 0100 0011 SNZ`
				`101k kkkk kkkk JMP addr`

2) CJE fr1, fr2, addr	4	4 or 6 (jump)
		(16 or 20)
			W,C,DC,Z	`0010 000f ffff MOV W,fr2`
				`0000 100f ffff MOV W,fr1-W`
				`0110 0100 0011 SNZ`
				`101k kkkk kkkk JMP addr`

Operation: op1 is compared to op2. If op1 is equal to op2, a jump to addr is executed. W is left holding the result of op1 - op2. **If CARRYX is specified, c affects result. Insert an STC before command to avoid strange results.**

CJNE op1, op2, addr — Compare op1 to op2 and jump if not equal

Command	Words	Cycles	Affects	Coding
1) CJNE fr, #literal, addr	4	4 or 6 (jump) (16 or 20)	W,C,DC,Z	`1100 kkkk kkkk MOV W,#lit` `0000 100f ffff MOV W,fr-W` `0111 0100 0011 SZ` `101k kkkk kkkk JMP addr`
2) CJNE fr1, fr2, addr	4	4 or 6 (jump) (16 or 20)	W,C,DC,Z	`0010 000f ffff MOV W,fr2` `0000 100f ffff MOV W,fr1-W` `0111 0100 0011 SZ` `101k kkkk kkkk JMP addr`

Operation: op1 is compared to op2. If op1 is not equal to op2, a jump to addr is executed. W is left holding the result of op1 - op2. **If CARRYX is specified, c affects result. Insert an STC before command to avoid strange results.**

CLC — Clear carry

Command	Words	Cycles	Affects	Coding
1) CLC	1	1 (4)	C	`0100 0000 0011 CLC`

Operation: C is cleared to 0.

CLR dest — Clear dest

Command	Words	Cycles	Affects	Coding
1) CLR fr	1	1 (4)	fr,Z	`0000 011f ffff CLR fr`
2) CLR W	1	1 (4)	W,Z	`0000 0100 0000 CLR w`
3) CLR !WDT	1	1 (4)	TO,PD	`0000 0000 0100 CLR wdt`

Operation: dest is cleared to 0. Z is set to 1 in command #1 and #2 while TO and PD are set to 1 in command #3. Prescaler is also cleared in command #3, if assigned.

CLRB dest_bit — Clear dest_bit

Command	Words	Cycles	Affects	Coding
1) CLRB op.bit	1	1 (4)	op.bit	`0100 bbbf ffff CLRB op.bit`

Operation: dest_bit is cleared to 0.

CLZ — Clear zero

Command	Words	Cycles	Affects	Coding
1) CLZ	1	1 (4)	Z	`0100 0100 0011 CLZ`

Operation: Z is cleared to 0.

CSA op1, op2 — Compare op1 to op2 and skip if above

Command	Words	Cycles	Affects	Coding

1) CSA fr, #literal	3	3 or 4 (skip) (12 or 16)	W,C,DC,Z	`1100 kkkk kkkk MOV W,#lit^$FF`
				`0001 110f ffff ADD W,fr`
				`0111 0000 0011 SC`

2) CSA fr1, fr2	3	3 or 4 (skip) (12 or 16)	W,C,DC,Z	`0010 000f ffff MOV W,fr1`
				`0000 100f ffff MOV W,fr2-W`
				`0110 0000 0011 SNC`

Operation: op1 is compared to op2. If op1 is greater than op2, the following instruction word is skipped. W is left holding the result of op1 + ~op2 in command #1 and op2 - op1 in command #2. **If CARRYX is specified, c affects result. Insert a CLC before command #1 and an STC before command #2 to avoid strange results.**

Note: Only one word is skipped by this instruction. **To avoid strange results, make sure that any instruction following CSA is a single-word instruction.**

CSAE op1, op2 — Compare op1 to op2 and skip if above or equal

Command	Words	Cycles	Affects	Coding

1) CSAE fr, #literal	3	3 or 4 (skip) (12 or 16)	W,C,DC,Z	`1100 kkkk kkkk MOV W,#lit`
				`0000 100f ffff MOV W,fr-W`
				`0111 0000 0011 SC`

2) CSAE fr1, fr2	3	3 or 4 (skip) (12 or 16)	W,C,DC,Z	`0010 000f ffff MOV W,fr2`
				`0000 100f ffff MOV W,fr1-W`
				`0111 0000 0011 SC`

Operation: op1 is compared to op2. If op1 is greater than or equal to op2, the following instruction word is skipped. W is left holding the result of op1 - op2. **If CARRYX is specified, c affects result. Insert an STC before command to avoid strange results.**

Note: Only one word is skipped by this instruction. **To avoid strange results, make sure that any instruction following CSAE is a single-word instruction.**

CSB op1, op2 — Compare op1 to op2 and skip if below

Command	Words	Cycles	Affects	Coding

1) CSB fr, #literal	3	3 or 4 (skip) (12 or 16)	W,C,DC,Z	`1100 kkkk kkkk MOV W,#lit`
				`0000 100f ffff MOV W,fr-W`
				`0110 0000 0011 SNC`

2) CSB fr1, fr2	3	3 or 4 (skip) (12 or 16)	W,C,DC,Z	`0010 000f ffff MOV W,fr2`
				`0000 100f ffff MOV W,fr1-W`
				`0110 0000 0011 SNC`

Operation: op1 is compared to op2. If op1 is less than op2, the following instruction word is skipped. W is left holding the result of op1 - op2. **If CARRYX is specified, c affects result. Insert an STC before command to avoid strange results.**

Note: Only one word is skipped by this instruction. **To avoid strange results, make sure that any instruction following CSB is a single-word instruction.**

CSBE op1, op2 — Compare op1 to op2 and skip if below or equal

Command	Words	Cycles	Affects	Coding

1) CSBE fr, #literal	3	3 or 4 (skip) (12 or 16)	W,C,DC,Z	`1100 kkkk kkkk MOV W,#lit^$FF`
				`0001 110f ffff ADD W,fr`
				`0110 0000 0011 SNC`

2) CSBE fr1, fr2	3	3 or 4 (skip) (12 or 16)	W,C,DC,Z	`0010 000f ffff MOV W,fr1`
				`0000 100f ffff MOV W,fr2-W`
				`0111 0000 0011 SC`

Operation: op1 is compared to op2. If op1 is less than or equal to op2, the following instruction word is skipped. W is left holding the result of op1 + ~op2 in command #1 and op2 - op1 in command #2. **If CARRYX is specified, c affects result. Insert a CLC before command #1 and an STC before command #2 to avoid strange results.**

Note: Only one word is skipped by this instruction. **To avoid strange results, make sure that any instruction following CSBE is a single-word instruction.**

CSE op1, op2 — Compare op1 to op2 and skip if equal

Command	Words	Cycles	Affects	Coding
1) CSE fr, #literal	3	3 or 4 (skip) (12 or 16)	W,C,DC,Z	`1100 kkkk kkkk MOV W,#lit` `0000 100f ffff MOV W,fr-W` `0111 0100 0011 SZ`
2) CSE fr1, fr2	3	3 or 4 (skip) (12 or 16)	W,C,DC,Z	`0010 000f ffff MOV W,fr2` `0000 100f ffff MOV W,fr1-W` `0111 0100 0011 SZ`

Operation: op1 is compared to op2. If op1 is equal to op2, the following instruction word is skipped. W is left holding the result of op1 - op2. **If CARRYX is specified, c affects result. Insert an STC before command to avoid strange results.**

Note: Only one word is skipped by this instruction. **To avoid strange results, make sure that any instruction following CSE is a single-word instruction.**

CSNE op1, op2 — Compare op1 to op2 and skip if not equal

Command	Words	Cycles	Affects	Coding
1) CSNE fr, #literal	3	3 or 4 (skip) (12 or 16)	W,C,DC,Z	`1100 kkkk kkkk MOV W,#lit` `0000 100f ffff MOV W,fr-W` `0110 0100 0011 SNZ`
2) CSNE fr1, fr2	3	3 or 4 (skip) (12 or 16)	W,C,DC,Z	`0010 000f ffff MOV W,fr2` `0000 100f ffff MOV W,fr1-W` `0110 0100 0011 SNZ`

Operation: op1 is compared to op2. If op1 is not equal to op2, the following instruction word is skipped. W is left holding the result of op1 - op2. **If CARRYX is specified, c affects result. Insert an STC before command to avoid strange results.**

Note: Only one word is skipped by this instruction. **To avoid strange results, make sure that any instruction following CSNE is a single-word instruction.**

DEC dest — Decrement dest

Command	Words	Cycles	Affects	Coding
1) DEC fr	1	1 (4)	fr,Z	`0000 111f ffff DEC fr`

Operation: dest is decremented. Z will be set to 1 if the result was 0; otherwise, Z will be cleared to 0. *The MOV W,--fr command is similar to DEC fr, except the result is moved to W.*

DECSZ dest — Decrement dest and skip if zero

Command	Words	Cycles	Affects	Coding
1) DECSZ fr	1	1 or 2 (skip) (4 or 8)	fr	`0010 111f ffff DECSZ fr`

Operation: dest is decremented. If result is 0, the next instruction word will be skipped. Z will be set to 1 if the result was 0; otherwise, Z will be cleared to 0.

Note: Only one word is skipped by this instruction. **To avoid strange results, make sure that any instruction following DECSZ is a single-word instruction.**

DJNZ dest, addr — Decrement dest and jump if not zero

Command	Words	Cycles	Affects	Coding

1) DJNZ fr, addr	2	2 or 4 (jump) (8 or 12)	fr	`0010 111f ffff DECSZ fr`
				`101k kkkk kkkk JMP addr`

Operation: dest is decremented. If the result is not 0, a jump to addr is executed.

IJNZ dest, addr — Increment dest and jump if not zero

Command	Words	Cycles	Affects	Coding

1) IJNZ fr, addr	2	2 or 4 (jump) (8 or 12)	fr	`0011 111f ffff INCSZ fr`
				`101k kkkk kkkk JMP addr`

Operation: dest is incremented. If the result is not 0, a jump to addr is executed.

INC dest — Increment dest

Command	Words	Cycles	Affects	Coding
1) INC fr	1	1 (4)	fr,Z	`0010 101f ffff INC fr`

Operation: dest is incremented. Z will be set to 1 if the result was 0; otherwise, Z will be cleared to 0. *The MOV W,++fr command is similar to INC fr, except the result is moved to W.*

INCSZ dest — Increment dest and skip if zero

Command	Words	Cycles	Affects	Coding
1) INCSZ fr	1	1 or 2 (skip) (4 or 8)	fr	`0011 111f ffff INCSZ fr`

Operation: dest is incremented. If result is 0, the next instruction word will be skipped. Z will be set to 1 if the result was 0; otherwise, Z will be cleared to 0.

Note: Only one word is skipped by this instruction. **To avoid strange results, make sure that any instruction following INCSZ is a single-word instruction.**

IREAD — Read instruction at MODE:W into MODE:W

Command	Words	Cycles	Affects	Coding
1) IREAD	1	4 (16)	W	`0000 0100 0001 IREAD`

Operation: the instruction or value at the 12-bit location MODE:W is read and stored into the MODE:W bits. This instruction can be used to read values in tables created with the DW directive. Use the MODE and MOVE instructions to read and write from the mode register.

JB src_bit, addr — Jump if src_bit is set

Command	Words	Cycles	Affects	Coding

1) JB op.bit, addr	2	2 or 4 (jump) (8 or 12)	none	`0110 bbbf ffff SNB op.bit`
				`101k kkkk kkkk JMP addr`

Operation: if src_bit is set, a jump to addr is executed.

JC addr — Jump if carry

Command	Words	Cycles	Affects	Coding

1) JC addr	2	2 or 4 (jump) (8 or 12)	none	`0110 0000 0011 SNZ`
				`101k kkkk kkkk JMP addr`

Operation: if carry flag is set, a jump to addr is executed.

JMP dest — Jump to dest

Command	Words	Cycles	Affects	Coding
1) JMP addr	1	3 (8)	PC	`101k kkkk kkkk JMP addr`
2) JMP W	1	3 (8)	PC	`0000 001f ffff JMP w`
3) JMP PC+W	1	3 (8)	PC,C,DC,Z	`0001 111f ffff ADD PC,W`

Operation: jump to address in dest. The lower 9 bits of the literal addr is moved into the program counter in command #1. W is moved into the program counter in command #2. W+1 is added to the program counter in command #3. Bit 9 of the program counter is cleared in command #2 and #3, so the jump destination will be in the first 256 words of any 512-word page. This instruction is useful for jumping into lookup tables comprised of RETW instructions, or jumping to particular routines. The flags are set as in an ADD instruction for command #3. **If CARRYX is specified, c affects result of command #3. Insert a CLC before command #3 to avoid strange results.**

JNB src_bit, addr — Jump if src_bit is not set

Command	Words	Cycles	Affects	Coding
1) JNB op.bit, addr	2	2 or 4 (jump) (8 or 12)	PC	`0111 bbbf ffff SB op.bit` `101k kkkk kkkk JMP addr`

Operation: if src_bit is not set, a jump to addr is executed.

JNC addr — Jump if carry is not set

Command	Words	Cycles	Affects	Coding
1) JNC addr	2	2 or 4 (jump) (8 or 12)	PC	`0111 0000 0011 SC` `101k kkkk kkkk JMP addr`

Operation: if carry flag is not set, a jump to addr is executed.

JNZ addr — Jump if zero in not set

Command	Words	Cycles	Affects	Coding
1) JNZ addr	2	2 or 4 (jump) (8 or 12)	PC	`0111 0100 0011 SZ` `101k kkkk kkkk JMP addr`

Operation: if zero flag is not set, a jump to addr is executed.

JZ addr — Jump if zero is set

Command	Words	Cycles	Affects	Coding

1) JZ addr	2	2 or 4 (jump)
		(8 or 12)
			PC	`0110 0100 0011 SNZ`
				`101k kkkk kkkk JMP addr`

Operation: if zero flag is set, a jump to addr is executed.

LCALL dest — Long call

Command	Words	Cycles	Affects	Coding

1) LCALL addr	1 - 4	3 - 6
		(8 - 20)
			PC	`{010x 1010 0011 CLRB/SETB 3.5}`
				`{010x 1100 0011 CLRB/SETB 3.6}`
				`{010x 1110 0011 CLRB/SETB 3.7}`
				`1001 kkkk kkkk LCALL addr`

Operation: call a subroutine outside the current 512-word page. Depending on the device size, from zero to three CLRB/SETB instructions will be assembled to point the page pre-select bits to dest's page. The bit set/clear instructions are followed by a call to dest. This instruction is only useful for SXs with more than 512 words.

Note: The LCALL command cannot use the auto page-set feature, @. The LCALL instruction does not reset the page select bits upon returning to the calling routine (if using RET). Therefore, your program must set these bits with the LSET $ instruction immediately after the LCALL, or the RETP instruction at the end of the called routine, both instructions set the page select bits to the current page. **This command is provided for backward compatibility only. It is suggested that the CALL @addr and RETP instructions be used instead for greater efficiency.**

LJMP dest — Long jump

Command	Words	Cycles	Affects	Coding

1) LJMP addr	1 - 4	3 - 6
		(8 - 20)
			PC	`{010x 1010 0011 CLRB/SETB 3.5}`
				`{010x 1100 0011 CLRB/SETB 3.6}`
				`{010x 1110 0011 CLRB/SETB 3.7}`
				`101k kkkk kkkk LJMP addr`

Operation: jump to a routine outside the current 512-word page. Depending on the device size, from zero to three CLRB/SETB instructions will be assembled to point the page pre-select bits to dest's page. The bit set/clear instructions are followed by a call to dest. This instruction is only useful for SXs with more than 512 words.

Note: The LJMP command cannot use the auto page-set feature, @. **This command is provided for backward compatibility only. It is suggested that the JMP @addr instruction be used instead for greater efficiency.**

LSET dest — Long set

Command	Words	Cycles	Affects	Coding

1) LSET addr	0 - 3	0 - 3
		(0 - 12)
			none	`{010x 1010 0011 CLRB/SETB 3.5}`
				`{010x 1100 0011 CLRB/SETB 3.6}`
				`{010x 1110 0011 CLRB/SETB 3.7}`

Operation: set page select bits to those in the current 512-word page referenced by dest. Depending on the device size, from zero to two CLRB/SETB instructions will be assembled. This instruction is only useful for SXs with more than 512 words.

MODE src — Set Mode to src

Command	Words	Cycles	Affects	Coding
1) MODE #literal	1	1 (4)	M	`0000 0101 kkkk MOV M,#lit`

Operation: src is moved into Mode register. This command is the same as MOV M,#literal. Use this command to initiate mode changes for port configuration commands. **On the SX18/28, this command only affects the lower 4 bits of the Mode register. On SX48/52, use MOV W,#lit and MOV M,W to affect all 5 bits.**

MOV dest, src — Move src into dest

Command	Words	Cycles	Affects	Coding
1) MOV fr, W	1	1 (4)	fr,none	`0000 001f ffff MOV fr,W`

2) MOV fr, #literal	2	2 (8)	fr,W	`1100 kkkk kkkk MOV W,#lit`
				`0000 001f ffff MOV fr,W`

3) MOV fr1, fr2	2	2 (8)	fr1,W,Z	`0010 000f ffff MOV W,fr2`
				`0000 001f ffff MOV fr1,W`

4) MOV fr, M	2	2 (8)	fr,W	`0000 0100 0010 MOV W,M`
				`0000 001f ffff MOV fr,W`

5) MOV W, fr	1	1 (4)	W,Z	`0010 000f ffff MOV W,fr`
6) MOV W, /fr	1	1 (4)	W,Z	`0010 010f ffff MOV W,/fr`
7) MOV W, fr-W	1	1 (4)	W,C,DC,Z	`0000 100f ffff MOV fr-W`
8) MOV W, ++fr	1	1 (4)	W,Z	`0010 100f ffff MOV W,++fr`
9) MOV W, --fr	1	1 (4)	W,Z	`0000 110f ffff MOV W,--fr`
10) MOV W, <<fr	1	1 (4)	W,C	`0011 010f ffff MOV W,<<fr`
11) MOV W, >>fr	1	1 (4)	W,C	`0011 000f ffff MOV W,>>fr`
12) MOV W, <>fr	1	1 (4)	W	`0011 100f ffff MOV W,<>fr`
13) MOV W, #literal	1	1 (4)	W	`1100 kkkk kkkk MOV W,#lit`
14) MOV W, M	1	1 (4)	W	`0000 0100 0010 MOV W,M`

15) MOV M, fr	2	2 (8)	W,M,Z	`0010 000f ffff MOV W,fr`
				`0000 0100 0011 MOV M,W`

16) MOV M, W	1	1 (4)	M	`0000 0100 0011 MOV M,W`
17) MOV M, #literal	1	1 (4)	M	`0000 0101 kkkk MOV M,#lit`

18) MOV !OPTION, fr	2	2 (8)	W,Z,OPT	`0010 000f ffff MOV W,fr`
				`0000 0000 0010 MOV !OPT,W`

19) MOV !OPTION, W	1	1 (4)	OPT	`0000 0000 0010 MOV !OPT,W`

20) MOV !OPTION, #literal	2	2 (8)	W,OPT	`1100 kkkk kkkk MOV W,#lit`
				`0000 0000 0010 MOV !OPT,W`

21) MOV !port, fr	2	2 (8)	W,Z,!port	`0010 000f ffff MOV W,fr`
				`0000 0000 0fff MOV !port,W`

22) MOV !port, W	1	1 (4)	!port	`0000 0000 0fff MOV !port,W`

23) MOV !port, #literal	2	2 (8)	W,!port	`1100 kkkk kkkk MOV W,#lit`
				`0000 0000 0fff MOV !port,W`

Operation: Src is moved into dest. Z will be set if the result is 0; otherwise, Z will be cleared. W is left holding the source value in command number 2, 3, 4, 18, 20, 21 and 23. C will be cleared if an underflow occurred; otherwise, C will be set to 1 in command number 7, 10 and 11. DC will be cleared if an underflow occurred in the lowest nibble; otherwise, DC will be set in command number 7. The value of C will be shifted into the LSB or MSB of W in command #10 and #11, respectively. C will be set to the previous MSB or LSB of fr in command #10 and #11 respectively. Command #12 moves the nibble-swapped value of fr into w. Instructions #6 through #12 are similar to NOT, SUB, INC, DEC, RL, RR and SWAP instructions, respectively, but have the additional feature of moving the result to W. **Only 4 bits are affected by #17; use #13 followed by #16 to affect 5 bits in the SX48/52. If CARRYX is specified, c affects result of command #7. Insert an STC before command #7 to avoid strange results.**

MOVB dest_bit, src_bit — Move src_bit to dest_bit

Command	Words	Cycles	Affects	Coding

1) MOVB op.bit1, op.bit2	4	4 (16)	op.bit1	`0111 bbbf ffff SB op.bit2`
				`0100 bbbf ffff CLRB op.bit1`
				`0110 bbbf ffff SNB op.bit2`
				`0101 bbbf ffff SETB op.bit1`

2) MOVB op.bit1, /op.bit2	4	4 (16)	op.bit1	`0110 bbbf ffff SNB op.bit2`
				`0100 bbbf ffff CLRB op.bit1`
				`0111 bbbf ffff SB op.bit2`
				`0101 bbbf ffff SETB op.bit1`

Operation: src_bit is moved into dest_bit in command #1. The one's complement of src_bit is moved into dest_bit in command #2.

MOVSZ dest, src — Move incremented or decremented src to dest and skip if zero

Command	Words	Cycles	Affects	Coding
1) MOVSZ W, ++fr	1	1 or 2 (skip) (4 or 8)	W	`0011 110f ffff MOVSZ W,++fr`
2) MOVSZ W, --fr	1	1 or 2 (skip) (4 or 8)	W	`0010 110f ffff MOVSZ W,--fr`

Operation: the incremented value (command #1) or decremented value (command #2) of src is moved into dest. The next instruction word will be skipped if the result was 0.

Note: Only one word is skipped by this instruction. **To avoid strange results, make sure that any instruction following MOVSZ is a single-word instruction.**

NOP — No operation

Command	Words	Cycles	Affects	Coding
1) NOP	1	1 (4)	none	`0000 0000 0000 NOP`

Operation: none. This instruction is useful to adjust the timing of a routine.

NOT dest — Not dest

Command	Words	Cycles	Affects	Coding
1) NOT fr	1	1 (4)	fr,Z	`0010 011f ffff NOT fr`
2) NOT W	1	1 (4)	W,Z	`1111 1111 1111 NOT w`

Operation: dest is converted into its one's complement value. Z will be set if the result was 0; otherwise, Z will be cleared. *The MOV W,/fr command is similar to #1, except the result is moved to W.*

OR dest, src — OR src into dest

Command	Words	Cycles	Affects	Coding
1) OR fr, W	1	1 (4)	fr,Z	`0001 001f ffff OR fr,W`

2) OR fr, #literal	2	2 (8)	fr,W,Z	`1100 kkkk kkkk MOV W,#lit`
				`0001 001f ffff OR fr,W`

3) OR fr1, fr2	2	2 (8)	fr,W,Z	`0010 000f ffff MOV W,fr2`
				`0001 001f ffff OR fr,W`

4) OR W, fr	1	1 (4)	W,Z	`0001 000f ffff OR W,fr`
5) OR W, #literal	1	1 (4)	W,Z	`1101 kkkk kkkk OR W,#lit`

Operation: src is OR'd into dest. Z will be set if the result was 0; otherwise, Z will be cleared.

PAGE addr12 — Set page select bits

Command	Words	Cycles	Affects	Coding
1) PAGE addr12	1	1 (4)	none	`0000 0001 0fff PAGE addr`

Operation: writes address bits 11 through 9 into page select bits PA2 through PA0 in preparation for a jump or call across a page boundary.

RET — Return from subroutine

Command	Words	Cycles	Affects	Coding
1) RET	1	3 (8)	W,PC	`0000 0000 1100 RET`

Operation: the top stack value is moved into the program counter and execution proceeds with the instruction following the most recent call instruction.

RETI — Return from interrupt routine

Command	Words	Cycles	Affects	Coding
1) RETI	1	3 (8)	C,DC,Z,PC	`0000 0000 1110 RETI`

Operation: W, STATUS, FSR and PC are popped off the shadow registers and execution proceeds with the instruction following the jump to interrupt.

RETIW — Return from interrupt routine, adjust RTCC

Command	Words	Cycles	Affects	Coding
1) RETIW	1	3 (8)	C,DC,Z,PC	`0000 0000 1111 RETIW`

Operation: W, STATUS, FSR and PC are popped off the shadow registers and execution proceeds with the instruction following the jump to interrupt. Value in W is added to RTCC. This is useful to create jitter-free, timed interrupts when using the interrupt-on-timer-rollover feature.

RETP — Return from subroutine across a page boundary

Command	Words	Cycles	Affects	Coding
1) RETP	1	3 (8)	PC	`0000 0000 1101 RETP`

Operation: the top stack value is moved into the program counter, return address bits 10 and 9 are written to page select bits PA1 and PA0, and execution proceeds with the instruction following the most recent call instruction. This instruction allows returning from a subroutine across page boundaries.

RETW — Assemble RETWs which load W with literal data upon return

Command	Words	Cycles	Affects	Coding
1) RETW literal {, literal…}	1 per literal	3 (8) per literal	PC	`1000 kkkk kkkk RETW #lit` `{1000 kkkk kkkk RETW #lit…}`

Operation: a list of RETWs are assembled each with one literal. This list can be accessed by JMP PC+W or JMP W instructions. This is useful for lookup tables.

RL dest — Rotate dest left

Command	Words	Cycles	Affects	Coding
1) RL fr	1	1 (4)	C	`0011 011f ffff RL fr`

Operation: dest is rotated left one bit. On entry, C must hold the value to be shifted into the least-significant bit of the dest value. On exit, C will hold the previous most-significant bit of the dest value. *The MOV W,<<fr command is similar to RL fr, except the result is moved to W.*

RR dest — Rotate dest right

Command	Words	Cycles	Affects	Coding
1) RR fr	1	1 (4)	C	`0011 001f ffff RR fr`

Operation: dest is rotated right one bit. On entry, C must hold the value to be shifted into the most-significant bit of the dest value. On exit, C will hold the previous least-significant bit of the dest value. *The MOV W,>>fr command is similar to RR fr, except the result is moved to W.*

SB src_bit — Skip if src_bit is set

Command	Words	Cycles	Affects	Coding
1) SB op.bit	1	1 or 2 (skip) (4 or 8)	none	`0111 bbbf ffff SB op.bit`

Operation: if src_bit is set, the following instruction word is skipped.

Note: Only one word is skipped by this instruction. **To avoid strange results, make sure that any instruction following SB is a single-word instruction.**

SC — Skip if carry is set

Command	Words	Cycles	Affects	Coding
1) SC	1	1 or 2 (skip) (4 or 8)	none	`0111 0000 0011 SC`

Operation: if C is set, the following instruction word is skipped.

Note: Only one word is skipped by this instruction. **To avoid strange results, make sure that any instruction following SC is a single-word instruction.**

SETB src_bit — Set src_bit

Command	Words	Cycles	Affects	Coding
1) SETB op.bit	1	1 (4)	op.bit	`0101 bbbf ffff SETB op.bit`

Operation: src_bit is set to 1.

SKIP — Skip the following instruction word

Command	Words	Cycles	Affects	Coding
1) SKIP	1	2 (8)	none	`0110 0000 0010 SKIP`

Operation: the following instruction word is skipped.

Note: Only one word is skipped by this instruction. **To avoid strange results, make sure that any instruction following SKIP is a single-word instruction.**

SLEEP — Enter sleep mode

Command	Words	Cycles	Affects	Coding
1) SLEEP	1	1 (4)	TO, PD	`0000 0000 0011 SLEEP`

Operation: The watchdog timer is cleared and the oscillator is stopped. TO is set and PD is cleared.

SNB src_bit — Skip if src_bit not set

Command	Words	Cycles	Affects	Coding
1) SNB op.bit	1	1 or 2 (skip) (4 or 8)	none	`0110 bbbf ffff SNB op.bit`

Operation: if src_bit is cleared, the following instruction word is skipped.

Note: Only one word is skipped by this instruction. **To avoid strange results, make sure that any instruction following SNB is a single-word instruction.**

SNC — Skip if carry not set

Command	Words	Cycles	Affects	Coding
1) SNC	1	1 or 2 (skip) (4 or 8)	none	`0110 0000 0011 SNC`

Operation: if C is cleared, the following instruction word is skipped.

Note: Only one word is skipped by this instruction. **To avoid strange results, make sure that any instruction following SNC is a single-word instruction.**

SNZ — Skip if zero is not set

Command	Words	Cycles	Affects	Coding
1) SNZ	1	1 or 2 (skip) (4 or 8)	none	`0110 0100 0011 SNZ`

Operation: if Z is cleared, the following instruction word is skipped.

Note: Only one word is skipped by this instruction. **To avoid strange results, make sure that any instruction following SNZ is a single-word instruction.**

STC — Set carry flag

Command	Words	Cycles	Affects	Coding
1) STC	1	1 (4)	C	`0101 0000 0011 STC`

Operation: carry is set.

STZ — Set zero flag

Command	Words	Cycles	Affects	Coding
1) STZ	1	1 (4)	Z	`0101 0100 0011 STZ`

Operation: Z is set.

SUB dest, src — Subtract src from dest

Command	Words	Cycles	Affects	Coding
1) SUB fr, W	1	1 (4)	fr,C,DC,Z	`0000 101f ffff SUB fr,W`

2) SUB fr1, #literal	2	2 (8)	fr,W,C,DC,Z	`1100 kkkk kkkk MOV W,#lit`
				`0000 101f ffff SUB fr,W`

3) SUB fr1, fr2	2	2 (8)	fr,W,C,DC,Z	`0010 000f ffff MOV W,fr`
				`0000 101f ffff SUB fr,W`

Operation: src is subtracted from dest. C will be cleared to 0 if an underflow occurred; otherwise, C will be set to 1. DC will be cleared to 0 if an underflow occurred in the least-significant nibble. Z will be set to 1 if the result was 0; otherwise, Z will be cleared to 0. *The MOV W,fr-W command is similar to #1, except the result is moved to W.* **If CARRYX is specified, c is added to result. Insert an STC before the first Sub on a register to avoid strange results.**

SUBB dest, src_bit — Subtract src_bit from dest

Command	Words	Cycles	Affects	Coding

1) SUBB fr, op.bit	2	2 (8)	Z	`0110 bbbf ffff SNB op.bit`
				`0000 111f ffff DEC fr`

1) SUBB fr, /op.bit	2	2 (8)	Z	`0111 bbbf ffff SB  op.bit`
				`0000 111f ffff DEC fr`

Operation: subtract src_bit from dest. If dest was decremented, Z will be set if the result was zero; else, Z will be cleared. This instruction is useful for subtracting the carry from the upper byte of a double-byte value after the lower byte has been subtracted.

SWAP dest — Swap nibbles in dest

Command	Words	Cycles	Affects	Coding
1) SWAP fr	1	1 (4)	none	`0011 101f ffff SWAP fr`

Operation: The high- and low-order nibbles of dest are swapped. *The MOV W,<>fr command is similar to SWAP fr, except the result is moved to W.*

SZ — Skip if zero flag set

Command	Words	Cycles	Affects	Coding
1) SZ	1	1 or 2 (skip) (4 or 8)	none	`0111 0100 0011 SZ`

Operation: if Z is set, the following instruction word is skipped.

Note: Only one word is skipped by this instruction. **To avoid strange results, make sure that any instruction following SZ is a single-word instruction.**

TEST src — Test src for zero

Command	Words	Cycles	Affects	Coding
1) TEST fr	1	1 (4)	Z	`0010 001f ffff TEST fr`
2) TEST w	1	1 (4)	Z	`1101 0000 0000 TEST w`

Operation: Z will be set if src is 0; otherwise, Z will be cleared.

XOR dest, src — XOR src into dest

Command	Words	Cycles	Affects	Coding
1) XOR fr, W	1	1 (4)	fr,Z	`0001 101f ffff XOR fr,W`
2) XOR fr, #literal	2	2 (8)	fr,W,Z	`1100 kkkk kkkk MOV W,#lit` `0001 101f ffff XOR fr,W`
3) XOR fr1, fr2	2	2 (8)	fr,W,Z	`0010 000f ffff MOV W,fr2` `0001 101f ffff XOR fr1,W`
4) XOR W, fr	1	1 (4)	W,Z	`0001 100f ffff XOR W,fr`
5) XOR W, #literal	1	1 (4)	W,Z	`1111 kkkk kkkk XOR W,#lit`

Operation: src is XOR’d into dest. Z will be set if the result was zero; otherwise, Z will be cleared.

Appendix C: SX-Key/Blitz Reserved Words

ADD	ENDR	OSC250KHZ	RL
ADDB	EQU	OSC31KHZ	RR
AND	ERROR	OSC4MHZ	RTCC
BANK	EXITM	OSC500KHZ	SB
BOR42	EXPAND	OSC62KHZ	SBIN
BOR26	FREQ	OSCHS	SDEC
BOR22	FSR	OSCXT	SC
BROWNOUT	FSTR	OSCXT1	SETB
BREAK	HS_OSC	OSCXT2	SHEX
C	ID	OSCXT3	SKIP
CALL	IF	OSCXT4	SLEEP
CARRYX	IFDEF	OSCXT5	SLEEPCLOCK
CASE	IFNDEF	OSCXTMAX	SNB
CJA	IJNZ	OSCXTMIN	SNC
CJAE	INC	OSCLP	SNZ
CJB	INCSZ	OSCRC	STACKX
CJBE	INDF	PA0	STACKX_OPTIONX
CJE	INDIRECT	PA1	STATUS
CJNE	IND	PA2	STC
CLC	IREAD	PAGE	STZ
CLR	JB	PC	SUB
CLRB	JC	PCL	SUBB
CLZ	JMP	PD	SWAP
CSA	JNB	PIC16C54	SX18L
CSAE	JNC	PIC16C55	SX28L
CSB	JNZ	PIC16C56	SYNC
CSBE	JZ	PIC16C57	SZ
CSE	LCALL	PIC16C58	TEST
CSNE	LJMP	PORT_B	TMR0
DC	LP_OSC	PORT_C	TO
DEC	LSET	PROTECT	TURBO
DECSZ	M	PROTECT_OFF	UBIN
DEVICE	MACRO	PROTECT_ON	UDEC
DJNZ	MODE	RA	UHEX
DRT18MS	MOV	RB	W
DRT1920MS	MOVB	RC	WATCH
DRT480MS	MOVSZ	RC_OSC	WATCHDOG
DRT60MS	NOCASE	RD	WDT
DRT60US	NOEXPAND	RE	WDT_OFF
DRT8MS	NOP	REPT	WDT_ON
DRT960MS	NOT	RESET	WREG
DRTOFF	OPTION	RET	XOR
DS	OPTIONX	RETI	XT_OSC
DW	OR	RETIW	Z
ELSE	ORG	RETP	ZSTR
END	OSC1MHZ	RETW	
ENDIF	OSC125KHZ	RF	
ENDM	OSC2MHZ	RG	

SX-Key/Blitz Reserved Words

Appendix D: SX-Key/Blitz Error Messages

SX-Key Error Messages

This appendix lists the SX-Key Assembler error messages. Most error messages below are followed by a short explanation and troubleshooting tips.

- **"=" must be preceded by a variable**
  - No valid symbol exists to the left of the "=". Check for mistyped symbol. Watch out for different case when the CASE directive is used.
  - Make sure symbol does not appear in Reserved Words list in Appendix C.

- **"\" only allowed in MACRO definition**
  - The macro argument symbol, "\", is a meaningless character outside of macros.

- **CALL must be to first half of page**
  - The destination address of the CALL instruction points to the second half of a page. See Calling Across Pages in Chapter 7 for more information.

- **Constant exceeds 32 bits**
  - The SX-Key assembler can not handle constants whose value is larger than 32 bits or 64 digits.

- **Constant exceeds 64 digits**
  - The SX-Key assembler can not handle constants whose value is larger than 32 bits or 64 digits.

- **Clock frequency must be from 400_000 to 110_000_000**
  - Designated frequency used in the FREQ directive is outside the range. The SX-Key can only clock the SX chip between 400 KHz and 110 KHz.

- **ELSE/ENDIF must be preceded by IF**
  - Check for missing or commented-out IF directive above the ELSE/ENDIF.

SX-Key/Blitz Error Messages

- **Empty string**
  - ➢ Look for undefined string within apostrophes.
- **ENDIF required to end IF block**
  - ➢ Check for missing or commented-out ENDIF directive below the IF.
- **ENDM required to end MACRO definition**
  - ➢ Check for missing or commented-out ENDM at the end of a MACRO definition.
- **ENDR must be preceded by REPT**
  - ➢ Check for missing or commented-out REPT directive above the ENDR.
- **ENDR required to end REPT block**
  - ➢ Check for missing or commented-out ENDR directive below the REPT.
- **EQU must be preceded by a label**
  - ➢ A valid symbol must precede the EQU directive. Check for mistyped symbol. Watch out for different case when the CASE directive is used.
  - ➢ Make sure symbol does not appear in Reserved Words list in Appendix C.
- **Error message contains control characters**
  - ➢ Error message must contain printable characters only.
- **Error message exceeds 64 characters**
  - ➢ Error message must be 64 characters or less in length.
- **EXITM/ENDM must be preceded by MACRO**
  - ➢ Check for missing or commented-out MACRO directive above the EXITM / ENDM.
- **Expected "++" or "--"**
  - ➢ The source operand in a MOVSZ must be preceded by ++ or --.

Appendix D: SX-Key/Blitz Error Messages

- **Expected “,”**
  - ➢ Check for missing arguments on a multi-argument mnemonic.

- **Expected “,” or end-of-line**
  - ➢ Check for invalid character(s) at the end of the line.

- **Expected a binary operator or “)”**

- **Expected a constant**

- **Expected a constant, variable, unary operator, or “(“**
  - ➢ Look for mnemonic with bad or missing arguments. Look for incomplete expressions.
  - ➢ Make sure symbol in use does not appear in Reserved Words list in Appendix C.

- **Expected a DEVICE parameter**
  - ➢ DEVICE directive contains a missing or invalid parameter. Look for misspellings, commas without trailing parameters, lower case when using CASE directive, etc.

- **Expected a label**

- **Expected a label, directive, or instruction**
  - ➢ Check for invalid arguments.
  - ➢ Check for mistyped mnemonic.

- **Expected a value from 0 to 64 or end-of-line**
  - ➢ Argument count on macros must be 0 to 64 or not specified.

- **Expected a value from 1 to 32**
  - ➢ The count parameter in the WATCH directive must be in the range of 1 to 32.

- **Expected a terminating quote**
  - ➢ Look for a string without a terminating quote, or apostrophe.

- **Expected an expression**
  - ➢ Look for a mnemonic with missing arguments.

- **Expected end-of-line**
  - Look for invalid character or argument at the end-of-line.
  - Look for incomplete expression.

- **Expected UDEC, SDEC, UHEX, SHEX, UBIN, SBIN, FSTR or ZSTR**
  - WATCH directive is missing formatter argument.

- **Expected W**
  - The W argument is missing in a mnemonic that requires it.

- **Expected WDT**
  - The clear-watchdog mnemonic must specify !WDT.

- **Expression is too complex**
  - SX-Key assembler can not handle the designated expression. Try simplifying the expression if possible.

- **ID cannot exceed 8 characters**
  - A maximum of 8 characters are allowed in the ID directive.

- **ID must be a string of up to 8 characters**
  - Make sure to use single quotes, or apostrophes, (‘), before and after the string. Make sure not to input control characters.

- **Location already contains data**
  - Assembled instruction overlaps used memory or crossed over last defined page barrier. Can also occur when the RESET directive is specified twice.

- **Label is already defined**
  - A symbol, or label, is already defined or is a reserved word. See the Reserved Words list in Appendix C. Make sure label’s position is not invalid, such as a label in a REPT block (this would make duplicate labels during the expansion). Make sure label starts with a letter or an underscore (_).

- **Limit of 32 nested REPTs exceeded**
  - ➢ The SX-Key assembler cannot process more than 32 nested REPT blocks.

- **Limit of 100,000 total REPT loops exceeded**
  - ➢ REPT count argument must be in the range 1..100,000.

- **List is too large**
  - ➢ List file generation can not complete because it is too large. Look for REPT blocks whose count is high, or whose final size, during assembly, is large.

- **Macro argument index is out of range**
  - ➢ The designated argument index is outside the specified range as set by the macro's definition.

- **Macro argument is not resolvable**

- **MACRO must be preceded by a label**
  - ➢ A valid symbol must precede the MACRO directive. Check for mistyped symbol.
  - ➢ Make sure symbol does not appear in Reserved Words list in Appendix C.

- **Macro stack overflow**
  - ➢ Macro is too complex; try simplifying it.

- **Nothing to assemble**
  - ➢ Must have source code entered or loaded up into the editor before assembling, programming, running or debugging.

- **Only one BREAK is allowed**
  - ➢ The SX chip does not support more than one breakpoint at a time.

- **Port is out of range**
  - ➢ Verify the port symbol or address in the mnemonic. See Appendix F for available ports.

SX-Key/Blitz Error Messages

- **Redundant DEVICE parameter**
  - ➢ The parameter specified conflicts with a previously specified device parameter.

- **REPT count must be greater than 0**

- **RESET address must be on first page**
  - ➢ The SX chip does not support a reset address outside of page 0.

- **Symbol exceeds 32 characters**
  - ➢ All symbols must be 32 characters in length or less and must start with a letter or underscore (_).

- **Symbol table full**
  - ➢ Too many symbols are defined. Must limit or combine any applicable symbols to assemble properly.

- **This directive cannot be preceded by a symbol**
  - ➢ Only the ORG, RESET, EQU, =, DS, DW, BREAK, MACRO and END directives can be preceded by a symbol.

- **Undefined Symbol**
  - ➢ Symbol is not defined above highlighted line. Check for mistyped symbol. Watch out for different case when the CASE directive is used.
  - ➢ Make sure symbol does not appear in Reserved Words list in Appendix C.

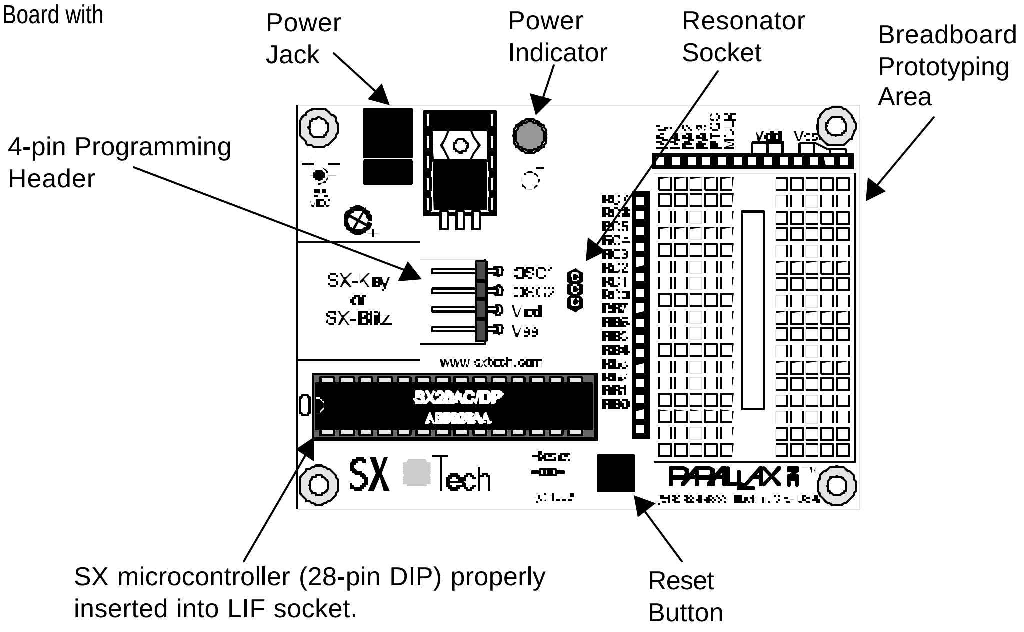
- **Unrecognized character**
  - ➢ Use single quotes (‘) instead of double quotes (“).

- **Variable must be followed by “=”**
  - ➢ Look for mistyped label.

Introduction

The Parallax SX Tech Board, the SX-Key Demonstration Board and the SX-Key QuickProto Board are learning tools for 28-pin and 18-pin DIP SX microcontrollers. The led28.src file on the SX-Key/Blitz Development System diskette demonstrates a simple program using the SX Tech Board and the SX28 microcontroller.

Figure E.1: SX Tech Board with SX chip inserted.



SX Tech Board Features

The SX Tech Board contains a socket and breadboard area to make development with the 28-pin SX DIP microcontroller easier. **The SX Tech Board does not support the 18-pin SX chip; please see the SX-Key QuickProto Board section, below, for more information.**

SX-Key Prototyping Boards

Features

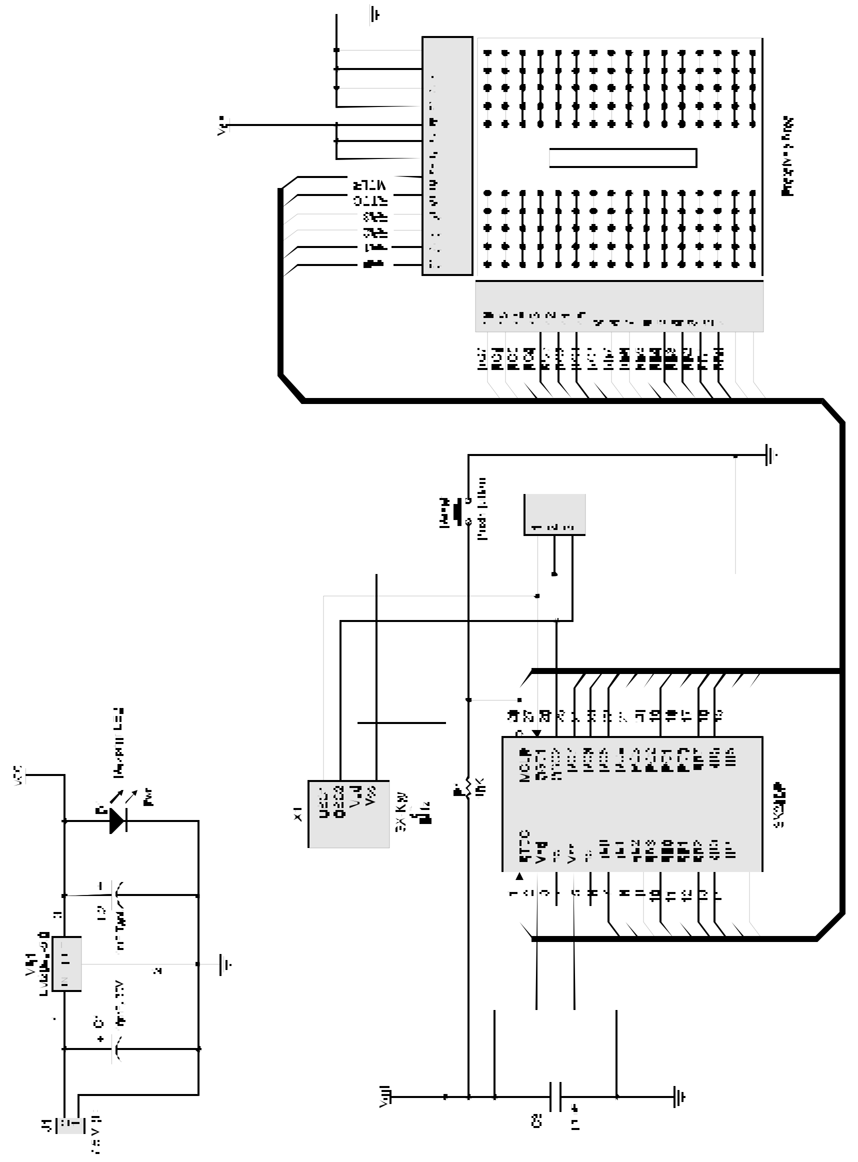
The SX Tech Board includes the following items:

- 7.5 VDC 1 A, center positive, power supply input;
- Power indicator LED;
- 28-pin LIF socket;
- 50 MHz ceramic resonator (in 3-pin socket);
- Breadboard area for prototyping;
- I/O pin headers adjacent to breadboard;
- Reset button.

Connecting and Downloading

See Chapter 2 for steps to connect and use the SX Tech board with the led28.src program.

SX-Key Demo Board Schematic



SX-Key Demo Board Features

The SX-Key Demo Board contains many pre-wired components to make development with the 28-pin SX DIP microcontroller easier. **The SX-Key Demo Board does not support the 18-pin SX chip; please see the SX-Key QuickProto Board section, below, for more information.**

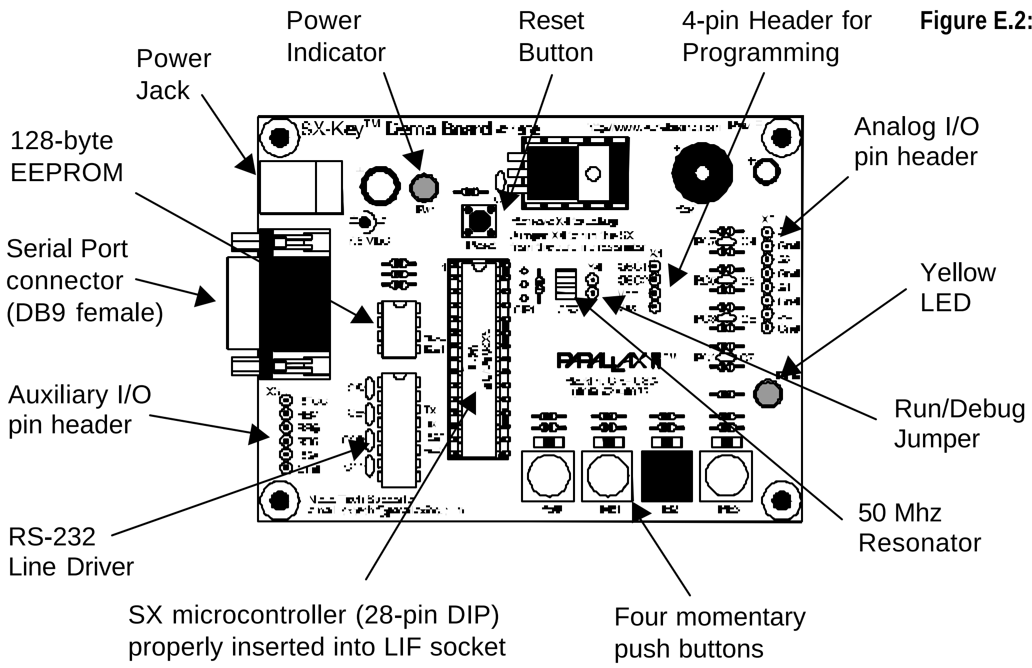


Figure E.2: SX-Key Demo Board with SX chip inserted.

Appendix E: SX-Key Prototyping Boards

Features

The SX-Key Demo Board includes the following items:

- 7.5 VDC 1 A, center positive, power supply input;
- Power indicator LED;
- 28-pin LIF socket;
- 50 MHz ceramic resonator;
- Jumper to enable / disable on-board resonator;
- (4) momentary pushbuttons;
- (1) yellow and (4) red LEDs to indicate status of buttons or I / O pins;
- 32 Ohm, 500 Hz – 8 KHz speaker;
- (2) digital to analog output pins;
- (2) analog to digital input pins;
- Auxiliary I / O pin header;
- RS-232 driver and DB9 (female) connector;
- 128 byte EEPROM;
- Reset button.

Virtual Peripheral Source Code Example

The sxdemo.src file on the SX-Key / Blitz Development System diskette works with the SX-Key Demo Board to demonstrate Virtual Peripherals. The Virtual Peripherals demonstrated by sxdemo.src are:

- One 19.2k baud UART for full-duplex communication between PC and the SX demo board over a standard RS-232 serial cable.
- Two 16-bit timers for controlling LED blink rate and speaker frequency output.
- Two 8-bit PWM / DAC outputs.
- Two ADC inputs.

Connecting and Downloading

See Chapter 2 for steps to connect and use the demo board with the sxdemo.src program.

Communication

The UART virtual peripheral allows communication between your PC and the SX chip on the demonstration board at 19.2k baud. See Chapter 2 for setup information.

Once the HyperTerminal connection is running, and the sxdemo.src program has already been downloaded to the SX in the demo board, press and release the Reset button on the demo board. You should hear a clicking noise, the yellow LED should flash and “SX Virtual Peripheral Demo” should appear on your screen.

Your cursor should appear next to a “>” prompt. Now you can type in simple commands to change the behavior of the virtual peripherals.

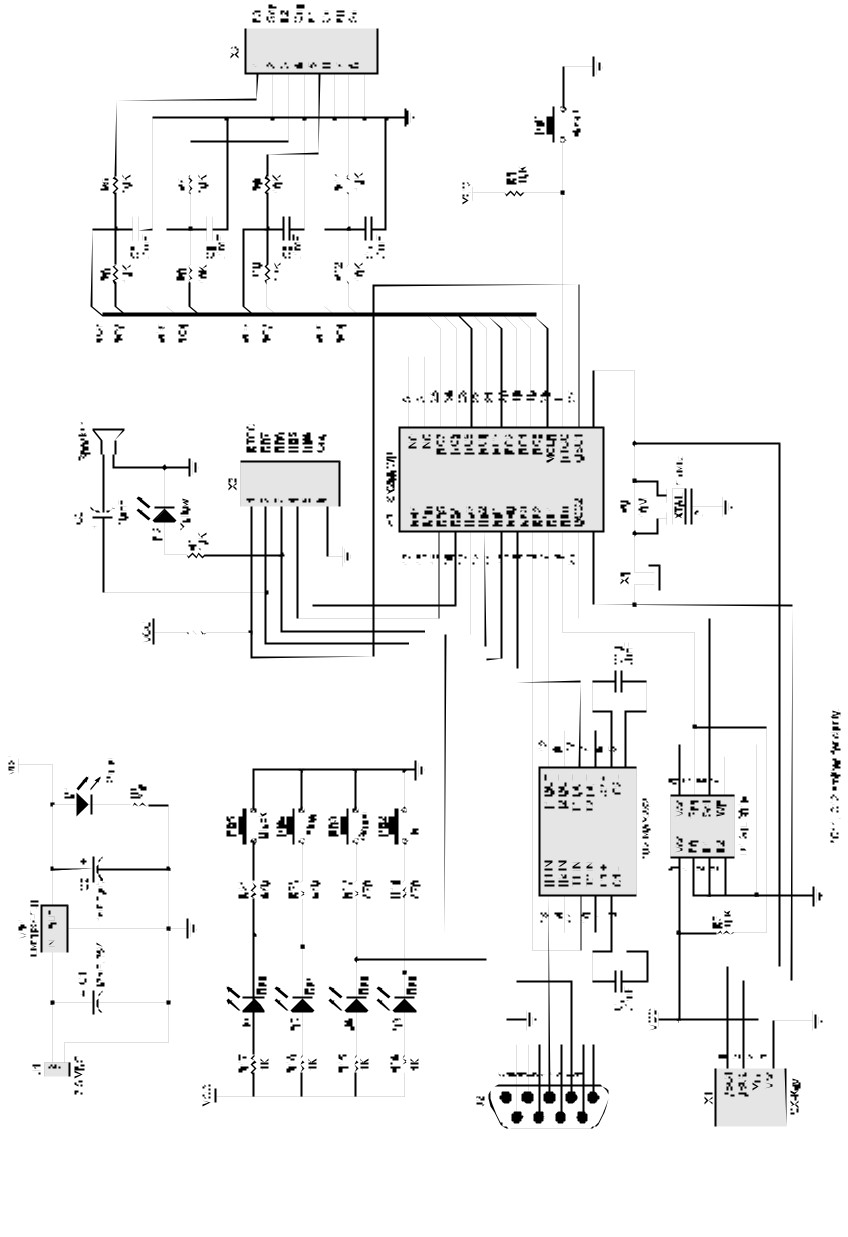
Configuring the Virtual Peripherals

Commands entered within HyperTerminal (or other properly configured terminal program) will modify the parameters used in each virtual peripheral. The SX is programmed to execute the virtual peripherals simultaneously. Table E.1 indicates the available commands and their function.

Table E.1: Virtual Peripheral Demo Commands.

Command	Description
Fxxx	Uses an independent 16-bit timer to control the output frequency / sound of the speaker. Parameter xxx is a number between 0 and FFF hex. Only low numbers are audible tones.
Txxx	Controls LED blinking rate. Parameter xxx is a number between 0 and FFF hex. Values between 0 and 6 display a noticeable LED flash and values up to FFF hex appear to make the LED continually stay “on” (though it is actually blinking).
Axx	Controls channel 1 PWM output on pin A0. Parameter xx is a number between 00 and FF hex, which will generate an analog voltage between 0 and 5 V. A simple RC circuit is attached to the I/O pin to filter the PWM.
Bxx	Controls channel 2 PWM output on pin A1. Parameter xx is a number between 00 and FF hex, which will generate an analog voltage between 0 and 5 V. A simple RC circuit is attached to the I/O pin to filter the PWM.
C	Channel 1 ADC input on pin A2. Using the capacitor on the A2 analog input pin, the SX reads voltage values on this pin and represents 0V to 5V as 00 to FF hex, respectively.
D	Channel 2 ADC input on pin A3. Using the capacitor on the A3 analog input pin, the SX reads voltage values on this pin and represents 0V to 5V as 00 to FF hex, respectively.

SX-Key Demo Board Schematic



SX-Key QuickProto Board Features

The SX-Key QuickProto Board contains pre-wired components and through-hole prototyping space to make development with the 18-pin SX DIP microcontroller easier. **The SX-Key QuickProto Board does not support the 28-pin SX chip; please see the SX-Key Demo Board section, above, for more information. The SX chip can be programmed in the QuickProto Board using either the SX-Key or the SX-Blitz.**

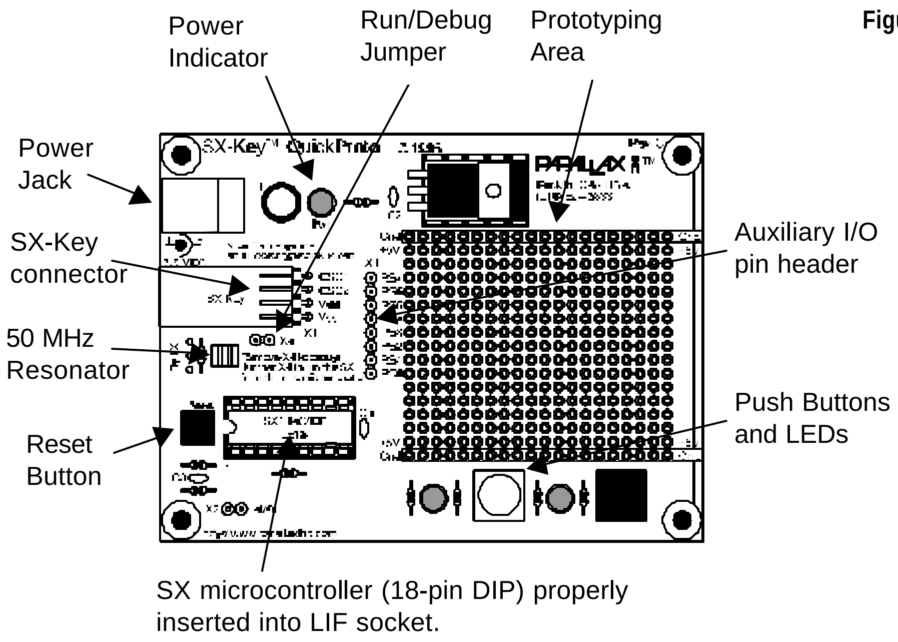


Figure E.3: SX-Key QuickProto Board.

Appendix E: SX-Key Prototyping Boards

Features

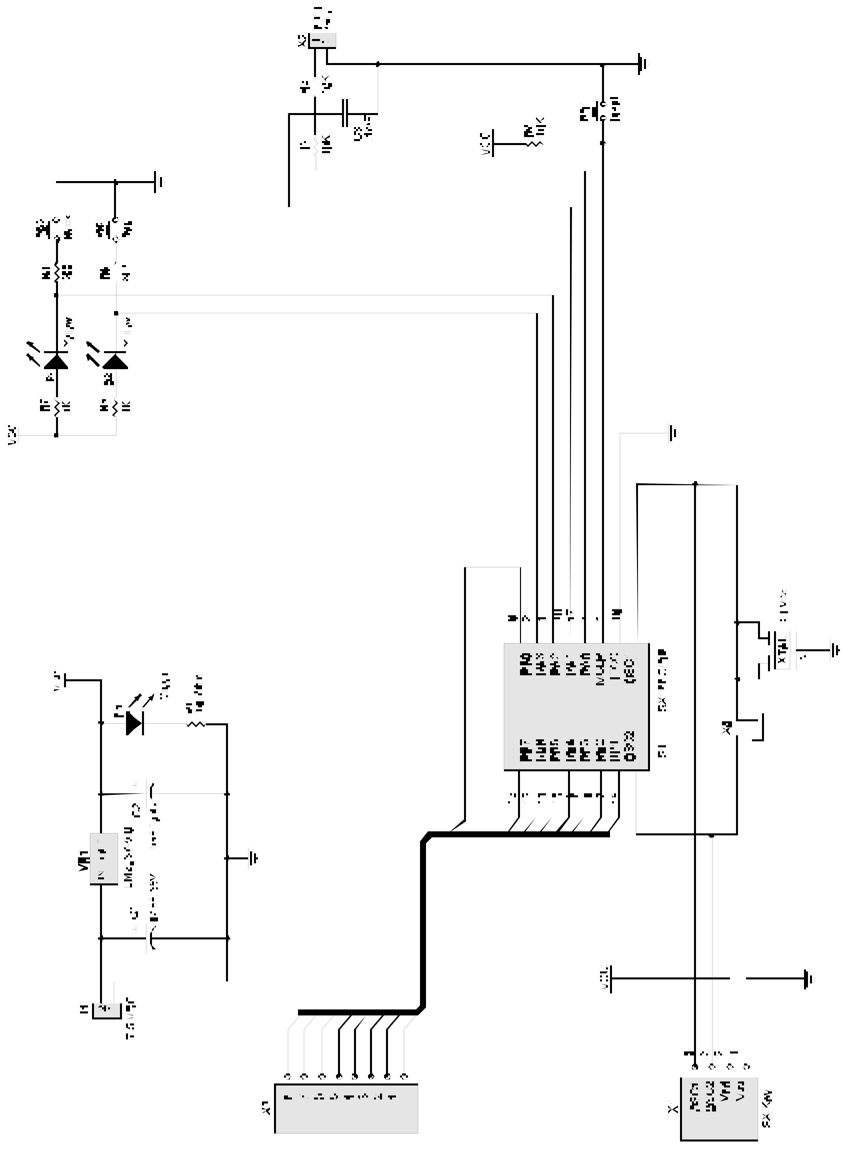
The SX-Key QuickProto Board includes the following items:

- 7.5 VDC 1 A, center positive, power supply input;
- Power indicator LED;
- 18-pin LIF socket;
- 50 MHz ceramic resonator;
- Jumper to enable/disable on-board resonator;
- (2) momentary pushbuttons;
- (2) LEDs to indicate status of buttons or I/O pins;
- (1) analog I/O pin;
- Auxiliary I/O pin header;
- 3.4 square inches of through-hole prototyping area;
- Reset button.

Connecting and Downloading

See Chapter 2 for steps to connect and use the QuickProto board with the led18.src program.

SX-Key QuickProto Board Schematic



Pinout Information and Descriptions (Not to scale or proportion)

Figure F.1: SX Pinouts.

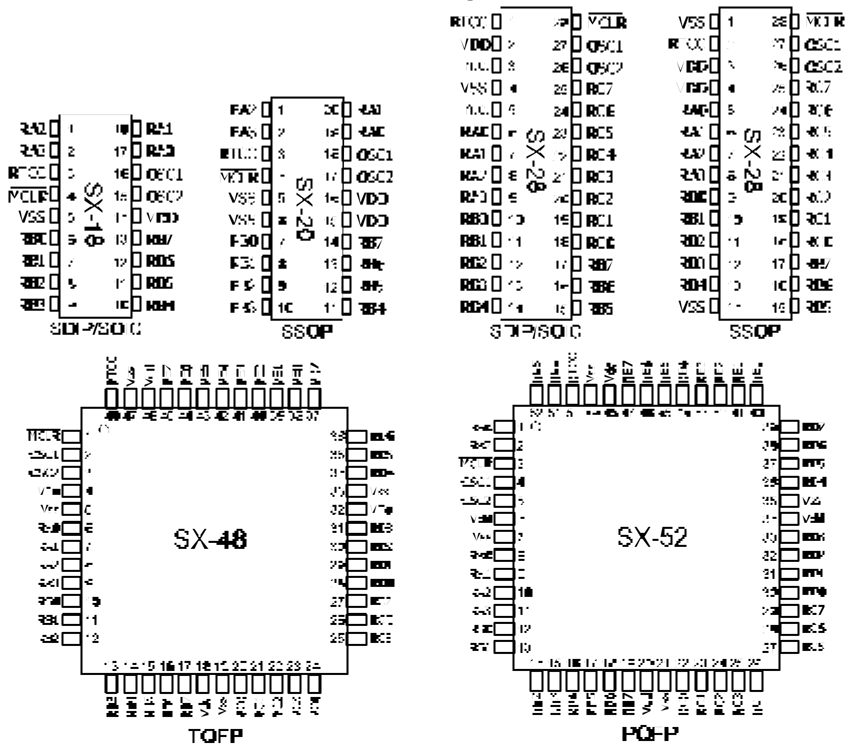


Table F.1: SX Pinout Descriptions.

Name	Type	Input Levels	Description
RA0 – RA7*	I/O	TTL/CMOS	Bi-directional I/O Pin, Complimentary Drive
RB0 - RB2	I/O	TTL/CMOS/ST	Bi-directional I/O Pin; MIWU mode; Comparator output, - input, + input
RB3 - RB7	I/O	TTL/CMOS/ST	Bi-directional I/O Pin; MIWU mode; (SX48/52 RB4 – RB7: T1 capture input 1, 2, PWM/compare out, ext. clk source)
RC0 - RC7	I/O	TTL/CMOS/ST	Bi-directional I/O Pin (SX48/52 RC0 – RC3: T2 capture input 1, 2, PWM/compare out, external clock source)
RD0 – RE7*	I/O	TTL/CMOS/ST	Bi-directional I/O Pin
RTCC	I	ST	Input to Real Time Clock/Counter
$\overline{\text{MCLR}}$	I	ST	Master Clear (reset) input (active low).
OSC1	I	ST	Oscillator crystal input - external clock input.
OSC2	I/O	CMOS	Weakly pulled to Vdd internally on RC mode.
Vdd	P	-	Positive supply for logic and I/O pins.
Vss	P	-	Ground Reference for logic and I/O pins.

* RA4 – RA7 is only available on the SX52.

Note: I = input, O = output, I/O = Input/Output, P = Power, - = Not Applicable, TTL = TTL input, CMOS = CMOS input, ST = Schmitt Trigger input, MIWU = Multi-Input Wake Up

Architecture

The Scenix SX chip offers 2K x 12 internal EE/Flash program memory (4K x 12 in the SX48/52) and up to 137 bytes of general purpose RAM memory (262 bytes in the SX48/52). The EE/Flash memory is organized in 512-word pages. The RAM memory is addressable directly or indirectly as well as semi-directly in the SX48/52). All special function registers are mapped into the data memory. Configuration registers do not appear in data memory and are only accessible through the use of the MODE register and the port configuration commands.

The ALU is 8-bits wide and is capable of arithmetic and Boolean operations. The 'W' register is the working register for the ALU. Typically, it holds one operand in a two-operand instruction. Depending on the instruction executed, the ALU may affect the values of the Carry (C), Zero (Z), and Digit Carry (DC) flags of the STATUS register.

The SX chip comes equipped with special features that reduce system cost and power requirements. The Power-On Reset (POR) and Device Reset Timer eliminate the need for external reset circuitry. The power saving SLEEP mode, watchdog timer, and code protect features reduce system cost and improve system integrity.

Instruction Pipeline

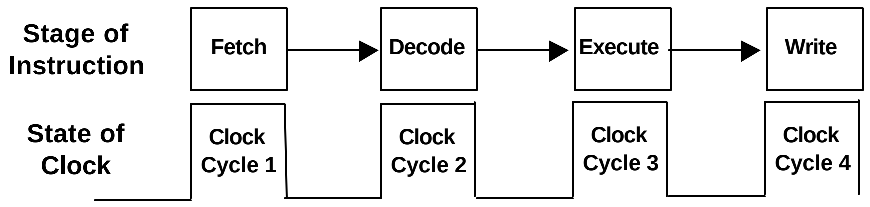


Figure F.2: Instruction Pipeline.

There are several stages an instruction must go through to actually execute within the SX chip. Specifically, there are four stages that are collectively referred to as the pipeline, and are shown in Figure F.2. The first instruction is fetched from memory on the first clock cycle. On the second clock cycle the first instruction is decoded and the second instruction is fetched. On the third clock cycle the first instruction is executed, the second instruction is decoded, and the third instruction is fetched. On the fourth clock cycle the first instruction's results are written to its destination, the second instruction is executed, the third instruction is decoded and the fourth instruction is fetched. Once the pipeline is full, instructions are executed at the rate of one per clock cycle (in Turbo mode). Instructions that directly alter the value in the program counter, i.e. jumps, calls, etc. require that the pipeline be cleared and subsequently refilled. When the pipeline is cleared, the fetch and decode stages are replaced with 'nop' instructions. This effectively nullifies the invalid instructions.

Read-Modify-Write Considerations

Use caution when performing successive SETB or CLRB operations on an I/O port pin. Since input data used for an instruction must be valid *during* the time the instruction is executed, and the result output from an instruction is valid *after* that instruction completes its operation, unexpected results from successive read-modify-write operations on I/O pins can occur when the SX is running at extremely high speeds. The SX has an internal write-back section to prevent such data errors from occurring but it is recommended that you buffer successive read-modify-write instructions performed on I/O pins of the same port at extremely high clock rates with a 'nop' instruction.

Also note, a read of an I/O pin actually reads the pin, not the output data latch. That is, if an output driver on a pin is enabled and driven high, but the external circuit is holding it low, a read of the port pin will indicate that the pin is low. Of course, externally driving an I/O pin while the output latch is driving it will result in damage to the SX chip. Care should be taken to not do this.

Register Map Structure

The SX18/28 RAM memory consists of a global bank of special function registers and eight banks of 16 general-purpose registers. The SX48/52 RAM memory consists of a global bank of special function registers and 16 banks of 16 general-purpose registers. Figures F.3 and F.4 demonstrate the structure of the registers for the SX18/28 and the SX48/52, respectively. In all SX instructions, bit 4 of the register address operand determines whether the global registers are accessed or whether a bank of general-purpose registers is accessed.

Figure F.3: SX18/28 RAM Register Map.

	Global		Bank 0	Bank 1	Bank 2		Bank 7
$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-	$08	$18-	$8	$8	$8		$8
$09-	$09	$19-	$9	$9	$9		$9
$0A-	$0A	$1A-	$A	$A	$A		$A
$0B-	$0B	$1B-	$B	$B	$B		$B
$0C-	$0C	$1C-	$C	$C	$C		$C
$0D-	$0D	$1D-	$D	$D	$D		$D
$0E-	$0E	$1E-	$E	$E	$E		$E
$0F-	$0F	$1F-	$F	$F	$F		$F

*Port C is available as general-purpose RAM in the SX18.

Figure F.4: SX48/52 RAM Register Map.

	Global		Bank 0*	Bank 1	Bank 2		Bank 15
$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-	$0A	$1A-	$A	$A	$A		$A
$0B-	$0B	$1B-	$B	$B	$B		$B
$0C-	$0C	$1C-	$C	$C	$C		$C
$0D-	$0D	$1D-	$D	$D	$D		$D
$0E-	$0E	$1E-	$E	$E	$E		$E
$0F-	$0F	$1F-	$F	$F	$F		$F

* Bank 0 is available only when using semi-direct addressing.

Special Function Registers

Special function registers are registers used by the CPU to control the operation of the device. The special function registers are contained within the first seven to ten locations of the global RAM bank as shown above and are described below.

Table F.2: Special Function Registers.

Addr	Name	Function
$00	IND	Used for indirect addressing
$01	RTCC / WREG	Real Time Clock Counter / WREG
$02	PC	Program Counter (low byte)
$03	STATUS	Holds Status bits of ALU
$04	FSR	File Select Register
$05	RA	Port A Control register
$06	RB	Port B Control register
$07	RC*	Port C Control register
$08	RD*	Port D Control register
$09	RE*	Port D Control register

*RC, RD and RE are available as general purpose RAM in the SX18. RD and RE are available as general purpose RAM in the SX28.

IND - The Indirect Register ($00)

This register, though not physically implemented, is used for indirect addressing. An instruction using IND as its operand actually performs the operation on the register pointed to by the contents of FSR. See Indirect Addressing, below, for more information.

RTCC Real Time Clock/Counter and WREG ($01)

RTCC is an 8-bit real-time timer/counter. In timer mode, the RTCC register will increment with every instruction cycle (without prescaler). In counter mode, the RTCC will increment with every cycle on the RTCC pin (with prescaler). The prescaler is used to lengthen the RTCC or watchdog timer effectively up to 16-bits. Depending on the RTW bit (OPTION.7), register $01 contains either the RTCC, (RTW is set) or the WREG, (RTW cleared). When WREG exists at $01, file register instructions (INC, DECSZ, etc) can be used directly on WREG. When doing this, use the register address $01, or the WREG symbol, rather than W. Using the W symbol instead of WREG to operate directly on the working register will result in errors or incorrectly assembled code.

PC - Program Counter ($02)

PC is a register that holds the lower 8-bits of the program counter. It is accessible at runtime to perform computed jumps and determine return addresses. Whenever an instruction is executed, and PC is the destination, the upper 3 or 4 bits of the STATUS register are loaded into the high byte of the program counter. This is necessary to achieve computed jumps and subroutine calls *across* page boundaries in code memory. Only 11 bits are used in SX18/28 parts. See The Jump Instruction for examples detailing how the program counter and the STATUS register are used in typical situations.

STATUS Register ($03)

This register holds the arithmetic status of the ALU, the page select bits, and the reset status. The status register is accessible during run-time but the resets bits, PD and TO, are read only. Care should be used when writing to the STATUS register as the ALU status bits will be updated upon completion of the write operation thereby leaving the STATUS register with a result that is different than intended. Therefore, it is recommended that only SETB, CLRB and PAGE instructions be used on this register.

Figure F.5: Status Register.

STATUS							
7	6	5	4	3	2	1	0
PA2	PA1	PA0	TO	PD	Z	DC	C

Bits 7-5: *Page select bits (PA2:PA0)

000 = Page 0 ($000 – $1FF)
001 = Page 1 ($200 – $3FF)
010 = Page 2 ($400 – $5FF)
011 = Page 3 ($600 – $7FF)
100 = Page 4 ($800 – $9FF)
101 = Page 5 ($A00 – $BFF)
110 = Page 6 ($C00 – $DFF)
111 = Page 7 ($E00 – $FFF)

* For devices of less than 4K of code space, unused bits of PA2:PA0 may be used as general purpose read / write bits.

Bit 4: Time Out bit (TO).

1 = Set to ‘1’ after power up, or executing a CLR !WDT or SLEEP instruction.
0 = A watchdog time-out occurred.

Bit 3: Power Down bit (PD).

1= Set to a ‘1’ after power up or executing a CLR !WDT instruction.
0 = Set to a ‘0’ by a SLEEP instruction

Bit 2: Zero Bit (Z).

1 = Result of math operation is zero
0 = Result of math operation is non-zero

Bit 1: Digit Carry (DC).

After Addition:
- 1 = A carry from bit-4 occurred
- 0 = No carry from bit-4 occurred

After Subtraction:
- 1 = No borrow from bit-4 occurred
- 0 = A borrow from bit-4 occurred

Bit 0: Carry (C).

After Addition:
- 1 = A carry occurred
- 0 = No carry occurred

After Subtraction:
- 1 = No borrow occurred
- 0 = A borrow occurred

The carry flag also serves as the ninth bit in RL and RR instructions. We can examine the operation of each of these instructions to further clarify the behavior of the carry flag. Consider the RR instruction first. When an RR instruction is performed on a RAM byte, the data in the RAM byte is rotated *through* the carry flag.

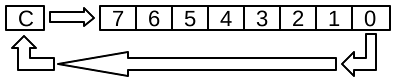


Figure F.6: Rotate-Right Operation.

Similarly, when an RL instruction is performed on a RAM byte, the data in the RAM byte is rotated through the carry flag.

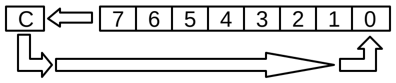


Figure F.7: Rotate-Left Operation.

FSR - File Select Register ($04)

The Scenix SX chip utilizes 12-bit op-codes. Instructions that specify a register as an operand can only express 5-bits of register address. This means that only registers from $00 up to $1F can be accessed. The File Select Register (FSR) along with the 5-bit register operand is used to provide the ability to access registers beyond $1F. Figure F.9 shows how the FSR's upper three bits select one of eight RAM banks on the SX18/28. Figure F.10 shows how the FSR's upper four bits select one of sixteen RAM banks.

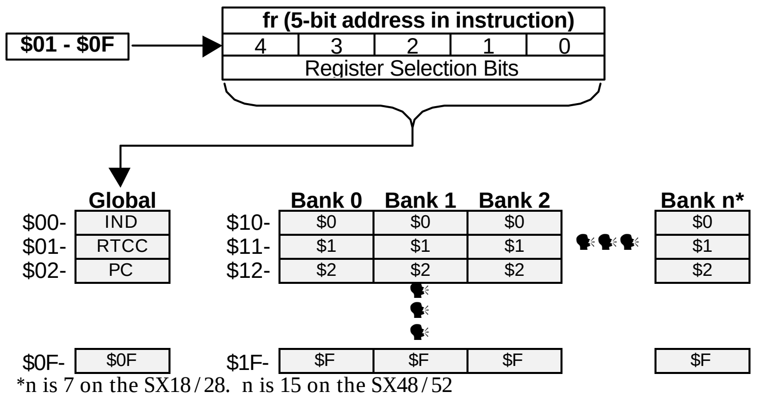
Special function and general-purpose register addresses $00 - $0F are 'global' in that they can always be accessed regardless of the contents of the FSR. Special function register $07 (RC) is available as general purpose RAM in 18-pin SX packages. Special function registers $08 (RD) and $09 (RE) are available as general-purpose RAM in 18 and 28-pin SX packages.

Direct Addressing

Global registers can be directly accessed at any time but general-purpose registers can only be directly accessed with in the current bank. The global registers are numbered $01 through $0F. Figure F.8 shows how the Global Registers are addressed in the SX18/28 and the SX48/52. Simply specify the desired global register address in the fr operand (the register address operand) of instructions as shown below:

```
mov     $0F, #$55          ;move $55 to global register $0F
```

Figure F.8: SX18/28/48/52 Global Register addressing (Direct).



General-purpose registers can only be accessed within one bank at a time. The general-purpose registers are numbered $10 through $1F. The active bank is controlled by the upper 3 bits of the FSR (SX18/28) as shown in figure F.9 or the upper 4 bits of the FSR (SX48/52) as shown in figure F.10.

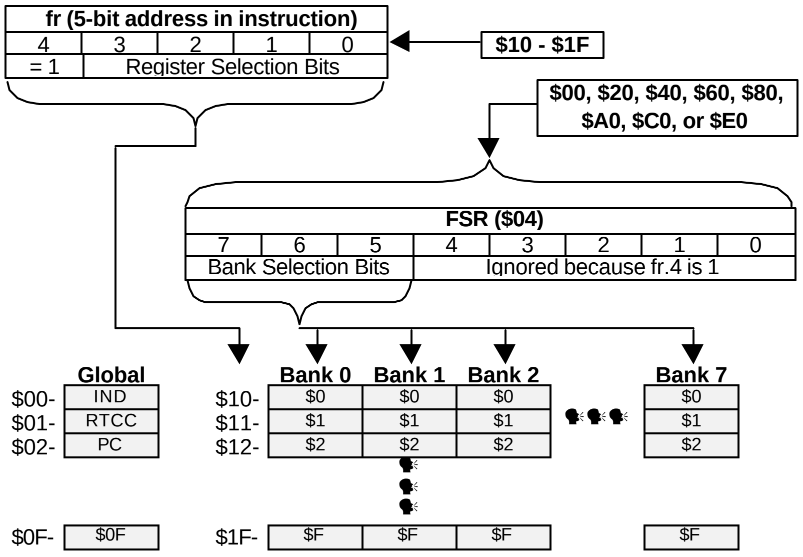


Figure F.9: SX18/28 General-Purpose Register addressing (Direct).

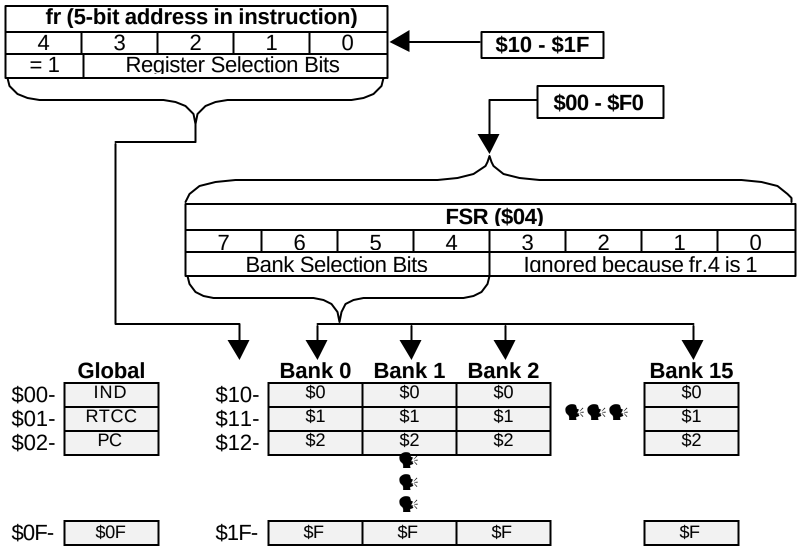


Figure F.10: SX48/52 General-Purpose Register addressing (Direct).

To ensure you are writing to the desired register you must first write the correct value to the FSR to select the proper bank. The following tables and code fragmenkts will show you how to directly access the banked registers on the SX18/28 and SX48/52.

Table F.3: SX18/28 Banks and FSR values.

SX18/28		SX48/52	
Desired bank	**FSR Value**	**Desired bank**	**FSR Value**
0	$00	0	$00
1	$20	1	$10
2	$40	2	$20
3	$60	3	$30
4	$80	4	$40
5	$A0	5	$50
6	$C0	6	$60
7	$E0	7	$70
--	--	8	$80
--	--	9	$90
--	--	10	$A0
--	--	11	$B0
--	--	12	$C0
--	--	13	$D0
--	--	14	$E0
--	--	15	$F0

This example clears register $10 in banks 3 and 6 on the SX18/28:

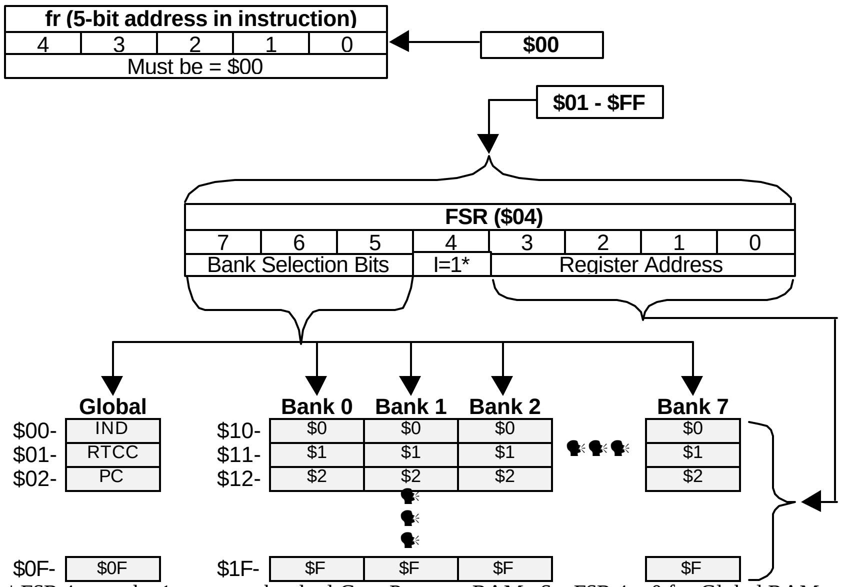
```
mov     FSR,#$60        ;Select Bank 3
clr     $10             ;Clear register $10 on Bank 3
mov     FSR,#$C0        ;Select Bank 6
clr     $10             ;Clear register $10 on Bank 6
```

This example clears register $10 in banks 3 and 6 on the SX48/52:

```
mov     FSR,#$30        ;Select Bank 3
clr     $10             ;Clear register $10 on Bank 3
mov     FSR,#$60        ;Select Bank 6
clr     $10             ;Clear register $10 on Bank 6
```

Indirect Addressing

To access any register via indirect addressing, simply move the 8-bit address of the register you wish to access into the FSR and use IND ($00) as the operand.



* FSR.4 must be 1 to access banked Gen. Purpose RAM. Set FSR.4 = 0 for Global RAM.

Figure F.11: SX18/28 Indirect register addressing.

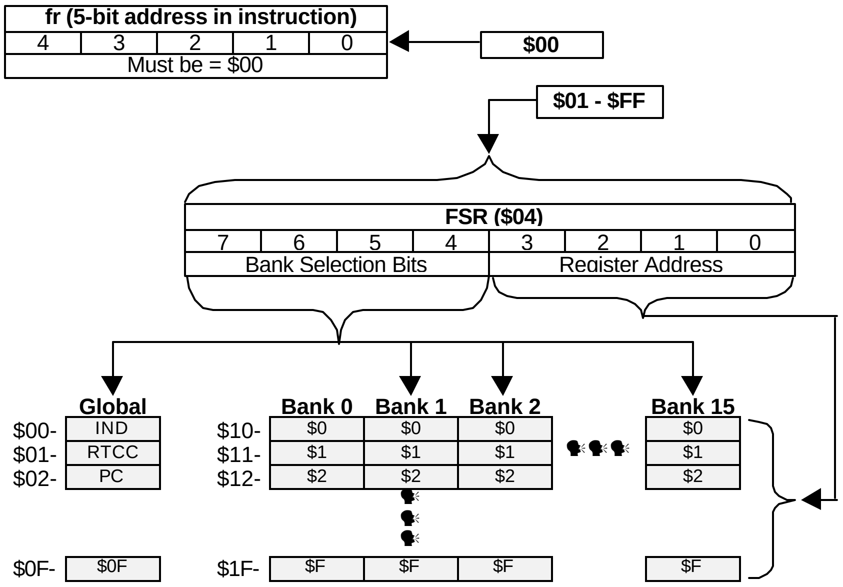


Figure F.12: SX48/52 Indirect register addressing.

This example for the SX18/28 will clear every General Purpose RAM register on every bank using indirect addressing.

```
Init    mov     FSR,#$10        ;FSR = addr of 1st RAM Reg.
Loop    clr     IND             ;Clear register
        inc     FSR             ;Point to next register
        setb    FSR.4           ;Keep us on G.P. RAM area
        cjne    FSR,#$10,Loop   ;Repeat until all registers
                                ;have been cleared
```

The Bank Instruction

Often it is desirable to set the bank select bits of the FSR with one instruction cycle (the MOV FSR, #literal commands above take two cycles). The SX instruction set offers such an instruction called Bank. The Bank instruction sets the upper bits of the FSR to point to the RAM bank required. Note: **On the SX48/52, the BANK instruction only selects even numbered banks. To select an odd numbered bank, use MOV FSR,#literal or add a SETB FSR.4 instruction after the BANK instruction.** Here's an example of how to use the Bank instruction on the SX18/28:

```
Zero    EQU     $00
One     EQU     $30
Two     EQU     $50
Three   EQU     $70
Four    EQU     $90
Five    EQU     $B0
Six     EQU     $D0
Seven   EQU     $F0

bank    Three           ;Make FSR point to bank 3
clr     $10             ;Clear register $10 (in bank 3)
bank    Six             ;Make FSR point to bank 6
mov     $10,#1          ;Set register $10 (in bank 6) to 1
```

The Jump Instruction

When a JMP instruction is executed, the lower nine bits of the program counter are loaded with the address of the label specified. The upper two or three bits of the program counter are loaded with the page select bits, PA2:PA0, from the STATUS register. Therefore, care must be used to ensure the page select bits are pointing to the correct page *before* the jump occurs.

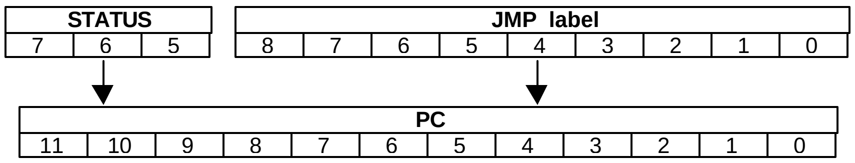


Figure F.13: Jump Instruction.

Jumping Across Pages

When a JMP instruction is executed and the intended destination is on a different page, you must set the page select bits to point to the desired page before the jump occurs. This can be done discretely with SETB and CLRB instructions or by writing a value to the STATUS register. The SX offers a new single-cycle instruction called PAGE that sets the page select bits for you. See "Dealing with Code Pages" in Chapter 7 for more information. *NOTE: Using the @ symbol in the JMP instruction (JMP @label) will cause the SX-Key assembler to insert the PAGE instruction before your JMP, during assembly.*

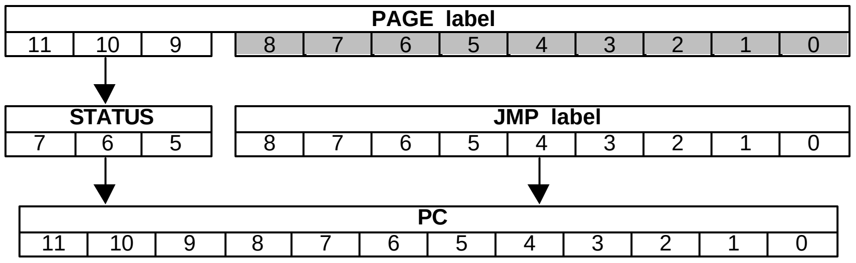


Figure F.14: Page-Jump Instructions.

Call Instruction

When a CALL instruction is executed, four things occur: 1) the current value of the program counter is incremented and pushed onto the top of the stack; 2) the lower eight bits of the address of the label are copied into the lower eight bits of the program counter; 3) the ninth bit of the PC is cleared to zero; and 4) the page select bits of the STATUS register are copied into the upper bits of the PC. Since bit 8 is cleared, the call destination must *start* in the lower half of any page of code space. i.e. $00-$FF, $200-$2FF, $400-$4FF, etc.

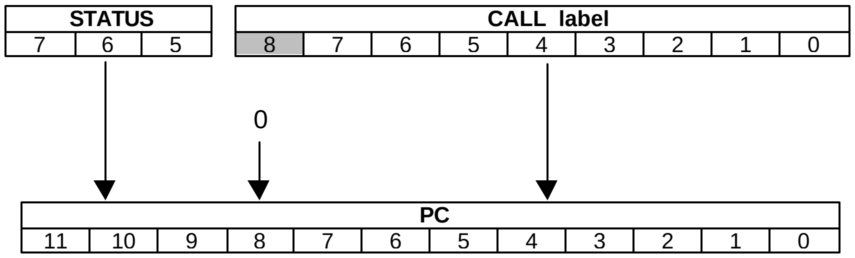


Figure F.15: Call Instruction.

Calling Across Pages

When it is necessary to call a subroutine that exists on a different page, you must set the page select bits to point to the desired page before the call is executed. This can be done discretely using SETB and CLRB instructions or by writing a value to the STATUS register. The SX offers a new single-cycle instruction called PAGE that sets the page select bits for you. See "Dealing with Code Pages" in Chapter 7 for more information. *NOTE: Using the @ symbol in the CALL instruction (CALL @label) will cause the SX-Key assembler to insert the PAGE instruction before your CALL, during assembly.*

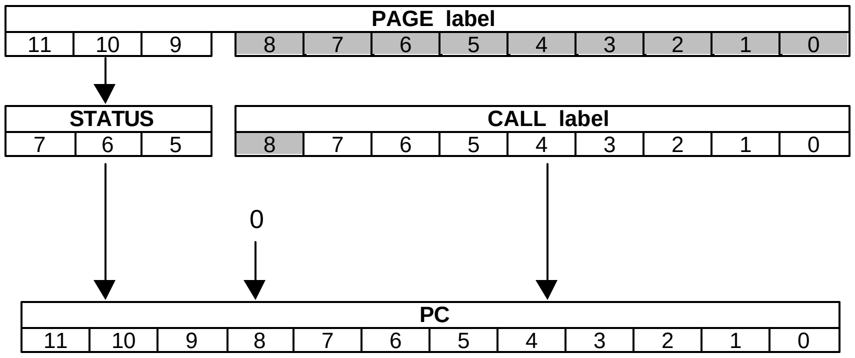


Figure F.16: Page-Call Instructions.

Returning from a subroutine

Subroutines are usually terminated with a return-type instruction. Before we discuss the different return instructions, we should describe the operation of the stack.

The Stack

The stack is an area of memory used to remember where to return to once a subroutine is complete. The stack is eight levels deep with the Stack Extend (StackX) option set and two levels deep by default. That means it can remember the return addresses for subroutines nested up to eight levels. The following explanation assumes that the SX has the Stack Extend option selected. The stack is capable of two operations; push and pop. The stack behaves like a plate holder in a salad buffet. A push is similar to placing a plate on the top of the stack and a pop is similar to removing a plate from the top of the stack.

The Push

When a subroutine is called, the return address is pushed onto the stack. Specifically, each address in the stack is moved to the next lower level in order to make room for the new address to be stored. Stack 1 gets the value that was in the program counter. Stack 8 is overwritten with what was in Stack 7. Consequently, the previous contents of Stack 8 are lost forever.

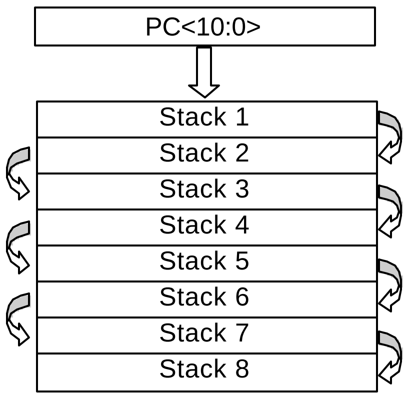
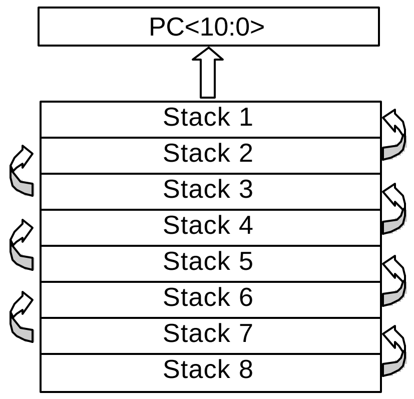


Figure F.17: Stack Push Operation.

The Pop

When a return instruction is executed, the stack is popped. Specifically, the content of Stack 1 is copied to the program counter and the content of each address in the stack is copied to the next higher level. Stack 1 gets the value that was in Stack 2, etc. until Stack 7 is overwritten with the contents of Stack 8. Consequently, the contents that were in Stack 8 are now duplicated in Stack 7.

Figure F.18: Stack Pop Operation.



Stack Overflow

As mentioned before, the stack can store up to eight return addresses (with Stack Extend on), or up to two return addresses by default. With each push, the stack stores another address. When the stack is full, the next push results in an overflow. The first time the stack is pushed into an overflow condition, the first address pushed is lost forever. If the stack were to be pushed again, the second address pushed would be lost also. A stack overflow condition inevitably leads to unintentional infinite loops or bizarre looping actions in your program. Care should be taken to ensure a stack overflow does not ever occur.

Stack Underflow

When the stack is popped more times than it has been pushed, a stack underflow occurs. Since a stack underflow causes unknown addresses to be stored in PC, a program may perform bizarre looping actions; such as a jump to unused program memory.

Returns

There are five different return instructions available on the SX. The RET (return) instruction simply pops the stack thereby setting the program counter to the instruction that followed the call. The RETW (return with literal in w) instruction behaves the same way but loads W with the literal value specified. RETP pops the stack and updates the page select bits to point to the page returned to. RETI pops the stack and the special shadow registers for W, STATUS, and the FSR, which were preserved during interrupt handling. RETIW behaves the same as RETI but also compensates the RTCC by adding the value in W to the RTCC.

Port Configuration Registers

Port A Registers

There are three registers used to configure the I/O pins of Port A. The TRIS_A register configures the data direction of the Port A pins as input or output. The LVL_A register configures the input pins as TTL or CMOS voltage level. The PLP_A register enables/disables pull up resistors on Port A input pins. To access these registers you must first write a particular value to the MODE register. Please refer to Table F.4 to find the values required in the MODE register to access the following Port A Registers. *Note: All the bits in the following registers are set to '1' on power up.*

TRIS_A – Data Direction Register

TRIS_A							
7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
-	-	-	-	RA3	RA2	RA1	RA0

A bit set to '1' in this register sets the corresponding I/O port pin to input (high z) mode.

A bit set to '0' in this register sets the corresponding I/O port pin to output mode.

LVL_A - TTL/CMOS Select Register

LVL_A							
7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
-	-	-	-	RA3	RA2	RA1	RA0

A bit set to ‘1’ in this register sets the input level of the corresponding port pin to TTL.

A bit set to ‘0’ in this register sets the input level of the corresponding port pin to CMOS.

PLP_A – Pull-Up Resistor Enable Register

PLP_A							
7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
-	-	-	-	RA3	RA2	RA1	RA0

A bit set to ‘1’ in this register disables the weak pull-up resistor on the corresponding port pin.

A bit set to ‘0’ in this register enables the weak pull-up resistor on the corresponding port pin.

Port B Registers

There are eight registers used to configure the I/O pins of Port B. The TRIS_B register configures the data direction of the Port B pins as input or output. The LVL_B register configures the input pins as TTL or CMOS voltage level. The PLP_B register enables/disables pull up resistors on Port B input pins. The ST_B register enables/disables the Schmitt-Trigger inputs on Port B input pins. The WKEN_B register enables/disables the multi-input wake up for interrupts on Port B input pins. The WKED_B register selects rising/falling edge detection on Port B input pins. The WKPND_B register contains the state of the MIWU pins. The CMP_B registers configures and provides the results from the comparator pins. To access these registers you must first write a particular value to the MODE register. Please refer to Table F.4 to find

the values required in the MODE register to access the Port B Registers.
Note: All the bits in the following registers are set to '1' on power up.

TRIS_B – Data Direction Register

TRIS_B							
7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
RB7	RB6	RB5	RB4	RB3	RB2	RB1	RB0

A bit set to '1' in this register sets the corresponding I/O port pin to input (high z) mode.

A bit set to '0' in this register sets the corresponding I/O port pin to output mode.

LVL_B - TTL/CMOS Select Register

LVL_B							
7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
RB7	RB6	RB5	RB4	RB3	RB2	RB1	RB0

A bit set to '1' in this register sets the input level of the corresponding port pin to TTL.

A bit set to '0' in this register sets the input level of the corresponding port pin to CMOS.

PLP_B – Pull-Up Resistor Enable Register

PLP_B							
7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
RB7	RB6	RB5	RB4	RB3	RB2	RB1	RB0

A bit set to '1' in this register disables the weak pull-up resistor on the corresponding port pin.

A bit set to '0' in this register enables the weak pull-up resistor on the corresponding port pin.

ST_B – Schmitt-Trigger Enable Register

ST_B							
7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
RB7	RB6	RB5	RB4	RB3	RB2	RB1	RB0

A bit set to ‘1’ in this register disables the Schmitt-Trigger input on the corresponding port pin.

A bit set to ‘0’ in this register enables the Schmitt-Trigger input on the corresponding port pin.

WKEN_B – Wake Up Enable Register

WKEN_B							
7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
RB7	RB6	RB5	RB4	RB3	RB2	RB1	RB0

A bit set to ‘1’ in this register disables the multi-input wake up for the corresponding port pin.

A bit set to ‘0’ in this register enables the multi-input wake up for the corresponding port pin.

WKED_B – Wake Up Edge Select Register

WKED_B							
7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
RB7	RB6	RB5	RB4	RB3	RB2	RB1	RB0

A bit set to ‘1’ in this register selects falling edge detection for the corresponding port pin.

A bit set to ‘0’ in this register selects rising edge detection for the corresponding port pin.

WKPND_B – MIWU Pending Register

WKPND_B							
7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
RB7	RB6	RB5	RB4	RB3	RB2	RB1	RB0

A bit set to ‘1’ in this register indicates a rising or falling edge was detected for the corresponding port pin.

A bit set to ‘0’ in this register indicates no edge was detected on the corresponding port pin since that bit was last cleared.

CMP_B – Comparator Enable Register

CMP_B							
7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
EN	OE	Rsvd	Rsvd	Rsvd	Rsvd	Rsvd	RES

EN	-	Comparator Enable; 0 = enabled, 1 = disabled
OE	-	Comparator Output Enable; 0 = enabled, 1 = disabled
Rsvd	-	Reserved for future use
RES	-	Comparator Result; (EN must = 0)

Port C Registers

There are four registers used to configure the I/O pins of Port C. The TRIS_C register configures the data direction of the Port C pins as input or output. The LVL_C register configures the input pins as TTL or CMOS voltage level. The PLP_C register enables/disables pull up resistors on Port C input pins. The ST_C register enables/disables the Schmitt-Trigger inputs on Port C input pins. To access these registers you must first write a particular value to the MODE register. Please refer to Table F.4 to find the values required in the MODE register to access the Port C Registers. *Note: All the bits in the following registers are set to ‘1’ on power up.*

TRIS_C – Data Direction Register

TRIS_C							
7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
RC7	RC6	RC5	RC4	RC3	RC2	RC1	RC0

A bit set to ‘1’ in this register sets the corresponding I/O port pin to input (high z) mode.

A bit set to ‘0’ in this register sets the corresponding I/O port pin to output mode.

LVL_C - TTL/CMOS Select Register

LVL_C							
7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
RC7	RC6	RC5	RC4	RC3	RC2	RC1	RC0

A bit set to ‘1’ in this register sets the input level of the corresponding port pin to TTL.

A bit set to ‘0’ in this register sets the input level of the corresponding port pin to CMOS.

PLP_C – Pull-Up Resistor Enable Register

PLP_C							
7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
RC7	RC6	RC5	RC4	RC3	RC2	RC1	RC0

A bit set to ‘1’ in this register disables the weak pull-up resistor on the corresponding port pin.

A bit set to ‘0’ in this register enables the weak pull-up resistor on the corresponding port pin.

ST_C – Schmitt-Trigger Enable Register

ST_C							
7	6	5	4	3	2	1	0
RC7	RC6	RC5	RC4	RC3	RC2	RC1	RC0

A bit set to '1' in this register disables the Schmitt-Trigger input on the corresponding port pin.

A bit set to '0' in this register enables the Schmitt-Trigger input on the corresponding port pin.

Control registers

There are three registers that configure the SX: MODE, OPTION, and Fuses. These registers allow the user to configure the SX in many ways. MODE and OPTION are run-time readable and writable while Fuses is written to only at program time.

Mode

The MODE register (simply called M in SX-Key mnemonics) is a run-time readable and writable register used to select the configuration registers for port operations.

MODE							
7	6	5	4	3	2	1	0
0	0	0	0	M3	M2	M1	M0

When a port configuration instruction is executed, such as MOV !RA, #1, the value of the MODE determines exactly which type of port configuration register will be written to. The right 4 bits in this register are set to '1' on power up. Below is an example detailing how the MODE register is used.

```
mov  M,#$0F      ;Set up MODE for Direction config.
mov  !RA,#$03    ;RA3:RA2 = Outputs, RA1:RA0 = Inputs
mov  M,#$0E      ;Set up MODE for Pull-Up config.
mov  !RA,#$01    ;RA3:RA1 = Normal,RA0 = Pull-up enabled
mov  M,#$0D      ;Set up MODE for TTL/CMOS config.
mov  !RA,#$02    ;RA3,RA2,RA0 = TTL, RA1 = CMOS
```

Table F.4: Mode value definitions.

MODE	Port A	Port B	Port C
$0F	TRIS_A	TRIS_B	TRIS_C
$0E	PLP_A	PLP_B	PLP_C
$0D	LVL_A	LVL_B	LVL_C
$0C		ST_B	ST_C
$0B		WKEN_B	
$0A		WKED_B	
$09		Swap W with WKPEN_B	
$08		Swap W with COMP_B	
$07 - $00			

The table above defines the allowed mode values and their functions. The gray areas are undefined at this time.

Option

The OPTION register is a run-time writable register used to configure the RTCC and the Watchdog Timer. The size of this register is affected by the OPTION Extend bit as noted in Table F.6.

OPTION							
7	6	5	4	3	2	1	0
RTW	RTI	RTS	RTE	PSA	PS2	PS1	PS0

When OPTION Extend = 0, bits 7 and 6 are implemented.
When OPTION Extend = 1, bits 7 and 6 read as '1's.

RTW	-	If = 0, register $01 is W
		If = 1, register $01 is RTCC

RTI	-	If = 0, RTCC roll-over interrupt is enabled If = 1, RTCC roll-over interrupt is disabled
RTS	-	If = 0, RTCC increments on internal instruction cycle If = 1, RTCC increments on transition of RTCC pin

RTE	-	If = 0, RTCC increments on low-to-high transition
		If = 1, RTCC increments on high-to-low transition

PSA	-	If = 0, prescaler is assigned to RTCC, divide rate determined by PS0-PS2 bits If = 1, prescaler is assigned to WDT, and divide rate on RTCC is 1:1

Table F.5: Prescaler division ratios.

PS2, PS1, PS0	RTCC Divide Rate	Watchdog Timer Divide Rate
000	1:2	1:1
001	1:4	1:2
010	1:8	1:4
011	1:16	1:8
100	1:32	1:16
101	1:64	1:32
110	1:128	1:64
111	1:256	1:128

Fuses

The Fuses register is accessible only at program time. The SX-Key Development System Software provides a convenient interface that is easy to use to customize the SX. You may use the predefined device directives in your source code to specify the bits in the Fuses register. See “Device Directive” in Chapter 6 for additional information.

Interrupts (SX18/28)

Description

Sometimes a particular task or event must have the immediate attention of the processor. An interrupt is a means to accomplish this. In theory, the processor stops whatever it was doing and immediately begins executing code located at a special location called the Interrupt Vector. Once the code located at the Interrupt Vector task has completed, the processor returns to where it was before the interruption occurred.

In reality, several things must occur in addition to the aforementioned to ensure proper operation of the interrupt and the rest of the program. For one thing, consider the likely possibility that the interrupt occurred when the W register held a number the main program was using for a calculation. More than likely, the interrupt service routine will use the W register for its purposes too. When the processor finishes the interrupt service routine and returns to what it was doing before, the W register will hold a different value than what it held before the interrupt occurred. This can lead to bizarre program execution. In addition to the W register, the Status and FSR registers must be ‘preserved’ across an interrupt.

Traditionally, these issues were dealt with by software within the interrupt service routine. The engineers at Scenix had the foresight to take the burden off the programmer and put it in the chip where it belongs.

The Specifics

When an interrupt occurs in an SX chip, the STATUS, FSR, and W registers are saved in special shadow locations, and additional interrupts ignored. The program counter is loaded with $00, (The Interrupt Vector), and the top three bits of the STATUS register, (PA2:PA0) are cleared to $0. When the interrupt service routine has completed and the ‘RETI’ instruction is executed, the Status, FSR, and W registers are restored and the interrupt is re-enabled. Since this occurs automatically, your interrupt service routine does not have to waste any valuable time copying several registers back and forth.

RTCC Interrupt

The SX chip offers one internal interrupt called the RTCC rollover. If enabled, when the RTCC increments from $FF to $00, an interrupt will be generated. The latency, or response delay, will be exactly three instruction cycles in Turbo Mode, and exactly eight cycles in Compatible Mode. When the interrupt is complete, there will be a three-cycle delay (Turbo mode) or an eight-cycle delay (Compatible mode) before main code begins executing. This is due to the pipeline. Whenever an instruction is executed and, because of the instruction, the program counter is changed, the pipeline must be flushed and refilled. See RTCC Rollover Interrupts in Chapter 7 for more information.

RB0-RB7 Interrupt

The SX offers eight sources of external interrupts; a change of state on any of RB0 – RB7 can generate an interrupt. These can be individually configured via the WKEN_B and WKED_B configuration registers. The latency, or response delay, will be five cycles in Turbo mode and ten cycles in Compatible mode. See Wake-Up (Interrupt) on Edge Detect in Chapter 7 for more information.

PERIPHERALS

Oscillator Driver

The SX chip offers a configurable oscillator driver that supports five types of oscillators:

LP:	Low Power Oscillator
XT:	Crystal or Resonator
HS:	High Speed Crystal or Resonator
RC:	Resistor/Capacitor
IRC:	Internal Resistor/Capacitor

XT, LP, and HS Mode

In XT, LP, and HS modes, a crystal or ceramic resonator can be connected to the SX chip as in Figures F.14 and F.15. The SX oscillator driver design requires the use of a parallel cut crystal. Use of a series cut crystal may give a frequency out of the crystal manufacturer's specifications. The values of the components can be determined from the Component Selection Tables below. Please note that some ceramic resonators have internal capacitors so that no external capacitors are required.

An external clock source can also be used to drive the OSC1 pin (leaving the OSC2 pin disconnected) as in Figure F.16.

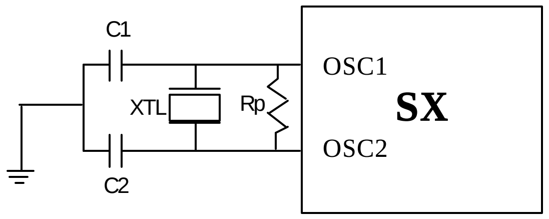


Figure F.19: SX with External Crystal.

Table F.4: Component Selection for Crystals.

Type	Frequency	C1	C2	Rp
XT	4 MHz	20 pF	47 pF	1 MΩ
HS	8 MHz	20 pF	47 pF	1 MΩ
HS	12 MHz	20 pF	47 pF	1 MΩ
HS	16 MHz	15 pF	30 pF	1 MΩ
HS	20 MHz	15 pF	30 pF	1 MΩ
HS	25 MHz	5 pF	20 pF	10 kΩ
HS	30 MHz	5 pF	20 pF	4.7 kΩ
HS	36 MHz	5 pF	15 pF	3.3 kΩ
HS	40 MHz	5 pF	15 pF	3.3 kΩ
HS	50 MHz	5 pF	10 pF	3.3 kΩ

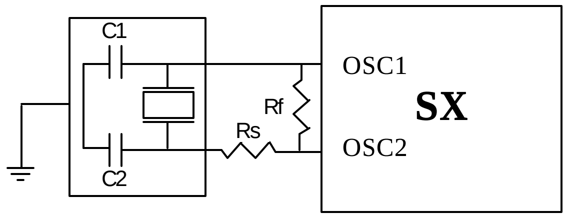


Figure F.20: SX with External Ceramic Resonator.

Table F.5: Component Selection for Ceramic Resonators.

Type	Frequency	C1	C2	Rp	Rs
XT	455 kHz	220 pF	220pF	1 MΩ	6.8 kΩ
XT	1 MHz	100 pF	100pF	1 MΩ	6.8 kΩ
XT	2 MHz	100 pF	100pF	100 kΩ	680 Ω
HS	4 MHz	100 pF	100pF	100 kΩ	0
HS	4 MHz	47 pF*	47 pF*	100 kΩ	470 Ω
HS	8 MHz	30 pF	30 pF	1 MΩ	0
HS	8 MHz	47 pF*	47 pF*	1 MΩ	470 Ω
HS	12 MHz	30 pF	30 pF	1 MΩ	0
HS	12 MHz	22 pF*	22 pF*	1 MΩ	0
HS	16 MHz	15 pF	15 pF	1 MΩ	0
HS	16 MHz	15 pF*	15 pF*	1 MΩ	0
HS	20 MHz	10 pF	10 pF	1 MΩ	0
HS	33 MHz	10 pF	10 pF	33 kΩ	0
HS	50 MHz	5 pF*	5 pF*	33 kΩ	0

* capacitors built in to resonator.

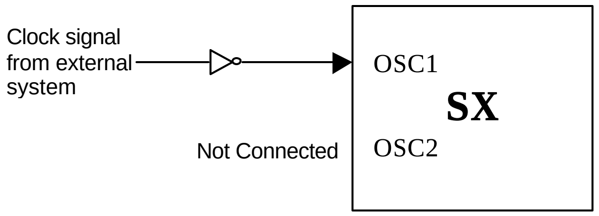


Figure F.21: SX with External system clock.

External RC Mode

For timing insensitive applications, the RC device option offers additional cost savings. The RC oscillator frequency is a function of the supply voltage, the resistor (Rext) and capacitor (Cext) values, and the operating temperature. In addition to this, the oscillator frequency will vary from unit to unit due to normal process variation. Furthermore, the difference in lead frame capacitance between package types will also affect the oscillation frequency, especially for low Cext values. Variation due to tolerance of external R and C components must also be considered.

Figure F.22: SX with External RC clock.

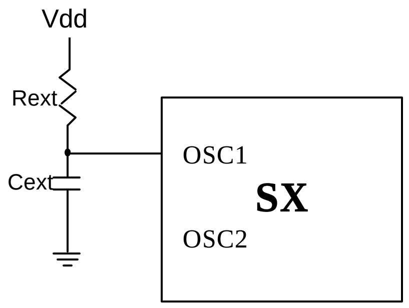


Figure F.17 shows the RC connections to the SX. For Rext values below 2.2kΩ, the oscillator operation may become unstable, or stop completely. For very high Rext values (e.g. 1 MΩ) the oscillator becomes sensitive to noise, humidity and leakage. The recommended Rext value is 3kΩ to 100kΩ.

Although the oscillator will operate with no external capacitor (Cext = 0 pF), using values above 20 pF is recommended for noise and stability reasons. With little external capacitance, the oscillation frequency can vary dramatically due to changes in external capacitance, such as PCB trace capacitance or package lead frame capacitance.

Internal RC Mode

The SX offers an internal 4 MHz RC oscillator for timing insensitive operations. Using the internal oscillator reduces external component count and system cost. The internal oscillator is configured via fuses in the Fuses byte. The SX-Key Development software's Device screen allows you to select the internal RC oscillator speed division factor (1, 2, 4, 8, 16, 32, 64 or 128). A Device directive is also offered, allowing source code to have the proper fuse configuration embedded within it.

SX Data Sheet

@ sign, 80-81
; (comments), 40
= (assignment), 41,45
= (equal), 46
< (less than), 46
> (greater than), 46
<> (not equal), 46
=< (equal or less than), 46
=> (equal or greater than), 46
|| (absolute value), 58
- (negative), 58
~ (not), 58
+ (addition), 58
- (subtraction), 58
* (multiplication), 58
/ (division), 58
// (modulus), 58
& (logical AND), 58
| (logical OR), 58
^ (logical XOR), 58
<< (shift left), 58
>> (shift right), 58
<> (reverse bits), 58
% (iteration symbol), 48
\ (argument symbol), 49
b (bit address), 90
f (file register), 90
k (constant), 90
!port, 90

- A -

ABSOLUTE VALUE (||),58
ADD, 91
ADDB, 91
ADDITION (+), 58
ADJUSTING INT. CYCLES, 75-78
AND, 91
AND (&) (logical), 58
ARGUMENT SYMBOL (\), 49
ASCII DATA TYPE, 59
ASSEMBLER DIRECTIVES, 39-55
  = (assignment), 41,45
  BREAK, 41,52
  CASE, 41,51
  DEVICE, 41,43
  DS, 41,45
  DW, 41,45-46
  ELSE, 41,46-47
  END, 41
  ENDIF, 41,46-47
  ENDM, 41,49
  ENDR, 41,47-48
  ERROR, 41,54-55
  EXITM, 41,49-50
  EXPAND, 41,50-51
  EQU, 41,45
  FREQ, 41,44
  ID, 41,44
  IF..ELSE..ENDIF, 41,46-47
  IFDEF..ELSE..ENDIF, 41,47
  IFNDEF..ELSE..ENDIF, 41,47
  MACRO, 41,49-51
  NOCASE, 41,51
  NOEXPAND, 41,50-51
  ORG, 41,44-45
  REPT, 41, 47-48
  RESET, 41,45
  WATCH, 41,52-54
ASSEMBLY, 39, 46-47
  Conditional, 46-7
  Program, typical, 39
ASSEMBLY CODE BOX, 28,29,32
ASSIGNMENT (=), 41,45

- B -

Index

BACKSPACE KEY, 31
BANK, 92,143
BINARY DATA TYPE, 59
BINARY OPERATORS (table), 58
BOR22, 43
BOR26, 43
BOR42, 43
BREAK (directive), 41,52
BREAKPOINTS, 32,52
  setting, 32,52
BROWNOUT (param.), 43

- C -

C (carry), (table) 90, 137
CALIBRATING IRC, 37
CALL, 79-82,85,80,88,92,145-146
CALLING
  Across pages, 79-82,145-146
CARRYX (param.), 43,88,90
CASE (directive), 41,51
CASE SENSITIVITY 51
CJA, 92
CJAE, 93
CJB, 93
CJBE, 94
CJE, 94
CJNE, 95
CLC, 95
CLR, 95
CLRB, 95,136
CLZ, 96
CMOS/TTL LOGIC
  (table), 64
  Configuration,61,64-65
CMP_B, 70
CODE BUTTON, 28
CODE PAGES, 79-82
CODE PROGRAM (param.), 36
COMMENTS (;), 39-40
COMPARATOR CONFIG., 61, 70-71
COMPARATOR REGISTER, 152
COMPARISON OPERATORS (table), 46
CONDITIONAL 46-47
  Assembly, 46-47
  Statements, 47
CONFIGURE WINDOW, 36
  Code Program param., 36
  Device Erase param., 36
  Fuse Program param., 36
  IRC Calibration, 36
CONTROL REGISTERS, 149-154
CRYSTAL COMPONENTS (table), 159
CRYSTAL- 1, 33,34, (table) 35
CRYSTAL- 2, 33,34, (table) 35
CRYSTAL- 3, 33,34, (table) 35
CRYSTAL- 4, 33,34, (table) 35
CRYSTAL- 5, 33,34, (table) 35
CRYSTAL- MAX, 33,34, (table) 35
CRYSTAL- MIN, 33,34, (table) 35
CSA, 96
CSAE, 96
CSB, 97
CSBE, 97
CSE, 98
CSNE, 98
CURRENT INSTRUCTION, 32
  Setting, 32
CURSOR KEYS, 30

- D -

DATA DIR. REG.,148,150,153
  TRIS_A, 148
  TRIS_B, 148

TRIC_C, 148
DATA TABLES, 77-79
8-bit, 77-78
12-bit, 78-79
DATA TYPES, 59
Size, 59
DATE CODE, SX, 5
DC (digit carry), (table) 90,137
DEBUG BUTTONS (figure), 28
DEBUG WINDOW, 28
DEBUG WINDOWS, 27-32
(figure), 28
Editing keys (table), 31
DEBUGGING, 27-32
requirements, 32
DEC, 98
DECIMAL DATA TYPE, 59
DECSZ, 99
DEFINE SPACE (directive), 41,45
DEFINE WORD (directive), 41,45
DEMO BOARD, SX-KEY,11-14,124-127
Configuring Hyperterminal, 13
Configuring Virtual Periph., 126-127
Connections, 7
Connecting SX-Key/Blitz, 7
Features, 83-84
Plugging SX into, 7
Schematic, 123
Sxdemo.src, 7
Virtual Peripheral Demo, 13
DETECTING EDGES, 66-70
DEVICE (directive) 41-43
Compatible (table), 44
HS_OSC, 44
LP_OSC, 44
RC_OSC, 44
XT_OSC, 44
PIC16C54, 44
PIC16C55, 44
PIC16C56, 44
PIC16C57, 44
PIC16C58, 44
PROTECT_OFF, 44
PROTECT_ON, 44
WDT_OFF, 44
WDT_ON, 44
Settings (table), 43
DEVICE ERASE (param.), 36
DEVICE WINDOW, (fig) 33-34
Load Hex button, 33-34
Options, 33-34
Oscillator, 33-34
Frequency, 41,43
Internal, 43
Program button, 33-34
Read button, 33-34
Save Hex button, 33-34
Verify button, 33-34
DIFFERENCES FROM PICS, 19-20
DIRECT/INDIR. ADDR.,139,143
DIRECTION CONGIF., PORT, 61-63
DIRECTIVES, 39-55
(table), 41
DISABLING INTERRUPTS, AUTO, 68
DISABLING MACRO EXPAND, 50-51
DISCUSSION LIST, ii
DIVISION (/), 58
DJNZ, 99
DRT18MS, 43
DRT1920MS, 43
DRT480MS, 43
DRT60MS, 43
DRT60US, 43
DRT8MS, 43
DRT960MS, 43
DRTOFF, 43

Index

DS (directive), 41,45
DW (directive), 41,45, 77-78

- E -

E^{2}Flash, 33-34
EDGE DETECTION
Configuration, 61,66-65
Interrupts, 61,68-70,157
EDGE SELECTION REG., 66-67
EDIT MENU, 25
EDITING KEYS, DEBUG (table), 31
EDITOR, 23-37
(figure), 23
Copy, 24
Cut, 24
Editing shortcut keys (table), 24
Menus, 24-26
Edit, 25
File, 24
Help, 26
Run, 25
Page down, 24
Page up, 24
Paste, 24
Tab, 24
Text bigger, 24
Text sizing, 24
Text smaller, 24
Windows, 26-37
ELSE (directive), 41,46-47
END (directive), 41
ENDIF (directive), 41,46-47
ENDM (directive), 41,49
ENDR (directive), 41,47-48
ENTER KEY, 31
EMAIL, SUPPORT, ii
EQU (directive), 41,45
EQUAL (=) (operator), 45
EQUAL OR LESS THAN (=<)(operator), 46
EQUAL OR GREATER THAN (=>) (operator), 46
ERRATA SHEET, 5
ERROR (directive), 41,54-55
ERROR MESSAGES, 115-120
ESC KEY, 31
EXITM (directive), 41,49-50
EXPAND (directive), 41,50-51
EXPANDING MACROS, 50-51
EXPRESSIONS, 57-58
EXTERNAL RC MODE, 160-161

- F -

FILE MENU, 24
FILE SELECT REGISTER, 138-143
FIND TEXT, 27
FIND/REPLACE TEXT, 27
FORMAT SETTING, WATCH
(table), 52
Watch (directive), 52-54
FR, 90
FREQ (directive), 32,41,44
FSR, 90,135, 138-143, (table) 141
FSR INDICATOR, 28
FSTR (formatter), 52,54
FTP SITE, ii
FUSE PROGRAM (param.), 36
FUSES, 156-157

- G -

GLOBAL LABELS, 56-57
GREATER THAN (>) (operator), 46

- H -

HELP MENU, 26
HEX FILES, 35
  Loading, 5
  Saving, 35
HEXADECIMAL DATA TYPE, 59
HOP BUTTON, 28,29
HS MODE, 158-159
HS_OSC (param.), 44

- I -

I/O PINS, 61-71
ID (param.), 33-34
ID (directive), 41,44
IND, 135,141-142
INDIRECT ADDRESSING, 141-142
INPUT/OUTPUT CONFIG., 61-63
INTERNAL RC MODE, 33-37
IF..ELSE..ENDIF (directive), 41,46-47
IFDEF..ELSE..ENDIF (directive), 41, 47
IFNDEF..ELSE..ENDIF (directive),41,47
IJNZ, 99
INC, 99
INCSZ, 100
INPUT (clock), 35,43
INSTRUCTION
  Multi-word (table), 87
  Pipeline, 132-133
  Quick ref. (table), 88
  Single-word (table), 86
  Summary (table), 85
INT FLAG, 28
INTERRUPT
  Adjusting cycles, 75-77
  Debugging, 73-74
  Latency delays, 72
  Multiple, 73
  Queueing, 73
  Routine size, 73
  Vector, 72
INTERRUPTS, 68-70, 72-77, 157-158
IRC, 36-37
  Calibration, 36-37
IRC CALIBRATION, 37
IREAD, 76,78-79,100
ITERATION SYMBOL (%), 47-48

- J -

JOG BUTTON, 28,29
JUMP, 79-80
JUMPING ACROSS PAGES, 79-80,101

- L -

LABELS, 39,56-57
  Global, 56-57
  Local, 56-57
  Macro, 56-57
LESS THAN (<) (operator), 46
LIST GENERATION, 24
LIST TOGGLE, 24
LOAD HEX BUTTON, 33-35
LOADING OBJECT FILES, 33-35
LOCAL LABELS, 56-57
LOGIC LEVEL CONGIF., 61,64-65
LP MODE, 158
LP_OSC (param.), 44
JB, 100
JMP, 101
JNB, 101
JNC, 101
JZ, 102
LCALL, 102

Index

LJMP, 103
LSET, 103
LVL_A, 149
LVL_B, 150
LVL_C, 153

- M -

M, 90
MACRO, 41, 49-51, 56-7
  (directive), 41,49-51
  Expanding, 51
  Labels, 56-57
MNEMONICS, 39
MODE, 103
MODE REGISTER, 61-62,154-155
MODE VALUES (table), 62,155
MODIFYING REGISTERS, 31
MODULUS (//), 58
MOV, 104
MOVB, 105
MOVSZ, 105
MULTI-WORD INST. (table), 87
MULTIPLICATION (*), 58

- N -

NEGATIVE (-), 58
NOCASE (directive), 41,51
NOEXPAND (directive), 50,51
NOP, 105
NOT, 106
NOT (~) (operator), 58
NOT EQUAL (<>) (operator), 46

- O -

OBJECT FILES, 35
  Loading, 35
  Saving, 35
OE (bit), 71
OP.BIT, 89
OPERATORS, 46,58
  Binary, 58
  Conditional, 46
  Unary, 58
OPT, 89
OPTION REGISTER, 156
OPTIONS, DEVICE WINDOW, 33-34
OPTIONX (param.), 43
OR, 106
OR, (|) (logical), 58
ORG (directive), 41,44-45
OSCHS (param.), 44
OSCILLATOR DRIVER, 158
OSCLP (param.), 44
OSCRC (param.), 44
OSCXT (param.), 44
OSCILLATOR, DEVICE WINDOW, 33-34
  Frequency, 44
  Internal, 33-37

- P -

PACKING LIST, 5-6
  SX Tech Tool Kit, 5
  SX University Kit, 6
  SX-Blitz, 6
PAGE, 80,106,136
PARALLAX INSTR. SET, 85-112
PC, 90,135,136
PD (power down), 90,137
PENDING REGISTERS, 68-70,73,152
PERIPHERALS, 158-161
PIC16C54 (param.), 44
PIC16C54A (param.), 44
PIC16C55 (param.), 44
PIC16C56 (param.), 44
PIC16C57 (param.), 44
PIC16C58 (param.), 44
PIC16C58A (param.), 44
PICS, 19-20
  Debugging, 19-20
  Fuses, 20
  Similarities with, 19
  Differences, 19-20
  I/O pins, 20
  Instruction execution, 19,20
  Interrupts, 20
  Oscillator, 20
  Pinout, 19
  Opcodes, 19
  Program space, 19
  RAM registers, 19
  Stack, 20
PINS, 61-71
  I/O, 61-71
PLP_A, 149
PLP_B, 150
PLP_C, 153
POLL BUTTON, 28,29
POLLING, 28-29
POP, 147
PORT A REGISTERS, 148-149
PORT B REGISTERS, 149-152
PORT C REGISTERS, 152-154
PORT CONFIG. REGISTERS, 148-154
PORT CONFIGURATION, 61-62
PRESCALER DIVISION RATIO, 156
PRINT WINDOW, 26
PRINTING, 26
PROGRAM BUTTON, 33-34,35
PROGRAM COUNTER (PC), 29, 32
  Setting, 32
PROTECT (param.), 43
PROTECT_OFF (param.), 44
PROTECT_ON (param.), 44
PULL-UP RESISTER CONFIG. ,61,62-63
PULL-UP RESISTER REG.,149,150,153
  PLP_A, 149
  PLP_B, 150
  PLP_C, 153
PUSH, 146

- Q -

QUICK PROTO BOARD,14-15, 128-130
  Connecting SX-Key/Blitz, 14
  Connections (figure), 15
  Features, 129
  Plugging SX into, 14
  Schematic, 130
QUICK START, 9-15
  SX-Key Demo Board, (fig) 11,11-14
  SX-Key Quick Proto Board, (fig) 14, 14-15

SX Tech Board, 9-11

- R -

RA, 134,135
RAM REGISTER MAP, 134, (figure),134
RB, 134
RC, 134
RC_OSC (param.), 44
READ BUTTON, 33-35
READ-MODIFY-WRITE, 133
READING, 35
REGISTER MAP, 134
REGISTERS, 29-31
Binary representation of, 29
Modifying, 30
Undoing modifications, 30
REGISTERS BUTTON, 30
REGISTERS WINDOW
(figure), 29, 28-29
REPEAT (REPT) (directive), 41,47-48
REPLACE TEXT, 27
REPT (directive), 41, 47-48
REQUIREMENTS, 17,32
Debugging, 32
System, 17
RES (bit), 71
RESERVED WORDS (table), 113
RESET (directive), 41,45
RESET BUTTON, 29,30
RESONATOR COMPONENTS
(table),160
RET, 106,146
RETI, 74-75,107,146
RETIW, 74-76,107,146
RETP, 107,146
RETW, 77-79,107
RETURNS, 148
RETURNING FROM SUBROUTINE, 146
RETW, 107,146
REVERSE BITS (<>), 58
RL, 108,146-147
ROTATE LEFT, 108, 146-147
ROTATE RIGHT, 108,146-147
RR, 108, 146-147
RTCC, 135,136
RTCC INTERRUPTS, 74-77,158
RTE, 74
RTI, 74
RTS, 74
RUN BUTTON, 28,29
RUN MENU, 25

- S -

SAVE HEX BUTTON, 33,34,35
SAVING OBJECT FILES, 35
SB, 108
SC, 108
SCHMITT-TRIGGER
(figure), 65
Configuration.,61,65-66
Registers., 151, 154
ST_B, 151
ST_C, 154
SBIN (formatter), 52
SDEC (formatter), 52
SHEX (formatter), 52
SETB, 109,136
SETTING BREAKPOINTS, 32,52
SETTING CURRENT INSTRUCTION, 32
SHIFT LEFT (<<), 58
SHIFT RIGHT (>>), 58
SIMILARITIES WITH PICS, 19
SINGLE-WORD INSTR. (table), 86

SKIP, 109
SKIP FLAG, 31
SKIP INSTRUCTIONS, 90
SLEEP, 68,70,71,77,109
SNB, 109
SNC, 110
SNZ, 110
SOCKETS, SX48/52, 8
SOURCE CODE WINDOW, 28
SPACE KEY, 31
SPECIAL FUNCTION REG., 135-143
ST_B, 65-66
ST_C, 65-66
STACK, 146-147
STACK OVERFLOW, 147
STACK UNDERFLOW, 148-148
STACKX (param.), 43
STACKX_OPTIONX (param.), 43
STATUS, 137
STATUS REGISTER, 136-138
STC, 110
STEP BUTTON, 29,30
STOP BUTTON, 29,30
STZ, 110
SUB, 111
SUBB, 111
SUBTRACTION (-), 58
SUPER PARALLAX ASSEMBLER
  @ Sign, 21-22
  Device settings, 22
  Differences, 21-22
  Directives and equates, 21
  Environment, 21
  Language syntax, 21
  Mnemonics, 21
  New mnemonics, 21
  Similarities with, 21
SUPPORT, ii, 5
  Discussion List, ii
  Email, ii
  FTP Site, ii
  Web Site, ii, 5
SWAP, 111
SX18L (param.), 43
SX28L (param.), 43
SX ARCHITECTURE, 132-133
  Instruction pipeline, 132-133
SX CHIP FEATURES, 83-84
SX DATA SHEET, 131-161
  C (carry), 137
  Comparator register, 152
  Control registers, 154-156
  Crystal components (table), 159
  DC (digit carry), 137
  Data direction registers,148,150,153
    TRIS_A, 148
    TRIS_B, 150
    TRIC_C, 153
  TTL/CMOS Selection reg., 149,150,153
    LVL_A, 149
    LVL_B, 150
    LVL_C, 153
  Direct/Indirect Addressing, 139,143
  External RC mode, 160-161
  File Select Register, 135, 138-143
  FSR, 135, 138-143
  Fuses, 156-157
  HS mode, 158
  IND, 142,143
  Indirect addressing, 141-143
  Internal RC mode, 161
  LP mode, 158
  Mode register, 154-155
  Option register, 156
  Oscillator driver, 158
  PC, 135,136
  PD (power down), 137

Index

Pending registers., 152
Peripherals, 158-161
POP, 147
Port A registers, 148-149
Port B registers, 149-152
Port C registers, 152-154
Port config. registers, 148-154
Prescaler division ratio, 156
Pull-Up resister reg., 149,150,153
    PLP_A, 149
    PLP_B, 150
    PLP_C, 153
Push, 146
RA, 135
RB, 135
RC, 135
Read-modify-write, 133
Resonator components (table), 160
Returns, 148
Returning from a subroutine, 146
Rotate left, 108, 137-138
Rotate right, 108,137-138
RTCC, 135,148
Schmitt-Trigger reg., 151, 154
    ST_B, 151
    ST_C, 154
Special function registers, 135-143
Stack, 146
Stack overflow, 147
Stack underflow, 148-149
Status, 137
Status register, 136-137
TO (time out), 137
Wake-Up Enable reg., 151
Wake-Up Edge reg., 151
WREG, 135
XT mode, 155
    Z (zero bit), 137
SX CHIP DATE CODES, 5
SX-KEY ERROR MESSAGES, 115-120
SX-KEY/BLITZ HARDWARE CONNECTION
    Clock source, 7
    Debugging, 7
    Four-pin connector (figure), 7
    Power, 7
    Programming, 7,35
SX-KEY/BLITZ INTERFACE, 23-37
SYMBOLS, 55-56
SYNC (param.), 43
SYSTEM REQUIREMENTS, 17
SZ, 112

- T -

TAB, 31
TECH BOARD, SX, 9-11
    Connections, 9
    Features, 122
    Plugging SX into, 9
    Schematic, 123
TEST, 112
TO (time out), 137
TRIS_A, 148
TRIS_B, 149
TRIC_C, 143
TRIS REGISTERS, 62
TTL/CMOS SELECTION REG., 149,150,153
    LVL_A, 149
    LVL_B, 150
    LVL_C, 153
TTL LOGIC LEVEL (table), 64
TURBO (param.), 43

- U -

UBIN (formatter), 52

UDEC (formatter), 52
UHEX (formatter), 52
UNARY OPERATORS (table), 58

- V -

VERIFY BUTTON, 33,34,35
VERIFYING, 35
VERSIONS, ii
VIRTUAL PERIPHERALS, 13-14

- W -

W, 90
WAKE-UP ENABLE REG., 151
WAKE-UP EDGE REG., 151
WAKE-UP INTERRUPTS, 61, 68-70
WALK BUTTON, 28,29-30
WARRANTY, ii
WATCH (directive), 41,52-54
WATCH BUTTON, 30
WATCH WINDOW, 28,31,52-53
    Modifying values, 31
WATCHDOG (param.), 43
WATCHING REGISTERS, 30, 52-54
WEBSITE, ii, 5
WKED_B, 66,68
WKPEN_B, 66,68
WREG, 135

- X -

XOR, 112
XOR (^) (logical), 5583
XT MODE, 158
XT_OSC (param.), 44

- Z -

Z (zero bit), 90,137
ZSTR (formatter), 52,54

Index