

Programming the SX Microcontroller

A Complete Guide by Günther Daubach

2ND EDITION

WARRANTY

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

14-DAY MONEY BACK GUARANTEE

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

COPYRIGHTS AND TRADEMARKS

This documentation is copyright 2004 by Parallax Inc. By downloading or obtaining a printed copy of this documentation or software you agree that it is to be used exclusively with Parallax products. Any other uses are not permitted and may represent a violation of Parallax copyrights, legally punishable according to Federal copyright or intellectual property laws. Any duplication of this documentation for commercial uses is expressly prohibited by Parallax Inc.

BASIC Stamp, Stamps in Class, Board of Education, Boe-Bot, SumoBot, Toddler, and SX-Key are registered trademarks of Parallax, Inc. HomeWork Board, Parallax, and the Parallax logo are trademarks of Parallax Inc. If you decide to use registered trademarks of Parallax Inc. on your web page or in printed material, you must state that "(trademark) is a (registered) trademark of Parallax Inc." upon the first appearance of the trademark name in each printed document or web page. Other brand and product names are trademarks or registered trademarks of their respective holders.

ISBN 1-928982-16-6

2007 Aug. 2nd Ed. 2nd Pr.

DISCLAIMER OF LIABILITY

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

TECHNICAL SUPPORT AND INTERNET RESOURCES

We offer a variety of internet resources and avenues for receiving technical support:

E-mail: support@parallax.com
Web: www.parallax.com/sx
Phone: 916.624.8333
Toll-free: 888.99.STAMP (Continental U.S. only)
Forums: forums.parallax.com/forums

Our moderated forums include one specifically for people interested in programming the SX microcontroller with Parallax assembly language SX-Key tools, and third-party SX/B (BASIC) and C compilers.

ERRATA

While great effort is made to assure the accuracy of our texts, errors may still exist. If you find an error, please send an email to editor@parallax.com. If an errata sheet is necessary, it will be posted to this book's product page (product #70002) at www.parallax.com.

Table of Contents

1	Section I - Tutorial _____	3
1.1	Introduction _____	3
1.2	SX Development - What You Need _____	5
1.2.1	The Tools _____	5
1.2.2	Prototyping Systems _____	5
1.2.3	A "Home-brew" Prototyping System _____	5
1.3	SX-Key Quick Start using the Parallax SX-Key _____	8
1.3.1	The SX-Key _____	8
1.3.2	Installing the SX-Key IDE Software _____	8
1.3.3	The First Program _____	9
1.3.4	The SX-Key Debugger Windows _____	15
1.3.4.1	The Registers (R) Window _____	15
1.3.4.2	The Code - List File (C) Window _____	15
1.3.4.3	The Debug (D) Control Window _____	16
1.3.5	Executing the First Program in Single Steps _____	16
1.3.6	Compound Instructions _____	17
1.3.7	Symbolic Addresses - Labels _____	20
1.3.8	Running the Program at Full Speed _____	21
1.3.9	Program Loops for Time Delays _____	23
1.3.10	Setting a Breakpoint _____	26
1.3.11	Where to Go Next _____	27
1.4	SX Configuration - ORG/DS - Conditional Branches _____	28
1.4.1	Configuration Directives _____	28
1.4.2	The ORG and DS Directives _____	29
1.4.3	Conditional Branches _____	31
1.5	Subroutines - Symbols - Data Memory _____	33
1.5.1	Subroutines _____	33
1.5.2	The Stack _____	34
1.5.3	Local Labels _____	36
1.5.4	Some More Considerations about Subroutines _____	38
1.5.5	Correctly Addressing the SX Data Memory _____	41
1.5.6	Clearing the Data Memory and Indirect Addressing _____	45
1.5.6.1	SX 18/20/28 _____	45
1.5.6.2	SX 48/52 _____	50
1.5.7	Symbolic Variable Names _____	52

Programming the SX Microcontroller

1.5.8	The EQU, SET and = Directives _____	53
1.5.9	Some Thoughts about Data Memory Usage _____	55
1.5.10	Don't Forget to Select the Right Bank _____	56
1.5.11	Saving the Current Bank in a Subroutine _____	58
1.5.12	Routines for an FSR Stack _____	60
1.5.13	The "#"-Pitfall _____	64
1.6	Arithmetic and Logical Instructions _____	67
1.6.1	Arithmetic Instructions _____	67
1.6.1.1	Addition _____	67
1.6.1.2	Skip Instructions _____	69
1.6.1.3	The TEST Instruction _____	69
1.6.1.4	Multi-Byte-Addition _____	70
1.6.1.5	Subtraction _____	71
1.6.1.6	Signed Numbers _____	72
1.6.2	Incrementing and Decrementing _____	73
1.6.3	Arithmetic Instructions and Multi-Byte Counters _____	74
1.6.4	The DEVICE CARRYX Directive _____	76
1.6.5	Logical Operations _____	76
1.6.5.1	AND _____	76
1.6.5.2	OR _____	78
1.6.5.3	XOR _____	79
1.6.5.4	NOT _____	80
1.6.6	Rotate instructions _____	80
1.6.6.1	Multiplication and Division _____	81
1.6.7	The SWAP Instruction _____	87
1.6.8	The DC (Digit Carry) Flag _____	87
1.6.9	MOV Instructions with Arithmetic Functions _____	87
1.7	MOV Instructions _____	90
1.7.1	Basic MOV Instructions _____	90
1.7.2	Compound MOV Instructions _____	91
1.8	Recognizing Port Signals _____	94
1.8.1	Recognizing Signal Edges at Port B _____	94
1.8.1.1	MODE and Port Configuration Registers _____	95
1.8.1.2	Signal Edges at Port B _____	97
1.8.1.3	De-bouncing Mechanical Contacts _____	99
1.9	Interrupts - The OPTION Register _____	102
1.9.1	Interrupts _____	102
1.9.1.1	Asynchronous Interrupts _____	104

Table of Contents

1.9.1.2	Synchronous (Timer-Controlled) Interrupts	106
1.9.1.3	The Prescaler	110
1.9.1.4	Interrupts Caused by Counter Overflows	116
1.9.2	The OPTION Register Bits and their Meaning	120
1.10	The Watchdog - Reset Reasons - Conditional Assembly	121
1.10.1	The Watchdog Timer	121
1.10.2	Determining Reset Reasons	124
1.10.3	Conditional Assembly	125
1.10.3.1	More Directives for Conditional Assembly	126
1.10.4	"To Watchdog or not to Watchdog..."	128
1.11	The Sleep Mode and Wake-up	129
1.11.1	Wake-ups Caused by Port B Signal Edges	129
1.11.2	Using the Watchdog Timer for Wake-ups	133
1.12	Macros - Expressions - Symbols - Device Configuration	135
1.12.1	Macro Definitions	135
1.12.1.1	Macros or Subroutines?	138
1.12.2	Expressions	140
1.12.3	Data Types	141
1.12.4	Symbolic Constants	141
1.12.5	The DEVICE Directives	144
1.12.6	The FREQ Directive (SX-Key only)	146
1.12.7	The ID Directive	146
1.12.8	The BREAK Directive (SX-Key only)	146
1.12.9	The ERROR Directive	147
1.12.10	The END Directive	147
1.13	The Analog Comparator	148
1.13.1	The Comparator and Interrupts	150
1.13.2	The Comparator and the Sleep Mode	151
1.14	System Clock Generation	152
1.14.1	The Internal Clock Generator	152
1.14.2	Internal Clock Generator with External R-C Network	153
1.14.3	External Crystal/Ceramic Resonator	153
1.14.4	External Clock Signals	154
1.14.5	External Clock Signal using a PLL	154
1.14.6	Selecting the Appropriate Clock Frequency	155
1.15	The Program Memory	157

Programming the SX Microcontroller

1.15.1	Organizing the Program Memory	157
1.15.1.1	The PAGE Instruction	158
1.15.1.2	The @jmp and @call Options	160
1.15.1.3	Subroutine Calls across Memory Pages	161
1.15.2	How to Organize Program Memory	163
1.16	Tables - RETIW and IREAD	164
1.16.1	Tables	164
1.16.1.1	The RETW Instruction	164
1.16.1.2	Reading Program Memory Using the IREAD Instruction	169
1.17	More SX Instructions	172
1.17.1	Compare Instructions	172
1.17.1.1	CJA (Compare and Jump if Above)	172
1.17.1.2	CJAE (Compare and Jump if Above or Equal)	172
1.17.1.3	CJB (Compare and Jump if Below)	173
1.17.1.4	CJBE (Compare and Jump if Below or Equal)	173
1.17.1.5	CJE (Compare and Jump if Equal)	173
1.17.1.6	CJNE (Compare and Jump if Not Equal)	174
1.17.2	Decrement/Increment with Jump	174
1.17.2.1	DJNZ (Decrement and Jump if Not Zero)	174
1.17.2.2	IJNZ (Increment and Jump if Not Zero)	174
1.17.3	Conditional Jumps	174
1.17.3.1	JNB (Jump if Not Bit set)	174
1.17.3.2	JNC (Jump if Not Carry set)	175
1.17.3.3	JNZ (Jump if Not Zero)	175
1.17.4	Conditional Skips	175
1.17.4.1	CSA (Compare and Skip if Above)	175
1.17.4.2	CSAE (Compare and Skip if Above or Equal)	175
1.17.4.3	CSB (Compare and Skip if Below)	176
1.17.4.4	CSBE (Compare and Skip if Below or Equal)	176
1.17.4.5	CSE (Compare and Skip if Equal)	176
1.17.4.6	CSNE (Compare and Skip if Not Equal)	176
1.17.5	MOV and Conditional Skip	177
1.17.5.1	MOVSZ (MOVE and Skip if Zero)	177
1.17.6	NOP (No Operation)	177
1.17.7	SKIP	177
1.18	Virtual Peripherals	179
1.18.1	The Software UART, a VP Example	179
1.18.1.1	The Transmitter	185

Table of Contents

1.18.1.2	The Receiver	187
1.18.1.3	Utility Routines	189
1.18.1.4	The Main Program	191
1.18.1.5	Handshaking	192
1.18.2	Conclusion	193
2	Section II - Reference	197
2.1	Introduction	197
2.2	The SX internals (simplified)	199
2.2.1	How SX Instructions are Constructed	202
2.2.2	Organization of the Data Memory and how to Address it	203
2.2.2.1	SX 18/20/28	203
2.2.2.2	SX 48/52	205
2.2.3	Organization of Program Memory and how to Access it	207
2.2.4	The SX Special Registers and the I/O Ports	209
2.2.4.1	The W Register	209
2.2.4.2	The I/O Registers (Ports)	210
2.2.4.3	Read-Modify-Write Instructions	210
2.2.4.4	Port Block Diagram	211
2.2.4.5	The Data Direction Registers	212
2.2.4.6	The Level Register	213
2.2.4.7	Pull-up Enable Registers	213
2.2.4.8	The Schmitt Trigger Enable Registers	214
2.2.4.9	The Port B Wake Up Configuration Registers	214
2.2.4.10	The Port B Analog Comparator	217
2.2.4.11	More Configuration Registers (SX 48/52)	217
2.2.4.12	Addressing the I/O Configuration Registers (SX 18/20/28)	218
2.2.4.13	Addressing the SX 48/52 I/O Configuration Registers	220
2.2.5	Interrupts, Watchdog and Brown-Out	222
2.2.5.1	Interrupts	222
2.2.5.2	The Watchdog Timer	225
2.2.5.3	Additional Bits in the OPTION Register	226
2.2.5.4	Monitoring V_{DD} - The Brown-Out Detection	227
2.2.5.5	Determining the Reason for a Reset	227
2.2.6	The Stack Memory	229
2.2.7	The FUSE Registers	230
2.2.7.1	The FUSE Registers (SX18/20/28)	230
2.2.7.2	The Fuse Registers (SX 48/52)	234
2.2.8	The SX 48/52 Multi-Function Timers	237

Programming the SX Microcontroller

2.2.8.1	PWM Mode _____	238
2.2.8.2	Software Timer Mode _____	238
2.2.8.3	External Event Counter _____	238
2.2.8.4	Capture/Compare Mode _____	239
2.2.8.5	The SX48/52 Timer Control Registers _____	239
3	Section III – Quick Reference _____	247
3.1	SX Pin Assignments _____	247
3.2	Commonly used Abbreviations _____	249
3.3	Instruction Overview _____	252
3.3.1	Comments on the Instruction Overview Tables _____	252
3.3.2	Instructions in Alphabetic Order _____	253
3.3.3	Instructions by Functions _____	256
3.4	Special Registers _____	260
3.4.1	Option _____	260
3.4.2	Status _____	260
3.4.3	FSR _____	261
3.5	Addressing the Port Control Registers _____	261
3.5.1	SX 18/20/28 _____	261
3.5.2	SX 48/52 _____	262
3.6	Port Control Registers _____	264
3.6.1	TRIS (Direction) _____	264
3.6.2	LVL (Level Configuration) _____	264
3.6.3	PLP (Pull-up Configuration) _____	265
3.6.4	ST (Schmitt Trigger Configuration) _____	265
3.6.5	WKEN_B (Wake Up Enable) _____	265
3.6.6	WKED_B (Wake Up Edge Configuration) _____	266
3.6.7	WKPND_B (Wake Up Pending Flags) _____	266
3.6.8	CMP_B (Comparator) _____	266
3.6.9	T1CNTA (Timer 1 Control A) (SX 48/52 only) _____	267
3.6.10	T1CNTB (Timer 1 Control B) (SX 48/52 only) _____	267
3.6.11	T2CNTA (Timer 2 Control A) (SX 48/52 only) _____	268
3.6.12	T2CNTB (Timer 2 Control B) (SX 48/52 only) _____	268
4	Section IV - Applications _____	271
4.1	Function Generators with the SX _____	271
4.1.1	A Simple Digital-Analog Converter _____	272

Table of Contents

4.1.2	A Ramp Generator _____	273
4.1.2.1	A Ramp Generator With a Pre-defined Frequency _____	273
4.1.3	Generating a Triangular Waveform _____	277
4.1.4	Generating Non-linear Waveforms _____	279
4.1.4.1	Sine Wave _____	280
4.1.4.2	Sine Generator with a Defined Frequency _____	282
4.1.4.3	Superimposed Sine-Waves _____	283
4.1.4.4	Generating a Sine Wave from a 1 st Quadrant Table _____	284
4.1.4.5	Generating Other Waveforms _____	287
4.2	Pulse Width Modulation (PWM) with the SX Controller _____	288
4.2.1	Simple PWM VP _____	288
4.2.2	PWM VP with constant Period _____	290
4.2.3	More Areas Where PWM is Useful _____	292
4.3	Analog-Digital Conversion with the SX _____	293
4.3.1	Reading a Potentiometer Setting _____	293
4.3.1.1	Reading more Potentiometer Settings _____	298
4.3.2	A/D Converter Using Bitstream Continuous Calibration _____	301
4.4	Timers as Virtual Peripherals _____	307
4.4.1	A Clock Timer - an Example _____	307
4.4.2	General Timer VPs und Timed Actions _____	311
4.4.2.1	Execution within the ISR _____	311
4.4.2.2	Testing the Timers in the Mainline Program _____	311
4.5	Controlling 7-Segment LED Displays _____	314
4.5.1	Program Variations _____	320
4.6	An SX Stopwatch _____	326
4.7	A Digital SX Alarm Clock _____	338
4.7.1	When the Clock is Wrong... _____	355
4.8	Voltage Converters _____	357
4.8.1	A Simple Voltage Converter _____	357
4.8.2	A Regulated Voltage Converter _____	358
4.9	Testing Port Outputs _____	361
4.10	Reading Keyboards _____	364
4.10.1	Scanning a Key Matrix, First Version _____	367
4.10.1.1	Decoding the Key Number _____	370
4.10.1.2	Initial "Quick Scan" _____	371

Programming the SX Microcontroller

4.10.2	Quick-Scan and 2-Key Rollover _____	372
4.10.3	Need more Port Pins for the Keyboard Matrix? _____	378
4.11	An “Artificial” Schmitt Trigger Input _____	380
4.12	A Software FIFO _____	382
4.13	I ² C Routines _____	388
4.13.1	The I ² C Bus _____	388
4.13.2	The Basic I ² C Protocol _____	389
4.13.2.1	“Master” and “Slave” _____	389
4.13.2.2	The Start Condition _____	389
4.13.2.3	Data Transfer, and Clock Stretching _____	389
4.13.2.4	Acknowledge Message from the Slave _____	390
4.13.2.5	The Stop Condition _____	390
4.13.2.6	The Idle State _____	390
4.13.2.7	Bus Arbitration _____	390
4.13.2.8	Repeated Transmissions _____	391
4.13.3	The I ² C Data Format _____	391
4.13.4	Bus Lines and Pull-up Resistors _____	394
4.13.5	I ² C Routines for the SX Controller _____	395
4.13.5.1	Common Program Modules _____	406
4.13.5.2	The Mainline program _____	407
4.13.5.3	The I ² C Master VP _____	408
4.13.5.4	The I ² C Slave VP _____	409
4.14	A “Hardware Timer” _____	411
4.15	A Morse Code Keyer _____	413
4.16	Robotics - Controlling the Parallax SX Tech Bot _____	425
4.16.1	Introduction _____	425
4.16.2	Controlling the SX Tech Bot Servos _____	428
4.16.3	The Basic Control Program _____	429
4.16.3.1	Calibrating the Servos _____	432
4.16.3.2	More Parts of the Control Program _____	433
4.16.4	Some Timing Considerations _____	436
4.16.5	The SX Tech Bot’s First Walk (in the Park) _____	437
4.16.6	Adding a “Joystick” to the SX Tech Bot _____	437
4.16.7	The SX Tech Bot “Learns” to Detect Obstacles _____	439
4.16.7.1	The Control Program for the Obstacle-Detecting SX Tech Bot _____	442
4.16.7.2	Some More Thoughts About the Obstacle-Detecting SX Tech Bot _____	447

Table of Contents

4.17	More Ideas for SX Tech Bot Applications _____	448
5	Index _____	453

Programming the SX Microcontroller

A Complete Guide by Günther Daubach

2ND EDITION

Section I - Tutorial

1 Section I - Tutorial

1.1 Introduction

This first part of the book is intended to give you a step-by-step introduction in how to use a development system for the SX controller, and how to write the first applications for the SX.

Development systems for the SX are offered by several vendors. In this introduction, we will describe the SX-Key development system offered by Parallax.

In this text, you will find several sections that are marked in gray, together with one of the symbols below:



The exclamation mark indicates important information. You should read this text in any case to avoid problems.



This symbol marks a section that contains useful additional information, which is not necessary for understanding the current topic.



The Tutorial part of this book does not describe every feature of the SX in detail. The "R" symbol followed by a chapter and a page number indicates that more information about a topic can be found in the reference section of this book.



This symbol marks a section that contains useful additional information, which is not necessary for understanding the current topic.

Throughout the text, we have to deal with addresses, data, and values. The SX handles and stores all these in binary format, i.e. as a collection of bits where each bit can be set (1) or cleared (0). As the data memory is organized in 8-bit registers, data is always handled in bytes, i.e. in groups of 8 bits. Instead of writing binary numbers, we will use two hexadecimal digits to represent the contents of a register in most cases. The SX program memory is addressed with 12 bit values. To represent an address, we usually use three hexadecimal digits. Sometimes, when it comes to time calculations, etc. it is easier for us human beings to do the calculations in decimal. In order to distinguish between the different number types, we use the similar notation, most of the SX Assemblers allow:

Programming the SX Microcontroller

A leading "%" for binary numbers, a leading "\$" for hexadecimal numbers, and no special character for decimal numbers, e.g.

`%1011 1100 = $BC = 188`

Sometimes, you may also find a notation like `0xbc`, an alternative notation for hexadecimal numbers. C programmers are used to it quite well. Most available Assemblers for the SX also accept the "postfix" notation for binary, and hexadecimal values, e.g. `10010101B`, or `FFH`, but we will not use this format in the book text.

In the tutorial example programs, we sometimes make use of instructions that are not always explained when they are used first. Please refer to the "Alphabetic Instruction Overview" in the Quick Reference section of this book when you want to learn more about the function of a specific instruction.

1.2 SX Development - What You Need

1.2.1 The Tools

When you plan to develop software and hardware for a new type of microcontroller, you usually need to buy new "tools", meaning a financial investment. For the SX, this is the case as well but fortunately, several vendors offer moderately priced development systems for the SX.

One reason why development systems for the SX can be offered cheaper than systems for other microcontrollers lies in the SX itself: It has "built-in" debugging capabilities, and due to the EEPROM program memory, and the in-system programming features, there is no need for UV EPROM erasers or in-circuit emulation systems.

1.2.2 Prototyping Systems

When you perform your first experiments with the SX, it is most likely that you do not have a finished PCB on hand, designed for the system you intend to develop. Various prototype boards are offered by Parallax, Ubicom, and other vendors.

All the boards come with the basic components that are required to get the SX up and running, like a voltage regulator, a reset circuitry, a clock generator, etc. Additional components like LEDs, switches or pushbuttons, RS-232 drivers, serial EEPROMs, components for A/D conversion, and filters for PWM outputs can be found on most of the boards as well. Ubicom also offers boards designed to test a specific SX feature in detail, like communications via the I²C bus, or a demonstration board for TCP/IP applications (this is a WEB server on a 4.5 by 8.5 mm PCB).

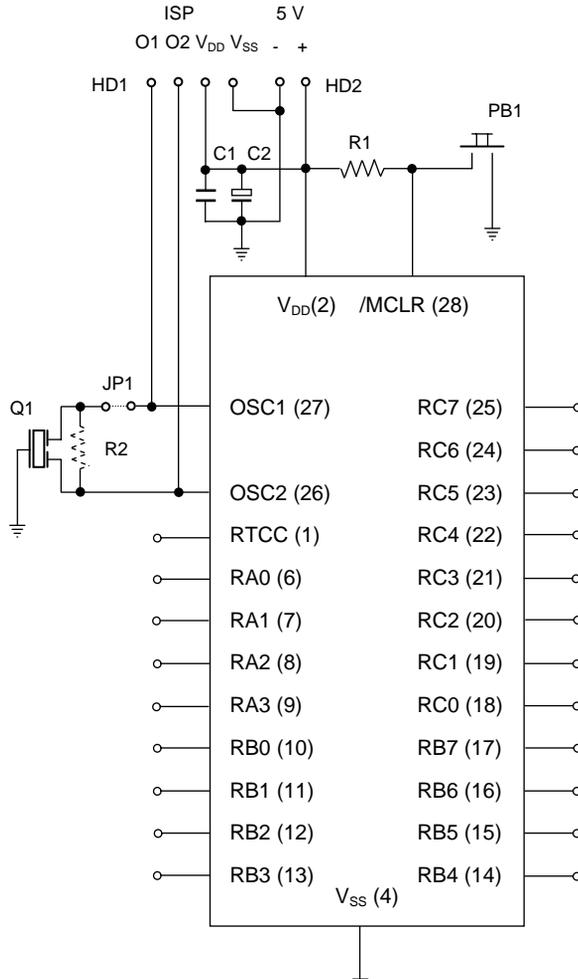
1.2.3 A "Home-brew" Prototyping System



Like all CMOS components, the SX can be damaged by excessive voltages produced by electro-static discharges. Therefore, take the usual safety measures that are required when handling static sensitive components. Also, make sure that the supply voltage does not exceed the maximum value specified in the SX datasheet (7.5 Volts).

Programming the SX Microcontroller

For the first experiments, you can build your own “homebrew” prototyping system as shown in the schematic below:



A simple SX Prototyping System

Bill of Material

Name	Dimension	Remark
R1	10 k Ω	
R2	Depends on the resonator, or crystal type	
C1	100 nF	Filter capacitors, use two or more if necessary.
C2	100 μ F Tantalum	
PB1	Pushbutton	Reset
Q1	50 MHz Ceramic resonator	Alternatively you may use a 50 MHz crystal
IC1	SX 28, DIP package	On a PCB, use a socket
JP1	Jumper	Open when the development system is connected to HD1
HD1	4-pin header connector, 1/10" spacing	Connector for development system
HD2	2-pin header connector	5V stabilized

In order to connect external components to the SX, you should provide header pins for the port lines, and for the RTCC input.

Take care that the leads connecting to OSC1 and OSC2 are as short as possible as the clock frequency may be up to 75 MHz (depending on the SX type used). It is also important to filter V_{DD} by placing capacitors as close as possible to the V_{DD} and V_{SS} pins of the SX.

1.3 SX-Key Quick Start using the Parallax SX-Key

1.3.1 The SX-Key

Parallax, Inc. has developed SX-Key® development system for the SX. The major component is a small PCB with a female 9-pin SUB-D connector to be connected to a serial COM port of a PC at one end, and with a 4-pole plug at the other end that connects to the 4-pin ISP header you will find on all prototype boards.



If you have built your own prototype board, double check that the header pins on your board are connected to the SX pins in an order that matches the one printed on the SX-Key plug, i.e. OSC1-OSC2-V_{DD}-V_{SS}. As this plug is not indexed, make sure that it is plugged in the right direction.

When you take a closer look at this small PCB, you will notice that it is crowded with various components, including an SX controller, a clock generator and a voltage converter that provides the programming voltage for the SX under test.

Together with the PC software which comes with the SX-Key you have a complete development system allowing you to write applications for the SX in Assembly language, program the SX, and test the application using the integrated debugger, or running the application “stand-alone”. All this is performed using the target SX on the prototype board, and not in a somehow simulated mode. This means that you can run an application in real-time speed but also in single steps for testing purposes. All this is controlled by the PC program via a serial connection.

1.3.2 Installing the SX-Key IDE Software

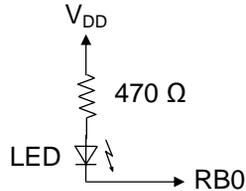
Together with the SX-Key programming tool, you should have received a CD-ROM with the SX-Key IDE software. You need a PC running a Windows-OS (Win 95 or greater). To install the software, run the setup program that is on the CD-ROM.

It makes sense to setup a shortcut to launch the SX-Key software from the desktop or from the Start menu.

Before taking the next steps, it is a good idea to visit Parallax's WEB Site (www.parallax.com) looking for newer versions of the SX-Key software. Parallax, Inc. usually offers new versions for download free of charge.

1.3.3 The First Program

For the first tests, we need an LED connected to port pin RB0 of the SX:



Some commercially available prototype boards do have an LED installed like this at the RB0 pin already. On some boards, the LED may be connected between RB0 and V_{SS} instead. For our first tests, this does not matter.

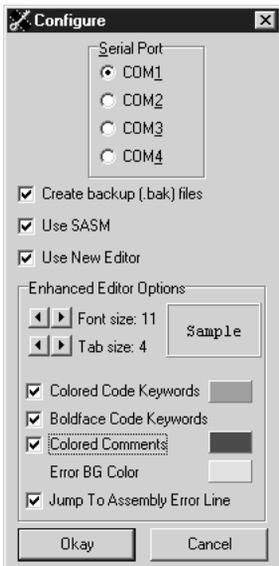
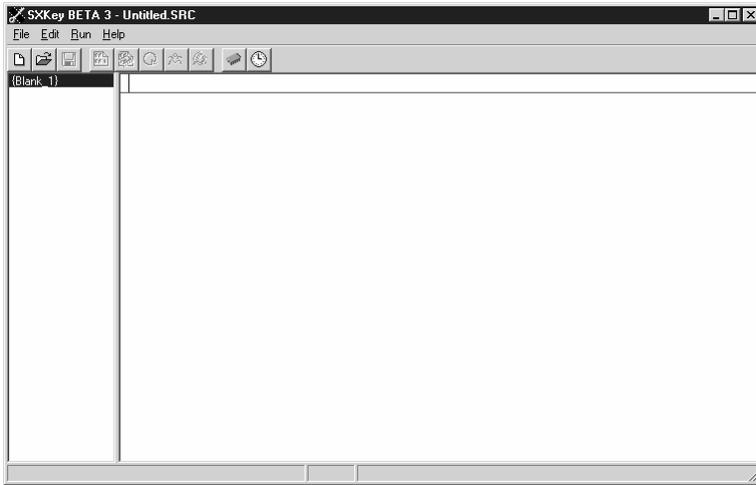
When you use a Parallax SX Tech board, you can easily position the two required components in the breadboarding area.

If not yet done, connect the SX-Key via a cable to a serial port of your PC and plug the other end on to the ISP header pins (double-check the correct orientation of the plug). Make sure that you use a "straight-through" type of serial cable, i.e. not a null-modem cable. If you need adapters to convert between 9-pin and 25-pin DB connectors, also make sure that the adapters are "straight-through". Finally, note whether the cable is connected to COM1, COM2, or any other COM port.

Make sure that the jumper that connects a resonator or crystal to the OSC1 pin on the prototype board is open. If there is no jumper, remove the resonator or crystal from the socket.

Now connect the power supply to the prototype board, and launch the SX-Key software on the PC. After the program has loaded you should see a window like this:

Programming the SX Microcontroller



This window shows the text editor of the new Parallax SX-Key IDE, Version 2. Compared to former versions, this new IDE has a lot of useful enhancements, like syntax highlighting, the possibility to open several files at the same time, and much more. You will use the editor to enter the application source code.

Select "Configure" from the "Run" menu, or press Ctrl-U as a shortcut to open the dialog box shown below:

Click a radio button in the "Serial Port" section to select the COM port you want to use for the SX-Key. For now, leave the other options in this dialog unchanged. See the Parallax documentation for the meaning of the other options. Finally, click "Okay" to close the dialog box.

Next, enter the following text in the editor window:

```

; =====
; Programming the SX Microcontroller
; TUT001.SRC
; =====
LIST Q = 37
DEVICE SX28L, TURBO, STACKX, OSCHS2
IRC_CAL IRC_FAST
FREQ 50_000_000
RESET 0

    mov    !rb, #%11111110
Loop
    clr    rb.0
    setb   rb.0
    jmp    Loop

```

It makes no difference if you type uppercase or lowercase letters. You may enter tabs or spaces to separate the words. The leading space we have inserted in some lines is not really required but it makes the text look a bit more "structured". You may also insert any number of empty lines at any place in order to structure the source code text.



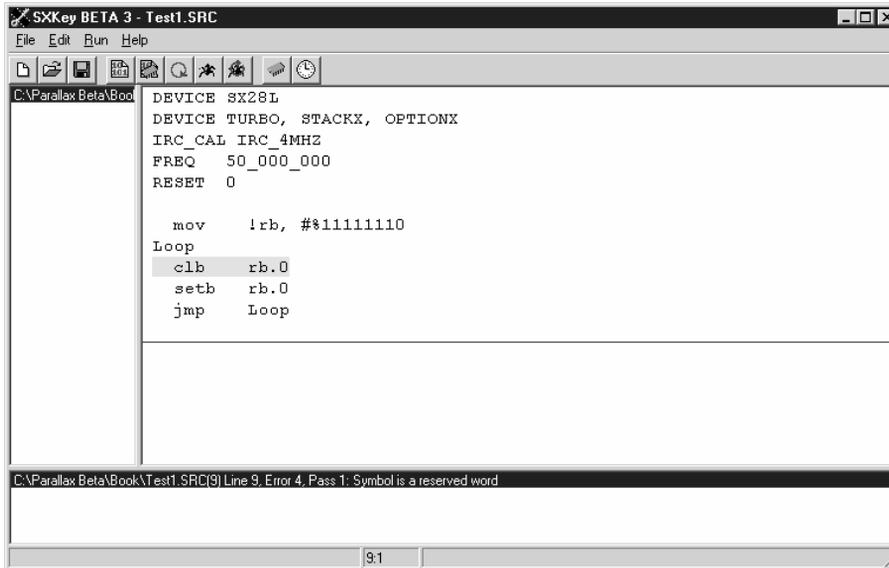
This and the following sample programs assume that you are using an SX 28 controller. In case you are using another SX type, replace the **DEVICE SX28L** directive with the respective **DEVICE** directive.

After you have entered your first program code, you need to save it. Either click on the diskette shortcut button, select "Save" from the "File" menu, or enter Ctrl-S on the keyboard to open the "Save Source as..." dialog. If you like, select or create a folder where the file shall be saved, and then enter the file name, e.g. TUT001 (the ".SRC" extension will be added automatically, so there is no need to type it in).

Next, click the Assemble button (the fourth button from the left), select "Assemble" from the "Run" menu or press Ctrl-A as a shortcut to assemble this little program. When a dialog opens, telling you that the file needs to be saved prior to assembling, click the "Ok" button. You may consider to mark the "Don't show this dialog again" option to avoid that it pops up whenever you assemble another program. When the status line at the bottom right of the editor window displays "Assembly Successful", the program could be assembled without errors, and it is most likely that you can execute it.

Should a dialog box pop up containing the message "Unable to assemble due to errors in source code", click the "Ok" button. The editor window should then look similar to the next figure.

Programming the SX Microcontroller



The screenshot shows a window titled "SXKey BETA 3 - Test1.SRC" with a menu bar (File, Edit, Run, Help) and a toolbar. The main text area contains the following assembly code:

```
DEVICE SX28L
DEVICE TURBO, STACKX, OPTIONX
IRC_CAL IRC_4MHZ
FREQ 50_000_000
RESET 0

    mov    lrb, #*11111110
Loop
    clb   rb.0
    setb  rb.0
    jmp   Loop
```

The line `clb rb.0` is highlighted. Below the code, a status bar displays the error message: "C:\Parallax Beta\Book\Test1.SRC(9) Line 9, Error 4, Pass 1: Symbol is a reserved word". The status bar also shows "9:1" on the left.

It is most likely that the error is caused by some misspelled text. The offending line is highlighted, and in the new window that has opened below the text, you will find a description of the error. In our example, the assembler error message reads, "Symbol is a reserved word". This may be a bit miss-leading, but assemblers don't have too much intelligence to detect each and every reason for an error. In our example, the word "clb" left of "rb" is misspelled, because it should read "clrb".

Make the necessary corrections, and press Ctrl-A again to assemble the modified program until the message "Assembly Successful" finally shows up in the status line.

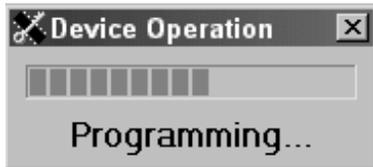
After having corrected any errors, you should again save your "masterpiece". Click the "Save" button (the button showing the diskette symbol), select "Save" from the "File" menu, or press the shortcut Ctrl-S.

When source code files become larger, it is a good idea to save them from time to time. Simply click the Save button or press the Ctrl-S shortcut to make sure that your work is not lost. Please note that the editor automatically generates a backup copy of the previously saved version of a saved file when the option "Create backup (.bak) files" is activated in the Configure dialog. This means that you will always have the current and the previous version of a program available on your disk drive. In order to archive certain versions of a source code file, use the "File - Save As" option to save the file under another name.

Now click the Debug button (the fourth button from the left with the bug symbol), select "Debug" from the "Run" menu, or type the Ctrl-D shortcut to launch the SX-Key debugger. As programs

are always executed on the target SX controller, they must be transferred to the SX before debugging can begin. Invoking the debugger means that the current version of the source code file is assembled, and then the resulting machine program is transferred to the SX (provided that there are no errors in the source code).

When you see a display like



you are very close to executing your first program.

Should an error message show up like this one:



click the "Abort" button, and find out what the problem is. There are several reasons for such message:

- No supply voltage connected to the prototype board, supply voltage too low, or too high.
- SX-Key not connected to the specified COM port, or wrong orientation of the ISP plug.
- The jumper between SX-Key pin OSC1 and the resonator is still in position.
- The orientation of the SX in the socket is wrong, is not plugged in at all, or a pin has been bent.
- The SX is defective.
- The SX-Key is defective.

Check the reasons in the given order. Hopefully, you will have found the reason before reaching the last two items in the list.

Programming the SX Microcontroller

After you have corrected the problem, press Ctrl-D again. Within a few seconds, the "Programming" message should come up. When the program has been transferred to the SX, the message box is closed, and the following windows will open:

The screenshot displays three windows from the SX Microcontroller programming software:

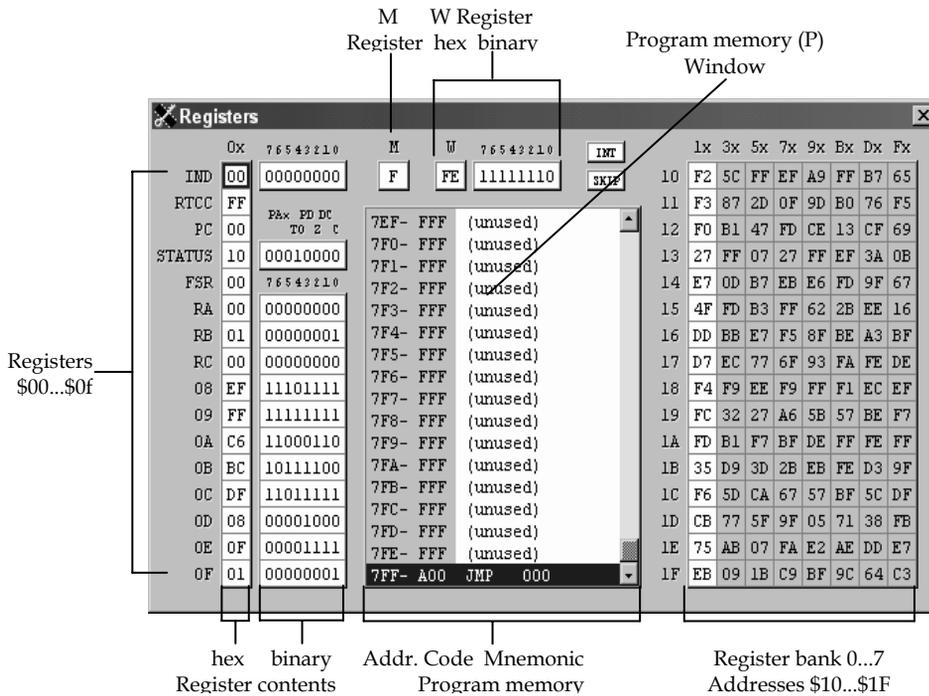
- Registers:** Shows the current state of the microcontroller's registers. The PC register is at 00, and the STATUS register is at 10. The program counter (PC) contains the value 11101111. The instruction at PC is 7FF-A00 JMP 000.
- Code - List File:** Shows the assembly code being executed. The code starts with device definitions and configuration constants, followed by a loop routine. The current instruction is at address 0000 0CFE: `mov 'rb, #11111110`.
- Debug:** Provides control over the execution process. It includes buttons for Idle, Step, Walk, Run, Poll, Stop, Reset, Registers, Code, Watch, Reset Pos., and Quit. The Update Speed is set to Max.

The size and the position of these windows depends on the screen resolution you have configured. You may move each of the windows, and you can also resize the "Code - List File" window if you like.

1.3.4 The SX-Key Debugger Windows

1.3.4.1 The Registers (R) Window

This window shows the SX "internals", i.e. its various registers. The figure below explains the different areas in that window:



In the middle of the "Registers" window, there is another area, that we will call "Program Memory" or "P" Window. Currently address \$7FF is highlighted in this window. (Note that the R window displays all values in hex or binary without leading "\$" or "%" signs.)

1.3.4.2 The Code – List File (C) Window

This window displays the assembly source code as you have entered it, plus some additional information. Actually, this is the so-called "List" file format showing the machine codes that were

Programming the SX Microcontroller

generated by the Assembler, together with the addresses where the machine codes are stored in program memory.

1.3.4.3 The Debug (D) Control Window

This window contains the buttons that are required to operate the debugger, and other buttons to open the debugger windows in case they have been closed or minimized. Click one of the buttons "Registers", "Code", or "Watch" to open the corresponding windows ("Watch" is inactive for now).

In the following text, we will use the short form "D", "R" or "C" to refer to one of the windows, and "P" for the program memory display in the center of the "R" window.

1.3.5 Executing the First Program in Single Steps

The highlight in the P window is positioned at program address \$7FF and in the C window the line containing **RESET** 0 is highlighted. After a reset, the SX controller sets the program counter, i.e. the register that contains the address of the instruction to be executed next to the address of the highest available program memory location. For the SX 28, this is \$7FF.

At this address, the assembler has inserted an instruction that makes the SX continue program execution at (i.e. jump to) the address defined by the **RESET** directive in our program (0 in our example).

The instruction which unconditionally causes the program execution to branch to another location is the **jmp** instruction. This instruction loads the program counter (PC) with the target address, i.e. the PC then "points" to the first program instruction to be executed.

By the way, you can use the scroll bar to the right of the P window to scroll the displayed text up and down. In this case, the highlighted line may be moved out of the window, but it keeps its position to mark the next instruction to be executed.

The same is true for the C window. Here you have horizontal and vertical scroll bars available to move the text.

Now left-click the "Step" button in the D window once. Alternatively, you may also enter Alt-S on the keyboard (make sure that the D window has focus in this case).

Now the window contents should change like this:

The screenshot shows two windows from a debugger. The top window is titled "Registers" and displays various CPU registers and their values. The bottom window is titled "Code - List File" and shows a list of assembly instructions with their addresses and hex values.

Registers Window:

Register	Value	Address	Instruction	Comment
IND	00	00000000		
RTCC	FF			
PC	01			
STATUS	00	00000000		
FSR	00	76543210		
RA	0E	00001110	000- CFE	MOV W, #FE
RB	0F	00001111	001- 006	MOV !RB, W
RC	00	00000000	002- 406	CLRB RB.0
08	EE	11101110	003- 506	SETB RB.0
09	C7	11000111	004- A02	JMP 002
0A	1C	00011100	005- FFF	(unused)
0B	10	00010000	006- FFF	(unused)
0C	FF	11111111	007- FFF	(unused)
0D	09	00001001	008- FFF	(unused)
0E	0F	00001111	009- FFF	(unused)
0F	01	00000001	00A- FFF	(unused)
			00B- FFF	(unused)
			00C- FFF	(unused)
			00D- FFF	(unused)
			00E- FFF	(unused)
			00F- FFF	(unused)
			010- FFF	(unused)

Code - List File Window:

```

1 07FB 0F7F    DEVICE $X28L
2 07FB 0F7F    DEVICE TURBO, STACKX, OPTIONX
3 =00000001    IRC_CAL IRC_4MHZ
4 =02FAF080    FREQ 50_000_000
5 =00000A00    RESET 0
6
7 0000 0CFE    mov    !rb, #%11111110
   0001 0006
8 =00000002    Loop
9 0002 0406    clr    rb.0
10 0003 0506    setb  rb.0
11 0004 0A02    jmp   Loop

```

The highlight has moved to address \$000, i.e. the SX has executed the **jmp 0** instruction that is located at address \$7ff which caused the jump to the new program address at \$000. As you can see, the highlight in the (C) window is now positioned on that line.

1.3.6 Compound Instructions

The P window displays the **MOV W, #FE** instruction where the C window has highlighted the **mov !rb, #%11111110** instruction, i.e. a different instruction.

The point here is that the SX does not "know" how to execute a **MOV W, #FE** instruction. The Assembler automatically has generated two separate instructions that are "familiar" to the SX:

```

MOV W, #FE
MOV !RB, W

```

Programming the SX Microcontroller

These two instructions were saved in two subsequent locations of the program memory. We will call such assembler instructions *compound instructions*. There are various compound instructions available to make programmer's life a bit easier because it saves you the extra work of writing two or more separate lines of code. (Later, we will see that compound statements can also cause situations to make programmer's lives quite hard.)

Now let's find out what the **mov !rb, #%11111110** instruction means. A **mov** instruction copies the contents of one register into another register or it copies a constant value into a register. "mov" is derived from "move", but it actually copies a value instead of moving it, i.e. the contents of the source remains unchanged after execution, where the target receives the new value.

The hash mark "#" left of the binary number %11111110 means that the constant value %11111110 shall be copied into a target called !rb here. The hash-sign is very important - if you leave it out, the assembler will assume that you want to copy the contents of register %11111110 to !rb instead!

"!rb" specifies the SX configuration register for port B. We will discuss port configuration in more detail later. For now, you should keep in mind that a cleared bit in the configuration register means that the associated port pin will become an output. To make clear which bits are set and cleared in the !rb register, we use binary notation here.

Here, we configure the RB.0 pin as an output - this is where we have connected the LED.

Actually, the **mov !rb, #%11111110** is composed by the two instructions

```
mov w, #%11111110  
mov !rb, w
```

i.e. the constant value %11111110 is copied into the w register, and then the contents of w is copied into !rb. The w register (the "Working" register) is used as temporary storage by many compound instructions. In general, w is a multi-purpose register used to hold one operand of arithmetic or logical instructions, to hold the result of special **mov** instructions with arithmetic/logical functions and as temporary storage like in the example above. The w register is similar to the "accumulator" found in other microcontrollers or microprocessors.

Now left-click the "Step" button once again, and you will notice that the highlight in the P window has moved to the second part of the compound instruction where the highlight in the C window did not move at all.

Click "Step" again to execute the **mov !rb, w** instruction, and to bring the highlight on to the **clr rb. 0** instruction. Click "Step" to let the SX execute this instruction as well, and check if the LED turns on.

If you have a prototype board where the LED is connected to V_{SS} with the other end (the cathode), click "Step" once more to execute the **setb rb. 0** instruction that should finally turn on the LED in this case.

If you managed to turn the LED on, you are all set - your first SX program works as expected!

In case the LED remains off, check the following:

- Is the constant that is moved into `!rb` actually `%11111110` (did you eventually forget the leading `"#"` or `"%"` characters)?
- Do all instructions address the right port (`rb`)?
- Do the `clrb rb.0` and `setb rb.0` instructions both contain the `"0"`, and not another digit?
- Is the LED really connected to pin 10 (RB0) of the SX 28?
- Is the polarity of the LED correct?

In case you have found an error in the program code, click the "Quit" button to return to the editor, make the necessary corrections, and enter Ctrl-D to launch the debugger again (the program will be assembled and transferred to the SX automatically).

If the problem was caused by hardware, you have (hopefully) disconnected power from the SX. This has caused the SX-Key software to display the error message "SX-Key not found on COMx". Click the "Abort" button to return to the Editor window and re-connect the power supply then.

If the program did work properly, instead of disconnecting the power supply, click the "Quit" button to leave the debugger back to the Edit window.

When you want to re-activate the debugger again later, it is not necessary to transfer the program to the SX again as long as you did not make changes in the source code. In this case, don't select "Run - Debug", and don't press the Ctrl-D shortcut. Instead, select "Run - Debug (reenter)", or press the shortcut Ctrl-Alt-D. This starts up the debugger immediately without transferring the program into the SX.

When the debugger is active again, address `$7ff` is highlighted as it was when you started the debugger the first time, and you may continue the debugging session.

Now let's find out what makes the LED turn on or off:

The `clrb rb.0` instruction clears a bit in the specified register (`rb` in this case), where `rb` is a pre-defined name the assembler "knows". The assembler replaces it with `$06`, the address of the Port B data register. Instead of `"rb"`, you could have written `"$06"` as well. Which bit shall be cleared is specified by the digit that follows the register name, separated by a period.

In our case, we clear bit 0 in the Port B data register. As the associated port pin has been configured as an output, this means that the output pin goes to low level and the LED is turned on.

Programming the SX Microcontroller

The next instruction **setb rb. 0** does just the opposite of **clrb** - it sets a bit in the specified register. In our case, it causes the output pin RB0 go to high level that turns the LED off again.

In the left part of the R Window, the contents of the first 16 registers are displayed in hexadecimal and binary format. When you watch the contents of address \$06 you will notice how the content changes as you keep clicking the "Step" button. Note that the hexadecimal value is displayed with a red background when it has recently changed, and that bit 0 in the binary display area is also shown with a red background when its state has changed. This helps you to quickly find out which data has changed after executing an instruction.

Notice that the contents of PC is always displayed with a red background as the program counter always changes its contents, either to address the next instruction in sequence, or another location after a **jmp**, **skip** or **call** instruction.

1.3.7 Symbolic Addresses – Labels

The last instruction in our program is a **jmp** instruction that sets PC back to address \$002 where the **clrb** instruction is stored. If you look at the P window, you notice that it shows **jmp 002**, but in our program, we have written **jmp Loop**.

We could also have written **jmp \$002** instead, but this would not be very flexible. Imagine what happens if we would insert another instruction immediately following the **mov !rb, #%11111110** instruction. This would "shift" all subsequent instructions "up" in program memory, and to reach the **clrb rb. 0** instruction, you would have to change the \$002 address parameter of the **jmp** instruction. Think what a "nice job" it would be to correct all the **jmp** instructions in a program consisting of hundreds of lines, and what could happen if you would forget to correct even one of them.

Fortunately, the assembler allows the definition of symbolic addresses that makes life a lot easier. When the assembler finds a word at the beginning of a line that is neither an instruction nor another "reserved word" (more about this later), it interprets it as a "label". In this case, it stores the word (**Loop** in our example) in an internal table (the symbol table) together with the address of the instruction that follows the label, either in the same line, or in the next line that is not empty, and does not contain a comment only (**\$002** here).

Whenever the assembler finds an instruction that is not followed by a numeric value, as in **jmp Loop**, it searches the symbol table for that word and replaces it with the numeric value that is stored there (**\$002** in our example).



SASM, the default assembler of the SX-Key 2.0 software expects that labels always begin in the leftmost column of a line, where other assemblers like the "old" SX-Key assembler accept leading white space. For compatibility reasons, it is a good idea to always let labels begin in the first column.

1.3.8 Running the Program at Full Speed

If you are tired of clicking the "Step" button, you might consider letting the program run at full speed. Click the "Run" button, and see what happens: The LED "glows" rather dim and not at about half of the full intensity as you might have expected.

There are two reasons for this phenomenon: Duty cycle and speed.

Here are the instructions that are continuously executed while the SX runs at full speed, together with the required clock cycles:

```

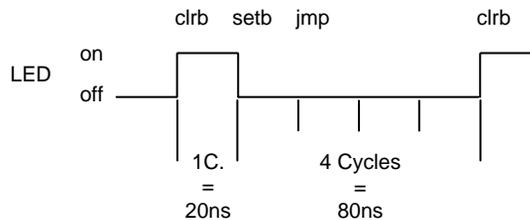
Loop
  clrb  rb.0    ; 1
  setb  rb.0    ; 1
  jmp   Loop    ; 3

```



You may add comments (like the number of clock cycles in the lines above) using the semicolon. The assembler will ignore all text in a line that follows a semicolon. If a line begins with a semicolon, all the rest of the line will be ignored. This is sometimes helpful to temporarily "comment out" instructions in a program.

The **clrb** and **setb** instructions take one clock cycle each, and the **jmp** requires three cycles. The diagram below shows the LED timing (assuming that you run the SX at 50 MHz clock):



After the **clrb** instruction, the LED is turned on. It takes one clock cycle (20 ns at 50 MHz system clock) until the **setb** instruction is executed, i.e. until the LED is turned off again. Then it takes three cycles to execute the **jmp** and another cycle for **clrb** until the LED is turned on again. This means that during 20% of one loop the LED is on, and 80% of the loop, the LED is off.

To extend the LED's on time, we need to add some "cycle eater" instructions between the **clrb** and **setb** instructions that "steal" three clock cycles. The SX "knows" a special "do nothing" instruction that exactly does this, the **nop** (for no operation).

Quit the debugger, and add three **nop** instructions like this:

```

Loop

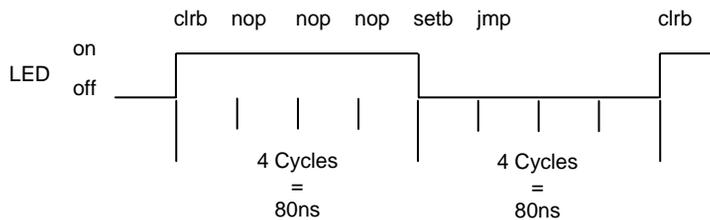
```

Programming the SX Microcontroller

```
clrb    rb.0    ; 1
nop     ; 1
nop     ; 1
nop     ; 1
setb    rb.0    ; 1
jmp     Loop    ; 3
```

As you did change the source code, you can't restart the debugger now, therefore press Ctrl-D to re-assemble the modified program, and to have it transferred into the SX. Then start the program at full speed by clicking the "Run" button.

The changes result in the following timing:



Now the LED is on and off for an equal time, i.e. it has a duty cycle of 50%.

One full loop now has a period of 8 clock cycles or 160 ns, and this means a frequency of 6.25 MHz! This is a frequency LEDs are not designed for, and this is the second reason why the LED may be darker than expected.

Instead of having the LED "blink" at this frequency, we want to make it blink slowly enough so that we really can see it blink. Before we enhance the program, try the following:

First, stop the full-speed execution by either clicking the "Stop" or "Reset" button. Now click the "Walk" button, and you will see the LED blink.

The "Walk" mode is similar to single stepping except that the debugger "clicks" the "Step" button for you a couple of times per second. As you may notice, the LED does not really blink, instead it "flickers". This is because the debugger needs some time to update the window contents between each step.

Click "Reset" or "Stop" to end the "Walk" mode. When you click "Reset", the SX is really reset, i.e. the PC is reset to \$7ff (and some other registers are initialized to specific values as well). When you click "Stop", the execution stops at the instruction which was executed when you clicked that button, and all registers reflect the status at that time, and you may continue program execution from that point.

1.3.9 Program Loops for Time Delays

Now we will slow down the SX in order to obtain a nicely blinking LED while the program is running at full speed.

In the last version of our program, we used three nop instructions to "eat up" time. In order to "waste" more time we will add some more program loops.

Don't enter the following statements, as we only need them for some time calculations:

```

Loop
  decsz $08           ; 1/2
  jmp Loop           ; 3
  clrb rb. 0         ; 1
Loop1
  decsz $08           ; 1/2
  jmp Loop1          ; 3
  setb rb. 0         ; 1
  jmp Loop           ; 3

```

In this code, we have added two more program loops, one following the **clrb** instruction, and one following the **setb**.

Within the loops, we use the **decsz** (decrement and skip on zero) instructions. This instruction decrements the contents of the specified register (at address \$08 in data memory here) by one. In case the content of the register ends up in zero after the decrement, the next instruction will be skipped (the **jmp** in our example).

Let's assume that the register at \$08 contains zero when we start the program. In this case, the first **decsz** instruction would change its contents to \$ff or 255. Because its content is not zero, the next instruction will not be skipped, i.e. the **jmp** instruction will be executed.



You may wonder why 0 - 1 results in 255 in the SX, and not in -1, as you might guess. This is because the SX (like most other controllers) does not "know" about negative numbers.

To understand the "underflow" from 0 to 255, let's see what happens when a value of 255 (or %11111111) is incremented by one. The "real" result would be %100000000, i.e. the 9th bit would be set, and all other bits cleared. As the registers in the SX can only hold eight bits, the exceeding 9th bit is lost. This means that 255 + 1 yields in 0 in the SX.

Decrementing a value of 0 by one is the reverse of incrementing a value of 255 by one, and this is why 0 - 1 yields in 255 in the SX.

This sequence is repeated until the content of \$08 finally reaches zero. Now the **jmp** will be skipped, and the **clrb** or **setb** instructions are executed.

Programming the SX Microcontroller

Again, we have added the number of clock cycles that each instruction requires. Note that the **dec sz** instructions usually take one cycle (when there is no skip), but two in case of a skip.

As a data register can hold 256 different values (0...255), each of these loops is executed 256 times where 255 times, the skip is not performed. Therefore, each loop takes $255 * (1+3) + 2 = 1,022$ clock cycles and one more cycle to clear or set the port bit. By adding the three more cycles for the final **jmp Loop** instruction, we end up in $1,023 * 2 + 3 = 2,047$ cycles that take $2,047 * 20 \text{ ns} = 40.94 \mu\text{s}$ which results in an LED blink frequency of 24.426 kHz - far beyond visibility!

Even if we would nest another delay loop within each of the two loops, the SX would still be too fast. Before considering to reduce the system clock rate, try this program:

```

; =====
; Programming the SX Microcontroller
; TUT002.SRC
; =====
include "Setup28.inc"

RESET 0
  mov  !rb, #%11111110
Loop
  decsz $08      ; 1/2
  jmp  Loop      ; 3
  decsz $09      ; 1/2
  jmp  Loop      ; 3
  decsz $0a      ; 1/2
  jmp  Loop      ; 3
  clrb rb.0     ; 1
Loop1
  decsz $08      ; 1/2
  jmp  Loop1     ; 3
  decsz $09      ; 1/2
  jmp  Loop1     ; 3
  decsz $0a      ; 1/2
  jmp  Loop1     ; 3
  setb rb.0     ; 1
  jmp  Loop      ; 3
```

Don't try to assemble, or run the program yet. Note the **include "Setup28.inc"** directive at the beginning of the code. The new SX-Key 2.0 software now allows to include one or more files within the code. This means that the assembler will open the file that is specified with the **include** directive, read its contents, and inserts the lines read at the location of the include directive. This feature is handy to add lines that are the same for many different programs.

Please create a new file in the SX-Key Editor environment, enter the following lines, and save it under the name "Setup28.inc" in the same directory where you have saved TUT002.SRC:

```

; =====
; Programming the SX Microcontroller
; Setup28.inc
; =====
LIST Q = 37
DEVICE SX28L, TURBO, STACKX, OSCHS2
IRC_CAL IRC_FAST
FREQ 50_000_000

```

As we need to make some definitions concerning the configuration of the SX chip in each and every program, these configuration lines are good candidates for an include file (in the next chapter, chip configuration is explained in more detail).

We will use Setup28.inc together with many other code samples in this tutorial section of the book. Should you use another SX chip, e.g. an SX20, you only need to change the DEVICE specification in Setup28.inc, and all programs using this include file will be automatically configured for an SX20 (when you assemble them with this modified include file).

After having saved the file, assemble the second tutorial program TUT002 that you have entered before.

In this program, we have nested three program loops before executing the **clrb** or **setb** instructions and we use three registers as delay counters (\$08, \$09, and \$0a).

In each loop, we first decrement \$08 until it is zero and then decrement \$09. If the content of \$09 has not yet reached zero, we repeat the 256 loops decrementing \$08 until \$09 is zero. Then we decrement \$0a, and repeat the previous steps until finally \$0a is zero.

Now let's figure the approximate time the three nested loops take:

As the "inner" loop is identical to the one in the previous code example, it takes 1,022 cycles to execute it. The "middle" and the "outer" loop are executed 256 times as well, therefore the total number of clock cycles required by the three nested loops is approximately $256 * 256 * 1.022 = 67 * 10^6$ cycles. For a complete LED on-off cycle, the total time delay is $2 * 67 * 10^6 * 20\text{ns} \approx 2.6$ seconds, i.e. the resulting LED blink frequency is about 0.38 Hz.

After you have entered this new version in the editor window, don't forget to save it under a new file name (e.g. TUT002.src), and then press Ctrl-D to launch the debugger.

Click the "Run" button to execute the program. Provided that you have correctly entered the program, the LED should now blink quite slowly.

While the program is running at full speed, the R, P, and C windows are not updated because this would slow down execution far beyond real-time. If you want to take a "snapshot" of the

Programming the SX Microcontroller

current register status, click the "Poll" button at any time while the program is executed at full speed.

When you want to execute this program in single steps, keep in mind that one nested delay loop now takes about 67 Million steps. Maybe that clicking the "Step" button 67 Million times is a good test for your left mouse button, but we don't take any responsibility for your hurting fingers.

Even the "Walk" mode takes far too long to execute one LED on-off cycle.

As such kind of loops can be found frequently in SX programs, there should be a way to "skip over" such loops and start single-stepping from there. Fortunately, the SX debugger allows setting a "breakpoint" that solves this problem.

1.3.10 Setting a Breakpoint

Setting a breakpoint means that you "tell" the debugger to execute the instructions beginning at the address, PC is currently pointing to, up to and including the instruction where you have set the breakpoint.

To set a breakpoint, first make sure that the program is halted by either clicking the "Stop" or "Reset" buttons. Then, in the C window, move the mouse pointer to the program line where you want to set the breakpoint and hit the left mouse button once. The debugger will display this line with a red background now, indicating that a breakpoint is active on that line. If necessary, scroll the text in the window up or down until the line you want is visible before setting the breakpoint.

In case the line with the breakpoint is the next line to be executed as well, only the left part of the line is marked with a red background while the rest of the line is highlighted with a blue background.

For example, click "Reset" for a "clean start", and then click on the line with the **cl rb rb.0** instruction. Finally, start the program at full speed with "Run".

You will notice that it takes a while until the LED is turned on. Once the LED is on, program execution halts due to the active breakpoint, and the **decsz \$08** instruction in **Loop1** is the next one to be executed.

If you like, you may single-step the program for a while but you can also click "Run" again to go through the program at full speed until the breakpoint is reached the next time. During that time, you will notice that the LED is turned off after a while, and finally is turned on again, when the program "hits" the breakpoint after executing the **cl rb** instruction.

Please note that there can only be one breakpoint active at a time. This is not a limitation of the SX-Key software but by the SX itself. As soon as you click another line in the C window, this new line will be highlighted in red, and the line marked before is reset to standard.

In order to remove an active breakpoint, simply click on the highlighted line once again.

Please note that due to the internal structure of the SX the instruction in the line marked for a breakpoint will be executed before the program actually halts. This may be confusing sometimes, when you set a breakpoint on a line with a jump or call instruction as you will see later. In addition, the SX does not stop program execution when the breakpoint is set to a line containing a **nop** instruction.

You may add a **BREAK** directive to the source code immediately before the instruction where the debugger shall activate a “default” breakpoint when it is invoked. You can change the position of the **BREAK** directive in the source code later without the need to re-assemble the code, i.e. you may re-start the debugger using the Debug (reenter) option, or by entering the Ctrl-Alt-D shortcut.

When the debugger is active, you may change the position of the breakpoint at any time by clicking on another line in the list window that contains executable code.

1.3.11 Where to Go Next

This ends the Quick-Start chapter for the SX key development system. In this chapter, you have learned some basic SX instructions, and programming techniques but most of the chapter was dedicated to the SX-Key development system.

In the next tutorial chapters, we will concentrate on more SX instructions and features, assuming that you are familiar with the development tool, you are using: The SX-Key system.



When you are done with debugging a program, you may want to have the SX execute the program “stand-alone”, i.e. without the SX-Key probe connected. In order to do so, it is important that you re-program the SX without the debug code. Select “Program” from the “Run” menu, click the “Program” shortcut button, or enter Ctrl-P. This instructs the SX-Key to transfer the “stand-alone” code into the SX program memory. It is also necessary that the SX has another clock source now. You can find more information about the various clock sources in chapter 1.14.

1.4 SX Configuration - ORG/DS - Conditional Branches

1.4.1 Configuration Directives

For the sample program shown before (and many other programs in the tutorial section of this book), we have created an include file "Setup28.inc" containing the following lines:

```
LIST          Q = 37
DEVICE       SX28L, TURBO, STACKX, OSCHS2
IRC_CAL     IRC_FAST
FREQ        50_000_000
```

Such statements are called "Assembler Directives". They do not cause the assembler to generate program code. Instead, they instruct the development system how to configure the SX when it transfers a program to the device. For configuration purposes, the SX has special registers called "Fuse Registers".

The **DEVICE** directives make it easy to define the settings of the fuse register bits because you need not to remember which fuse bit is used to set a specific option like the turbo mode or an extended stack and option register.

(2.2.7.1 - 234) The parameters following a **DEVICE** directive each define a specific configuration. For example:

SX28L informs the assembler that the generated program shall be transferred into an SX28 device.

TURBO activates the turbo mode, i.e. each standard instruction will be executed in one clock cycle.

STACKX or **OPTIONX** activate an 8-level return stack for subroutines and an extended option register.

OSCHS2 defines the drive mode for an external crystal, or ceramic resonator.

The **LIST Q = 37** directive instructs the Assembler to suppress certain warnings.

The **IRC_CAL IRC_FAST** directive defines that the internal RC oscillator shall be set to the fastest mode which actually disables the oscillator calibration performed by the SX-Key in order to reduce the chip programming time.

The **FREQ** directive defines the clock frequency that shall be generated by the SX-Key hardware (50 MHz in this case).





You may wonder why the SX offers the “non-turbo” mode, where each standard instruction is executed in four clock cycles, the smaller 2-level return stack, and the reduced option register functionality. This is due to compatibility reasons to some similar microcontroller devices manufactured by other vendors. In “real life”, you will never use this reduced functionality together with an SX. Why would you drive a Porsche in the to lowest gears only?

As you can see, a **DEVICE** directive can be followed by more than one parameter. Use commas to separate each of the **DEVICE** parameters.

The **IRC_CAL** and **FREQU** directives are internal for the SX-Key debugger. They specify the calibration of the SX-internal clock oscillator and the clock frequency in Hz, the debugger shall generate when the SX runs at full speed. Note that you may insert underscores for better readability of the frequency parameter and in any other numerical value.

Each program also requires a **RESET** directive together with an address (or a symbolic address) to inform the Assembler where the main program execution starts. The assembler generates a **jmp** instruction to that address in the highest location of the program memory. We did not place the **RESET** directive in the Setup28.inc include file because the reset address or the symbolic name of that address may change in different programs.

From now on, we will start each program with a directive like this:

```
include "Setup28.inc"
```

assuming that Setup28.inc contains the directives as shown above.

When you are using a development system other than the SX-Key, make sure that the SX is clocked at 50 MHz because many of the sample programs assume this clock speed when generating timing delays, interrupts, etc. In this case, it may be also be necessary to modify the definitions in the include file, or to insert the device configuration lines in each source code file when an **include** directive is not supported.

1.4.2 The ORG and DS Directives

In the previous program, we have used three nested loops to "slow down" the SX in order to see the LED blink. To build a loop that is executed a certain number of times, you need a loop counter that is incremented or decremented (as in our example) until a specific value has been reached (0 in our case). We have used registers at address \$08, \$09 and \$0a in the data memory as loop counters in this example.

Let's re-write this example and make it a bit more "generic":

Programming the SX Microcontroller

```
; =====  
; Programming the SX Microcontroller  
; TUT003. SRC  
; =====  
include "Setup28.inc"  
  
RESET    Main  
  
    org $08  
Counter1 ds 1  
Counter2 ds 1  
Counter3 ds 1  
  
    org $100  
Main  
    mov    !rb, #%11111110  
Loop  
    decsz Counter1  
    jmp Loop  
    decsz Counter2  
    jmp Loop  
    decsz Counter3  
    jmp Loop  
    clrb  rb.0  
  
Loop1  
    decsz Counter1  
    jmp Loop1  
    decsz Counter2  
    jmp Loop1  
    decsz Counter3  
    jmp Loop1  
    setb  rb.0  
  
    jmp Loop
```

Following the **RESET** directive, you can see the **org \$08** ("originate") directive. This informs the assembler that definitions following this directive shall begin at address \$08 (in data memory in this case).

In the next line, you find the statement **Counter1 ds 1**. As mentioned before, the assembler interprets a word that has no pre-defined meaning - like "Counter1" in this case - as a label and adds the word together with a numeric value that represents its address to the symbol table. The address of the label **Counter1** is \$08 because we have instructed the assembler to continue counting from that address on with the previous **org \$08** directive.

The **ds** following the label name means "define space", and the **1** following **ds** instructs the assembler to "set aside" one byte for **Counter1**. This also increments the assembler's internal address-counter by one. Therefore, the next label, **Counter2** will be located at address \$09. As **ds 1** also reserves one byte for **Counter2**, **Counter3** is located at address \$0a.

You may think of the three reserved bytes as three variables, each having a size of one byte, named **Counter1**, **Counter2** and **Counter3**.

Within the loops, you will notice that the **decsz** instructions no longer refer to "hard-coded" addresses in data memory, but they use the variable names instead.

Using symbolic addresses or names for variables makes a program much more readable, and it is a lot easier to modify it later.

Note that there is another **org** directive in the program: **org \$100**. As before, it instructs the assembler to set its internal address pointer to the specified value (\$100 in our example).

The next instruction (the **mov w, #%11111110**, i.e. the first instruction of the compound **mov !rb, #%11111110** instruction) will be coded into that address. This means that our program no longer begins at address \$000 in program memory but at address \$100 instead.

As we have placed the label **Main** in front of the first instruction, we can use that label to specify this address together with the **RESET** directive. You may use any other name but **Main** for the main program entry point. The SX-Key debugger has a special function that allows you to click one button to jump to the line labeled "Main". Therefore, it is a good idea to use this name for the main program entry point.

Note that one of the **org** directives specifies an address in data memory, where the other one specifies an address in program memory. The assembler "sorts that out" automatically. Also note that in case of the label **Main**, we have a "forward declaration", i.e. the label is referred to in the source code before it is actually defined. Again, the assembler takes care of this (it actually does an extra run through the source code "collecting" all the label definitions before generating the instruction code).

1.4.3 Conditional Branches

In the sample programs before, we have already used the **decsz** instruction to build the delay loops. Let's discuss such kinds of instructions in more detail now.

Without the capability to change the flow of program execution depending on certain conditions, a microcontroller would be rather useless as it could only execute "straight through" types of programs. Therefore, the SX comes with a set of conditional skip instructions, like the **decsz** instruction.

```
decsz Counter1  
  jmp Loop
```

for example, decrements the **Counter1**, i.e. it subtracts one from the former content, and the result becomes the new content of **Counter1**. In case that the result yields in zero, the instruction immediately following the **decsz** is skipped (the **jmp Loop** in our example).

Programming the SX Microcontroller

If you know microprocessors or other controllers, you may wonder why the SX does not "know" basic instructions like **jz** or **jnz** (Jump if Zero or Jump if Not Zero), but compound instructions only. This is because each instruction code always consists of just one word, 12 bits wide, but 12 bits are not enough to hold a register address as well as a jump address.

Therefore, keep in mind that the instruction following a conditional skip is not executed when the condition is true.

You may wonder why we have indented the **jmp** instructions following the **decsz** instructions. The first reason is to make the program more readable, and the indentation indicates that this instruction is not always executed. The second reason is much more important.

Look at the following code snippet:

```
decsz Counter1  
  mov !rb, #%11111110
```

On the first glance, this sequence seems to be fine, but it contains a "ticking bomb"! Remember that the **mov !rb, #%11111110** is a compound instruction as the SX does not provide a basic instruction that can copy a constant value directly into a port configuration register. The assembler translates the instructions like this:

```
decsz Counter1  
  mov w, #%11111110  
  mov !rb, w
```

You can now immediately see what the problem is: The **decsz** instruction does not "know" about compound instructions. All it does is adding an extra one to the PC register in order to skip the next instruction in case the condition is true. In our example, the **mov w, #%11111110** instruction will be skipped but not the **mov !rb, w** instruction! The result is that **!rb** will receive a random value, depending on the current content of the **w** register, and this is definitely not what you want.



Therefore, the instruction that immediately follows a skip *must never* be a compound instruction; otherwise, strange results may occur!

There are two ways to avoid that dangerous situation: Not using compound statements at all, or paying special attention to it.

Not using compound statements at all is possible because you can always write the basic instructions that make a compound instruction, but this means additional typing work, and increases the size of the source code.

Indenting the instruction following a skip makes it easier to double-check for not using compound instructions at such places.

1.5 Subroutines - Symbols - Data Memory

1.5.1 Subroutines

If you look at the previous LED-Blinker program, you will notice that the nested delay loops are duplicated, i.e. one is executed before turning the LED on, and the other is executed before the LED is turned off again.

Subroutines can help to avoid such duplicates, as shown in the following program version:

```

; =====
; Programming the SX Microcontroller
; TUT004.SRC
; =====
include "Setup28.inc"

RESET    Main

org $08
Counter1 ds 1
Counter2 ds 1
Counter3 ds 1

org $000
TimeEater
Loop1
    decsz Counter1
    jmp Loop1
    mov Counter1, #50
    decsz Counter2
    jmp Loop1
    decsz Counter3
    jmp Loop1
    ret

    org $100
Main
    mov    !rb, #%11111110
Loop
    call TimeEater
    clrb  rb.0
    call TimeEater
    setb  rb.0
    jmp   Loop

```

Here we have moved the delay loops to a subroutine called **TimeEater**. A subroutine is a sequence of instructions terminated with a **ret** (Return) instruction.

To execute the instructions in a subroutine, you use the **call** instruction together with the address where the first instruction of the subroutine is located. In our example, we have defined the

Programming the SX Microcontroller

label **TimeEater** as a symbolic address for the subroutine entry, and therefore we use that label together with the **call** instructions.

Similar to a **jmp** instruction, a **call** unconditionally causes a branch to the specified address, i.e. the content of PC is changed accordingly.

As soon as the **ret** instruction is reached, the content of PC is restored to point to the address of the instruction immediately following the **call** which caused the branch.

As you can see, there are two call instructions in this program, both invoking the **TimeEater** subroutine.

Within **TimeEater** you find the nested delay loops with the extension that **Counter1** is now initialized to 50 when it underflows. This makes the delay a bit shorter in order to increase the LED's blink frequency.

Instead of duplicating the delay loops twice in the program, we have now just one set of delay loops in the subroutine, and we call the subroutine twice instead.

In this example, this only saves a few words in program memory but you can imagine that subroutines help to save a remarkable amount of program memory space.

Besides this, subroutines also help structuring a program. Think of subroutines as "black boxes" that perform a specific task. So the calling program needs not to take care of the details, it just calls the subroutines, expecting that the subroutines do their job properly.

1.5.2 The Stack



(2.2.6 - 229) Previously, we had mentioned that a **ret** instruction terminates a subroutine, and that it restores the PC register to point at the instruction following the call. This means that the content of PC+1 as return address must be saved somewhere before loading it with the entry-address of the subroutine.

If there were just one fixed register to save the return address, it would not be possible to call another subroutine from within a subroutine, although this is common programming practice.

The second call of a subroutine would overwrite the previously saved value, i.e. the return address for the first-level subroutine call would get lost.

Therefore, a memory structure called "Stack" is used so save a certain number of return addresses to allow for nested subroutine calls. A stack structure is also called LIFO (Last In First Out) as this describes the way, data can be stored and retrieved.

You can think of a LIFO as a stack of dishes. If you put a new dish on the stack, it is a good idea to later remove this dish first in order to avoid a disaster.

The SX has a stack that can hold up to eight return addresses (in "compatibility mode", the SX18/20/28 stack can only hold two addresses). This means that the maximum nesting depth for subroutines is eight. If you like, single-step through the following program to see what happens when this depth is exceeded:

```
; =====  
; Programming the SX Microcontroller  
; TUT005. SRC  
; =====  
include "Setup28.inc"  
  
RESET   Mai n  
  
org     $000  
sr1     call sr2  
        ret  
sr2     call sr3  
        ret  
sr3     call sr4  
        ret  
sr4     call sr5  
        ret  
sr5     call sr6  
        ret  
sr6     call sr7  
        ret  
sr7     call sr8  
        ret  
sr8     call sr9  
        ret  
sr9     ret  
  
org     $100  
  
Main  
    call sr1  
    jmp  Main
```

Programming the SX Microcontroller

In the SX, the stack is dedicated to storing return addresses only. It cannot be used to store other data, and there are no PUSH or POP instructions available, that you may know from microprocessors or other microcontrollers. This is not possible because in the SX, data and program code are stored in different memory, having different size.

In order to make use of the 8-level stack, make sure that the SX is configured accordingly, i.e. include the **DEVICE OPTIONX** or **DEVICE STACKX** directive in your program code.

1.5.3 Local Labels

In the previous LED-Blinker sample program, we have used the label **Loop** to name the main program loop, and the label **Loop1** to name the delay loop within the subroutine.

Try what happens when you rename the delay loop in the subroutine to **Loop** as well:

```
TimeEater  
Loop    decsz Counter1  
        jmp Loop  
        mov Counter1, #50  
        decsz Counter2  
        jmp Loop  
        decsz Counter3  
        jmp Loop
```

When you try to assemble this modified program, you will get an error message like "Label is already defined". Obviously, it is not possible that two labels representing different addresses can have the same name, otherwise the assembler would not "know" which address it should use when the program contains a reference to a label.

On the other hand, when programs become larger, you must be creative "inventing" new label names that are not only unique but also describe the meaning of what the labels stand for.

If you think of subroutines that are "generic" enough to be used in different programs, you must even take care that a subroutine copied to, or included with another source code does not make use of labels that are already defined elsewhere in this code.

Fortunately, most assemblers for the SX allow the definition of local labels. Modify the false code sample above to look like

```
TimeEater  
:Loop  decsz Counter1  
        jmp :Loop  
        mov Counter1, #50  
        decsz Counter2  
        jmp :Loop  
        decsz Counter3  
        jmp :Loop  
; ...
```

```

Main
  mov    !rb, #%11111110
: Loop
  call  TimeEater
  clrb  rb.0
  call  TimeEater
  setb  rb.0
  jmp   : Loop

```

and try to assemble the program again. This time, the assembler does not complain, and the program runs as expected although the subroutine and the main program loop make use of labels named **: Loop**.

The leading colons in front of the label names make **: Loop** local labels. A local label is valid only in the area of a program that is enclosed by two non-local, i.e. public labels.

This means that the first **: Loop** label in our example is valid from **TimeEater** up to **Main**, and the second **: Loop** label is valid from **Main** through the rest of the program code, where **TimeEater** and **Main** are both public labels.

This tip shall beware you from unnecessary headaches:

Just "for fun", insert a new global label **Foo** in **TimeEater**, following the first **jmp : Loop** instruction:

```

TimeEater

```

```

: Loop

```

```

  decsz Counter1

```

```

    jmp : Loop

```

```

Foo

```

```

  mov Counter1, #50

```

```

  decsz Counter2

```



and try to assemble the program. This time, the assembler will report that the label **: Loop** in the line following the **decsz Counter2** instruction is not defined.

According to the definition of the range in which a local label is valid, this is correct because the **: Loop** label following **decsz Counter2** is now only valid between **Foo** and **Main**, and there is no definition for **: Label** in this area.

Here, in this small program, such problems can be localized quite easily but imagine how difficult this might be in a large program with many local labels. Therefore, pay special attention when inserting new global labels "in the middle" of a program.

Programming the SX Microcontroller

When the assembler comes across a global label, it is stored in the symbol table, but also in a "Recent Global Label" buffer.

When the assembler comes across a local label, it builds the full label name by appending the local label name to the recent global label, and stores this name in the symbol table. For example, when the recent global label is **Subroutine**, a local label **:NoClear** would be stored as **Subroutine:NoClear** in the symbol table.

When the assembler comes across the reference to a local label, it first builds the full name from the recent global symbol and the local label, before searching it in the symbol table. On the other hand, the assembler also accepts references to full names.

This means that you can refer to a specific local label even from a location that follows a new global label. However, in this case, you must specify the label in full, like in the following example:

```
org    $000

Subroutine
    clr    w
:NoClear
    mov    $09, w
    ret

org    $100

Main
    call   Subroutine
    call   Subroutine:NoClear
    call   :NoClear           ; This causes an error
```

Although this code is of no specific use, it demonstrates how two different entry points of a subroutine can be accessed from the main program. First, the "regular" call enters the subroutine at the main entry point that causes **w** to be cleared, using the global name **Subroutine**. The second call refers to a local label within the subroutine by specifying the "full" label name (**Subroutine:NoClear**) i.e. **w** is not cleared. The third call causes an error because the assembler tries to locate the label **Main:NoClear** which does not exist.

1.5.4 Some More Considerations about Subroutines

You may have wondered why we have originated the **TimeEater** subroutine at address \$000, and why the main program begins at \$100 now.

To understand this, it is important to know that the call instruction only allows eight bits to specify the address of a subroutine. This is how the instruction code for call is structured:

```
1001 aaaa aaaa
```

When the assembler translates a **call** instruction, it replaces the lower eight bits **aaaa aaaa** in the instruction code with the lower eight bits of the address that is specified together with the call instruction. As a result, the entry point of a subroutine can only be specified by values between \$000 and \$0ff.

Therefore, it makes sense to reserve the lower area in program memory from \$000 through \$0ff for subroutines, and let the main program begin at \$100.

Later we will see that address \$000 has a special meaning when we discuss interrupts.

When the program memory from \$000...\$0ff is not large enough to hold the code for all of your subroutines, there are two options:

1. Usage of another program memory page (we will discuss this later in the tutorial).
2. Jump to a higher address in program memory - here comes an example:

```
RESET      Main  
org        $000  
  
SR1  
  jmp      _SR1  
  
org        $100  
Main  
  
; Initializations  
  
: Loop  
  ; Instructions within the main loop  
  
  call     SR1  
  jmp      : Loop  
  
_SR1  
  ; Instructions within the subroutine  
  
ret
```

As you can see, the subroutine **SR1** requires one word only in the program area from \$000 to \$0ff for the **jmp _SR1** instruction where the remaining code for the subroutine is located, following the main program loop in memory above \$100. The only little disadvantage is the additional word required for the **jmp** instruction and the three extra clock cycles needed to execute the **jmp**.

Fortunately, the assembler reports an error when you try to translate a program where the entry point of a subroutine lies above \$0ff (this may happen when you add more instructions to a subroutine that cause following subroutines being "advanced up" in program memory. Error mes-

Programming the SX Microcontroller

sages will be similar to "Address is not within lower half of memory page" or "ERROR: CALL must be to first half of page".

As mentioned before, a subroutine should behave like a "black box", i.e. it should be possible to call it from any location in the program, without any side effects caused by the subroutine. This especially means that the subroutine may not make changes to register contents which are important for other parts of the program.

For example, a function in the C programming language can fulfill this requirement when all arguments are passed to the function by value, and when the function uses local variables only. In addition, a function can optionally return a value to the calling program.

Requirements like this cannot be realized with the SX that easy. The main reason is the lack of a stack for data that C uses for passing arguments and for storing local variables.

When you write subroutines, take special care that no register contents are changed that are required to remain unchanged by the calling program. On the other hand, it is almost impossible to write a subroutine that does not at least modify the content of the **w** register. Therefore, the calling program should never trust that **w** remains unchanged after a subroutine call. The same is true for the flags in the status register that we will address later. Often, the **w** register, or the status registers are used to return a result from the subroutine to the calling program. Here is an example:

```
; The subroutine multiplies the contents of number by three  
; and returns the result in w.
```

```
;  
TimesThree  
    mov w, Number  
    add w, Number  
    add w, Number  
ret
```

```
mov Number, #2  
call TimesThree
```

```
; w now holds 6
```

Note, that this simple subroutine does not handle results greater than 255.

A good method to protect variables from being overwritten by a subroutine is the usage of different memory banks that are dedicated to the subroutines and the main program. We will discuss this in the next chapter.

Sometimes, subroutines need to store intermediate results. In such cases, it is a good idea to reserve one or more variables that can be used for temporary storage. By convention, the contents of such variables are only valid from the point where a subroutine has saved a value there, until it terminates. This means that no other part of the program may make assumptions on the vari-

able's contents. Nor can any subroutine assume that the variables hold specific values when it is called next.

When you write programs with nested subroutine calls, take care that a subroutine at a lower level does not make use of the same temporary variables. This would make its contents invalid for the calling subroutine.

Ubicom recommends the definition of variables named **l o c a l T e m p 0**, **l o c a l T e m p 1**, **l o c a l T e m p 2**, etc. as temporary storage. The subroutine at the highest nesting level may use **l o c a l T e m p 0**, the subroutine at the next lower level may use **l o c a l T e m p 1**, etc.

Even when you make use of clearly defined conventions how to use variables, it is always a good idea to double-check the integrity of variable usage, and it is also helpful to add comments to each subroutine to explain which variables might be changed, and on with variable contents the subroutine relies.

1.5.5 Correctly Addressing the SX Data Memory



(2.2.2 - 203) The reference section in this book describes how the data memory is divided in eight memory banks of 32 bytes each (SX 18/20/28), where the first 16 bytes are always located in the first bank (bank 0), no matter which bank is currently active. Within this first bank, only the upper eight bytes can be used as general-purpose registers where the lower eight registers stand for the SX'es special registers.

Let's try to learn more about memory banks with this little program (this version contains a bug as you will see):

```

; =====
; Programming the SX Microcontroller
; TUT006.SRC
; =====
i n c l u d e "Setup28. i n c"

R E S E T   M a i n

M a i n
  i n c $10
  i n c $30
  i n c $50
  i n c $70
  j m p   M a i n

```

Programming the SX Microcontroller

Enter and assemble this program, and then step through it with the debugger. After the first step, the **inc \$10** instruction will be executed that increments the memory location at \$10. As you step through this instruction, watch the displayed content of \$10, and see how it increments as expected.

Now, when you step through the **inc \$30** instruction you might expect that the contents of memory location \$30 will increment, but this is not the case. Instead, \$10 is incremented again. The same happens when you step through the next two instructions. Each time, \$10 is incremented, but not \$50 or \$70.

The problem here is that the instruction code for an **inc** only provides five bits for an address in data memory:

0010 101f ffff

When the assembler translates the instruction, it replaces the five bits that are marked with "f" with the lower five bits of the address argument that is part of the **inc** instruction.

Using our example, the following codes result:

```
$10           ->           0001 0000  
inc          ->           0010 101f ffff  
Instruction code ->       0010 1011 0000
```

```
$30           ->           0011 0000  
inc          ->           0010 101f ffff  
Instruction code ->       0010 1011 0000
```

```
$50           ->           0101 0000  
inc          ->           0010 101f ffff  
Instruction code ->       0010 1011 0000
```

```
$70           ->           0111 0000  
inc          ->           0010 101f ffff  
Instruction code ->       0010 1011 0000
```

As you can see, the instruction code is always the same, this is why each of the **inc** instructions addresses \$10, and not the other registers as you might have expected.

You may compare the organization of the SX data memory with a parking garage. Each parking deck has 32 lots, numbered from 0 to 31 where the lots 0 to 15 are reserved for special people, while the remaining lots from 16 to 31 are open to the public.

Let's assume, you will borrow your car to a friend, and you have arranged that he can pick up the car in our (fancy) parking garage.

Later, you park the car in deck 5, lot 16 and call your friend: "You can pick up my car in lot 16".

You can be sure, your friend will get mad about you because you forgot to tell him in which deck you have left the car for him. Neither did you tell him the type of the car nor the license plate number. How should he ever find your car when all lots No. 16 in all decks are occupied by cars?

This is similar to the SX data memory: For the upper 16 bytes in each bank, it is not enough to specify the address because this would always be a value from \$10 to \$1f. In addition, you must tell the SX which memory bank (deck) it shall use.

Now let's improve our program:

```

; =====
; Programming the SX Microcontroller
; TUT007. SRC
; =====
include "Setup28.inc"

RESET    Mai n

Mai n
    bank $10
    inc  $10

    bank $30
    inc  $30

    bank $50
    inc  $50

    bank $70
    inc  $70

    jmp  Mai n

```

Add the new bank instructions to the "old" example, assemble the modified version, and single-step through it using the debugger. Now you will notice that the registers \$10, \$30, \$50 and \$70 change as expected.

After a reset, please single-step the program once again, but this time, note how the value in the FSR register at address \$04 changes whenever a **bank** instruction is executed. It will hold the values \$00, \$20, \$40 and \$60 in this sequence.

The **bank** instruction copies the upper three bits of the instruction argument into the upper three bits of the FSR while the other bits in the instruction argument are ignored. Thus, instead of writing bank \$10, we could have written bank \$00 up to bank \$1f instead without any difference. The only important fact is that the upper three bits in the argument must all be cleared in order to select bank 0.

Programming the SX Microcontroller

Due to "historical reasons" (compatibility to other controller devices), the SX data memory is also called "data file", and one location in the data file is called "file register". This explains the name of the FSR: File Select Register.

Let's visualize how the data memory is physically addressed:

The complete address is composed of the three upper FSR bits and the lower five bits of the instruction code. In our example, the addresses for the four `inc` instructions are built as follows:

```
bank $10 ->      0001 0000
FSR       ->      000? ????
inc $10   -> 0010 1011 0000
Address   ->      0001 0000 = $10

bank $30 ->      0011 0000
FSR       ->      001? ????
inc $30   -> 0010 1011 0000
Address   ->      0011 0000 = $30

bank $50 ->      0101 0000
FSR       ->      010? ????
inc $50   -> 0010 1011 0000
Address   ->      0101 0000 = $50

bank $70 ->      0111 0000
FSR       ->      011? ????
inc $70   -> 0010 1011 0000
Address   ->      0111 0000 = $70
```

The three bits highlighted in gray are the three upper bits in the bank instruction's argument and the three upper bits in the FSR, and the bits shown in inverse are the five lower bits from the instruction code.

You may ask why we use different address arguments for the `inc` instructions when the assembler generates the same code anyway. Actually, you could use \$10 as address argument for all the `inc` instructions, and the program would work as fine as before. However, the program becomes more readable if we use the address arguments that indicate which file registers we really mean. Nevertheless, you may never forget that this is not enough to uniquely specify an address in the data file – in addition, you need to specify the right bank before accessing a memory location.

Now, let's enhance the program by inserting the `inc $08` and `inc $0f` instructions following each `bank` instruction in the code:

```
; =====
; Programming the SX Microcontroller
; TUT008. SRC
; =====
include "Setup28.inc"

RESET   Main
```

```

Main
bank $10
inc $08
inc $0f
inc $10

bank $30
inc $08
inc $0f
inc $30

bank $50
inc $08
inc $0f
inc $50

bank $70
inc $08
inc $0f
inc $70

jmp Main

```

As you single-step through the program, you will notice that the **inc \$08** and **inc \$0f** instructions following each **bank** instruction always increment the registers at \$08 or \$0f, no matter which bank is currently selected.

The reason is that data addresses in the range from \$00 up to \$0f always address physical registers that are located in bank 0, where these physical registers do not exist in banks 1 through 7. This is an exception to the rule how data memory addresses are composed: The upper three bits of an address will be always cleared to 000 when the address contained in the instruction code is below \$10, and the upper three FSR bits are ignored in this case.

This adds another "strange" behavior to our parking garage example: The parking lots 0 through 15 only exist in deck 0. When you try to drive into one of these lots in another deck, you will always be "beamed" down to deck 0.

Think of the data addresses from \$00 through \$0f as "Global Variable Area", i.e. you can always reach variables in this area, no matter what bank is currently selected. Also keep in mind that the lower eight bytes in this area are reserved for the SX "special registers".

1.5.6 Clearing the Data Memory and Indirect Addressing

1.5.6.1 SX 18/20/28

When starting the debugger, or after a reset, you may have noticed that the SX data memory holds random data at this time. Sometimes, it makes sense to setup the data memory for a "clean

Programming the SX Microcontroller

start” before executing the application code by initializing all memory locations to zero (except the special registers at \$00 through \$07).

The following “monster” program clears the contents of the registers from \$08 to \$3f:

```

; =====
; Programming the SX Microcontroller
; TUT009.SRC
; =====
include "Setup28.inc"

RESET    Main

Main
    clr  $08
    clr  $09
    clr  $0a
    clr  $0b
    clr  $0c
    clr  $0d
    clr  $0e
    clr  $0f

    bank $10
    clr  $10
    clr  $11
    clr  $12
    clr  $13
    clr  $14
    clr  $15
    clr  $16
    clr  $17
    clr  $18
    clr  $19
    clr  $1a
    clr  $1b
    clr  $1c
    clr  $1d
    clr  $1e
    clr  $1f

    bank $30
    clr  $30
    clr  $31
    clr  $32
    clr  $33
    clr  $34
    clr  $35
    clr  $36
    clr  $37
    clr  $38
    clr  $39
    clr  $3a
    clr  $3b
    clr  $3c
```

```

clr $3d
clr $3e
clr $3f
jmp Main

```

Before you really enter this program, and consider adding more **bank** and **clr** instructions for the rest of the memory, let's search for a better solution.

But before we do so, there is one thing we can even learn from this "dumb" program: There is no need to repeat the **bank** instruction as long as all the following instructions refer to registers in the same bank.

Our current task is to clear all registers in data memory with the exception of registers \$00 through \$07, and not to use a single **clr** instruction for each of these registers.

This task could be easily solved if the **clr** instruction could accept the contents of a variable as an address of the location to be cleared, instead of a constant address as argument. We might then place the **clr** inside of a program loop that increments the address argument until we have cleared all memory.

Fortunately, the SX allows us to do something like this by writing

```
clr ind
```

This kind of addressing a register is also called *Indirect Addressing* because in this case, the address is not directly specified by the instruction argument. Instead, it is read from another register in memory. The SX has one special register that "delivers" the indirect address - it is called the FSR (File Select Register).

When the assembler "sees" an **ind** or **indf** as instruction argument, it sets the address part of the instruction code to %00000, i.e. the "virtual" register at address \$00 will be accessed. Actually, this means that the SX accesses the register whose address is represented by the eight bits in FSR.

With this information, your first idea might be to write the following instructions in order to clear the data memory:

```

clr fsr
:ClearData
clr ind
incsz fsr
jmp :ClearData

```

Don't run these instructions in the SX because some more steps are required before:

First, the FSR is cleared to start at address \$00, and inside the loop, the register is cleared which is currently addressed by the FSR contents. While the program executes the loop, FSR is incremented each time, so when the loop is done, all registers from \$00 through \$ff should be cleared.

Programming the SX Microcontroller

As you know, registers at \$00 through \$07 are "restricted area", i.e. this area must not be cleared because it contains the SX special registers, including FSR itself which is located at \$04. Even worse, the program counter, PC is located at \$03. Imagine what happens if the PC register would be indirectly cleared - this loop would never end!

The first idea for skipping this "restricted area" is to not clear the FSR but to initialize it to \$08 at the beginning of the code. Unfortunately, this does not solve our problem because the addresses \$20...\$27, \$40...\$47, etc. are also "mapped" into that area (see the "fancy" parking garage example before) - again, the special registers would be "illegally" touched.

The sequence below solves the problem:

```
clr   fsr           ; 1
      ; 2
:ClearData         ; 3
  sb   fsr.4        ; 4
      Setb fsr.3     ; 5
  clr   ind         ; 6
  ijnz  fsr, :ClearData ; 7
```

We have added line numbers as comments in order to make it easier to refer to specific lines.

The table below shows some values in hexadecimal and binary to help you understanding the following explanations:

Hex	Binary							
Bits	7	6	5	4	3	2	1	0
\$00	0	0	0	0	0	0	0	0
\$08	0	0	0	0	1	0	0	0
\$09	0	0	0	0	1	0	0	1
\$10	0	0	0	1	0	0	0	0
\$1F	0	0	0	1	1	1	1	1
\$20	0	0	1	0	0	0	0	0
\$28	0	0	1	0	1	0	0	0

In line 1, FSR is cleared, i.e. it contains \$00 then.

Line 4 contains the instruction **sb fsr.4** (Skip if Bit). The instruction skips the next instruction when bit 4 in FSR is set. This is not the case now, therefore, the next **setb fsr.3** instruction in line 5 is executed, so bit 3 in FSR is set. This means that FSR now contains \$08, i.e. the "restricted area" has been successfully skipped.

The **ijnz fsr, ClearData** (Increment and Jump if Not Zero) is a compound instruction, i.e. the assembler generates two instructions. The instruction first increments the content of FSR, and then performs a jump to **ClearData** in case the increment did not overflow the contents of FSR to zero.

On the first "round" through the loop, register 8 is cleared (in line 6), FSR is incremented to \$09 (in line 7), and the jump to **ClearData** is executed.

FSR contains \$09 in line 4 now. As bit 4 is not yet set, line 5 is executed again. As FSR bit 3 is already set, another **setb fsr. 3** instruction does not change the contents of FSR at all.

This will be continued until FSR has finally reached a value of \$10. From then on, line 5 will be skipped.

As soon as FSR is incremented to \$20, i.e. when bit four is clear, line 5 will be executed again that sets FSR's bit three. So, this skips the "restricted area" again.

This is repeated until FSR finally overflows from \$ff to \$00 causing the loop to terminate.

While the loop is executed, the FSR contents also holds the values \$28...\$2f, \$48...\$4f, etc., although these addresses are all mapped into the address space \$08...\$0f in bank 0, i.e. the registers \$08...\$0f will be cleared "very thoroughly". As this is not a problem, it does not make sense to add another exception that avoids those "multiple clears".

When you execute the following program in the "animated" or "Walk" mode of your debugger, you can easily trace how the file registers are cleared:

```

; =====
; Programming the SX Microcontroller
; TUT010. SRC
; =====
include "Setup28.inc"

RESET Main

Main
clr    fsr
ClearData
sb     fsr. 4
Setb   fsr. 3
clr    ind
ijnz   fsr, ClearData
jmp    $

```

After the program has cleared the data memory, it enters into an endless loop when it comes to the **jmp \$** instruction that might look a bit strange to you.

We did not forget to add some hexadecimal digits after the \$ sign. The "stand-alone" \$ in this case, tells the assembler to replace it with the value, the assembler currently has stored in the internal address counter. In this case, this is the address of the **jmp** instruction itself. We could have reached the same effect by writing

```

Forever
jmp Forever

```

Programming the SX Microcontroller

The only way to release the SX out of this endless loop is to either do a reset, or to turn off the power supply.

In case you do a reset, and then want to run the program again, it is most likely that all register contents are already cleared, and so you can no longer see the effect of the **clr** instructions.

To "fill" the registers with random data, remove the power supply for a while, and then restart the debugger after you have powered up the system again.



Whenever a program contains the initial loop to clear the data memory, single-stepping the program with the debugger means that you would have to step through all the **ClearData** loop cycles.

Remember that your debugger has a "Breakpoint", or an "Execute-to-Cursor" feature that helps you to skip over the **ClearData** loop at high speed.

Indirect addressing is not only helpful to clear the data memory. When you are used to other programming languages, you know the concept of arrays and array indices.

When you indirectly address a group of registers in data memory that are located at consecutive addresses, these registers are nothing else but the elements of an array, and the FSR is the array index. We will present more examples for indirect addressing as we go ahead in this tutorial.

1.5.6.2 SX 48/52



(2.2.2.2 - 205) The "big brothers" of the "little" SX controllers described before come with 262 bytes of available data memory. In addition, the special registers and the port data registers are also mapped into the data address space, requiring ten more addresses (for INDF, RTCC, PD, STATUS, FSR, and Ports RA through RE). This results in a total of 272 different addresses that must be accessed.

The address space is divided into a "Global Register Bank", and 16 banks (Bank 0..F) of 16 bytes each giving the total of 262 locations.

In order to clear the data memory, we can also use the FSR to indirectly address most locations in data memory. Similar to the "smaller" SX controllers, the instruction argument **ind** or **indf** causes the assembler to set the address part of the instruction code to %00000, and the SX 48/52 will access the register whose address is represented by the eight FSR bits.

This means that 256 bytes can be addressed this way. When the FSR has a value from \$00 through \$1f, a register in the "Global Register Bank" is accessed. In this bank, the special registers and the

port data registers are located at \$00...\$09, plus six general purpose registers at \$0a through \$0f. When the FSR has a value from \$10 through \$ff, an indirect memory access addresses one of the registers in bank 1 through bank F, totaling in 256 different locations. As you can see, the 16 bytes in bank 0 cannot be accessed this way.

In order to access the 16 registers in bank 0, we must use semi-direct addressing, i.e. the highest bit (bit 4) in the address part of an instruction code must be set, and the upper four bits of FSR must be cleared. Besides this, the semi-direct addressing mode allows access to all 256 registers in banks 0 through F, but does not allow access to the global and special registers.

The following code clears all SX 48/52 data registers:

```

; Clear the SX 48/52 data memory
;
mov fsr, #0a      ; As indirect addressing covers the
                  ; global registers and banks 1 to F,
                  ; we must initialize FSR to $0a in
                  ; order to not touch the "special
                  ; registers" (IND...RE).
ClearRAM
clr ind
incsz fsr
  jmp ClearRam

; As registers in bank 0 cannot be addressed indirectly,
; we must clear them using semi-direct addressing
;
clr fsr          ; "Point" FSR to bank 0
clr $10          ; Now clear the registers
clr $11          ; NOTE: Bit 4 in the instruction
clr $12          ; code must be set in order
clr $13          ; to address a register in
clr $14          ; bank 0, and not a global
clr $15          ; register.
clr $16
clr $17
clr $18
clr $19
clr $1a
clr $1b
clr $1c
clr $1d
clr $1e
clr $1f

```

At the beginning of this code, we initialize FSR to \$0a. This makes sure that we leave the special registers and the port registers alone, i.e. the first register that will be cleared is the global register at \$0a.

We then simply indirectly clear the register that is addressed via the FSR, and increment FSR until its value has finally reached \$00 again.

Programming the SX Microcontroller

In order to clear the 16 registers in bank 0, we use semi-direct addressing, i.e. we set FSR to \$00 in order to address bank 0, and then use 16 separate **clr** instructions. Because semi-direct addressing requires that bit four in the instruction code is set, the address arguments of the **clr** instructions always start with \$1.

1.5.7 Symbolic Variable Names

In earlier example programs, we already have made use of symbolic names (or addresses) for variables in data memory, because symbolic names are much more readable and understandable for us human beings. In this chapter, we have used hexadecimal addresses instead, to better show the structure of the data memory.

Lets have a look at a program that first clears the data memory, and then increments some registers in various banks, using symbolic addresses.

Here, we first introduce a new include file, called "Clr2x.inc" that contains the code required to clear all data memory in an SX 2x chip, i.e. the code that we had discussed before. As clearing all data memory is a common task, found in many SX applications, it makes sense to create an include file once, and then use it with the **include** directive for many applications.

Please enter the following code lines, and save them in a file named "Clr2x.inc":

```
; =====  
; Programming the SX Microcontroller  
; Clr2x.inc  
; =====  
; Clear all data memory  
  
    clr        fsr  
  
:ClearData  
    sb        fsr.4  
    setb     fsr.3  
    clr        ind  
    ijnz     fsr, :ClearData
```

Note that the code in this include file makes use of a local label only, **:ClearData**. You should always only use local labels within include files that generate code in order to make it possible that such code can be included at any place in the main program without interfering with other current global labels.

Next, please enter the following program code:

```
; =====  
; Programming the SX Microcontroller  
; TUT011.SRC  
; =====  
include "Setup28.inc"
```

```

RESET      Main
org        $10                ; 1
BankA     equ $                ; 2
Counter1  ds 1                 ; 3

org        $30                ; 4
BankB     equ $                ; 5
Counter2  ds 1                 ; 6

org        $50                ; 7
BankC     equ $                ; 8
Counter3  ds 1                 ; 9

org        $70                ; 10
BankD     equ $                ; 11
Counter4  ds 1                 ; 12

org        $100               ; 13

Main

; Includes code to clear all data memory
;
include "Clr2x.inc"

; Testing data memory addressing
TestAddr
bank      BankA                ; 21
inc       Counter1             ; 22

bank      BankB                ; 23
inc       Counter2             ; 24

bank      BankC                ; 25
inc       Counter3             ; 26
bank      BankD                ; 27
inc       Counter4             ; 28

jmp       TestAddr             ; 29

```

You already know the **org** directive. In line 1, for example, the assembler is instructed to place subsequent definitions starting at address \$10.

1.5.8 The EQU, SET and = Directives

New in the program above is the directive

BankA equ \$

Programming the SX Microcontroller

in line 2. The **equ** (Equates) directive is used to assign a constant value to a symbolic name (or label). In this case, the label **BankA** is defined, and it receives the value of the current position of the assembler-internal address counter (\$10 in this case).

Alternatively, we could have written

```
BankA equ $10
```

but should you ever intend to move **BankA** to another memory bank, you would have to modify both, the **org** and **equ** arguments, where using the "\$" only requires a modification of the **org** argument, i.e. one reason for a possible bug has been eliminated.

Instead of using the **equ** directive, you may also use the equals sign (and some assemblers, like SASM also allow set instead of =), i.e. instead of **BankA equ \$** you may write

```
BankA = $
```

or

```
BankA set $
```

instead.

The difference between **equ** and **=/set** is that an assignment once made with **equ** cannot be changed throughout the rest of the program, where **=** and **set** allow re-definitions. When you assign a value to a label using **=** or **set**, this assignment remains in force until the assembler comes across a new **=/set** assignment for the same label, while working through the source code from top to bottom.

For example, the following source lines will cause a "Symbol already defined" error:



```
Donal d equ 5
```

```
;
```

```
; Donal d is 5 now
```

```
;
```

```
Donal d equ 6 ; error
```

where the following lines will assemble without error:

```
Donal d = 5
```

```
;
```

```
; Donal d is 5 now
```

```
;
```

```
Donal d = 6
```

```
;
```

```
; Donal d is 6 now
```

Line 3 contains the definition **Counter1 ds 1**

This defines a symbolic address named **Counter1** at the current memory position (\$10), and it reserves one byte of memory. This variable is located in bank 0 (or in **BankA** if you prefer the symbolic bank name).

Lines 4 through 12 similarly define the constants **BankB**, **BankC** and **BankD** and the variables **Counter2**, **Counter3** and **Counter4**. The variables are located in memory banks 1, 2 and 3.

In lines 15 to 20, we clear the data memory, and in lines 21 to 29, we test addressing variables in different memory banks as in the former demonstration program, this time using symbolic addresses.

1.5.9 Some Thoughts about Data Memory Usage

Let's go through the structure of the SX2x data memory again to see what segments are available:

Addresses \$00...\$07:	8 reserved registers
Addresses \$08...\$0f:	8 available registers, always addressable
Addresses \$10...\$1f:	16 available registers in bank 0
Addresses \$30...\$3f:	16 available registers in bank 1
Addresses \$50...\$5f:	16 available registers in bank 2
Addresses \$70...\$7f:	16 available registers in bank 3
Addresses \$90...\$9f:	16 available registers in bank 4
Addresses \$b0...\$bf:	16 available registers in bank 5
Addresses \$d0...\$df:	16 available registers in bank 6
Addresses \$f0...\$ff:	16 available registers in bank 7

As we already had mentioned in the chapter dealing with subroutines, the SX does not offer a method to realize the concept of local and global variables "per-se" but the structure of the data memory allows for a concept that comes close to it.

In the address space from \$08 through \$0f you should locate global variables, i.e. variables that can be easily accessed from the main program as well as from subroutines without the need to switch banks because this area can always be reached, no matter what bank is currently selected.

The remaining blocks of 16 bytes in eight banks (0..7) should be dedicated to subroutines, or to major blocks in the main program having different functionality. If possible, each of the subroutines or the major blocks should use a different memory bank.

There are cases where this concept cannot always be maintained consequently, especially when the eight bytes in bank 0 are not enough to hold all global variables.

Programming the SX Microcontroller



(2.2.2.2 - 205) The data memory of the "big" SX 48/52 controllers is organized a bit differently. You can find detailed information about the SX 48/52 in the reference part of this book.

1.5.10 Don't Forget to Select the Right Bank



Always keep a close eye on the currently selected memory bank. When you forget to activate the correct bank, unpredictable memory contents will be the result in most cases that might crash a program, and often, such errors are hard to find, even with a debugger.

Please have a look at the following program:

```

; =====
; Programming the SX Microcontroller
; TUTO12. SRC
; =====
include "Setup28.inc"

RESET                Main

org                  $10
Counter             ds 1

org                  $30
Timers              equ $
Timer1              ds 1

org                  $000

; -----
; Subroutine for time delays
; -----
Delay
  bank              Timers
  clr               Timer1

: Loop
  decsz             Timer1
  jmp               : Loop
  ret

org                  $100
; -----
; Main program
; -----
Main
  bank              Counter
```

```

: OuterLoop
  cl r      Counter

: InnerLoop
  call     Delay
  incsz    Counter
  jmp      : InnerLoop
  jmp      : OuterLoop

```

At the beginning, the program defines names for the memory bank **Timers** which holds the **Timer** variable, while no name is defined for the global bank that “hosts” **Counter**.

The main program first selects bank **Counter** to make sure that the bank selection does not depend on random settings of the upper three bits in the FSR. As you can see, a bank can also be selected by specifying the symbolic name of a variable that is defined in that bank instead of using a separately specified bank name, when no separate name is defined for that bank.

Then, the **Counter** variable is cleared and execution enters into **: InnerLoop**. Inside this loop, the **Delay** subroutine is called, and then **Counter** is incremented. **: InnerLoop** is repeated until **Counter** contains a value of zero. If this is the case, execution jumps back to **: OuterLoop**.

The **Delay** subroutine switches the bank to **Timers**, clears the **Timer** variable in this bank and the program loop then decrements **Timer** until it reaches zero. Finally, the return to the calling program is executed.

Please enter the program, and activate the debugger. Set a breakpoint on the **jmp : OuterLoop** instruction, and run the program at full speed.

After **: OuterLoop** has been executed 256 times, the program should stop at the breakpoint, so wait and see...

Well, you can wait as long as you want, the breakpoint will never be reached. To “bail out” of the program, do a reset.

Did you already figure out the problem?

The “bad guy” here is the **Delay** subroutine. It activates the **Timers** bank in order to access the **Timer** variable, does the delay loop until **Timer** contains zero, and returns to the main program, leaving bank **Timers** selected!

The main program - on the other hand - “believes” that bank **Main** is active, and “thinks” it increments the **Counter** variable. (In our “Virtual Parking Garage” we did go to the right lot, but at the wrong deck.) In reality, the main program increments the first memory location in the **Timers** bank and this is the **Timer** variable that currently holds a value of zero. As this is the case at each return from the **Delay** subroutine, this memory location can never be zero in the main program.

Programming the SX Microcontroller

To fix that bug, add a **bank Main** instruction immediately before the **ret** instruction in the **Delay** subroutine.

Imagine how cumbersome it can be to find such a nasty bug in a large program - therefore, always keep a close eye on the bank selections!

1.5.11 Saving the Current Bank in a Subroutine

In a small program like the one shown before, it is acceptable that the subroutine switches back to bank **Main** before returning, but for a generic subroutine, it does not make sense to switch to a specific bank on return because the subroutine cannot "know" which bank was active before. Here, the following example shows one solution:

```
org      $08
local Temp0  ds 1

org      $30
Timers    equ $
Timer1    ds 1

org      $000

;-----
; Subroutine for time delays
;-----
Delay

    mov     local Temp0, fsr

    bank   Timers
    clr    Timer1

: Loop
    decsz  Timer1
    jmp   : Loop

    mov     fsr, local Temp0
    ret
```

Here, we have declared a variable **local Temp0** in the "global" section of the data memory, and the subroutine copies the content of FSR into that variable, before FSR is changed. Immediately before returning from the subroutine, the original content of FSR is restored from **local Temp0**.

This makes the subroutine more generic, but problems might occur if you try to use nested subroutines, like in this example:

```
org      $08
local Temp0  ds 1

org      $10
BankA      equ $
```

```

org      $30
BankB   equ  $

org      $50
BankC   equ  $

SR1
  mov    localTemp0, fsr      ; LocalTemp0 now "points" to BankA
  bank   BankB
  ;
  ; some instructions
  ;
  call   SR2
  ;
  ; some more instructions
  ;
  mov    fsr, localTemp0
ret

SR2
  mov    localTemp0, fsr      ; LocalTemp0 now "points" to BankB
  bank   BankC
  ;
  ; some instructions
  ;
  mov    fsr, localTemp0
ret

Main
  bank   BankA
  call   SR1
  ;
  ; etc.

```

Here, **SR1** saves the FSR to **localTemp0** that currently contains the address of the current bank in the main program (i.e. **BankA**), before changing FSR to address **BankB**. Later, **SR1** calls **SR2** and **SR2** again saves FSR to **localTemp0**. This means that the original contents of FSR saved in **localTemp0** that are required to restore FSR for the main program are lost.

It is obvious that just one variable is not enough to save the FSR when it comes to nested sub-routines.

The code below makes use of two temporary variables to fix that problem:

```

org      $08
localTemp0 ds 1
localTemp1 ds 1

org      $30
BankB   equ  $

org      $50
BankC   equ  $

```

```
SR1
  mov     localTemp0, fsr
  bank   BankB
  ;
  ; some instructions
  ;
  call   SR2
  ;
  ; some more instructions
  ;
  mov     fsr, localTemp0
ret

SR2
  mov     localTemp1, fsr
  bank   BankC
  ;
  ; some instructions
  ;
  mov     fsr, localTemp1
ret

Main
  bank   BankA
  call   SR1
  ;
  ; etc.
```

Another chance you have, is to insert a **bank** instruction immediately following the subroutine call to make sure that the correct bank is selected no matter what changes the subroutine made to the FSR.

Whatever solution you choose, paying special attention to keeping the right bank set avoids cumbersome bug-fixing later.

1.5.12 Routines for an FSR Stack

As you know, return addresses for subroutines are automatically stored in an eight-level stack by the SX. Why should we not create a similar structure to save the FSR before switching banks in a subroutine, and restore the FSR from the "software stack" before returning?

Besides this, a stack buffer can be used to temporarily save other data as well.

The following program demonstrates the implementation of a stack, the necessary subroutines to save and restore the FSR, and how these routines are used:

```
; =====
; Programming the SX Microcontroller
; TUT013.SRC
; =====
include "Setup28.inc"
```

```

RESET   Main

org     $10
BankA   equ $
org     $30
BankB   equ $
org     $50
BankC   equ $
org     $70
BankD   equ $
org     $90
BankE   equ $
org     $b0
BankF   equ $
org     $d0
BankG   equ $

; -----
; Bank for the FSR Stack
; -----
org     $f0
Stack   equ $
CurrFSR ds 1
SP      ds 1
ST      ds 7

org     $000

; -----
; Save the FSR to the stack
; -----
PushFSR
    mov     w, FSR
    bank   Stack
    mov     CurrFSR, w
    mov     w, #ST
    add     w, SP
    mov     fsr, w
    mov     w, CurrFSR
    mov     ind, w
    inc     sp
    mov     w, CurrFSR
    mov     fsr, w
ret

; -----
; Restore the FSR from the stack
; -----
PopFSR
    bank   Stack
    dec     SP
    mov     w, #ST
    add     w, SP
    mov     fsr, w

```

Programming the SX Microcontroller

```
    mov    w, ind
    mov    fsr, w
ret

SR1
    call  PushFSR
    bank  BankC
    call  SR2
    call  PopFSR
ret

SR2
    call  PushFSR
    bank  BankD
    call  SR3
    call  PopFSR
ret

SR3
    call  PushFSR
    bank  BankE
    ;
    ; some instructions
    ;
    call  PopFSR
ret

Main
    bank  Stack
    clr  SP

    bank  BankB
    call SR1
    jmp  $
```

In bank **Stack**, we have defined some variables:

CurrFSR: 1 byte temporary storage for the FSR
SP: 1 byte for the stack pointer
ST: 7 bytes for the stack memory

We can also describe the stack memory as an array of 7 bytes, and the stack pointer is an index into that array.

The **PushFSR** subroutine is used to save the current contents of the FSR in the stack:

First, the current content of the FSR is copied to the W register before FSR is changed by the **bank Stack** instruction that selects the **Stack** bank.

Next, the content of **w** (i.e. a copy of the original FSR value) is stored in **CurrFSR** because we need **w** for other purposes.

The next task is to point FSR to the next empty location in the stack. The **SP** variable contains the offset to the next empty location, and the main program has initially cleared **SP**.

First, the instruction

```
mov w, #ST
```

copies the address (and not the contents) of the first stack location to **w**, and

```
add w, SP
```

adds the offset into the stack to **w**. Now **w** contains the absolute address of the next empty stack location. The instruction

```
mov fsr, w
```

copies this absolute address to the FSR.

```
mov w, CurrFSR
```

copies the original contents of FSR to **w**, and finally,

```
mov ind, w
```

saves this value to the indirectly addressed stack item. Before returning from the subroutine, we increment the stack pointer **SP** to hold the offset of the next empty location in the stack, and we then restore the original contents of the FSR.

The **PopFSR** subroutine retrieves the contents of FSR that was saved last in the stack, and copies that value into the FSR.

The instruction

```
bank Stack
```

activates the **Stack** bank, and thus modifies the FSR, but this is no problem here as we are going to change FSR anyway.

As **SP** contains the offset into the stack to the next empty location, we first must decrement **SP** in order to set the offset to the location where FSR was saved last. Again, we need to store the absolute address of this location in FSR for indirect addressing:

```
mov w, #ST  
add w, SP  
mov fsr, w
```

and finally,

```
mov w, ind  
mov fsr, w
```

set FSR to the value that was saved last in the stack.

Programming the SX Microcontroller

It is important that **SP** is cleared before **PushFSR** is called the first time. Here, this is done at the very beginning of the main program.

To demonstrate how the stack routines work, the main program selects **BankB**, then calls **SR1**, which calls **SR2**, and **SR2**, calls **SR3**. Each of the subroutines first calls **PushFSR** to save the FSR, and then selects another bank. Each subroutine calls **PopFSR** immediately before returning in order to restore FSR.

Single-step this program to see how the FSR is saved at the beginning of a subroutine, and how it is restored to its former value before the subroutine terminates.

Please note that this version of **PushFSR** does not handle a stack overflow situation, which will occur when **PushFSR** is called more than seven times without calling **PopFSR** in between. In addition, **PopFSR** has no protection against stack underflow that can occur when **PopFSR** is called without a previous call to **PopFSR**.

The stack size of seven bytes is large enough when you want to use the stack just to save and restore the FSR in subroutines, because the SX allows for a maximum of eight levels of nested subroutines. As **PushFSR** and **PopFSR** are also subroutines that don't require to internally save FSR on the stack, a nesting level of seven remains for other subroutines that might save and restore the FSR.

Besides using the stack routines to save and restore the FSR within subroutines, you can use the "software stack" as temporary storage for other values as well. In this case, you might consider increasing the stack size.

Although the stack routines make it easy to write "generic" routines that don't require explicitly declared variables for temporary storage, you should keep in mind that pushing a value requires 17 clock cycles, and that a pop requires 13 clock cycles while using a temporary variable requires one or two clock cycles for the save and the restore.

1.5.13 The "#"-Pitfall



Before ending this chapter, we want to make you aware of an error that can easily be made, and that could cause you some sleepless nights.

Remember the sentence "first, the instruction **mov w, #ST** copies the address (and not the contents) of the first stack location to w".

An instruction parameter with a leading "#" symbol indicates that the value given after the "#" shall be interpreted as constant or "literal" value, and we have used instructions like

mov Number, #2

more than once. For example, the compound instruction above, copies the constant value 2 into a variable called Number.

Instead of this, you could also write the following sequence:

InitValue equ 2
mov Number, #InitValue

During translation, the assembler replaces **InitValue** with the value that has been assigned to that label before (2 in our example).

On the other hand, the following sequence is not allowed:

InitValue equ #2
mov Number, InitValue

The "#" character must be part of the instruction parameter, and it cannot be included in a constant definition.

A symbolic address too is nothing else but a numerical value. The only difference is that the assembler has automatically assigned a value (the current contents of the internal address counter).

For example, when the assembler translates the instruction

mov w, #ST

in our example program, it replaces **ST** with \$f2 because this is the address of **ST**. Imagine what would happen if you forget to add the "#" character, so that the instruction would read

mov w, ST

instead. Now the **mov** instruction does not copy the address of ST to w, but the contents of the first item in the stack instead.

You can be caught in the same trap if you intend to copy a constant value to a register, like in

mov Number, #15

when you forget to type the "#" so that the instruction reads

mov Number, 15

instead. In this case, the contents of the register at 15 (or \$0f) will be copied to w, but not the constant value 15.

Please pay extra attention when using constant values - never forget the leading "#" - this makes you sleep longer and deeper!

Programming the SX Microcontroller

The previous example programs used to define a "Bank Name" as in

```
org $30  
BankC equ $  
Var1 ds 1  
Var2 ds 1  
Var3 ds 2
```

Later in the program, we have used the symbolic bank name, as in

```
bank BankC  
inc Var1
```



Instead of using a special bank name, you can also use the name of any variable declared in that bank as an instruction parameter, as in

```
bank Var1  
inc Var1
```

Because the **bank** instruction just copies the upper three bits of the argument to the upper three bits of the FSR, any address within a **bank** is a valid argument for the bank instruction as the upper three bits are always the same.

Nevertheless, in most cases, it makes sense to use a specific bank name as this does describe the "class" of a bank better than the name of a variable that is part of this "class". This makes it easier to read the program (not for the assembler, but for us human beings), and it helps keeping track whether the right bank is selected.

1.6 Arithmetic and Logical Instructions

1.6.1 Arithmetic Instructions

The SX controllers have not been designed to perform sophisticated calculations but as high-speed controllers for communications, real-time system control, etc.

Nevertheless, there are basic arithmetic instructions available you may use to perform calculations that are more complex by combining them in a program. As the SX works on byte level, arithmetic and logical instructions operate on byte-level too, i.e. the operands have the size of one byte each, and the result has the size of one byte (plus one bit, as we will see).

1.6.1.1 Addition

For example, the add instruction has the following syntax:

add op1, op2

This instruction calculates the sum of op1 and op2, and places the result in op1. We can say that op1 is the target because it finally holds the result.

There are two basic variants of the add instruction:

add fr, w

add w, fr

Both variants calculate the sum of the contents of a file register (fr) and w. The first variant stores the result in fr, and the second one places the result in the w register. In both cases, the content of the right operand remains unchanged.

There are two more compound add instructions which the assembler replaces by two basic instructions:

```
add    fr, #Constant ->   mov    w, #Constant
                               add    fr, w
```

```
add    fr1, fr2    ->   mov    w, fr2
                               add    fr1, w
```

The first instruction stores **fr + Constant** in **fr**, and the second instruction stores **fr1 + fr2** in **fr1**.

Note that both instructions modify the contents of **w**, and both instructions must not immediately follow a skip instruction.

When adding two one-byte values, there are two possible special cases:

Programming the SX Microcontroller

1. The result is greater than 255

As the target can only store values up to 255, the target would contain a wrong result in this case because the ninth bit of the result is lost. Therefore, bit 0 in the STATUS register is set in this case. This bit is commonly known as "Carry Flag", "C Flag", or "Overflow Flag".

If an addition result is greater than 255, the C flag is set, otherwise it is cleared.

You can address the C flag just like any other flag using the syntax

\$03. 0

because the STATUS register has address \$03 in data memory. As the assembler accepts a pre-defined name for the STATUS register, you may also write

Status. 0

As it is sometimes necessary to set or clear the carry flag before a specific instruction is performed, you may use the instructions

clrb Status. 0

setb Status. 0

to clear or set the carry flag, or use the instructions

clc

stc

Actually, the assembler translates all variants of the set and clear instructions into **clrb \$03. 0** or **setb \$03. 0** instruction codes.

2. The result is zero

There is another flag, the "Zero Flag" or "Z Flag", indicating a zero result. This flag is located in bit 2 of the STATUS register. It is set, when an operation yields in a zero result, otherwise the Z flag is cleared.

To clear or set the Z flag, there are two pre-defined instructions available:

clz

stz

After an addition, the Z flag is set, when the result is zero, and there are two reasons for that: Either, both operands contain zero, or the result is exactly 256, or binary 1 0000 0000. In the second case, the C flag will also be set because this is an overflow situation.

1.6.1.2 Skip Instructions

To allow a program to react on special cases, there are conditional instructions that perform a skip, depending on the status of the C or Z flags:

sz (skip if zero)

snz (skip if not zero)

sc (skip if carry)

snc (skip if not carry)

These instructions skip over the next instruction in case the specified condition is true, i.e. the next instruction is executed if the specified condition is false.

In addition, there are skip instructions that allow testing any bit in a file register:

sb fr. bit (skip if bit)

snb fr. bit (skip if not bit)

1.6.1.3 The TEST Instruction

There are some instructions that do not set the Z flag if the target register contains zero after execution of such instructions. There are also cases where it is necessary to test if a register contains zero after other instructions have been executed that have modified the Z flag in between. In this case, use the instructions

test w

test fr

to test whether the W register or a file register contains zero. Both instructions set the Z flag in case the specified register contains zero, otherwise, the Z flag is cleared.

The test instruction does not modify the register contents.

1.6.1.4 Multi-Byte-Addition

In case a single-byte-precision is not enough, you can reserve two or more bytes for each operand, and perform the addition in two or more steps. The example below performs a 16-bit addition:

```
org    $08
Op1L  ds 1
Op1H  ds 1
Op2L  ds 1
Op2H  ds 1

Main
mov Op1H, #$10 ; Op1: 0001 0000 1001 1100 = $109c = 4,252
mov Op1L, #$9c ;
mov Op2H, #$30 ; Op2: 0011 0000 0111 1011 = $307b = 12,411
mov Op2L, #$7b ; -----
; + : 0100 0001 0001 0111 = $4117 = 16,663

; Op1 = Op1 + Op2
;
add Op1L, Op2L ; $9c + $7b = $117 (Carry = 1)
addb Op1H, c   ; $10 + $01 = $11
add Op1H, Op2H ; $11 + $30 = $41
```

Beginning at address \$08, we have reserved four bytes for the variables **Op1L**, **Op1H**, **Op2L** and **Op2H**. **Op1H** is used to store the high byte of the first operand, and the high byte of the result. **Op1L** takes the low byte of the first operand and the result. The **Op2H** and **Op2L** variables contain the 16-bit value of the second operand.

To test the addition, we assign constant values to the operands (we have chosen values that result in an overflow in the low byte).

Similar to an "addition by hand", the digits are added from "right to left", i.e. the low order byte comes first, and then the high order byte.

Note the **addb** instruction. It allows to "add" any bit of a register to the contents of another register. As you can see, the assembler supports **c** as pre-defined name for bit 0 in the STATUS register, i.e. the carry flag. **Addb fr. c** is a compound instruction that is replaced by the two instructions

```
snc
inc fr
```

After a possible overflow while adding the low bytes has been handled by the **addb Op1H. c** instruction, we finally add the two high operand bytes.

Similarly, you can add variables using more than two bytes to represent greater values.

Let's make a short excursion to present another method how to address variables that logically belong together, like the two bytes **Op1L** and **Op1H** of the first operand and **Op2L** and **Op2H** of the second operand. Look at the following version of the 16-bit addition:

```
org    $08
Op1    ds    2
Op2    ds    2
```

Main



```
mov    Op1+1, #$10 ; Op1:  0001 0000 1001 1100 = $109c
mov    Op1,    #$9c ;
mov    Op2+1, #$30 ; Op2:  0011 0000 0111 1011 = $307b
mov    Op2,    #$7b ; -----
                                ; Sum:  0100 0001 0001 0111 = $4117
                                ; Op1 = Op1 + Op2
                                ;
add    Op1,    Op2 ; $9c + $7b = $117 (Carry = 1)
addb   Op1+1, c   ; $10 + $01 = $11
add    Op1+1, Op2+1 ; $11 + $30 = $41
```

Here, we have defined two 16-bit variables **Op1** and **Op2**. The instructions refer to the low bytes of the operands by using the "base names", **Op1** or **Op2**, and to refer the high bytes, they use the "base names" plus 1, i.e. **Op1+1** or **Op2+1**.

Here we make use of arithmetic operations, the assembler can perform at assembly-time. Because **Op1** and **Op2** are both symbols that represent addresses (\$08 and \$0a), the assembler replaces the expression **Op1+1** by \$09, and the expression **Op2+1** by \$0b. These two values are the addresses of the operand's high bytes.

1.6.1.5 Subtraction

The **sub** instruction performs 8-bit subtraction operations, and its general syntax is

```
sub op1, op2
```

and it performs the operation

$$op1 = op1 - op2$$

Similar to the add instruction, there are two basic variants:

```
sub fr, w
```

Programming the SX Microcontroller

sub w, fr

and two compound instructions that use w as temporary storage:

sub fr, #Constant

sub fr1, fr2

The C and the Z flag are changed by the sub instructions. The Z flag is set when the result of a subtraction is zero, i.e. when **op1** and **op2** are equal.

If **op1** is greater than or equal to **op2**, the C flag is set, otherwise it is cleared, indicating a "borrow", i.e. if the result is positive, C is set and otherwise it is cleared.

If a result is wrong in case the C flag is cleared depends on how we interpret the result. If we allow for positive values in a range from 0 to 255, negative numbers are not allowed, and we have an error when C is cleared.

1.6.1.6 Signed Numbers

We can interpret the contents of a register as signed number. In this case, the highest bit (bit 7) indicates the sign, and the remaining 7 bits (6...0) are used to represent the value. The table below shows some important values:

decimal	binary	hex
+127	0111 1111	7F
+1	0000 0001	01
+0	0000 0000	00
-1	1111 1111	FF
-128	1000 0000	10

The representation of negative numbers may look strange on the first glance, but on the other hand, it is quite logical. If we decrement a register, containing %0000 0001, i.e. if we calculate 1-1, the result is %0000 0000. If we then decrement the register again, its new contents is %1111 1111 so this must be the equivalent of -1 because the result of 0-1 is -1.

Well, except that bit 7 is set, indicating a negative number, the remaining 7 bits (111 1111) don't look like -1.

If we setup the rule that negative numbers are represented in two's complement format, we come closer to an understanding. To convert a number into its two's complement, negate all bits, and add one to the result. Let's do it for -1:

Value -1: 1111 1111
negation: 0000 0000
plus 1: 0000 0001 = +1

The same conversion for -128:

```
Value -128:    1000 0000
negation:     0111 1111
plus 1:       1000 0000 = +128
```

As you can see, we include the sign bit when doing the conversion in order to get the correct result.

Let's assume we wanted to use the SX to build a pocket calculator that should subtract 30 - 50 as an example. The result that we expect is -20.

The SX performs the calculation as follows:

```
30 = $1e = %0001 1110
-
50 = $32 = %0011 0010
=
$ec = %1110 1100 (C flag is clear)
```

Because bit 7 is set, we know that the result is negative, i.e. we must display a minus sign. Before we can display the digits, it is also necessary to convert the negative number into its positive equivalent by finding its two's complement:

```
; Result contains $1110 1100
;
not Result    ; Result is %0001 0011
inc Result    ; Result is %0001 0100 = $14 = 20
```

We now can display the digits, and the user of our SX-Calculator will see the correct result: -20.

As you can see this is simply a matter of interpretation - the SX itself does not "know" about negative numbers it strictly respects the simple rules of binary arithmetic.

1.6.2 Incrementing and Decrementing

We have used the **inc** and **dec** instructions in examples before, but we want to discuss them here for completeness. The general syntax is:

```
inc fr
dec fr
```

The instructions increment or decrement the specified register, i.e. they add or subtract 1 and set the Z flag in case the register contents is zero after the operation, otherwise, the Z flag is cleared. The instructions do not modify the C flag.

Programming the SX Microcontroller

The combined instructions **incsz** and **decsz** often are useful because they perform an increment or decrement and the test for a zero result. Note that these are basic instructions each requiring one word in program memory only. The instructions have the following syntax:

incsz fr

decsz fr

The specified file register is incremented or decremented. When the register content is zero after the operation, the instruction following **incsz** or **decsz** is skipped. Note that this skipped instruction must not be a compound instruction.

Both instructions do not change the Z and C flags.

Note that none of the increment or decrement instructions can be used to increment or decrement the W register. If you want to do that, use **add w, #1** or **sub w, #1** instead. Keep in mind that these instructions change both, the Z and the C flag.

1.6.3 Arithmetic Instructions and Multi-Byte Counters

Now, that we have discussed the multi-byte addition, we can use it to construct delay loops that make use of numbers greater than 255. To make them work as expected, some points require attention.

The following instructions seem to be correct on the first glance, but they don't work as intended:

```
: Loop
  inc Counter
  addb Counter+1, c
  sz
  jmp : Loop
```

Because the **inc** instruction does not change the C flag, **Counter+1** will never be incremented. The following instructions fix that problem:

```
: Loop
  add Counter, #1
  addb Counter+1, c
  sz
  jmp : Loop
```

Using the **add** instruction instead of an **inc** to increment Counter sets the C flag when **Counter** overflows from 255 to 0.

To generate longer delays, the following instructions seem to be useful:

```
: Loop
  add Counter, #1
  addb Counter+1, c
  addb Counter+2, c
  sz
  jmp : Loop
```

Again, we have a problem here because the generated delay is much shorter than expected.

To find out what's wrong, we should remember that **addb** is a compound instruction. In the code below, we have replaced the **addb** instructions by the instructions that "make" **addb**:

```
: Loop
  add Counter, #1
  snc
    inc Counter+1
  snc
    inc Counter+2
  sz
  jmp : Loop
```

"Adding" C is performed by skipping the **inc** in case C is not set. Now you can see the reason for the problem:

Because an **inc** never changes C, the status of C reflects the result of **add Counter, #1** throughout the rest of the loop, i.e. if **Counter** has overflowed, C is set, and so, both **snc** instructions do not skip. Therefore, **Counter+1** is incremented (as it should), but **Counter+2** is incremented as well when Counter has an overflow, and not only when **Counter+1** overflows.



When you want to test the above program using the debugger to find out the bug, you have the problem that you need to step through the loop until it comes to an overflow. When you execute the program in the animated or "Walk" mode, you would have to stop that mode shortly before the overflow occurs to continue in single steps from there. Both methods are time-consuming.

Here comes the trick: After single-stepping through the loop a couple of times to check if it works correctly in general, you can assume that the SX will execute further loop cycles correctly as well. Now simply set the contents of **Counter** to a value of say \$fe or \$ff. To do this, left-click the location in the debugger window that displays the current contents of **Counter**. This field opens for entry then, and allows you to type in the new value.

Then continue single stepping through the loop and notice how both registers (**Counter+1** and **Counter+2**) are incremented when **Counter** overflows.

The following code finally works as expected:

```
: Loop
  add Counter, #1
  addb Counter+1, c
  snz
    inc Counter+2
  sz
  jmp : Loop
```

Knowing that the **addb** instruction in reality performs an **inc** when C is set, and also knowing that this **inc** sets the Z flag when **Counter+1** overflows to zero, we use **snz** to test the Z flag, and

Programming the SX Microcontroller

increment **Counter+2** if necessary, i.e. when the Z flag is set. In this case, the Z flag is affected by the **inc Counter+2** instruction, and as long as **Counter+2** does not overflow, the **jmp :Loop** instruction is executed.

In case, the **addb Counter+1, c** instruction did not cause an overflow, i.e. the Z flag is clear, **snz** will skip the **inc Counter+2** instruction, and the next instruction is **sz**. As Z still is clear, **jmp :Loop** will be executed.

1.6.4 The DEVICE CARRYX Directive



If you include a **DEVICE CARRYX** directive in the configuration section of a program, the "Carry Extended" bit in the SX fuse register will be set when the program is transferred into the SX. In this case, add and sub instructions work differently.

As you could see in the previous examples, it was necessary to use an extra **addb** instruction to add the overflow to the next higher digit in multi-byte operations.



When the **CARRYX** option is activated, **add** and **sub** instructions automatically add the C flag to the result. Therefore, it is no longer necessary to add an overflow. On the other hand, you must make sure that C has a defined state before you execute a single-byte **add** or **sub**, or before you perform the **add** or **sub** for the first byte of a multi-byte operation.

Use **clc** to clear the carry flag before an **add**, but use **stc** to set the flag before a **sub**. This is also true for **mov** instructions that perform arithmetic operations (we will describe them later in this chapter).

When you do not correctly clear or set the carry flag, it is most likely, that **add** and **sub** instructions return wrong results.

For all the samples in this tutorial, we assume that the **CARRYX** option is *not* active.

1.6.5 Logical Operations

Logical operations are executed on bit-level, but always "in parallel" for all bits in a register.

1.6.5.1 AND

The general syntax of the **and** instruction is

and op1, op2

A logical AND between the bits of **op1** and **op2** is performed, and the result is stored in **op1**. This is the truth table for each bit:

AND		
op1	op2	Result
0	0	0
0	1	0
1	0	0
1	1	1

The result is one and only one when both operand bits are one, or the result is zero if at least one of the operand bits is zero.

The following basic variants of the **and** instruction are available:

and fr, w

and w, fr

and w, #Constant

In addition, the two compound instructions are available that use **w** as temporary storage:

and fr, #Constant

and fr1, fr2

All **and** instructions set the Z flag when after the operation all bits in the target are cleared, otherwise Z is cleared.

To test the status of a single bit in a register, you normally will use an **sb** or **snb** instruction. An **and** instruction allows to test any number of bits in a register with only one instruction, like in the example below:

```
mov w, #%10010011 ; Original: 10010011
and w, #%00001111 ; Mask: 00001111
                  ; Result: 00000011
```

```
sz
  jmp :NotZero      ; jump is executed
```

```
mov w, #%10010000 ; Original: 10010000
and w, #%00001111 ; Mask: 00001111
                  ; Result: 00000000
```

```
sz
  jmp :NotZero      ; jump is not executed
```

This method is also called "masking bits". The constant value specified with the **and** instructions above is the "mask". All bits that are cleared in the mask are "masked out" in the result (marked gray above). This means that it does not matter which status these bits have in **w** because they never can influence the result of the **and** operation.

Programming the SX Microcontroller

As the content of the target register is modified by this operation, it can be used to clear specific bits in the target instead of using **clrb** instructions for several single bits.

1.6.5.2 OR

The general syntax of the **or** instruction is

or op1, op2

A logical OR between the bits of **op1** and **op2** is performed, and the result is stored in **op1**. This is the truth table for each bit:

OR		
op1	op2	Result
0	0	0
0	1	1
1	0	1
1	1	1

The result is zero and only zero when both operand bits are zero, or the result is one if at least one of the operand bits is one.

The following basic variants of the **or** instruction are available:

or fr, w

or w, fr

or w, #Constant

In addition, the two compound instructions are available that use **w** as temporary storage:

or fr, #Constant

or fr1, fr2

All **or** instructions set the Z flag when after the operation all bits in the target are cleared, otherwise Z is cleared.

Or instructions can be used instead of **setb** instructions in order to set several bits at the same time, like in

```
mov  w, 01100000 ; Original: 01100000
or   w, 01001010 ; Mask:    01001010
                    ; Result:   01101010
```

All bits that are set in the constant value specified with the **or** instruction will be set in the target register (marked gray). (Bits that were already set in the target remain set).

1.6.5.3 XOR

The general syntax of the **xor** instruction is

xor op1, op2

A logical EXCLUSIVE-OR between the bits of **op1** and **op2** is performed, and the result is stored in **op1**.

This is the truth table for each bit:

XOR		
op1	op2	Result
0	0	0
0	1	1
1	0	1
1	1	0

The result is one, when one of the two operand bits is one and the other bit is zero, or the result is zero if both operand bits are equal.

The following basic variants of the xor instruction are available:

xor fr, w

xor w, fr

xor w, #Constant

In addition, the two compound instructions are available that use w as temporary storage:

xor fr, #Constant

xor fr1, fr2

All or instructions set the Z flag when after the operation all bits in the target are cleared, otherwise Z is cleared.

Use the **xor** instruction to negate certain bits in a register, i.e. bits previously set will be cleared and vice versa. See the following example:

```

mov  w, 01100010 ; Original: 01100010
xor  w, 01001010 ; Mask:      01001010
                        ; Result:   00101000

```

All bits that are set in the constant value specified with the **xor** instruction will be inverted in the target register. The other bits remain unchanged.

Programming the SX Microcontroller

1.6.5.4 NOT

The general syntax of the **not** instruction is

not op

The bits of the operand are negated, and the result is stored in the operand.

This is the truth table for each bit:

NOT	
op	Result
0	1
1	0

The result is one, when the operand bit is zero, and the result will be zero when the operand bit is one.

The following basic variants of the not instruction are available:

not fr

not w

All not instructions set the Z flag when after the operation all bits in the target are cleared, otherwise Z is cleared.

1.6.6 Rotate instructions

The syntax of the rotate instructions is:

rl fr (rotate left)

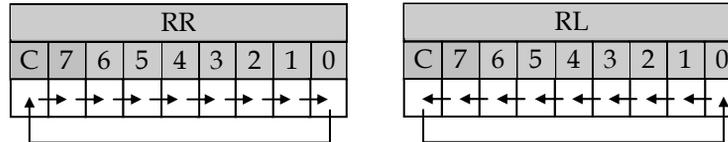
rr fr (rotate right)

The instructions move the bits in a register to the left or to the right by one position.

When **rl** is executed, bit **fr. 0** receives the status of the C flag, and the C flag receives the previous status of **bit fr. 7**.

When **rr** is executed, **bit fr. 7** receives the status of the C flag, and the C flag receives the previous status of **bit fr. 0**.

The following diagrams show the operation of the rotate instructions:



Here, the C flag is used as a "buffer" for the "rotated-out" bit; the Z flag is not changed by these instructions. Before executing a rotate instruction, make sure that the C flag is correctly set or cleared depending on what value shall be "rotated-in".

Rotate instructions are useful to perform multiplications, but also to convert serial to parallel data and vice versa. Programs for scanning LED or keyboard matrices usually make use of rotate instructions as well.

1.6.6.1 Multiplication and Division

The SX does not support special multiplication or division instructions. In order to multiply or divide values, other instructions must be used to build the necessary algorithm.

Let's use the simple multiplication $11 * 4$ as an example to demonstrate different possibilities to solve that multiplication:

The program below uses the method of repeated additions:

```

; =====
; Programming the SX Microcontroller
; TUT014.SRC
; =====
include "Setup28.inc"
RESET    Main
org      $08          ; 1
F1       ds 1         ; 2
F2       ds 1         ; 3
Result  ds 1         ; 4

org      $100        ; 5
Main     ; 6
    mov  F1, #11     ; 7
    mov  F2, #4      ; 8
    clr  Result      ; 9
:MulLoop ; 10
    add  Result, F1  ; 11
    decsz F2         ; 12
    jmp  :MulLoop   ; 13
jmp     $           ; 14

```

In lines 2, 3 and 4, we define three variables, two for the factors, and one for the result.

Programming the SX Microcontroller

In Lines 7 and 8, the two factors, **F1** and **F2** receive the two values 11 and 4, and in line 9, we clear **Result** because we cannot assume that it contains zero.

Within **:Mul Loop**, we add **F1** to **Result** until the contents of **F2**, the other factor is zero. In our example, **F1** is added to **Result** four times. We expect a result of 44. When you debug the program, you will notice a result of \$2c. Convert this into decimal ($2 * 16 + 12 = 44$) to make sure that the program works as expected.

As (almost) always, this simple program bears some cases of trouble:

The case that the multiplication can result in a value of above 255 has been ignored here. We would have to test the C flag after each addition to find out if an overflow occurred.

Before improving the program, let's find out the "worst-case", i.e. what is the maximum result that a multiplication of two bytes could produce:

$$255 * 255 = 65,025 = \$fe01$$

This means a size of two bytes for the **Result** variable is large enough to hold the maximum possible result.

This is the enhanced program:

```
; =====  
; Programming the SX Microcontroller  
; TUT015. SRC  
; =====  
include "Setup28.inc"  
RESET      Main  
  
org        $08          ; 1  
F1         ds 1         ; 2  
F2         ds 1         ; 3  
Result     ds 2         ; 4  
  
org        $100        ; 5  
Main      ; 6  
    mov    F1, #255    ; 7  
    mov    F2, #255    ; 8  
    clr   Result      ; 9  
    clr   Result+1    ; 10  
:Mul Loop ; 11  
    add   Result, F1   ; 11  
    addb  Result+1, c  ; 12  
    decsz F2          ; 13  
    jmp  :Mul Loop    ; 14  
jmp     $             ; 15
```

In this program, we have reserved two bytes for **Result**, and therefore we must clear both **Result** bytes in lines 9 and 10.

Instead of a single byte addition within **:Mul Loop**, we now execute a two-byte addition using the newly inserted **addb Result t+1, c** instruction.

We have initialized both factors to the maximum possible value in order to test the worst-case.

To test the program, run it in full-speed and then perform a halt. You should find the result in the registers at \$0a and \$0b.

We could further enhance the program to allow for 2-byte factors. In this case, we would have to increase the number of result bytes. The rule is simple: The number of bytes required for the result is equal to the sum of bytes in both factors.

Both programs presented so far have another problem: Try to multiply $255 * 0$. The result in this case is not zero as you might expect.

The problem is caused by the fact that on entry into **:Mul Loop**, **F1** is immediately added to **Result t**, i.e. we already have performed the multiplication $F1 * 1$. Then, the **decisz** instruction decrements **F2** (containing 0) down to 255. This finally completes the disaster.

To catch that bug, we should test if **F2** is zero before entering the loop. If this is the case, we are all set, and can skip the loop.

When we asked you to test the multiplication $255 * 255$, we suggested that you would run the program at full speed because we did not want to keep you busy with stepping through the loop 255 times.

This brings us to another point: Although repeated addition is a simple algorithm to multiply two numbers, it can take a while, depending on the contents of **F2** in our case. "A while" means just some microseconds here, but even this may be too long for real-time applications.

For an alternative algorithm, let's take another look at the **rl** instruction, and how it influences the contents of a register initialized to 11 (\$0b):

```
mov Result t, #11 ; $0b = %0000 1011 = 11
clc
rl Result t      ; $16 = %0001 0110 = 22
```

As you know, the status of the C flag is rotated into bit 0 of the target register, it is necessary to clear C before the **rl** as we cannot assume that it is cleared - therefore, never forget the **clc** instruction here.

You can see that after the **rl**, the value in **Result t** has doubled. This is a fact that is quite easy to understand. If we take a decimal number, move all digits one position to the left, and add a zero to the right, we have performed a multiplication by 10 (i.e. with the base of the number system). In case of binary numbers, the base is two, and a left-shift means a multiplication by 2.

Based upon this, it is easy to perform the sample calculation $11 * 4$ when we add another line:

Programming the SX Microcontroller

```
rl Result      ; $2c = %0010 1100 = 44
```

In general, as long as **F2** contains a value that is a power of two, we only need to take care that the result buffer is large enough before applying the **rl** instruction to the result accordingly. Here is an example:

```

; =====
; Programming the SX Microcontroller
; TUT016.SRC
; =====
include "Setup28.inc"
RESET      Main

org        $08
F1         ds 1
F2         ds 1
Result     ds 2

org        $100
Main
    mov     F1, #11
    mov     F2, #4
    clr     Result
    clr     Result+1
    test    F2
    snz
        jmp :Done
    mov     Result, F1
    clc
:MulLoop
    rl     Result
    rl     Result+1
    rr     F2
    sb     F2.0
        jmp :MulLoop
:Done
jmp     $
```

This program multiplies **F1** and **F2**, where **F2** must contain a value that is a power of two (the program does not verify if this is the case, so please don't fool it). The result buffer has a size of two bytes, large enough for two one-byte factors.

Before entering **:MulLoop** we check if **F2** is zero. In this case, we're already done, and the result buffer already contains the correct result (zero).

When **F2** does not contain zero, we first copy **F1** into the result buffer, clear the carry flag, and then enter the loop. While testing the loop, we should keep a close eye on the status of the carry flag.

Inside the loop, we rotate the low result byte to the left. The very first **rl** shifts a zero into bit 0 of the result because the C flag is clear. In case bit 7 of the low result byte was set before, the C flag is set now otherwise, it is clear.

Next, we rotate left the upper result byte. If C is set (because bit 7 of the low result byte was set before), bit 0 of the high result byte is set (which is OK). In any case, the C flag is cleared because the upper result byte contained all zeros due to the initialization.

Next, we rotate **F2** to the right. Because C is cleared, bit 7 of **F2** will be cleared after the **rr** instruction. As we assume that **F2** may only contain values that are equal to a power of 2, we can further assume that exactly one bit in **F2** is set (in our example, this is bit 2 when we enter the loop). The **rr** instruction does the inverse of the **rl** instruction, i.e. we divide **F2** by two whenever we cycle through the loop until the contents of **F2** has reached a value of one (bit **F2.0** is set).

If this is the case, we exit the loop, and the multiplication is finished.

Unfortunately, in real-life, we must assume that **F2** can contain values that are not equal to a power of two, e.g. 12. Nevertheless, we can represent 12 as a sum of powers of two: $12 = 8 + 4$, and instead of $F1 * 12$, we can write $F1 * (8 + 4)$ or $F1 * 8 + F1 * 4$.

In general, it is possible to represent any integer number as a sum of powers of two, e.g. 23:

$$23 = 16 + 4 + 2 + 1 \quad (\text{where } 1 = 2^0)$$

and $F1 * 23$ can be represented as

$$F1 * 16 + F1 * 4 + F1 * 2 + F1 * 1.$$

Then next multiplication example makes use of this rule:

```

; =====
; Programming the SX Microcontroller
; TUT017. SRC
; =====
include "Setup28.inc"
RESET      Main

org        $08
F1         ds 2           ; 1
F2         ds 1           ; 2
Result     ds 2           ; 3
Counter    ds 1           ; 4

org        $100

Main
mov        F1, #255       ; 5
mov        F2, #255       ; 6
clr        F1+1           ; 7
clr        Result        ; 8

```

Programming the SX Microcontroller

```
    clr    Result+1      ; 9
    mov    Counter, #8   ; 10
:Mul Loop
    sb     F2.0          ; 11
    jmp    :Continue     ; 12
    add    Result, F1     ; 13
    addb   Result+1, c    ; 14
    add    Result+1, F1+1 ; 15
:Continue
    clc                    ; 16
    rl     F1              ; 17
    rl     F1+1            ; 18
    rr     F2              ; 19
    decsz  Counter        ; 20
    jmp    :Mul Loop      ; 21
    jmp    $              ; 22
```

This program accepts any values from 0 through 255 for **F1** and **F2**, and stores the result of **F1*F2** in **Result** and **Result+1**, where **Result** contains the low byte, and **Result+1** the high byte. (This order should be familiar to Intel programmers - Motorola friends are free, of course, to change that order).

In line 1, we reserve two bytes for **F1** although the maximum allowed value for **F1** is only 255. The additional byte is required because **F1** will be multiplied by powers of two later, acting as a temporary result buffer.

We have declared a new variable **Counter** in line 4 - we need it to count the eight bits in **F2**.

In lines 5 and 6, we initialize both factors to the maximum worst-case value (255), and lines 7...9 clear the "extension" of **F2** and the result buffer.

Counter is initialized to 8 in line 10. Because **F2** can now contain any number, and not only powers of two, we must test each bit in **F2**, and we can no longer terminate the loop when **F2.0** is set. This is why we need the new **Counter** variable.

Inside **:Mul Loop**, we test if **F2.0** is currently set. If this is the case, we must add **F1**, multiplied with the current power of two to the result. This happens in lines 13...15.

In lines 17 and 18, we multiply **F1** by two in order to have the correct value available for the next possible **add** to the result.

Finally, we divide **F2** by 2 in line 19, and then decrement **Counter**. **:Mul Loop** is executed eight times, once for each bit in **F2**.

As you have seen in the sample programs, an **rr** instruction can be used to divide a value by two. The algorithms shown for multiplication can be similarly used for division by using an **rr** instruction instead of an **rl** instruction, and repeated additions must be replaced with repeated subtractions.

1.6.7 The SWAP Instruction

This instruction is a bit special because it modifies the bits in a register, usually not required for arithmetic or logical operations. The syntax is

swap fr

The instruction exchanges the lower four, and the upper four bits in a register (these groups of four bits are called "Nibbles"). It does not modify any flags.

If, for example a register contains %00001111 before a **swap**, it will contain %11110000 after the **swap**.

1.6.8 The DC (Digit Carry) Flag

The STATUS register contains another flag that is changed by some instructions, the Digit Carry or DC flag. This flag is located in bit 1 of the STATUS register.

An **add** instruction sets this flag, on an overflow from bit 3 to bit 4 (i.e. from the low to the high nibble); otherwise the DC flag is cleared.

A **sub** instruction clears this flag on an underflow from bit 4 to bit 3; otherwise, the DC flag is cleared.

The DC flag is useful when you want to perform BCD arithmetic, i.e. when register contents represent binary coded decimals.

1.6.9 MOV Instructions with Arithmetic Functions

We already have used **mov** instructions in almost all samples so far, and you will find more details about **mov** instructions in the next chapter.

Here, we will discuss such **mov** instructions that do not just copy a value into a target register, but this special class of **mov** instructions also performs arithmetic or logical operations "on the fly" while moving values. This interesting class of instructions can be used for comparisons and other tests where the original contents of a register shall be maintained.

All these **mov** instructions perform an arithmetic or logical operation on a value that is contained in a register (the source), and move the result into **w**, without changing the original contents of the source.



Note that it is necessary to clear or set the carry flag before executing a **mov** that does an addition or subtraction in case the **DEVICE CARRYX** option is active. (See the explanation earlier in this chapter for more details.)

Programming the SX Microcontroller

Now let's describe the `mov` instructions in this class:

```
mov w, /fr    ( w = not fr)
```

Similar to the `not` instruction, the content of `fr` is negated, and the result is stored in `w`. `fr` remains unchanged. The Z flag is set if the result is zero, otherwise it is cleared.

```
mov w, fr-w  ( w = fr-w )
```

The difference between `fr` and `w` is stored in `w`, and `fr` remains unchanged. The Z flag is set if the result is zero, otherwise it is cleared. The C flag is cleared if the result is negative otherwise, it is set. The DC flag will be cleared on an underflow from the high to the low nibble otherwise, it is set.

This instruction, for example, is useful when you want to test if a register contains a specific value, like in

```
mov w, #123  
mov w, Value-w  
sz  
jmp :NotEqual
```

Here, we initialize `w` with the compare value, and then execute the `mov w, Value-w` instruction. If the variable `Value` also contains 123, the result in `w` will be zero, and the Z flag is set. Therefore the `jmp :NotEqual` instruction is skipped in this case.

```
mov w, ++fr  ( w = fr+1 )
```

The incremented content of `fr` is stored in `w`, and `fr` remains unchanged. The Z flag is set if the result is zero, otherwise it is cleared.

This instruction can be helpful when you increment a variable in a loop, and you want to avoid that it overflows:

```
: Loop  
mov w, ++Counter  
mov w, #1  
snz  
mov Counter, w  
jmp : Loop
```

This code is a bit "tricky": The `mov w, ++Counter` instruction sets `w` to zero in case it contains 255, and the Z flag is set. Next, the `mov w, #1` instruction "prepares" a new value for `Counter`. As this `mov` instruction does not change any flags, the status of the Z flag still reflects the result of the previous instruction. The `snz` instruction always skips the `mov Counter, w` instruction, except when Z is set, and this is the case when `Counter` contains 255.

As a result, the content of `Counter` is changed from 255 to 1, and it never yields in a zero value.

```
mov w, --fr  ( w = fr-1 )
```

The decremented content of **fr** is stored in **w**, and **fr** remains unchanged. The Z flag is set if the result is zero, otherwise it is cleared.

mov w, <<fr (w = rl fr)

The left-rotated content of **fr** is stored in **w**, and **fr** remains unchanged. The status of the C flag is copied to bit 0 of the result, and the status of bit **fr. 7** is copied to the C flag.

mov w, >>fr (w = rr fr)

The right-rotated content of **fr** is stored in **w**, and **fr** remains unchanged. The status of the C flag is copied to bit 7 of the result, and the status of bit **fr. 0** is copied to the C flag.



When performing mov and rotate instructions, make sure that the C flag is initialized to a defined state before.

mov w, <>fr (w = swap fr)

The content of **fr** is stored in **w** with the high and low nibbles exchanged, and **fr** remains unchanged. No flags are changed by this instruction.

1.7 MOV Instructions

The mnemonic **mov** is derived from "move". This usually means that something is moved from one place to another, e.g. when you move from one home to another, you belongings located in the old home before the move will be located in the new home after the move.

The SX **mov** instructions act as "data-movers" as well and they also "move" something (the contents of a register (the source register) or a constant value) to a new location (the target register), but they leave the source register unchanged. They actually transfer a copy of the source into the target.

Depending on the type of the **mov** instruction and the resulting target value, the flags are changed or not.

The general syntax of a **mov** instruction is

mov Target, Source

this means the move goes from "right to left". The right instruction argument defines the source, and the left argument specifies the target.

Some **mov** instructions are basic SX instructions, where others are compound instructions. Let's address the basic instructions first:

1.7.1 Basic MOV Instructions

mov fr, w

This instruction copies the contents of **w** to the specified file register **fr**. No status flags are changed by this instruction.

mov w, fr

This instruction copies the contents of the specified file register **fr** to **w**. When **w** contains zero after the mov, the Z flag is set otherwise, it is cleared.

mov w, #Constant

This instruction sets the contents of **w** to the specified constant value. No status flags are changed by this instruction.



Never forget to place the leading hash symbol "#" in front of the constant value (or label). If it is missing, the assembler creates a **mov w, fr** instruction instead, i.e. **w** will receive the contents of a register with the address specified by **Constant**, but not the constant value itself.

mov w, m

This instruction copies the contents of the MODE (**m**) register to **w**. No flags are changed by this instruction. The value in **m** specifies which group of the port configuration registers is currently active.

mov m, w

This instruction copies the contents of **w** to the MODE (**m**). No flags are changed by this instruction. The value in **m** specifies which group of the port configuration registers is currently active.

mov m, #Constant

This instruction sets the contents of the MODE register (**m**) to the specified constant value. No status flags are changed by this instruction.



Never forget to place the leading hash symbol "#" in front of the constant value (or label). If it is missing, the assembler creates a **mov m, fr** instruction instead, i.e. **m** will receive the contents of a register with the address specified by **Constant**, but not the constant value itself.

mov !option, w

This instruction copies the contents of **w** to the OPTION register. We will discuss the OPTION register together with interrupts. No status flags are changed by this instruction.

mov !Port, w

This instruction copies the contents of **w** to the specified port configuration register (**!ra, !rb, !rc**, etc.). No status flags are changed by this instruction. Make sure that the MODE (**m**) register is correctly set before in order to select the desired group of port configuration registers.

1.7.2 Compound MOV Instructions

In addition to the basic **mov** instructions described before, the assembler supports a set of compound **mov** instructions. The assembler generates two basic **mov** instructions as replacements for the compound **mov** instructions.



Because the compound **mov** instructions use **w** as temporary storage, be aware that the previous content of **w** is lost!

As compound statements are replaced by two basic instructions, a compound **mov** *may never follow immediately a conditional skip instruction!*

Programming the SX Microcontroller

mov fr, #Constant

This instruction sets the contents of a register to the specified constant value. The status flags are not changed by this instruction, and **w** contains the constant value.



Never forget to place the leading hash symbol "#" in front of the constant value (or label). If it is missing, the assembler generates a **mov fr1, fr2** instruction instead, i.e. **fr** will receive the contents of a register with the address specified by **Constant**, but not the constant value itself.

mov fr1, fr2

This instruction copies the contents of register **fr2** to register **fr1**. If **fr2** contains zero, the Z flag is set, otherwise it is cleared. **w** is set to the contents of **fr2**.

mov fr, m

This instruction copies the contents of the MODE register (**m**) to the register **fr**. The flags are not changed by this instruction. **w** is set to the contents of **m**.

mov m, fr

This instruction copies the contents of the register **fr** to the MODE register (**m**). The flags are not changed by this instruction. **w** is set to the contents of **fr**.

mov !option, fr

This instruction copies the contents of register **fr** to the OPTION register. If **fr** contains zero, the Z flag is set, otherwise it is cleared. **w** is set to the contents of **fr**.

mov !option, #Constant

This instruction sets the OPTION register to the specified constant value. The flags are not changed by this instruction. **w** is set to the constant value.



Never forget to place the leading hash symbol "#" in front of the constant value (or label). If it is missing, the assembler generates a **mov !option, fr** instruction instead, i.e. the OPTION register will receive the contents of a register with the address specified by **Constant**, but not the constant value itself.

mov !Port, fr

This instruction copies the contents of register **fr** to the specified port configuration register (**!ra, !rb, !rc**, etc.). If **fr** contains zero, the Z flag is set otherwise, it is cleared. **w** is set to the contents of **fr**.

mov !port, #Constant

This instruction sets the specified port configuration register (**!ra**, **!rb**, **!rc**, etc.) to the specified constant value. The flags are not changed by this instruction. **w** is set to the constant value.



Never forget to place the leading hash symbol "#" in front of the constant value (or label). If it is missing, the assembler generates a **mov !Port, fr** instruction instead, i.e. the configuration register will receive the contents of a register with the address specified by **Constant**, but not the constant value itself.

Instead of using compound **mov** instructions, you can always combine basic **mov** instructions to achieve the same effects. Sometimes, it can even save clock cycles and program memory when you keep in mind that compound instructions are combined basic instructions, like in the following example:

```
mov Factor1, #44
mov Factor2, #44
```

When we write the basic instructions that "make" the compound instructions, the sample looks like this:

```
mov w, #44
mov Factor1, w
mov w, #44
mov Factor2, w
```

As you can see now, the second **mov w, #44** is not necessary because **w** already contains 44, so that the instructions

```
mov w, #44
mov Factor1, w
mov Factor2, w
```

have the same effect. Of course, you could also write

```
mov Factor1, #44
mov Factor2, w
```

but this makes the program quite difficult to understand, especially for someone who does not know about the internals of compound statements. The following context completes this:

```
mov w, ra
mov PortStatus, w
mov Factor1, #44
mov Factor2, w
```

Now, someone might assume that **Factor2** receives the value read from **ra** into **w** because it is not obvious that the **mov Factor1, #44** instruction modifies **w**.

1.8 Recognizing Port Signals

1.8.1 Recognizing Signal Edges at Port B

To test the examples in this chapter, a pushbutton is required that is connected between port pin RB3 of the SX and V_{SS} . Most of the commercially available demo boards have this pushbutton already installed.

We will enhance the sample program shown in the introduction in a way that the LED blinks faster when the pushbutton is pressed. This is the enhanced program:

```
; =====  
; Programming the SX Microcontroller  
; TUT018. SRC  
; =====  
include "Setup28.inc"  
RESET    Mai n  
  
TRIS     equ $0f  
PLP      equ $0e  
  
org $08  
Counter1 ds 1  
Counter2 ds 1  
Counter3 ds 1  
Ti me    ds 1  
  
org $000  
Ti meEater  
mov      Counter1, Ti me  
:Loop  
    setb  Ti me. 5  
    sb   rb. 3  
    clrb Ti me. 5  
    decsz Counter3  
    jmp  :Loop  
    decsz Counter2  
    jmp  :Loop  
    decsz Counter1  
    jmp  :Loop  
    ret  
  
org $100  
Mai n  
    mov  Ti me, #$30  
    mode PLP  
    mov  !rb, #%11110111  
    mode TRIS  
    mov  !rb, #%11111110  
Loop  
    call Ti meEater  
    clrb rb. 0
```

```

call   TimeEater
setb   rb. 0
jmp    Loop

```

In this program version, we have defined a new variable **Time** that defines the current LED blink frequency, and the main program initializes it to \$30.

The **TimeEater** subroutine now initializes **Counter1** with the contents of **Time**, and not with a constant value.

1.8.1.1 MODE and Port Configuration Registers



(2.2.4.12 - 218) The contents of the MODE (**m**) register defines which port configuration register is accessed by the **mov !r?, ???** instruction.

After a reset, the port direction (TRIS) registers are selected by default. In the main program, we first set **m** to select the Pull-up Configuration registers with the **mode PLP** instruction. We have defined constants for the necessary values for **m** at the beginning of the program. This is easier than keeping all that values in mind.

The **mov !rb, #%11110111** instruction clears bit 3 in the Pull-up register, i.e. a pull-up resistor is activated for port pin **RB. 3**, and this is the pin with the pushbutton. Without a pull-up resistor, the port input will "float" as long as this pushbutton is open, i.e. its state would be undefined. But as the pull-up resistor pulls the input level up to V_{DD} , the port bit will read 1 in this case.

In the inner delay loop, we read port bit **rb. 3**, i.e. the status of the pushbutton, and we change the value in **Time** accordingly:

```

setb   Time. 5
sb     rb. 3
clr    Time. 5

```

First, we set bit 5 in **Time**, i.e. it contains \$30. If **rb. 3** is set, the pushbutton is open, and in this case, we don't change **Time**. If the pushbutton is down, bit **Time. 5** is cleared, i.e. **Time** now contains \$10. Therefore, **TimeEater** generates a shorter delay and the LED will blink faster.



As you can see, bit **Time. 5** is set each time when the program executes the loop, although it is only necessary to set the bit when the pushbutton is open. On the other hand, this would require an additional test, and a **jmp** instruction that requires more execution time and additional program memory than setting **Time. 5** "in preparation".

Run the modified program at full speed, and try to change the blink frequency by pressing the push button. You will notice that this is not that easy. In order to successfully change the frequency, you must press the button while the LED is on, and you must release it while the LED is off. As long as the LED blinks slowly, you should manage it but if the LED blinks quickly, you need a "fast finger".

The reason for this problem is that the button read is "static", i.e. we don't recognize the fact that the button state has changed, Instead we query the button state as it currently is. When you hold it down too long, this state will be read whenever **TimeEater** is called, and this causes a toggle of the bit **Time. 4** as long as you keep the button pressed.

We could fix this problem by adding more instructions that save the last button state and toggle the "speed bit" **Time. 4** only if the button state is different from the previously saved state. On the other hand, why should we do this if the SX has this functionality "built in"?

1.8.1.2 Signal Edges at Port B



(2.2.4.9 - 214) Port B is equipped with some additional registers that can be accessed when the **MODE (m)** register is set to certain values. One of these registers is named **WKPND_B** (Wake-up Pending). A bit in this register is set, when the associated port pin has registered a signal change. After a reset, negative edges, i.e. high-low transitions, set the bits by default.

There is another configuration register for Port B that allows changing the default. If you clear a bit in the **WKED_B** (Wake-up Edge) register, the input is configured to set the associated bit in the **WKPND_B** register on a positive edge, i.e. a low-high transition.

A bit in the **WKPND_B** register signaling a detected edge on the associated port input pin, remains set until it is cleared by software. Then it is "armed" to indicate the next signal edge that might occur on that input.

To clear the bits in the **WKPND_B** register, a **mov !rb, w** instruction is used where usually **w** is cleared before in order to reset all bits in the **WKPND_B** register. The "trick" here is that this **mov** instruction actually does an exchange. This means the content of **w** is copied into the **WKPND_B** register, where the previous content of this register is copied to **w**. This makes it possible to read the contents of **WKPND_B** although this register "officially" is write-only.

Because the primary use of these registers is to control how signal edges shall wake up a "sleeping" SX, they contain the "Wake-up" part in their names. Besides this, the registers also control interrupts, but they can also be used directly to test for signal edges. (We will address wake-up and interrupts later).

Now let's try a sample program that makes use of these features:

Programming the SX Microcontroller

```

; =====
; Programming the SX Microcontroller
; TUT019.SRC
; =====
include "Setup28.inc"
RESET    Main

TRIS     equ $0f
PLP      equ $0e
WKPND_W  equ $09

org $08
Counter1 ds 1
Counter2 ds 1
Counter3 ds 1
Time     ds 1
Button   ds 1

org $000
TimeEater
    mode    WKPND_W
    clr     w
    mov     !rb, w
    and     w, #%00001000
    snz
        jmp :NoButton
    xor     Time, #%00100000
:NoButton
    mov     Counter1, Time
:Loop
    decsz   Counter3
        jmp :Loop
    decsz   Counter2
        jmp :Loop
    decsz   Counter1
        jmp :Loop

    ret

org $100
Main
    mov     Time, #$30
    mode    TRIS
    mov     !rb, #%11111110
    mode    PLP
    mov     !rb, #%11110111
    mode    WKPND_W
    clr     w
    mov     Button, w
    mov     !rb, w

Loop
    call    TimeEater
    clrb   rb.0
    call    TimeEater

```

```

setb   rb.0
jmp    Loop

```

At the beginning of this program, we have defined a new constant **WKPND_B**. This is the value that is required in the MODE register to access the **WKPND_B** register in order to exchange its contents with **w**.

In the main program, we clear the **WKPND_B** register to avoid that an action is executed after a reset because at reset, all bits in this register are set.

We have also declared a new variable **Button** that is cleared in the main program. We don't need this variable here, but we'll use it for an enhancement soon.

```

TimeEater
mode   WKPND_B
clr    w
mov    !rb, w
and    w, #%00001000
snz
      jmp :NoButton
xor    Time, #%00100000
:NoButton

```

This part of the subroutine performs the actions necessary detecting a "button down" event. The **clr w** and **mov !rb, w** instructions clear the **WKPND_B** register, and **w** holds the previous contents of the register. Note that the **mov !rb, w** instruction does not alter the Z flag, therefore, we cannot test it to see whether the contents of **w** is zero.

Instead, we mask out bit 3 (**and w, #%00001000**). When bit 3 is not set, i.e. no "button down" event, the jump to **:NoButton** is executed. In the other case, the **xor Time, #%00100000** instruction toggles bit 5 in **Time** in order to select the fast or slow blink frequencies for the LED, i.e. **Time** either contains \$30 or \$10.

When you run the program at full speed, you will notice that the toggle between the two blink speeds does not always work as expected. This is caused by contact "bouncing", i.e. before the pushbutton contact finally stays closed when you press and hold it down, it opens and closes a couple of times (within a time period in the area of some milliseconds). Because the SX is that fast, the program interprets the bounce as separate button "down events" where each of them toggles the **Time** variable contents.

1.8.1.3 De-bouncing Mechanical Contacts

There are several methods used to eliminate the bouncing of mechanical contacts. One idea is to use an RC network to "smoothen" the contact signal. Another idea is to use a pushbutton with a make and a brake contact, each connected to one port input (we will show this in a later application example). Another method is to wait a certain period after the first button-down has been

Programming the SX Microcontroller

detected. If the button is still down after that period, we can assume that the button is "really" down.

We are going to use this method in an enhanced version of the **TimeEater** subroutine:

```
Ti meEater
  mode   WKPND_B
  clr    w
  mov    !rb, w
  mov    Button, w
  mov    Counter1, Ti me
: Loop
  decsz  Counter3
  jmp    : Loop
  sb     Button. 3
  jmp    : Conti nue
  clr    Button
  sb     rb. 3
  jmp    : Conti nue
  xor    Ti me, #%00100000
: Conti nue
  decsz  Counter2
  jmp    : Loop
  decsz  Counter1
  jmp    : Loop
  ret
```

At the beginning of **TimeEater**, we read the **WKPND_B** register as before, and save the result in the **Button** variable that we are going to use now. In case of a button-down event, bit **Button. 3** is set now. After finishing the inner delay loop, we now test if **Button. 3** is set. In this case, we clear the **Button** variable to prepare it for the next event. Then we read port **bit rb. 3** directly to find out if the button is still down, i.e. if this bit is clear. In this case, we assume that the button is "really down", and toggle bit **Time. 5** as before.

Here, we make use of the inner delay twice - it serves as timer for the de-bouncing delay and for the blink delay.

If you are using a pushbutton that produces very long bounces, it may happen that the delay provided by the inner loop is too short. In this case, simply move the instructions "behind" the middle delay loop that decrements **Counter2** or even "behind" the outer delay loop:

```
  decsz  Counter2
  jmp    : Loop
  decsz  Counter1
  jmp    : Loop

  sb     Button. 3
  jmp    : Conti nue
  clr    Button
  sb     rb. 3
  jmp    : Conti nue
  xor    Ti me, #%00100000
```

: Continue

Be aware that this makes the program react much slower on a button push, i.e. if you press it too short, the button-down will not be recognized.

1.9 Interrupts - The OPTION Register

1.9.1 Interrupts

The sample programs in the previous chapter made an LED blink at a low frequency. In order to do this, it was necessary to "slow-down" the SX using nested delay loops.

In reality, the SX has been designed to perform more important tasks than just make an LED blink. "Blinking an LED" is one of those tasks, the SX can easily handle "in the background" while taking care of other things like serial communications, A/D conversion, reading switches or buttons, frequency and time measurement, speed control, Internet communications, etc.

In theory, it would be one solution to insert code within the inner delay loop to handle other tasks. In the previous example, we did this to read the button state. You could also call subroutines from there to handle other tasks.

Often, handling other tasks requires a variable number of clock cycles, i.e. the delay caused by the loop "around" such task handlers would no longer be constant. This might be acceptable for a blinking LED but not for a program, that transmits serial data at a high baud rate. Here, precise timing is essential.

The scenario described above is a typical "multi-tasking" scenario, i.e. the SX must be able to perform several tasks "at the same time". Of course, the SX can only execute one instruction after another, but if you make sure that each task is assigned a certain "slice" of execution time, it looks as if the SX is performing several tasks in parallel.

This important feature is the basis for Virtual Peripherals that make the SX so uniquely different from other microcontrollers. We will address Virtual Peripherals in a later chapter of this tutorial.

In addition, there might occur random events that do not match a pre-defined time frame, like a button press or the edges of a square wave with varying pulse length, etc.

The magic word that describes the method to solve such requirements is "Interrupt". If a certain event occurs, the SX interrupts the current sequence of instructions, executes a special subroutine that handles the event, and then continues with executing the previously interrupted sequence.

Similar to a subroutine, the return address is saved internally, i.e. the address of the instruction that comes next to the last one executed before the interrupt. On return from the interrupt, this address is restored to the program counter.

You certainly will recall from the chapter dealing with subroutines, that it is important to make sure that the subroutines do not change register contents that are required by the calling program. We have shown some techniques how to save and restore the contents of the FSR there. When dealing with interrupts, the problem is that an interrupt can occur at any time, and there is

no chance to save important register contents before entering the interrupt handler. This means that the interrupt handler would have to save and restore important registers like W, STATUS and FSR.

Fortunately, the designers of the SX have built in so called "shadow registers" for W, STATUS and FSR that always hold duplicates of the original register's contents. While handling an interrupt, the shadow registers are no longer updated, so that they hold the contents of the registers as they were before the interrupt. When interrupt handling is finished, the contents of the shadow registers are used to restore the original values in W, STATUS and FSR without any additional code in the application program.

Another problem might occur when - while one interrupt is serviced - another interrupt is triggered. Because the SX can always only handle one interrupt at a time, further interrupts must be disabled until the previous interrupt has been serviced. Again, this is automatically handled by the SX, so there is no need to provide special instructions for that.

To handle an interrupt, a special subroutine is called, and the SX assumes that this routine begins at address \$000 in program memory. This subroutine is also called the Interrupt Service Routine (ISR). To return from an ISR, you cannot use a "regular" **ret** instruction, but you must use **reti** or **retiw** instead. These instructions make sure that the register contents are restored from the shadow registers as described before, and that the program counter PC is loaded with the correct return address.

The **reti** (Return from Interrupt) instruction returns from the ISR as described before. The **retiw** (Return from Interrupt with W) performs an additional very useful operation that we'll explain soon.

In the beginning of this chapter, we mentioned that several events can be the reason for an interrupt:

- Asynchronous events, e.g. signal edges at a port B input caused by a button press.
- Synchronous events, i.e. interrupts that are generated whenever a certain time has elapsed.
- Interrupts after a certain number of (synchronous or asynchronous) events, or interrupts caused by a counter overflow. Here an external signal is fed into an SX input. A signal edge increments an internal counter. When this counter overflows, an interrupt is issued.



In order to enable interrupts, bit 6 in the OPTION register must be cleared, therefore programs must contain an **OPTIONX** or **STACKX** directive to allow write access to this bit.

Programming the SX Microcontroller

1.9.1.1 Asynchronous Interrupts



(2.2.4.9 - 214) In the previous chapter, we have made use of the **WKPND_B** register at Port B that allows detecting signal edges at any of the eight Port B inputs. In addition, each of these inputs can be configured to cause an interrupt when the configured signal edge (positive or negative) is detected.

In order to have one of the Port B inputs cause interrupts, it first must be configured as an input, i.e. the associated bit in the **TRIS_B** register must be set (this is the default after a reset).

In addition, the associated bit in the **WKED_B** register must be set when a negative signal edge shall trigger the interrupt (this is the default after a reset). To have a positive edge trigger the interrupt, the bit must be cleared.

So far, the setup of the Port B registers is identical to the steps we have taken in the last chapter. In addition, we must now clear the associated bit in the **WKEN_B** (Wake-up Enable) register to enable the interrupt.

As an example for an asynchronous interrupt, we use the "LED-Blinker" application with some modifications and enhancements. To test this program, the LED should be connected to RB0, and the pushbutton to RB3 as before.

The task of this program is to keep the LED blinking in the "foreground" while detecting a button-down state, and the necessary de-bouncing shall be done in the "background", i.e. the foreground task shall not "notice" that "something happens in the background".

```
;  
; =====  
; Programming the SX Microcontroller  
; TUT020. SRC  
; =====  
include "Setup28.inc"  
RESET      Main  
  
TRIS       equ $0f  
PLP        equ $0e  
WKEN       equ $0b  
WKPND_W    equ $09  
  
org        $08  
  
Counter1   ds 1  
Counter2   ds 1  
Counter3   ds 1  
Time       ds 1  
Bounce     ds 2  
local Temp0 ds 1  
  
org $000
```

```

;-----
; ISR
;-----
    mov     localTemp0, m
    clr     Bounce
    clr     Bounce+1
:DeBounce
    decsz   Bounce
    jmp     :DeBounce
    decsz   Bounce+1
    jmp     :DeBounce
    mov     w,    #%00100000
    sb     rb. 3
    xor     Time, w
    mode    WKPND_W
    clr     w
    mov     !rb, w
    mov     m, localTemp0
    reti

TimeEater
    mov     Counter1, Time
:Loop
    decsz   Counter3
    jmp     :Loop
    decsz   Counter2
    jmp     :Loop
    decsz   Counter1
    jmp     :Loop
    ret

org $100

Main
    mov     Time,    #$30
    mode    TRIS
    mov     !rb,    #%11111110
    mode    PLP
    mov     !rb,    #%11110111
    mode    WKPND_W
    clr     w
    mov     !rb,    w
    mode    WKEN
    mov     !rb,    #%11110111

Loop
    call    TimeEater
    clr     rb. 0

    call    TimeEater
    setb   rb. 0
    jmp    Loop

```

Programming the SX Microcontroller

In this program version, we configure input RB3 to trigger an interrupt at a positive signal edge:

```
mode   WKEN_B
mov    !rb, #%11110111
```

To avoid that a set bit in the **WKPND_B** register triggers an interrupt immediately after reset, we clear the **WKPND_B** register before enabling the interrupt.

The ISR begins at the fixed address \$000, and it first saves the contents of the MODE register.



As the SX does not automatically save and restore the MODE register, it is important to do this within the ISR in case the ISR is about to change the contents of MODE. Our example program would even work without saving and restoring **m** because the main program does only access the port configuration registers before enabling the interrupt, but you should not forget to build in this safety measure in a generic ISR.

After saving **m**, the ISR generates a time-delay to de-bounce the button, and then reads the button input at **rb. 3** to find out if the button is still down. If so, bit **Time. 5** is toggled, similar to the previous program version. Next, the **WKPND_B** register is cleared.



It is very important to clear the **WKPND_B** register inside the ISR. If you fail to do so, another interrupt will be triggered immediately after the ISR returns and this means that there is no more time available for the main program.

Note that a bit in the **WKPND_B** register is also set when a port pin is configured as an output, and its state changes in the direction that has been configured in the **WKED_B** register. Therefore, you usually would not clear the associated bit in the **WKEN_B** register to avoid that output state-changes cause interrupts.

If you like, you may test this: Comment out the `mov !rb, w` instruction in the ISR, and activate the debugger. First, reset the SX, then press the pushbutton for a short time, and single-step through the code.

Finally, the ISR restores **m** and returns with **reti**.

Note that the program execution "stays" in the ISR for a while because of the delay loop inside that is used to de-bounce the pushbutton. In a "real-life" application, this is not very elegant because too much processing time might be "stolen" from the main program. We'll discuss better solutions in the following sections.

1.9.1.2 Synchronous (Timer-Controlled) Interrupts



(2.2.5.1 - 222) In the previous sample program, the main program handled the task to blink the LED, but this is a typical "background" task to be handled by an ISR.

The SX offers an excellent method to deal with synchronous (or timer-controlled) interrupts. Essentially, this feature is what makes the powerful concept of Virtual Peripherals possible!

Please enter the following program, and then test it by using the debugger in single-step mode:

```

; =====
; Programming the SX Microcontroller
; TUTO21. SRC
; =====
include "Setup28.inc"
RESET    Main

org $000
; -----
; ISR
; -----
    inc $09
    reti

Main
    mov rtcc, #$fa
    mov !option, #%10001000
: Loop
    inc $08
    jmp : Loop

```

In the main program, first, the RTCC register is initialized to \$fa, and then the OPTION register is set to \$88 (you will soon see what this means).

See how RTCC starts counting up after you have executed the third **inc \$08** instruction for the first time. Continue single stepping, until RTCC changes from \$ff to \$00, and see how the program flow changes into the ISR code that increments the register at \$09.

When you execute the **reti** instruction, verify that the program execution is continued with the next instruction in the main program (the last **inc \$08** before **jmp : Loop**).

The RTCC Register (Real Time Clock Counter) is incremented at each system clock cycle. The **inc** instructions require one clock cycle, i.e. each time an **inc** is executed, RTCC is incremented once. The **jmp** requires three clock cycles, i.e. each time a **jmp** is executed, RTCC is incremented by three. Also note, that jumping to the entry point of the ISR at \$000, and the **reti** each require

Programming the SX Microcontroller

three clock cycles. When you execute the program in animated or "Walk" mode, you can see, how the ISR is periodically called and how the contents of \$09 increments for each interrupt.



If the RTCC overflows within the three clock cycles required for the **jmp** at the end of the main **: Loop**, some debuggers do not execute the ISR. This is a problem with the debugger - you can be sure that the SX really performs an interrupt here when it runs at full speed.

To enable interrupts triggered by RTCC overflows, you must load the OPTION register with %10001000 or \$88. We will discuss the meaning of the OPTION bits later in more detail.

At the beginning of the main program, we have initialized the RTCC register to \$fa. Usually, it is not necessary to initialize this register. We have done it here in order to have an interrupt triggered after a few single steps to make it easier for you. The **inc** instructions in the main program and in the ISR have been added to keep the SX "busy" somehow. In a real program, these will be replaced by instructions that are more meaningful.

When we assume a 50 MHz system clock, it is easy to determine the time interval between two ISR calls. The RTCC register overflows every 256 clock cycles of 20 ns length each. This means that an interrupt is triggered every $256 * 20 \text{ ns} = 5.12 \mu\text{s}$.

For applications that require an exact timing, it is important that the interval between two ISR calls can be defined by the program. As you can see in our sample program, we have forced the first interrupt to come up earlier by initializing RTCC to \$fa. In order to obtain shorter interrupt periods, it is an idea to initialize RTCC to a certain value before returning from the ISR. You can test this by inserting the **mov rtcc, #fa** instruction immediately before the **reti** instruction. Single-step the modified program, and see what happens.

Let's assume that we want an interrupt every microsecond at a system clock frequency of 50 MHz. This means that an interrupt must be triggered every 50 clock cycles. To achieve this, it is not sufficient to load RTCC with $256 - 50 = 206$ at the end of the ISR. You must keep in mind that the instructions within the ISR already have taken clock cycles while RTCC was incremented, and three more clock cycles were required to enter the ISR. Another three clock cycles are "stolen" to return from the ISR.

In order to find out the correct RTCC initialization you would have to sum all these clock cycles, and whenever you make a change in the ISR code, you would have to re-calculate the sum. Things get even more complicated when the execution time of the ISR instructions is not constant. This can easily happen when conditional skips are involved. Fortunately, there is no need to do all this manually because the SX keeps track of the clock cycles "used" within the ISR. Because the RTCC keeps incrementing after its overflow has triggered the interrupt, its contents at the end of the ISR exactly holds the number of clock cycles "used" so far.

Therefore, we take the number of clock cycles that shall elapse between two interrupts (n_C), subtract this value from the current contents of RTCC in order to get the correct initialization value for RTCC.

To understand this, assume for a moment that the ISR did not "use" any clock cycles, and that the call and return do not require any clock cycles either. This would mean that RTCC contains zero, i.e. the "magic value" that has triggered the interrupt. In order to have the next interrupt triggered after RTCC has been incremented for n_C cycles, it is necessary to decrement it now by n_C . This is why we subtract n_C .

In reality, less than n_C cycles are required until the next interrupt to be "just in time" because the ISR really did "use" clock cycles. As mentioned before, this cycle count is contained in RTCC and so the correction is done automatically.

There is even more good news: The **reti** instruction has been included to do exactly that! Similar to **reti**, **reti w** returns from an interrupt but it also adds the contents of **w** to the RTCC.

In order to store $(RTCC - n_C)$ to the RTCC, we just need to store $-n_C$ to **w** before executing the **reti w** instruction.

To achieve an interrupt period of $1\ \mu\text{s}$ (50 clock cycles), for example, just write the following code:

```
mov w, #-50  
reti w
```

Congratulations to the SX development team - this "little trick" is one of the features that make the SX so powerful!



As already mentioned before, at return from an interrupt, the contents of **w** and other registers are restored. Even though **w** is now used to return the RTCC initialization, the former contents of **w** is still restored before the interrupted program is continued, so don't worry.

When you define an interrupt period of n_C clock cycles, you must make sure that the total cycles for instructions within the ISR does not exceed a value of n_C-7 . If this is the case, interrupt calls get lost, i.e. RTCC will overflow again while the previously triggered ISR is still active. The value n_C-7 already considers the so-called "latency time" that is required to read in new instructions into the instruction pipeline of the SX.

Programming the SX Microcontroller



It is a good advice to always determine the "worst-case" number of clock cycles required by an ISR that handles synchronous interrupts in order to make sure that this value is less than $nC-7$, where nC specifies the number of clock cycles between two interrupts.

Also, note that small values of nC take away more processing time from the mainline program. Because some instructions, like a `jmp`, require three clock cycles, this is the minimum number of cycles that must be available in the mainline program.

1.9.1.3 The Prescaler

As mentioned before, at 50 MHz system clock, the maximum time interval between two interrupts can be $5.12 \mu\text{s}$ if you just use the RTCC to count clock cycles.

In order to obtain longer interrupt intervals, you can make use of the integrated prescaler that can be "hooked" between the system clock and the RTCC.

We make use of the prescaler in the program below (assuming that an LED is connected to RB0):

```

; =====
; Programming the SX Microcontroller
; TUT022.SRC
; =====
include "Setup28.inc"
RESET    Main

TRIS     equ $0f

org      $08
Timer    ds 1
Counter  ds 1

org      $000
; -----
; ISR
; -----
    decsz Timer
    reti
    xor    rb, #%00000001
    reti

Main
    mode  TRIS
    mov   !rb, #%11111110
    mov   !option, #%10000111
: Loop
    inc   Counter
    jmp  : Loop
```

In this program, we initialize the OPTION register with %10000111, i.e. bit 3 (PSA - Prescaler) is cleared. This “installs” the prescaler between the system clock and the RTCC. The lower three bits in OPTION determine the divide-by factor of the prescaler. When all three bits are set, this factor is 1:256.

Thus, the interval between two interrupts is now

$$20 \text{ ns} * 256 * 256 \approx 1.3 \text{ ms.}$$

Because we again want to blink an LED, this interval is too short. If we increase it by another factor of 256, we end up at a period of 0.34 seconds, and this is an acceptable value.

Therefore, we have defined the **Timer** variable that is decremented by the ISR. When **Timer** finally reaches zero (after 256 interrupts), **xor rb, #00000001** is executed that toggles the level at **rb. 0**, turning the LED on or off.

Compared to the previous programs, this version is much smaller, and the mainline program no longer needs to take care of turning the LED on or off. Currently, the mainline program does nothing else but incrementing **Counter** (we inserted this here to make the mainline program “think” that it is at least “good for something”).



A “stylistic” comment: In this program, the ISR contains two **reti** instructions. This is against “good programming style” where a routine should have just one entry-point and one exit-point only. However, when you do real-time programming, it is often important to get a task done as fast as possible, therefore, whatever is faster (but still is save) is OK.

Sometimes, this causes code to look a bit “cryptic” - in this case, meaningful comments help you and others to understand it later.

In the next program sample, we will make use of a timer-controlled interrupt to blink the LED and to read and de-bounce a pushbutton that shall toggle the blink frequency. This time, we will not make use of the Port B edge-detection feature.

Let's discuss the timing first, before writing the program (this should be the “usual” approach):

Without the prescaler, the maximum delay between two interrupts can be 5.12 μs at 50 MHz system clock. Let's use a time-base of 2 μs here, which is equivalent to 100 clock cycles.

For de-bouncing we'll wait about 40 ms after a detected button-down until we accept a pressed button. This is equivalent to 20,000 ISR calls. For this value, we need a 16-bit counter or a nested loop with two 8-bit counters. With the “inner” counter covering the full range of 256, the outer counter must count down from 20,000 / 256 (≈ 78) to zero.

Programming the SX Microcontroller

By multiplying the 40 ms-interval by ten, we achieve a 400 ms-interval to toggle the LED resulting in a slow blink frequency of $1/(2 * 400 \text{ ms}) = 1.25 \text{ Hz}$. For the fast blink, we multiply the 40 ms-interval by 5, which results in a frequency of 2.5 Hz.

The program below uses these values. It has some more "bells and whistles" that we will discuss:

```
; =====
; Programming the SX Microcontroller
; TUT023.SRC
; =====
include "Setup28.inc"
RESET      Main

TRIS       equ $0f
PLP        equ $0e

org        $08
Time       ds 1 ; 1
Timer256  ds 1 ; 2
Timer78   ds 1 ; 3
TimerLED  ds 1 ; 4
Counter   ds 1 ; 5
ISRState  ds 1 ; 6

org        $000
; -----
; ISR
; -----
decsz Timer256 ; 7 1/2
  jmp :ExitISR ; 8 3
decsz Timer78 ; 9 1/2
  jmp :ExitISR ; 10 3
mov Timer78, #78 ; 11 2

mov w, ISRState ; 12 1
jmp pc+w ; 13 3
jmp :WaitOn ; 14 3
jmp :WaitOff ; 15 3
:WaitOn ; 16
  snb rb.3 ; 17 1/2
  jmp :Continue ; 18 3
  inc ISRState ; 19 1
  xor Time, #%00001111 ; 20 2
  jmp :Continue ; 21 3
:WaitOff ; 22
  sb rb.3 ; 23 1/2
  jmp :Continue ; 24 3
  clr ISRState ; 25 1
:Continue ; 26
  decsz TimerLED ; 27 1/2
  jmp :ExitISR ; 28 3
  mov TimerLED, Time ; 29 2
  xor rb, #%00000001 ; 30 2
:ExitISR ; 31
```

```

    mov w, #- 100      ; 32    1
    reti w            ; 33    3
Main                 ; 34
    clr ISRstate      ; 35
    mov Time, #10     ; 36
    clr Timer256      ; 37
    mov Timer78, #78  ; 38
    mov TimerLED, Time ; 39
    mode TRIS         ; 40
    mov !rb, #%11111110 ; 41
    mode PLP          ; 42
    mov !rb, #%11110111 ; 43
    mov !Option, #%10001000 ; 44
: Loop                ; 45
    inc Counter       ; 46
    jmp : Loop        ; 47

```

In lines 1...6, we define the variables that are required. **Time** is used to hold the initialization for the LED frequency (5 or 10). **Timer256**, **Timer78** and **TimerLED** are the three timer counters for the required time intervals. **Counter** is used to keep the mainline program "busy", and **ISRState** is a variable that controls the operations within the ISR.

The mainline program initializes the variables and the ports (lines 35...43), and in line 44, it enables the RTCC interrupt without prescaler.

Within the ISR, **Timer256** is decremented first. As long as it does not underflow, the ISR is terminated immediately. After 256 ISR calls, **Timer256** underflows, and **Timer78** is decremented.

After 19,968 ISR calls, **Timer78** underflows. In this case, **Timer78** is initialized to 78 again, and the subsequent ISR instructions are executed.

At this point, there can be two different cases that must be considered:

- State 0: The button was not pressed before, i.e. we should see if it is pressed now.
- State 1: The button was pressed before, i.e. we should see if it has been released in the meantime.

The **ISRState** variable tells us, which state is currently active, and we now must find a way to branch to different parts of the ISR, depending on the contents of **ISRState** in order to handle the different states (two in our example).

The **jmp pc+w** instruction makes it easy to code such "calculated" branches. This happens in lines 12 and 13. First, **w** takes the contents of **ISRState**, and then a **jmp pc+w** is executed.

This instruction adds the contents of **w** to the lower eight bits of the program counter (PC register). Here, **w** can contain 0 or 1. After the execution of the **jmp pc+w** instruction PC is automati-

Programming the SX Microcontroller

cally incremented, i.e. it now points to **jmp :WaitOn**. In case **w** contains zero, this is the next instruction to be executed. In case **w** contains one, PC points to **jmp :WaitOff** after the **pc+w** operation, i.e. this will be the next instruction to be executed in this case.

Instruction sequences like

```
jmp pc+w
jmp :WaitOn
jmp :WaitOff
```

are also called "Jump Tables".



Routines that perform different actions depending on the contents of a state variable are also called "State Engines" or "State Machines". We will use state engines in various other examples in this book because they make it easy to handle timed operations that are typical for synchronous interrupts.

When **ISRState** is zero, the instructions following line 16 will be executed. In case **rb. 3** reads 1, the input line has high level, and no button is pressed. The state remains unchanged, and the ISR is left.

If **rb. 3** reads 0, the input line is low, i.e. the button is down now. In this case, **ISRState** is set to one, and **xor Time, #00001111** modifies the contents of **Time** that shall either be 10 or 5. If we represent those values in binary, you can see why this **xor** instruction modifies the **Time** values as desired:

10 = %0000 1010

5 = %0000 0101

In order to change the value from 5 to 10, and vice versa, the bits in the lower nibble must be inverted, and this is what the **xor** actually does.

When **ISRState** is one, the instructions following line 22 are executed, where we test if the pushbutton has been released already. If this is the case, (**rb. 3** is 1), **ISRState** is reset to zero otherwise, **ISRState** remains unchanged. Because the test for the button status is only executed after 19,964 interrupts, i.e. about every 40 milliseconds, the button has time enough to "bounce" in the meantime.

Right of the line number comments, we have added the clock cycles which each instruction requires. To make sure that the total number of clock cycles required by the ISR is less than the RTCC initialization (minus 7) we must determine the worst-case number of cycles. In our ISR, this is the case, when the following instructions are executed in sequence:

```
decsz Timer256 ; 7 2
decsz Timer78 ; 9 2
mov Timer78, #78 ; 11 2
```


Programming the SX Microcontroller

```
clr ISRState          ; 25  1
:Continue             ; 26
  decsz TimerLED      ; 27  1/2
  jmp :ExitISR        ; 28  3
  mov TimerLED, Time  ; 29  2
  xor rb, #%0000001   ; 30  2
:ExitISR              ; 31
; mov w, #-100        ; 32  1
; mov w, #-200        ; 32  1
retiw                  ; 33  3
```

In addition, change the following lines in the mainline program:

```
; mov Timer78, #78    ; 38
  mov Timer78, #39    ; 38

; mov !Option, #%10001000 ; 44
  mov !Option, #%10000111 ; 44
```

Now, the ISR is called after 51,200 clock cycles have elapsed ($256 * 200$), i.e. it is called every millisecond. The controller load, caused by the ISR is now less than 0.08%, so it can be almost ignored.

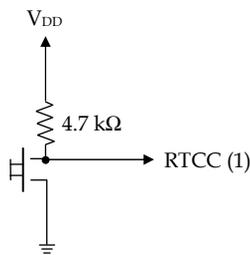
1.9.1.4 Interrupts Caused by Counter Overflows



(2.2.5.1.3 - 224) This third variant of interrupts that can be handled by the SX is very similar to timer-controlled interrupts. Again, an overflow of the RTCC triggers an interrupt. However, here, the RTCC is not clocked by the system clock but by an external source instead, that is fed into the RTCC pin of the SX.

Optionally, the prescaler can be used to divide the external signal, and you can configure the RTCC input to react on positive or negative signal edges.

To test the next example program, you should connect a pushbutton to the RTCC input. Because this input has no internal pull-up resistor, you must add an external resistor, as shown in the schematic below (we assume that an LED is still connected to RB0):



The program below should toggle the LED state after you have pressed the button five times:

```

; =====
; Programming the SX Microcontroller
; TUT024.SRC
; =====
include "Setup28.inc"
RESET    Mai n

TRIS     equ        $0f
PLP      equ        $0e

org      $000
;-----
; ISR
;-----
    xor    rb,      #%00000001
    mov    rtcc,    #251
    reti

org      $100
Mai n
    mov    !rb,     #%11111110
    clrb  rb.0
    mov    rtcc,    #251
    mov    !option, #%10111000
Loop
    inc   8
    jmp   Loop

```

This sample program configures port pin **rb. 0** as output to control the LED, and we set this output pin to low in order to turn on the LED. Then we initialize RTCC to 251 (256 - 5). The OPTION register is configured in order to make negative edges at the RTCC input increment the RTCC and that an RTCC overflow triggers an interrupt.

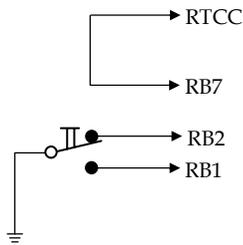
As there is nothing else to do in the mainline program, we let it increment the register at \$08.

When you press the button five times (hoping that it won't bounce), the RTCC is incremented each time you push the button. When it overflows, the ISR toggles **rb. 0** to turn the LED on or off. Before leaving the ISR, RTCC is re-initialized to 251.

When you test the program at full-speed, it is most likely that the LED will toggle after less than five button pushes. The SX is simply too fast! Therefore, the RTCC also counts the negative edges caused by button bounces.

Next, let's look at an example that shows, how the SX can "feed" its own RTCC input. We use this here to de-bounce a button that generates the signals for the RTCC input. For this example, we need a button with make and break contacts, connected to RB3 and RB2. A jumper between RB7 and the RTCC input is also required:

Programming the SX Microcontroller



This is the program:

```

; =====
; Programming the SX Microcontroller
; TUT025. SRC
; =====
include "Setup28.inc"
RESET      Main

TRIS       equ    $0f
PLP        equ    $0e

org        $08
ButtStat   ds    1

org        $000
; -----
; ISR
; -----
xor        rb,    #%00000001
mov        rtcc,  #251
reti

org        $050
Main
clr        ButtStat
mode       TRIS
mov        !rb,  #%01111110
mode       PLP
mov        !rb,  #%11111001
clrb       rb.0
setb       rb.7
mov        rtcc, #251
mov        !option, #%10111000

Loop
mov        w,    ButtStat
jmp        pc+w
jmp        :WaitForOn
jmp        :WaitForOff

:WaitForOn
snb        rb.1
```

```

    jmp    Loop
    clrb  rb. 7
    inc   ButtStat
    jmp   Loop

:WaitForOff
    snb   rb. 2
    jmp   Loop
    setb  rb. 7
    clr   ButtStat
    jmp   Loop

```

The mainline program first clears the **ButtStat** variable that is used to control the state-engine handling the button states. Then Port B is configured: **rb. 2** and **rb. 1** are inputs with active pull-up resistors, **rb. 0** is an output to drive the LED, and **rb. 7** is an output that feeds back into the RTCC input.

We then turn on the LED, and set **rb. 7** (and the RTCC input) to high level.

Then we enter the main loop. There, the jump table brings us to **:WaitForOn** or **:WaitForOff**, depending on **ButtStat**'s contents. Initially, **ButtStat** is zero, so we enter **:WaitForOn**. This state remains active until the button's make contact closes the first time. In this case, **rb. 7** (and RTCC) is set to low, and **ButtStat** is set to one.

Now, the jump table brings us to **:WaitForOff**. We stay in this state, until the button's break contact closes the first time. This happens when the button is released. In this case, **rb. 7** (and RTCC) is set to high, and **ButtStat** is cleared again.

This perfectly de-bounces the button, and the negative edge, generated in **:WaitForOn** triggers the RTCC.

The ISR is identical to the one in the previous example.



You may have noticed that the mainline program originates at address \$050 instead of \$100 in this example program. The reason for this is that the **jmp pc+w** can only address targets that are located in the first half of a program memory page. The **jmp pc+w** instruction adds the contents of *w* to the lower eight bits of the program counter and clears bit 9. Therefore a calculated jump can only target an address in the range from \$000 through \$0ff (this is similar to subroutine calls).

When using the **jmp pc+w** instruction, double-check that this condition is respected; otherwise, strange things will happen.

Note that most assemblers report an error when you try to call a subroutine outside of the first half of a page, but they do not report an error when targets of **jmp pc+w** are "out of limits".

Programming the SX Microcontroller

In a "real-life" application, it is possibly not the best idea to keep the mainline program busy with de-bouncing a pushbutton that in the end supplies an interrupt reason.

Before ending the chapter about interrupts, let's summarize the meaning of the OPTION register bits because their settings are important to configure the various interrupt features of the SX:

1.9.2 The OPTION Register Bits and their Meaning

	Bit 7 - RTW:	(RTCC or W)
	1 =	RTCC register can be accessed at \$01
	0 =	W can be accessed at \$01
	Bit 6 - RTI:	(RTCC Interrupt)
	1 =	No interrupt on RTCC overflows
	0 =	Interrupt on RTCC overflows
	Bit 5 - RTS:	(RTCC Source)
	1 =	RTCC counts external signals
	0 =	RTCC counts system clock cycles
	Bit 4 - RTE:	(RTCC Edge)
1 =	Negative edges trigger the RTCC	
0 =	Positive edges trigger the RTCC	
Bit 3 - PSA:	(Prescaler Assignment)	
1 =	Prescaler is assigned to watchdog	
0 =	Prescaler is assigned to RTCC	
Bit 2...0:	Prescaler divide-by factor (1:2...1:256 for RTCC, (1:1...1:128 for watchdog)	
Note that programs must contain an OPTIONX or STACKX directive when the RTW or RTI bits in the OPTION register shall be modified – otherwise, these bits are read-only.		

1.10 The Watchdog - Reset Reasons - Conditional Assembly

1.10.1 The Watchdog Timer



(2.2.5.2 - 225) Even when you have carefully designed and thoroughly tested an SX program, it is still possible that it may contain bugs or that a specific combination of external events and signals will not be handled correctly because you had assumed that they can never occur. It is also possible that failures of external components or devices cause such an error situation.

Microcontrollers like the SX are often used for process control systems, where it is very important to make sure that possible errors or malfunctions do not cause dangerous situations. Imagine what might happen if a motor running at full speed is not turned off in case of such an error!

Typical SX applications continuously execute a main loop. In case of an error, it is most likely that the program "hangs" at a specific location or in a subroutine. This causes that the program no longer cycles through the main loop at all, or at another speed as usual.

The watchdog timer (WDT) can help to detect such situations, and perform a system reset in case of a failure. Similar to an un-educated dog that would attack you unless you keep shouting: "Sit!", it is necessary to periodically reset the WDT to avoid a system reset.

A watchdog timer works similar to the "dead-man" button in a railroad locomotive. Here, the engineer must hit the "dead-man" button regularly; if he doesn't, the train will automatically stop.

Usually, resetting the WDT is done in a program loop that is cycled through periodically, and that is most likely to be stopped on a failure. When this happens, the WDT runs into time-out, and executes a system reset.

The instruction

clr !wdt

is used to clear the watchdog timer. In addition, the WDT must be enabled. The setting of a bit in the Fuse Register enables or disables the WDT. By default it is disabled, and in order to enable it, add the

DEVICE WATCHDOG

directive to the program source code. This instructs the development system to set bit 3 (WDTE) in the Fuse register while programming the SX device.

In addition, at program initialization, you must define the timeout period for the WDT.

A special 8-bit counter is dedicated to the WDT, and a separate internal signal is generated to clock this counter. When the counter overflows, a system reset is performed. The **clr !wdt** in-

Programming the SX Microcontroller

struction resets the counter, i.e. if this instruction is always executed in time, before the WDT counter overflows, the program will run normally.

As an option, the internal prescaler can be used to divide the internal clock signal in order to generate longer WDT timeout periods. In this case, the prescaler is no longer available for the RTCC. To activate the prescaler for the WDT, OPTION bit 3 (PSA) must be set:

OPTION							
7	6	5	4	3	2	1	0
RTW	RTI	RTS	RTE	PSA	PS2	PS1	PS0

Here is a little example program (we assume that an LED is connected to RB0):

```
; =====  
; Programming the SX Microcontroller  
; TUT026.SRC  
; =====  
include "Setup28.inc"  
DEVICE WATCHDOG  
RESET Main  
  
org $08  
Counter ds 3  
  
Main  
  clr Counter  
  clr Counter+1  
  clr Counter+2  
  clr rb  
  mov !rb, #%11111110  
  mov !option, #%11111111  
  
:Loop1  
  clr !wdt  
  decsz Counter  
  jmp :Loop1  
  decsz Counter+1  
  jmp :Loop1  
  decsz Counter+2  
  jmp :Loop1  
  setb rb.0  
  
:Loop2  
  jmp :Loop2
```

The **DEVICE WATCHDOG** directive instructs the development system to set the WDTE bit in the Fuse register when programming the SX.

```
mov !option, #%11111111
```

assigns the prescaler to the WDT, and sets the divide-by factor to 1:128.

After reset, the LED connected to **rb. 0** is turned on. The program then enters into a three-level delay loop. Within the loop, the **cl r !wdt** instruction is executed during each loop cycle.

When the delay time has elapsed, the LED is turned off, and the program enters into an endless loop without clearing the WDT.

Run the program at full-speed, and watch the LED. It should be on for about two seconds (while the delay loop is active). About two seconds, after the LED has been turned off, the WDT should time-out and reset the SX. After reset, the program is re-entered, and the LED should be on again.

Note that SX development systems cannot run the debugger while the watchdog is enabled.

The internal WDT clock is approximately 14 kHz, i.e. it has a period of 71.4 μ s. The table below summarizes the WDT time-out periods, depending on the prescaler's divide-by factor. It also contains the settings for bits 2...0 (PS2...PS0) in the OPTION register:

PS2	PS1	PS0	Divide-by Factor for the WDT	WD Timeout Period
0	0	0	1:1	18 ms
0	0	1	1:2	37 ms
0	1	0	1:4	73 ms
0	1	1	1:8	146 ms
1	0	0	1:16	293 ms
1	0	1	1:32	585 ms
1	1	0	1:64	1,2 s
1	1	1	1:128	2,3 s

Note that - different from the RTCC factors - the divide-by factors range from 1:1 to 1:128 here.

In the example program, we have selected a divide-by factor of 1:128, therefore, the WDT resets the SX about two seconds after the LED has been turned off. (Again, we have found a new method to blink an LED.)

Programming the SX Microcontroller

1.10.2 Determining Reset Reasons

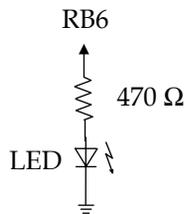
In certain cases, an application should react differently on a power-on reset ("Cold-Boot") or a watchdog reset ("Warm-Boot").

At a Cold-Boot, for example, it might be necessary to clear all data registers, to configure the ports, and to initialize certain variables, while after a Warm-Boot it might only necessary to re-configure the ports.

Another option might be that the program enters a special "Safety Mode" when the WDT caused a reset. In this mode, all port outputs could stay inactive and - except generating an error signal - the program does not perform any other tasks.

To find out if the WDT caused a reset, you can test bit 4 (TO - Time Out) in the STATUS register. After a WDT reset, this bit is clear, after a power-on reset, this bit is set. This bit is also set by the instructions **clr !wdt** and **sleep** (we'll discuss the **sleep** instruction later in this text).

The next example program is based upon the previous one. It detects a WDT reset and generates an alarm beep in this case. We assume that a piezo buzzer is connected to pin RB7. If you don't have a piezo buzzer available, connect another LED to RB6 (not RB7) according to the schematic below:



```
; =====  
; Programming the SX Microcontroller  
; TUT027. SRC  
; =====  
include "Setup28.inc"  
SOUND  
DEVICE WATCHDOG  
RESET Main  
  
org $08  
Counter ds 3  
  
Main  
  mov !option, #%11111111  
  clr Counter  
  clr Counter+1  
  clr Counter+2  
  sb status.4
```

```

    jmp WarmBoot
    clr  rb
    mov  !rb, #%11111110

: Loop1
    clr  !wdt
    decsz Counter
    jmp  : Loop1
    decsz Counter+1
    jmp  : Loop1
    decsz Counter+2
    jmp  : Loop1
    setb rb. 0
: Loop2
    jmp  : Loop2

WarmBoot

ifdef SOUND
    mov  !rb, #%01111111
: Loop1
    mov  Counter+1, #30
: Loop2
    clr  !wdt
    decsz Counter
    jmp  : Loop2
    decsz Counter+1
    jmp  : Loop2
    xor  rb, #%10000000
    jmp  : Loop1
else
    mov  !rb, #%10111111
    setb rb. 6
: Loop1
    clr  !wdt
    jmp  : Loop1
endif

```

At the very beginning of this program, we have inserted the label **SOUND** (we will explain its meaning later).

After the usual initializations that are required after any kind of reset, we test **status. 4** and continue the program similar to the last example if this bit is set.

In case the WDT has caused the reset, the bit is clear, and the program execution continues at WarmBoot.

1.10.3 Conditional Assembly

Following the label **WarmBoot**, you will notice the **ifdef SOUND** directive. This directive is used to control the conditional assembly. This means that the assembler translates the statements be-

Programming the SX Microcontroller

tween the line containing the **ifdef** and the line containing the **else** directive only when label **SOUND** is defined.

In case label **SOUND** is not defined, the assembler will translate the statements between the **else** and the **endif** directive.

In general, the **else** directive is optional. When it is missing, no instructions will be generated when the **ifdef** condition evaluates to false.

In our example, we have placed the label **SOUND** right at the beginning of the program source, therefore the statements in the **ifdef** section are generated to drive the piezo buzzer.

To drive the buzzer, we use a nested delay loop with two levels, and toggle the buzzer output whenever the delay time is over. While in this loop, it is important to clear the WDT in order to avoid another reset.

In case you don't have the buzzer connected, but the LED at RB6 instead, simply out-comment the **SOUND** label with a leading semicolon. Now, the assembler generates the instructions in the **else** segment that turn on the LED at RB6 instead.

To stop the buzzer, or to turn off the "Alarm LED", press the reset button on the prototype board, or disconnect the power supply.

As you can see, conditional assembly can be used to generate different program versions from one source code.

1.10.3.1 More Directives for Conditional Assembly

Besides the **ifdef...else...endif** structure described before, there are more options for conditional assembly:

```
ifndef <Symbol >  
    <Code Block 1>  
[else  
    <Code Block 2>]  
endif
```

Here, the instructions in **<Code Block 1>** are generated when **<Symbol >** is not defined, and the instructions in **<Code Block 2>** are generated when **<Symbol >** is defined. The **else** directive and **<Code Block 2>** are optional.



A symbol that controls an **ifdef** or **ifndef** structure must be either a label (as in our example), or being defined with an **EQU** directive. Symbols that are defined with the **SET** or **=** directive are not accepted.

```
if <Condi ti on>
    <Code Block 1>
[else
    <Code Block 2>]
endif
```

Here, the instructions in **<Code Block 1>** are generated when **<Condi ti on>** evaluates to true, otherwise the instructions in **<Code Block 2>** are generated. The **else** directive and **<Code Block 2>** are optional.

```
rept <Count>
    <Code Block>
endr
```

This structure repeats the instructions in **<Code Block>** as often as specified by **<Count>**. For example,

```
rept 3
    inc Counter
    inc Counter+1
endr
```

causes the assembler to generate code for the following instructions:

```
inc Counter
inc Counter+1
inc Counter
inc Counter+1
inc Counter
inc Counter+1
```

Within the **<Code Block>** you may use the percent sign that is replaced by the repeat counter. For example:

```
rept 3
    mov ra, #%
endr
```

generates

```
mov ra, #1
mov ra, #2
mov ra, #3
```

Programming the SX Microcontroller

The repeat count may also be defined elsewhere in the program, e.g.

```
HowOften = 5  
rept HowOften  
  mov ra, #%  
endr
```

Logical expressions for an **if** directive may use the following operands:

Operands for Logical Expressions	
Symbol	Meaning
op1 = op2	op1 is equal to op2
op1 <> op2	op1 is not equal to op2
op1 < op2	op1 smaller than op2
op1 > op2	op1 greater than op2
op1 <= op2	op1 smaller than or equal to op2
op1 >= op2	op1 greater than or equal to op2
exp1 AND exp2	Logical AND
exp1 OR exp2	Logical OR
exp1 XOR exp2	Logical Exclusive OR
NOT exp	Negation of exp

Example for a logical expression:

```
if Version >= 5 AND Version < 10
```

1.10.4 "To Watchdog or not to Watchdog..."

As mentioned before, debuggers don't work when the SX is configured with an enabled watchdog. This is not a problem of the debuggers, but it is due to the internal structure of the SX devices.

Place a global symbol **DEBUG** at the beginning of your program, and later insert the conditional

```
ifndef DEBUG  
  DEVICE WATCHDOG  
endif
```

This makes it easy to "switch" the program between the test and production version.

1.11 The Sleep Mode and Wake-up

Low power consumption is an important issue for battery-powered systems. The power consumption of the SX controller mostly depends on the system clock frequency. At 5 V V_{DD} , the data sheet specifies a typical supply current of 60 mA at 50 MHz clock, but only 6 mA at 4 MHz internal clock.

The sleep instruction sets the SX into "Sleep Mode". In this mode, the supply current drops down to about 10 μ A when the watchdog timer is disabled.

While in sleep mode, the system clock is halted, i.e. no more instructions are executed, but the contents of all registers are preserved.

To exit the sleep mode, a "Wake-up Event" must occur. This can either be a signal edge at one of the Port B pins or a reset caused by the WDT.

You may compare the sleep mode and possible wake-ups with a human that goes to sleep. To wake up in the morning early enough to go to work, you set an alarm clock to wake you up at a specific time. But while you are asleep, there may be several reasons that wake you up before the scheduled time: The baby might cry, the phone may ring, the burglar alarm could be triggered, you wake up and feel the need to go to the bathroom, etc.

1.11.1 Wake-ups Caused by Port B Signal Edges

Such wake-ups are similar to the "non-scheduled" wake-ups listed with the "human example" above.

Waking up the SX by signals at Port B pins makes sense, when the SX shall react on certain external events, like a pressed pushbutton, increase of a voltage above a certain level, etc. Because all eight Port B pins may be configured to trigger the wake-up, up to eight different external signals can be recognized.

Together with interrupts, we already have shown how to configure the Port B inputs in order to generate an interrupt on positive or negative signal edges. Similarly, these signal edges may be used to trigger a wake-up.

The following steps are required for configuration:

- Load MODE with \$0b (selects the WKEN_B register)
- Clear the bits for the corresponding inputs in **!rb** that shall trigger a wake-up

Programming the SX Microcontroller

In this configuration, negative signal edges are detected. In order to react on positive edges, additional steps are required:

- Load MODE with \$0a (selects the WKED_B register)
- Clear the bits for the corresponding inputs in **!rb** that shall react on positive edges.

When the SX detects a wake-up, the PC is loaded with the highest program memory address, and program execution starts there, performing a **jmp** to the address specified with the **RESET** directive in the program source code. It is the same sequence that is executed at power-on reset, or when a watchdog reset occurs.

In order to find out the reason for a reset, the STATUS register delivers the necessary information. The sleep instruction clears STATUS bit 3 (PD - Power Down), i.e. if this bit is clear after a reset, it was caused by a wake-up event.

In the previous chapter, we already mentioned STATUS bit 4 (TO - Time Out) that gives information about a watchdog reset. The table below summarizes the events that set or clear these bits:

Status Bits set or cleared by		
Value	Bit 4 (TO - Time Out)	Bit 3 (PD - Power Down)
0	WDT Overflow	sleep
1	Power On, clr !wdt, sleep	Power On, clr !wdt

In order to find out the reset reason, these bits must be tested. The table below summarizes the reset reasons:

Reset Reasons		
TO	PD	Reason
0	0	Wake-up by Watchdog
0	1	Watchdog Timeout (System Fault)
1	0	Wake-up by signal edge at an RB pin
1	1	Power on

The program below demonstrates how the sleep mode is activated, and how to react on a wake-up (we assume that an LED is connected to rb.0 and that a pushbutton is connected to RB3 that pulls the input low when the button is pressed:

```

; =====
; Programming the SX Microcontroller
; TUT028.SRC
; =====
include "Setup28.inc"
DEVICE    WATCHDOG
RESET     Main

TRIS      equ    $0f
PLP       equ    $0e
WKEN      equ    $0b
WKED      equ    $0a
WKPND_W   equ    $09

org        $08
Counter   ds    4
Time      ds    1

org        $100
Main
    mov    w, #50
    sb    status.3
        mov    w, #25
    mov    Time, w

    mode    WKEN
    mov    !rb, #%11111111
    mode    PLP
    mov    !rb, #%11110111
    mode    TRIS
    mov    !rb, #%11111110

    clr    Counter
    clr    Counter+1
    clr    Counter+2
    clrb  rb.0

    mov    Counter+3, #11
:Loop1
    mov    Counter+2, Time
:Loop2
    decsz Counter
        jmp  :Loop2
    decsz Counter+1
        jmp  :Loop2
    decsz Counter+2
        jmp  :Loop2
    xor    rb, #%00000001
    decsz Counter+3
        jmp  :Loop1

    mode    WKED
    mov    !rb, #%11111111
    mode    WKPND_W

```

Programming the SX Microcontroller

```
clr w
mov !rb, w
mode WKEN
mov !rb, #%11110111
sleep
```

To analyze this program, let's first look at the last statement: It reads **sleep**, i.e. after the program has performed the preceding instructions, the SX is entered into sleep mode.

In the beginning of the program (after **Main**), we first must find out the reset reason. Here, we initialize **w** with 50 as the possible value to be copied to **Time**.

Next, we test the Power Down bit in the STATUS register. When this bit is set, we have a power-on reset. (As the watchdog is disabled here, this bit can never be set due to a watchdog event here.) In case of a wake-up, this bit is clear, and **w** is initialized to 25. **w** is then copied to **Time**, the variable that controls the LED blink frequency, i.e. after a power-on, the frequency is low, and after a wake-up, the frequency is high.

We then set all **WKEN_B** bits temporarily, i.e. we disable the wake-up/interrupt functionality for the "regular" program execution, configure the LED output and activate a pull-up for the pushbutton.

Next, we clear the counters, turn the LED on, and enter a nested delay loop that toggles the LED state five times.

We then setup the Port B pins to react on negative signal edges (although this is the default, it is a good idea to make sure that the bits are really set), clear the **WKPND_B** register, and finally enable the wake-up function for the RB3 pin.

Finally, the SX is set to sleep mode.

Testing the program with the debugger requires some "tricks" to make the program work as expected.

After the program has been assembled and transferred to the SX, turn off and on the power supply to make sure that **status. 3** (Power Down) is set.

Then re-activate the debugger and check if bits 4 and 3 of the STATUS register are both set, and run the program at full speed. Now, the LED should slowly blink five times. During that time, the debugger should indicate that the program is running. When the LED stops blinking, the debugger should report that the sleep mode is active.

Now press the button at RB3; the debugger should report an idle mode now, i.e. it does not automatically start the SX. Again, run the program at full speed - the LED should now blink five times at a higher frequency, before the SX enters the sleep mode again.

When you end the sleep mode by pressing the reset button, or by issuing a reset from the debugger, the LED will blink at the higher frequency, i.e. the program has not detected a power-down. Therefore, you need to repeat the steps above to enter that state again (power off, power on, restart the debugger).



When Port B pins are enabled for wake-up/interrupts during "regular" program execution, a signal edge at one of these pins will cause an interrupt (in our program example, this could happen when you press the pushbutton while the LED blinks). This can cause unpredictable results when no ISR is in place to handle such interrupts. Therefore, it is a good idea to enable the port pins for wake-up/interrupts immediately before entering the sleep mode. *Never forget to clear the WKPD register before enabling wake-up/interrupts.*

In "real life", it is most likely that the SX must react on interrupts as well as on wake-up events. As long as interrupts are triggered by the RTCC, no special safety measures are required when you enable the wake-up port bits immediately before entering the sleep mode.

When Port B inputs are used for interrupts and for wake-ups, it is necessary to let the ISR check the WKPD register to find out the interrupt reason. If none of these bits is set, the interrupt was caused by an RTCC overflow (in case this kind of interrupt is enabled). If one of the bits corresponding to interrupt inputs are set, handle these events in the ISR and if one of the wake-up inputs caused the interrupt, simply ignore that event. However, the better solution is to disable the wake-up pins until the system is ready for sleep mode.

In any case, do not forget to clear the WKPD register before returning from the ISR!

1.11.2 Using the Watchdog Timer for Wake-ups

This wake-up is similar to the "scheduled" wake-up in the "human" example above, i.e. it wakes up the SX after an elapsed time.

For certain applications, it is not necessary that the SX is active all the time. For example, to monitor a temperature value that is changing very slowly. In this case it might be sufficient to read that value every second, and report changes only when the temperature has changed by more than one degree since the last report.

The next program example is similar to the one we have used to demonstrate the watchdog timer. We have added the conditional assembly directive **ifdef GO_SLEEP** here:

```

; =====
; Programming the SX Microcontroller
; TUT029.SRC
; =====
include "Setup28.inc"
DEVICE WATCHDOG

```

Programming the SX Microcontroller

```
RESET    Main
GO_SLEEP

org      $08
Counter ds 3

Main
clr      Counter
clr      Counter+1
clr      Counter+2
clr      rb
mov      !rb, #%11111110
mov      !option, #%11111111

: Loop1
  clr    !wdt
  decsz  Counter
  jmp    : Loop1
  decsz  Counter+1
  jmp    : Loop1
  decsz  Counter+2
  jmp    : Loop1
  setb   rb.0

ifdef GO_SLEEP
  sleep
else
: Loop2
  jmp    : Loop2
endif
```

When you run this program, the LED will turn on for about two seconds. Then the SX enters the sleep mode, and the watchdog timer will wake it up after another two seconds, i.e. the LED is turned on again. This cycle is repeated forever as long as the power supply is on.

Now hook up a multi-meter in series with the V_{DD} power supply line in order to measure the supply current, and comment out the `clr rb` instruction to disable the LED for that test (to avoid a measurement error caused by the LED current).

```
Main
clr      Counter
clr      Counter+1
clr      Counter+2
; clr    rb
```

Re-assemble the program and run it again watching the multi-meter. You will notice that the current readout changes between a high and a low value. The low value indicates that the SX is in sleep mode, and the high value indicates the "regular" program execution.

In addition, now also comment out the `GO_SLEEP` label at the beginning of the program. When you re-assemble the program and run it again, the supply current will stay at the high level. This

is because now the SX stays in run-mode all the time, and the watchdog timer causes a reset instead of a wake-up.

Note: The readout may not actually indicate the current drawn by the SX. Depending on what prototype board you are using, other components like a power LED and a voltage regulator may add to the measured value.

1.12 Macros - Expressions - Symbols - Device Configuration

1.12.1 Macro Definitions

Macro definitions allow you to define "your own" instructions that go beyond the regular instructions supported by the assembler. Here is an example:

```
mul 2 MACRO 1
  cl c
  rl \1
endm
```

This defines a new "instruction", **mul 2** that has the following syntax:

```
mul 2 fr
```

i.e. it expects the address of a file register. If you, for example, use

```
mul 2 $08
```

to call that macro, the assembler generates the following instructions instead:

```
cl c
rl $08
```

The general syntax for a macro definition is

```
<Macro Name> MACRO [<Argument Count>]
  <Instructions>
ENDM
```

Each macro definition begins with a unique name that you will use later to call the macro. Next comes the keyword **MACRO** optionally followed by a number that specifies the number of arguments to be passed to the macro (up to 64 arguments are allowed).

Then, within the macro-body, the instructions follow. The assembler will replace the macro call by these instructions. The **ENDM** keyword terminates the macro definition.

Here are some more examples for macro definitions:

```
Bushigh MACRO
  setb ra. 0
  setb ra. 1
```

Programming the SX Microcontroller

endm

Macro call: **BusHigh**

This macro does not expect any arguments; it sets both port pins RA0 and RA1 to high level.

```
SwapRegs MACRO 2  
  mov w, \1  
  mov LocalTemp2, w  
  mov w, \2  
  mov \1, w  
  mov w, LocalTemp2  
  mov \2, w  
endm
```

Macro call: **SwapRegs fr1, fr2**

Modifies: **w, LocalTemp2**

This macro expects two register addresses as arguments and it exchanges the contents of the two registers.

Whenever the assembler finds the placeholder for the first argument (**\1**) in the macro body, it replaces it with the first argument of the macro call. Similarly, the placeholder for the second argument (**\2**) is replaced by the second argument of the macro call.

For example, when you call the macro like this:

```
SwapRegs, Val 1, Val 2
```

The assembler replaces it with the following instructions:

```
mov w, Val 1  
mov LocalTemp2, w  
mov w, Val 2  
mov Val 1, w  
mov w, LocalTemp2  
mov Val 2, w
```

In addition, the assembler also replaces the symbolic names **Val 1** and **Val 2** by the addresses assigned to these symbols. We have left in the symbol names her for clarity.

Within a macro definition, you may use other directives, like such for conditional assemblies, as in

```
BusHigh MACRO  
  ifdef BUS_A  
    setb ra.0  
    setb ra.1  
  else  
    setb rc.0  
    setb rc.1  
  endif  
endm
```

When **BUS_A** is defined, the assembler will generate the instructions

```
setb ra.0  
setb ra.1
```

to replace the macro, otherwise it will generate the instructions

```
setb rc.0  
setb rc.1
```

There is another method how this macro could be defined:

```
BusHigh MACRO  
  ifdef BUS_A  
    setb ra.0  
    setb ra.1  
    exitm  
  endif  
  setb rc.0  
  setb rc.1  
endm
```

Here, the instructions **setb ra.0** and **setb ra.1** will be generated when **BUS_A** is defined. The **EXITM** directive terminates the definition of a macro. Here, this skips the **setb rc.0** and **setb rc.1** instructions. If **BUS_A** is not defined, these two instructions will be generated instead.

The placeholder **\0** has a special meaning. It is replaced by the macro's argument count. Here is an example:

```
ClrRegs MACRO 3  
  rept \0  
    clr \%  
endm
```

Macro call: **ClrRegs fr1, fr2, fr3**

For example, the assembler will replace the macro call

```
ClrRegs Counter, Sum, Temp
```

with

```
clr Counter  
clr Sum  
clr Temp
```

(The symbolic addresses will be replaced by the actual register addresses as well).

When you later decide to change the argument count for that macro, you will only have to change the count following the macro keyword, but there is no need to change the macro body.

Programming the SX Microcontroller

A macro may also call other macros, no matter if these macros are defined before or after the macro, that calls them. In case you get an error message like "Macro Stack Overflow", it is most likely that you have tried to define a "recursive" macro, as in

```
Mac1 MACRO
  clr w
  Mac1 ; Here, Mac1 "calls itself"
endm
```

Note that there may be a recursion across two or more macros as well:

```
Mac1 MACRO
  clr w
  Mac2
endm
```

```
Mac2 MACRO
  clr w
  Mac1
endm
```

Here, **Mac1** calls **Mac2**, where **Mac2** calls **Mac1** again. In this case, the assembler does not report an error when it reads the macro definitions, but if one of the two macros is referenced later in the program, an error will be reported.

1.12.1.1 Macros or Subroutines?

Similar to a subroutine, a macro combines a set of instructions, i.e. you don't have to place the same sequence of instructions several times throughout the program code. Instead, you place them once in the macro body, and refer to the macro in the remaining code.

Instead of

```
BusHigh MACRO
  setb ra.0
  setb ra.1
endm
```

you could define a subroutine instead:

```
BusHigh
  setb ra.0
  setb ra.1
ret
```

In the remaining code, instead referring the macro you would then use **call BusHigh** instructions.

In our short example, this is not very efficient because using a subroutine requires two extra program words for the **call** and the **ret** plus six extra clock cycles at run-time, where the two "real" instructions require two program words and two clock cycles only.

When you define macros that contain a larger number of instructions, a subroutine might be more efficient. Using a macro means that this number of instructions will be repeated as often as the macro is referred to in the program. Using a subroutine means that these instructions only exist once in the subroutine-body (plus an extra **ret** instruction), and calling the subroutine requires just one additional program word per **call**.

On the other hand, macros don't require the extra six clock cycles for the **call** and **ret** instructions.

As you can see, it is not easy to define a "rule of thumb" when to use macros or subroutines.

This little example program is good for a surprise:

```

org $08 ; 1
flags ds 1 ; 2

LedOn MACRO ; 3
    clrb rb.0 ; 4
ENDM ; 5

Main ; 6
:Loop ; 7

; some instructions

snb Flags.3 ; 8
LedsOn ; 9
jmp :Loop ; 10

```



The assembler will report, that **:Loop** is undefined in line 10 when you try to assemble the program. This looks strange because **:Loop** seems to be defined in line 6.

The problem is caused by the miss-spelled macro call in line 9. Instead of **LedOn**, we have written **LedsOn** (with an extra "s"). This makes the assembler assume that **LedsOn** is a new global label, and therefore, **:Loop** in line 10 is local to **LedsOn** but it is no longer local to **Main**.

Programming the SX Microcontroller



The example contains another possible pitfall in lines 8 and 9:

Similar to compound instructions, that never must immediately follow a skip instruction, be careful when placing macros immediately after a skip. As long as the macro expands into just one instruction, like in the example above, you are in good shape, but trouble is guaranteed when you try the same, calling a "multi-line" macro.

1.12.2 Expressions

Whenever a value is expected as instruction argument, or as argument for a directive, you may use an expression instead. We have already made use of this when referring to multi-byte registers, as in

```
org $08
Counter ds 3
```

```
clr Counter
clr Counter+1
clr Counter+2
```

The symbol **Counter** represents a value of eight (the address of this variable in data memory). By adding one or two to this "base address", we could also refer to the two bytes following **Counter** in data memory.

The table below summarizes the operators that can be used to build expressions:

Expression Operators		
Symb.	Meaning	Remark
+	Addition	
-	Subtraction	
*	Multiplication	
/	Division	a / b returns the integer part of the division
//	Modulus	a // b returns the remainder of a / b
&	log. AND	returns false (0) or true (FFFFFFFF, -1)
	log. OR	returns false (0) or true (FFFFFFFF, -1)
^	log. Exklus.-OR	returns false (0) or true (FFFFFFFF, -1)
<<	Shift left	w << n shifts w-bits by n digits to the left
>>	Shift right	w >> n shifts w-bits by n digits to the right
><	Swap nibbles	
	Absolute value	unary
-	Negative value	unary
~	Logical NOT	unary, returns false (0) or true (FFFFFFFF, -1)

Numbers are allowed to a maximum size of up to 32 bits, and results may not have a size of more than 32 bits. As the SX processes 8-bit values, results that exceed the size of 8 bits are truncated. Therefore, make sure that truncation does not remove significant bits.

Expressions are evaluated from left to right without respecting any operator hierarchy. For example, the expression

2 + 3 * 4

yields in 20 and not in 14 as you might expect when applying standard algebraic rules. If another operator precedence is required, use parentheses. Thus,

(2 + 3) * 4

yields in 14.

1.12.3 Data Types

Numeric values may have the following formats in expressions:

Numeric Formats			
Type	Syntax	Example	Remark
decimal	xxxxx	1234	
hex	\$xxxx	\$abcd	
hex	0xxxxh	0abcdh	old style
binary	%xxxx	%10110110	
binary	xxxxb	10110110b	old style
ASCII	'x'	'U'	

The "old style" formats are supported for compatibility reasons. We don't recommend using them in new source code.

1.12.4 Symbolic Constants

We have been using symbolic constants in many of the programs so far. At this place, we want to look at some more examples how symbolic constants can help to make programs more flexible, and more readable.

According to Murphy's Law: "Constants are always variable", this will happen to you as well, while developing SX applications. You can be sure that requirements for modifications will come up after a program is out of beta, running perfectly, and has been finally released for production.

Programming the SX Microcontroller

Maybe, it makes sense to change the port pin assignments in order to simplify the PCB layout. Maybe, a clock crystal with a slightly different frequency as the one you have based the timing on is cheaper, or whatever other reason it is - program adaptations are always the consequence.

If you, for example, have "hard-coded" the port pins in the program, you now have the burden to go through all of the source code in order to change the pin assignments. Often, just changing a `clrb rb. 0` to `clrb rb. 1` is not enough to use Port B pin 1 now, instead of pin 0.

Remember that the port configuration might be changed as well. Often port bits are "masked" out with an and instruction, i.e. the port masks must be changed as well, etc.

The following code style makes such modifications a lot easier:

```
-----  
; Port assignment  
;  
Temperature    equ ra. 0 ;-----+    1  
                ;                |    2  
TRIS_A         equ          %11111111 ;    3  
  
EmergencyOff1  equ rb. 0 ;-----+    4  
EmergencyOff2  equ rb. 1 ;-----+    5  
                ;                |    6  
TRIS_B         equ          %11111111 ;    7  
WKPEN_B       equ          %11111100 ;    8  
WKPED_B       equ          %11111101 ;    9  
  
SignalLED     equ rc. 0 ;-----+   10  
WarningLED    equ rc. 1 ;-----+   11  
Relay        equ rc. 2 ;-----+   12  
                ;                |   13  
TRIS_C        equ          %11111000 ;   14  
-----  
  
TRIS          equ $0f          ; 15  
PLP           equ $0e          ; 16  
WKEN_B       equ $0b          ; 17  
WKED_B       equ $0a          ; 18  
WKPD_W       equ $09          ; 19  
  
; I/O port configuration  
;  
mode TRIS          ; 20  
mov w, #TRIS_A    ; 21  
mov !ra, w        ; 22  
mov w, #TRIS_B    ; 23  
mov !rb, w        ; 24  
mov w, #TRIS_C    ; 25  
mov !rc, w        ; 26  
  
; Wake-up/Interrupt configuration  
;  
mode WKPD_W      ; 27
```

```

clr w; 28
mov !rb, w ; 29

mode WKED_B ; 30
mov w, #WKPED_B ; 31
mov !rb, w ; 32

mode WKEN_B ; 33
mov w, #WKPEN_B ; 34
mov !rb, w ; 35

;
; Hundreds of code lines later...
;

; Turn on signal LED
;
clrb SignalLED ; 36

; Activate the relay
;
clrb Relay ; 37

; Handle temperature status
;
sb Temperature ; 38
jmp :TooCold ; 39

;
; More instructions
;
:TooCold ; 40

```

At the beginning of this program code, we define symbolic names for each I/O signal like

```
Signal equ <Port>. <Bit>
```

(see lines 1, 4, 5, 10, and 12).

Later, when a port bit must be set or cleared, we use the symbolic names that were defined above.

Because there are no instructions available that set or clear single bits in the port configuration registers, we have defined constant bit patterns (lines 3, 7, 8, 9, and 14) that are loaded into the `!r?` registers (lines 21, 23, 25, 31, and 34).

If you consequently respect that scheme, changing port assignments later, is easy - changes are necessary in the initial definition section only.

In addition, macros can help to make a program even more generic: In the example above, we have assumed negative logic for the outputs, i.e. an LED or the relay become active, when an

Programming the SX Microcontroller

output goes low. In case your "Hardware Guru" finds out later that another inverting driver is required to drive the relay, the relay output must now assume positive logic.

Having defined the macro

```
RelayOn MACRO  
  clrb Relay  
ENDM
```

at the beginning of the program and calling that macro instead of placing **clrb Relay** instructions in the code (see line 37) makes this modification easy:

Just replace the **clrb Relay** instruction in the macro body with **setb Relay**, and you are all done.

1.12.5 The DEVICE Directives

Some of the available **DEVICE** directives have been used in the program examples already, because they were required to create the programs for debugging.



(2.2.7 - 230) **DEVICE** directives instruct the development system to set or clear bits in the SX "Fuse" registers. These registers are located in EEPROM, i.e. its contents do not get lost when VDD is turned off. "Fuses" are used to configure the SX for a specific need.

At this place, you can find a table that summarizes the most important **DEVICE** directives. The general syntax for a **DEVICE** directive is

```
DEVICE <Configuration>[, <Configuration>...]
```

The keyword **DEVICE** is followed by at least one more keyword that specifies a configuration. More words can be added in a comma-separated list.

One program may have more than one line with **DEVICE** directives.

DEVICE Directives			
Keyword	Description	Default	Remark
BANKS1	Configures memory size, 1 page, 1 bank	BANKS8	
BANKS2	Configures memory size, 1 page, 2 banks	BANKS8	
BANKS4	Configures memory size, 4 pages, 2 banks	BANKS8	
BANKS8	Configures memory size, 4 pages, 8 banks	BANKS8	
BOR22	Brownout reset at $V_{DD} < 2,2$ V	BOROFF	
BOR26	Brownout reset at $V_{DD} < 2,6$ V	BOROFF	
BOR42	Brownout reset at $V_{DD} < 4,2$ V	BOROFF	
BOROFF	Disable brownout	BOROFF	
CARRYX	ADD/SUB with Carry	disabled	
IFBD0	Disable internal oscillator feedback resistor	IFBD1	
IFBD1	Enable internal oscillator feedback resistor	IFBD1	
IRCDIV1	Internal RC oscillator, 4 MHz	IRCDIV1	
IRCDIV128	Internal RC oscillator, 32 kHz	IRCDIV1	
IRCDIV32	Internal RC oscillator, 128 kHz	IRCDIV1	
IRCDIV4	Internal RC oscillator, 1 MHz	IRCDIV1	
OPTIONX	Enable 8-level stack, and 8-bit OPTION register	disabled	1
OSC4MHZ	Internal RC oscillator, 4 MHz	OSC4MHZ	
OSC1MHZ	Internal RC oscillator, 1 MHz	OSC4MHZ	
OSC128KHZ	Internal RC oscillator, 128 kHz	OSC4MHZ	
OSC32KHZ	Internal RC oscillator, 32 kHz	OSC4MHZ	
OSCHS1	External oscillator, high-speed 1	OSCRC	
OSCHS2	External oscillator, high-speed 2	OSCRC	
OSCHS3	External oscillator, high-speed 3	OSCRC	
OSCLP1	External oscillator, low power 1	OSCRC	
OSCLP2	External oscillator, low power 2	OSCRC	
OSCRC	External oscillator, RC network	OSCRC	
OSCXT1	External oscillator, crystal 1	OSCRC	
OSCXT2	External oscillator, crystal 2	OSCRC	
PINS18	Specifies the SX device type	PINS18	
PINS20	Specifies the SX device type	PINS18	
PINS28	Specifies the SX device type	PINS18	
PINS48	Specifies the SX device type	PINS18	
PINS52	Specifies the SX device type	PINS18	
PROTECT	Program memory is read-protected	unprotected	
SX18AC	Specifies the SX device type	PINS18	
SX20AC	Specifies the SX device type	PINS18	
SX28AC	Specifies the SX device type	PINS18	
SX48BD	Specifies the SX device type	PINS18	
SX52BD	Specifies the SX device type	PINS18	
SYNC	Synchronized port-read (2 clock cycles delay for	disabled	
TURBO	Turbo mode enabled, 1 clock cycle/standard	disabled	1
WATCHDOG	Watchdog timer enabled	disabled	

Remark: 1: SX 18/20/28 only

Programming the SX Microcontroller

Note that the above device options are valid for the SASM Assembler. Depending on the version additional, or different options may be available. The SX-Key Assembler supports most of these options, but uses different keywords for some of them. Please refer to the documentation that comes with your assembler for more information about the device options.

1.12.6 The **FREQ** Directive (SX-Key only)

This directive is unique to the SX-Key. It specifies the clock frequency, the SX-Key probe shall generate. The syntax is

FREQ <Frequency in Hz>

Underscores are allowed (e.g. 50_000_000) for better readability.

1.12.7 The **ID** Directive

Within the SX EEPROM, there is room for eight characters that can optionally store any string of characters. This is a good place to store a program-id, and a version number. When you later read the program memory, you can identify the stored program. The syntax is:

ID '<String>'

for example:

ID 'BLINK1.0'

1.12.8 The **BREAK** Directive (SX-Key only)

This directive sets a pre-defined breakpoint for the SX-Key debugger. This is useful when you need to have a breakpoint set on a specific program line for a longer debugging session (obviously, the **BREAK** will sit close to your "most beloved bug" in the program).

The breakpoint will be set to the instruction in the line that comes next to the line with the **BREAK**.

While in a debugging session, you can set the breakpoint on any other instruction at any time by left-clicking the new line in the Code window.

There is only one **BREAK** directive at a time allowed in the source code. When you only change the position of the **BREAK** directive in the source code, it is not necessary to re-assemble the code, i.e. you may invoke the debugger using Debug (Reenter), or the Ctrl-Alt-D shortcut.

1.12.9 The ERROR Directive

This directive is useful to catch errors in macros.

The syntax for ERROR is:

```
ERROR ' <Message>'
```

When the assembler comes across an ERROR directive, it stops compilation, and displays **<Message>** in the status line. Here is an example:

```
PINS = 18  
  
if PINS = 18  
  DEVICE SX18L  
else  
  if PINS = 28  
    DEVICE SX28L  
  else  
    ERROR 'Illegal value for PINS'  
  endif  
endif
```

1.12.10 The END Directive

This directive indicates the end of the program source code. Any text that follows the **END** is ignored by the assembler. You can use this to add remarks and notes to the end of the source code file without the need to begin each line with a leading semicolon.

1.13 The Analog Comparator

Each SX device comes with an integrated analog converter. The two comparator inputs can optionally be accessed via Port B pins 1 and 2. An additional option connects the comparator output to Port B pin 0 when there is a need to control external components with the comparator output signal.

When Port B pins are configured for comparator input or output, the standard I/O pins are disabled.

In order to configure port B for comparator use, there is another configuration register available, the **CMP_B** register. This register can be accessed via `!rb` when the `MODE` register is initialized to `$08`.

The **CMP_B** register bits have the following meaning:

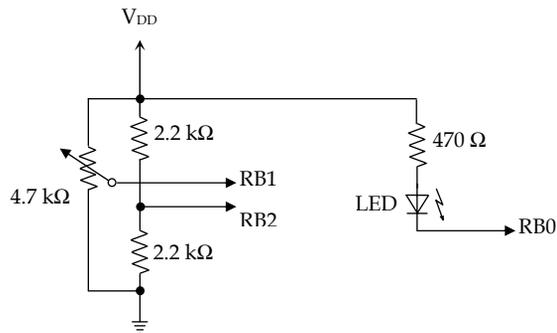
CMP_B							
7	6	5	4	3	2	1	0
EN	OE	-	-	-	-	-	Res

If bit 7 (EN - Enable) is clear, the comparator is enabled, and its inputs are fed through to pins RB1 and RB2.

If bit 6 (OE - Output Enable) is clear in addition to bit 7, the comparator output is enabled, and fed through to pin RB0.

In order to read the comparator output by software, a "trick" is used, that is similar to reading the **WKP_D_B** register. The `mov !rb, w` instruction exchanges the contents of `w` and the **CMP_B** register (provided that `MODE` is set to `$08`), i.e. before `mov !rb, w`, the `w` register must contain the comparator configuration bits, and **after** `mov !rb, w`, bit `w.0` represents the current comparator output status.

The schematic below shows the circuit we will be using to test the comparator.



```

; =====
; Programming the SX Microcontroller
; TUT030.SRC
; =====
include "Setup28.inc"

RESET    Main
CMP      equ $08

Main
mode CMP
IFDEF CMPOUT
    mov  !rb, #%00000000
ELSE
    mov  !rb, #%01000000
ENDIF

; Read the comparator output
;
;
: Loop
mode CMP
mov  !rb, #%00000000

; w.0 now is set to the comparator
; output state

    mov $08, w                ; <-- set a breakpoint here
    jmp :Loop

```

Please define a breakpoint at the marked line, and then run the program; it will stop at the breakpoint.

When you now turn the potentiometer slowly from one end to the other, the LED should change its state when the potentiometer is close to its center position (assuming, you are using a linear type). As you can see, the comparator keeps working even if the SX has halted program execution.

The table below summarizes the relationship between the comparator output and the input voltages:

Comparator	
Input	Output
$U_{RB1} > U_{RB2}$	low (0)
$U_{RB1} < U_{RB2}$	high (1)

Programming the SX Microcontroller

Now run the program a couple of times, and note how the contents of **w** change when you increase or decrease the voltage at RB1 by turning the potentiometer left and right, when the program stops at the breakpoint.

1.13.1 The Comparator and Interrupts

Instead of polling the comparator output, it makes sense to trigger an interrupt when the comparator output state changes.

As you know, Port B pins can be used to trigger interrupts when the associated bit in the **WKEN_B** register is cleared. In this case, a positive or negative signal edge at this pin (depending on the bit in the **WKED_B** register) causes an interrupt, and the associated bit in the **WKPD_B** register is set.

If you clear bit 0 in the **WKEN_B** register when the comparator is enabled, a high-low edge of the comparator output state will cause an interrupt, in case the comparator output is enabled.

```
; =====
; Programming the SX Microcontroller
; TUT031.SRC
; =====
include "Setup28.inc"

CMP      equ $08
WKPEN    equ $0b
WKPPD_W  equ $09

RESET    Main

org      $000
ISR
    mode WKPPD_W
    clr  w
    mov  !rb, w          ; clear the "pending" bit
    mode CMP
    mov  !rb, w          ; re-init the comparator and
                        ; get the output state to w.0
    mov  $08, w         ; <-- set a breakpoint here
    reti

org      $100
Main
    mode CMP
    clr  w
    mov  !rb, w          ; Enable comparator inputs and output
    mode WKPPD_W
    mov  !rb, w          ; Clear any "pending" bits
    mode WKPEN
    mov  !rb, #%11111110 ; Enable rb.0 interrupts

: Loop
    jmp : Loop          ; Loop forever...
```

In this program example, we enable the comparator (including its output), and enable RB0 to trigger interrupts on negative signal edges (the default). Then the mainline program has done its work, entering into an endless holding pattern.

When the comparator output state changes from high to low, the ISR is called. In the ISR, we clear the "pending" bit, and then re-initialize the comparator, exchanging the contents of the **CMP** register with **w**. Finally, we save the contents of **w** in **\$08** and return from the ISR.

There may be cases that an interrupt shall be triggered when the comparator output changes from low to high, and when it changes from high to low. To achieve this, connect the RB0 pin to another available Port B pin (RB3...7). Configure this pin as input and clear the associated bits in the **WKEN_B** and **WKED_B** registers for that pin. Now, a positive comparator output edge triggers an interrupt through this input, and a negative edge triggers an interrupt through RB0, as before.

In the ISR, you can test which bit in the **WKPD_B** register is set in order to find out which signal edge caused the interrupt.

1.13.2 The Comparator and the Sleep Mode

The comparator remains enabled in sleep mode when it was enabled before. If the comparator output is enabled, an output change can on one hand, control external components connected to RB0. On the other hand, this change can also wake up the SX, if RB0 is enabled in the **WKEN_B** register.

Please note that the comparator, when enabled, causes higher power consumption. Therefore, if the comparator is not needed in sleep mode, it is a good idea to disable it before executing the sleep instruction.

1.14 System Clock Generation

There are various methods how to generate the system clock for the SX:

- Internal clock generator
- Internal clock generator with external R-C network
- Internal driver with external crystal or ceramic resonator
- External clock signal

1.14.1 The Internal Clock Generator

The internal clock generator makes use of an internal R-C network. At a frequency of 4 MHz, it has a tolerance of $\pm 8\%$. The precision of the generated frequency depends on the ambient temperature, the supply voltage, and on the tolerances of the internal components.

An internal clock divider allows for lower clock frequencies. Bits 6 and 5 in the fuse register must be set according to the table below:

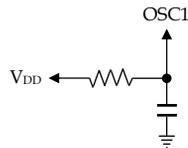
Internal R-C Clock		
FUSE bits		
DIV1 (6)	DIV0 (5)	Frequency
0	0	4 MHz
0	1	1 MHz
1	0	128 kHz
1	1	33 kHz

The assembler directives **OSC4MHZ**, **OSC1MHZ**, **OSC128KHZ** and **OSC32KHZ** can be used to instruct the development system to set the fuse bits accordingly when a program is transferred to the SX.

When the internal clock generator is enabled, SX-pins OSC1 and OSC2 should be left open.

1.14.2 Internal Clock Generator with External R-C Network

The external R-C network is connected to the OSC1 pin according to the schematic below:



In order to configure the SX for this clock mode, add the **DEVICE OSCRC** to your program.

The SX datasheet recommends values between 3 k Ω and 100 k Ω for the resistor, and a capacitor greater than 20 pF.

Again, the precision of the clock frequency depends on the ambient temperature, the supply voltage, and on the tolerances of the internal and external components.

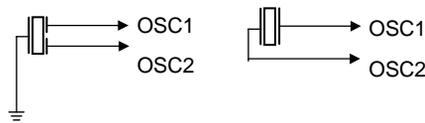
The SX datasheet does not specify the relationship between generated frequencies and R-C time constants. The "breadboard" system used to prepare this book generated a clock frequency of about 1.4 kHz with a 47 k Ω /10 nF R-C network, and 7 kHz with 10 k Ω /10 nF.

1.14.3 External Crystal/Ceramic Resonator

When you use an external crystal or a ceramic resonator, the accuracy of the clock frequency is remarkably increased, and clock frequencies up to the maximum value specified for the SX can be obtained.

The internal driver's level can be adjusted to the requirements of individual crystals/resonators. Therefore, in most cases, no additional external components, like resistors or capacitors are required. The directives **OPTION OSCXT**. . . define how the fuse bits shall be set to obtain a specific level.

Connect a crystal or ceramic resonator to the SX according to the schematics below:



Programming the SX Microcontroller

Depending on the type of crystal or ceramic resonator you are using, it may be necessary to connect an additional resistor to the OSC1 and OSC2 pins, and/or capacitors between ground and the OSC1, and OSC2 pins. Please refer to the data sheets published by Ubicom for more details.

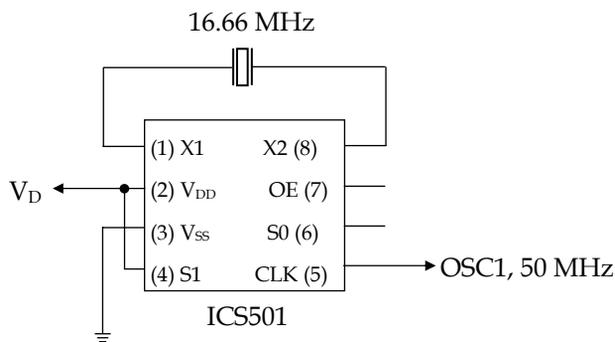
1.14.4 External Clock Signals

When an external clock signal is available in a system, it may be used to clock the SX as well. This signal is fed into the OSC1 pin, and the OSC2 is left open in this case.

1.14.5 External Clock Signal using a PLL

An interesting method to generate an external clock signal is the use of special ICs, designed for this purpose, like the ICS501 "Micro Clock". These components allow generating clock frequencies up to the maximum rate, using a cost-effective crystal.

The schematic below shows the ICS501 configured to generate a 50 MHz clock, derived from a 16.66 MHz crystal:



Depending on the levels at the S0 and S1 pins (V_{DD} , V_{SS} or open), various combinations multiplication factors can be obtained (see the table below).

ICS501		
S1	S2	Factor
0	0	4
0	H	5,3125
0	1	5
H	0	6,25
H	H	2
H	1	3,125
1	0	6
1	H	3
1	1	8

0 = V_{SS} , 1 = V_{DD} , H = open



The OE (Output Enable) pin is connected to V_{DD} via an internal pull-up resistor. When connected to V_{SS} , the clock output is disabled and goes to hi-Z. Nevertheless, this mode is not suited to isolate the SX OSC1 pin from the clock during programming because a voltage of 12.5 V is applied to this pin when a program is transferred to the SX. Use a jumper to open the clock line while programming.

1.14.6 Selecting the Appropriate Clock Frequency

All SX devices allow for clock frequencies up to 50 MHz, some devices can even be clocked at 75 MHz.

Which clock frequency you should select depends on the specific application. Of course could you always drive the SX with the maximum allowed clock frequency, and use timer code to “slow it down” when necessary.

Think of driving a car: Although you know that the engine can make up to say 6,000 Rpm, you normally drive the car at lower rates, just to save gas, and keep the environment a bit cleaner.

Similarly, the SX consumes less power at lower clock rates. Especially for battery-driven systems, this is an important factor. Therefore, for a specific application, you should select a clock rate that is just fast enough to perform the required tasks in the expected time frame.

Programming the SX Microcontroller

When you use a PLL clock generator as described in chapter 1.14.5, and if there are two free I/O pins available, you can even control the S0 and S1 inputs of the ICS501 chip with the SX I/O pins. This makes it possible that the SX can change its clock speed depending on certain events.

As an example let's assume that an application shall monitor the temperature in a room. When this temperature exceeds a certain limit, some characters shall be transmitted via RS-232 to a PC system. Again, when the temperature goes below this level, another string of characters shall be sent.

You could use an NTC as temperature sensor in a voltage divider circuit, and feed the voltage across the NTC into one of the SX analog comparator inputs, where the other comparator input is fed with a reference voltage.

Most of the time, the application program will run in a loop, checking if the comparator output has changed. This can be performed at a low rate because the room temperature usually changes quite slowly.

When it comes to a case that characters must be sent via the serial line, the SX could increase its clock speed for that task, and slow down again after having sent the characters.

While designing SX-based systems, you should also consider RFI problems. At clock speeds of 50 MHz, or even higher, RFI is a critical factor when the board layout, shielding, and filtering is not done properly. Ubicom has released some application notes dealing with this topic that you can download from Ubicom's site.

Own experiences have shown that RFI problems are less critical when a ceramic resonator or a crystal is used instead of an external PLL clock generator due to the fact that the internal driver together with a ceramic resonator or a crystal produces a signal close to a sine wave on the OSC2 pin, where PLLs generate a square wave signal with much more harmonics.



External clock generators, like the ICS501 described before, usually generate square-wave output, where the SX together with a crystal, or ceramic resonator generates clock signals close to an ideal sine-wave. Always be aware of RFI problems that may arise at clock frequencies of 50 or 75 MHz. Harmonics of the clock may "nicely" fall in the range of VHF radio, or ATC frequencies.

1.15 The Program Memory

1.15.1 Organizing the Program Memory



(2.2.3 - 207) When we discussed subroutines, you have learned that the first instruction of a subroutine must be located in the first 256 words of the program memory because the instruction code of a call has space for eight address bits only.

The instruction code of a **jmp** is

101a aaaa aaaa

Here, nine bits are available for a jump address, i.e. a **jmp** instruction can target 512 different words in program memory.

Please test the following program in single steps:

```

; =====
; Programming the SX Microcontroller
; TUT032. SRC
; =====
include "Setup28.inc"
RESET    Mai n

org      $000
Mai n
  jmp    Far

org      $1fe
Far
  jmp    TooFar

org      $200
TooFar
  jmp    Mai n

```

When you execute the **jmp TooFar** instruction, program execution is not continued at **TooFar** but at **Mai n** again.

TooFar specifies the address \$200 or %0010 0000 0000 but the 10th bit does not "fit" into the instruction code of the **jmp**. Therefore, the assembler has simply truncated that bit, and coded a **jmp \$000** instruction instead, i.e. the address represented by the **Mai n** label.

It would be too bad if the SX did not allow a method to address the remaining 1,792 or 3,583 words in the SX 18/20/28 or SX 48/52 devices.

Programming the SX Microcontroller

To address the full range of program memory locations, a "trick" is used, similar to addressing the data memory. You certainly remember the "Crazy Parking Garage" we have used to explain the data memory organization.

Program memory is similarly organized, but this time like a regular parking garage with 512 lots in four or eight decks.

In order to address all memory locations, an 11-bit or 12-bit address is required:

```
000 0000 0000 = $000
111 1111 1111 = $7FF
1000 0000 0000 = $800 (SX 48/52
1111 1111 1111 = $FFF only)
```

The two or three bits marked gray are stored in the upper three bits of the STATUS register. These bits are called "Page Bits" because they select one page of 512 words in program memory.

The full memory addresses for **call** or **jmp** instructions are composed of these page bits plus the bits contained in the instruction code. As a call instruction code only contains eight address bits, the ninth bit is always cleared, and so a call can only target the first 256 words in a page.

After a reset, the upper three STATUS bits are cleared, i.e. by default the first 512 words in program memory will be addressed.

To select another page, it is necessary to set or clear the page bits in the STATUS register. You can do this with **clrb** and **setb** instructions, but there is an easier method available:

1.15.1.1 The PAGE Instruction

Similar to the **bank** instruction that allows switching data memory pages, the **page** instruction is used to change the upper three page bits in STATUS. Here is the previous example, now switching pages:

```
; =====
; Programming the SX Microcontroller
; TUT033. SRC
; =====
include "Setup28.inc"
RESET    Main

org      $000
Main
    jmp   Far

org      $1fe
Far
    page $200
    jmp   TooFar

org      $200
```

```
TooFar
page $000
jmp Main
```

When you single-step this version, you will see that the jumps now perform as expected. Note how the page instructions change the upper bits in STATUS.

The **page** instruction takes the upper three bits of the instruction argument and sets the upper three bits (the page select bits) in STATUS accordingly. The lower bits in the instruction argument are ignored, i.e. they can have any value.



In the example above, the **page** instruction arguments are constants that specify the start of the two memory pages (\$200 and \$000). It makes more sense to define symbolic names for the memory pages, and then using these names instead, e.g.:

```
org $200
PageA equ $
; ...
page PageA
```

As the lower bits of the instruction arguments are ignored, you may also pass any label that is defined within a memory page as instruction parameter, e.g. **page Far** or **page TooFar** in our example.



It is a good idea to place **page** instructions immediately before the **jmp** or **call** instructions that require page switching, although it is possible to have other instructions between **page** and the **jmp** or **call** instructions, but this could be the seed for later disaster.

Imagine what happens if you later insert a **jmp** that shall execute a jump within the current page following the page instruction!

Programming the SX Microcontroller

Try what happens when you replace the **org \$1fe** directive by **org \$1ff** in the last program example. The assembler will display an error message like "Location already contains data" or "Overwriting same program counter location" for the line that contains the **page \$000** instruction.

This looks strange at first glance, so let's track the generated code:

Address Code



```
$1ff           page $200  
$200           jmp TooFar
```

According to the **org \$1ff** directive, the origin for the next instruction following that directive is \$1ff - this is where the code for page \$200 goes. The **jmp TooFar** instruction code is stored in the next location, which is at \$200.

Next, the **org \$200** directive sets the origin for the next instruction following that directive to \$200, this is where the code for page \$000 should go. As this location is already "occupied" by the code for **jmp TooFar**, the assembler is right to complain about that situation.

1.15.1.2 The @jmp and @call Options

The assembler accepts **jmp** and **call** instructions with a leading "@" sign. The @ instructs the assembler to automatically generate a **page** instruction before the **jmp** or **call** instruction code in order to select the correct page for the **jmp** or **call**.

Using this feature makes our example look like this:

```
;  
; =====  
; Programming the SX Microcontroller  
; TUT034. SRC  
; =====  
include "Setup28.inc"  
RESET    Main  
  
org      $000  
  
Main  
  jmp    Far  
  
org      $1fe  
  
Far  
  @jmp   TooFar  
  
org      $200  
  
TooFar  
  @jmp   Main
```

While single stepping this version, you will notice that the assembler has inserted the **page** instruction codes before the **jmp TooFar** and **jmp Main** codes.

1.15.1.3 Subroutine Calls across Memory Pages

For calling subroutines across memory pages, the same rules apply as for jumps across pages with the limitation that subroutines can only begin within the first half of any page.

The memory address for a subroutine call is composed like this:

```

000 0000 0000 = $000
110 1111 1111 = $6FF
1000 0000 0000 = $800 (SX 48/52
1110 1111 1111 = $EFF only)

```

The bits marked gray are taken from the upper STATUS bits, the bits marked black are always cleared, and the lower eight bits are taken from the address part of the call instruction code.

To call a subroutine that is not located in the current page, you must either place a **page** instruction before the **call**, or use the **@call** option to let the assembler generate the required **page** instruction for you.

The program below does not yet work as expected, as you will see when you single-step it:

```

; =====
; Programming the SX Microcontroller
; TUT035. SRC
; =====
include "Setup28.inc"
RESET    Main

org      $100
Main
    jmp   Far

org      $1fd
Far
    call @Farther
    jmp   Main

org      $200
Farther
    clr   w
    ret

```

While you execute the first steps, the program seems to work as expected, but this will dramatically change when you execute the **jmp Main** instruction that follows the subroutine call.

Programming the SX Microcontroller

Instead jumping back to **Main**, the program counter points into "nowhere land" at address \$300. When you look at the upper bits in the STATUS register, you will note the reason for the problem. Here, bit five is set, caused by the page \$200 instruction that the assembler has generated before the subroutine call. As **jmp Main** is coded with an address of \$100, the resulting full address is

%011 0000 0000, i.e. \$300.

According to the definition, the **ret** instruction restores the lower eight bits in PC from the stack. Fortunately, the stack does not only save the lower eight bits of an address, but the complete return address instead. There is an extended return instruction, **retp** (return with page) available that restores the lower eight bits in PC and the upper three bits in STATUS.

Replace the **ret** by a **retp** instruction, and the program will work as expected. Note how bit 5 in status is reset when you step the **retp** instruction.



The **retp** instruction can also be used to return from a subroutine that is located in the same page as the **call** instruction. Therefore, it is a good idea to use the **retp** instruction instead of **ret** when writing generic subroutines that might be placed in another memory page later or in other programs. There is actually no reason, why not using **retp** instead of **ret** throughout a program as **retp** does not require more program memory space or additional instruction cycles.

Let's summarize the most important points you should take care of when you are using two or more program memory pages:

- If the target of a **jmp** or **call** instruction lies in another page than in the current one, select the target page using the **page** instruction before doing the **jmp** or **call**.
- A leading "@" sign in front of a **jmp** or **call** instructs the assembler to insert a **page** instruction before the **jmp** or **call** to automatically select the correct page.
- Returning from a subroutine that is called from another page requires the **retp** instruction in order to restore the page select bits in the STATUS register.



At first glance, it might be an idea to always use the "@"-option together with **jmp** and **call** instructions to make sure that the correct bank is selected. Please keep in mind that the current versions of the SX assemblers always insert a **page** instruction, no matter if it is required or not (i.e. when the **jmp** or **call** targets are located in the current page). This means that additional program memory and clock cycles are unnecessarily wasted.



(2.2.3 - 207) The size of the "large" SX 48/52 controllers is 4,096 words. You can find more information about these devices in the reference part of this book.

1.15.2 How to Organize Program Memory

As jumps and subroutine calls across memory pages require an additional word for the **page** instruction and one additional clock cycle, the parts of a program that belong together logically, should be placed in one memory page, together with any subroutines that are called from this part of the code only.

Good "candidates" for a location in another memory page are program modules that are rarely called.

If the lower half of a page is not large enough to hold all required subroutines, you might also consider placing such subroutines in another page.

We already have discussed the possibility, just to place the entry-points of subroutines within the first half of the page, where the entry-points are **jump** instructions that send the program execution to the upper half of the page. This method takes three additional clock cycles for the jump. When you place the complete subroutine in the first half of another bank, just one additional clock cycle is required for the **bank** instruction, and both the **jump** and the **bank** instruction occupy one word in program memory.

For now, discussing those "minor differences" seem to come close to "cookie cutting" however, as programs get larger and larger, and time resources less and less, you will understand the importance of such "space and time savers".

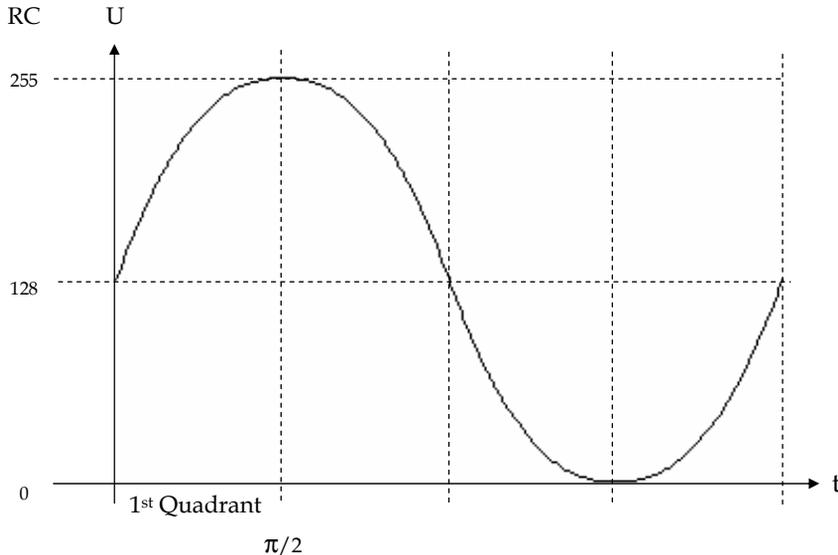
1.16 Tables - RETIW and IREAD

1.16.1 Tables

1.16.1.1 The RETW Instruction

There are applications that require the conversion of one value into another value. Sometimes, this can be obtained by performing simple arithmetic calculations. However, if the two values are related by some non-linear function, things are getting more complicated.

As an example, let's discuss a program for a sine-wave generator. We assume that the port pins RC7...RC0 are connected to an external 8-bit D/A converter. The program shall provide the required values at Port B so that the D/A converter output signal is a sine wave with a certain frequency. You can find a simple D/A converter in the examples section of this book.



The figure above shows the desired the RC output values as a function of time. In general, to generate a sine wave it would be sufficient to determine the required values for one quadrant. The other values can be obtained by mirroring. For simplicity reasons, we assume that here the values shall be determined for the full period.

In the program, we will use a counter (**Ix**) that is incremented at fixed periods. The time it takes to count **Ix** from 0 up to 255 is equivalent to the sine wave's period.

The contents of **I_x** follow a linear time function that must be transformed into a sine function in order to obtain the values that must be sent to the output port. Calculating the sine function using the available arithmetic instructions would be quite difficult and time-consuming. The solution to that problem is a table. Using a table allows us to make the required calculations "outside" the SX only once. If we then place the results in that table, the SX is fast enough to read the table items when necessary.

In other words, the table must contain the values for $f(I_x)$ where $f(I_x)$ is $(\sin(I_x / 255 * 2\pi) + 1) * 128$ here.

To realize such tables, the SX offers the **retw** (return with w) instruction. Its syntax is

retw <Constant>

This is a variant of the "regular" **ret** instruction. Just like **ret**, **retw** returns from a subroutine, and program execution continues with the instruction following the call however, **retw** initializes **w** to the **<Constant>** specified as instruction argument before returning.

This is similar to returning a value from a function in C, where usually integer return values are contained in the accumulator on function return.

The subroutine in the next example is designed to return the first 16 table values:

```

; =====
; Programming the SX Microcontroller
; TUT036. SRC
; =====
include "Setup28.inc"
RESET   Mai n

org     $08
Ix      ds     1

org     $000

; Subroutine returns f(w) in w
;
WToSin
  jmp   pc+w
  retw  127
  retw  130
  retw  133
  retw  136
  retw  139
  retw  143
  retw  146
  retw  149
  retw  152
  retw  155
  retw  158
  retw  161

```

Programming the SX Microcontroller

```
    retw 164
    retw 167
    retw 170
    retw 173

Main
  clr   Ix
loop
  mov   w, Ix
  call  WToSin
  inc   Ix
  clrb  Ix.4
  jmp   loop
```

The mainline program allows you to test the program in single steps using the debugger. **Ix** is the "Index" into the table. As this table has 16 items only, the **clrb Ix.4** instruction limits the possible contents of **Ix** to the range 0...15.

The **jmp pc+w** instruction in the subroutine branches to one of the **retw** instructions that return the function value that is assigned to the corresponding value in **Ix** (passed to the subroutine in **w**).

To make it easier, the assembler allows more than one instruction argument for **retw** instructions:

retw <Constant>, <Constant>, <Constant>, ...

For each **<Constant>** in the comma-delimited list, the assembler generates a separate **retw** instruction.

This allows us to write the program for the complete table like this:

```
; =====
; Programming the SX Microcontroller
; TUT037. SRC
; =====
include "Setup28.inc"
RESET  Main

org    $08
Ix     ds    1
Sin    ds    1

org    $000

; Subroutine returns f(w) in w
;
WToSin
  jmp   pc+w
  retw 127, 130, 133, 136, 139, 143, 146, 149, 152, 155, 158, 161, 164, 167, 170, 173
  retw 176, 178, 181, 184, 187, 190, 192, 195, 198, 200, 203, 205, 208, 210, 212, 215
  retw 217, 219, 221, 223, 225, 227, 229, 231, 233, 234, 236, 238, 239, 240, 242, 243
  retw 244, 245, 247, 248, 249, 249, 250, 251, 252, 252, 253, 253, 253, 254, 254, 254
```

```

retw 254, 254, 254, 254, 253, 253, 253, 252, 252, 251, 250, 249, 249, 248, 247, 245
retw 244, 243, 242, 240, 239, 238, 236, 234, 233, 231, 229, 227, 225, 223, 221, 219
retw 217, 215, 212, 210, 208, 205, 203, 200, 198, 195, 192, 190, 187, 184, 181, 178
retw 176, 173, 170, 167, 164, 161, 158, 155, 152, 149, 146, 143, 139, 136, 133, 130
retw 127, 124, 121, 118, 115, 111, 108, 105, 102, 99, 96, 93, 90, 87, 84, 81
retw 78, 76, 73, 70, 67, 64, 62, 59, 56, 54, 51, 49, 46, 44, 42, 39
retw 37, 35, 33, 31, 29, 27, 25, 23, 21, 20, 18, 16, 15, 14, 12, 11
retw 10, 9, 7, 6, 5, 5, 4, 3, 2, 2, 1, 1, 1, 0, 0, 0
retw 0, 0, 0, 0, 1, 1, 1, 2, 2, 3, 4, 5, 5, 6, 7, 9
retw 10, 11, 12, 14, 15, 16, 18, 20, 21, 23, 25, 27, 29, 31, 33, 35
retw 37, 39, 42, 44, 46, 49, 51, 54, 56, 59, 62, 64, 67, 70, 73, 76
retw 78, 81, 84, 87, 90, 93, 96, 99, 102, 105, 108, 111, 115, 118, 121, 124

```

```

Main
clr    Ix
loop
mov    w, Ix
call   WtoSin
mov    Sin, w
inc    Ix
jmp    loop

```

As this table contains 256 items now, it is no longer necessary to limit the range for **Ix** to 0...16.

When your debugger can display a watch window, it is a good idea to let it display **Ix** and the new **Sin** variable in that window.

Unfortunately, this program has a severe bug, as you will see:

First, single step some loop cycles, and see how the subroutine obtains values from the table, how they are returned, and stored in the **Sin** variable.

Then change the contents of **Ix** at address \$08 to \$fe and continue single stepping. Note what happens in the subroutine when **Ix** holds \$ff. You will notice that the program "hangs" - it executes the **jmp pc+w** instruction forever, and never returns from the subroutine.

There are two reasons for that problem:

The instruction **jmp pc+w** adds the 8-bit value in **w** to the program counter. The **jmp pc+w** instruction is located at address \$000 in program memory, i.e. after reading the instruction, **pc** points to \$001.

When **w** contains \$ff, the sum of **pc** and **w** should result in \$100 which is the address of the last **retw** instruction in the table but internally, the addition **pc+w** is limited to eight bits, and so the overflow into bit 9 of **pc** is lost. This means that **pc** contains \$000 after the addition, i.e. the address of the **jmp pc+w** instruction. This is why the program keeps executing this instruction forever.

Programming the SX Microcontroller



Keep in mind that the instruction **jmp pc+w** only allows to address targets that are located in the lower half of a memory page.

Because the **jmp pc+w** instruction itself "eats up" one memory location in the lower half of the page, one word is missing to implement a table of 256 items this way. In case **jmp pc+w** is not located at the beginning of a page, even more space for table items would get lost.

Fortunately, there is a "trick" how to handle a 265-items table as shown in this example:

```
; =====
; Programming the SX Microcontroller
; TUT038. SRC
; =====
include "Setup28.inc"
RESET   Main

org     $08
Ix      ds    1
Sin     ds    1

org     $000

; Subroutine returns f(w) in w
;
WtoSin
page   SinTable
jmp    w

Main
clr    Ix
loop
mov    w, Ix
call  WtoSin
page  Main
mov    Sin, w
inc    Ix
jmp    loop

org     $200
SinTable
retw   127, 130, 133, 136, 139, 143, 146, 149, 152, 155, 158, 161, 164, 167, 170, 173
retw   176, 178, 181, 184, 187, 190, 192, 195, 198, 200, 203, 205, 208, 210, 212, 215
retw   217, 219, 221, 223, 225, 227, 229, 231, 233, 234, 236, 238, 239, 240, 242, 243
retw   244, 245, 247, 248, 249, 249, 250, 251, 252, 252, 253, 253, 254, 254, 254
retw   254, 254, 254, 254, 253, 253, 253, 252, 252, 251, 250, 249, 249, 248, 247, 245
retw   244, 243, 242, 240, 239, 238, 236, 234, 233, 231, 229, 227, 225, 223, 221, 219
retw   217, 215, 212, 210, 208, 205, 203, 200, 198, 195, 192, 190, 187, 184, 181, 178
retw   176, 173, 170, 167, 164, 161, 158, 155, 152, 149, 146, 143, 139, 136, 133, 130
retw   127, 124, 121, 118, 115, 111, 108, 105, 102, 99, 96, 93, 90, 87, 84, 81
retw   78, 76, 73, 70, 67, 64, 62, 59, 56, 54, 51, 49, 46, 44, 42, 39
retw   37, 35, 33, 31, 29, 27, 25, 23, 21, 20, 18, 16, 15, 14, 12, 11
```

retw	10,	9,	7,	6,	5,	5,	4,	3,	2,	2,	1,	1,	1,	0,	0,	0
retw	0,	0,	0,	0,	1,	1,	1,	2,	2,	3,	4,	5,	5,	6,	7,	9
retw	10,	11,	12,	14,	15,	16,	18,	20,	21,	23,	25,	27,	29,	31,	33,	35
retw	37,	39,	42,	44,	46,	49,	51,	54,	56,	59,	62,	64,	67,	70,	73,	76
retw	78,	81,	84,	87,	90,	93,	96,	99,	102,	105,	108,	111,	115,	118,	121,	124

It is important that the table is originated at the beginning of a page (\$200 in our example).

The **WtoSi n** subroutine now selects the bank where the table is located and then executes a **jmp w** instead of the **jmp pc+w** instruction. This instruction replaces the lower eight bits in the program counter by the contents of **w**, i.e. **w** should contain the offset to the instruction in the selected page that shall be target of the jump, and here, this can be a value from \$00 through \$ff, i.e. 256 different values.

As the complete return address is saved to the stack on subroutine calls, and because **retw** (similar to **retp**) restores the complete address to **pc**, the return to the main program works correctly. In order to make the **jmp Loop** instruction work properly, it is necessary to reset the page select bits that were changed by the **WtoSi n** subroutine to address page \$000 again.

You can find more program examples how to generate waveforms using tables in the "Application Examples" section of this book.

1.16.1.2 Reading Program Memory Using the IREAD Instruction

There is another method to build tables in program memory that are created with the **dw** directive, using the **iread** (immediate read) instruction. Different from the **retw** instruction that returns an 8-bit value, **iread** makes it possible to read all 12 bits stored in a program memory location.

Using iread is a bit "tricky" - let's demonstrate it with the following program:

```

; =====
; Programming the SX Microcontroller
; TUT039.SRC
; =====
include "Setup28.inc"
RESET   Main

org     $08
Ix      ds 1
Data    ds 2

Main
  mov   Ix, #Table

Loop
  mov   m, #Table >> 8

  mov   w, Ix

```

Programming the SX Microcontroller

```
i read
mov  Data, w
mov  Data+1, m
inc  Ix
test Data
sz
    jmp Loop
test Data+1
sz
    jmp Loop
jmp  Main
org  $400
Table
    dw  ' PARALLAX'
    dw  12, 123, 1234, 0
```

When your debugger allows watching variables, configure a watch window that displays the contents of **Data** in character format as well as in 12-bit unsigned decimal format.

At memory page \$400, we have defined the table. As you can see, the **dw** directive is used for initializing locations in the program memory to constant values. The **dw** directive accepts character strings, like 'PARALLAX'. In this case, for each character in the string, the lower eight bits of a memory location will be set to the ASCII code of that character (the upper four bits are cleared). The **dw** directive also accepts numerical constants like 12, 123, 1234, or 0. For each numerical constant, the assembler initializes one 12-bit memory location with the specified value with the upper bits cleared when necessary. The greatest number that can be stored in a memory location is \$fff or 4,095 in decimal.

We use the **Ix** variable as table index. The instruction **mov Ix, #Table** copies the lower eight bits of the table address to **Ix**, i.e. **Ix** now "points" to the first table item.

As **Ix** is only eight bits wide, this is not enough to fully address all table items.

The expression **Table >> 8** is calculated at assembly-time, and its result are the upper four bits of the table address. This value is stored in the **m** register's lower four bits 3...0.

The contents of **Ix** are copied to **w** before executing the **i read** instruction.

The **i read** instruction expects the address to be read in **m:w**. This means that the upper four address bits are expected in the lower four bits (3...0) of **m** and the lower eight bits of the address are expected in **w**. In our example, this is the case because **m** and **w** were set accordingly before.

The 12-bit contents of the addressed memory location is returned by **iread** in **m:w**. Similar to the format that was used to pass an address to **i read**, the result's upper four bits (11..8) are returned in the lower four bits of **m** (3...0) and the lower eight bits of the result (7...0) are returned in **w**.

In our program, the return value is stored to **Data** (lower eight bits) and **Data+1** (upper four bits). It then increments the table index **Ix**.

The program then tests if the value stored in **Data+1: Data** is \$000. In this case, the program loops back to **Main** in order to re-initialize the table index **Ix**. Otherwise, the program stays in **Loop** by reading the next value from the table.

If you test the program in single-step mode or in "slow-motion", you can see, the values read from the table displayed in the watch window.

The size of a table read with `iread` is not limited to 256 items because the instruction uses direct addressing via **m: w**.

1.17 More SX Instructions

In this chapter, we will briefly discuss those SX instructions that we have not used in example programs so far. The clock cycles specified are valid for "turbo mode".

1.17.1 Compare Instructions

This family of instructions consists of compound instructions, i.e. the assembler generates two or more instruction codes instead.



Note that these instructions must not immediately follow a skip instruction.

Please also note that the Carry flag must be set or cleared in case the CARRYX option is enabled. You will find the necessary clear or set instruction together with the instruction syntax in parentheses.

Note that compound statements must not immediately follow next to any of the conditional skip instructions described here.

1.17.1.1 CJA (Compare and Jump if Above)

Syntax: (clc)

cja op1, op2, Address

A jump to the specified address will be executed when **op1** is greater than **op2**. There are two variants of this instruction:

cja fr, #Constant, Address

cja fr1, fr2, Address

The instructions require 4 words in memory and 4 clock cycles (6 if the jump is taken). The C, DC, and Z flags and the W register are changed.

1.17.1.2 CJAE (Compare and Jump if Above or Equal)

Syntax: (stc)

cjae op1, op2, Address

A jump to the specified address will be executed when **op1** is greater than or equal to **op2**. There are two variants of this instruction:

cjae fr, #Constant, Address

cjae fr1, fr2, Address

The instructions require 4 words in memory and 4 clock cycles (6 if the jump is taken). The C, DC, and Z flags and the W register are changed.

1.17.1.3 CJB (Compare and Jump if Below)

Syntax: (stc)

cj b op1, op2, Address

A jump to the specified address will be executed when **op1** is smaller than **op2**. There are two variants of this instruction:

cj b fr, #Constant, Address

cj b fr1, fr2, Address

The instructions require 4 words in memory and 4 clock cycles (6 if the jump is taken). The C, DC, and Z flags and the W register are changed.

1.17.1.4 CJBE (Compare and Jump if Below or Equal)

Syntax: (clc)

cj be op1, op2, Address

A jump to the specified address will be executed when **op1** is smaller than or equal to **op2**. There are two variants of this instruction:

cj be fr, #Constant, Address

cj be fr1, fr2, Address

The instructions require 4 words in memory and 4 clock cycles (6 if the jump is taken). The C, DC, and Z flags and the W register are changed.

1.17.1.5 CJE (Compare and Jump if Equal)

Syntax: (stc)

cj e op1, op2, Address

A jump to the specified address will be executed when **op1** is equal to **op2**. There are two variants of this instruction:

cj e fr, #Constant, Address

cj e fr1, fr2, Address

The instructions require 4 words in memory and 4 clock cycles (6 if the jump is taken). The C, DC, and Z flags and the W register are changed.

Programming the SX Microcontroller

1.17.1.6 CJNE (Compare and Jump if Not Equal)

Syntax: (stc)

cj ne op1, op2, Address

A jump to the specified address will be executed when **op1** is not equal to **op2**. There are two variants of this instruction:

cj ne fr, #Constant, Address

cj ne fr1, fr2, Address

The instructions require 4 words in memory and 4 clock cycles (6 if the jump is taken). The C, DC, and Z flags and the W register are changed.

1.17.2 Decrement/Increment with Jump

1.17.2.1 DJNZ (Decrement and Jump if Not Zero)

Syntax:

dj nz fr, Address

Register **fr** is decremented. If the content of **fr** is not zero after the decrement, the jump to the specified address will be executed. The instruction requires 2 words in memory and 2 clock cycles (4 if the jump is taken). No flags are changed.

1.17.2.2 IJNZ (Increment and Jump if Not Zero)

Syntax:

ij nz fr, Address

Register **fr** is incremented. If the content of **fr** is not zero after the increment, the jump to the specified address will be executed. The instruction requires 2 words in memory and 2 clock cycles (4 if the jump is taken). No flags are changed.

1.17.3 Conditional Jumps

1.17.3.1 JNB (Jump if Not Bit set)

Syntax:

jnb fr.Bit, Address

When the specified bit in **fr** is clear, the jump to the specified address will be executed. The instruction requires 2 words in memory and 2 clock cycles (4 if the jump is taken). No flags are changed.

1.17.3.2 JNC (Jump if Not Carry set)

Syntax:

jnc Address

When the carry flag is clear, the jump to the specified address will be executed. The instruction requires 2 words in memory and 2 clock cycles (4 if the jump is taken). No flags are changed.

1.17.3.3 JNZ (Jump if Not Zero)

Syntax:

jnz Address

When the zero flag is clear, the jump to the specified address will be executed. The instruction requires 2 words in memory and 2 clock cycles (4 if the jump is taken). No flags are changed.

1.17.4 Conditional Skips

1.17.4.1 CSA (Compare and Skip if Above)

Syntax: (clc)

csa op1, op2

The next instruction will be skipped when **op1** is greater than **op2**. There are two variants of this instruction:

csa fr, #Constant

csa fr1, fr2

The instructions require 3 words in memory and 3 clock cycles (4 if the skip is executed). The C, DC and Z flags and the W register are changed.

1.17.4.2 CSAE (Compare and Skip if Above or Equal)

Syntax: (stc)

csae op1, op2

The next instruction will be skipped when **op1** is greater than or equal to **op2**. There are two variants of this instruction:

csae fr, #Constant

csae fr1, fr2

The instructions require 3 words in memory and 3 clock cycles (4 if the skip is executed). The C, DC and Z flags and the W register are changed.

1.17.4.3 CSB (Compare and Skip if Below)

Syntax: (stc)

csb op1, op2

The next instruction will be skipped when **op1** is smaller than **op2**. There are two variants of this instruction:

csb fr, #Constant

csb fr1, fr2

The instructions require 3 words in memory and 3 clock cycles (4 if the skip is executed). The C, DC and Z flags and the W register are changed.

1.17.4.4 CSBE (Compare and Skip if Below or Equal)

Syntax: (stc)

csbe op1, op2

The next instruction will be skipped when **op1** is smaller than or equal to **op2**.

There are two variants of this instruction:

csbe fr, #Constant

csbe fr1, fr2

The instructions require 3 words in memory and 3 clock cycles (4 if the skip is executed). The C, DC and Z flags and the W register are changed.

1.17.4.5 CSE (Compare and Skip if Equal)

Syntax: (stc)

cse op1, op2

The next instruction will be skipped when **op1** is equal to **op2**. There are two variants of this instruction:

cse fr, #Constant

cse fr1, fr2

The instructions require 3 words in memory and 3 clock cycles (4 if the skip is executed). The C, DC, and Z flags, and the W register are changed.

1.17.4.6 CSNE (Compare and Skip if Not Equal)

Syntax: (stc)

csne op1, op2

The next instruction will be skipped when `op1` is not equal to `op2`. There are two variants of this instruction:

`csne fr, #Constant`

`csne fr1, fr2`

The instructions require 3 words in memory and 3 clock cycles (4 if the skip is executed). The C, DC, and Z flags, and the W register are changed.

1.17.5 MOV and Conditional Skip

1.17.5.1 MOVSZ (MOVE and Skip if Zero)

Syntax:

`movsz w, ++fr`

`movsz w, --fr`

The incremented or decremented value of the specified file register is copied to `w`. The next instruction will be skipped when `w` contains zero. The instructions require 1 word in memory and 1 clock cycle (2 if the skip is executed). The contents of `fr` remain unchanged and no flags are changed.

1.17.6 NOP (No OPeration)

Syntax:

`nop`

This instruction requires 1 word in memory and 1 clock cycle. It can be used to fine-tune the delay time of a loop, for example.

1.17.7 SKIP

Syntax:

`ski p`

This instruction unconditionally skips the next instruction, it must not be followed by a compound instruction. It requires 1 word in memory and 2 clock cycles.

You may wonder what an instruction is good for that always performs a skip over the next instruction, so here is an example:

```

Delay1
  mov w, #8
  ski p
Delay2
  mov w, #16

```

Programming the SX Microcontroller

```
    mov Counter, w
: Loop
    decsz Counter
    jmp : Loop
ret
```

This is a subroutine having two entry-points, **Del ay1** and **Del ay2**. When you call **Del ay1**, a short delay will be the result, calling **Del ay2** causes a delay that is approximately twice as long. As you can see, when you call **Del ay1**, the **skip** instruction is used to jump over the initialization of **w** for the longer delay time that is required for **Del ay2**.

1.18 Virtual Peripherals

This chapter deals with the most powerful feature of the SX controllers: The Virtual Peripherals, or shortly “VPs”. (“Virtual Peripheral”, and “VP” are trademarks of Uvicom.)

Compared to other microcontrollers, there are no SX controllers available with various internal peripheral units like UARTs, A/D converters, I²C interfaces, PWM (pulse width modulation) drivers, etc. All SXes only come with a comparator, and the “larger” devices have an additional timer. (The new IP2020 devices also contain serializers/deserializers).

This means that the number of different microcontroller devices offered by Uvicom is much smaller compared to the lists of other manufacturers. At Uvicom, you can only make a selection between devices with a different number of I/O ports, the maximum clock frequency, the ambient temperature, and the package. This makes it easy to keep an overview, and helps to reduce the required items on stock.

Due to the fast throughput (up to 100 Mips) of the SX controllers, required peripheral units can be realized just in software as Virtual Peripherals. Some applications require a minimum of additional external components.

In almost all cases, when VPs are realized in software, an exact timing is required. The timer-controlled interrupts that are supported by the SX devices is an ideal basis for such timing. This concept also allows to run the VPs “in the background” as separate tasks.

In general, the code of VPs needs not always to be executed within an interrupt service routine (ISR) – any program module that replaces a hardware peripheral can be called a VP.

It is also possible to combine two or more VPs for different tasks within one application, as long as the timing is guaranteed for all VPs.

Uvicom, and other sources offer a wide collection of Virtual Peripherals that you can download from the Internet free of charge.

1.18.1 The Software UART, a VP Example

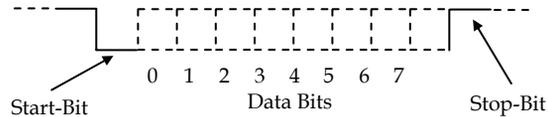
As an example, let's analyze the code for a UART, realized as Virtual Peripheral. The original code was first published by Parallax, Inc. and it is a good example how a task can be realized with just a few instructions when they are combined in a clever way.

Let's first look at the most important points of serial data communications.

The incoming serial data are fed into an input line of the UART, and then converted into parallel data, where the outgoing data are converted from parallel representation into a serial stream of bits that are available at an output line.

Programming the SX Microcontroller

The following diagram shows the timing for transmitting one byte of data:



The transfer speed is specified in Baud, where this value defines how many bits are sent per second. A rate of 19,200 Baud means that 19,200 bits are sent per second, and so the transfer time (or the bit period) for each bit is $1/19,200 \approx 52 \mu\text{s}$.

For each byte to be transferred, the signal line is pulled to low level for one bit period. This indicates the start of a transmission, and therefore this first null bit is called the start bit.

Next, the signal line is pulled to high or low eight times for each data bit, depending on the status of the bits. Please note that in most cases the bits are transferred in "reverse order", i.e. the low order bit comes first, and the high order bit comes last.

Finally, the signal line is pulled to high level for at least one bit period to indicate the end of transmission. Therefore, this bit is called the stop bit.

This is the sample code:

```

; =====
; Programming the SX Microcontroller
; TUTO40
; =====
include "Setup28.inc"
RESET   Main

; ***** Program Variables *****
;
; Port assignment: Bit variables
;
rx_pin   EQU    ra.2      ; UART receive input
tx_pin   EQU    ra.3      ; UART transmit output

; ***** Register definitions (bank 0)
;
org      8                ; First register address in main memory
; bank
temp     ds      1        ; Temporary storage
byte     ds      1        ; Temporary UART shift register
flags    DS      1        ; Program flags register
number_low ds      1      ; Low receive byte
number_high ds      1     ; High receive byte
hex      ds      1        ; Value of received hex number
string   ds      1        ; Indirect pointer for text output

```

```

rx_flag      EQU      flags.5           ; Signals reception of one byte

serial       org      10h               ; Variables in bank 3
              =        $                ; UART bank

tx_high      ds       1                 ; Low transmit byte
tx_low       ds       1                 ; High transmit byte
tx_count     ds       1                 ; Number of remaining bits to be sent
tx_divide    ds       1                 ; Counter for transmit timer (/16)
rx_count     ds       1                 ; Number of bits received
rx_divide    ds       1                 ; Counter for receive timer
rx_byte      ds       1                 ; Buffer for received bytes

; The next three parameters determine the baud rate of the UART.
; The values of baud_bit, and int_period control the baud rate as follows:
;
; Baud rate = 50 MHz / (2 ^ baud_bit * int_period * RTCC_prescaler)
; Important: - 1 <= baud_bit <= 7
;            - int_period must be less than 256 and longer than
;              the total time required by the ISR instructions. Changing
;              int_period will also influence the timing of other VPs
;              that are possibly executed within the ISR.
;            - start_delay must be set to (2 ^ baud_bit) * 1,5 + 1.
;
; Values for various baud rates:
;
; *** 2400 Baud (For baud rates below 2,400, the RTCC prescaler must be
;               activated.)
;baud_bit     =         7
;start_delay  =       128+64+1
;int_period   =       163
;
; *** 9600 baud
;baud_bit     =         5
;start_delay  =       16+8+1
;int_period   =       163
;
; *** 19,200 Baud
;baud_bit     =         4
;start_delay  =       16+8+1
;int_period   =       163
;
; *** 38,400 baud
;baud_bit     =         3
;start_delay  =       8+4+1
;int_period   =       163
;
; *** 57,600 baud
;baud_bit     =         2
;start_delay  =       4+2+1
;int_period   =       217
;
; *** 115.2k baud
;baud_bit     =         1

```

Programming the SX Microcontroller

```
;start_delay = 2+1+1
;int_period = 217
;
; *** 230.4k baud (for higher baud rates, int_period must be reduced - see
; above!)
;baud_bit = 0
;start_delay = 1+0+0
;int_period = 217
;
;*****
; Virtual Peripheral UART
;
; Length: 67 bytes (total)
; Authors: Chip Gracey, President, Parallax Inc.
; modified by Craig Webb,
; Consultant to Scenix Semiconductor, Inc.
; Written: 97/03/10 to 98/7/09
;
;*****
;***** ISR CODE *****
; Remark: The ISR code must always start at address $000. Time-critical
; VP code (e.g. for A/D converters) should be placed before the
; code of VPs that have a variable execution time, like this UART
; code.
interrupt ORG $000
;
;**** Virtual Peripheral: Universal Asynchronous Receiver Transmitter (UART)
; This routine sends and receives serial RS232 data. It is configured for
; the frequently used "8-N-1" format (8 Bits, no parity, 1 stop bit).
; RECEIVE: The rx_flag will be set as soon as a valid data byte is available.
; It is then the responsibility of the code that processes the data
; byte to reset the rx_flag again.
; TRANSMIT: The transmit routine expects the inverted data in the register-
; pair tx_high, and tx_low, where the byte to be sent must be
; inverted and stored in tx_high. In tx_low, Bit 7 must be set,
; and the other bits are ignored. Then the number of bits to be sent
; (10 = 1 start bit + 8 data bits + 1 stop bit) must be stored in
; tx_count. As soon as tx_count contains a value > 0, the transmitter
; starts sending the data. The calling application may test tx_count
; in order to determine if a transmission is still in progress, i.e.
; if tx_count > 0.
; This VP has variable execution times. Therefore, it should be located behind
; time-critical code (e.g. for A/D converters, timers, PWMs, etc.) in the ISR.
;
; Note: The transmit and receive code is independent from each other. If
; the transmitter or the receiver is not required in an application,
; its code may be deleted. Take care NOT to delete the initial
; bank serial instruction.
;
; Input variables: tx_low (bit 7 only)
; tx_high, tx_count
;
; Output variables: rx_flag, rx_byte
```

```

; Required clock cycles (in turbo mode):
;
; Transmit: 9 cycles (while idle)
;           19 cycles (while sending)
;           + 1 cycle for the common bank instruction
;
; Receive: 9 cycles (while idle)
;          16 cycles (at start)
;          13 cycles (while reading the next bit)
;          17 cycles (at the end of reception)
;
; bank serial ; Select the "serial" bank ; 1
: transmit
  clrb tx_divide.baud_bit ; Clear transmit timer flag ; 2
  inc tx_divide ; Increment the counter ; 3
  stz ; Set Z flag for next instruction ; 4
  snb tx_divide.baud_bit ; Execute the transmit routine on ; 5
  ; (2 ^ baud_bit)-th interrupt.
  test tx_count ; Are we sending data? ; 6
  jz :receive ; no, continue with the receiver ; 7
  clc ; yes, prepare stop bit, and ; 8
  rr tx_high ; shift to next bit ; 9
  rr tx_low ; ; 10
  dec tx_count ; Decrement bit counter ; 11
  movb tx_pin, /tx_low.6 ; Output next bit ; 12
: receive
  movb c, rx_pin ; Store received bit in carry flag ; 13
  test rx_count ; Are we receiving? ; 14
  jnz :rxbit ; yes, continue receiving ; 15
  mov w, #9 ; no, prepare 9 bits ; 16
  sc ; if no start bit, continue ; 17
  mov rx_count, w ; if start bit, set bit counter ; 18
  mov rx_divide, #start_delay ; Set 1.5 bit periods ; 19
: rxbit
  djnz rx_divide, :rxdone ; In the "middle" of the next bit? ; 20
  setb rx_divide.baud_bit ; yes, set 1 bit period ; 21
  dec rx_count ; Last bit? ; 22
  sz ; no, ; 23
  rr rx_byte ; save bit ; 24
  snz ; yes, ; 25
  setb rx_flag ; set flag ; 26
: rxdone
  mov w, #-int_period ; Interrupt every int_period ; 27
  ; clock cycles
  ; (163 for 19, 200 Bd)
: end_int
  reti ; Leave the ISR ; 28

; ***** PROGRAM DATA *****
;
;

```

Programming the SX Microcontroller

```
; Character strings for user interface (must be located in the first half
; of a program memory page).
;
_hello      dw      13, 10, 13, 10, 'SX 2400-230.4K UART Virtual Peripheral Demo'
_prompt     dw      13, 10, '>', 0
;
; ***** SUBROUTINES *****
;
; Read a byte from the UART
;
get_byte
    jnb     rx_flag, $           ; Wait until a byte has been received ; 29
    clrb   rx_flag             ; Reset rx_flag ; 30
    mov    byte, rx_byte       ; Save Byte (w also contains Byte) ; 31
                                ; "Fall through", to echo the received
                                ; character.
;
; Send a byte via the UART
;
send_byte
    bank   serial              ; 32
:wait
    test   tx_count            ; Wait for a pause ; 33
    jnz    :wait               ; 34

    not    w                   ; Prepare bits (negative logic) ; 35
    mov    tx_high, w          ; Save data byte ; 36
    setb   tx_low.7            ; Set start bit ; 37
    mov    tx_count, #10       ; 1 start + 8 data + 1 stop bit ; 38
    retp                                ; Return with page adjust ; 39
;
; Send a character string that begins at the address in w.
;
send_string
    mov    string, w           ; Save the address ; 40
:loop
    mov    w, string           ; Read next character ; 41
    mov    m, #0               ; using an indirect address. ; 42
    iread                                ; ; 43
    mov    m, #$0f             ; Adjust m register ; 44
    test   w                    ; Last character? ; 45
    snz                                ; No, continue ; 46
    retp                                ; Yes, return with page adjust ; 74
    call   send_byte           ; Send character ; 48
    inc   string               ; Address next character ; 49
    jmp   :loop                ; Continue until the end of text ; 50
;
; ***** MAIN PROGRAM CODE *****
;
    ORG    100h                ; 51
;
; Program execution after power-on or reset starts here.
;
Main
```

```

    mov    ra,    #%1011          ;Initialize port RA          ; 52
    mov    !ra,  #%0100          ;Set RA's inputs and outputs ; 53
include "Clr2x.inc"

    mov    !option, #%10011111   ;Enable RTCC interrupt    ; 59
; Main program loop
    mov    w,    #_hello         ;Send the hello string    ; 60
    page   send_string          ;                          ; 61
    call   send_string          ;                          ; 62
:loop
    call   get_byte             ;Get a byte from the UART ; 63
    ; <Add more program code here, as required>

    jmp    :loop                ;Loop back for next character ; 64
END                                           ;End of program code      ; 65

```

A Virtual Peripheral should be designed in a way that performs its task “in the background”, so that the main application does not “see” it. For example, the **send_byte** subroutine tests **tx_count** in line 33 if it contains zero to find out if the UART is ready to send another byte. At this place, the subroutine does not “know” that the UART is realized in software. **tx_count** could be the status register of a hardware UART as well. The same rules apply to the remaining interface to the UART VP.

In this sample program, there are actually two Virtual Peripherals, one is the serial transmitter, and the other is the serial receiver.

Now let’s analyze the details of the program code:

1.18.1.1 The Transmitter

The code for the transmitter is a bit easier to understand. Therefore, let’s have a look at it first. Important is the correct timing, and so we’ll find out how the transmitter timing is generated.

At 19,200 Baud, the ISR is called every 163 clock cycles because `retiw` with `w = - 163` is executed in line 28. At a system clock of 50 MHz, the ISR is called every 3,26 μ s, and so, the ISR must be called 16 times for one bit period of $\approx 52 \mu$ s.

Programming the SX Microcontroller

Lines 2 to 7 read

```
    clrb    tx_divide.baud_bit    ; Clear transmit timer flag      ; 2
    inc     tx_divide             ; Increment the counter          ; 3
    stz     tx_divide.baud_bit    ; Set Z flag for next instruction ; 4
    snb     tx_divide.baud_bit    ; Execute the transmit routine on ; 5
                                     ; (2 ^ baud_bit)-th interrupt.
    test    tx_count             ; Are we sending data?          ; 6
    jz      :receive             ; no, continue with the receiver ; 7
```

where **baud_bit** is 4 for 19,200 Baud, and **tx_divide** is the timer counter for the transmitter.

In line 2 bit 4 (the **baud_bit**) in **tx_divide** is always cleared, and so, the contents of **tx_divide** can never become greater than 15, i.e. it can hold 16 different values from 0...15, and this equal to the number of 16 ISR calls mentioned above.

In line 3, **tx_divide** is incremented, and in line 4 the Z flag is set “in preparation” (you’ll soon see why). When the value of **tx_divide** has not reached 16 (i.e. bit 4 is clear), the ISR was not called 16 times yet. In this case, line 6 is skipped, and execution continues with line 7. The jump to **:receive** will be executed because we have set the Z flag before.

When bit 4 in **tx_divide** is set, it is time to send a bit in case there are more bits to be sent. Therefore **test tx_count** in line 6 is executed. Now, the Z flag is set or cleared depending on the contents of **tx_count**. If it is set, no more bytes are to be sent, and the jump to **:receive** is performed. Otherwise, execution continues with line 8.

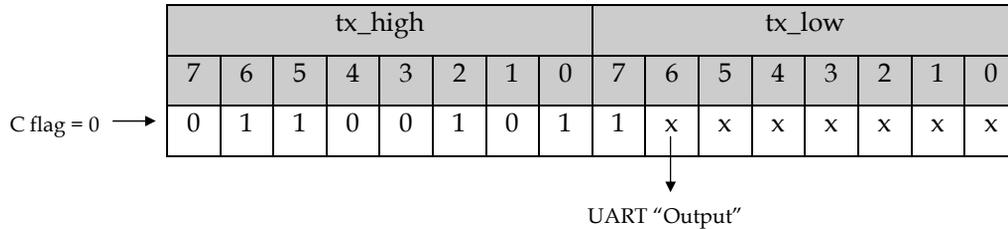
This is an example for real clever coding! Setting the Z flag “in preparation” provides that the **jz :receive** instruction is executed when it is not yet time for another bit and when there are no more bits.

The next lines read:

```
    clc                ; yes, prepare stop bit, and      ; 8
    rr     tx_high     ; shift to next bit              ; 9
    rr     tx_low      ;                               ; 10
    dec    tx_count    ; Decrement bit counter          ; 11
    movb   tx_pin, /tx_low.6 ; Output next bit                ; 12
```

At this place it is important to know how the bits are arranged in **tx_high** and **tx_low** before sending the start bit:

Let's assume that we want to send the bit pattern 10011010 (or \$9a).



According to the definition of the transmitter interface, **tx_high** must contain the inverted value of the byte to be sent (i.e. instead of 10011010, we store 01100101), and bit 7 in **tx_low** must be set (the remaining bits are don't care).

In line 8, the carry flag is cleared, and so a zero bit is "rotated in" when **rr tx_high** is executed. The contents of **tx_high** is shifted one bit position to the right, and the contents of bit 0 is moved into the C flag.

The **rr tx_low** instruction in line 10 shifts bit 7 to bit 6, and the former contents of bit 0 in **tx_high** (now in the C flag) is moved into bit 7 of **tx_low**.

Bit 6 of **tx_low** is "connected" to the serial output of the transmitter. The state of this bit is copied to **tx_pin** in line 12. As the argument of the **movb** instruction is **/tx_low. 6**, the bit is negated in order to compensate the negation when the send byte was stored in **tx_high** before.

At start of a transmission for a new data byte, the output will go low in any case because bit 7 in **tx_low** is 1. This is how the start bit is generated.

In line 11, **tx_count** with its initial contents of 10 is decremented in order to stop the transmitter after 10 cycles (see line 6). At the 10th (and last) cycle, the zero bit that was initially shifted into **tx_high. 7** has "arrived" at **tx_low. 6** (the output). So **tx_pin** will be finally set to high level, and this is how the stop bit is generated.

It is amazing to see how a complete UART transmitter can be realized in just eleven lines of program code!

1.18.1.2 The Receiver

Before getting into the receiver details, some initial thoughts are in order:

- A start bit may occur at the receiver input at arbitrary times. Therefore, it is important to test the input line as often as possible. In the **:receive** section of the code, this is done at every call of the ISR, i.e. every 3.26 μ s at 19,200 Baud.

Programming the SX Microcontroller

- When a start bit has been detected, a delay of 1.5 bit periods is necessary in order to “hit” the status of the first data bit in the “middle” of its period.
- After reading the first data bit, the delay between the reads must be just one bit period in order to “hit” the status of the remaining bits in the “middle” of their periods.

```
: receive
  movb    c, rx_pin          ;Store received bit in carry flag    ; 13
  test   rx_count           ;Are we receiving?                ; 14
  jnz    :rxbit             ; yes, continue receiving          ; 15
  mov    w, #9              ; no, prepare 9 bits                ; 16
  sc     ; if no start bit, continue          ; 17
  mov    rx_count, w        ; if start bit, set bit counter      ; 18
  mov    rx_divide, #start_delay ;Set 1.5 bit periods          ; 19
```

In line 13, the status of the receiver input line is stored in the C flag, and in line 14, we test if the receiver is already active. This is the case when **rx_count** is greater than zero. In this case, execution continues in line 20 (: **rx_bit**).

When the receiver is not yet active but when the C flag (holding the status of the input line) is clear, we have just received a start bit. In this case, **rx_count** is set to 9 (this value was stored in w “in preparation” before in line 16).



As you know, you must not follow a skip instruction with a compound statement, like **mov rx_count, #9**. Lines 16 and 18 are a good example how to move a “constant” value into a register by storing this value in W before.

Setting **rx_count** to a value greater than zero indicates that the receiver is active (this test is performed in line 14, as described above).

In line 19, **rx_divide** receives the value of **start_delay**. At 19,200 Baud, **start_delay** is defined as 25. The ISR is invoked every 3.26 μ s, so this counter provides a delay of 81.5 μ s. Bit period times 1.5 at 19,200 Baud is $1.5 * 52 \mu\text{s} = 78 \mu\text{s}$. This means that at a delay of 81.5 μ s the next bit is not exactly “hit” in the middle, but a bit later. Due to rounding errors, it is not possible to exactly generate a delay of 78 μ s here, but it is still an acceptable timing.

No matter if a start bit has been detected or not, to following lines will be executed:

```
: rxbit
  djnz   rx_divide, :rxdone ;In the “middle” of the next bit? ; 20
  setb   rx_divide.baud_bit ; yes, set 1 bit period          ; 21
  dec    rx_count          ;Last bit?                          ; 22
  sz     ; no,              ; 23
  rr     rx_byte           ; save bit                          ; 24
  snz    ; yes,            ; 25
  setb   rx_flag          ; set flag                          ; 26
: rxdone
```

As long as the receiver is not busy, **rx_divide** is loaded with 25 in line 19 each time the ISR is invoked. This means that **rx_divide** contains 24 after the decrement in line 20, and thus the jump to : **rxdone** will always be executed.

While the receiver is active, **rx_divide** will reach a value of zero after 16 ISR calls (or 25 calls for the start bit). In this case, bit **baud_bit** in **rx_divide** is set in line 21. At 19,200 Baud, **baud_bit** is 4. As **rx_divide** is zero here, setting its bit 4 changes its contents to 16, and this is exactly the value that is required for the “regular” bit period delay.

In line 22 **rx_count**, the counter for the received bits is decremented. If it is not yet zero, **rx_byte** is shifted one bit position to the right in line 24, and the status of the newly received bit in the C flag is moved into bit 7 of **rx_byte**.



Note that the C flag has already received the status of the input line in line 13. Therefore it is important that none of the instructions between lines 13 and 24 modify the C flag.

When **rx_count** finally reaches zero, the instruction in line 26 is executed, and **rx_flag** is set. This is the signal for the calling routine that a new byte has been received which is available in **rx_byte**.

Although a data byte only contains of 8 bits, the **rr rx_byte** instruction in line 24 was executed 9 times. This is correct because the first bit read was the start bit, and after 9 shifts, this bit has “dropped off” bit 0 of **rx_byte**.

Again, the code for the receiver with its 14 lines is surprisingly small due to the clever coding!

Now let’s look at the rest of the program code that is used to test and demonstrate the UART VP.

1.18.1.3 Utility Routines

; Read a byte from the UART

;

get_byte

jnb	rx_flag, \$; Wait until a byte has been received ;	29
clrb	rx_flag	; Reset rx_flag ;	30
mov	byte, rx_byte	; Save Byte (w also contains Byte) ;	31

This subroutine is used to receive one character from the UART VP. It keeps waiting in line 29 until **rx_flag** is set to indicate that a new character is available. It is important to clear **rx_flag** in line 30. Finally, the new data byte is copied from **rx_byte** to **byte**. Because the **mov** instruction in line 31 is a compound instruction, the W register also contains the received byte.

get_byte is not terminated with a **ret** instruction. Therefore, program execution “falls through” to line 32 to the **send_byte** subroutine which sends the character contained in W.

Programming the SX Microcontroller

```
send_byte
bank      serial                                ; 32

: wait
  test    tx_count                            ; Wait for a pause      ; 33
  jnz     : wait                               ; 34

  not     w                                  ; Prepare bits (negative logic) ; 35
  mov     tx_high, w                          ; Save data byte             ; 36
  setb    tx_low.7                            ; Set start bit              ; 37
  mov     tx_count, #10                       ; 1 start + 8 data + 1 stop bit ; 38
  retp                                         ; Return with page adjust     ; 39
```

The **bank serial** instruction in line 32 is important because the subroutine needs to access variables in this bank, and we cannot assume that this bank is always selected when the subroutine is called. (In **get_byte**, it is not necessary to select the **serial** bank because this subroutine references variables in this bank only after the ISR was executed which selects this bank anyway).

Lines 33 and 34 cause the **send_byte** subroutine to wait while the UART transmitter is still busy sending a character.

According to the interface definition of the UART transmitter VP, the inverted byte to be sent must be stored in **tx_high**, and bit 7 in **tx_low** must be set. This happens in lines 35 to 37. In line 38, the required bit count is stored in **tx_count** which actually starts the transmitter as soon as **tx_count** is greater than zero.

The subroutine is terminated with a **ret** instruction in line 39. It makes sense to use **ret** instead of **ret** here in order to place this generic subroutine into another page of program memory if necessary.

```
send_string
mov       string, w                            ; Save the address          ; 40

: loop
  mov     w, string                            ; Read next character       ; 41
  mov     m, #0                                ; using an indirect address. ; 42
  i read                                     ;                               ; 43
  mov     m, #$0f                              ; Adjust the m register     ; 44
  test    w                                    ; Last character?           ; 45
  snz     ; No, continue                        ;                               ; 46
  retp                                         ; Yes, return with page adjust ; 74
  call    send_byte                            ; Send character            ; 48
  inc     string                               ; Address next character     ; 49
  jmp     : loop                               ; Continue until the end of text ; 50
```

This subroutine is used to send a string of characters stored in program memory via the UART transmitter. Each character is read from program memory using the **i read** instruction in line 43 and then sent by calling the **send_byte** subroutine. The program loop is repeated until a terminating zero-byte is found in the string.

Again, this subroutine is terminated with a **ret** instruction which makes it “generic”, i.e. it may be located in any lower half of a page of program memory.

1.18.1.4 The Main Program

The main program begins in line 52 after the **Main** label. It first clears all registers in the data memory as explained before in chapter 1.5.6. Then the following instructions are executed:

```

mov    w, #_hello                ;Send the hello string      ; 60
page   send_string              ;                          ; 61
call   send_string              ;                          ; 62

:loop
call   get_byte                 ;Get a byte from the UART    ; 63

      ; <Add more program code here, as required>

jmp    :loop                    ;Loop back for next character ; 64

```

First, the string that is stored in program memory starting at **_hello** is sent via the UART transmitter, and then the main program enters into an endless loop where it echoes all characters received from the UART receiver back to the transmitter.

The levels on the I/O pins (**rx_pin**, and **tx_pin**) of the SX device are 0 and +5 Volts. In order to connect these signals to a standard RS-232 port of a PC that runs a terminal program, it is necessary to convert these levels to +12/-12 Volts. Some available prototype boards like the SX-Key Demo Board from Parallax have an RS-232 driver chip (MAX-232) installed on the board that interfaces the SX signals to a 9-pin SUB-D connector which allows you to directly connect the board to one of the PC's COM ports. Configure the terminal program running on the PC to communicate with the right COM port, and set it to 19,200 Baud, 8-N-1. You should also turn the internal echo, and any handshaking off.

When you start the program on the SX, the terminal program on the PC should display

```
SX 2400-230. 4K UART Virtual Peripheral Demo
```

```
>
```

When you then enter any characters via the PC keyboard, these characters should be echoed in the terminal display area.

Programming the SX Microcontroller



If you want to supply the system clock from the SX-Key to the SX during this experiment, the PC's COM port that controls the SX-Key cannot be used for communication. Therefore, you will need to use another COM port for the terminal program.

As an alternative, you may use a crystal or ceramic resonator for clock generation for this experiment. When the demo board you are using has no connectors that allow you to plug in the resonator, it is a good idea to solder the resonator to a plug that fits on the header pins normally used to plug in the SX-Key.

1.18.1.5 Handshaking

The UART VP in this chapter does not handle any handshaking protocol. We simply assume that the receiver connected to our transmitter is fast enough to handle all characters we send at the highest possible speed. On the other hand, we “know” that we are fast enough to handle all incoming characters as we simply echo them back.

In “real live”, the situation is not always that easy. The receiving device might take longer than we expect to process the data we send, and we might perform more complicated tasks on incoming data than just sending them back.

Therefore, some kind of flow control called “handshaking” might be necessary to avoid that transmitted or received data bytes get lost.

Such flow control can be realized by “hardware” or by “software”.

Hardware handshaking requires two additional signal lines between the two devices that communicate via a serial line. Each device controls one of the two lines in order to “tell” the other device that it is ready to receive more data. On a standard serial RS-232 port, the CTS (Clear To Send) and RTS (Request To Send) lines are used for this purpose. Implementing hardware handshaking in the UART VP code is quite easy.

Software handshaking on the other hand does not require additional signal lines. Instead, special characters sent via the serial data stream are used to start and stop transmission of more data. Frequently used is the “XON/XOFF” protocol. The two communicating devices send an XOFF character when they are not able to receive more data, and an XON character when they are ready to receive new data. Implementing this protocol is a bit more complicated because it requires filtering the XON/XOFF characters. In addition, it might be necessary to buffer any characters that are sent from a device before it recognizes an XOFF character. In the application examples section of this book you can find the code for a FIFO buffer that is ideally suited for such purpose.

1.18.2 Conclusion

Let's summarize some important points about Virtual Peripherals (VPs) before ending this quite large chapter:

- Software-VPs in SX controllers replace hardware-based peripheral units found in other microcontrollers. This makes it possible to “tailor” the peripherals to individual requirements.
- VPs are usually executed within an Interrupt Service Routine (ISR) that is invoked at a timer-controlled constant repeat-rate. This rate controls the basic timing of all VPs defined in the ISR. When an ISR contains two or more VPs, the timing must be adjusted to match the requirements of the most time-critical VP.
- VPs that require an exact timing should be executed before any other VPs than have a variable execution time. If necessary, execution times of such VP-code can be adjusted by adding **nop** instructions.
- The worst-case execution time of all VPs within an ISR must not be longer than the time period between two invocations of the ISR minus seven clock cycles.
- VPs should communicate with the other parts of the application code via well-defined interfaces (e.g. using flags, and/or variables). The existence of VPs should be transparent to the other parts of the application code.
- It is extremely important that VPs do not change the contents of registers that are used by other parts of the application, like the current memory bank selection.
- If necessary, you may have VPs call subroutines but keep in mind that an interrupt-controlled VP may be invoked at any time, even when the main program has recently called some nested subroutines. Make sure that the nesting depth of eight will never be exceeded because the return stack of the SX devices is limited to a maximum of eight levels.
- It is possible (and often necessary) to store the code of VPs as subroutines in another program memory bank. Keep in mind to call such “far” routines by using the **@** modifier together with the **call** (or set the bank accordingly before), and don't forget to terminate such subroutines with a **retp** instruction.



MIDI devices usually communicate at a Baud rate of 31,250. Here are the required parameters for the UART VP to adjust it to the “MIDI rate”:

```
baud_bit      = 3
start_delay   = 13
int_period    = 200
```

Programming the SX Microcontroller

You can find more examples for Virtual Peripherals in the “Applications” section of this book, and it is a good idea to frequently visit the Internet sites of Ubicom, Parallax, and other SX-related areas when you are looking for new VPs. Keep in mind that in many cases somebody else has already “invented the wheel” just before you.

Programming the SX Microcontroller

A Complete Guide by Günther Daubach

2ND EDITION

Section II - Reference

2 Section II - Reference

2.1 Introduction

The family of the SX Controllers consists of only a few members. The major difference between the SX types is the number of available I/O pins, and the memory size. There are no SX Controllers with special integrated peripherals like UARTs, ADCs, I2C/SPI interfaces, etc., except an analog comparator. In addition, the SX48/52 devices provide two integrated multi-function timers. The lack of integrated peripherals does not mean that you can't use them - they are "constructed" in software instead.

Uvicom has developed the concept of "Virtual Peripherals(" to realize various peripherals for the SX controllers, and in the meantime, many Virtual Peripherals are available ranging from UARTs, ADCs up to the implementation of TCP/IP stacks. As peripherals in most cases are time-critical, the SX controllers contain a clock counter (the RTCC - real-time clock counter) that allows to trigger interrupts in precisely timed periods. Due to the fact that even the "slowest" SX controller can be clocked by up to 50 MHz, and that in "Turbo Mode", one instruction usually is executed in only one clock cycle, there are enough resources to even run several Virtual Peripherals within one application. For higher speed demands, there are SX controllers available that can be clocked at 75 MHz.

These are the most important features of the SX 18/20/28 devices:

- 50, or 75 MIPS speed at 50, or 75 MHz clock. Clock frequencies may range from 0 up to the maximum frequency specified for the device.
- 2048 * 12 or 4096 * 12 (SX 48/52) bits of EEPROM program memory that can be programmed through the device's clock inputs.
- 136 or 262 (SX 48/52) bytes of static RAM for variable data.
- Internal clock generator with an internal RC network that can also drive external RC networks, crystals or ceramic resonators. An external clock signal can also be applied.
- Analog comparator.
- Optional detection of supply voltage drops (brown-out)
- Integrated watchdog timer with separate internal clock generator.
- Sleep mode with minimum power consumption. Wake up can be triggered through 8 input lines, or by the watchdog timer.
- Instructions that allow reading the program memory for easy construction of tables.

Programming the SX Microcontroller

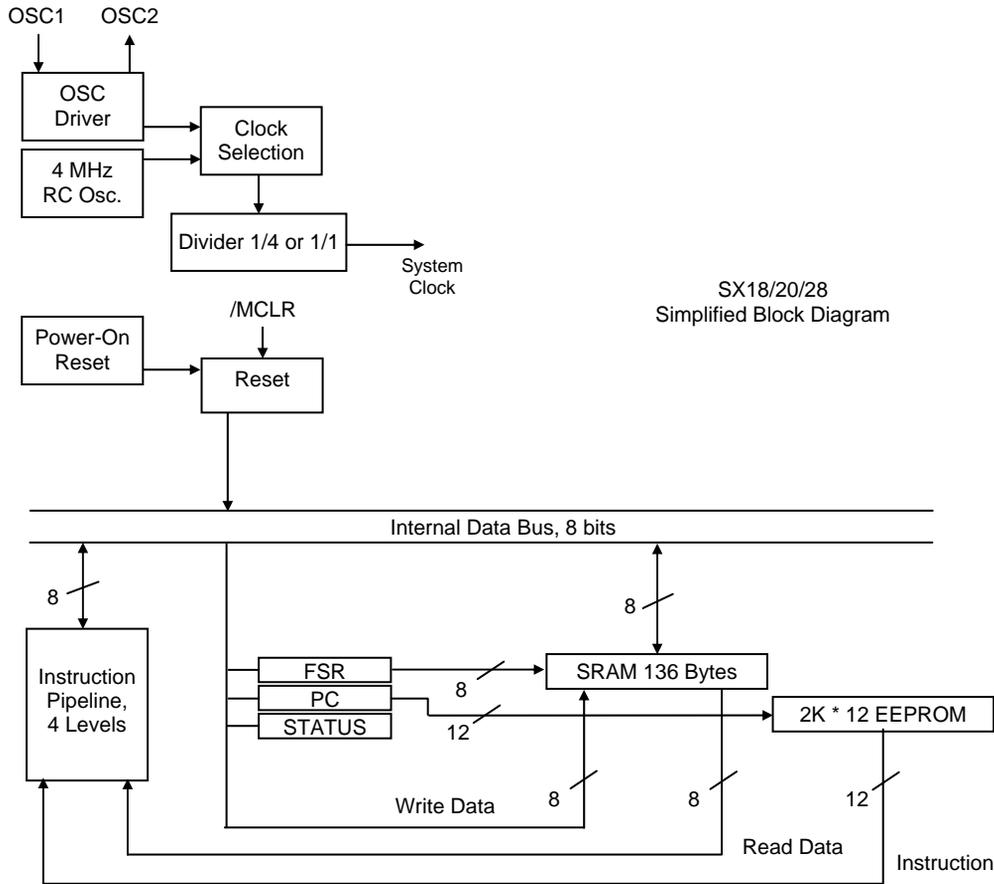
- Most of the instructions are executed in one clock cycle (20 Nanoseconds at 50 MHz clock).
- The 8-level stack memory allows nesting of subroutine calls up to that level.
- Interrupt processing (one level) with automatic save and restore of the most important registers, and a defined response time (60 ns for internal, 100 ns for external interrupt events at 50 MHz clock).
- All I/O lines can be separately programmed to function as inputs or outputs. Inputs can be programmed to have CMOS or TTL characteristics. Optional internal pull-up resistors can be activated for input lines, and most of the input lines can be programmed to act as Schmitt Trigger inputs.
- Output lines can source or sink 30 mA.
- Cost-effective development systems.

In addition, the SX 48/52 devices have the following features:

- 36 (SX 48) or 40 (SX 52) I/O lines.
- 4096 * 12 bits of EEPROM program memory.
- 262 bytes of static RAM
- Two multi-function timers that can be used as PWMs, software timers, external event counters, or in capture/compare mode.

2.2 The SX internals (simplified)

The SX controllers are complete computers on a silicon chip. This means that the internal structure is quite complex. Therefore, we will start with the most important components shown in the block diagram below. Later in this section, we will add more components as they are described.



Programming the SX Microcontroller

Two important sections in the SX are the static RAM (SRAM) with 136 bytes (SX 48/52: 262 bytes), and the EEPROM with 2048 words of 12 bits each (SX 48/52: 4096 words). The EEPROM is used to store the program instructions, and constant value tables, where the SRAM holds variable data managed by the application program. The data in the SRAM is stored as long as the device is connected to a power supply, and the data in the EEPROM remains intact even when power is disconnected.

The SX controllers use the so-called Harvard architecture, i.e. program and data memory are separate blocks, addressed via separate lines.

Addresses for the program memory are taken from the PC (Program Counter) register. The PC register is 11 bits wide (SX 48/52: 12 bits). The instruction code that is stored in program memory, currently addressed by the contents of the PC register is read and executed, and the PC register is then incremented to point to the next memory location.

To allow for program branches and subroutine calls, it is necessary to have instructions that change the contents of the PC register. Because each instruction code is always 12 bits wide, it cannot hold the complete address information for a branch or call. Instead, the PC register is loaded with a value composed of the three upper bits stored in the STATUS register, and the address bits that are part of the instruction code.

The address for the data memory is composed of the three upper bits in the FSR (File Select Register), and the address bits that are part of the instruction code.

When the supply voltage is connected to the device, or when the /MCLR line is pulled to low level, the device is reset, i.e. some internal registers are loaded with defined values, and - most important - the PC register is loaded with the address of the topmost location in program memory. The instruction in that location causes a branch to the first instruction of the application program.

To execute the internal functions in correctly timed order, a clock signal is required that provides the necessary timing for those functions. This clock signal can either be generated by an internal generator, or supplied from an external source.

The internal clock generator can generate frequencies up to 4 MHz, and external clocks may range from 0 up to 50, or 75 MHz depending on the device type. External clock signals are fed through the OSC1 pin. The internal generator can also drive external components connected to the OSC1 and OSC2 pins, like RC networks, crystals, or ceramic resonators.

The divider that can be optionally activated divides the clock signal by four to reduce the speed for the so-called "Compatibility Mode" that is used to obtain a timing similar to other microcontrollers. For new applications, you will usually like to run the SX at full speed. The SX48/52 devices do not support this mode.

Higher clock frequencies cause an increased power consumption of the SX device. Therefore, if power consumption is a critical factor, the clock frequency should not be higher than necessary for the required execution speed. Lower clock frequencies also reduce RFI problems that might occur.

Each instruction that is addressed by the PC register is executed in four steps:

- Reading the instruction from program memory
- Decoding the instruction
- Executing the instruction
- Saving the result

Many microcontrollers perform these steps during four subsequent clock cycles, which means that the execution of one instruction usually takes four clock cycles.

The SX controllers also require such four clock cycles for each instruction, but by using a four-level instruction pipeline, the effective execution speed becomes one clock cycle per instruction. After a reset, the first instruction is stored in the first stage of the pipeline, but while this instruction is decoded, the next instruction is already read from program memory, and saved into the next stage of the pipeline.

During the third clock cycle, the first instruction is executed, the second one is decoded, and a third one will be loaded.

Finally, at the fourth clock cycle, the result of the first instruction is stored, the second instruction is executed, the third one is decoded, and a fourth instruction is loaded into the last stage of the pipeline.

From now on, the pipeline is "full", and at each clock cycle, the result of one instruction will be stored, resulting in an effective speed of one instruction per clock cycle. This means that an SX with 50 MHz reaches a throughput of 50 MIPs (50 Mega Instructions Per second) - this is an extraordinary value for a microcontroller!

As long as the program code is "straight", i.e. the PC register is incremented after each instruction read, instructions keep "flowing" through the pipeline, as described above.

When a program branch, or a subroutine call is performed, or an interrupt is triggered, the PC register is loaded with a new value, and the pre-loaded instructions in the pipeline must be discarded in order to fill the pipeline with instructions starting at the new location in program memory. This means that branches, and subroutine calls require more clock cycles than "straight" instructions (3 cycles in worst-case).

Programming the SX Microcontroller

2.2.1 How SX Instructions are Constructed

The program memory is organized in a way that the PC register always addresses a word, 12 bits wide, and all basic SX instructions are defined as one-word instructions, i.e. there are no instructions that "share" two or more words in program memory.

Let's take an **inc** instruction as an example:

inc fr

This instruction increments the contents of a byte in data memory, where **fr** (file register) stands for its address. The instruction code for **inc fr** is

0010 101f ffff

The first 7 bits will be decoded by the SX as the increment instruction, and the remaining 5 bits specify the address of the location in data memory whose value shall be incremented. This means that the five address bits allow to only specify 32 different locations.

As the SX data memory is larger than 32 bytes, these five address bits do not allow to specify the full address for all locations in data memory. The full address is composed by these five bits (as low-order bits), and the three upper bits in the FSR (File Select Register). You can find a more detailed description in the next chapter.

Another example is the jump instruction:

jmp addr

which ends the "straight" program execution and branches to another location in program memory specified by the **addr** argument. The instruction code for **jmp** is:

101a aaaa aaaa

The first three bits are decoded by the SX as a jump instruction, and the remaining nine bits specify the target.

Nine bits allow to represent values from 0 (0 0000 0000) through 511 (1 1111 1111).

Another instruction that leaves "straight" program execution is the subroutine call:

call addr

which is coded as

1001 aaaa aaaa

In this case, the address part of the instruction code is only eight bits wide, i.e. it can represent values from 0 (0000 0000) through 255 (1111 1111).

In both cases, the available range of address values is not large enough to directly address all locations in program memory. To build a complete address, here the higher three bits stored in

the STATUS register are combined with the address bits read from the instruction code. In case of a **call** instruction, the "missing" 9th bit is always set to zero. You can find a more detailed description in the next chapter.

2.2.2 Organization of the Data Memory and how to Address it

2.2.2.1 SX 18/20/28

Address	Bank 0	Bank 1	Bank 2	Bank 3	Bank 4	Bank 5	Bank 6	Bank 7
\$00	INDF							
\$01	RTCC							
\$02	PC							
\$03	STATUS							
\$04	FSR							
\$05	Port A							
\$06	Port B							
\$07	Port C							
\$08	Reg \$08	Reg \$08	Reg \$08	Reg \$08	Reg \$08	Reg \$08	Reg \$08	Reg \$08
\$09	Reg \$09	Reg \$09	Reg \$09	Reg \$09	Reg \$09	Reg \$09	Reg \$09	Reg \$09
\$0a	Reg \$0a	Reg \$0a	Reg \$0a	Reg \$0a	Reg \$0a	Reg \$0a	Reg \$0a	Reg \$0a
\$0b	Reg \$0b	Reg \$0b	Reg \$0b	Reg \$0b	Reg \$0b	Reg \$0b	Reg \$0b	Reg \$0b
\$0c	Reg \$0c	Reg \$0c	Reg \$0c	Reg \$0c	Reg \$0c	Reg \$0c	Reg \$0c	Reg \$0c
\$0d	Reg \$0d	Reg \$0d	Reg \$0d	Reg \$0d	Reg \$0d	Reg \$0d	Reg \$0d	Reg \$0d
\$0e	Reg \$0e	Reg \$0e	Reg \$0e	Reg \$0e	Reg \$0e	Reg \$0e	Reg \$0e	Reg \$0e
\$0f	Reg \$0f	Reg \$0f	Reg \$0f	Reg \$0f	Reg \$0f	Reg \$0f	Reg \$0f	Reg \$0f
\$10	16 General Purpose Registers (\$10...\$1f)	16 General Purpose Registers (\$30...\$3f)	16 General Purpose Registers (\$50...\$5f)	16 General Purpose Registers (\$70...\$7f)	16 General Purpose Registers (\$90...\$9f)	16 General Purpose Registers (\$b0...\$bf)	16 General Purpose Registers (\$d0...\$df)	16 General Purpose Registers (\$f0...\$ff)
\$11								
\$12								
\$13								
\$14								
\$15								
\$16								
\$17								
\$18								
\$19								
\$1a								
\$1b								
\$1c								
\$1d								
\$1e								
\$1f								

In this table, all addresses are specified in hexadecimal notation with a leading \$ sign.

Programming the SX Microcontroller

The memory locations in the first 8 rows in column "Bank 0" contain the internal SX registers. The meaning of these registers will be explained in further sections of this book. Here, you should keep in mind that such memory locations may not be used to store arbitrary data. It is also important to note that the first 8 rows in column "Bank 1" through "Bank 7" do not represent different physical memory locations. Instead, these addresses are always "mapped" to Bank 0.

Memory locations in the next 8 rows in column Bank 0 contain general purpose registers \$08...\$0f that may be used to store any kind of data. Again, these rows in the other columns do not represent physical memory locations, they are "mapped" to Bank 0.

As mentioned before, SX instruction codes provide only 5 bits to specify a file register address, i.e. they can only represent 32 unique values.

As the block diagram shows, the FSR is used to hold the address of a data location. The lower five bits are used for indirect addressing, where the upper three bits are used to select a memory bank. You may use a **mov** instruction, **setb** or **clr b** instructions, or the special **bank** instruction to modify these three bits in the FSR. The **bank** instruction takes the upper three bits of its argument, and copies them into the upper three bits of the FSR without modifying the lower five bits.

FSR							
7	6	5	4	3	2	1	0
Bank Select			Indirect Register Address				

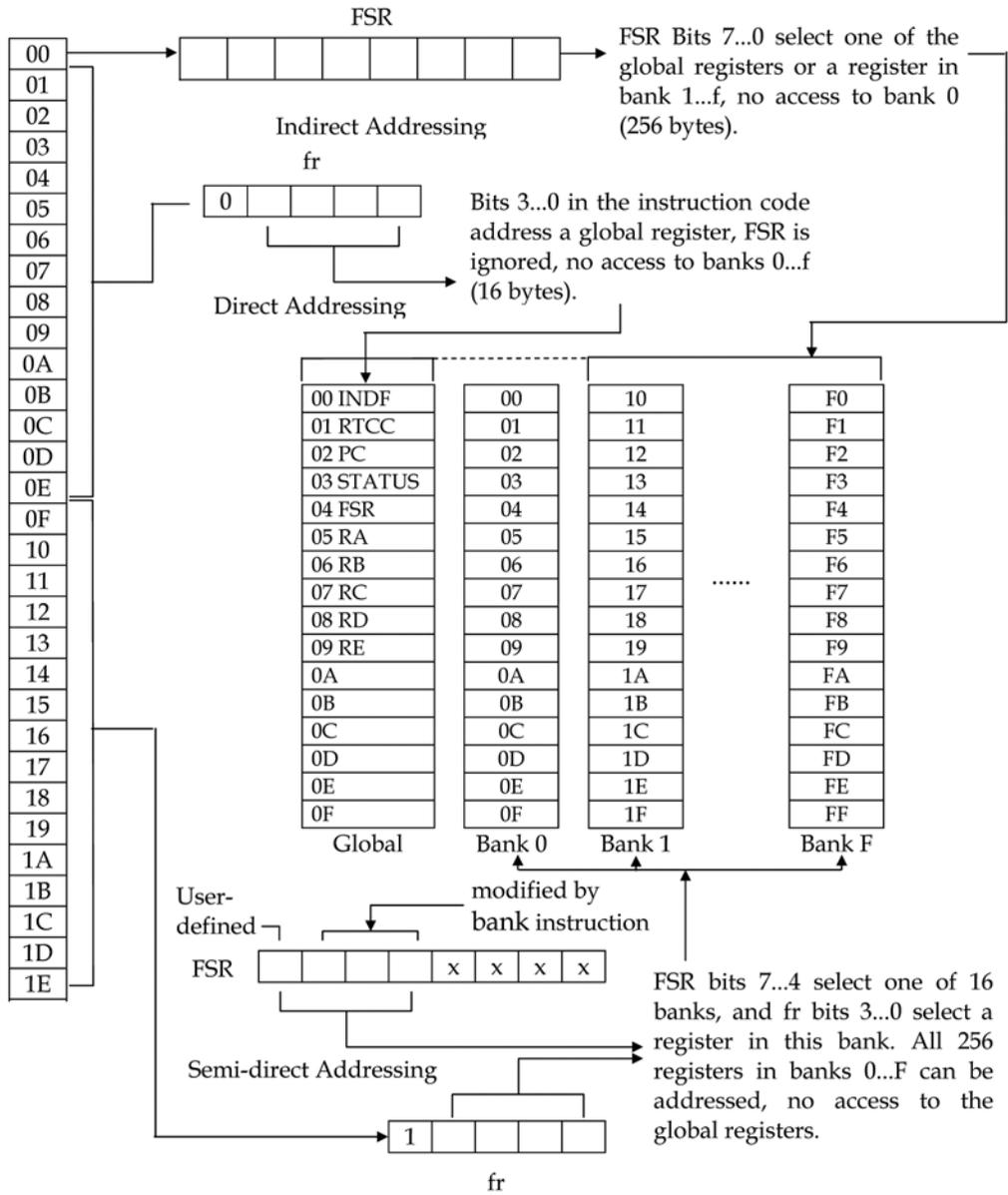
The SX 18/20/28 controllers come with 136 bytes of data memory that can be addressed with an 8-bit value. There are two addressing modes, direct, and indirect addressing.

For direct addressing, the 8-bit address is composed of the upper three FSR bits (the "bank selection bits"), and the five address bits contained in the instruction code as described above.

When the five address bits in the instruction code are all zeroes, the virtual INDF register at location \$00 is addressed, and this causes the SX to take the contents of the eight bits in the FSR as the address for the data location to be accessed. This is called indirect addressing. Before using this addressing mode, it is necessary to load the FSR with the required address value. By incrementing or decrementing the FSR it is possible to access subsequent memory locations from within a program loop in order to manipulate a table or an array of registers.

As long as the address part in an instruction ranges from \$00 through \$0f, the upper three bits in the FSR are ignored, and are cleared in the full address. This means that the registers physically located in Bank 0 from \$00...\$0f will always be accessed, no matter what bank is currently selected. Therefore, these eight memory locations are handy to store "global data" that shall be accessed from various parts of the application program without the need to select a specific bank before.

2.2.2.2 SX 48/52



Programming the SX Microcontroller

The SX 48/52 devices come with a data memory of 262 bytes, i.e. using an 8-bit address is not sufficient to address all memory locations. Therefore, the addressing modes have been enhanced.

Direct addressing is used to access the global registers. In order to access the global registers, the address argument **fr** of an instruction must have bit 4 (the highest bit) cleared, i.e. **fr** must have a value from \$00 through \$0f. This is similar to addressing a global register with the SX 18/20/28 devices. Other memory banks cannot be accessed using direct addressing.

For semi-direct addressing, the address argument **fr** of an instruction must have bit 4 (the highest bit) set, i.e. **fr** must have a value from \$10 through \$1f. Again, this is similar to addressing a register in a memory bank with the SX 18/20/28 devices. The complete address is composed by the upper four FSR bits, and the lower four bits of the **fr** argument. Register banks 0 through F can be accessed, but not the global registers, i.e. 256 bytes. The upper four FSR bits select the bank, and the lower four **fr** bits select a register in that bank.

Different from the SX 18/20/28 devices, the **bank** instruction in SX 48/52 devices copies bits 6...4 of the argument into FSR bits 6...4. FSR bit 7 must be modified by a separate instruction, e.g. **setb** or **clrb**.

As an alternative, the **_bank** macro can help making bank switching easier, which should be called instead of a **bank** instruction:

```
_bank macro 1
bank \1                ; For SX18/20/28 change
                        ; FSR bits 7...5,
                        ; for SX48/52 change
                        ; FSR bits 6...4
IFDEF SX48_52         ; For SX48/52 change
IF \1 & %10000000    ; FSR bit 7
    setb fsr.7
ELSE
    clrb fsr.7
ENDIF
ENDIF
endm
```

Different from the diagram shown above, banks 0...F are divided into two sections of eight registers each, and FSR bit 7 selects the upper or lower section, and FSR bits 6...4 select the bank. In an application program, the physical location of the registers does not matter as long as each of them has a unique address, so you don't need to care about that detail.

For indirect addressing, FSR is loaded with the address of the memory location that shall be accessed, and access is performed through the virtual register \$00 (INDF). Again, 256 different memory locations can be accessed. Different from semi-direct addressing, the global registers, and not the registers in bank 0 will be accessed when FSR contains values from \$00 through \$0f. FSR contents from \$10 through \$ff access registers in banks 1...F.

2.2.3 Organization of Program Memory and how to Access it

The program memory in SX 18/20/28 devices comes with 2048 words of 12 bits, and the memory locations have addresses from \$000 through \$7ff. These values can be represented by an 11-bit value. Actually, the PC register is 12 bits wide, but the highest bit is ignored in that devices.

The SX 48/52 devices come with 4096 words of program memory with addresses from \$000 through \$FFF, and all 12 bits in the PC are used.

As mentioned before, a **jmp** instruction has only 9 bits available for the address information, so that two or three additional bits are required to compose a full address. These bits are stored in the upper three bits of the STATUS register, and it is necessary to make sure that those bits contain the required value before executing a **jmp**. (The meaning of the other STATUS register bits will be explained later).

For **call** instructions, the upper two or three bits are also taken from the upper STATUS register bits, the 9th bit will always be cleared, and the lower 8 bits are taken from the address argument of the **call** instruction. This means that subroutines can only begin within the lower half of a memory page.

According to this information, the program memory is organized as shown in this diagram:

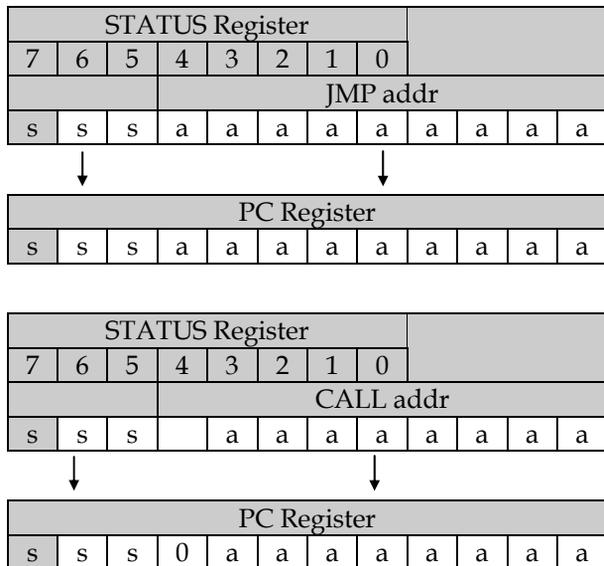
Page 0	Page 1	Page 2	Page 3	Page 4	Page 5	Page 6	Page 7
\$000	\$200	\$400	\$600	\$800	\$a00	\$c00	\$e00
\$0ff	\$2ff	\$4ff	\$6ff	\$8ff	\$aff	\$cff	\$eff
\$100	\$300	\$500	\$700	\$900	\$b00	\$d00	\$f00
\$1ff	\$3ff	\$5ff	\$7ff	\$9ff	\$bff	\$dff	\$fff

Pages 0 through 3 are available in all SX controllers, where pages 4 through 7 are only available in SX 48/52 devices.

In the lower half of each page (marked white or light gray), subroutines may begin, where the upper half of each page cannot be reached by **call** instructions.

Programming the SX Microcontroller

The following figures show how addresses are composed for **jmp** and **call** instructions (the highest bit is only used in SX 48/52 devices):



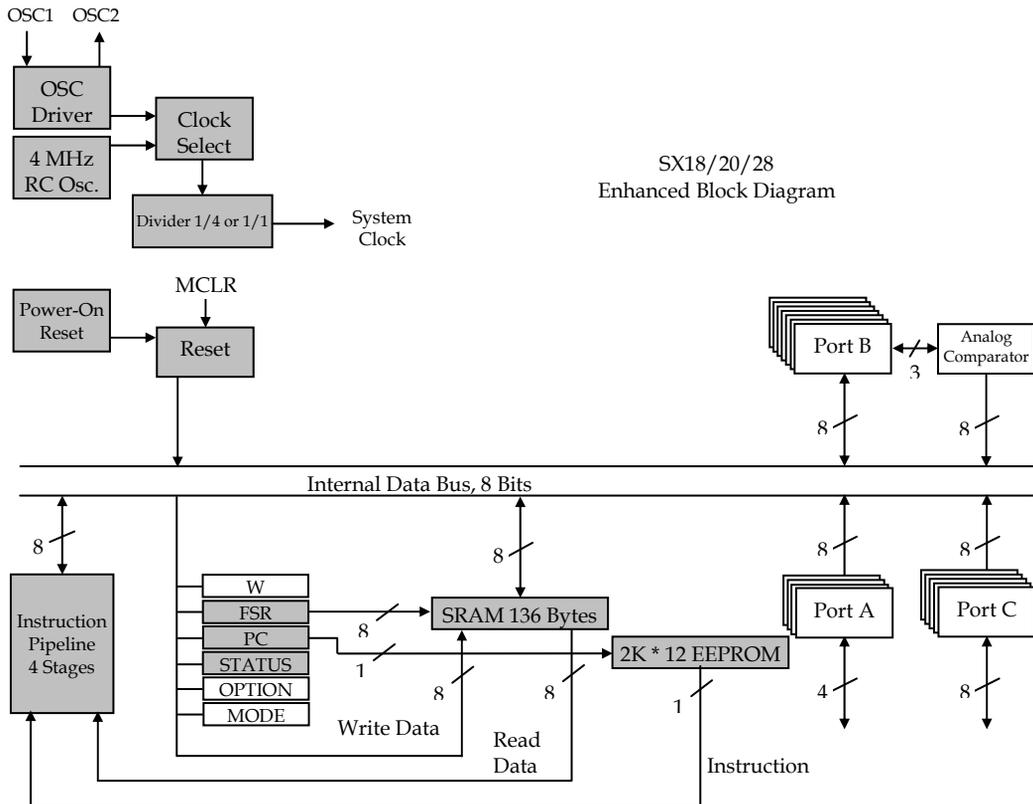
If a **jmp** or **call** shall be executed whose target lies in a memory page other than the one currently selected one, bits 7...5 or 6...5 in the STATUS register must first be set to the new page. Instead of using a **mov** instruction, or **setb** and **clrb** instructions to modify the upper bits in the STATUS register, you should use the **page** instruction instead, that only modifies bits 6 and 5 (SX 18/20/28) or bits 7...5 (SX 48/52) in the STATUS register.

As long as **jmp** or **call** instructions target locations within the currently selected page, there is no need to execute another **page** instruction unless the STATUS register bits (7,) 6, or 5 have been changed by some reason.

As Bit 7 in the STATUS register is not needed to build addresses in the SX 18/20/28 devices, this bit may be used as general-purpose flag if it is not intended that an application shall be ported to an SX 48/52 device later.

2.2.4 The SX Special Registers and the I/O Ports

The next block diagram shows some new components that will be described in this chapter. The components already explained are marked gray.



2.2.4.1 The W Register

The W register (Working Register) is a multi-purpose register used as temporary storage for data or results, or it is used to hold one operand for arithmetic or logical operations. The W register is somehow similar to the accumulator known in other systems.

Programming the SX Microcontroller

2.2.4.2 The I/O Registers (Ports)

The SX 18/20 controllers come with 12 I/O pins, the SX 28 controller comes with 20 I/O pins, and the SX 48/52 devices offer 36 or 40 I/O lines.

Eight I/O lines are grouped together to make up a port that can be accessed similar to the other data registers. The port registers are mapped into memory bank 0 at address locations \$05 (Port A, or **ra**), \$06 (Port B, or **rb**), and \$07 (Port C, or **rc**). In the SX 18, port C is not connected to output pins, and it cannot be used as an I/O port.

The SX 48/52 devices have two additional ports, Port D (**rd**) at address \$08, and Port E (**re**) at address \$09.

With the exception of the SX52, the upper four lines of Port A are not connected to output pins.

Because the port registers are mapped into the global register bank they can always be accessed, no matter what bank is currently selected (with the exception that the semi-direct addressing mode of the SX 48/52 devices does not allow access to the global registers).

Port A has symmetrical output drivers that means the voltage drop across a load connected to one of the Port A outputs is equal, no matter if the output sources a current against V_{SS} , or if it sinks a current from V_{DD} .

2.2.4.3 Read-Modify-Write Instructions

Read-modify-write instructions are instructions that read a register, modify its value, and write back the result to the same register. Examples for such instructions are **clrb fr. bit** (clear a bit in **fr**), **setb fr. bit** (set a bit in **fr**), **inc fr** (increment **fr**), **dec fr** (decrement **fr**).

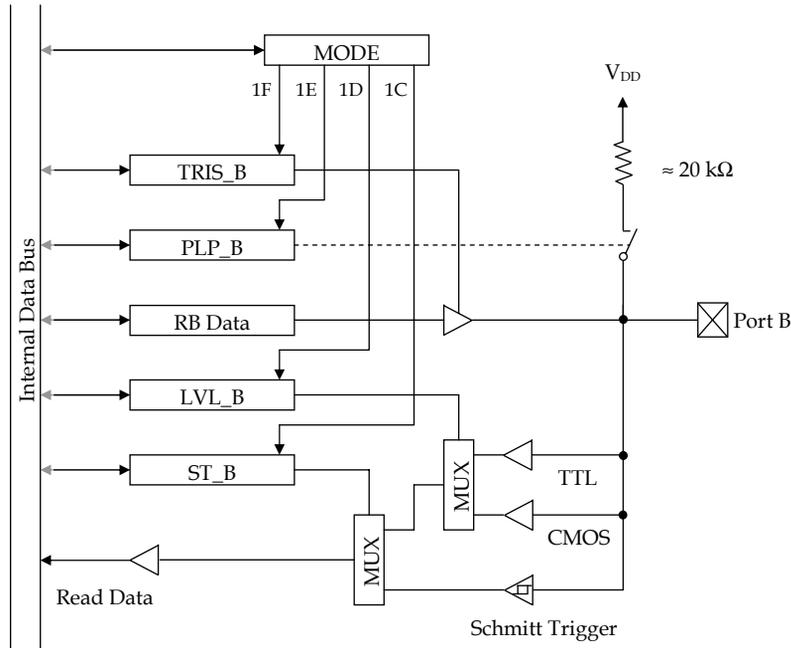
It is obvious that the **inc** and **dec** instructions must read the register value, increment or decrement it, and then write it back to the register, using the ALU to perform that operation. Instructions that manipulate the state of a single bit (like **setb** and **clrb**, but also **stc**, **clc**, **stz**, and **clz**) cannot directly access the bit in the register. Again, the register value is read, the bit is set or cleared using the ALU, and the result is written back to the register.

When you manipulate port data registers, you must be careful when two or more successive read-modify-write instructions are executed. As "reading" a port means by default, that the levels at the port pins are read, there might be situations that the port level does not yet reflect the last setting when the next read is executed. This is especially true at high clock rates, like 50 MHz. Remember that the SX uses an instruction pipeline, i.e. while the result of an instruction is written, the next instruction is already executed.

In order to obtain correct results, either do not use successive read-modify-write instructions accessing the same port, or insert **nop** instructions in between in order to generate a short delay.

2.2.4.4 Port Block Diagram

The block diagram below shows the function blocks that make up one Port B pin. The same diagram applies to the other ports with the exception that Port A cannot be configured for Schmitt Trigger levels, i.e. the ST register and the associated logic is missing.



The control registers TRIS through ST are write only in SX 18/20/28 devices, and read/write in SX 48/52 devices. These registers are all mapped into the data address space from \$05 (Port A) through \$09 (Port E) (ports D and E: SX 48/52 only).

The content of the MODE register selects which control register is currently mapped into the data address space. To write to the selected control register, use the **mov !r?, w** instruction.

When a bit in the TRIS register is set, the output buffer following the data register (also called "data latch") is set to high-impedance, i.e. the port pin is configured as an input. Clear that bit to enable the buffer in order to configure the pin as an output. Note that this does not disable the input circuitry, i.e. you can still read the port pin. By default, the actual level at the output pin is read. The SX 48/52 devices can be configured that the content of the data register is read instead.

Programming the SX Microcontroller

When a bit in the PLP register is cleared, the internal pull-up resistor is connected between the port pin and V_{DD} .

To set an output pin to a certain level (provided that the output driver is enabled), the associated bit in the data register must be set (high level) or cleared (low level).

The bits in the LVL register control the multiplexer, which selects one of the two input buffers. When a bit is set, the TTL buffer is enabled. A cleared bit selects the CMOS buffer.

When a bit in the ST register is cleared, the second multiplexer selects the Schmitt Trigger buffer for input. In this case, the bit in the LVL register is ignored.

After a reset, all bits in the control registers are set, i.e. all pins are configured as inputs with TTL level, no pull-up resistors, and the Schmitt Trigger buffers disabled, and the MODE register is set to \$1F, i.e. the TRIS registers are selected.

2.2.4.5 The Data Direction Registers

The block diagram shows more registers stacked behind the port registers. These registers are required to configure the parameters of the I/O ports, as described before. The most important configuration register is the data direction register. Depending on the bits in the data direction register, the associated port pins (RA7...RA0) are configured either as an inputs, or as outputs.

TRIS_A							
7	6	5	4	3	2	1	0
RA7	RA6	RA5	RA4	RA3	RA2	RA1	RA0

The figure above shows the Port A data direction register. If a bit in this register is set, the associated port line is configured as an input. If the bit is cleared, the line becomes an output. Bits 7...4 are available in SX 52 devices only.

The configuration of the other ports is similar to the Port A configuration:

TRIS_B							
7	6	5	4	3	2	1	0
RB7	RB6	RB5	RB4	RB3	RB2	RB1	RB0

TRIS_C							
7	6	5	4	3	2	1	0
RC7	RC6	RC5	RC4	RC3	RC2	RC1	RC0

The same scheme is also true for Ports D and E in the SX 48/52 devices.

The abbreviation TRIS is derived from "Three-State". If you "look" into an I/O line from the outside, it has either high (V_{DD}) or low (V_{SS}) level if configured as an output, or it has high impedance (hi-Z) if configured as an input, and this is identical to the Three-State characteristics of other components. In applications, there are situations that SX I/O lines must act as Three-State outputs, and this can easily be accomplished by re-configuring port lines at run-time.

After a reset, all bits in the TRIS registers are set, i.e. all I/O lines are configured as inputs to hi-Z state. This guarantees a safe starting condition and avoids that lines wired to external signal outputs cause short circuits which could happen if they were randomly configured as outputs after reset.

Lines that are configured as outputs can sink or source a maximum current of 30 mA.

2.2.4.6 The Level Register

Associated to each I/O port is another configuration register that allows to configure the threshold of the high/low input levels to CMOS or TTL mode. These registers are called LVL_A, LVL_B, LVL_C, LVL_D, and LVL_E.

To configure an input for TTL level, the associated bit in the level register must be set, and to configure for CMOS level, the bit must be cleared.

When configured for TTL level, an input interprets voltages between 0 and 1.4 V as low level (0), and voltages above 1.4 V as high level (1). When configured for CMOS level, an input interprets voltages between 0 and $V_{DD}/2$ as low level (0), and voltages above $V_{DD}/2$ as high level.

As long as an I/O line is configured as output, the level setting is ignored. After a reset, all LVL register bits are set, i.e. all inputs are configured to TTL level.

2.2.4.7 Pull-up Enable Registers

Another configuration register available for all ports is the pull-up enable register that allows to connect an internal resistor between an input line, and V_{DD} . For example, pull-up resistors are required if a switch is connected between an input and V_{SS} in order to pull the input line to a defined level (V_{DD}) in case the switch is open. Without the resistor, an open input would "float", i.e. having an undefined status.

The registers are called PLP_A, PLP_B, PLP_C, PLP_D, and PLP_E. When a bit in a PLP register is cleared, the pull-up resistor for the associated input is activated. If the bit is set, the resistor is de-activated. After a reset, the bits in all PLP registers are set, i.e. the pull-up resistors are all de-activated. When a port line is configured as output, the setting of the pull-up enable bit is ignored, and the pull-up resistor is not active.

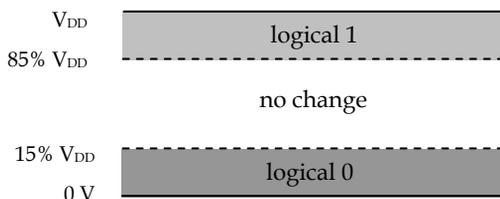
Programming the SX Microcontroller

The internal pull-up resistors have values of about 20 k Ω . When using longer external signal lines it may be necessary to use external pull-up resistors to increase the noise immunity.

2.2.4.8 The Schmitt Trigger Enable Registers

In addition to the option of setting an input to CMOS or TTL level, all input lines (except the Port A lines) can be configured to act as Schmitt Trigger inputs. The Schmitt Trigger enable registers are called ST_B, ST_C, ST_D, and ST_E (ST_D, and ST_E available in SX 48/52 devices only).

When an input has Schmitt Trigger characteristics, the levels shown in this diagram are valid:



Starting at an input level of less than 15% of V_{DD} , the input is considered to have a logical level of 0. When the voltage at the input is increased, the logical level remains at 0 until the voltage exceeds 85% of V_{DD} . The logical level then changes to 1. When the input voltage is then reduced from this point on, the logical level remains at 1 as long as the voltage is above 15% of V_{DD} . As soon as the input voltage goes below 15% of V_{DD} , the logical level turns back to 0.

When a bit is cleared in an ST register, the associated input has Schmitt Trigger characteristics, and the CMOS/TTL setting is ignored. If the bit is set, the setting of the LVL bit controls the behavior of the input.

After a reset, all bits in the ST registers are set, i.e. no Schmitt Trigger characteristics are active on any input.

Configuring an input as Schmitt Trigger, helps to reduce errors caused by external noise, and by slowly changing input levels.

2.2.4.9 The Port B Wake Up Configuration Registers

The SX controllers support the so-called "Sleep Mode". After a sleep instruction, program execution stops, and the device's power consumption is reduced. One method to "wake up" a "sleeping" SX device is to generate a signal transition on a port B input line. The bits in the WKEN_B (Wake-up Enable B) register control which Port B input lines may wake up the SX. Inputs configured as wake up inputs can also be used to trigger an interrupt at regular program execution.

If a bit in the WKEN_B register is cleared, the associated input line is activated for wake up and interrupt trigger. If the bit is set, this feature is disabled for this line.

After reset, all bits in the WKEN_B register are set, i.e. no line will cause a wake up or an interrupt event.

Another register, called WKED_B (Wake-up Edge B) is used to configure which transition on an input line shall cause a wake up or an interrupt. If a bit in the WKED_B register is set, the associated input will cause a wake up or an interrupt on a negative, i.e. a high-to-low transition. If the bit is cleared, a positive, i.e. a low-to-high transition will cause a wake up or an interrupt.

After reset, all bits in the WKED_B register are set, i.e. all Port B inputs are configured for negative transitions.

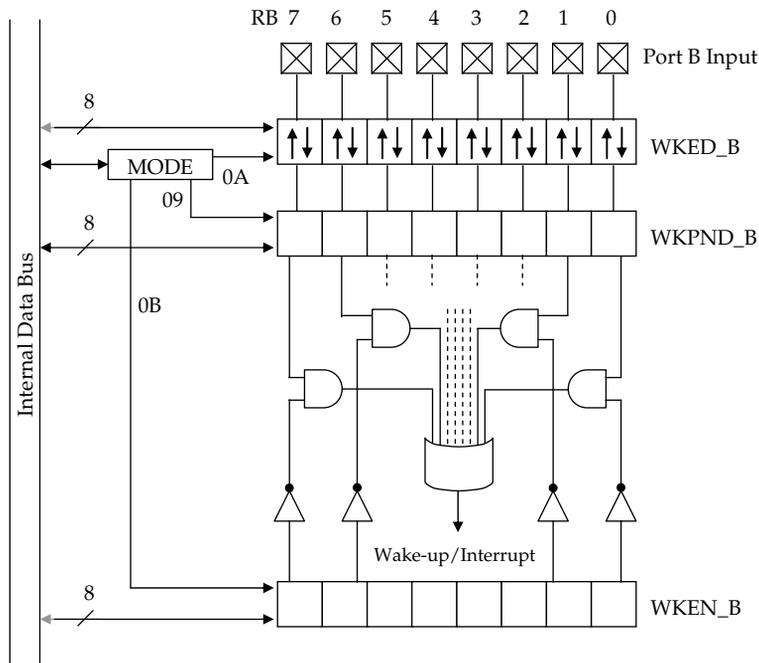
Because up to eight Port B inputs can be configured to issue a wake up or an interrupt, it is important that the application program is able to determine which input caused an event. Therefore, another register, the WKPND_B (Wake-up Pending B) register is available. When a set bit is set in this register, this indicates that an event occurred on the associated input line since the bit was cleared last.

The bits in the WKPND_B register are always set when a signal transition (as defined in the WKED_B register) has occurred on the associated input line, no matter if the associated bit in the WKEN_B register is cleared or set.

As the bits in the WKPND_B register are not automatically cleared when an event occurred, it is important that the application program clears the bits in order not to miss any new events, or to avoid that a set bit immediately triggers another interrupt.

Programming the SX Microcontroller

The blocks that make up the multi-input wake-up/interrupt circuit are shown in the block diagram below:



The MODE register is used to select one of the three configuration registers. Depending on its value, WKED_B (\$0A), WKPND_B (\$09), or WKEN_B (\$0B) is selected.

The bits in the WKEN_B register control if a port bit shall cause a wake-up/interrupt or not. In case a bit is cleared, its inverted state sets the associated AND gate's input to 1, i.e. the wake-up/interrupt feature for this input is enabled.

The bits in the WKED_B register control if a negative or a positive signal edge shall set the associated bit in the WKPND_B register. When a bit is set, the negative signal edge is selected, otherwise the positive edge.

When a port input signal edge occurs that matches the setting in the WKED_B register, the associated bit in the WKPND_B register is set, and the other input of the associated AND gate is set. Provided that the bit in WKEN_B is also set, the AND gate's output goes to 1, and the output of the OR gate goes to 1 too, and a wake-up or interrupt event is triggered.

Note that the bits in the WKPND_B register remain set until they are cleared by a `clr w - mov !rb, w` instruction sequence. "Writing" `w` to WKPND_B actually means "exchanging" the contents

of WKPND_B and w, i.e. after execution of this instruction, WKPND_B is cleared, and w contains the former value of WKPND_B. This allows the application program to test which port input line has caused an interrupt or a wake-up, and to clear the flag at the same time.

The SX 48/52 devices also allow for reading the WKED_B and WKEN_B registers.

2.2.4.10 The Port B Analog Comparator

The Port B lines RB0, RB1, and RB2 can be configured to connect the internal analog comparator to the "outside world". For this purpose, there is another configuration register available for Port B, called CMP_B (Comparator B). The bits in this register have the following meaning:

CMP_B							
7	6	5	4	3	2	1	0
EN	OE	-	-	-	-	-	Res

If bit 7, the EN (Enable) bit is cleared, the comparator is activated, i.e. port lines RB1, and RB2 act as comparator input lines. If bit 5, the OE (Output Enable) bit is cleared, the comparator output signal is connected to the RB0 line, making it possible that external components can be directly controlled by the comparator output signal.

Bit 0 the Res (Result) bit indicates the status of the comparator output. This bit must be tested by the application program to determine the current comparator output status. The bit is set, when the voltage between RB2 and V_{SS} is greater than the voltage between RB1 and V_{SS} . When the comparator output on the RB0 line is active (bit OE in CMP_B cleared), a program can also test the status of bit rb.0 to determine the current comparator output.

If the SX device enters the sleep mode with the comparator activated, the comparator remains active, i.e. if its output is enabled, this output changes according to the voltage difference between the two inputs. If this feature is not needed, it is recommended to de-activate the comparator before entering the sleep mode in order to reduce power consumption.

2.2.4.11 More Configuration Registers (SX 48/52)

The SX 48/52 devices come with two internal Multi-Function Timers that require additional configuration registers associated to Ports B and C. A description of these registers can be found in the chapter 2.2.8.

Programming the SX Microcontroller

2.2.4.12 Addressing the I/O Configuration Registers (SX 18/20/28)

The configuration registers described before are not mapped into the address space of the data memory as the port registers are.

To write a constant value into a port register, you usually would use the assembly instructions

```
mov w, #Const  
mov r?, w
```

First, the `w` register receives a value between 0 and 255, and then this value is copied into the specified port register `ra`, `rb`, or `rc`.

When you replace the `r?` part of the second `mov` instruction by `!r?`, like in

```
mov !ra, w
```

this indicates that the contents of `w` shall not be copied into a port register but into one of the port configuration registers, but this does not specify which configuration register shall be the target.

The contents of the MODE (or `m`) register determines which type of configuration register is currently available for the `mov !r?` instruction. In order to set the `m` register to a specific value, you might code

```
mov m, #RegSelect
```

or, as an alternative, you can use the `mode` instruction, like in

```
mode RegSelect
```

Please note that both instructions only copy the lower four bits of the instruction argument into the `m` register. As the SX 18/20/28 devices ignore the higher bits, this is fine with these types, but not for SX 48/52 devices that allow for greater values in the MODE register.

The following diagram shows the contents of the `m` register required to access a specific port configuration register (SX 18/20/28):

MODE (m)	Port RA	Port RB	Port RC
\$xf	TRIS_A	TRIS_B	TRIS_C
\$xe	PLP_A	PLP_B	PLP_C
\$xd	LVL_A	LVL_B	LVL_C
\$xc	-	ST_B	ST_C
\$xb	-	WKPEN_B	-
\$xa	-	WKPED_B	-
\$x9	-	WKPND_B → W	-
\$x8	-	COMP_B → W	-
\$x7...\$x0	-	-	-

The "x" in the m column hex numbers means that the upper four bits are ignored and may have any value. In the following text, we assume that the bits are all cleared.

After a reset, the MODE register is initialized to \$0f, i.e. the port direction registers can be accessed by default without the need to write to the m register before.

Here is a coding example:

```

mode #$0f ; access the TRIS registers
mov w, #%11110000
mov !rc, w ; configure RC3...0 as outputs

mode #0e ; access the PLP registers
mov w, #%11111100
mov !rc, w ; activate pull-ups for RC1 and RC0

mode #0c ; access the ST registers
mov w, #%00111111
mov !rb, w ; Schmitt Triggers on RB7 and RB6

```

A special method is used to access the COMP_B and WKPND_B registers for reading (m = \$08 or \$09). Instructions like

```
mov !rb, w
```

allow to copy the contents in w to a configuration register, but an instruction like

```
mov w, !rb
```

to copy the contents of a configuration register to the w register is not available.

Nevertheless, there must be a way to "read" the contents of the COMP_B and WKPND_B registers in order to obtain the comparator result, and to find out which Port B input line caused a wake up or an interrupt.

When the m register contains \$08 or \$09, and a

```
mov !rb, w
```

instruction is executed, the content of w is copied into the COMP_B or WKPND_B register, and the former content of the register is copied to w. All this is done within effectively one instruction cycle. This means that after execution of the **mov !rb, w** instruction, w contains the contents of the control register as it was before the **mov**. In other words, the contents of w and the control register are swapped. This is especially useful in case of the WKPND_B register to read the wake-up pending bits, and to clear them "on the fly" by setting W to 0 before the **mov**.

Here is an example:

```

mov m, $09 ; Address WKPND_B.
mov w, #0
mov !rb, w ; Clear bits WKPND_B and copy the
               ; status of the wake-up bits to w to

```

Programming the SX Microcontroller

```
; test for the wake-up or interrupt
; reason.
```

2.2.4.13 Addressing the SX 48/52 I/O Configuration Registers

Similar to the SX 18/20/28 devices, instructions like `mov !ra, w`, `mov !rb, w`, etc. are used to access the control registers with the MODE register previously set to a value to select the correct register type. The possible values for the MODE register are shown in the table below:

SX48/52 MODE (m) Register and mov !r?, w					
m	mov !ra, w	mov !rb, w	mov !rc, w	mov !rd, w	mov !re, w
\$00		read T1CPL	read T2PL		
\$01		read T1CPH	read T2CPH		
\$02		read T1R2CML	read T2R2CML		
\$03		read T1R2CMH	read T2R2CMH		
\$04		read T1R1CML	read T2R1CML		
\$05		read T1R1CMH	read T2R1CMH		
\$06		read T1CNTB	read T2CNTB		
\$07		read T1CNTA	read T2CNTA		
\$08		exchange CMP_B			
\$09		exchange WKPND_B			
\$0a		write WKED_B			
\$0b		write WKEN_B			
\$0c		read ST_B	read ST_C	read ST_D	read ST_E
\$0d	read LVL_A	read LVL_B	read LVL_C	read LVL_D	read LVL_E
\$0e	read PLP_A	read PLP_B	read PLP_C	read PLP_D	read PLP_E
\$0f	read TRIS_A	read TRIS_B	read TRIS_C	read TRIS_D	read TRIS_E
\$10		clear Timer 1	clear Timer 1		
\$11					
\$12		write T1R2CML	write T2R2CML		
\$13		write T1R2CMH	write T2R2CMH		
\$14		write T1R1CML	write T2R1CML		
\$15		write T1R1CMH	write T2R1CMH		
\$16		write T1CNTB	write T2CNTB		
\$17		write T1CNTA	write T2CNTA		
\$18		exchange CMP_B			
\$19		exchange WKPND_B			
\$1a		write WKED_B			
\$1b		write WKEN_B			
\$1c		write ST_B	write ST_C	write ST_D	write ST_E
\$1d	write LVL_A	write LVL_B	write LVL_C	write LVL_D	write LVL_E
\$1e	write PLP_A	write PLP_B	write PLP_C	write PLP_D	write PLP_E
\$1f	write TRIS_A	write TRIS_B	write TRIS_C	write TRIS_D	write TRIS_E

These abbreviations are used for the timer registers:

T1CPH, T2CPH: Timer 1/2 capture (1), high byte

T1CPL, T2CPL: Timer 1/2 capture (1), low byte

T1R1CMH, T2R1CMH: Timer 1/2 register R1, high byte

T1R1CML, T2R1CML: Timer 1/2 register R1, low byte

T1R2CMH, T2R2CMH: Timer 1/2 register R2, high byte

T1R2CML, T2R2CML: Timer 1/2 register R2, low byte

In addition to the SX 18/20/28 devices, the SX 48/52 control registers can also be read. To do this, load the M register with the correct value, and then perform a **mov !r?, w** instruction. This does not change the contents of the specified control registers, but copies its contents into the W register.

If the CMP_B or WKPND_B registers are written, the content of w is copied into the registers, and the former register contents are returned in w.

The table shows, that m register values ranging from \$10 through \$1F are valid here, i.e. the m register bit 4 is set in this case. As the mode instruction only writes the lower four bits of the instruction argument into the M register, the mode instruction cannot be used to load values greater than \$0F into m.

You need to either use the two instructions

```
mov w, #<M Value>  
mov m, w
```

or the following macro definition:

```
_mode MACRO 1  
  IFDEF SX48_52  
    mov w, #\1  
    mov m, w  
  ELSE  
    mov m, #\1  
  ENDIF  
ENDM
```


The ISR reacts on the interrupt and performs the necessary instructions to handle the interrupt event.

When the ISR has done its job, it must terminate with a **reti** (return from interrupt) or a **retiw** (return from interrupt with w) instruction. This returns the program execution to the interrupted program segment, and enables new interrupts. There, the instruction immediately following the one that was last executed before the ISR was called will be executed next. Therefore, it is necessary to save the address (i.e. the contents of the PC register) of that instruction before branching into the ISR. Fortunately, this is automatically performed by the SX controller, so there is no need for a specific instruction required in the application program. This return address is stored in a dedicated register, and not in the stack memory for subroutine return addresses.

In most cases, the ISR needs to make use of some of the controller's special registers like W, STATUS, or FSR. At invocation of an ISR these registers are automatically saved in "shadow registers", and on execution of a **reti** or **retiw** instruction, the original register contents are automatically restored from these shadow registers. This feature of the SX controllers is an important enhancement compared to other microcontrollers. It makes the design of ISRs much easier, and saves the additional execution time that would be required when the ISR would have to take care of saving and restoring these registers.

2.2.5.1.2 *Timed Interrupts*

Timed interrupts are interrupts that are invoked constantly after a certain time has elapsed. This type of interrupts is very important to realize Virtual Peripherals for the SX where precise timing very often is a key issue.

To obtain a time basis, the system clock is used to clock the Real Time Clock Counter (RTCC). As an option, the system clock may be divided by the prescaler to obtain longer periods. The RTCC is incremented on each active transition on its input, and when the RTCC overflows from 255 to 0, an interrupt will be triggered.

The time between two interrupts is determined by the following factors:

- System clock frequency
- Prescaler divide-by factor
- Contents of the RTCC at the end of the ISR (the RTCC can be set to any value between 0 and 255 when the ISR terminates)

Programming the SX Microcontroller

2.2.5.1.3 Interrupts Triggered After a Certain Number of Events

This mode is similar to timed interrupts, but this time, the RTCC is clocked from an external signal at the RTCC input, and not by the system clock. Again, an interrupt is triggered when the RTCC overflows, and at termination of the ISR, the RTCC register should be set to a new initial value.

Loading the RTCC with 255, for example, would cause an interrupt on each active transition on the RTCC input, loading it with 254 would generate an interrupt every second transition, etc.

2.2.5.1.4 Configuring the Various Interrupt Modes

To allow transitions on Port B inputs trigger an interrupt, the corresponding bits in the WKEN_B configuration register must be cleared, and WKED_B bits must be cleared if positive transitions shall trigger an interrupt.

If transitions at Port B inputs shall only cause wake up events, it makes sense to clear the bits in the WKEN_B register immediately before entering the sleep mode in order to avoid interrupts caused by transitions on those lines during regular program execution.

It is important that within the ISR the WKPND_B register bits are cleared even if there is no need to find out the interrupt reason, because otherwise a new interrupt would occur immediately after termination of the ISR.

For timed interrupts the OPTION register bits must be configured as follows:

OPTION							
7	6	5	4	3	2	1	0
RTW	RTI	RTS	RTE	PSA	PS2	PS1	PS0
1	X	0	0	X	X	X	X

RTW: 1

RTI: 0 = Interrupts are enabled

1 = Interrupts are disabled

RTS: 0

RTE: 0

PSA: 0 = use prescaler for RTCC

1 = no prescaler

PS2...PS0: For PSA = 0, these bits determine the prescaler divide by factor:

PS2	PS1	PS0	Divide by factor for the RTCC
0	0	0	1:2
0	0	1	1:4
0	1	0	1:8
0	1	1	1:16
1	0	0	1:32
1	0	1	1:64
1	1	0	1:128
1	1	1	1:256

Note that OPTION bit 6 (RTI) is the "main switch" to turn interrupts on or off.

For counting events, the OPTION register bits must be configured as follows:

OPTION							
7	6	5	4	3	2	1	0
RTW	RTI	RTS	RTE	PSA	PS2	PS1	PS0
1	X	1	X	n.a.	n.a.	n.a.	n.a.

RTW: 1

RTI: 0 = Interrupts are enabled
 1 = Interrupts are disabled

RTS: 1

RTE: 0 = RTCC is incremented on positive transitions at the RTCC input
 1 = RTCC is incremented on negative transitions at the RTCC input

Note that OPTION bit 6 (RTI) again is the "main switch" to turn interrupts on or off.

2.2.5.2 The Watchdog Timer

The watchdog timer (WDT) can be activated to reset the controller after a certain time-period in case the WDT is not cleared before that time has elapsed. This timer is an 8-bit counter, similar to the RTCC with the exception that the WDT is clocked by an internally generated separate clock signal of approximately 14 kHz. As an option, the prescaler can be used to divide down that frequency (in this case, the prescaler is not available for timed interrupts).

As soon as the WDT register overflows, it generates a reset, and program execution starts at the main entry point that is also executed first after power on.

Programming the SX Microcontroller

During regular program execution, it is important to periodically clear the WDT in order to avoid resets caused by WDT overflows. The

clr !wdt

instruction is available to perform that task. Usually, this instruction is inserted at a location in the main application program that is executed often enough to reset the WDT in time as long as the application performs regularly.

To assign the prescaler to the WDT, Bit 3 (PSA - Prescaler Assign) in the OPTION register must be set. Again, OPTION bits 2...0 (PS2...0) configure the divide by factor:

PS2	PS1	PS0	Divide by factor for the WDT
0	0	0	1:1
0	0	1	1:2
0	1	0	1:4
0	1	1	1:8
1	0	0	1:16
1	0	1	1:32
1	1	0	1:64
1	1	1	1:128

Note that this time the divide by factors range from 1:1 up to 1:128. When the maximum divide by factor is configured, a reset is generated after approximately 2.3 seconds, and the shortest period at a 1:1 ratio is approximately 18 ms.

The watchdog timer can also be used to wake up the SX controller from sleep mode as the WDT continues to work (when activated) during sleep mode. This makes it possible to develop systems with a minimum of power consumption when monitoring external conditions is only required after a certain time period between 18 ms and 2.3 seconds.

To activate the WDT, bit 2 in the "Fuse" register must be set (more about the fuse registers later in this section). The SX assemblers usually support a DEVICE WATCHDOG directive to control the setting of that bit.

2.2.5.3 Additional Bits in the OPTION Register

By default, the RTCC register is mapped into the global data register bank at location \$01, and it may be accessed by read and write instructions. The W register cannot be accessed via a data

register address by default. Clearing the OPTION register bit 7 (RTW - RTCC or W) bit maps the W register to address \$01 instead of the RTCC.

As mentioned before, OPTION bit 6 (RTI - Real Time Interrupt) is the "main switch" to turn interrupts generated by RTCC overflows on or off.

By default, bits 7 and 6 in the OPTION register are configured as read-only, i.e. they cannot be modified by program instructions. To configure these bits for read/write, bit 7 in the FUSEX fuse register must be cleared (more about the Fuse registers later in this section). The SX assemblers usually support a DEVICE OPTIONX directive to control the setting of that bit. Usually, this directive also enables the eight-level return stack, where the DEVICE STACKX directive usually also configures OPTION bits 7 and 6 as read/write, i.e. only one directive is required to activate the "enhanced" SX features.

2.2.5.4 Monitoring V_{DD} - The Brown-Out Detection

The SX data registers are located in SRAM memory, i.e. they will lose their contents when the supply voltage V_{DD} is turned off, and when V_{DD} drops below a certain level, there is no guarantee that the registers keep their original contents. The same might be true for any external components, connected to the SX controller.

In case of a short V_{DD} drop, it can happen that the status of external components changes, or that the SX registers lose their original contents. If this voltage drop is not long enough to cause a reset, it is most likely that the system will no longer function correctly.

In order to capture such errors, the SX controllers have a "built-in" brown-out detection that causes a system reset if the supply voltage drops below a certain level.

Use the DEVICE directive together with BOR42, BOR26, BOR22, or BOROFF to select a level of 4.2, 2.6, 2.2 Volts, or to turn off the brown-out detection.

To activate the brownout detection, bits 5 and 4 in the FUSEX fuse register must be configured accordingly (more about the Fuse registers later in this section).

2.2.5.5 Determining the Reason for a Reset

After resets caused by a power-on, or brown-out event, it is usually necessary to initialize certain registers in data memory to set up the status for a "clean" start. This situation is also called a "cold boot".

When the watchdog timer causes a reset, there are chances that (depending on the application program) certain registers still contain valid data, i.e. in this case, the contents of such registers might not be reset. This situation is also called a "warm boot".

Programming the SX Microcontroller

Furthermore it is possible that the WDT or transitions at port RB inputs cause a reset from sleep mode. This case is not an error condition, but an expected event, and it is most likely that none of the register contents should be re-initialized.

The STATUS register bits 4 and 3 reflect the reason for a reset, and thus allow an application program to perform the right actions:

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

TO (Time Out):

This bit is set when the power supply is turned on, and when the **clr !wdt** and **sleep** instructions are executed. When the WDT causes a reset, this bit is cleared.

When this bit is tested immediately after entering the main application program, the code may branch into the "cold boot section" if the bit is set, and into the "warm boot section", when the bit is cleared.

PD (Power Down):

This bit is set when the power supply is turned on, and when the **clr !wdt** instruction is executed, but it is cleared after **sleep** instruction.

This allows the application program to test if the TO and PD bits are both cleared, which indicates a wake up caused by the WDT or by a port B signal transition. In case the TO bit is cleared, and the PD bit is set, the WDT caused a reset due to an error.

The ALU and the STATUS Register

Arithmetic and logical operations are performed by the Arithmetic and Logical Unit (ALU) function block. The block diagram shows that the ALU takes two input values, one from the W register, and one from a data register. Thus, the W register has a special importance for arithmetic and logical operations, as it "delivers" one of the two operands.

The result of an arithmetic or logical operation is usually taken from the ALU, and it is stored in the data register specified together with the arithmetic or logical instruction, i.e. into the register that contained the second operand before. In addition, the SX controllers support several instructions that place the result into the W register, leaving the original contents of the data register unchanged.

Depending on the result of arithmetic or logical instructions, the ALU generates a status information that is stored in some of the STATUS register bits. The application program may later test these bits:

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

- Z (Zero) The Z flag is set when an operation results in 0. This may be true after an addition, or subtraction as well as after an increment, or decrement operation. The Z flag will also be set by some **mov** instructions when the register contains zero after the **mov**.
- C (Carry) The C flag is set then the result of an addition causes an overflow, i.e. when the result is greater than 255, otherwise, it will be cleared. After a subtraction, the C flag will be cleared if the result is negative, i.e. when the operation caused an underflow, otherwise, it will be set.
- The C flag is also used as 9th bit for rotate instructions.
- DC (Digit Carry) The DC flag is set when an addition caused an overflow from bit 3 to bit 4, otherwise, the flag will be cleared. After a subtraction, the bit will be cleared when the difference of the lower four bits results in a negative value, otherwise, the flag will be set. The DC flag is helpful when performing BCD arithmetic operations.

2.2.6 The Stack Memory

To make sure that the calling program can be continued correctly after a subroutine call (similar to an interrupt), it is necessary to save the "return address", i.e. the address of the instruction in the calling program code that must be executed next after a return from the subroutine. When leaving the subroutine, this address must be moved into the PC register to resume "regular" program execution.

Usually, there are situations that one subroutine might call another subroutine, etc. In order to make that possible (extended to a certain level of "nested" subroutine calls), it is not sufficient to just save one PC register contents for later restore. In order to save more than one return address, usually a stack memory is used. The SX controllers maintain a return address stack that can hold up to eight return addresses, i.e. subroutine calls may be nested up to a level of eight.

The management of the stack memory is internally handled by the SX controller, i.e. there is no need to add specific instructions in the application program. Whenever a **call** instruction is executed, the return address is "pushed" on the stack, and a **ret** instruction "pops off" the most recently saved return address from the stack releasing the stack memory for another "push".

Programming the SX Microcontroller

It is important to know that there is no way to check for stack overflows, i.e. if more than 8 return addresses are "pushed" on the stack, the oldest address is lost, so the application program must be designed in a way that stack overflow situations may not occur.

Also, there is no way to test for stack underflows. This may occur when a **ret** instruction is executed in a program without a prior **call**. Unpredictable results are guaranteed in such cases.

It is also important to configure the SX device to make the eight-level stack available by clearing bit 7 in the FUSEX fuse register. Otherwise, the stack memory is configured to the default that only allows for two return addresses.

The stack memory in the SX controllers is only dedicated to store subroutine return addresses. It is not available as a temporary storage for variable data like in most microprocessors.

2.2.7 The FUSE Registers

2.2.7.1 The FUSE Registers (SX18/20/28)

The fuse registers are assigned to dedicated memory areas in the EEPROM section of the SX controllers that cannot be changed by program instructions. Instead, these memory locations are accessed during programming the SX EEPROM memory in order to configure the general device characteristics that are valid until the EEPROM memory is re-written during the next programming cycle.

The name "Fuse Registers" has a historical background, which comes from microcontrollers with program memory that could only be programmed once. Such devices contained "fuse links" for the device options. Similar to a fuse, these links were "burned" once, or left intact for a specific option.

The SX devices come with three fuse registers (FUSE, FUSEX, and DEVICE) of 12 bits each. The meaning of the bits in those registers will be explained below. The fuse bits are set or cleared during device programming, and the assemblers usually support various DEVICE directives to control these bits from the application source code. When a DEVICE directive is not included in the source code, defaults are assumed.

2.2.7.1.1 The FUSE register

FUSE											
11	10	9	8	7	6	5	4	3	2	1	0
/TURB O	/SYN C	res	res	/IRC	DIV1, /IFBD	DIV0, FOSC2	res	/CP	WDTE	FOSC1	FOSC0

Bits marked with a leading slash (/) have negative logic, i.e. the designated function is active if the bit is cleared.

Bits marked as "res" are reserved for future use.

/TURBO: 0 = Turbo mode is active, i.e. execution of a "straight" instruction takes one clock cycle, i.e. the instruction pipeline is active.

1 = "Compatibility mode" is active, i.e. execution of a "straight" instruction takes four clock cycles, i.e. the instruction pipeline is not active.

Use the assembler directive `DEVICE TURBO` to activate the "turbo" mode (default: No turbo mode).

/SYNC: 0 = Sync mode for port inputs is active (for debugging purposes). This feature is only available in turbo mode.

1 = Sync mode is not active.

While sync mode is active, reading of input data is synchronized with the system clock. This mode is mainly required for debugging purposes. Use the assembler directive `DEVICE SYNC` to activate that mode (default: No sync mode).

/IRC: 0 = Internal system clock generator is activated.

1 = Internal system clock generator is not activated, pins OSC1 und OSC2 perform as configured by the FOSC2 and FOSC0 bits.

When one of the assembler directives `DEVICE IRCDIVx`, or `OSCxxxHz` is specified, this bit is cleared to activate the internal clock generator (default: Disabled).

DIV1...0: If the internal clock generator is active (`/IRC = 0`), these bits determine the divide by factor for this generator:

00 = 4 MHz

01 = 1 MHz

10 = 128 kHz

11 = 32 kHz

Use the assembler directives `DEVICE OSC4MHZ`, `OSC1MHZ`, `OSC128KHZ`, or `OSC32KHZ` to select the desired clock frequency (default: 4 MHz).

/IFBD: (Internal Feedback Disable) If `/IRC` and `/IFBD` are both set, the internal resistor between OSC1 and OSC2 is enabled, if `/IFBD` is cleared, an external resistor must be used. Use the assembler directive `DEVICE IFBD` to set this option (default: Enabled).

Programming the SX Microcontroller

/CP: (Code Protection) If this bit is cleared, the program code stored in the SX program memory cannot be read back, but only deleted by reprogramming the EEPROM memory. Actually, reading the EEPROM memory is possible with the CP bit cleared, but scrambled data will be read instead of the actual contents of the program memory.

Use the assembler directive `DEVICE PROTECT` to activate this mode (default: Disabled).

WDTE: 0 = Watchdog Timer is disabled.
1 = Watchdog Timer is enabled.

Use the assembler directive `DEVICE WATCHDOG` to enable the watchdog timer (default: Disabled).

FOSC2...0: These three bits configure the external oscillator driver, i.e. they define how the SX controller shall interact with external oscillator components if the internal oscillator is disabled (`/IRC=1`), and FUSE register bit 5 is used as FOSC2 bit when the internal oscillator is disabled (`/IRC=1`). In this case, the three bits FOSC2...0 configure the mode of the oscillator driver as follows, and the `OSCxx` `DEVICE` directives may be used to select a specific mode:

000 = LP1 - Crystal, low power, 32 kHz (OSCLP1)
001 = LP2 - Crystal/Resonator, low power, 32 kHz ... 1MHz (OSCLP2)
010 = XT1 - Crystal/Resonator, low power, 32 kHz ... 10 MHz (OSCXT1)
011 = XT2 - Crystal/Resonator, 1 MHz ... 24 MHz (OSCXT2)
100 = HS1 - Crystal/Resonator, 1 MHz ... 50 MHz (OSCHS1)
101 = HS2 - Crystal/Resonator, 1 MHz ... 50 MHz (OSCHS2)
110 = HS3 - Crystal/Resonator, 1 MHz ... 50 MHz (OSCHS3)
111 = EXTRC - External RC-network (OSCRC)

Default: OSCRC

2.2.7.1.2 The FUSEX Register

FUSEX											
11	10	9	8	7	6	5	4	3	2	1	0
IRCTrim2	Pins	IRCTrim1	IRC-Trim0	/OPTIONX, /STACKX	/CF	BOR1	BOR0	BOR TRIM1	BOR TRIM0	BP1	BP0

IRCTrim: Bits IRCTrim2...0 are used to fine-tune the internal RC oscillator to the specified 4 MHz clock frequency ($\pm 8\%$). Devices leave the factory un-tuned, and the pro-

programming system should test the clock frequency, and make the necessary adjustments. The lowest frequency is generated with all bits cleared, and the highest frequency can be obtained with all bits set.

Pins: Defines the number of pins the device has (18/20 or 28). Use the assembler directives like `DEVICE PINS18`, `SX18AX`, `PINS20`, `SX20AC`, `PINS28` or `SX28AC` to define the device type (default: `PINS18`).

`/OPTIONX`, `/STACKX`:

When this bit is set, `OPTION` bits 7 (`RTW`) and 6 (`RTI`) are read only, and the stack size is limited to two levels. In order to obtain an 8-level stack, and make the `OPTION` bits read/write, clear this bit using the assembler directive `DEVICE OPTIONX`, or `STACKX` (default: 2-level stack, bits 7 and 6 read-only).

`/CF`: When this bit is cleared, add and subtract operations use the carry flag as input. When the bit is set, the carry flag is ignored by such operations. Use the assembler directive `DEVICE CARRYX` to clear that flag (default: Carry flag is ignored).

BOR1, 0: These bits configure the brownout reset function, and the threshold voltage:

00 = 4.2 V
01 = 2.6 V
10 = 2.2 V
11 = Brown-out reset disabled

Use the assembler directives `DEVICE BOR42`, `BOR26` or `BOR22` to select a voltage, and `BOROFF` to disable this feature (default: Brownout disabled).

BORTRIM: These bits are used to fine-tune the selected threshold. They are adjusted by the programming system.

BP1, BP0: These bits are defined during the device fabrication process, and they should not be changed unless you want to limit the device's internal memory sizes to the following values:

00 = 1 Page, 1 Bank
01 = 1 Page, 2 Banks
10 = 4 Pages, 4 Banks
11 = 4 Pages, 8 Banks

Reducing the memory size might result in a decrease of device programming time. Use the assembler directives `DEVICE BANKS1`, `BANKS2`, `BANKS4`, or `BANKS8` to select a configuration (default: `BANKS8`).

Programming the SX Microcontroller

2.2.7.1.3 The DEVICE Register

DEVICE											
11	10	9	8	7	6	5	4	3	2	1	0
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.

This register is read-only, and it contains information about the SX device type: \$FFE = SX 18/20/28.

2.2.7.2 The Fuse Registers (SX 48/52)

2.2.7.2.1 The FUSE Register

FUSE											
11	10	9	8	7	6	5	4	3	2	1	0
-	/SYNC	-	-	/IRC	DIV1, /IFBD	DIV0, FOSC2	XTLBUF_EN	/CP	WDTE	FOSC1	FOSC0

Bits marked with a leading slash (/) have negative logic, i.e. the designated function is active if the bit is cleared.

/SYNC: 0 = Sync mode for port inputs is active (for debugging purposes). This feature is only available in turbo mode.

1 = Sync mode is not active.

While sync mode is active, reading of input data is synchronized with the system clock. This mode is mainly required for debugging purposes. Use the assembler directive `DEVICE SYNC` to activate that mode (default: No sync mode).

/IRC: 0 = Internal system clock generator is activated.

1 = Internal system clock generator is not activated, pins OSC1 und OSC2 perform as configured by the FOSC2 and FOSC0 bits.

When one of the assembler directives `DEVICE IRCDIVx`, or `OSCxxxHZ` is specified, this bit is cleared to activate the internal clock generator (default: Disabled).

DIV1...0: If the internal clock generator is active (`/IRC = 0`), these bits determine the divide by factor for this generator:

00 = 4 MHz

01 = 1 MHz

10 = 128 kHz

11 = 32 kHz

- Use the assembler directives `DEVICE OSC4MHZ`, `OSC1MHZ`, `OSC128KHZ` or `OSC32KHZ` to set up the divide by factor (default: 4 MHz).
- `/IFBD`: (Internal Feedback Disable) If `/IRC` and `/IFBD` are both set, the internal resistor between `OSC1` and `OSC2` is enabled, if `/IFBD` is cleared, an external resistor must be used. Use the assembler directive `DEVICE IFBD` to set this option (default: Enabled).
- `XTLBUF_EN`: When this bit is set, the internal buffer for crystal or ceramic resonator clock devices is activated. If you do not use such device, clear the bit to reduce power consumption.
- `/CP`: (Code Protection) If this bit is cleared, the program code stored in the SX program memory cannot be read back, but only deleted by reprogramming the EEPROM memory. Actually, reading the EEPROM memory is possible with the `CP` bit cleared, but scrambled data will be read instead of the actual contents of the program memory.
- Use the assembler directive `DEVICE PROTECT` to activate this mode (default: Disabled).
- `WDTE`:
0 = Watchdog Timer is disabled.
1 = Watchdog Timer is enabled.
- Use the assembler directive `DEVICE WATCHDOG` to activate this mode (default: Disabled).
- `FOSC2...0`: These three bits configure the external oscillator driver, i.e. they define how the SX controller shall interact with external oscillator components if the internal oscillator is disabled (`/IRC=1`), and `FUSE` register bit 5 is used as `FOSC2` bit when the internal oscillator is disabled (`/IRC=1`). In this case, the three bits `FOSC2...0` configure the mode of the oscillator driver as follows, and the `OSCxx` `DEVICE` directives may be used to select a specific mode:
- 000 = LP1 - Crystal, low power, 32 kHz (`OSCLP1`)
 - 001 = LP2 - Crystal/Resonator, low power, 32 kHz ... 1MHz (`OSCLP2`)
 - 010 = XT1 - Crystal/Resonator, low power, 32 kHz ... 10 MHz (`OSCXT1`)
 - 011 = XT2 - Crystal/Resonator, 1 MHz ... 24 MHz (`OSCXT2`)
 - 100 = HS1 - Crystal/Resonator, 1 MHz ... 50 MHz (`OSCHS1`)
 - 101 = HS2 - Crystal/Resonator, 1 MHz ... 50 MHz (`OSCHS2`)
 - 110 = HS3 - Crystal/Resonator, 1 MHz ... 50 MHz (`OSCHS3`)
 - 111 = EXTRC - External RC-network (`OSCRC`)
- Default: `OSCRC`

Programming the SX Microcontroller

2.2.7.2.2 The FUSEX Register

FUSEX											
11	10	9	8	7	6	5	4	3	2	1	0
IRCTrim2	/SLEEPCLK	IRCTrim1	IRCTrim0	-	/CF	BOR1	BOR0	BOR TRIM1	BOR TRIM0	DRT1	DRT0

IRCTrim: Bits IRCTrim2...0 are used to fine-tune the internal RC oscillator to the specified 4 MHz clock frequency ($\pm 8\%$). Devices leave the factory un-tuned, and the programming system should test the clock frequency, and make the necessary adjustments. The lowest frequency is generated with all bits cleared, and the highest frequency can be obtained with all bits set.

/SLEEPCLK: When this bit is cleared, clock generation continues while the device is in sleep mode (for fast wake up). Set this bit if this feature is not required in order to reduce power consumption in sleep mode. Use the assembler directive `DEVICE SLEEPCLK` to enable the clock (default: Clock disabled).

/CF: When this bit is cleared, add and subtract operations use the carry flag as input. When the bit is set, the carry flag is ignored by such operations. Use the assembler directive `DEVICE CARRYX` to clear that flag (default: carry flag is ignored).

BOR1, 0: These bits configure the brownout reset function, and the threshold voltage:

00 = 4,2 V

01 = 2,6 V

10 = 2,2 V

11 = Brown-out reset disabled

Use the assembler directives `DEVICE BOR42`, `BOR26` or `BOR22` to select a voltage, and `BOROFF` to disable this feature (default: Brownout disabled).

BORTRIM: These bits are used to fine-tune the selected threshold. They are adjusted by the programming system.

DRT1/0: These two bits define the delay reset timer timeout period:

00 = 60 ms (WDRT60)

01 = 1 s (WDRT960)

10 = 0.25 ms (WDRT006)

11 = 18 ms (WDRT184)

Use the assembler directives `DEVICE WDRTxxx`, as specified in parentheses above to select a delay time (default: 18 ms).

2.2.7.2.3 The DEVICE Register

DEVICE												
11	10	9	8	7	6	5	4	3	2	1	0	
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	

This register is read-only, and it contains information about the SX device type: \$001 = SX 48/52.

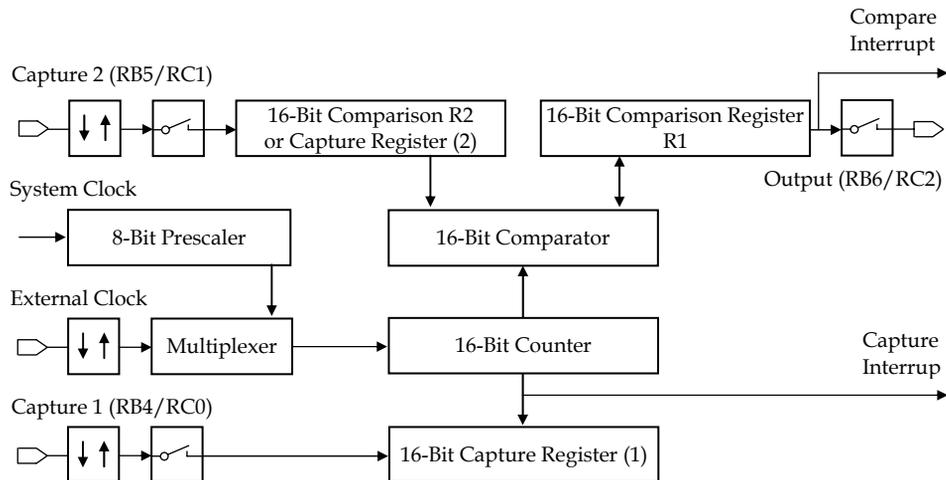
2.2.8 The SX 48/52 Multi-Function Timers

In addition to the standard timers (RTCC and watchdog), the SX 48/52 devices come with two Multi-Function Timers T1 and T2. These timers are useful to replace a software solution for generating PWM signals, counting events, and generating longer time delays.

Each timer comes with a free-running 16-bit counter. At reset, the counters are initialized with \$0000, and then, they start continuously counting upwards. The counters can either be clocked from the system clock (through an 8-bit prescaler), or from an external transition at the external clock pin. This input can be configured to sense positive, or negative transitions.

Each counter has associated 16-bit capture, and comparison registers. As an option, various events can be used to trigger an interrupt, or to generate an output signal.

The block diagram shows the components of one timer:



Programming the SX Microcontroller

Registers R1, R2, and the capture registers can be accessed by `mov !rb, w` (Timer1), or `mov !rc, w` (Timer2) instructions, where the remaining registers cannot be accessed via software.

Timer 1 shares its input and output lines with the Port B pins 4..7, and timer 2 shares its input and output lines with the Port C pins 0..3. If a timer is active, those pins can no longer be used for "regular" I/O purposes.

2.2.8.1 PWM Mode

In this mode, the timer generates a square wave signal with programmable frequency, and duty cycle. For this purpose, the contents of the two comparison registers determine for how long the signal is high, and low.

The 16-bit counter starts with a contents of 0, and keeps incrementing until it has reached the value of R1. The counter is reset to 0, the output is toggled, and (if enabled) an interrupt is generated.

Next, the counter keeps incrementing until it now has reached the value of R2. Again, the counter is reset to 0, the output signal is toggled, and an interrupt is triggered (if enabled).

These two steps are repeated continuously. The contents of R1 and R2 determine the frequency and the duty cycle of the generated output signal. When R1 and R2 contain the same value, a square wave with a duty cycle of 50% is generated. In order to generate a signal with a constant frequency, and a varying duty cycle, the sum $R1+R2$ must remain constant, i.e. to change the duty cycle, increase the value of one register, and decrease the value of the other register by the same amount.

In PWM mode, the 16-bit counter is clocked through the prescaler from the system clock. The prescaler can be set to divide-by factors from 1 to 256 in steps of powers of two.

2.2.8.2 Software Timer Mode

This mode is similar to the PWM mode with the difference that the output signal is not toggled. Instead, the application program must react on the interrupts that indicate a match between the counter and R1, or between the counter and R2. An additional interrupt is generated when the counter overflows from \$ffff to \$0000.

2.2.8.3 External Event Counter

Again, this mode is similar to the PWM mode, but here, the 16-bit counter is clocked from an external signal instead from the system clock. The external input can be configured in order to have positive or negative transitions increment the counter.

2.2.8.4 Capture/Compare Mode

In this mode, the 16-bit counter is clocked by the prescaled system clock, and it keeps incrementing without being reset. A valid transition at one of the two inputs causes that the current counter contents is stored in the associated capture register. This makes it easy to determine the time difference between two external events.

In addition, the counter contents are continuously compared against the contents of register R1. If both are equal, an interrupt is generated (if enabled), and the output signal is toggled. Different from the PWM mode, the counter is not reset in this case but it keeps incrementing.

In order to obtain a fixed period between the interrupts and output toggles, the ISR must load a new value into R1 whenever an interrupt is triggered.

The two inputs Capture 1 and Capture 2 can be configured to have positive or negative transitions trigger the capture.

Capture register 1 is a separate register dedicated to capture the counter contents only, where Register R2 is used for the capture register 2.

As an option, each capture event can also issue an interrupt, and various flags allow the ISR to determine the interrupt reasons.

In addition, a 16-bit counter overflow can also trigger an interrupt, and set a flag. This is important when the time between two external events is long enough to allow for one or more counter overflows. If the ISR keeps track of the number of overflows, it is possible to calculate the time elapsed between two external events.

2.2.8.5 The SX48/52 Timer Control Registers

The various modes for each of the two timers are configured by bits in two 8-bit control registers:

2.2.8.5.1 The T1CNTA Register

T1CNTA Register							
7	6	5	4	3	2	1	0
T1CPF2	T1CPF1	T1CPIE	T1CMF2	T1CMF1	T1CMIE	T1OVF	T1OVIE

T1CPF2: Timer 1 Capture Flag 2 - This flag is automatically set in the capture/compare mode when a capture event occurs at capture input 2. It remains set until cleared by the application software.

Programming the SX Microcontroller

- T1CPF1:** Timer 1 Capture Flag 1 - This flag is automatically set in the capture/compare mode when a capture event occurs at capture input 1. It remains set until cleared by the application software.
- T1CPIE:** Timer 1 Capture Interrupt Enable - If this bit is set, a valid transition at the capture inputs 1 or 2 generates an interrupt.
- T1CMF2:** Timer 1 Comparison Flag 2 - This bit is automatically set when the contents of the 16-bit counter, and of R2 match, in case R2 is configured as comparison register. It remains set until cleared by the application software.
- T1CMF1:** Timer 1 Comparison Flag 1 - This bit is automatically set when the contents of the 16-bit counter, and of R1 match. It remains set until cleared by the application software.
- T1CMIE:** Timer 1 Comparison Interrupt Enable - If this bit is set, a match between the 16-bit counter, and R1 or R2 generates an interrupt.
- T1OVF:** Timer 1 Overflow Flag - This bit is automatically set when the contents of the 16-bit counter overflows from \$FFFF to \$0000. It remains set until cleared by the application software.
- T1OVIE:** Timer 1 Overflow Interrupt Enable - If this bit is set, an overflow of the 16-bit counter generates an interrupt.

2.2.8.5.2 The T1CNTB Register

T1CNTB Register							
7	6	5	4	3	2	1	0
RTCCOV	T1CPEDG	T1EXEDG	T1PS2...T1PS0			T1MC1...T1MC0	

- RTCCOV:** RTCC Overflow Flag - This bit is automatically set, when the Real Time Clock Counter overflows from \$FF to \$00. It remains set until cleared by the application software. This flag is not directly associated to the Multi-Function Timers, but it makes it easier to find out the interrupt reason in the ISR.
- T1CPEDG:** Timer 1 Capture Edge - This bit determines which transition at inputs 1 and 2 shall trigger an event. If the bit is set, positive (low-high) transitions are recognized, and if the bit is cleared, negative (high-low) transitions are sensed.
- T1EXEDG:** Timer 1 External Event Clock Edge - This bit determines which transition at the external clock input shall increment the 16-bit counter. If the bit is set, positive

(low-high) transitions are sensed, and if the bit is cleared, negative (high-low) transitions are sensed instead.

T1PS2...0: Timer T1 Prescaler - These three bits configure the prescaler divide-by factor:

- 000 - 1/1
- 001 - 1/2
- 010 - 1/4
- 011 - 1/8
- 100 - 1/16
- 101 - 1/32
- 110 - 1/64
- 111 - 1/128

T1MC1...0: Timer 1 Mode Control - These two bits configure the timer 1 mode:

- 00 - Software Timer Mode
- 01 - PWM Mode
- 10 - Capture/Compare Mode
- 11 - External Event Counter

2.2.8.5.3 *The T2CNTA Register*

T2CNTA Register							
7	6	5	4	3	2	1	0
T2CPF2	T2CPF1	T2CPIE	T2CMF2	T2CMF1	T2CMIE	T2OVF	T2OVIE

T2CPF2: Timer 2 Capture Flag 2 - This flag is automatically set in the capture/compare mode when a capture event occurs at capture input 2. It remains set until cleared by the application software.

T2CPF1: Timer 2 Capture Flag 1 - This flag is automatically set in the capture/compare mode when a capture event occurs at capture input 1. It remains set until cleared by the application software.

T2CPIE: Timer 2 Capture Interrupt Enable - If this bit is set, a valid transition at the capture inputs 1 or 2 generates an interrupt.

T2CMF2: Timer 2 Comparison Flag 2 - This bit is automatically set when the contents of the 16-bit counter, and of R2 match, in case R2 is configured as comparison register. It remains set until cleared by the application software.

Programming the SX Microcontroller

- T2CMF1:** Timer 2 Comparison Flag 1 - This bit is automatically set when the contents of the 16-bit counter, and of R1 match. It remains set until cleared by the application software.
- T2CMIE:** Timer 2 Comparison Interrupt Enable - If this bit is set, a match between the 16-bit counter, and R1 or R2 generates an interrupt.
- T2OVF:** Timer 2 Overflow Flag - This bit is automatically set when the contents of the 16-bit counter overflows from \$FFFF to \$0000. It remains set until cleared by the application software.
- T2OVIE:** Timer 2 Overflow Interrupt Enable - If this bit is set, an overflow of the 16-bit counter generates an interrupt.

2.2.8.5.4 The T2CNTB Register

T1CNTB Register							
7	6	5	4	3	2	1	0
PORTDR	T2CPEDG	T2EXEDG	T2PS2...T2PS0			T2MC1...T2MC0	

- PORTDR:** Port Read Mode - This bit is not associated to Timer 2. It is used to configure how data is read from the I/O ports RA...RE. When the bit is clear, data is read directly from the port pins, and if the bit is set, data is read from the port registers. Under regular conditions, it makes no difference, which mode is used, but if - for example - a port line is configured as an output, and the port bit is set, the line is pulled to high level. When by some reason, an external component pulls the line down to low level, it makes a difference: Reading the port bit returns a 0, but reading the port register returns a 1.
- T2CPEDG:** Timer 2 Capture Edge - This bit determines which transition at inputs 1 and 2 shall trigger an event. If the bit is set, positive (low-high) transitions are recognized, and if the bit is cleared, negative (high-low) transitions are sensed.
- T2EXEDG:** Timer 2 External Event Clock Edge - This bit determines which transition at the external clock input shall increment the 16-bit counter. If the bit is set, positive (low-high) transitions are sensed, and if the bit is cleared, negative (high-low) transitions are sensed.

T2PS2...0: Timer T2 Prescaler - These three bits configure the prescaler divide-by factor:

000 - 1/1

001 - 1/2

010 - 1/4

011 - 1/8

100 - 1/16

101 - 1/32

110 - 1/64

111 - 1/128

T2MC1...0: Timer 2 Mode Control - These two bits configure the timer 2 mode:

00 - Software Timer Mode

01 - PWM Mode

10 - Capture/Compare Mode

11 - External Event Counter

Programming the SX Microcontroller

A Complete Guide by Günther Daubach

2ND EDITION

Section III – Quick Reference

3 Section III – Quick Reference

3.1 SX Pin Assignments

RA2	1	18	RA1
RA3	2	17	RA0
RTCC	3	16	OSC1
/MCL	4	15	OSC2
V _{SS}	5	14	V _{DD}
RB0	6	13	RB7
RB1	7	12	RB6
RB2	8	11	RB5
RB3	9	10	RB4

SX 18
DIP/SOIC Package

RA2	1	11	RA1
RA3	2	12	RA0
RTCC	3	13	OSC1
/MCL	4	14	OSC2
V _{SS}	5	15	V _{DD}
V _{SS}	6	16	V _{DD}
RB0	7	17	RB7
RB1	8	18	RB6
RB2	9	19	RB5
RB3	10	20	RB4

SX 20
SSOP Package

RTCC	1	28	/MCL
V _{DD}	2	27	OSC1
	3	26	OSC2
V _{SS}	4	25	RC7
	5	24	RC6
RA0	6	23	RC5
RA1	7	22	RC4
RA2	8	21	RC3
RA3	9	20	RC2
RB0	10	19	RC1
RB1	11	18	RC0
RB2	12	17	RB7
RB3	13	16	RB6
RB4	14	15	RB5

SX 28
DIP/SOIC Package

V _{SS}	1	28	/MCL
RTCC	2	27	OSC1
V _{DD}	3	26	OSC2
V _{DD}	4	25	RC7
RA0	5	24	RC6
RA1	6	23	RC5
RA2	7	22	RC4
RA3	8	21	RC3
RB0	9	20	RC2
RB1	10	19	RC1
RB2	11	18	RC0
RB3	12	17	RB7
RB4	13	16	RB6
V _{SS}	14	15	RB5

SX 28
SSOP Package

Note: Some of the types/packages are no longer produced – they have been included here for completeness.

3.2 Commonly used Abbreviations

The table below lists abbreviations that are used in technical documents for the SX controllers.

Abbreviation	Meaning
ADC	A nalog- D igital C onverter – A device that converts an analog value (e.g. a voltage) into a digital representation.
C	C arry bit – This bit in the Status register is set when certain instruction results cause an overflow.
CMP_B	C omparator B control register – The bits in this register control the SX analog comparator at port B, and return the output status of the comparator.
CPU	C entral P rocessing U nit – The part of a computer/controller that performs the execution of the program instructions.
DC	D igit C arry bit – This bit in the Status register is set when certain instruction results cause an overflow from bit 3 to bit 4.
EEPROM	E lectrically E rasable P rogrammable R ead O nly M emory – A memory device that can be read at high speed, and that maintains the stored contents even when it is not powered. The device can be erased and re-programmed by special electrical signals.
FR	F ile R egister – A storage in the SX data memory.
FSK	F requency S hift K eying – A method to transmit serial digital data by varying the frequency of a carrier signal depending whether a high or low status is to be transmitted (mostly used by modems).
FSR	F ile S elect R egister – This register contains an indirect data address or the bits that select the current data memory bank.
HTTP	H yper T ext T ransfer P rotocol – The protocol, most commonly used to transfer web pages via the Internet.
I/O	I nterface I nput/ O utput – The operation of reading data from an input line, or sending data to an output line.
ICMP	I nternet C ontrol M essage P rotocol – A protocol for diagnostic messages.
IND, INDF	I ndirect through F SR – This instruction argument is used to indicate indirect addressing.
IP	I nternet P rotocol – The fundamental Internet protocol.

Programming the SX Microcontroller

Abbreviation	Meaning
LVL_A, LVL_B, LVL_C, LVL_D, LVL_E	Level registers - The bits in these registers select whether a port input line shall have TTL or CMOS characteristics.
MCU	Microcontroller Unit
MIPS	Mega instructions per second - A measure, how many instructions a controller or computer can execute per second.
MIWU	Multi-Input Wake-up
PC	Program Counter - The register that addresses the instruction in program memory to be executed.
PD	Power Down - This bit in the Status register is set upon power-up and cleared by the SLEEP instruction.
PDM	Pulse Duration Modulation - A method to impose a signal on a carrier signal by varying the pulse duration of the carrier signal.
PLP_A, PLP_B, PLP_C, PLP_D, PLP_E	Pull-up registers - The bits in these registers activate or de-activate the internal pull-up resistors of the port input lines.
POP3	Post Office Protocol version 3 - The protocol, most commonly used to receive e-mail via the Internet.
PPP	Point-to-Point Protocol - A protocol for point-to-point links, like between modems via a telephone line.
PS2:PS0	Prescaler Divide-By Factor - These bits in the OPTION register select the prescaler divide-by factor.
PSA	Prescaler Assignment - This bit in the OPTION register selects if the prescaler shall be assigned to the RTCC, or to the watchdog timer.
PSK	Phase Shift Keying - A method to transmit serial digital data by varying the phase of a carrier signal depending whether a high or low status is to be transmitted.
PWM	Pulse Width Modulation - A method to impose a signal on a carrier signal by varying the duty cycle of the carrier signal.
R/C	Resistor/Capacitor - A combination of a resistor and a capacitor to generate a time-constant.
RA, RB, RC, RD, RE	Register A (B, C, D, E) - The port data latch registers.
RFC	Request For Comments - The name of documents that describe and define Internet-related standards and methods.

Abbreviation	Meaning
RISC	Reduced Instruction Set Controller – The Uvicom microcontrollers use a RISC-based architecture, i.e. a limited set of instructions that can be executed at high speed.
RTC_ES	RTCC Input Edge Select – This bit in the OPTION register selects if a rising or falling edge at the RTCC input pin shall increment the RTCC register content.
RTS	RTCC Trigger Selection – This bit in the OPTION register selects if the RTCC shall be incremented by transitions on the RTCC input pin, or by the system clock.
RTW	RTCC or W – This bit in the OPTION register controls if the RTCC or the W register shall be mapped into address \$01 of the data memory.
SMTP	Simple Mail Transfer Protocol – The protocol, most commonly used to send e-mail via the Internet.
SRAM	Static Random Access Memory – Read/Write storage that maintains the stored contents as long as the device is powered, without the need of any dynamic read/write/refresh cycles.
ST_B, ST_C, ST_D, ST_E	Schmitt Trigger registers – The bits in these registers control whether a port input line shall have Schmitt Trigger characteristics or not.
STATUS	The Status Register – This Register contains various flags that indicate certain instruction results. It also contains the program memory page select bits.
TCP	Transmission Control Protocol – A connection-based protocol for full-duplex end-to-end communication channels.
TO	Time Out – This bit in the Status register is cleared in case the SX has been reset by the watchdog timer.
UART	Universal Asynchronous Receiver Transmitter – A device that receives and sends serial data.
UDP	User Datagram Protocol – A connection-less protocol to send data from a transmitter to a receiver.
VP	Virtual Peripheral™ – An implementation of a peripheral by software.
W	Working Register – The “Accumulator” of the SX controller. This register is used to hold an operand or temporary values.
WKED_B	Wake-up edge B register – The bits in this register select the active signal edges at port B input that shall trigger a wake-up or an interrupt.
WKPND_B	Wake-up pending B register – The bits in this register indicate which port B input line has recently caused a wake-up or an interrupt.
Z	Zero bit – This bit in the Status register is set when certain instructions return a zero result.

3.3 Instruction Overview

3.3.1 Comments on the Instruction Overview Tables

If the “Cycles” column contains two values, the first one is valid when the branch is not taken, and the second one specifies the number of clock cycles required when the branch is executed. All values respect the “Turbo Mode” - for the “Compatibility Mode”, multiply the values by four.

Instructions marked light gray must not follow a skip or conditional skip instruction.

Before execution of instructions that are marked dark gray in the left column, the C flag must be set to a defined value in case the CARRYX option has been activated. You can find the required instruction (stc) or (clc) in the “Operations” column set in parentheses.

Footnotes:

- (1) Prescaler = 0, if assigned to the WDT.
- (2) Push - Return address is saved to the stack.
- (3) Pop - Return address is restored from the stack.
- (4) Restore - w, STATUS and FSR are restored.
- (5) Restore Page - Bits PA2...0 are restored.

Abbreviations:

Addr = Address
C = C- (Carry-) Flag
DC = DC- (Digit Carry-) Flag
fr = File Register
Const = Constant
pc = Program Counter
(pc) = pc will only be changed when a condition evaluates to true
PA = Page Address Bit
PD = Power Down Bit
TO = Time Out Bit
Z = Z- (Zero-) Flag

3.3.2 Instructions in Alphabetic Order

SX Assembly Instructions in Alphabetic Order				
Instruction	Words	Cycles	Changes	Operation
ADD fr, #Const	2	2	fr, w, C, DC, Z	fr = fr + Const (CLC)
ADD fr, w	1	1	fr, C, DC, Z	fr = fr + w (CLC)
ADD fr1, fr2	2	2	fr, w, C, DC, Z	fr1 = fr1 + fr2 (CLC)
ADD w, fr	1	1	w, C, DC, Z	w = w + fr (CLC)
ADDB fr1, /fr2.Bit	2	2	fr1, Z	fr1 = fr1 + NOT fr2.Bit
ADDB fr1, fr2.Bit	2	2	fr1, Z	fr1 = fr1 + fr2.Bit
AND fr, #Const	2	2	fr, w, Z	fr = fr AND Const
AND fr, w	1	1	fr, Z	fr = fr AND w
AND fr1, fr2	2	2	fr1, w, Z	fr1 = fr1 AND fr2
AND w, #Const	1	1	w, Z	w = w AND Const
AND w, fr	1	1	w, Z	w = w AND fr
BANK fr	1	1	fsr	fr.(7-5) -> fsr.(7-5)
CALL Addr	1	3	pc	pc = Addr, Push (2)
CJA fr, #Const, Addr	4	4/6	w, C, DC, Z, (pc)	pc = Addr, if fr > Const (CLC)
CJA fr1, fr2, Addr	4	4/6	w, C, DC, Z, (pc)	pc = Addr, if fr1 > fr2 (STC)
CJAE fr, #Const, Addr	4	4/6	w, C, DC, Z, (pc)	pc = Addr, if fr >= Const (STC)
CJAE fr1, fr2, Addr	4	4/6	w, C, DC, Z, (pc)	pc = Addr, if fr1 >= fr2 (STC)
CJB fr, #Const, Addr	4	4/6	w, C, DC, Z, (pc)	pc = Addr, if fr < Const (STC)
CJB fr1, fr2, Addr	4	4/6	w, C, DC, Z, (pc)	pc = Addr, if fr1 < fr2 (STC)
CJBE fr, #Const, Addr	4	4/6	w, C, DC, Z, (pc)	pc = Addr, if fr <= Const (CLC)
CJBE fr1, fr2, Addr	4	4/6	w, C, DC, Z, (pc)	pc = Addr, if fr1 <= fr2 (STC)
CJE fr, #Const, Addr	4	4/6	w, C, DC, Z, (pc)	pc = Addr, if fr = Const (STC)
CJE fr1, fr2, Addr	4	4/6	w, C, DC, Z, (pc)	pc = Addr, if fr1 = fr2 (STC)
CJNE fr, #Const, Addr	4	4/6	w, C, DC, Z, (pc)	pc = Addr, if fr <> Const (STC)
CJNE fr1, fr2, Addr	4	4/6	w, C, DC, Z, (pc)	pc = Addr, if fr1 <> fr2 (STC)
CLC	1	1	C	C-Flag = 0
CLR !wdt	1	1	wdt, TO, PD	!wdt = 0, TO = 1, PD = 1 (1)
CLR fr	1	1	fr, Z	fr = 0
CLR w	1	1	w, Z	w = 0
CLRB fr.Bit	1	1	fr.Bit	fr.Bit = 0
CLZ	1	1	Z	Z = 0
CSA fr, #Const	3	3/6	w, C, DC, Z, (pc)	pc++, if fr > Const (CLC)
CSA fr1, fr2	3	3/6	w, C, DC, Z, (pc)	pc++, if fr1 > fr2 (STC)
CSAE fr, #Const	3	3/6	w, C, DC, Z, (pc)	pc++, if fr >= Const (STC)
CSAE fr1, fr2	3	3/6	w, C, DC, Z, (pc)	pc++, if fr1 >= fr2 (STC)
CSB fr, #Const	3	3/6	w, C, DC, Z, (pc)	pc++, if fr < Const (STC)
CSB fr1, fr2	3	3/6	w, C, DC, Z, (pc)	pc++, if fr1 < fr2 (STC)
CSBE fr, #Const	3	3/6	w, C, DC, Z, (pc)	pc++, if fr <= Const (CLC)
CSBE fr1, fr2	3	3/6	w, C, DC, Z, (pc)	pc++, if fr1 <= fr2 (STC)
CSE fr, #Const	3	3/6	w, C, DC, Z, (pc)	pc++, if fr = Const (STC)
CSE fr1, fr2	3	3/6	w, C, DC, Z, (pc)	pc++, if fr1 = fr2 (STC)
CSNE fr, #Const	3	3/6	w, C, DC, Z, (pc)	pc++, if fr <> Const (STC)

SX Assembly Instructions in Alphabetic Order				
Instruction	Words	Cycles	Changes	Operation
CSNE fr1, fr2	3	3/6	w, C, DC, Z, (pc)	pc++, if fr1 <> fr2 (STC)
DEC fr	1	1	fr, Z	fr = fr - 1
DECSZ fr	1	1/3	fr	fr = fr - 1, pc++, if fr = 0
DJNZ fr, Addr	2	2/4	fr, (pc)	fr = fr - 1, pc = Addr, if fr <> 0
Instruction	Words	Cycles	Changes	Operation
IJNZ fr, Addr	2	2/4	fr, (pc)	fr = fr + 1, pc = Addr, if fr <> 0
INC fr	1	1	fr, Z	fr = fr + 1
INCSZ fr	1	1/3	fr, (pc)	fr = fr + 1, pc++, if fr = 0
IREAD	1	4	w, m	(m:w) -> m:w
JB fr.Bit, Addr	2	2/4	(pc)	pc = Addr, if fr.Bit = 1
JC Addr	2	2/4	(pc)	pc = Addr, if C = 1
JMP Addr	1	3	pc	pc = Addr
JMP pc+w	1	3	pc, C, DC, Z	pc = Addr+w (CLC)
JMP w	1	3	pc	pc = w
JNB fr.Bit, Addr	2	2/4	pc	pc = Addr, if fr.Bit = 0
JNC Addr	2	2/4	pc	pc = Addr, if C = 0
JNZ Addr	2	2/4	pc	pc = Addr, if Z = 0
JZ Addr	2	2/4	pc	pc = Addr, if Z = 1
MODE Const	1	1	m	m = Const
MOV !option, #Const	2	2	Option, w	Option = Const
MOV !option, fr	2	2	Option, w, Z	Option = fr
MOV !option, w	1	1	Option	Option = w
MOV !port, #Const	2	2	!port, w	Port-Config. = Const
MOV !port, fr	2	2	!port, w, Z	Port-Config. = fr
MOV !port, w	1	1	!port	Port-Config. = w
MOV fr, #Const	2	2	fr, w	fr = Const
MOV fr, w	1	1	fr	fr = w
MOV fr1, fr2	2	2	fr1, w, Z	fr1 = fr2
MOV m, #Const	1	1	m	m = Const
MOV m, fr	2	2	m, w, Z	m = fr
MOV m, w	1	1	m	m = w
MOV w, #Const	1	1	w	w = Const
MOV w, fr	1	1	w, Z	w = fr
MOV w, /fr	1	1	w, Z	w = NOT fr
MOV w, ++fr	1	1	w, Z	w = fr + 1
MOV w, <<fr	1	1	w, C	w = RL fr
MOV w, <>fr	1	1	w	w = SWAP fr
MOV w, >>fr	1	1	w, C	w = RR fr
MOV w, --fr	1	1	w, Z	w = fr - 1
MOV w, fr-w	1	1	w, C, DC, Z	w = fr - w (STC)
MOV w, m	1	1	w	w = m
MOVB fr1.Bit, /fr2.Bit	4	4	fr1.Bit	fr1.Bit = NOT fr2.Bit
MOVB fr1.Bit, fr2.Bit	4	4	fr1.Bit	fr1.Bit = fr2.Bit
MOVSZ w, ++fr	1	1	w	w = fr + 1, pc++, if w = 0
MOVSZ w, --fr	1	1	w	w = fr - 1, pc++, if w = 0

SX Assembly Instructions in Alphabetic Order				
Instruction	Words	Cycles	Changes	Operation
NOP	1	1	-	-
NOT fr	1	1	fr, Z	fr = fr XOR \$FF
NOT w	1	1	w, Z	w = w XOR \$FF
OR fr, #Const	2	2	fr, w, Z	fr = fr OR Const
OR fr, w	1	1	fr	fr = fr OR w
OR fr1, fr2	2	2	fr1, w, Z	fr1 = fr1 OR fr2
OR w, #Const	1	1	w, Z	w = w OR Const
OR w, fr	1	1	w, Z	w = w OR fr
PAGE Addr	1	1	PA2-0	PA2...PA0 = Addr.(11...9)
RET	1	3	pc	Pop (3)
RETI	1	3	pc, C, DC, Z	Pop, Restore (3) (4)
RETIW	1	3	pc, C, DC, Z	Pop, Restore, RTCC += w (3) (4)
RETP	1	3	pc	Pop, Restore Page (5)
RETW Const	1	3	pc	w = Const, Pop (3)
RL fr	1	1	fr, C	fr.(7..1) = fr.(6..0), fr.0 = C, C = fr.7
RR fr	1	1	fr, C	fr.(6..0) = fr.(7..1), fr.7 = C, C = fr.0
SB fr.Bit	1	1/2	(pc)	pc++, if fr.Bit = 1
SC	1	1/2	(pc)	pc++, if C = 1
SETB fr.Bit	1	1	fr.Bit	fr.Bit = 1
SKIP	1	2	pc	pc++
SLEEP	1	1	IO, PD	IO = 1, PD = 0, Stop clock
SNB fr.Bit	1	1/2	(pc)	pc++, if fr.Bit = 0
SNC	1	1/2	(pc)	pc++, if C = 0
SNZ	1	1/2	(pc)	pc++, if Z = 0
STC	1	1	C	C = 1
STZ	1	1	Z	Z = 1
SUB fr, #Const	2	2	fr, w, C, DC, Z	fr = fr - Const (STC)
SUB fr, w	1	1	fr, C, DC, Z	fr = fr - 1 (STC)
SUB fr1, fr2	2	2	fr1, w, C, DC, Z	fr1 = fr1 - fr2 (STC)
SUBB fr1, /fr2.Bit	2	2	fr1, Z	fr1 = fr1 - NOT fr2.Bit (STC)
SUBB fr1, fr2.Bit	2	2	fr1, Z	fr1 = fr1 - fr2.Bit (STC)
SWAP fr	1	1	fr	fr.(7..4) = fr.(3..0), fr.(3..0) = fr(7..4)
SZ	1	1/2	(pc)	pc++, if Z = 1
TEST fr	1	1	Z	Z = 1, if fr = 0
TEST w	1	1	Z	Z = 1, if w = 0
XOR fr, #Const	2	2	fr, W, Z	fr = fr XOR Const
XOR fr, w	1	1	fr, Z	fr = fr XOR w
XOR fr1, fr2	2	2	fr, W, Z	fr1 = fr1 XOR fr2
XOR w, #Const	1	1	w, Z	w = w XOR Const
XOR w, fr	1	1	w, Z	w = w XOR fr

3.3.3 Instructions by Functions

SX Assembly Instructions by Functions				
Logical Instructions				
Instruction	Words	Cycles	Changes	Operation
AND fr, #Const	2	2	fr, w, Z	fr = fr AND Const
AND fr, w	1	1	fr, Z	fr = fr AND w
AND fr1, fr2	2	2	fr1, w, Z	fr1 = fr1 AND fr2
AND w, #Const	1	1	w, Z	w = w AND Const
AND w, fr	1	1	w, Z	w = w AND fr
NOT fr	1	1	fr, Z	fr = fr XOR \$FF
NOT w	1	1	w, Z	w = w XOR \$FF
OR fr, w	1	1	fr	fr = fr OR w
OR fr1, fr2	2	2	fr1, w, Z	fr1 = fr1 OR fr2
OR w, #Const	1	1	w, Z	w = w OR Const
OR w, fr	1	1	w, Z	w = w OR fr
XOR fr, #Const	2	2	fr, W, Z	fr = fr XOR Const
XOR fr, w	1	1	fr, Z	fr = fr XOR w
XOR fr1, fr2	2	2	fr, W, Z	fr1 = fr1 XOR fr2
XOR w, #Const	1	1	w, Z	w = w XOR Const
XOR w, fr	1	1	w, Z	w = w XOR fr

SX Assembly Instructions by Functions				
Arithmetic and Shift Instructions				
Instruction	Words	Cycles	Changes	Operation
ADD fr, #Const	2	2	fr, w, C, DC, Z	fr = fr + Const (CLC)
ADD fr, w	1	1	fr, C, DC, Z	fr = fr + w (CLC)
ADD fr1, fr2	2	2	fr, w, C, DC, Z	fr1 = fr1 + fr2 (CLC)
ADD w, fr	1	1	w, C, DC, Z	w = w + fr (CLC)
ADDB fr1, /fr2.Bit	2	2	fr1, Z	fr1 = fr1 + NOT fr2.Bit
ADDB fr1, fr2.Bit	2	2	fr1, Z	fr1 = fr1 + fr2.Bit
CLC	1	1	C	C-Flag = 0
CLR !wdt	1	1	wdt, TO, PD	!wdt = 0, TO = 1, PD = 1 (1)
CLR fr	1	1	fr, Z	fr = 0
CLR w	1	1	w, Z	w = 0
CLZ	1	1	Z	Z = 0
DEC fr	1	1	fr, Z	fr = fr - 1
DECSZ fr	1	1/2	fr	fr = fr - 1, pc++, if fr = 0
DJNZ fr, Addr	2	2/4	fr, (pc)	fr = fr - 1, pc = Addr, if fr <> 0
IJNZ fr, Addr	2	2/4	fr, (pc)	fr = fr + 1, pc = Addr, if fr <> 0
INC fr	1	1	fr, Z	fr = fr + 1
INCSZ fr	1	1/2	fr, (pc)	fr = fr + 1, pc++, if fr = 0
RL fr	1	1	fr, C	fr.(7..1) = fr.(6..0), fr.0 = C, C = fr.7
RR fr	1	1	fr, C	fr.(6..0) = fr.(7..1), fr.7 = C, C = fr.0
STC	1	1	C	C = 1
STZ	1	1	Z	Z = 1

SX Assembly Instructions by Functions				
Arithmetic and Shift Instructions				
Instruction	Words	Cycles	Changes	Operation
SUB fr, #Const	2	2	fr, w, C, DC, Z	fr = fr - Const (STC)
SUB fr, w	1	1	fr, C, DC, Z	fr = fr - 1 (STC)
SUB fr1, fr2	2	2	fr1, w, C, DC, Z	fr1 = fr1 - fr2 (STC)
SUBB fr1, /fr2.Bit	2	2	fr1, Z	fr1 = fr1 - NOT fr2.Bit (STC)
SUBB fr1, fr2.Bit	2	2	fr1, Z	fr1 = fr1 - fr2.Bit (STC)
SWAP fr	1	1	fr	fr.(7..4) = fr.(3..0), fr.(3..0) = fr.(7..4)

SX Assembly Instructions by Functions				
Bit Manipulation				
Instruction	Words	Cycles	Changes	Operation
ADDB fr1, fr2.Bit	2	2	fr1, Z	fr1 = fr1 + fr2.Bit
ADDB fr1, fr2.Bit	2	2	fr1, Z	fr1 = fr1 + fr2.Bit
CLRB fr.Bit	1	1	fr.Bit	fr.Bit = 0
MOVB fr1.Bit, /fr2.Bit	4	4	fr1.Bit	fr1.Bit = NOT fr2.Bit
MOVB fr1.Bit, fr2.Bit	4	4	fr1.Bit	fr1.Bit = fr2.Bit
SB fr.Bit	1	1/2	(pc)	pc++, if fr.Bit = 1
SETB fr.Bit	1	1	fr.Bit	fr.Bit = 1
SUBB fr1, /fr2.Bit	2	2	fr1, Z	fr1 = fr1 - NOT fr2.Bit (STC)
SUBB fr1, fr2.Bit	2	2	fr1, Z	fr1 = fr1 - fr2.Bit (STC)
SNB fr.Bit	1	1/2	(pc)	pc++, if fr.Bit = 0

SX Assembly Instructions by Functions				
MOV Instructions				
Instruction	Words	Cycles	Changes	Operation
MOV !option, #Const	2	2	Option, w	Option = Const
MOV !option, fr	2	2	Option, w, Z	Option = fr
MOV !option, w	1	1	Option	Option = w
MOV !port, #Const	2	2	!port, w	Port-Config. = Const
MOV !port, fr	2	2	!port, w, Z	Port-Config. = fr
MOV !port, w	1	1	!port	Port-Config. = w
MOV fr, #Const	2	2	fr, w	fr = Const
MOV fr, w	1	1	fr	fr = w
MOV fr1, fr2	2	2	fr1, w, Z	fr1 = fr2
MOV m, #Const	1	1	m	m = Const
MOV m, fr	2	2	m, w, Z	m = fr
MOV m, w	1	1	m	m = w
MOV w, #Const	1	1	w	w = Const
MOV w, fr	1	1	w, Z	w = fr
MOV w, /fr	1	1	w, Z	w = NOT fr
MOV w, ++fr	1	1	w, Z	w = fr + 1
MOV w, <<fr	1	1	w, C	w = RL fr

Programming the SX Microcontroller

SX Assembly Instructions by Functions				
MOV Instructions				
Instruction	Words	Cycles	Changes	Operation
MOV w, <>fr	1	1	w	w = SWAP fr
MOV w, >>fr	1	1	w, C	w = RR fr
MOV w, --fr	1	1	w, Z	w = fr - 1
MOV w, fr-w	1	1	w, C, DC, Z	w = fr - w (STC)
MOV w, m	1	1	w	w = m
MOVB fr1.Bit, /fr2.Bit	4	4	fr1.Bit	fr1.Bit = NOT fr2.Bit
MOVB fr1.Bit, fr2.Bit	4	4	fr1.Bit	fr1.Bit = fr2.Bit
MOVSZ w, ++fr	1	1	w	w = fr + 1, pc++, if w = 0
MOVSZ w, --fr	1	1	w	w = fr - 1, pc++, if w = 0

SX Assembly Instructions by Functions				
Program Flow				
Instruction	Words	Cycles	Changes	Operation
CALL Addr	1	3	pc	pc = Addr, Push (2)
CJA fr, #Const, Addr	4	4/6	w, C, DC, Z, (pc)	pc = Addr, if fr > Const (CLC)
CJA fr1, fr2, Addr	4	4/6	w, C, DC, Z, (pc)	pc = Addr, if fr1 > fr2 (STC)
CJAE fr, #Const, Addr	4	4/6	w, C, DC, Z, (pc)	pc = Addr, if fr >= Const (STC)
CJAE fr1, fr2, Addr	4	4/6	w, C, DC, Z, (pc)	pc = Addr, if fr1 >= fr2 (STC)
CJB fr, #Const, Addr	4	4/6	w, C, DC, Z, (pc)	pc = Addr, if fr < Const (STC)
CJB fr1, fr2, Addr	4	4/6	w, C, DC, Z, (pc)	pc = Addr, if fr1 < fr2 (STC)
CJBE fr, #Const, Addr	4	4/6	w, C, DC, Z, (pc)	pc = Addr, if fr <= Const (CLC)
CJBE fr1, fr2, Addr	4	4/6	w, C, DC, Z, (pc)	pc = Addr, if fr1 <= fr2 (STC)
CJE fr, #Const, Addr	4	4/6	w, C, DC, Z, (pc)	pc = Addr, if fr = Const (STC)
CJE fr1, fr2, Addr	4	4/6	w, C, DC, Z, (pc)	pc = Addr, if fr1 = fr2 (STC)
CJNE fr, #Const, Addr	4	4/6	w, C, DC, Z, (pc)	pc = Addr, if fr <> Const (STC)
CJNE fr1, fr2, Addr	4	4/6	w, C, DC, Z, (pc)	pc = Addr, if fr1 <> fr2 (STC)
CSA fr, #Const	3	3/6	w, C, DC, Z, (pc)	pc++, if fr > Const (CLC)
CSA fr1, fr2	3	3/6	w, C, DC, Z, (pc)	pc++, if fr1 > fr2 (STC)
CSAE fr, #Const	3	3/6	w, C, DC, Z, (pc)	pc++, if fr >= Const (STC)
CSAE fr1, fr2	3	3/6	w, C, DC, Z, (pc)	pc++, if fr1 >= fr2 (STC)
CSB fr, #Const	3	3/6	w, C, DC, Z, (pc)	pc++, if fr < Const (STC)
CSB fr1, fr2	3	3/6	w, C, DC, Z, (pc)	pc++, if fr1 < fr2 (STC)
CSBE fr, #Const	3	3/6	w, C, DC, Z, (pc)	pc++, if fr <= Const (CLC)
CSBE fr1, fr2	3	3/6	w, C, DC, Z, (pc)	pc++, if fr1 <= fr2 (STC)
CSE fr, #Const	3	3/6	w, C, DC, Z, (pc)	pc++, if fr = Const (STC)
CSE fr1, fr2	3	3/6	w, C, DC, Z, (pc)	pc++, if fr1 = fr2 (STC)
CSNE fr, #Const	3	3/6	w, C, DC, Z, (pc)	pc++, if fr <> Const (STC)
CSNE fr1, fr2	3	3/6	w, C, DC, Z, (pc)	pc++, if fr1 <> fr2 (STC)
DECSZ fr	1	1/3	fr	fr = fr - 1, pc++, if fr = 0
DJNZ fr, Addr	2	2/4	fr, (pc)	fr = fr - 1, pc = Addr, if fr <> 0
IJNZ fr, Addr	2	2/4	fr, (pc)	fr = fr + 1, pc = Addr, if fr <> 0
INCSZ fr	1	1/3	fr, (pc)	fr = fr + 1, pc++, if fr = 0
JB fr.Bit, Addr	2	2/4	(pc)	pc = Addr, if fr.Bit = 1
JC Addr	2	2/4	(pc)	pc = Addr, if C = 1

SX Assembly Instructions by Functions				
Program Flow				
Instruction	Words	Cycles	Changes	Operation
JMP Addr	1	3	pc	pc = Addr
JMP pc+w	1	3	pc, C, DC, Z	pc = Addr+w (CLC)
JMP w	1	3	pc	pc = w
JNB fr.Bit, Addr	2	2/4	pc	pc = Addr, if fr.Bit = 0
JNC Addr	2	2/4	pc	pc = Addr, if C = 0
JNZ Addr	2	2/4	pc	pc = Addr, if Z = 0
JZ Addr	2	2/4	pc	pc = Addr, if Z = 1
MOVSZ w, ++fr	1	1	w	w = fr + 1, pc++, if w = 0
MOVSZ w, --fr	1	1	w	w = fr - 1, pc++, if w = 0
PAGE Addr	1	1	PA2-0	PA2...PA0 = Addr.(11...9)
RET	1	3	pc	Pop (3)
RETI	1	3	pc, C, DC, Z	Pop, Restore (3) (4)
RETIW	1	3	pc, C, DC, Z	Pop, Restore, RTCC += w (3) (4)
RETP	1	3	pc	Pop, Restore Page (5)
RETW Const	1	3	pc	w = Const, Pop (3)
SB fr.Bit	1	1/2	(pc)	pc++, if fr.Bit = 1
SC	1	1/2	(pc)	pc++, if C = 1
SKIP	1	2	pc	pc++
SNB fr.Bit	1	1/2	(pc)	pc++, if fr.Bit = 0
C	1	1/2	(pc)	pc++, if C = 0
SNZ	1	1/2	(pc)	pc++, if Z = 0
SZ	1	1/2	(pc)	pc++, if Z = 1
TEST fr w	1	1	Z	Z = 1, if fr = 0

SX Assembly Instructions by Functions				
System Control				
Instruction	Words	Cycles	Changes	Operation
MODE Const	1	1	m	m.(3-0) = Const
BANK fr	1	1	Fsr	fr.(7-5) -> fsr.(7-5)
IREAD	1	4	w, m	(m:w) -> m:w
SLEEP	1	1	TO, PD	TO = 1, PD = 0, Stop clock

SX Assembly Instructions by Functions				
Miscellaneous Instructions				
Instruction	Words	Cycles	Changes	Operation
NOP	1	1	-	-

3.4 Special Registers

3.4.1 Option

OPTION									
7	6	5	4	3	2 ... 0				
RTW 1 = \$01: RTCC 0 = \$01: W	RTI RTCC Interrupt 1 = disabled 0 = enabled	RTS RTCC-Clock 1 = extern 0 = intern	RTE RTCC External Clock Edge 1 = negative 0 = positive	PSA Prescaler 1 = WDT 0 = RTCC	Prescaler Divide-by				
					3	2	1	RTCC	Watchdog
					0	0	0	1/2	1/1
					0	0	1	1/4	1/2
					0	1	0	1/8	1/4
					0	1	1	1/16	1/8
					1	0	0	1/32	1/16
					1	0	1	1/64	1/32
					1	1	0	1/128	1/64
					1	1	1	1/256	1/128

3.4.2 Status

STATUS								
7 ... 5				4	3	2	1	0
Program Memory Page Address				TO 0 = Watchdog Timeout 1 = Power Up, clr !wdt, sleep	PD 0 = sleep, 1 = Power Up, clr !wdt	Z Zero Flag	DC Digit Carry	C Carry
7	6	5	Page					
0	0	0	0 = \$000-\$1ff					
0	0	1	1 = \$200-\$3ff					
0	1	0	2 = \$400-\$5ff					
0	1	1	3 = \$600-\$7ff					
1	0	0	4 = \$800-\$9ff					
1	0	1	5 = \$a00-\$bff					
1	1	0	6 = \$c00-\$dff					
1	1	1	7 = \$e00-\$fff					

3.4.3 FSR

FSR			
7 ... 5			4 ... 0
Data Memory Bank Address			Register Address
7	6	5	Bank
0	0	0	0 = \$00-\$1f
0	0	1	1 = \$20-\$3f
0	1	0	2 = \$40-\$5f
0	1	1	3 = \$60-\$7f
1	0	0	4 = \$80-\$9f
1	0	1	5 = \$a0-\$bf
1	1	0	6 = \$c0-\$df
1	1	1	7 = \$e0-\$ff

3.5 Addressing the Port Control Registers

3.5.1 SX 18/20/28

Before writing to a port control register using a **mov !r?, w** instruction, the MODE (**m**) register must be loaded with a value in order to access the desired control register. The following table lists the allowed values for **m**, and which control registers are affected by a subsequent **mov !r?, w** instruction for SX 18/20/28 controllers.

MODE (m) Register and mov !r?, w (SX18/20/28)			
m	mov !ra, w	mov !rb, w	mov !rc, w
\$00			
\$01			
\$02			
\$03			
\$04			
\$05			
\$06			
\$07			
\$08		exchange CMP_B	
\$09		exchange WKPND_B	
\$0a		write WKED_B	
\$0b		write WKEN_B	
\$0c		write ST_B	write ST_C
\$0d	write LVL_A	write LVL_B	write LVL_C
\$0e	write PLP_A	write PLP_B	write PLP_C
\$0f	write TRIS_A	write TRIS_B	write TRIS_C

Programming the SX Microcontroller

3.5.2 SX 48/52

The table below lists the allowed values for *m*, and which control registers are affected by a subsequent **mov !r?, w** instruction for SX 48/52 controllers.

MODE (m) Register and mov !r?, w (SX48/52)					
m	mov !ra, w	mov !rb, w	mov !rc, w	mov !rd, w	mov !re, w
\$00		read T1CPL	read T2PL		
\$01		read T1CPH	read T2CPH		
\$02		read T1R2CML	read T2R2CML		
\$03		read T1R2CMH	read T2R2CMH		
\$04		read T1R1CML	read T2R1CML		
\$05		read T1R1CMH	read T2R1CMH		
\$06		read T1CNTB	read T2CNTB		
\$07		read T1CNTA	read T2CNTA		
\$08		exchange CMP_B			
\$09		exchange WKPND_B			
\$0a		write WKED_B			
\$0b		write WKEN_B			
\$0c		read ST_B	read ST_C	read ST_D	read ST_E
\$0d	read LVL_A	read LVL_B	read LVL_C	read LVL_D	read LVL_E
\$0e	read PLP_A	read PLP_B	read PLP_C	read PLP_D	read PLP_E
\$0f	read TRIS_A	read TRIS_B	read TRIS_C	read TRIS_D	read TRIS_E
\$10		clear Timer 1	clear Timer 2		
\$11					
\$12		write T1R2CML	write T2R2CML		
\$13		write T1R2CMH	write T2R2CMH		
\$14		write T1R1CML	write T2R1CML		
\$15		write T1R1CMH	write T2R1CMH		
\$16		write T1CNTB	write T2CNTB		
\$17		write T1CNTA	write T2CNTA		
\$18		exchange CMP_B			
\$19		exchange WKPND_B			
\$1a		write WKED_B			
\$1b		write WKEN_B			
\$1c		write ST_B	write ST_C	write ST_D	write ST_E
\$1d	write LVL_A	write LVL_B	write LVL_C	write LVL_D	write LVL_E
\$1e	write PLP_A	write PLP_B	write PLP_C	write PLP_D	write PLP_E
\$1f	write TRIS_A	write TRIS_B	write TRIS_C	write TRIS_D	write TRIS_E

These abbreviations are used for the timer registers:

T1CPH, T2CPH: Timer 1/2 capture (1), high byte

T1CPL, T2CPL: Timer 1/2 capture (1), low byte

T1R1CMH, T2R1CMH: Timer 1/2 register R1, high byte

T1R1CML, T2R1CML: Timer 1/2 register R1, low byte

T1R2CMH, T2R2CMH: Timer 1/2 register R2, high byte

T1R2CML, T2R2CML: Timer 1/2 register R2, low byte

3.6 Port Control Registers

3.6.1 TRIS (Direction)

TRIS_? (Direction) Register								
	7	6	5	4	3	2	1	0
A	1= Input 0= Output							
B	1= Input 0= Output							
C	1= Input 0= Output							
D	1= Input 0= Output							
E	1= Input 0= Output							

Dark gray: SX 52 only

Light gray: SX 48/52 only

3.6.2 LVL (Level Configuration)

LVL_? (Level Configuration) Register								
	7	6	5	4	3	2	1	0
A	1= TTL 0= CMOS							
B	1= TTL 0= CMOS							
C	1= TTL 0= CMOS							
D	1= TTL 0= CMOS							
E	1= TTL 0= CMOS							

Dark gray: SX 52 only

Light gray: SX 48/52 only

3.6.3 PLP (Pull-up Configuration)

PLP_? (Pull-up Configuration) Register								
	7	6	5	4	3	2	1	0
A	1 = inactive 0 = active							
B	1 = inactive 0 = active							
C	1 = inactive 0 = active							
D	1 = inactive 0 = active							
E	1 = inactive 0 = active							

Dark gray: SX 52 only

Light gray: SX 48/52 only

3.6.4 ST (Schmitt Trigger Configuration)

ST_? (Schmitt Trigger Configuration) Register								
	7	6	5	4	3	2	1	0
B	1 = disable 0 = enable							
C	1 = disable 0 = enable							
D	1 = disable 0 = enable							
E	1 = disable 0 = enable							

Dark gray: SX 52 only

Light gray: SX 48/52 only

3.6.5 WKEN_B (Wake Up Enable)

WKEN_B (Wake Up Enable) Register								
	7	6	5	4	3	2	1	0
B	1 = disable 0 = enable							

3.6.6 WKED_B (Wake Up Edge Configuration)

WKED_B (Wake Up Edge Configuration) Register								
	7	6	5	4	3	2	1	0
B	1 = neg. 0 = positive							

3.6.7 WKPND_B (Wake Up Pending Flags)

WKPND_B (Wake Up Pending Flags) Register								
	7	6	5	4	3	2	1	0
B	1 = Edge 0 = none							

mov !rb, w copies the contents in w to WKPND_B, and moves the previous contents of WKPND_B into w.

3.6.8 CMP_B (Comparator)

CMP_B (Comparator) Register								
	7	6	5	4	3	2	1	0
B	Comparator 1 = disable 0 = enable	Output 1 = disable 0 = enable						Result 1 = $U_{RB2} > U_{RB1}$ 0 = $U_{RB2} < U_{RB1}$

mov !rb, w copies the contents in w to CMP_B, and moves the previous contents of CMP_B into w.

3.6.9 T1CNTA (Timer 1 Control A) (SX 48/52 only)

T1CNTA Register							
7	6	5	4	3	2	1	0
T1CPF2	T1CPF1	T1CPIE	T1CMF2	T1CMF1	T1CMIE	T1OVF	T1OVIE
1 = Capture event at input 2 * 0 = no event	1 = Capture event at input 1 * 0 = no event	1 = Interrupt if T1CPFx = 1 0 = disable interrupt	1 = R2 and timer counter are equal* 0 = not equal	1 = R1 and timer counter are equal * 0 = not equal	1 = Interrupt if T1CMFx = 1 0 = disable interrupt	1 = Timer overflow 0 = no overflow	1 = Interrupt if overflow 0 = disable interrupt

* This bit remains set until cleared by the application.

3.6.10 T1CNTB (Timer 1 Control B) (SX 48/52 only)

T1CNTB-Register										
7	6	5	4 ... 2				1 ... 0			
RTCCOV	T1CPEDG	T1IEXEDG	T1PS2 ... 0 (Prescaler)				T1MC1 ... 0 (Mode)			
1 = RTCC overflow * 0 = no overflow	1 = positive edge at input 1 and 2 0 = negative edge	1 = positive edge at external clock 0 = negative edge	4	3	2	Divide-by		1	0	Mode
			0	0	0	1/1		0	0	Software Timer
			0	0	1	1/2		0	1	PWM
			0	1	0	1/4		1	0	Capture/Compare
			0	1	1	1/8		1	1	External event
			1	0	0	1/16				
			1	0	1	1/32				
			1	1	0	1/64				
1	1	1	1/128							

* This bit remains set until cleared by the application.

3.6.11 T2CNTA (Timer 2 Control A) (SX 48/52 only)

T2CNTA-Register							
7	6	5	4	3	2	1	0
T2CPF2	T2CPF1	T2CPIE	T2CMF2	T2CMF1	T2CMIE	T2OVF	T2OVIE
1 = Capture event at input 2 * 0 = no event	1 = Capture event at input 1 * 0 = no event	1 = Interrupt if T2CPF _x = 1 0 = disable interrupt	1 = R2 and timer counter are equal * 0 = not equal	1 = R2 and Timer are equal * 0 = not equal	1 = Interrupt if T2CMF _x = 1 0 = disable interrupt	1 = Timer counter overflow 0 = no overflow	1 = Interrupt if timer counter overflow 0 = disable interrupt

* This bit remains set until cleared by the application.

3.6.12 T2CNTB (Timer 2 Control B) (SX 48/52 only)

T2CNTB Register											
7	6	5	4 ... 2				1 ... 0				
PORTRD	T1CPEDG	T1HxEDG	T1PS2 ... 0 (Prescaler)				T1MC1 ... 0 (Mode)				
1 = Read port registers 0 = Read outputs	1 = positive edge at input 1 and 2 0 = negative edge	1 = positive edge at external clock 0 = negative edge	4	3	2	Divide-by		1	0	Mode	
			0	0	0	1/1		0	0	Software Timer	
			0	0	1	1/2		0	1	PWM	
			0	1	0	1/4		1	0	Capture/Compare	
			0	1	1	1/8		1	1	External Event	
			1	0	0	1/16					
			1	0	1	1/32					
			1	1	0	1/64					
1	1	1	1/128								

Programming the SX Microcontroller

A Complete Guide by Günther Daubach

2ND EDITION

Section IV - Applications

4 Section IV - Applications

This part of the book presents various application examples that have all been tested in “real life”. Nevertheless, these are not “the ultimate solutions” – they have been included to give you some ideas and hints for your own applications and projects.

Please note that neither the author nor the publisher can make any guarantee that the shown examples are free of errors, functionally suited for specific applications, or safe. If you plan to use any of these applications, or parts of the software presented here in life support systems, or under conditions where failure of the software would endanger the life or safety of the user, it is your own obligation to decide if the software or parts of it are suitable for such applications.

As an important source for ideas, ready-to-use Virtual Peripherals, and tutorial material, you should visit Ubicom’s Web Site frequently as it is regularly updated with new information, application notes, and links to other URLs that offer valuable information about the SX.

4.1 Function Generators with the SX

If you hook up a D/A converter to the SX’s port pins, digital values that are output to those pins can be converted into analogue signals.

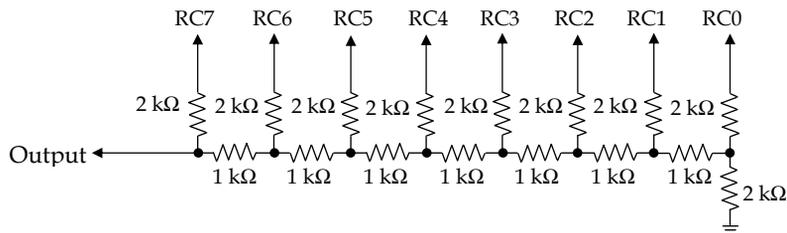
There are a great number of various DACs available in the market that can be used together with the SX, so you should be able to find the right parts quite easily.

Depending on what precision is asked for, 8-bit converters can be used, but an interesting alternative is the use of 12-bit converters. As you know, the SX is able to read data tables from its program memory using the **i read** instruction. After an **i read**, the lower 8 bits of the memory read can be found in **w**, and the higher 4 bits are stored in **m** ready to be sent to a 12-bit DAC connected to “1 ½” ports.

Other precisions as 16 bits are also possible, finally limited by the number of free port pins available.

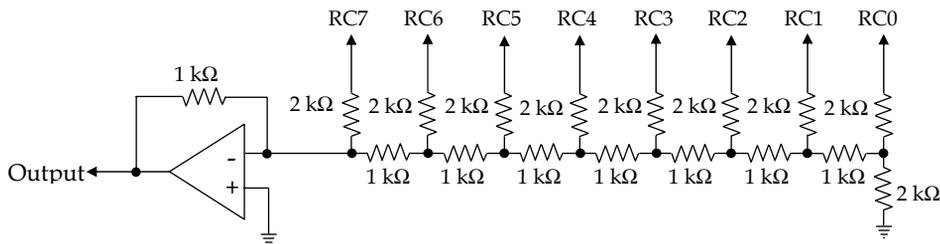
4.1.1 A Simple Digital-Analog Converter

For your first tries, hooking up a simple R-2R network to the SX is fine enough:



Please note that the output of this circuit cannot drive heavy loads, and that the linearity and precision mainly depends on the resistor tolerances.

If you “spend” an additional OP-amp, the output can drive higher loads without loss of linearity:



You can test the following program examples with each of the two DACs. In case you have a “better” one on hand, you could use it as well.

The sample programs are designed to drive the DAC through port C.

Depending on what signal you want to generate, it might be necessary to filter the output through a low pass (if you are using the OP amp version, you could add a capacitor parallel to the feedback resistor to turn the OP amp into a low pass filter. For the first tests, there is no need for filtering though.

4.1.2 A Ramp Generator

Generating a ramp signal is an easy task:

```

; =====
; Programming the SX Microcontroller
; APP001.SRC
; =====
include "Setup28.inc"

RESET    Mai n
org      $000

Mai n
  clr rc
  mov !rc, #0

Loop
  inc rc      ; 1
  jmp Loop   ; 3

```

This simple program configures all Port C pins as outputs, and keeps incrementing the Port C data register that periodically goes through the values from 0 to 255.

If you hook up an oscilloscope to the DAC output, you can check the output signal.

The program loop requires 4 clock cycles, i.e. 80 ns (at 50 MHz clock frequency). Therefore, after $256 * 80$ ns rc has counted from 0 to 255, resulting in a frequency of approximately 49 kHz for the ramp signal.

4.1.2.1 A Ramp Generator With a Pre-defined Frequency

In most cases, a frequency of 49 kHz is not what you really want, therefore, this simple program requires some enhancements in order to generate a pre-defined frequency.

The first idea would be to add some delay loops inside the main loop to reduce the frequency, but we should keep in mind that one of the real strengths of the SX controller is its capability of precisely timed interrupts that allow for time-controlled Virtual Peripherals. Therefore, let's make use of this feature to come to a general-purpose solution.

Before starting the editor to type in the new program, some planning, and calculations are a good time investment, so let's first define the specs for the Virtual Peripheral:

- Generate a ramp signal as a "background task"
- Desired Frequency, e.g. 1 kHz

Programming the SX Microcontroller

- Interface to the mainline program: Bit 0 in variable **Flags** is used to control the generator: set = generator active, clear = generator disable

In the next step, it is necessary to figure out the correct timing. A frequency of 1 kHz means a period of 1 ms, i.e. **rc** must go through the values from 0...255 within one millisecond, and this means that **rc** must be incremented every $1\text{ ms}/256 \approx 3.9\text{ }\mu\text{s}$. If we assume a system clock of 50 MHz, each clock cycle takes 20 ns, and if we calculate $3.9\text{ }\mu\text{s}/20\text{ ns}$, we end up in 195. This means **rc** must be incremented every 196 clock cycles.

When we reverse this calculation to find out the generated frequency, the result is $1/(195 * 256 * 20\text{ ns}) = 998.4\text{ Hz}$. The reason for the difference from the expected 1 kHz is caused by some rounding errors we have made.

If we accept that the generated signal amplitude is less than V_{DD} , we can easily generate exactly 1 kHz by reducing the maximum value in **rc** from 255 to 249, i.e. we only do 250 **rc** increments per signal period, and do an increment every 200 clock cycles. The resulting frequency is now $1/(200 * 250 * 20\text{ ns}) = 1\text{ kHz}$.

Now let's write the program:

```

; =====
; Programming the SX Microcontroller
; APP002.SRC
; =====
include      "Setup28.inc"

id          ' RampGen'
reset       Main

            org   $08

flags0     ds    1
flags1     ds    1
local Temp0 ds    1
local Temp1 ds    1
local Temp2 ds    1
isrTemp    ds    1

dacEnable  equ   flags0.0

toggleDac  macro
xor        flags0,  #00000001
endm

            org   $10
bank0      =     $
dacOutVal  ds    1

; *****
; Port Assignment
```

```

; *****
RC_latch    equ    %00000000    ; Port C latch init
RC_DIR     equ    %00000000    ; Port C DIR value

; *****
; Port and pin definitions
; *****

DAC_PORT    equ    rc

; *****
; Program constants
; *****

int_period  equ    200    ; RTCC Interrupt rate
DIR_W      equ    $0F    ; Write Port Direction

INTERRUPT_ORG    equ    $000
MAIN_ORG         equ    $1fb
MAIN_PROGRAM_ORG equ    $600

    org    INTERRUPT_ORG

; *****
ISR
; *****

    bank dacOutVal
    snb   dacEnable
    jmp   :Continue
    clr   dacOutVal
    jmp   :dacOut
:Continue
    inc   dacOutVal
    cjb   dacOutVal, #250, :dacOut
    clr   dacOutVal
:dacOut
    mov   DAC_PORT, dacOutVal

isrOut
    mov   w, #-int_period
    reti

    org   MAIN_ORG

Main
    page  _Main
    jmp   _Main

    org   MAIN_PROGRAM_ORG

_Main

```

Programming the SX Microcontroller

```
mov w, #RC_latch ; Initialize RC data latch
mov rc, w
mode DDIR_W      ; point MODE to write DDIR register
mov w, #RC_DDIR  ; Setup RC Direction register
mov !rc, w

include "Clr2x.inc"

RTCC_ON          = %10000000      ; Enables RTCC at address $01 (RTW hi)
                  ; *WREG at address $01 (RTW lo) by
                  ; default
RTCC_PS_OFF     = %00001000      ; Assigns prescaler to RTCC (PSA lo)
PS_111          = %00000111      ; RTCC = 1:256, WDT = 1:128

OPTIONSETUP     equ RTCC_ON | RTCC_PS_OFF | PS_111

mov !option, #OPTIONSETUP
bank local Temp0
setb dacEnable
mainLoop
decsz local Temp0
jmp mainLoop
decsz local Temp1
jmp mainLoop
decsz local Temp2
jmp mainLoop
toggleDac
jmp mainLoop

END
```

In this program, we do not increment **rc** directly, but use a separate variable **dacOutVal**. This is because we need to test if the increment has exceeded the maximum value, i.e. if the variable content has reached 250. In this case, we reset **dacOutVal** to 0. If we would reset **rc** directly, the value 250 would be sent to the output lines for some clock cycles before it goes to 0. This is not a big deal here, but in general, it is a good idea to avoid “spikes” on the output lines,

At the beginning of the ISR, we check if the **dacEnable** bit is set. If this is not the case, we force **dacOutVal** to 0, and skip the instructions that are responsible for ramp generation. Forcing **dacOutVal** to 0 is important to avoid that the DAC output signal remains “stuck” at some arbitrary level when the generator is turned off, depending on the contents of **dacOutVal** at that time.

At the end of the ISR, **dacOutVal** is copied into **rc**, to send the current output value to the DAC, and we return from the interrupt with **w** initialized to -200 to make sure that the ISR gets called again after 200 clock cycles.

As the mainline program has nothing else to do, we keep it busy by counting through a 3-level delay loop, and toggle the generator on/off flag when the delay has ended, just to demonstrate that the on/off toggle works as expected.

4.1.3 Generating a Triangular Waveform

The next step from a ramp to a triangular waveform is not too difficult. All we have to do is continuously increment the DAC value from 0 to a maximum, and back to 0 again.

```

; =====
; Programming the SX Microcontroller
; APP003.SRC
; =====
include      "Setup28.inc"
id          'TriGen'
reset       Main

          org     $10
flags0     ds     1
flags1     ds     1
localTemp0 ds     1
localTemp1 ds     1
localTemp2 ds     1
isrTemp    ds     1

dacSlope   equ    flags0.0

bank0      org     $10
           =       $

dacOutVal  ds     1

; *****
; Port Assignment
; *****

RC_latch   equ    %00000000           ; Port C latch init
RC_DIR     equ    %00000000           ; Port C DIR value

; *****
; Port and pin definitions
; *****

DAC_PORT   equ    rc

; *****
; Program constants
; *****

int_period equ    100                 ; RTCC Interrupt rate
DIR_W      equ    $0F                 ; Write Port Direction

          org     $000

; *****

```

Programming the SX Microcontroller

```
ISR
; *****
    snb  dacSlope          ; If Flags.0 set, negative slope
       jmp :Down
    inc  dacOutVal        ; Generate positive slope
    cjb  dacOutVal, #250, isrOut ; Continue if less than 250,
    dec  dacOutVal        ; else reset to 249
    setb dacSlope        ; Set flag for negative slope
:Down
    dec  dacOutVal        ; Generate negative slope
    sz   ; Continue is greater than 0,
       jmp isrOut        ; else
    clrb dacSlope        ; clear the slope direction flag

isrOut
    mov  w, #-int_peri od
    reti w

Main
    org  $1fb
    page _Main
    jmp  _Main

_Main
    org  $600
    mov  w, #RC_latch    ; Initialize RC data latch
    mov  rc, w
    mode DDIR_W          ; point MODE to write DDIR register
    mov  w, #RC_DDIR    ; Setup RC Direction register
    mov  !rc, w

include "Clr2x.inc"

RTCC_ON      = %10000000    ; Enables RTCC at address $01 (RTW high)
              ; *WREG at address $01 (RTW low) by
              ; default
RTCC_PS_OFF  = %00001000    ; Assigns prescaler to RTCC (PSA low)
PS_111       = %00000111    ; RTCC = 1:256, WDT = 1:128

OPTIONSETUP  equ RTCC_ON | RTCC_PS_OFF | PS_111

    mov  !option, #OPTIONSETUP
    bank local Temp0
mainLoop
    jmp  mainLoop
END
```

The Triangle VP makes use of Flags.0 as the “Slope Direction Flag” to keep track of the slope that is currently generated. If the flag is set, a negative slope is generated (i.e. **dacOutVal** is decremented). Otherwise, **dacOutVal** is incremented, resulting in a positive slope.

Please note how the change between the slopes is performed. If while generating the positive slope, **dacOutVal** has reached 250, the slope direction must be reversed. As **rc** already outputs 249, the next value to be sent to **rc** is 248. Therefore, **dacOutVal** is decremented from 250 to 249, and the instructions for the negative slope are executed before leaving the ISR, thus **dacOutVal** is decremented once more before its contents (248) is copied to **rc**.

If **dacOutVal** reaches 0 during the negative slope generation, the **dacSlope** flag is cleared. At this time, **rc** outputs 0, i.e. the next time the ISR is called, **rc** must be set to one. This is the case because **dacOutVal** is incremented at the next interrupt before it is copied to **rc**.

As it takes 250 steps to generate the positive slope, and another 250 steps to generate the negative slope, it takes 500 steps to generate one complete period of the triangular signal. Therefore, it is necessary to call the ISR twice as often, i.e. every 100 clock cycles in order to maintain a frequency of 1 kHz.

4.1.4 Generating Non-linear Waveforms

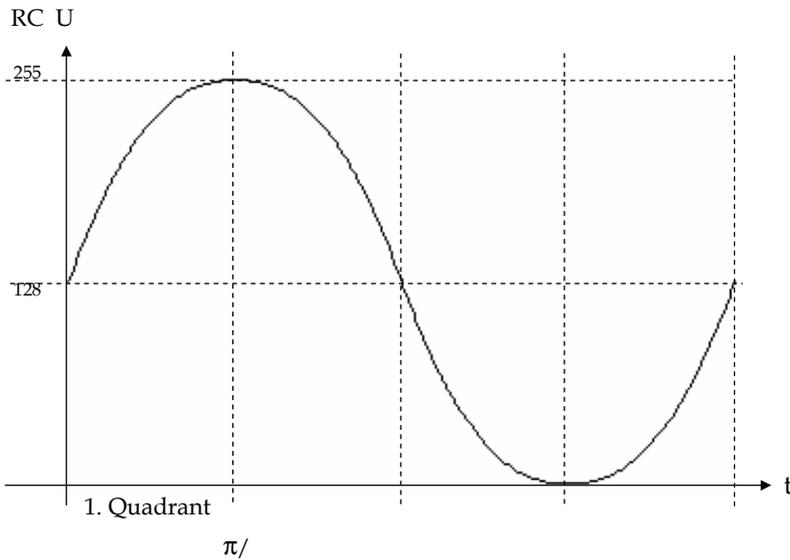
The approach to generate non-linear waveforms by calculating $f(t)$ in real-time is not a practical solution in many cases because the available arithmetical instructions of the SX are not powerful enough to perform the necessary calculations fast enough.

Using a table to store enough points of the required waveform is a good alternative because table-lookups can be performed very fast.

Let's pick up the example shown in the tutorial section of this book, and use it to generate a sine wave. For the first approach, we use a table that contains points for the sine function in the range from 0 up to 2π .

Programming the SX Microcontroller

4.1.4.1 Sine Wave



```

; =====
; Programming the SX Microcontroller
; APP004.SRC
; =====
include "Setup28.inc"

id      'SinGen1'
reset   Main

        org $08

dacIndex ds1

; *****
; Port Assignment
; *****

RC_latch equ %00000000    ; Port C latch init
RC_DIR   equ %00000000    ; Port C DIR value

; *****
; Pin Definitions
; *****
DAC_PORT equ rc

; *****

```

```

; Program constants
; *****
int_period equ 195 ; RTCC Interrupt rate
DDIR_W equ $0F ; Write Port Direction

ISR
    org $000
    mov w, dacIndex ; Load parameter for WtoSin
    call WtoSin ; Subroutine returns f(w) in w
    page ISR ; Adjust the page
    mov DAC_PORT, w ; Output the value
    inc dacIndex ; Next Index

ISRout
    mov w, #-int_period
    reti w

WtoSin
    page SinTable
    jmp w

    org $1fb

Main
    page _Main
    jmp _Main

; *****
;
; org $400
; *****
SinTable
    retw 127, 130, 133, 136, 139, 143, 146, 149, 152, 155, 158, 161, 164, 167, 170, 173
    retw 176, 178, 181, 184, 187, 190, 192, 195, 198, 200, 203, 205, 208, 210, 212, 215
    retw 217, 219, 221, 223, 225, 227, 229, 231, 233, 234, 236, 238, 239, 240, 242, 243
    retw 244, 245, 247, 248, 249, 249, 250, 251, 252, 252, 253, 253, 253, 254, 254, 254
    retw 254, 254, 254, 254, 253, 253, 253, 252, 252, 251, 250, 249, 249, 248, 247, 245
    retw 244, 243, 242, 240, 239, 238, 236, 234, 233, 231, 229, 227, 225, 223, 221, 219
    retw 217, 215, 212, 210, 208, 205, 203, 200, 198, 195, 192, 190, 187, 184, 181, 178
    retw 176, 173, 170, 167, 164, 161, 158, 155, 152, 149, 146, 143, 139, 136, 133, 130
    retw 127, 124, 121, 118, 115, 111, 108, 105, 102, 99, 96, 93, 90, 87, 84, 81
    retw 78, 76, 73, 70, 67, 64, 62, 59, 56, 54, 51, 49, 46, 44, 42, 39
    retw 37, 35, 33, 31, 29, 27, 25, 23, 21, 20, 18, 16, 15, 14, 12, 11
    retw 10, 9, 7, 6, 5, 5, 4, 3, 2, 2, 1, 1, 1, 0, 0, 0
    retw 0, 0, 0, 0, 1, 1, 1, 2, 2, 3, 4, 5, 5, 6, 7, 9
    retw 10, 11, 12, 14, 15, 16, 18, 20, 21, 23, 25, 27, 29, 31, 33, 35
    retw 37, 39, 42, 44, 46, 49, 51, 54, 56, 59, 62, 64, 67, 70, 73, 76
    retw 78, 81, 84, 87, 90, 93, 96, 99, 102, 105, 108, 111, 115, 118, 121, 124

; *****
;
; org $600
; *****
_Main

```

Programming the SX Microcontroller

```
    mov    w, #RC_latch          ; Initialize RC data latch
    mov    rc, w
    mode  DDIR_W                ; point MODE to write DDIR register
    mov    w, #RC_DDIR         ; Setup RC Direction register
    mov    !rc, w

include "Clr2x.inc"

RTCC_PS_OFF = %00001000        ; Assigns prescaler to RTCC (PSA 10)
PS_111     = %00000111        ; RTCC = 1:256, WDT = 1:128

OPTIONSETUP = RTCC_PS_OFF | PS_111

    mov    !option, #OPTIONSETUP
    jmp    @mainLoop

; *****
; MAIN PROGRAM CODE
; *****

mainLoop
    jmp    mainLoop
END
```

To generate a complete sine wave period, all 256 values in the table must be read in ascending order and sent to **rc**. We use **dacIndex** as an index into the table, and **dacIndex** is incremented at each ISR call. As the ISR is called after 195 clock cycles, we can calculate the frequency of the sine wave: $1/(195 * 256 * 20\text{ns}) = 1.0016 \text{ kHz}$.

4.1.4.2 Sine Generator with a Defined Frequency

If the tolerance of 1.6 Hz is not acceptable, we cannot simply reduce the maximum value for **dacIndex** to 249, and call the ISR every 200 clock cycles, as in the ramp generator sample above. In this case, the table would not be completely scanned, resulting in a partly sine wave. One method to increase the precision of the generated signal frequency is to reduce the table item count to 250 items. This requires a re-calculation of all the table values.

A helpful tool to calculate the values for such tables is a spreadsheet program like Microsoft's Excel (any other program will do the job as well, as long as it has the required functions available). Put the table index values into subsequent cells in one column, and let the spreadsheet program calculate the rounded function results into the cells of another column. Then use the export function of the program to save the contents of the result column to an ASCII file. You can then edit and re-format this file and finally paste it into the SX assembly source code file.

Here are the necessary modifications:

```
id    'Si nGen2'
```

```

int_period equ    200                ; Call the ISR every 4 μs

    org    $000
ISR
    mov    w, dacIndex
    call   WtoSin
    page   ISR
    mov    DAC_PORT, w
    inc    dacIndex
    cjb    dacIndex, #250, isrOut
    clr    dacIndex
isrOut
    mov    w, #-int_period
    retiw

;***** Table for sin(x/249) for 0 >= x/249 <= 2π *****
;
SinTable
retw 127, 130, 133, 137, 140, 143, 146, 149, 152, 155, 159, 162, 165, 168, 171, 174
retw 177, 180, 183, 185, 188, 191, 194, 196, 199, 202, 204, 207, 209, 212, 214, 216
retw 218, 221, 223, 225, 227, 229, 231, 232, 234, 236, 238, 239, 241, 242, 243, 244
retw 246, 247, 248, 249, 250, 250, 251, 252, 252, 253, 253, 254, 254, 254, 254, 254
retw 254, 254, 254, 253, 253, 252, 252, 251, 250, 250, 249, 248, 247, 246, 244, 243
retw 242, 241, 239, 238, 236, 234, 232, 231, 229, 227, 225, 223, 221, 218, 216, 214
retw 212, 209, 207, 204, 202, 199, 196, 194, 191, 188, 185, 183, 180, 177, 174, 171
retw 168, 165, 162, 159, 155, 152, 149, 146, 143, 140, 137, 133, 130, 127, 124, 121
retw 117, 114, 111, 108, 105, 102, 99, 95, 92, 89, 86, 83, 80, 77, 74, 71
retw 69, 66, 63, 60, 58, 55, 52, 50, 47, 45, 42, 40, 38, 36, 33, 31
retw 29, 27, 25, 23, 22, 20, 18, 16, 15, 13, 12, 11, 10, 8, 7, 6
retw 5, 4, 4, 3, 2, 2, 1, 1, 0, 0, 0, 0, 0, 0, 0
retw 1, 1, 2, 2, 3, 4, 4, 5, 6, 7, 8, 10, 11, 12, 13, 15
retw 16, 18, 20, 22, 23, 25, 27, 29, 31, 33, 36, 38, 40, 42, 45, 47
retw 50, 52, 55, 58, 60, 63, 66, 69, 71, 74, 77, 80, 83, 86, 89, 92
retw 95, 99, 102, 105, 108, 111, 114, 117, 121, 124

```

Using this ISR, and the modified table together with the sample program should result in a sine wave signal of exactly 1 kHz.

4.1.4.3 Superimposed Sine-Waves

Please look at this modified ISR:

```

id    'SinGen3'

int_period equ    200                ; Call the ISR every 4 μs

dacIndex ds 2                        ; Note: dacIndex now takes 2 bytes!
dacOut   ds 2                        ; New variable, 2 bytes

    org    $000
ISR
    mov    w, dacIndex                ; First index
    call   WtoSin                    ; Get the function value and

```

```
page ISR
mov  dacOut, w          ; save it
inc  dacIndex          ; Next table item
sb   dacIndex.0        ; If index is even, don't
    jmp :Output        ; touch the second index
mov  w, dacIndex+1     ; Second index
call WtoSin            ; Get the function value and
page ISR
mov  dacOut+1, w       ; save it
inc  dacIndex+1       ; Next table item
:Output
clc
rr   dacOut            ; First value / 2
clc
mov  w, >>dacOut+1    ; Second value / 2 -> w
add  dacOut, w         ; Add values and
mov  DAC_PORT, dacOut ; pass the sum to the outputs

isrOut
    mov w, #-int_period
    reti
```

This ISR contains two sine wave generators, where the second one works “at half speed” because it is only executed when the first generator’s index counter contains an odd value.

Starting at label **:Output**, the current values of both signals are divided by two, summed, and then sent to the output port.

Instead of dividing both function values by two, you could use a table where the amplitude is limited to 127 instead of 255.

4.1.4.4 Generating a Sine Wave from a 1st Quadrant Table

As a sine wave is symmetric in all four quadrants, it makes sense to use a table that only contains the function values for one quadrant. The values for the remaining three quadrants can be generated quite easily. In this example, we use a table with values for the 1st quadrant, limited to an amplitude of 127.

For the first and third quadrant, the table index value must be incremented from 0 to the highest table item, and for the second and fourth quadrant, the index must be decremented from the highest table item down to 0.

For the first and second quadrant, an offset of 128 is added to the table values to find the output value. For the third and fourth quadrant, the output value is calculated by subtracting the table value from the 128 offset.

```
; =====
; Programming the SX Microcontroller
; APP005.SRC
; =====
include "Setup28.inc"
```

```

RESET    Main

org      $08
Ix       ds    1
Flags   ds    1
Offset  ds    1

Q13     equ   Fl ags. 0
Q34     equ   Fl ags. 1

org      $000

;**** VP to generate a sine wave from a 1st quadrant table ****
;
; Output:      Signal is generated at Port C,
;              Values: 0..255
;
; Uses:       Ix, Flags, Offset
;
Sine1Q
  snb    Q13          ; If not 1st or 3rd quadrant, con-
    jmp  :Down        ; tinue at :Down (2nd/4th quadr.)
  inc    Ix           ; Next index
  sz     ;             ; If no overflow, continue and
    jmp  :Continue    ; read the table
  dec    Ix           ; Adjust Ix
  setb   Q13          ; Activate 2nd/4th quadrant
:Down    ;             ; Find index for 2nd/4th quadr.
  dec    Ix           ; Previous index
  sz     ;             ; If not 0, continue and
    jmp  :Continue    ; read the table
  clrb   Q13          ; De-select 2nd/4th quadrant
  xor    Fl ags, #00000010 ; Toggle 1st, 2nd and 3rd, 4th
                                ; quadrant
:Continue
  mov    w, Ix        ; Get table index
  call   WtoSin       ; Read function value
  page   Sine1Q       ; Adjust page
  sb     Q34          ; If not 3rd/4th quadrant,
    jmp  :Quadr1_2    ; go to 1st/2nd quadrant
  mov    w, Offset-w  ; Offset - f(Ix)
  jmp   :Output

:Quadr1_2 ;             ; 1st/2nd quadrant
  add    w, Offset    ; Offset + f(Ix)

:Output
  mov    rc, w        ; Send the result to rc
  mov    w, #-200     ; Call the ISR every 4 µs
  reti

; ** Subroutine to read the wave table ****
;
WtoSin

```

Programming the SX Microcontroller

```
page SinTable_1Q
jmp w

org $100

; ** Mainline program *****
;
; Main
clr rc ; Initialize Port C
clr Ix
clr Flags
mov Offset, #128
mov !rc, #0 ; All rc pins are outputs
mov !option, #%10011111 ; Enable RTCC interrupt

Loop ; Just loop...
jmp Loop

org $200

; ***** Table for sin(x/255) for 0 >= x/255 <= pi/2 *****
;
SinTable_1Q
retw 0, 1, 2, 2, 3, 4, 5, 6, 7, 8, 9, 9, 10, 11, 12
retw 12, 13, 14, 15, 16, 16, 17, 18, 19, 20, 21, 22, 23, 23, 24
retw 25, 26, 26, 27, 28, 29, 29, 30, 31, 32, 32, 33, 34, 35, 36, 36
retw 37, 38, 39, 39, 40, 41, 41, 42, 43, 44, 44, 45, 46, 47, 47, 48
retw 49, 50, 50, 51, 52, 52, 53, 54, 54, 55, 56, 57, 57, 58, 59, 59
retw 60, 61, 61, 62, 63, 64, 64, 65, 66, 66, 67, 68, 68, 69, 69, 70
retw 71, 71, 72, 73, 73, 74, 75, 75, 76, 77, 77, 78, 78, 79, 80, 80
retw 81, 81, 82, 83, 83, 84, 84, 85, 86, 86, 87, 87, 88, 88, 89, 90
retw 90, 91, 91, 92, 92, 93, 93, 94, 94, 95, 95, 96, 96, 97, 97, 98
retw 98, 99, 99, 100, 100, 101, 101, 102, 102, 103, 103, 104, 104, 105, 105, 105
retw 106, 106, 107, 107, 108, 108, 108, 109, 109, 110, 110, 110, 111, 111, 112, 112
retw 112, 113, 113, 113, 114, 114, 114, 115, 115, 115, 116, 116, 116, 117, 117, 117
retw 118, 118, 118, 118, 119, 119, 119, 120, 120, 120, 120, 121, 121, 121, 121, 121
retw 122, 122, 122, 122, 123, 123, 123, 123, 123, 124, 124, 124, 124, 124, 125
retw 125, 125, 125, 125, 125, 125, 125, 126, 126, 126, 126, 126, 126, 126, 126
retw 126, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127
```

The frequency of the resulting sine wave signal is $1/(200 * 4 * 256 * 20 \text{ ns}) \approx 244 \text{ Hz}$.

This sample program reads a table with 256 items containing the values for one quadrant of the sine function. This means that this table has a resolution factor of four compared to the table that represents a complete sine wave period of four quadrants.

If you don't need that resolution, you can reduce the number of table items in order to fine-tune the signal frequency. Try to reduce the table item count to just a few items, and add a low pass filter to the DAC output instead, and you will notice that "good quality" sine waves can be obtained with just a few table items.

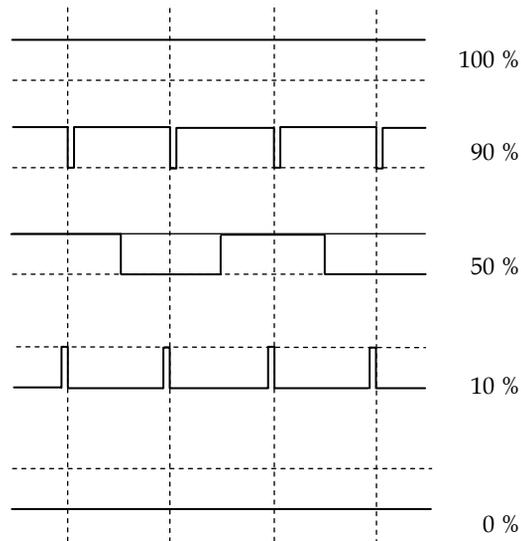
4.1.4.5 Generating Other Waveforms

By changing the contents of the data tables, the sample programs can be used to generate any other periodical waveform as well. If you need higher resolutions above 256 table items, you can use the **i read** instruction to read a table located in program memory.

4.2 Pulse Width Modulation (PWM) with the SX Controller

A square wave signal with varying duty-cycle can be used to control the brightness of LEDs, the speed of DC motors, the temperature of a heating element, and for many other purposes. If the PWM signal is fed through a low pass filter, it can also be used to build a Digital/Analog Converter.

The SX can only drive loads with low power consumption, like LEDs, but using the right driver allows to control whatever load is necessary. As the driver only needs to switch between the two stages “on” or “off”, the power dissipation is much less compared to analog drivers.



The diagram above shows the square wave signal at a PWM output for various duty cycles.

4.2.1 Simple PWM VP

The next program uses a simple algorithm to control the pulse width of the generated signal with the disadvantage that its frequency varies when the pulse width is changed, i.e. the output signal does not follow the waveform shown in the above diagram. (The algorithm is based upon a sample application published by Parallax, Inc.)

```

; =====
; Programming the SX Microcontroller
; APP006.SRC
; =====
include "Setup28.inc"
RESET    Main

org      $08
PwmAcc   ds 1           ; Current value for PWM
PwmVal   ds 1           ; Contents determines pulse width
rbBuff   ds 1           ; Buffer for Port B output data
Timer    ds 2           ; Delay counter (for demonstration
                        ; purposes)

org      $000

; ** PWM Virtual Peripheral *****

; This part of the ISR defines the current pulse width.
;
;
clr      rbBuff         ; Clear port data in advance
add      PwmAcc, PwmVal ; Set current PWM value
snc     ; Time to toggle the output ?
        setb rbBuff.0

; This part of the ISR is for demonstration purposes only, it con-
; tinuously modifies the contents of PwmVal, i.e. the pulse width.
;
;
:Change_PW
decsz   Timer          ; Decrement 1st timer
        jmp  :Exit      ; No underflow, so exit
decsz   Timer+1        ; Decrement 2nd timer
        jmp  :Exit      ; No underflow, so exit

        mov  Timer+1, #10 ; Re-initialize the 2nd timer
        dec  PwmVal      ; Decrease pulse width
;
; End of demo part

:Exit
        mov  rb, rbBuff  ; Output port data
        mov  w, #-200
        reti

org      $100

; ** Mainline program *****
;
Main
        mov  PwmVal, #$80
        mov  !rb, #%11111110
        mov  !option, #%10011111 ; Enable RTCC interrupt

```

Programming the SX Microcontroller

```
Loop                ; Keep looping...
  jmp Loop
```

To test the program, connect an LED between pin RB0 and V_{DD} via a current-limiting resistor (220 Ω is a good value).

While the program is running, the LED should change its brightness from dark to bright periodically.

The current pulse width is determined by the contents of **PwmVal**. The demo part of the ISR decrements **PwmVal** after a time delay and so its contents go through all possible values from 255 down to 0.

If you check the output signal with an oscilloscope, you will notice that the pulse width, and the frequency of the generated signal are changing over a large range.

This is caused by the fact that the time intervals between calls of the ISR, and an overflow of **PwmAcc** are not in a constant ratio.

4.2.2 PWM VP with constant Period

The following VP generates a PWM signal with a constant period and frequency and variable pulse width.

```
; =====
; Programming the SX Microcontroller
; APP007.SRC
; =====
include "Setup28.inc"
RESET  Main

org    $08
PwmAcc ds 1           ; Current value for pulse width
PwmVal ds 1           ; Contents determines pulse width
rbBuff ds 1           ; Buffer for Port B output
Timer  ds 1           ; Delay counter (for demonstration
                    ; purposes)
Incr    ds 1           ; Increment value for demonstration

PwmPin equ rbBuff.0   ; PWM bit in the buffer
Trigger equ rb.1      ; Trigger output for Oscilloscope

org    $000

; ** PWM VP (constant frequency) *****

; This part of the ISR does the PWM part
;
; setb Trigger          ; Set oscilloscope trigger
; setb PwmPin           ; Set PWM bit in advance
```

```

csb PwmAcc, PwmVal ; If PWM value reached,
clrb PwmPin ; clear PWM bit
inc PwmAcc ; Increment current value
mov w, ++PwmAcc ; Test if PwmAcc = 255
snz ; if so,
clr PwmAcc ; clear PwmAcc
sz
jmp :Exit

; This part of the ISR is for demonstration purposes only, it con-
; tinuously modifies the contents of PwmVal, i.e. the pulse width.
;
; Timers
decsz Timer ; Decrement timer
jmp :Exit ; If no underflow, continue
mov Timer, #15 ; Re-initialize the timer
add PwmVal, Incr ; Increment or decrement PwmVal
sz ; If PwmVal = 0 toggle increment/
; decrement
jmp :Exit ; decrement
sb Incr.7 ; If increment = -1,
jmp :Minus
inc Incr ; set it to
inc Incr ; +1
jmp :Exit
:Minus
dec Incr ; If increment = +1, set it
dec Incr ; to -1
dec PwmVal ; Adjust PwmVal from 0
dec PwmVal ; to 254
;
; End of demo part

:Exit
mov rb, rbBuff ; Output port data
clrb Trigger ; Clear oscilloscope trigger
mov w, #-195
reti w

org $100

; ** Mainline program *****
;
Main
mov PwmVal, #0
clr PwmAcc
clr Timer
clr Timer+1
mov Incr, #1
mov !rb, #%11111100
mov !option, #%10011111 ; Enable RTCC interrupt

Loop
jmp Loop

```

Programming the SX Microcontroller

This VP generates a positive edge at the PWM output after 255 ISR calls. The time difference between the positive and negative edges depends on the contents of **PwmVal**.

If **PwmVal** contains 0, the output remains continuously low, and if **PwmVal** contains 255, the output remains continuously high. To allow for the special case **PwmVal** = 255, **PwmAcc** is only incremented up to 254.

Based on the timing shown in the program the period of the generated signal is $195 * 255 * 20 \text{ ns} = 994,5 \mu\text{s}$, and its frequency is approximately 1 kHz (1,005.53 Hz).

The demo section in the ISR continuously changes the duty cycle from 0% up to 100%, and then from 100% down to 0%. You will notice the “smooth” brightness changes of the LED connected to RB0.

You may hook up an oscilloscope to monitor the PWM output signal at RB0. In order to get a steady display on the oscilloscope when the duty cycle reaches 0% or 100%, a trigger signal is provided at the RB1 output. Connect this pin to the external trigger input of the scope.

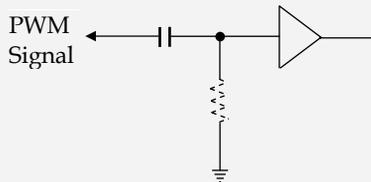
4.2.3 More Areas Where PWM is Useful

If you sent the PWM signal through a low pass filter, PWM can be used as a “low cost” DAC. Typical applications are the generation of DTMF signals for telephones, or FSK signals for modems, etc.

When you use PWM to control safety-critical systems, you must take care that the PWM output can never remain “stuck” at a level that keeps the driven item at full load (e.g. a motor running at full speed) when the program “hangs” by some reason.

Using the watchdog timer is one method to reset the system to a save state.

An additional safety measure is to feed the PWM signal into the driver through a capacitor that is large enough not to round the PWM signal slopes (see the diagram below). Now, if the PWM output remains “stuck” at high level, the level at the other end of the capacitor drops down to 0 according to the time-constant made up by the capacitor, and the input resistance of the driver. If the input resistance is relatively large, it may be necessary to add an extra resistor to reduce the time constant to an acceptable value.



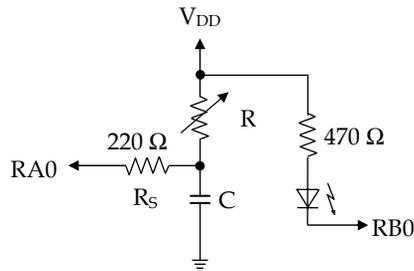
4.3 Analog-Digital Conversion with the SX

Although digital systems are all around, most of the natural parameters are still analogue, and this will obviously not change in the near future. In order to process such analog data with the SX an analog to digital conversion is required.

Due to its high speed, in many cases, the SX itself can be used to perform the necessary conversion without the need of an externally connected ADC.

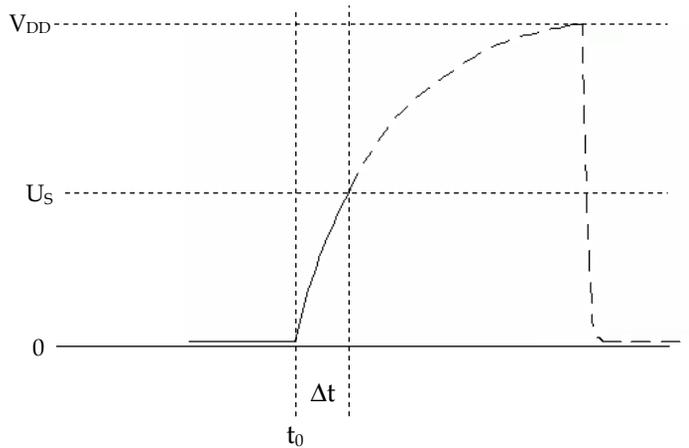
4.3.1 Reading a Potentiometer Setting

A simple solution to convert the value of a resistor into a digital value makes use of a simple RC network, as shown in the diagram below:



When the port pin is configured as a CMOS input, i.e. when it has high impedance, the capacitor will be charged through resistor R . After a certain time, the voltage across the capacitor C reaches the threshold level of $50\% V_{DD}$, so reading the port bit results in a change from 0 to 1.

If the port pin is configured as an output with low level, C is discharged across R_S relatively fast. R_S is used to limit the current the output must sink below $30\ \text{mA}$, a save value for R_S is $220\ \Omega$.



The diagram above shows the voltage across C as a function of time, where at t_0 the port pin is switched to high impedance. The voltage rises up to V_{DD} following an exponential function. Before the time interval Δt has elapsed, reading the port input returns 0, and after Δt the port bit is set to 1. Where Δt is a function of R and C, and if we assume that C has a constant value, Δt is almost proportional to R.

When the port pin changes to low level, C is discharged down to a small voltage relatively fast.

Before writing the software, we should make some calculations to find out the right timing, and the values for R and C.

If we assume that the analogue value shall be converted into a digital value in the range from 0...255, an 8-bit counter would have to be incremented exactly up to 255 within Δt . Special care should be taken that the counter does not overflow within that time, or the result would be useless.

When we also assume that the counter will be incremented within an ISR that is called every 200 clock cycles, i.e. every $4 \mu\text{s}$ (at 50 MHz system clock), counting from 0 up to 255 takes $4 \mu\text{s} * 256 \approx 1 \text{ ms}$ which is the value for Δt .

If we set the port input to CMOS, the threshold will be at 2.5 V (for $V_{DD} = 5 \text{ V}$).

When we use a potentiometer with a maximum value of 100 k Ω , we now need to calculate what capacitance is required to reach a voltage of 2.5 V after 1 ms. The formula is

$$C = \Delta t / (-\ln(1 - U_{\text{max}}/U_S) * R)$$

Using the values defined, the calculation is $1\text{ms} / (-\ln(1 - 2.5 / 5) * 100\text{k}\Omega) \approx 14.43\text{ nF}$, so a 12 nF capacitor is a good approximation.

Choosing a slightly smaller capacity adds an additional safety against counter overflow because Δt is reduced.

The demo program below shows how the ADC is implemented as Virtual Peripheral. We also have included the PWM VP in order to control the brightness of the LED connected to RB0.

```

; =====
; Programming the SX Microcontroller
; APP008.SRC
; =====
include "Setup28.inc"
RESET    Main

IsrPer   equ 200                ; The number of clock cycles
                                ; defines how often the ISR will
                                ; be called. Adjust this value
                                ; if necessary.

TRIS     equ $0f
LVL      equ $0d

org      $08
ADCVal   ds 1                   ; ADC Result
ADCStat  ds 1                   ; Status bits for ADC control
Timer    ds 1                   ; Time counter for the ADC
raMask   ds 1                   ; Current data for TRIS_A
PWMAcc   ds 1                   ; Counter for PWM

Charge   equ ADCStat.0          ; Mode Flag for ADC (0 = discharge,
                                ;                               1 = charge)
Trigger  equ ADCStat.1          ; Flag gets set during charge
                                ; when threshold is reached
ADCPin   equ ra.0               ; Port pin for ADC
PWMPin   equ rb.0               ; Port pin for PWM output

org      $000

; ** VP to read a potentiometer setting *****
;
; ADC

snb      Charge                 ; Jump to handle the current ADC
        jmp :Charge             ; mode

:Discharge
        incsz Timer             ; Discharge C during 256 ISR calls
        jmp :ADCExit           ; Still discharging, so
                                ; continue
        setb Charge            ; Change mode, and set
        xor raMask, #%00000001 ; port pin to
        mov !ra, raMask        ; input

```

Programming the SX Microcontroller

```
: Charge ; Conversion - charge C
  sb ADCPin ; When the voltage across C is
            ; less than the threshold,
            ; keep charging
    jmp :Continue
  snb Trigger ; If threshold flag already set,
            ; continue
    jmp :Continue
  setb Trigger ; Set the threshold flag, and
  mov ADCVal, Timer ; save the contents of counter
                    ; as result
: Continue
  incsz Timer ; Increment the counter and
  jmp :ADCExit ; continue if no overflow

  mov w, #$ff ; Initialize w to $FF
  sb Trigger ; If threshold flag was set during
            ; conversion, we have a good value
            ; otherwise, we set the result
            ; to the maximum
    mov ADCVal, w

  clr ADCStat ; Set mode to discharge, and clear
            ; the threshold flag
  xor raMask, #%00000001 ; Set RA.0 as output
  mov ra, raMask ; with low level to discharge
  mov !ra, raMask ; C

: ADCExit

; ** PWM VP, pulse width is controlled by the ADC result *****
;
; PWM
  setb PwmPin ; Set PWM bit in advance
  csb PwmAcc, ADCVal
  clrb PwmPin
  inc PwmAcc
  mov w, ++PwmAcc
  snz
  clr PwmAcc

  mov w, #-IsrPer
  reti w

org $100

; ** Mainline program *****
;
; Main
  clr ADCStat ; Initialize variables
  clr Timer
  mode LVL ; Set RA.0 to
  mov !ra, #%11111110 ; CMOS
  mode TRIS
```

```

mov  !rb,  #%11111110      ; rb.0 is PWM output
mov  raMask,  #%11111110  ; Initialize Port A mask and
clr  ra.0                ; set RA.0 to low to
                          ; discharge C and
mov  !ra,  raMask        ; set RA.0 as output

mov  !option,  #%10011111 ; Enable RTCC interrupt

Loop ; Nothing to do here for now...
  jmp Loop

```

Due to the initialization at the beginning of the mainline program, the ADC VP starts in the discharge mode, i.e. **ra.0** is output with low level, and it remains in that state for 256 ISR calls. Then the **Charge** flag gets set, and **ra.0** becomes an input.

At the next ISR call, the execution continues at **:Charge** because the **Charge** flag is set now. Here, **ra.0** is tested, and if the bit is set, the voltage across the capacitor has reached the input threshold. In this case, the content of **Timer** is copied to **ADCVal**, and the **Trigger** flag is set. This flag indicates that **ADCVal** already contains a value when the ISR is called the next times in order to avoid that **Timer** is copied to **ADCVal** again.

Each time the ISR is called, **Timer** is incremented until it overflows. If at that time, **Trigger** is not set, the voltage across C did not reach the input threshold, and this is an “out-of-range” situation. In this case, **ADCVal** is set to \$ff.

Finally, the flags **Charge** and **Trigger** are cleared, and **RA.0** becomes an output with low level for the discharge mode.

The next routine in the ISR is the PWM VP that we already described before. Here, the pulse width is controlled by **ADCVal** to change the brightness of an LED driven from the **rb.0** pin.

If you test this application, it might be necessary to either change the value of the capacitor, or the definition of **IsrPer** at the beginning of the program so that the full potentiometer range influences the LED’s brightness. This is caused by the tolerances of the components and the threshold of the SX inputs.

You will also note that – when the LED is quite dim – its brightness even changes when you don’t change the potentiometer position. As with all ADCs, at least the lowest digit is uncertain, and may therefore randomly toggle between 0 and 1 for each conversion.

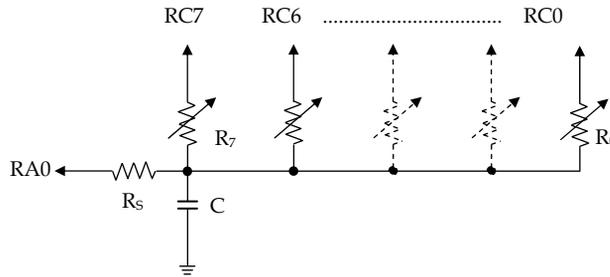
You can improve this relatively simple ADC by increasing the time constant of the RC network by giving C a greater capacity. This increases Δt , and you will need to use a 16-bit **Timer** counter. By rotating right the ADC result one or two bits, you can eliminate the uncertain lower bits.

Programming the SX Microcontroller

4.3.1.1 Reading more Potentiometer Settings

The previous example application can easily be extended to read up to 16 different potentiometer settings with an SX 28, having two port bits available to communicate with other units, e.g. via RS-232 or I²C.

One end of all potentiometers is connected to the capacitor, but the other ends of the potentiometers are not connected to V_{DD} but to separate port pins. These port pins are configured as inputs (hi-Z) by default, except the one connected to the potentiometer currently read. This pin is



configured as output set to high level.

The following program assumes that eight potentiometers are connected the Port C, and uses Port A.0 as read/discharge line.

```

; =====
; Programming the SX Microcontroller
; APP009. SRC
; =====
include "Setup28.inc"
RESET Main

IsrPer equ 200 ; The number of clock cycles
; defines how often the ISR will
; be called. Adjust this value
; if necessary.

TRIS equ $0f
LVL equ $0d
org $08
Std equ $
ADCVal ds 1 ; ADC Result
ADCStat ds 1 ; Status bits for ADC control
Timer ds 1 ; Time counter for the ADC
raMask ds 1 ; Current data for TRIS_A
pMask ds 1 ; Mask selects pot to read
PotId ds 1 ; Number of pots to read

Charge equ ADCStat. 0 ; Mode Flag for ADC (0 = discharge,
; 1 = charge)
```

```

Trigger equ ADCStat. 1          ; Flag gets set during charge
                                ; when threshold is reached
ADCGo   equ ADCStat. 3          ; ADC enabled when this bit is set
ADCPin  equ ra. 0                ; Port pin for ADC

org     $30
Pots    ds 8                      ; Storage for 8 pot readings

org     $000

; ** VP to read a potentiometer setting *****
;
ADC

sb      ADCGo                    ; Don't convert if ADCGo is
    jmp :ADCExit                 ; clear
snb     Charge                   ; Branch to the current
    jmp :Charge                  ; ADC mode

:DisCharge                        ; Discharge C during 256 ISR calls
    incsz Timer                  ; Still discharging, so
    jmp :ADCExit                 ; continue

    setb Charge                  ; Change mode, and set
    xor raMask, #%00000001       ; Port pin to
    mov !ra, raMask              ; input
    clrb ADCGo                   ; Stop the ADC

:Charge                            ; Conversion - charge C
    sb ADCPin                    ; When the voltage across C is
                                ; less than the threshold,
                                ; keep charging

    jmp :Continue                ;

    snb Trigger                  ; If threshold flag already set,
                                ; continue

    jmp :Continue                ;
    setb Trigger                  ; Set the threshold flag, and
                                ; save the contents of counter
    mov ADCVal, Timer            ; as result

:Continue                            ;
    incsz Timer                  ; Increment the counter and
    jmp :ADCExit                 ; continue if no overflow
    mov w, #$ff                  ; Initialize w to $FF
    sb Trigger                    ; If threshold flag was set during
                                ; conversion, we have a good value
                                ; otherwise, we set the result
                                ; to the maximum

    mov ADCVal, w

    clr ADCStat                  ; Set mode to discharge, and clear
                                ; the threshold flag
    mov !rc, #$ff                ; Turn off potentiometer
    xor raMask, #%00000001       ; Set RA.0 as output
    mov ra, raMask                ; with low level to discharge

```

Programming the SX Microcontroller

```
    mov !ra, raMask          ; C
:ADCExit
    mov  w, #-IsrPer
    retiw

; ** Insert the subroutine to send a changed pot reading here
;
SendValue
    ret

org    $100

; ** Mainline program *****
;
Main
    ; Clear the data memory
    ;
    clr fsr

include "Clr2x.inc"

mode LVL    ; Set RA.0 to
mov !ra, #%11111110          ; CMDS

mode TRIS
mov !rc, #%11111111          ; RC7...0 are inputs first
mov raMask, #%11111110      ; Initialize Port A mask and
mov !ra, raMask              ; set RA.0 to low to
clr b ra.0                   ; discharge C
clr PotId                    ; Pot Id = 0
mov pMask, #%11111110        ; Mask for pot 0
mov !option, #%10011111      ; Enable RTCC interrupt

:Loop
mov !rc, pMask                ; Configure one Port C pin as output
mov w, /pMask
mov rc, w
setb ADCGo                    ; Enable the ADC

:Wait                          ; Wait until ADC is ready
snb ADCGo
    jmp :Wait

    mov w, #Pots              ; Indirectly address the storage
    add w, PotId              ; for the last pot reading
    mov fsr, w
    cje ind, ADCVal, :SameValue ; If the reading has changed,
    mov ind, ADCVal          ; save new reading, and
    call SendValue          ; send the new reading (Value is
                            ; in w and in ADCVal)

:SameValue
inc PotId                    ; Select next potentiometer
```

```

clrb PotId. 3      ; Allow 0..7 only
stc                ; Rotate the pot mask
rl  pMask          ; to the left
sc                ; If the 0 has "arrived" in C,
  clrb pMask. 0    ; clear bit 0
jmp  :Loop         ; Next reading

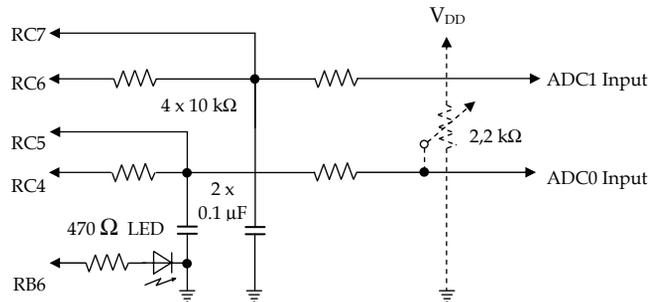
```

The ADC VP is almost identical to the previous example with the exception that it only does a conversion when the **ADCGo** flag is set, and the VP clears that flag when the conversion is finished. The mainline program can test that flag to test if a conversion is still in progress. After a conversion, the VP sets all Port C pins to hi-Z, i.e. all potentiometers are disconnected from V_{DD} while the capacitor is discharged.

Here, the mainline program takes control to select the potentiometer being read, to start the ADC, and to save the readings in the table **Pots**. If a potentiometer reading has changed since the last conversion, it calls **SendValue** subroutine. In this sample program, the routine is "empty" – in real-life applications, the routine could send information about the reading change to other modules via RS-232, for example. The new reading is stored in **w** and in **ADCValue** and the potentiometer id is stored in **PotId** when the subroutine is called.

4.3.2 A/D Converter Using Bitstream Continuous Calibration

The next ADC VP code was published by Ubicom and Parallax. It shows an interesting approach how to do A/D conversion, and it has a higher precision as the previously shown method. On the other hand, it requires two port pins per ADC instead of one. The diagram below shows the required components. (The SX-Key Demo Board has these components already in place, except the potentiometer):



For testing, you may connect the potentiometer to the ADC0 input, alternatively, you can feed a variable voltage (0...5 +V) into the ADC0 input.

Programming the SX Microcontroller

The program below contains two ADC VPs, and a PWM VP used to control the LED's brightness depending on the result of ADC0.

```
; =====
; Programming the SX Microcontroller
; APP010. SRC
; =====
include "Setup28.inc"
RESET    Main

TRIS     equ $0f
LVL      equ $0d
PWMPin   equ rb.6           ; Port pin for PWM output

IntPer   equ 163

org      $08
PWMAcc   ds 1               ; Counter for PWM

org      50h                ; Bank 2 variables
analog   equ $              ; ADC bank
port_buff ds 1              ; Buffer used by all ADCs
adc0     ds 1                ; ADC0 - Value
adc1     ds 1                ; ADC1 - Value
adc0_acc ds 1                ; ADC0 - Accumulator
adc1_acc ds 1                ; ADC1 - Accumulator
adc0_count ds 1             ; Time counter

org      $000

        bank analog         ; Select ADC bank

; ** VP for two A/D Converters *****
;
; ADC

        mov    w, >>rc      ; Read current state of ADCs
        not    w              ; Turn inputs to outputs
        and    w, #%01010000 ; Mask ADC1 and ADC0
        or     port_buff, w   ; Save new value in buffer
        mov    rc, port_buff  ; Refresh charge/discharge lines

        sb     port_buff.4    ; ADC0 above threshold ?
        incsz  adc0_acc       ; If so, increment accumulator,
        inc    adc0_acc       ; and avoid overflow by skipping
        dec    adc0_acc       ; the second inc instruction

        sb     port_buff.6    ; ADC0 above threshold ?
        incsz  adc1_acc       ; If so, increment accumulator,
        inc    adc1_acc       ; and avoid overflow by skipping
        dec    adc1_acc       ; the second inc instruction

        inc    adc0_count     ; Increment time counter
        jnz    :done_adcs     ; Continue if not yet done
```

```

mov   adc0, adc0_acc           ; Update ADC0 value
mov   adc1, adc1_acc           ; Update ADC1 value
clr   adc0_acc                 ; Clear ADC0 accumulator
clr   adc1_acc                 ; Clear ADC1 accumulator

:done_adcs

; ** PWM VP, controlled by ADC0 *****
;
; PWM
setb  PwmPin                   ; Set PWM bit in advance
csb   PwmAcc, adc0             ; If PWM value reached,
    clr  PwmPin                ;   clear PWM bit
inc   PwmAcc                   ; Increment current value
mov   w, ++PwmAcc              ; Test if PwmAcc = 255
snz   ; if so,
    clr  PwmAcc                ;   clear PwmAcc

clr   port_buff               ; Clear PWM port buffer

mov   w, #-IntPer              ; Call ISR every 'IntPer' cycles
reti w

org   $100

; ** The mainline program *****
Main

clr   rc                       ; Initialize Port C
mov   !rc, #%10101111          ; Configure Port C I/O pins
mov   !rb, #%10111111          ; Configure Port B output for LED
mode  LVL                      ; Set Port C inputs to
mov   !rc, #0                  ;   CMOS
mode  TRIS                     ; Restore MODE to direction

include "Clr2x.inc"

mov   !option, #%10011111      ; Enable RTCC interrupt

:loop
jmp   :loop

```

Port C pins 7 and 5 are configured as CMOS inputs, i.e. both inputs have a threshold level of about 2.5 V.

Port C pins 6 and 4 are configured as outputs. If an output is set to high level, the connected capacitor will be charged through the 10 k Ω resistor. When the output is low, the capacitor will be discharged through the same 10 k Ω resistor. Charging and discharging times depend on the input voltage that is fed through another 10 k Ω resistor.

Programming the SX Microcontroller

If the input voltage is high, the charging time is short, and the discharge time is long, and vice versa.

Inputs RC7 and RC 5 reverse status whenever the voltage across the capacitors reaches the input threshold of approximately 50% V_{DD} . By setting the levels at outputs RC6 and RC4 to high and low accordingly, the voltages across the capacitors “hover” around the threshold levels of the inputs RC7 and RC5.

The program determines the ratio between the capacitor charge and discharge times where this ratio depends on, and is proportional to the input voltage.

Each conversion takes 256 ISR calls, and the calls are counted in **adc0_count** for both ADCs. When this counter overflows, the conversion is finished, the contents of the variables **adc0_acc** and **adc1_acc** are copied to the result variables **adc0** und **adc1** and finally, **adc0_acc** and **adc1_acc** are cleared.

During a conversion, the following instructions are executed:

```
mov w, >>rc           ; Read current state of ADCs
not w                  ; Turn inputs to outputs
and w, #%01010000     ; Mask ADC1 and ADC0
or port_buff, w        ; Save new value in buffer
mov rc, port_buff      ; Refresh charge/discharge lines
```

At program start, **rc** was cleared, i.e. outputs RC6 and RC4 both are at low level, and the discharge phase is active.

When the previous instructions are executed, bits 6 and 4 of Port C are set now, i.e. the outputs are set to high level, and the charge phase begins.

These instructions are executed each time the ISR is called, but the state of the outputs at RC6 and RC4 don't change as long as the inputs at RC7 and RC5 are still low.

However, if – for example – the voltage at RC5 exceeds the threshold level, the following results are obtained:.

```
port_buff = 0000 0000
rc         = 0111 xxxx

mov w, >>rc      : w      = 0011 1xxx
not w           : w      = 1100 0xxx
and w, #%01010000 : w      = 0100 0000
or port_buff, w : port_buff = 0100 0000
mov rc, port_buff : rx     = 0100 0000
```

Caused by input RC5 that is set now, output bit RC4 is cleared now, so the discharge phase becomes active. Similarly, if RC7 would be set, the same result would occur at output RC6.

This sequence of instructions demonstrates how operations for several port bits can be performed with just a few instructions when the port assignment is done the right way. If you would hook

up the same RC network to Port C bits 3..0, the same sequence of instructions could control two additional ADCs.

After taking care of the output lines, the following instructions are executed for ADC0:

```

sb    port_buff.4          ; ADC0 above threshold ?
      incsz adc0_acc        ; If so, increment accumulator,
inc   adc0_acc            ; and avoid overflow by skipping
dec   adc0_acc            ; the second inc instruction sb

```

When input bit 5 was previously set (i.e. the voltage across C was greater than the input threshold), bit 4 in **port_buff** is cleared now, i.e. the instruction **incsz adc0_acc** will be executed as well as the following **inc** and **dec** instructions, i.e. in the end, **adc0_acc** will be incremented by one.

In the special case when **adc0_acc** contains \$ff, **incsz adc0_acc** changes its contents to \$00. This means that the **inc** instruction will be skipped, and only the **dec** instruction will be executed. The result in this case is that **adc0_acc** will not be changed at all in order to avoid an overflow. This is a very “clever” sequence of instructions to keep a register from overflowing.

The same sequence of instructions is then executed for ADC1.

In the last part of the ADC VP, the time counter **adc0_count** is incremented:

```

inc   adc0_count          ; Increment time counter
jnz   :done_adcs         ; Continue if not yet done

mov   adc0, adc0_acc      ; Update ADC0 value
mov   adc1, adc1_acc      ; Update ADC1 value
clr   adc0_acc           ; Clear ADC0 accumulator
clr   adc1_acc           ; Clear ADC1 accumulator

```

When it overflows after 256 ISR calls, the readings are saved, and the **adc0_acc** and **adc1_acc** registers are cleared for a new conversion.

The next VP in the ISR is the PWM VP that we already have described. Here, the VP is controlled from the **adc0** variable, and it is used to control the brightness of the LED connected to RB.6.

Some important points you should note:

For best precision, it is important that the ADC VP is executed after a constant number of clock periods. Therefore, other VPs with varying execution times must be executed after the ADC VP is finished. Therefore, it makes sense to place that VP at the very beginning of the ISR.

On the other hand, this VP has variable execution times that might influence time-critical VPs following the ADC VP in the ISR. You can find an alternate version of this ADC VP in Uvicom’s VP library that has a constant execution time.

Programming the SX Microcontroller

The impedance of the ADC inputs is determined by the $10\text{ k}\Omega$ resistors, where the other ends of these resistors are connected to a voltage of approximately $0.5 * V_{DD}$.

4.4 Timers as Virtual Peripherals

Almost every microcontroller application requires constant time intervals, and the SX controllers allow the “construction” of all kinds of timers easily as VPs, executed in the ISR.

4.4.1 A Clock Timer – an Example

The program below maintains several timers that are intended to control a digital clock.

```

; =====
; Programming the SX Microcontroller
; APP011. SRC
; =====
include "Setup28.inc"
RESET    Main

org      $08
Flags    ds 1

org      $50
Timers   equ $
us4      ds 1           ; 4us counter
Msec     ds 1           ; 1/1000 sec counter
HSec     ds 1           ; 1/100 sec counter
TSec     ds 1           ; 1/10 sec counter
Sec      ds 1           ; 1 sec counter
Sec10    ds 1           ; 10 sec counter
Min      ds 1           ; 1 min counter
Min10    ds 1           ; 10 min counter
Hour     ds 1           ; 1 hour counter
Hour10   ds 1           ; 10 hour counter

TickOn   MACRO
    setb Flags.0           ; Turn ticker on
ENDM

TickOff  MACRO
    clrb Flags.0           ; Turn ticker off
ENDM

SkipIfTick MACRO
    sb Flags.0           ; Skip if ticker is on
ENDM

org      $000

; ** Clock VP *****
;
Clock

    Bank Timers
    mov    w, #250           ; 4ms * 250 = 1 ms
    dec    us4

```

Programming the SX Microcontroller

```
snz
  mov us4, w
snz
  Tick0n
snz
  inc MSec                ; every millisecond
mov w, #10                ;
mov w, MSec-w            ; Z if MSec = 10      1
snz
  clr MSec
snz
  inc HSec                ; every 1/100 sec
mov w, #10                ;
mov w, HSec-w            ; Z if HSec = 10
snz
  clr HSec
snz
  inc TSec                ; every 1/10 sec
mov w, #10                ;
mov w, TSec-w            ; Z if TSec = 10
snz
  clr TSec
snz
  inc Sec                 ; every second
mov w, #10                ;
mov w, Sec-w             ; Z if sec = 10
snz
  clr Sec
snz
  inc Sec10               ; every 10 seconds
mov w, #6                 ;
mov w, Sec10-w           ; Z if Sec10 = 6
snz
  clr Sec10
snz
  inc Min                 ; every minute
mov w, #10                ;
mov w, Min-w             ; Z if Min = 10
snz
  clr Min
snz
  inc Min10               ; every 10 minutes
mov w, #6                 ;
mov w, Min10-w           ;
snz
  clr Min10
snz
  inc Hour                ; every hour
mov w, #10                ;
mov w, Hour-w            ;
snz
  clr Hour
snz
  inc Hour10              ; every 10 hours
mov w, #3
```

```

mov w, Hour10-w          ; Z if Hour10 = 3
snz                      ;
  clr Hour10             ; every day

mov w, #-200             ; Call ISR every 4us
reti w

org $100

; ** Main program *****
;
Main
include "Clr2x.inc"

mov !rb, #00110000       ; Set Port B outputs
mov !option, #10011111   ; Enable RTCC interrupt
bank TimerS              ; Select timer bank

: Loop

SkipIfTick               ; The timer sets the tick flag
  jmp :Loop              ; every Millisecond

; Output various timer bits for test purposes
;
movb rb.0, /Sec.0
movb rb.7, /Sec.0        ; Seconds tick to buzzer
movb rb.1, /Sec10.0
movb rb.2, /Min.0
movb rb.3, /Min10.0
movb rb.6, Hour.0       ; This LED has positive logic

TickOff                  ; Clear the tick flag
  jmp :Loop

```

For demonstration purposes, the mainline program drives various LEDs, and a small loud-speaker. Each of the LEDs hooked up to outputs RB0...RB3 is connected to V_{DD} via a current-limiting resistor, the LED at RB6 is connected to V_{SS} via a current-limiting resistor, and the loud-speaker is driven by RB7. If you use an SX-Key Demonstration board, these components are already in place on the board.

If you use another prototyping system you might have to change the mainline program in order to correctly control the LEDs and the buzzer.

The Timer VP shown here, does not use any **jmp** instructions, but conditional skip instructions only. Therefore, the execution time of this VP is always constant, i.e. it can be located "in front" of other time-critical VPs in the ISR code.

Programming the SX Microcontroller

The ISR is called every 200 clock cycles, i.e. every $200 * 20 \text{ ns} = 4 \mu\text{s}$.

When you multiply this time with a factor of 250, the resulting time interval is 1 ms.

```
mov w, #250          ; 4ms * 250 = 1 ms
dec us4
snz
  mov us4, w
snz
  Tick0n
snz
  inc Msec           ; every millisecond
```

To count this interval, **us4** is used and decremented at each invocation of the ISR. When its contents reaches 0, the **mov us4, w** resets it to 250, and because this instruction does not change the zero flag, the **Tick0n** macro, and the **inc Msec** instruction are only executed when **us4** was zero before.

Tick0n is a macro that sets the “Tick Flag”. The mainline program may check that flag to execute an action every millisecond, e.g. refreshing a display, and the mainline program must reset the tick flag again.

Msec is the counter for the millisecond intervals. The instructions

```
mov w, #10
mov w, MSec-w       ; Z if MSec = 10
snz
  clr Msec
snz
  inc Hsec          ; every 1/100 sec
```

take care that **Msec** is reset when its contents is greater than 9, and in this case **Hsec**, the counter for 1/100 seconds is incremented.

Similarly, the counters for the other time intervals up to the 10-hours counter are updated.

The contents of these counters hold the time information that is required to control the digits of a digital clock, and there is no need to make any binary to decimal conversions.

We will use this Timer VP in two other chapters to build a stopwatch and a digital alarm clock.

While the program is running, you can watch the LEDs to see how the contents of the seconds, minutes and the one-hour counters change (the LEDs each display the contents of the lowest bit of the assigned counter) and the buzzer generates a sound similar to a mechanical clock.

The mainline program tests the tick flag and updates the LED display every microsecond only.

4.4.2 General Timer VPs und Timed Actions

Based upon the example above, you can derive timers for other time intervals quite easily, but sometimes, it requires some clever combination of divide ratios, and ISR calling intervals to end up with the required time especially when the ISR calling intervals are “dictated” by other VPs in the ISR.

In most cases, it is necessary to perform a certain action if the time interval has elapsed. There are several possibilities how and where to perform that action:

4.4.2.1 Execution within the ISR

The instructions

```
decsz Timer
  jmp :Continue
mov Timer, #InitValue
call Action
:Continue
```

decrement the timer, and when its contents reaches 0, it is reset to the initial value, and a subroutine is called that performs the required action (instead of calling the subroutine, you might consider to add the necessary instructions directly).

Here, you must keep in mind that the instructions that make up the subroutine, or the instructions inserted directly, “eat up” available ISR clock cycles, i.e. additional clock cycles are “stolen” from the mainline program, and you must make sure that the maximum number of possible ISR clock cycles is not exceeded.

Performing the action directly from within the ISR also makes it difficult to maintain a constant execution time whether the action is taken or not, and therefore no other time-critical VPs can follow this part of the code in the ISR.

4.4.2.2 Testing the Timers in the Mainline Program

When the mainline program executes a main loop often enough, the timer can be checked within this loop. When it reaches a specific value, 0 for example, the necessary action can be performed from the mainline program.

Here, it is important to make sure that a timer overflow is detected in the mainline program in any case. If the code in the ISR looks like this

```
decsz Timer
  jmp :Continue
mov Timer, #InitValue
:Continue
```

it does not make sense to do a test in the mainline program like this:

Programming the SX Microcontroller

```
: Loop
  test Timer
  sz : Loop
  call Action
  ;
  ; mode instructions
  ;
  jmp : Loop
```

As the overflow is detected by the ISR, and the timer is assigned its initial value in this case, the mainline program will never “see” the zero value in **Timer**.

This test is an improvement:

```
: Loop
  mov w, #InitValue
  mov w, Timer-w
  sz : Continue
  call Action
  ;
  ; more instructions
  ;
  jmp : Loop
```

Here, the mainline program tests if **Timer** contains the initial value, and performs the action in this case, but this method can cause problems as well.

If the mainline program does not execute the main loop fast enough, it may happen that the ISR has decremented **Timer** from the initial value down to the next value before the mainline program has “seen” the initial value at all.

On the other hand, if the execution of the action is fast enough that **Timer** still holds the initial value at the next run through the main **loop**, so the action would be executed again, although no new timer overflow happened in the meantime.

When the Timer VP sets a flag like in

```
decsz Timer
  jmp : Continue
mov Timer, #InitValue
setb Timeout          ; Set timeout flag
: Continue
```

and the mainline program checks this flag, and resets it like in

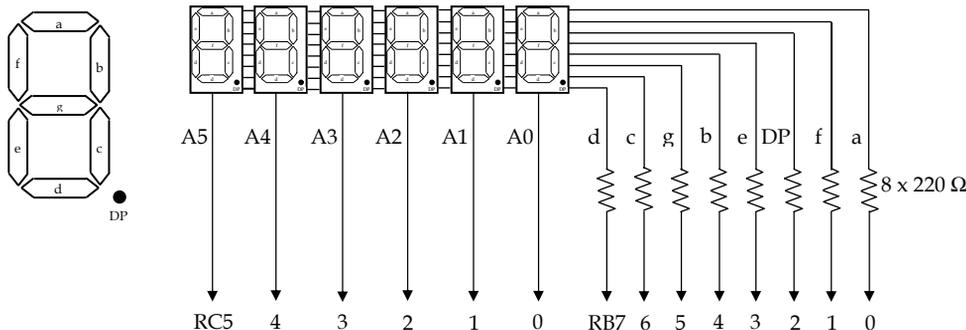
```
: Loop
  sb Timeout          ; Check the timeout flag
  jmp : Continue
  call Action
  clrb Timeout
: Continue
;
```

```
; more instructions  
;  
jmp :Loop
```

the problem to avoid multiple calls of the action is solved. In the Clock Timer example, you already saw the use of the “Tick Flag” that is used in the mainline program to trigger the action of updating the LED display and the buzzer output.

4.5 Controlling 7-Segment LED Displays

The diagram, below left, shows a usual 7-segment LED display unit with an additional LED for the decimal point. The diagram, below right, shows how six display units with common anode can be connected to the SX controller.



In this diagram, the eight cathode rows are connected to the port pins RB7...0 across the 220 Ω resistors, and the six separate anodes are connected to port pins RC5...0.

In other words, the segment LEDs and the decimal point LEDs are located in a matrix where the cathodes are connected to the rows, and the anodes to the columns. Actually, it does not matter which cathode row is connected to what port pin as a table in the software makes the necessary assignments. The connections shown here are just an example.

To turn on a segment at a certain matrix position, the column line is set to high, and the row line to low level. For example, to turn on segment g in the leftmost display position, set RC5 to high, and RB5 to low level.

To control the matrix, each of the column lines is set to high level periodically to select a display digit, and each time, the row lines of the segments to be turned on are set to low level.

Each column line should be activated at least 100 times per second to avoid a flickering display.

When you determine the value for the current-limiting resistors, you must consider the worst case that all eight LEDs in a digit might be on, and the port output for the column must source the total current for eight LEDs.

With the 7-segment units used to build the prototype for this book, with 220 Ω resistors, the current per LED was limited to 14.7 mA. This means a total current of approximately 118 mA at worst case.

Because each RC output is turned on during 1/6th of each scan, the effective current is less than 20 mA.

The SX data sheet specifies a maximum current of 30 mA per output, but it does not specify if higher peak currents are allowed. While testing this application and other applications, it never happened that an output was damaged due to the current peaks. Nevertheless, if higher currents are to be handled, you might consider adding transistors or special driver components to increase the possible current load.

When you test the application in single step mode, it can happen that the program stops at a position where all LEDs in a 7-segment unit remain turned on. In this case, the port output must source the current for all 8 LEDs for a longer time.

In the prototype, the total current in this case was 48 mA although the multiplication of 8 * 14.7 mA yields in about 118 mA. Obviously did the port output limit the current to the value of 48 mA, but for a “long life” of your SX controller, you should avoid such situations. If you are running the program in Debug “Run” mode, you should do a “Reset” rather than a “Stop” to hold the program.

In “real life” it makes sense to activate the watchdog timer to avoid, that the program remains “stuck” with LEDs turned on.

The following program is an example how to control the 7-segment display matrix:

```

; =====
; Programming the SX Microcontroller
; APP012. SRC
; =====
include "Setup28.inc"
RESET   Main

org     $08
Std     = $
Counter ds 3           ; Counter for time delay in the
                       ; mainline program (for demo
                       ; only)

org     $30
Leds    = $           ; The LED bank
Column ds 1           ; Column mask
Digit   ds 1           ; Current display digit
Digits  ds 6           ; Digits buffer

org     $50
Timers  = $
Millsec ds 1           ; Counter for Milliseconds

org     $000
; ** VP to drive a 7-segment LED display matrix with 6 digits *****

```

Programming the SX Microcontroller

```
;
Seg_7
bank Timers
mov w, #250 ; Initialization value in w
dec Millsec ; When 1 ms has elapsed,
snz ; re-initialize the counter
mov Millsec, w ; and
sz ; refresh the display, else
jmp :LedExit ; no action

bank Leds
mov w, #Digits ; Indirectly read data for
add w, Digit ; the current digit
mov fsr, w ;
mov w, ind ;
and w, #%00001111 ; Don't allow values > 15
call Decode ; Decimal --> 7-Segment
clr rc ; Set all columns to low
mov rb, w ; Set the segment rows
bank Leds
mov rc, Column ; Set one column line to high
clc ; and prepare for
rl Column ; next column and for
inc Digit ; next digit
mov w, #6 ; If digit > 5,
mov w, Digit-w
snz
clr Digit ; digit = 0,
mov w, #1
snz
mov Column, w ; and column mask = %00000001

:LedExit
mov w, #-200 ; Call the ISR every 4us
reti w

; ** Subroutine returns the 7-segment coding in w when called with
; a hexadecimal digit in w.
;
; Note: : Negative logic - 0-Bits turn on the associated LED
;
Decode
jmp pc+w

; Segments
; dcbefg
retw %00100100 ; 0
retw %10101111 ; 1
retw %01000110 ; 2
retw %00001110 ; 3
retw %10001101 ; 4
retw %00011100 ; 5
retw %00010100 ; 6
retw %10101110 ; 7
```

```

retw %00000100 ; 8
retw %00001100 ; 9
retw %10000100 ; A
retw %00010101 ; b
retw %01110100 ; C
retw %00000111 ; d
retw %01010100 ; E
retw %11010100 ; F

org $100

; ** Mainline program *****
;
Main
include "Clr2x.inc"

mov rb, #$ff
mov !rb, #0 ; Outputs for cathodes
clr rc
mov !rc, #%11000000 ; Outputs for anodes

bank Leds
mov Column, #1 ; Initialize the column mask
mov !option, #%10011111 ; Enable RTCC interrupt

mov Counter+2, #8 ; Initialize the time delay

Loop

bank Timers
sb Millsec.1 ; "Borrow" bit 1 of millisecs
jmp Loop
decsz Counter ; Time delay
jmp Loop
decsz Counter +1
jmp Loop
decsz Counter +2
jmp Loop

mov Counter+2, #8 ; Re-initialize the time delay

bank Leds

inc Digits ; Increment the lowest digit,
sb Digits.4 ; if > 15,
jmp Loop
clrb Digits.4 ; reset it to 0, and
inc Digits+1 ; increment next digit
sb Digits+1.4
jmp Loop ; etc. ...
clrb Digits+1.4

inc Digits+2

```

Programming the SX Microcontroller

```
sb Digits+2.4
  jmp Loop
clrb Digits+2.4

inc Digits+3
sb Digits+3.4
  jmp Loop
clrb Digits+3.4

inc Digits+4
sb Digits+4.4
  jmp Loop
clrb Digits+4.4

inc Digits+5
sb Digits+5.4
  jmp Loop
clrb Digits+5.4
jmp Loop
```

The VP executed in the ISR takes care of driving the display matrix, so the mainline program must not take care of that task. Values copied into the **Digits** registers are automatically displayed.

The mainline program “borrows” a bit of the **Millisec** timer counter in the ISR to clock a three level delay loop for demonstration purposes. When the time delay has elapsed, the **Digits** registers are incremented. It is important to limit the contents of each **Digit** register to a maximum of 15 (\$F) as each position can only display digits from 0...9 and letters from a...f.

The ISR is called every 4 μ s, and the **Millisec** counter is initialized to 250, i.e. it overflows after 1 ms, and the display routine is then executed.

```
bank Leds
mov w, #Digits          ; Indirectly read data for
add w, Digit           ; the current digit
mov fsr, w              ;
mov w, ind              ;
and w, #%00001111      ; Don't allow values > 15
call Decode             ; Decimal --> 7-Segment
```

The contents of **Digits** register are read into w for the current display position, and the higher for bits are reset as a safety measure to avoid values > 15. Then follows a call of the **Decode** subroutine.

```

Decode
  jmp pc+w

  ; Segments
  ; dcgbe. fa
  retw %00100100          ; 0
  retw %10101111         ; 1
  ;
  ; etc.
  ;
  retw %11010100         ; F

```

This subroutine reads the bit pattern that is required to turn on the LEDs for the value in *w*, and it returns that pattern in *w*. When you have connected the segment rows to port outputs other than shown in the schematic, you need to adapt this table accordingly. If you only want to display the 10 decimal digits 0...9, the last 6 table entries can be removed.

Here you can see why it is important not to allow a value in *w* that is larger than the number of table elements minus one, otherwise the **jmp pc+w** instruction might lead into “nowhere land”.

The final instructions in the VP

```

  clc                ; and prepare for
  rl Column          ; next column and for
  inc Digit          ; next digit
  mov w, #6          ; If digit > 5,
  mov w, Digit-w
  snz
  clr Digit          ; digit = 0,
  mov w, #1
  snz
  mov Column, w     ; and column mask = %00000001

```

set the next higher bit in the column mask and select the next display digit. If the highest digit is exceeded, the lowest digit is selected again, and the column mask is reset to the first column.

The selected timing of 1 ms means that every millisecond a new digit is driven, and each digit is refreshed every 6 ms which results in a frequency of approximately 167 Hz – fast enough that you will not notice a flickering display.

The instructions

```

  clr rc              ; Set all columns to low
  mov rb, w           ; Set the segment rows
  bank Leds
  mov rc, Column     ; Set one column line to high

```

drive the current 7-segment unit. It makes sense to first turn off the last activated unit before activating the next one by using the **clr rc** instruction in order to avoid that the “old” digit has the LEDs for the “new” digit turned on for a short while. Here, a “short while” means just two in-

Programming the SX Microcontroller

struction cycles, i.e. 40 ns, so you will definitely not see this “glitch”, but it is always a good idea to avoid un-necessary spikes in order to reduce noise.

We will use the 7-Segment VP in some other applications for more important tasks than just to display the contents of a counter as shown in this program.

4.5.1 Program Variations

With a simple enhancement, the program can be used to display the contents of any three SX registers in hexadecimal on the 6-digits display:

```
; =====
; Programming the SX Microcontroller
; APP013. SRC
; =====
include "Setup28.inc"
RESET    Main

; ** Macro copies the contents of a register into two positions
;    of the display buffer.
;
; Call:      MovDisp <First Display Buffer Register>, <Register>
;
; Uses:      : w
;
MovDisp MACRO 2
    mov w, \2                ; Display value -> w
    mov Digits + (\1 * 2), w ; Save value to left and right
    mov Digits + 1 + (\1 * 2), w ; display buffer item
    swap Digits + 1 + (\1 * 2) ; Exchange upper and lower
                                ; nibble in left digit
ENDM

org      $08
Std      = $
Value    ds 3                ; Three registers for demonstration
                                ; purposes

org      $30
Leds     = $
Column   ds 1                ; The LED bank
Digit    ds 1                ; Column mask
Digits   ds 6                ; Current display digit
; Digits buffer

org      $50
Timers   = $
Millsec  ds 1                ; Counter for Milliseconds

org      $000

    MovDisp 0, Value          ; Refresh the display
    MovDisp 1, Value+1        ; Buffer
    MovDisp 2, Value+2        ;
```

```

; ** VP to control a 7 segment LED matrix with 6 Digits *****
;
; Seg_7
;
bank Timers
mov w, #250
dec Millsec
snz
    mov Millsec, w
sz
    jmp :LedExit

bank Leds
mov w, #Digits           ; Indirectly read data for
add w, Digit             ; the current digit
mov fsr, w
mov w, ind
and w, #%00001111       ; Don't allow values > 15
call Decode              ; Decimal --> 7-Segment
clr rc                   ; Set all columns to low
mov rb, w                ; Set the segment rows
bank Leds
mov rc, Column          ; Set one column line to high
clc                       ; and prepare for
rl Column               ; next column and for
inc Digit               ; next digit
mov w, #6                ; If digit > 5,
mov w, Digit-w
snz
    clr Digit            ; digit = 0,
mov w, #1
snz
    mov Column, w       ; and column mask = %00000001

:LedExit
mov w, #-200             ; Call the ISR every 4us
reti w

; ** Subroutine returns the 7-segment coding in w when called with
; a hexadecimal digit in w.
;
; Note: : Negative logic - 0-Bits turn on the associated LED
;
Decode
    jmp pc+w

; Segments
;
;          dcgbe.fa
retw %00100100          ; 0
retw %10101111          ; 1
retw %01000110          ; 2

```

Programming the SX Microcontroller

```
retw %00001110 ; 3
retw %10001101 ; 4
retw %00011100 ; 5
retw %00010100 ; 6
retw %10101110 ; 7
retw %00000100 ; 8
retw %00001100 ; 9
retw %10000100 ; A
retw %00010101 ; b
retw %01110100 ; C
retw %00000111 ; d
retw %01010100 ; E
retw %11010100 ; F

org $100

; ** Main line program *****
;
Main
include "Clr2x.inc"

mov rb, #$ff
mov !rb, #0 ; Outputs for cathodes
clr rc
mov !rc, #%11000000 ; Outputs for anodes

bank Leds
mov Column, #1 ; Initialize the column mask
mov !option, #%10011111 ; Enable RTCC interrupt

mov Value, #$ef
mov Value+1, #$cd
mov Value+2, #$ab

Loop
jmp Loop
```

At the beginning of the program, we have defined the **MovDisp** macro that copies the contents of the register to be displayed into two subsequent positions in the display buffer. As the 7-Segment VP clears the upper four bits in a value, we don't need to take care about it here in the macro.

In this example program, we have reserved three bytes for the **Value** variable that is initialized with \$abcdef.

At start of the ISR, the **MovDisp** macro is called for each byte of **Value** to copy its contents to the display buffer. As the contents of **Value** is constant in this example, it would be sufficient to do this copy just once, but in "real life", we must assume that the contents of **Value** might change at any time.

A bit more complex is the task to display the contents of two registers in decimal on three display digits each:

```

; =====
; Programming the SX Microcontroller
; APP014. SRC;
; =====
include "Setup28.inc"
RESET    Main

org      $08
Std      = $
Value    ds 2           ; Test values
BCD      ds 3           ; Buffer for BCD digits
Hex      ds 1           ; Buffer for Hex value

org      $30
Leds     = $           ; The LED bank
Column   ds 1           ; Column mask
Digit    ds 1           ; Current display digit
Digits   ds 6           ; Digits buffer

org      $50
Timers   = $
Millsec  ds 1           ; Counter for Milliseconds

org      $000

; ** VP to control a 7 segment LED matrix with 6 Digits *****
;
; Seg_7

bank Timers
mov w, #250
dec Millsec
snz
    mov Millsec, w
sz
    jmp :LedExit

bank Leds
mov w, #Digits           ; Indirectly read data for
add w, Digit             ; the current digit
mov fsr, w               ;
mov w, ind               ;
and w, #%00001111       ; Don't allow values > 15
call Decode              ; Decimal --> 7-Segment
clr rc                   ; Set all columns to low
mov rb, w                ; Set the segment rows
bank Leds
mov rc, Column           ; Set one column line to high
clc                       ; and prepare for
rl Column                ; next column and for
inc Digit                ; next digit

```

Programming the SX Microcontroller

```
mov w, #6 ; If digit > 5,
mov w, Digit-w
snz
  clr Digit ; digit = 0,
mov w, #1
snz
  mov Column, w ; and column mask = %00000001

:LedExit
mov w, #-200 ; Call the ISR every 4us
reti w

; ** Subroutine returns the 7-segment coding in w when called with
; a hexadecimal digit in w.
;
; Note: : Negative logic - 0-Bits turn on the associated LED
;
;
Decode
  jmp pc+w

; Segments
;
;
;   dcgbe.fa
retw %00100100 ; 0
retw %10101111 ; 1
retw %01000110 ; 2
retw %00001110 ; 3
retw %10001101 ; 4
retw %00011100 ; 5
retw %00010100 ; 6
retw %10101110 ; 7
retw %00000100 ; 8
retw %00001100 ; 9

; ** Subroutine converts value in Hex into a 3-digits BCD number and
; stores the result in BCD+2...BCD
;
;
HexToBCD
  clr BCD+2 ; Clear the
  clr BCD+1 ;
  clr BCD ; BCD buffer
  mov w, #100 ; Determine leftmost digit
:LedLoop100
  sub Hex, w
  snc
  inc BCD+2
  snc
  jmp :LedLoop100
  add Hex, w
  mov w, #10 ; Determine middle digit
:LedLoop10
  sub Hex, w
  snc
  inc BCD+1
  snc
```

```

    jmp :Loop10
    add Hex, w
    mov BCD, Hex                ; The remainder goes into the
    ret                        ; rightmost digit

org    $100

; ** Mainline program *****
;
Main

include "Clr2x.inc"

    mov    rb, #$ff
    mov    !rb, #0              ; Outputs for cathodes
    clr    rc
    mov    !rc, #%11000000     ; Outputs for anodes
    bank  Leds
    mov    Column, #1          ; Initialize the column mask
    mov    !option, #%10011111 ; Enable RTCC interrupt

    mov    Value, #123
    mov    Value+1, #234

    mov    Hex, Value
    call   HexToBCD

Loop
    mov    Hex, Value          ; Display Value in Digits+2...
    call   HexToBCD           ; Digits as decimal number
    mov    Digits, BCD
    mov    Digits+1, BCD+1
    mov    Digits+2, BCD+2

    mov    Hex, Value+1       ; Display Value+1 in Digits+5...
    call   HexToBCD           ; Digits+3 as decimal number
    mov    Digits+3, BCD
    mov    Digits+4, BCD+1
    mov    Digits+5, BCD+2

    jmp   Loop

```

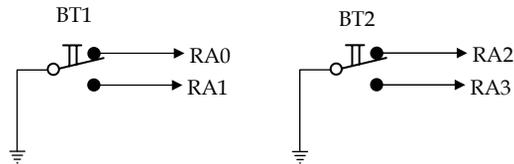
To convert a byte into a 3-digits decimal representation, we need to convert a hex number into its BCD (Binary Coded Decimal) representation. This is done in the **HexToBCD** subroutine by repeated subtraction of 100 and 10, i.e. first, we divide the value by 100 and the integer part of the result goes into the leftmost digit. Then we divide the remainder by 10, and the integer part goes into the middle digit. The remainder finally goes into the rightmost digit.

Again, we display the constant contents of two registers (**Value**, and **Value+1**) in this demo program. This time, the display buffer is refreshed in the mainline program.

4.6 An SX Stopwatch

The VPs presented in the previous chapters for a clock timer, and to control a 7-segment, 6-digits display is a good basis to “construct” a stopwatch.

In addition to the components shown in the 7-segment diagram, we need two pushbuttons that should be connected according to the following diagram:



Here we use pushbuttons with a make and a break contact in order to make de-bouncing easy.

Next, we should define the functions of the stopwatch:

- Precision 1/100 second, Display from 00:00:00 to 59:59:59.
- Start with BT1, display of running time. If the last stop time was not cleared before, add new stop time to this time.
- Stop with BT1, display the new stop time.
- Display an elapsed time when BT2 is pressed, and blink the decimal point in the right-most digit to indicate that the clock is still running.
- Display a new elapsed time when BT2 is pressed, the clock is still running, and an elapsed time is currently displayed.
- Display the running time again when BT1 is pressed, the clock is running, and an elapsed time is currently displayed.
- Clear the stop time with BT2 when the clock is stopped.

Clock timing and display control is handled by the two VPs we have already shown, and the user-interface is handled in the mainline program.

It is a good idea to design the mainline program as a “State Engine” because, according to the specifications, there are different reactions on button presses required, depending on the status of the stopwatch:

- Clock stopped
 - BT1 = Start the clock
 - BT2 = Clear the stop time
- Clock is running, running time is displayed
 - BT1 = Stop the clock
 - BT2 = Display elapsed time, keep clock running
- Clock is running elapsed time is displayed
 - BT1 = Display running time
 - BT2 = Display new elapsed time

Here comes the stopwatch program:

```

; =====
; Programming the SX Microcontroller
; APP015. SRC
; =====
include "Setup28.inc"
RESET    Main

TRIS     = $0f
PLP      = $0e

; ** Macro definitions *****
;
SkipIfBT1IsUp MACRO          ; Skip if BT1 is not pressed
    snb ra. 0
ENDM

SkipIfBT2IsUp MACRO          ; Skip if BT2 is not pressed
    snb ra. 2
ENDM

SkipIfBT1IsDown MACRO        ; Skip if BT1 is pressed
    snb ra. 1
ENDM

SkipIfBT2IsDown MACRO        ; Skip if BT2 is pressed
    snb ra. 3
ENDM

TickOn MACRO                  ; Turn 1 ms ticker on
    setb Flags. 0
ENDM

TickOff MACRO                  ; Turn 1 ms ticker off

```

Programming the SX Microcontroller

```
    clrb Flags. 0
ENDM

SkipIfTick MACRO          ; Skip if ticker is on
    sb Flags. 0
ENDM

BlinkOn MACRO            ; Turn on blink mode
    setb Flags. 3
ENDM

BlinkOff MACRO          ; Turn off blink mode
    clrb Flags. 3
ENDM

SkipIfBlink MACRO       ; Skip if blink mode is on
    sb Flags. 3
ENDM

ClockOn MACRO           ; Turn the clock timer on
    setb Flags. 1
ENDM

ClockOff MACRO          ; Turn the clock timer off
    clrb Flags. 1
ENDM

SkipIfClockIsOn MACRO  ; Skip if the clock timer is on
    sb Flags. 1
ENDM

SetWaitRelease MACRO   ; Turn on mode "Wait for button
    setb Flags. 2      ; release"
ENDM

ClrWaitRelease MACRO   ; Turn off mode "Wait for button
    clrb Flags. 2     ; release"
ENDM

SkipIfWaitRelease MACRO ; Skip is mode "Wait for button
    sb Flags. 2      ; release" is on
ENDM

org     $08
Std     = $
Temp    ds 1          ; Temporary storage
Ix      ds 1          ; Index variable
State   ds 1          ; State for "State Engine"
Flags   ds 1          ; Various flags
FsrSave ds 1          ; Temporary Storage for FSR

org     $30
Leds    = $           ; Variables for the 7-Segment VP
Column  ds 1          ; Column mask
Digit   ds 1          ; Current display position
```

```

Digits  ds 6           ; Digits buffer

org     $50
Timers  = $           ; Timer bank
us4     ds 1           ; 4us counter
Msec    ds 1           ; 1/1000 sec counter
HSec    ds 1           ; 1/100 sec counter
TSec    ds 1           ; 1/10 sec counter
Sec     ds 1           ; 1 sec counter
Sec10   ds 1           ; 10 sec counter
Min     ds 1           ; 1 min counter
Min10   ds 1           ; 10 min counter

org     $000

; ** Timer-VP for the stopwatch *****
;
; Timer

Bank Timers
mov     w, #250        ; 4ms * 250 = 1 ms
dec     us4
snz
    mov us4, w
snz
    Tick0n            ; Set 1ms ticker

SkipIfClockIs0n      ; If clock is off, continue with
    jmp :Display      ; 7-Segment VP
snz
    inc Msec           ; every millisecond
mov     w, #10
mov     w, MSec-w      ; Z if MSec = 10
snz
    clr Msec
snz
    inc Hsec           ; every 1/100 sec
mov     w, #10
mov     w, HSec-w      ; Z if HSec = 10
snz
    clr Hsec
snz
    inc Tsec           ; every 1/10 sec
mov     w, #10
mov     w, TSec-w      ; Z if TSec = 10
snz
    clr Tsec
snz
    inc Sec            ; every second
mov     w, #10
mov     w, Sec-w       ; Z if sec = 10
snz
    clr Sec
snz

```

Programming the SX Microcontroller

```
inc Sec10          ; every 10 seconds
mov w, #6
mov w, Sec10-w    ; Z if Sec10 = 6
snz
clr Sec10
snz
inc Min          ; every minute
mov w, #10
mov w, Min-w     ; Z if Min = 10
snz
clr Min
snz
inc Min10       ; every 10 minutes
mov w, #6
mov w, Min10-w
snz
clr Min10

; ** VP to control a 7-segment, 6-digits LED display *****
;
;
: Display
SkipIfTick      ; Action only if the timer has
jmp :ISRExit    ; set the 1ms tick
TickOff        ; Clear the tick flag
bank Leds      ; Get value for current digit
mov w, #Digits ; from Digits
add w, Digit
mov fsr, w
mov w, ind     ; indirect and
call Decode   ; 7-seg pattern to W
clr rc        ; Turn off the current digit
mov rb, w
SkipIfBlink    ; If blinking is on,
jmp :NoBlink
test Digit     ; check if 1/100 seconds digit is
sz            ; the current one, and
jmp :NoBlink
bank Timers    ; if so, get bit 0 of the
movb c, TSec.0 ; 1/10 seconds timer to C.
bank Leds
snc           ; If set, turn on the DP
clrb rb.2    ; LED.

: NoBlink
mov rc, Column ; Output column data
bank Leds
clc           ; Select next column
rl Column
inc Digit    ; Select next digit
mov w, #6    ; If we are past digit 5,
mov w, Digit-w
snz ;
clr Digit   ; activate digit 0,
mov w, #1   ; and select
snz ;
```

```

    mov Column, w          ; column 0 too.

:ISRExit
    mov w, #-200          ; Call the ISR every 4us
    reti w

; ** Subroutine returns the 7-seg pattern for a decimal digit in w
;
; Decode
    jmp pc+w
    retw %00100100        ; 0
    retw %10101111        ; 1
    retw %01000110        ; 2
    retw %00001110        ; 3
    retw %10001101        ; 4
    retw %00011100        ; 5
    retw %00010100        ; 6
    retw %10101110        ; 7
    retw %00000100        ; 8
    retw %00001100        ; 9

org    $100

; ** Mainline program *****
;
; Main
include "Clr2x.inc"

    mode PLP
    mov !ra, #%11110000    ; Pull-up for buttons
    mode TRIS
    mov rb, #$ff
    mov !rb, #0            ; Outputs for cathodes
    clr rc
    mov !rc, #%11000000    ; Outputs for anodes

    bank Leds
    mov Column, #1         ; Initialize column mask
    mov !option, #%10011111; Enable RTCC interrupt
    jmp @MainLoop         ; Continue with main program loop

    org $200
; ** Subroutine copies the current time information into the
;    display buffer.
;
;
; TimeToDisp
    mov FSRsave, fsr      ; Save FSR
    mov Ix, #5           ; Initialize index

: Copy
    mov w, #Hsec          ; Timer base address --> w,
    add w, Ix             ; add offset,
    mov fsr, w            ; setup indirect address,

```

Programming the SX Microcontroller

```
    mov Temp,    ind      ; copy timer variable to Temp.
    mov w,      #Digits  ; Digits base address --> w,
    add w,      Ix       ; add offset,
    mov fsr,    w        ; setup indirect address,
    mov ind,    Temp     ; copy timer variable --> Digits.
    dec Ix      ; Next lower index
    sb Ix.7     ; If not < 0,
    jmp :Copy   ; copy more digits
    mov fsr,    FSRSave  ; Restore FSR
ret    ; Done...

; ** Subroutine clears all timer registers
;
ClrTime
    mov FSRSave, fsr     ; Save FSR
    mov Ix,      #7      ; Initialize index

:Clear
    mov w,      #Hsec    ; Timer base address --> w,
    add w,      Ix       ; add offset,
    mov fsr,    w        ; setup indirect address,
    clr ind     ; clear timer variable.
    dec Ix      ; Next lower index
    sb Ix.7     ; If not < 0,
    jmp :Clear  ; clear mode variables
    mov fsr,    FSRSave  ; Restore FSR
ret    ; Done...

; ** Main program loop
;
MainLoop

    mov w, #2          ; If State <> 2 (elapsed time),
    mov w, State-w    ;
    sz
        call TimeToDisp ; copy and display the time

    skipIfWaitRelease ; Wait for button release if
        jmp :ExecModes ; flag is set.
    skipIfBT1IsUp     ; BT1 not released,
        jmp MainLoop   ; continue waiting...
    skipIfBT2IsUp     ; BT2 not released,
        jmp MainLoop   ; continue waiting
    clrWaitRelease    ; Clear the wait flag

:ExecModes            ; "State Engine"
    mov w, State      ; Select mode according to current
    jmp pc+w          ; state
    jmp ModeStop
    jmp ModeRun
    jmp ModeElap

ModeStop              ; Clock is stopped
    skipIfBT1IsDown   ; BT1 not pressed, check
    jmp :TestBT2     ; BT2
```

```

ClockOn          ; BT1 pressed, turn clock on, and
SetWaitRelease  ; activate mode "Wait for Button
                ; Release"
inc State       ; Next state is "ModeRun"
jmp MainLoop

:TestBT2        ; Check BT2
SkipIfBT2IsDown ; BT2 not pressed,
  jmp MainLoop  ; no action
call ClrTime    ; BT2 pressed, clear time and
SetWaitRelease  ; activate mode "Wait for Button
                ; Release"
jmp MainLoop

ModeRun         ; Clock is running
SkipIfBT1IsDown ; BT1 not pressed, check
  jmp :TestBT2  ; BT2
ClockOff       ; BT1 pressed, stop the clock, and
SetWaitRelease ; activate mode "Wait for Button
                ; Release"
clr State      ; Next State is ModeStop
jmp MainLoop

:TestBT2        ; Check BT2
SkipIfBT2IsDown ; BT2 not pressed,
  jmp MainLoop  ; no action
inc State       ; BT2 pressed, next State is
                ; ModeElap
BlinkOn        ; Set the blink flag to make DP
                ; in the rightmost digit blink
SetWaitRelease ; Activate mode "Wait for Button
                ; Release"
jmp MainLoop

ModeElap       ; Display elapsed time
SkipIfBT1IsDown ; BT1 not pressed, check
  jmp :TestBT2  ; BT2
SetWaitRelease ; Activate mode "Wait for Button
                ; Release"
BlinkOff       ; BT1 pressed, clear the blink flag
dec State      ; Next state is ModeRun
jmp MainLoop

:TestBT2        ; Check BT2
SkipIfBT2IsDown ; BT2 not pressed,
  jmp MainLoop  ; no action
call TimeToDisp ; Display new elapsed time
SetWaitRelease  ; Activate mode "Wait for Button
                ; Release"
jmp MainLoop    ; again, and again...

```

Programming the SX Microcontroller

At the beginning of the program, several macros are defined that make it easier to read the remainder of the program, and that allow an easier change of the port pin assignments. In this case, you only need to change the macros, while the rest of the code needs no modifications.

The Timer VP has been enhanced by two lines:

```
SkipIfClockIsOn          ; If clock is off, continue with  
jmp :Display           ; 7-Segment VP
```

in order to enable and disable the timer. The milliseconds timer is always active, no matter what the state of the **Clock** flag is so that the **Tick** flag is set as this is required by the 7-Segment VP.

The remainder of the Timer VP is identical to the previously shown program code, except that we have removed the hours timers as they are not needed here.

The 7-Segment VP is also almost identical to the version already shown. We have added the following instructions:

```
SkipIfBlink             ; If blinking is on,  
jmp :NoBlink           ;  
test Digit             ; check if 1/100 seconds digit is  
sz                     ; the current one, and  
jmp :NoBlink           ;  
bank Timers           ; if so, get bit 0 of the  
movb c, TSec.0        ; 1/10 seconds timer to C.  
bank Leds             ;  
snc                   ; If set, turn on the DP  
clrb rb.2            ; LED.
```

:NoBlink

These instructions make the decimal point in the rightmost digit blink when the display shows an elapsed time. This feature is controlled by the **Blink** flag bit of the **Flags** variable, and it is tested here by calling the **SkipIfBlink** Macro.

Only when the rightmost digit is active (**Digit = 0**) the decimal point must eventually be turned on. If this is the case, the contents of bit 0 in the 1/10 seconds timer is “borrowed” to turn DP on or off.

The mainline program first clears all data registers, initializes the ports, enables the RTCC interrupt, and then enters into the main program loop that is located in the second program memory page.

In this second page, you first find the Subroutines **TimeToDisp** that copies the current timer contents into the display buffer, and **ClrTime** that clears the timer registers.

The main program loop begins with the following instructions:

Main

```
mov w, #2 ; If State <> 2 (elapsed time),
```

```

mov w, State-w          ;
sz                       ;
  call TimeToDisp       ; copy and display the time

```

The current timer contents are copied to the display buffer registers except when the content of State is 2 (Display Elapsed Time). Then it eventually waits for BT1 and/or BT2 to be released:

```

SkipIfWaitRelease      ; Wait for button release if
  jmp :ExecModes        ; flag is set.
SkipIfBT1IsUp          ; BT1 not released,
  jmp Main              ; continue waiting...
SkipIfBT2IsUp          ; BT2 not released,
  jmp Main              ; continue waiting
ClrWaitRelease         ; Clear the wait flag

```

This is an intermediate state of the “State Engine”. During that state, the 7-segment display is still updated (in case the clock is running, and does not show an elapsed time), but further actions are “put on hold” until the buttons are released (this is how de-bouncing is handled here).

```

: ExecModes              ; "State Engine"
  mov w, State           ; Select mode according to current
  jmp pc+w               ; state
  jmp ModeStop
  jmp ModeRun
  jmp ModeElap

```

These instructions act as “State Selector”, i.e. depending on the current state the necessary instructions are executed.

While in the “Stop” state, the instructions

```

ModeStop                ; Clock is stopped
  SkipIfBT1IsDown       ; BT1 not pressed, check
  jmp :TestBT2          ; BT2
  ClockOn                ; BT1 pressed, turn clock on, and
  SetWaitRelease        ; activate mode "Wait for Button
  ; Release"
  inc State              ; Next state is "ModeRun"
  jmp Main

: TestBT2                ; Check BT2
  SkipIfBT2IsDown       ; BT2 not pressed,
  jmp Main              ; no action
  call ClrTime           ; BT2 pressed, clear time and
  SetWaitRelease        ; activate mode "Wait for Button
  ; Release"
  jmp Main

```

are executed. If BT1 is pressed, the clock will start, and the “Run” state is activated. The macro **SetWaitRelease** activates the intermediate state that waits for button releases.

When BT2 is pressed, all timer registers are cleared, i.e. the stop time is reset to zero, and again, the intermediate mode to wait for button releases is activated.

Programming the SX Microcontroller

During the “Run” mode, the clock keeps running, and the following instructions take care of the necessary actions:

```
ModeRun          ; Clock is running
  SkipIfBT1IsDown ; BT1 not pressed, check
    jmp :TestBT2  ; BT2
  ClockOff        ; BT1 pressed, stop the clock, and
  SetWaitRelease  ; activate mode "Wait for Button
                  ; Release"
  clr State       ; Next State is ModeStop
  jmp Main

:TestBT2         ; Check BT2
  SkipIfBT2IsDown ; BT2 not pressed,
    jmp Main      ; no action
  inc State       ; BT2 pressed, next State is
                  ; ModeElap
  BlinkOn        ; Set the blink flag to make DP
                  ; in the rightmost digit blink
  SetWaitRelease ; Activate mode "Wait for Button
                  ; Release"
  jmp Main
```

When BT1 is pressed, the macro **ClockOff** stops the timer, but the **timer** register contents are still maintained and displayed, i.e. the current stop time. **clr State** resets the state to “Stop”, and because a button was pressed, we need to wait for the button release.

When BT2 is pressed, the user wants to see an elapsed time. Therefore, the next state, **ModeElap** is set, **BlinkOn** “informs” the 7-Segment VP to blink the rightmost decimal point. As the contents of **State** is 2 now, the instruction **call TimeToDisp** at the beginning of the main loop is skipped now, i.e. the time display is no longer refreshed, but the timer continues to update the time „in the background”. Again, we need to wait for the button to be released.

While an elapsed time is displayed, the following instructions are executed:

```
ModeElap        ; Display elapsed time
  SkipIfBT1IsDown ; BT1 not pressed, check
    jmp :TestBT2  ; BT2
  SetWaitRelease  ; Activate mode "Wait for Button
                  ; Release"
  BlinkOff        ; BT1 pressed, clear the blink flag
  dec State       ; Next state is ModeRun
  jmp Main

:TestBT2         ; Check BT2
  SkipIfBT2IsDown ; BT2 not pressed,
    jmp Main      ; no action
  call TimeToDisp ; Display new elapsed time
  SetWaitRelease  ; Activate mode "Wait for Button
                  ; Release"
  jmp Main ; again, and again...
```

When BT1 is pressed, the blink flag is cleared and the “Run” state becomes active, i.e. the display buffer will be updated from the timer registers again.

If BT2 is pressed, the current content of the timer running in the „background“ is copied to the display buffer in order to display a new elapsed time.

In both cases, we need to wait for the buttons to be released.

(Stop)Watch Out – The Macro “Pitfall” !

Macros are a nice aid to make a program more readable, and more “generic”, on the other hand, they can also add a “pitfall” to the programmer’s hard life:

Just for fun, see what happens if you change the name of the **Ski pI fCl ockI s0n** macro in the definition to **SkpI fCl ockI s0n** (remove the “i” in “Ski p”) and re-assemble the program. When you are using the SASM Assembler you may get an error reporting that labels must begin in column 1 for the line where the **Ski pI fCl ockI s0n** macro is called. The SX-Key assembler will not complain at all, and SASM will also be happy if you happen to have not entered the leading space in before the **Ski pI fCl ockI s0n** macro call.



This macro is used to skip the Timer VP in case the clock is in “Stop” state.

Ski pI fCl ockI s0n

```
    j mp : Di spl ay
```

snz

Now that **Ski pI fCl ockI s0n** is no longer defined as a macro, but **SkpI fCl ockI s0n** instead, the Assembler “assumes” that **Ski pI fCl ockI s0n** is a global label now, and so the **j mp : Di spl ay** instruction will always be executed, and the Timer VP is never active..

4.7 A Digital SX Alarm Clock

The step from a stopwatch to a “real” digital clock with alarm function is not too large but the program requires some additional enhancements that are necessary to set the time, and the alarm time.

The schematic for the digital clock is the same as for the stopwatch. If you like, you could connect a loudspeaker to port pin RC7 in order to generate an acoustic alarm.

Again, it makes sense to develop a state engine that handles the user interface, but before writing the engine, we first should define the functions for the two buttons (we call them “Mode” (BT1), and “Set” (BT2) here):

- Default “Clock mode” – The current time is displayed, and if an alarm is enabled, the two decimal points in the seconds digits shall be on. When the alarm time has been reached, the two decimal points shall blink, and the buzzer shall generate an intermittent tone alarm sound.
 - No alarm active:
 - “Mode” activates the “Set clock” function to adjust the time.
 - “Set” activates setting of the alarm time.
 - Alarm active:
 - Any button press stops the alarm.
- “Set clock” mode – The display shows the time when “Mode” was pressed, and the digit that can be currently changed is marked with a blinking decimal point (starting with the one-seconds digit).
 - “Set” increments the current decimal.
 - “Mode” activates the next digit to be changed.
 - When “Mode” is held down when the 10-hours digit is active, and “Set” is pressed while “Mode” is down, the entered time will be accepted, and the clock starts with this new time as soon as both buttons are released.
 - When “Mode” is pressed and released and “Set” was not pressed while “Mode” was down, the entered time will be discarded, and the former clock setting remains unchanged, i.e. the display shows the current clock time again that was updated in the background during the “Set clock” mode.
- “Set alarm” mode – The display shows the last entered alarm time, and the digit that can be currently changed is marked with a steady decimal point (starting with the one-seconds digit).
 - “Set” increments the current decimal.
 - “Mode” activates the next digit to be changed.

- When “Mode” is held down when the 10-hours digit is active, and “Set” is pressed while “Mode” is down, the entered time will be accepted as the new alarm time, and the alarm is enabled.
- When “Mode” is pressed and released and “Set” was not pressed while “Mode” was down, the entered alarm time will be discarded, and the alarm will be disabled.

As you can see, the state engine is more complex than the one controlling the stopwatch user interface. There are three major states: the “Clock” state, the “Set clock” state, and the “Set alarm” state. Within these major states, there are different sub-states, depending on the major state, but most of the sub-states in “Set clock”, and “Set alarm” are identical.

This is the program listing:

```

; =====
; Programming the SX Microcontroller
; APP016. SRC
; =====
include "Setup28.inc"
RESET    Main

TRIS     = $0f
PLP      = $0e

Buzzer   = rc. 7

; ** Macro Definitions *****
;
DotOn MACRO          ; Decimal point LED on
    clrb rb. 2
ENDM

SkipIfModeIsUp MACRO ; Skip if "Mode" button released
    snb ra. 0
ENDM

SkipIfModeIsNotDown MACRO ; Skip if "Mode" button not pressed
    sb ra. 1
ENDM

SkipIfSetIsNotDown MACRO ; Skip if "Set" button pressed
    sb ra. 3
ENDM

SkipIfSetIsUp MACRO   ; Skip if "Set" button released
    snb ra. 2
ENDM

SkipIfSetIsDown MACRO ; Skip if "Set" button pressed
    snb ra. 3
ENDM
TickOn MACRO          ; Turn ticker on

```

Programming the SX Microcontroller

```
    setb Flags. 0
ENDM

TickOff MACRO                ; Turn Ticker off
    clrb Flags. 0
ENDM

SkipIfTick MACRO            ; Skip if ticker is on
    sb Flags. 0 ;
ENDM

BlinkOn MACRO; Turn blink on
    setb Flags. 1
ENDM

BlinkOff MACRO              ; Turn blink off
    clrb Flags. 1
ENDM

SkipIfBlink MACRO          ; Skip if blink is on
    sb Flags. 1
ENDM

ArmAlarm MACRO              ; Enable the alarm function
    setb Flags. 2
ENDM

ClrAlarm MACRO              ; Disable the alarm function
    clrb Flags. 2
    clrb Flags. 3
ENDM

SkipIfAlarmSet MACRO       ; Skip if alarm is enabled
    sb Flags. 2
ENDM

SkipIfAlarm MACRO          ; Skip if alarm is triggered
    sb Flags. 3
ENDM

SetAlarm MACRO              ; Trigger the alarm
    setb Flags. 1
    setb Flags. 3
ENDM

SettingClock MACRO         ; Set mode flag "Set Clock"
    setb Flags. 4
ENDM

SettingAlarm MACRO         ; Set mode flag "Set alarm"
    clrb Flags. 4
ENDM

SkipIfSettingClock MACRO   ; Skip if mode "Set Clock" is
    sb Flags. 4            ; active
```

```

ENDM

org      $08
Std      = $ ; Global memory bank
Temp     ds 1; Temporary storage
Ix       ds 1; Index variable
State    ds 1; Current mode of state engine
SubState ds 1; Current sub-mode of state engine
Flags    ds 1; Various flags

org      $30
Leds     = $ ; LED bank
Column   ds 1; Column mask
Digit    ds 1; Current display digit
Digits   ds 6; Digits buffer
Alarm    ds 6; Storage for alarm time

org      $50
Timers   = $ ; Timer bank
us4      ds 1; 4us counter
Msec     ds 1; 1/1000 sec counter
HSec     ds 1; 1/100 sec counter
TSec     ds 1; 1/10 sec counter
Sec      ds 1; 1 sec counter
Sec10    ds 1; 10 sec counter
Min      ds 1; 1 min counter
Min10    ds 1; 10 min counter
Hour     ds 1; 1 hour counter
Hour10   ds 1; 10 hour counter

org      $000

; ** Clock VP *****
;
; Timers

Bank Timers
mov w, #250 ; 4ms * 250 = 1 ms
dec us4
snz
    mov us4, w
snz
    Tick0n
snz
    inc Msec ; every millisecond
mov w, #10
mov w, MSec-w ; Z if MSec = 10
snz
    clr Msec
snz
    inc Hsec ; every 1/100 sec
mov w, #10
mov w, HSec-w ; Z if HSec = 10
snz

```

Programming the SX Microcontroller

```
    clr Hsec
    snz
    inc Tsec      ; every 1/10 sec
    mov w, #10
    mov w, TSec-w ; Z if TSec = 10
    snz
    clr Tsec
    snz
    inc Sec      ; every second
    mov w, #10
    mov w, Sec-w ; Z if sec = 10
    snz
    clr Sec
    snz
    inc Sec10    ; every 10 seconds
    mov w, #6
    mov w, Sec10-w ; Z if Sec10 = 6
    snz
    clr Sec10
snz
    inc Min      ; every minute
    mov w, #10
    mov w, Min-w ; Z if Min = 10
    snz
    clr Min
    snz
    inc Min10    ; every 10 minutes
    mov w, #6
    mov w, Min10-w
    snz
    clr Min10
    snz
    inc Hour     ; every hour
    mov w, #10
    mov w, Hour-w
    snz
    clr Hour
    snz
    inc Hour10   ; every 10 hours
    mov w, #3
    mov w, Hour10-w
    snz
    clr Hour10   ; every day
; ** VP to control a 7-segment, 6-digits LED display *****
;
; Display
    SkipIfTick   ; Action only if the timer has
    jmp :ISRExit ; set the 1ms tick
    TickOff      ; Clear the tick flag
    bank Leds    ; Get value for current digit
    mov w, #Digits ; from Digits
    add w, Digit ;
    mov fsr, w   ;
    mov w, ind   ; indirect
```

```

and w, #%01111111 ; Mask bit 7 (the blink id)
call Decode ; 7-seg pattern to W
clr rc ; Turn off the current display
mov rb, w ; Output line data
SkipIfBlink ; If dots shall blink,
    jmp :DotOn ;
bank Timers ; "borrow" from TSec, and
movb c, TSec.0 ; save bit 0 in C
bank Leds
sc ; If TSec.0 is set, "On Phase",
    jmp :NoBlink ; else "Off Phase" for blinking

:DotOn
bank Leds ; Turn on if bit 7 is set in
snb ind.7 ; the digit's data storage
    DotOn

:NoBlink
mov rc, Column ; Output column data
SkipIfAlarm ; If alarm is triggered,
    jmp :NoBeep ;
bank Timers ;
snb Sec.0 ; send a 500 Hz signal to the
    jmp :NoBeep ; buzzer every other second
movb Buzzer, MSec.0

:NoBeep
bank Leds
clc
rl Column ; Select next column
inc Digit ; Select next digit
mov w, #6 ; If we are past digit 5,
mov w, Digit-w ;
snz ;
    clr Digit ; activate digit 0
mov w, #1 ;
snz ;
    mov Column, w ; and select column 0

:ISRExit
mov w, #-200 ; Call the ISR every 4us
reti w

; ** Subroutine returns the 7-seg pattern for a decimal digit in w
;
Decode
    jmp pc+w
    retw %00100100 ; 0
    retw %10101111 ; 1
    retw %01000110 ; 2
    retw %00001110 ; 3
    retw %10001101 ; 4
    retw %00011100 ; 5
    retw %00010100 ; 6

```

Programming the SX Microcontroller

```
    retw %10101110 ; 7
    retw %00000100 ; 8
    retw %00001100 ; 9

org    $100

; ** Mainline program *****
;
Main
; Clear the data memory
;
    clr    fsr
ClearRam
    sb     fsr.4
        Setb fsr.3
    clr    ind
    ijnz   fsr, ClearRam

    mode PLP
    mov   !ra, #%11110000    ; Pull-up for buttons
    mode TRIS
    mov   rb, #$ff          ; Set all RB outputs to high
    mov   !rb, #0           ; Enable segment outputs
    clr   rc
    mov   !rc, #%01000000   ; Outputs for anodes and buzzer

    bank Leds
    mov   Column, #1        ; Initialize column mask
    mov   !option, #%10011111 ; Enable RTCC interrupt
    jmp   @MainLoop        ; Continue with main program loop

    org $200

; ** Subroutine waits until the "Mode" button is released
;
WaitReleaseM
    SkipIfModeIsUp
        jmp WaitReleaseM
    ret

; ** Subroutine waits until the "Set" button is released
;
WaitReleaseS
    SkipIfSetIsUp
        jmp WaitReleaseS
    ret

; ** Subroutine waits until both buttons are released
;
WaitReleaseBoth
    SkipIfSetIsUp
        jmp WaitReleaseBoth
    SkipIfModeIsUp
        jmp WaitReleaseBoth
```

```

ret

; ** Main program loop
;
MainLoop
  mov w, State          ; Jump table for state engine
  jmp pc+w
  jmp Clock
  jmp SetClock
  jmp SetAlarmTime

; ** Handle state "Clock is running"
;
Clock
  mov Ix, #5           ; Copy the current time information
: Copy                ; to the display buffer
  mov w, #Sec         ; Base is the Hour10 timer register,
  add w, Ix           ; add the index,
  mov fsr, w          ; address the timer register and
  mov Temp, ind       ; read it indirect
  mov w, #Digits      ; Base is Digits+5
  add w, Ix           ; add the index,
  mov fsr, w          ; and
  mov ind, Temp       ; write data indirect
  dec Ix              ; Next lower position
  sb Ix.7             ; If Ix <> -1,
  jmp :Copy           ; copy more digits

  SkipIfAlarmSet     ; If the alarm is enabled,
  jmp :NoAlarm
  setb Digits.7      ; turn on the decimal points in
  setb Digits+1.7    ; the seconds digits

; ** Check if it is time to trigger the alarm
;
  mov Ix, #5         ; For this test, the current con-
: CheckAlarm         ; tents of all 6 digits is com-
  mov w, #Digits     ; pared with the contents of
  add w, Ix          ; the alarm time buffer.
  mov fsr, w
  mov Temp, ind
  clrb Temp.7
  mov w, #Alarm
  add w, Ix
  mov fsr, w
  mov w, ind
  mov w, Temp-w
  sz
  jmp :NoAlarm       ; If any digit is different, cancel
  dec Ix             ; the compare
  sb Ix.7
  jmp :CheckAlarm
  SetAlarm          ; Trigger the alarm if all digits
                   ; are equal

```

Programming the SX Microcontroller

```
:NoAlarm
  SkipIfAlarm          ; If the alarm is not triggered,
    jmp :CheckButtons ; check the buttons
  SkipIfModeIsNotDown ; If the alarm is triggered,
    jmp :StopAlarm    ; it can be turned off with
  SkipIfSetIsNotDown  ; any of the buttons
    jmp :StopAlarm
  jmp MainLoop

:StopAlarm             ; Turn off the alarm
  ClrAlarm            ; Clear the flag, and wait
  call WaitReleaseBoth ; until both buttons are released
  jmp MainLoop

:CheckButtons          ; Read the buttons
  SkipIfModeIsNotDown ; If the "Mode" button is pressed,
    inc State         ; set state to "Set Clock"
  mov w, #2
  SkipIfSetIsNotDown  ; If the "Set" button is pressed,
    mov State, w      ; set state to "Set Alarm"
  jmp MainLoop

; ** Handle state "Set Clock"
;
SetClock
  mov w, SubState     ; Jump table for sub-states
  jmp pc+w
  jmp InitSetClock    ; Do the initializations
  jmp MarkDigit       ; Display the decimal point
  jmp CheckButtons    ; Read the buttons
  jmp NextDigit       ; Select next digit

; ** Handle state "Set Alarm"
;
SetAlarmTime
  mov w, SubState     ; Jump table for sub-states
  jmp pc+w
  jmp InitSetAlarm    ; Do the initializations
  jmp MarkDigit       ; Display the decimal point
  jmp CheckButtons    ; Read the buttons
  jmp NextDigit       ; Select next digit

; ** Initializations for "Set Clock"
;
InitSetClock
  SettingClock        ; Set the mode flag
  bank Timers
  BlinkOn             ; Enable blink in the display VP
  bank Leds
  clrb Digits+1..7   ; Clear the decimal point in the
                    ; 10 seconds buffer as it might
                    ; be set when alarm is enabled
InitCommon
  clr Ix              ; Index = 0
  inc SubState        ; Next sub-state: "MarkDigit"
  jmp MainLoop
```

```

; ** Initializations for "Set Alarm"
;
;
InitSetAlarm
  SettingAlarm      ; Set the mode flag
  ClrAlarm          ; Disable the alarm
  mov  Ix, #5        ; Copy the saved alarm time to
; Copy              ; the display buffer
  mov  w, #Alarm
  add  w, Ix
  mov  fsr, w
  mov  Temp, ind
  mov  w, #Digits
  add  w, Ix
  mov  fsr, w
  mov  ind, Temp
  dec  Ix
  sb   Ix.7
  jmp  :Copy
  call WaitReleaseS ; Wait until the "Set" button is
;                               ; released
  jmp  InitCommon   ; Continue with common instructions

; ** Turn the decimal point on
;
;
MarkDigit
  mov  w, #Digits      ; Indirectly address the current
  add  w, Ix           ; digit, and set the decimal
  mov  fsr, w          ; point flag
  setb ind.7
  call WaitReleaseM   ; Wait until the "Mode" key is
;                               ; released
  inc  SubState        ; Next sub-state: CheckButtons
  jmp  MainLoop

; ** Read the buttons, and increment the current digit
;
;
CheckButtons
  SkipIfModeIsNotDown ; If "Mode" button pressed,
  inc  SubState        ; next sub-state: NextDigit
  SkipIfSetIsDown     ; If "Set" button pressed,
  jmp  MainLoop
  mov  w, #Digits      ; increment current digit, so
  add  w, Ix           ; indirectly address that digit,
  mov  fsr, w          ;
  mov  Temp, ind       ; copy it to Temp,
  clrb Temp.7         ; clear the decimal point flag,
  inc  Temp            ; and increment it

  mov  w, Ix           ; Depending on the current digit,
  jmp  pc+w           ; check for the maximum value:
  jmp  :Max9          ; 0..9 (1 seconds)
  jmp  :Max5          ; 0..5 (10 seconds)
  jmp  :Max9          ; 0..9 (1 minutes)

```

Programming the SX Microcontroller

```
    jmp :Max5           ; 0..5 (10 minutes)
    jmp :Max9           ; 0..9 (1 hours)
    jmp :Max2_1        ; 0..1 or 0..2 (10 hours)
:Max9
    mov w, #10          ; If digit > 9,
    mov w, Temp-w
    snz
        clr Temp       ; digit = 0
    jmp :Continue
:Max5
    mov w, #6           ; If digit > 5,
    mov w, Temp-w
    snz
        clr Temp       ; digit = 0
    jmp :Continue
:Max2_1
    dec fsr             ; If 1 hours <= 4,
    mov w, #4
    mov w, ind-w
    mov w, #3           ; 10 hours < 3 are valid,
    snc                 ; else 10 hours < 2
                       ; are valid only
        mov w, #2
    inc fsr
    mov w, Temp-w      ; If digit > 2 or 1
    snz
        clr Temp ; digit = 0

:Continue
    setb Temp.7        ; Set the decimal point flag
    mov ind, Temp      ; Copy the new value to the display
                       ; buffer, and
    call WaitReleaseS ; wait for "Set" button released
    jmp MainLoop

; ** Select next digit, and terminate the "Set???" mode if necessary
;
NextDigit
    mov w, #Digits     ; Indirectly address the current
    add w, Ix          ; digit, and
    mov fsr, w         ;
    clrb ind.7         ; clear the decimal point flag
    mov SubState, #1  ; Next sub-state is: MarkDigit
    mov w, #6
    inc Ix             ; Set index to next digit
    mov w, Ix-w       ; If Ix < 6,
    sz
        jmp MainLoop  ; enter MarkDigit, else
                       ; terminate the "Set???" mode

; ** When the "Set" button is pressed while the "Mode" button is held
; down, copy the display digits to the clock registers, or to the
; alarm time buffer (depending on the "Set???" mode).
;
;
    clr Ix             ; Ix is used as the "copy" flag
                       ; here, if Ix = 0, don't copy
```

```

    mov w, #5 ; "Prepare" w for "mov Ix, w"
: WaitRelease ; Wait for button release(s)
  SkipIfSetIsNotDown ; If "Set" is pressed, set
    mov Ix, w ; Ix = 5
  SkipIfModeIsUp ; If "Mode" is still held,
    jmp : WaitRelease ; continue waiting for release,
  call WaitReleaseBoth ; else wait until all buttons
    ; are released
  sb Ix.0 ; If Ix = 0, don't copy
    jmp : ExitSet

: Copy ; Copy the display buffer to the
  mov w, #Digits
  add w, Ix
  mov fsr, w
  mov Temp, ind
  mov w, #Sec ; Timer registers
  SkipIfSettingClock ; or the
    mov w, #Alarm ; alarm time buffer
  add w, Ix
  mov fsr, w
  mov ind, Temp
  dec Ix
  sb Ix.7
    jmp : Copy
  SkipIfSettingClock ; If an alarm time was set,
    ArmAlarm ; enable the alarm

: ExitSet
  BlinkOff ; Turn off blinking
  clr State ; Reset state and
  clr SubState ; sub-state
  jmp MainLoop ; Repeat the main loop forever...

```

The Timer VP was taken from the chapter “A Clock Timer – an Example” without modifications, so there is no need to discuss the details again here.

The 7-Segment VP has one additional feature: If bit 7 in one of the **Digits** registers is set, the decimal point for this digit will be turned on, and if the **Blink** flag is also set, the decimal point will blink. Here are the new instructions:

```

    and w, #%01111111 ; Mask bit 7 (the blink id)
    call Decode ; 7-seg pattern to W
    clr rc ; Turn off the current display
    mov rb, w ; Output line data
    SkipIfBlink ; If dots shall blink,
      jmp : DotOn
    bank Timers ;
      "borrow" from TSec, and
    movb c, TSec.0 ; save bit 0 in C
    bank Leds
    sc ; If TSec.0 is set, "On Phase",

```

Programming the SX Microcontroller

```
    jmp :NoBlink          ; else "Off Phase" for blinking
:DotOn
    bank Leds             ; Turn on if bit 7 is set in
    snb ind. 7           ; the digit's data storage
    clrb rb. 2
:NoBlink
```

Because now bit 7 of a **Digits** register may be set, it is important to clear that bit before calling the **Decode** subroutine to avoid a jump into “Nowhere Land”.

After **:DotOn**, bit 7 of the current **Digits** register is checked, and if this is the case, the **DotOn** macro is invoked to turn the decimal point row on.

If you use port assignments that differ from the stopwatch schematic, you need to change that macro.

If the **Blink** flag is set, **:DotOn** will only be reached, when bit 0 in **TSec** is currently set to obtain a blinking decimal point for the “Set clock” state.

In addition, the 7-segment VP does the job of sounding the buzzer in case of an alarm:

```
SkipIfAlarm              ; If alarm is triggered,
    jmp :NoBeep          ;
    bank Timers          ;
    snb Sec. 0           ; send a 500 Hz signal to the
    jmp :NoBeep          ; buzzer every other second
    movb Buzzer, MSec. 0
    : NoBeep
```

When no alarm is active, the program continues at **:NoBeep**, otherwise bit 0 of the seconds timer (**Sec. 0**) is used to turn the intermittent buzzer sound on or off. If the bit is set, the current state of **Msec. 0** is sent to the buzzer output to generate a 500 Hz sound.

*If you use port assignments that differ from the stopwatch schematic, you need to change the definition for **Buzzer**.*

The mainline program first clears the data memory, initializes the ports and the column mask **Column** next, enables the RTCC interrupt, and then enters into the main program loop.

This loop begins with the main state selection for the state engine:

```
Main
    mov w, State          ; Jump table for state engine
    jmp pc+w
    jmp Clock
    jmp SetClock
    jmp SetAlarmTime
```

The “Clock” state first updates the time display, and then checks if an alarm is enabled. In this case, the instructions

```

setb Digits.7 ; turn on the decimal points in
setb Digits+1.7 ; the seconds digits

```

turn on the decimal points in the seconds digits, and the current time is compared against the alarm time.

Next, the program checks if an alarm has been released, and in this case, both buttons are checked for the “button down” state that turns off the alarm. If no button is pressed, this check is repeated each time the main loop is executed until the user finally wakes up, and hits one of the buttons.

When no alarm is active, the buttons are checked “regularly”, and if the user has pressed one of the buttons, the main state is changed to **SetClock**, or to **SetAlarm**.

The **SetClock** mode uses another jump table to select one of the sub-states:

```

SetClock
  mov w, SubState ; Jump table for sub-states
  jmp pc+w
  jmp InitSetClock ; Do the initializations
  jmp MarkDigit ; Display the decimal point
  jmp CheckButtons ; Read the buttons
  jmp NextDigit ; Select next digit

```

The first sub-state is **InitSetClock** where the necessary flags are being set, variables are initialized, and the blink mode for decimal points is turned on before activating the next sub-state **MarkDigit**. When the main loop is executed the next time, the instructions

```

MarkDigit
  mov w, #Digits ; Indirectly address the current
  add w, Ix ; digit, and set the decimal
  mov fsr, w ; point flag
  setb ind.7
  call WaitReleaseM ; Wait until the "Mode" key is
  ; released
  inc SubState ; Next sub-state: CheckButtons
  jmp Main

```

will be executed. Here, it is important to wait for the release of the “Mode” button first, as it is most likely that it is held down for a while, or possibly bounces. This makes sure that the next sub-state **CheckButtons** will not be activated before the button is released as this state would immediately react on the still pressed “Mode” button.

Other than in the stopwatch program, waiting for button releases is handled by a subroutine call here, i.e. the program is “on hold” until the user decides to release a button. This is fine because it is not necessary here to update the time display while waiting for a button release.

The next sub-state **CheckButtons** takes care of controlling the digits entry.

```

CheckButtons
  SkipIfModeIsNotDown ; If "Mode" button pressed,

```

Programming the SX Microcontroller

```
    inc SubState           ; next sub-state: NextDigit
    SkipIfSetIsDown      ; If "Set" button pressed,
    jmp Main
    mov w, #Digits       ; increment current digit, so
    add w, Ix            ; indirectly address that digit,
    mov fsr, w           ;
    mov Temp, ind        ; copy it to Temp,
    clrb Temp. 7         ; clear the decimal point flag,
    inc Temp             ; and increment it
```

When the user presses the “Mode” button, the sub-state is incremented to **NextDigit** in order to make the next display digit active for modification. When the user hits the “Set” button instead, the contents of the current digit will be incremented that is stored in the **Temp** variable.

Depending on the current digit that is subject to be changed, different maximum values are allowed before the digit is reset to zero again:

For the one-seconds and one-minutes all values 0..9 are valid, but the ten-seconds and ten-minutes digits shall allow values 0..5 only. The one-hour digit can also accept all values from 0 through 9, while the ten-hour digit is a bit “tricky”:

If the one-hour digit contains a value less than 4, the maximum value is 2 for the hours 20, 21, 22, and 23, but if the one-hour digit is above 3, the ten-hours digit may only accept values of 0 and 1 for the hours 00 up to 19. The following instructions take care of this:

```
    mov w, Ix             ; Depending on the current digit,
    jmp pc+w             ; check for the maximum value:
    jmp :Max9            ; 0..9 (1 seconds)
    jmp :Max5            ; 0..5 (10 seconds)
    jmp :Max9            ; 0..9 (1 minutes)
    jmp :Max5            ; 0..5 (10 minutes)
    jmp :Max9            ; 0..9 (1 hours)
    jmp :Max2_1         ; 0..1 or 0..2 (10 hours)
:Max9
    mov w, #10           ; If digit > 9,
    mov w, Temp-w
    snz
    clr Temp            ; digit = 0
    jmp :Continue
:Max5
    mov w, #6           ; If digit > 5,
    mov w, Temp-w
    snz
    clr Temp            ; digit = 0
    jmp :Continue
:Max2_1
    dec fsr             ; If 1 hours <= 4,
    mov w, #4
    mov w, ind-w
    mov w, #3           ; 10 hours < 3 are valid,
    snc                 ; else 10 hours < 2
    mov w, #2           ; are valid only
    inc fsr
```

```

mov w, Temp-w           ; If digit > 2 or 1
snz                     ;
  clr Temp              ; digit = 0

```

Finally, the decimal point flag must be restored and the new value needs to be copied to the **Digits** buffer before waiting for the release of the "Set" button. Then the main loop is executed again without changing the sub-state in order to allow more changes of the current digit.

```

: Continue
  setb Temp.7           ; Set the decimal point flag
  mov ind, Temp         ; Copy the new value to the display
                        ; buffer, and
  call WaitReleaseS    ; wait for "Set" button released
  jmp Main

```

In case the user has hit the "Mode" button instead of the "Set" button, **NextDigit** is the next sub-state that usually selects the next display digit for modification before returning to the **CheckButtons** sub-state:

```

NextDigit
  mov w, #Digits       ; Indirectly address the current
  add w, Ix             ; digit, and
  mov fsr, w           ;
  clrb ind.7           ; clear the decimal point flag
  mov SubState, #1    ; Next sub-state is: MarkDigit
  mov w, #6
  inc Ix               ; Set index to next digit
  mov w, Ix-w         ; If Ix < 6,
  sz
  jmp Main            ; enter MarkDigit, else
                    ; terminate the "Set???" mode

```

When the user hits the "Mode" button while the 10-hours digit is active for modification, things are different. In this case, the time entry is finished, and in case the "Set" button was pressed with "Mode" still down, the entered time must be copied either into the alarm time buffer, or into the timer registers, depending on the current main state.

```

  clr Ix               ; Ix is used as the "copy" flag
                        ; here, if Ix = 0, don't copy
  mov w, #5           ; "Prepare" w for "mov Ix, w"

: WaitRelease         ; Wait for button release(s)
  SkipIfSetIsNotDown ; If "Set" is pressed, set
  mov Ix, w           ; Ix = 5
  SkipIfModeIsUp     ; If "Mode" is still held,
  jmp :WaitRelease   ; continue waiting for release,
  call WaitReleaseBoth ; else wait until all buttons
                        ; are released
  sb Ix.0            ; If Ix = 0, don't copy
  jmp :ExitSet

: Copy                ; Copy the display buffer to the
  mov w, #Digits

```

Programming the SX Microcontroller

```
add w, Ix
mov fsr, w
mov Temp, ind
mov w, #Sec ; Timer registers
SkipIfSettingClock ; or the
    mov w, #Alarm ; alarm time buffer
add w, Ix
mov fsr, w
mov ind, Temp
dec Ix
sb Ix.7
    jmp :Copy
SkipIfSettingClock ; If an alarm time was set,
    ArmAlarm ; enable the alarm

:ExitSet
    BlinkOff ; Turn off blinking
    clr State ; Reset state and
    clr SubState ; sub-state
    jmp Main ; Repeat the main loop forever...
```

The last instructions following **:ExitSet** turn off the **Blink** flag, and clear the status values in order to activate the default “Clock” state when the main loop is executed next time.

The major state **SetAlarmTime** is identical to the **SetClock** state with the exception of the initialization state. For all other sub-states, the same sub-state handlers can be used for both major states, and because the **NextDigit** sub-state handler uses the **SkipIfSettingClock** to determine whether the new time entered shall be copied into the alarm time buffer or into the timer registers, this sub-mode handler can also be used for both major states. As mentioned before, the initialization handler for **SetAlarmTime** is different from the one required for the **SetClock** major state:

```
InitSetAlarm
    SettingAlarm ; Set the mode flag
    ClrAlarm ; Disable the alarm
    mov Ix, #5 ; Copy the saved alarm time to
:Copy ; the display buffer
    mov w, #Alarm
    add w, Ix
    mov fsr, w
    mov Temp, ind
    mov w, #Digits
    add w, Ix
    mov fsr, w
    mov ind, Temp
    dec Ix
    sb Ix.7
    jmp :Copy
    call WaitReleaseS ; Wait until the "Set" button is
    ; released
    jmp InitCommon ; Continue with common instructions
```

The **SettingAlarm** macro sets a flag to “inform” the copy routine where to copy the time data.

The currently saved alarm time is copied to the display buffer, so that the alarm time entered last can be modified instead of starting with 00:00:00. The remaining actions are the same for both “Set” states, therefore the **jmp InitCommon** branches to the common instructions of the **SetClock** sub-state handler.

4.7.1 When the Clock is Wrong...

Caused by tolerances of the crystal or ceramic resonator that controls the system clock, it may happen that the clock is too slow or too fast.

Instead of exchanging the crystal or ceramic resonator, a simple software solution can solve that problem, and for this example, let’s assume that the clock is slow by 2 Seconds per day:

2 Seconds/Day at a system clock of 50 MHz are equivalent to an error of $2s/20 \cdot 10^9 = 100 \cdot 10^6$ clock cycles/day.

One day has $86.4 \cdot 10^3$ Seconds, and because the ISR gets called every $4 \mu s$, the error is equal to $86.4 \cdot 10^3 / 4 \cdot 10^{-6} = 21,6 \cdot 10^9$ ISR calls/day.

If we divide the number of ISR calls per day by the clock cycles per day, we yield $21.6 \cdot 10^9 / 100 \cdot 10^6 = 216$, i.e. at each 216th ISR call, the ISR must be called one clock cycle earlier than usual to correct the error.

As this value cannot be derived from the time counters in the Clock VP, it makes sense to add a new timer counter for this error correction that keeps track of how often the ISR has been called:

```
org      $50
Timers  = $          ; Timer bank
CorrT   ds 1        ; Counter for error correction
us4     ds 1        ; 4us counter
```

To initialize this counter, and to define the correction factor, it makes sense to define constants at the beginning of the program code:

```
Beeper   = rc. 7
CorrInit = 216
ISRCorr  = 1
```

The following macros are useful as well:

```
SetAdjust MACRO
    setb Fl ags. 5
ENDM
```

```
ClrAdjust MACRO
    clrb Fl ags. 5
ENDM
```

```
SkipIfNoAdjust MACRO
    snb Fl ags. 5
```

Programming the SX Microcontroller

ENDM

To first initialize **CorrT**, add the following instructions in the mainline program:

```
bank Timers
mov CorrT, #CorrInit
bank Leds
mov Column, #1           ; Initialize column mask
```

Modify the first instructions of the ISR as follows:

```
Bank Timers
mov w, #CorrInit
dec CorrT
snz
    mov CorrT, w
snz
    SetAdjust
mov w, #250
```

The ISR routine “exit” should be enhanced by the instructions

```
: ISRExit
mov w, -200
bank Timers
SkipIfNoAdjust
    mov w, #(ISRCorr - 200)
ClrAdjust
reti w
```

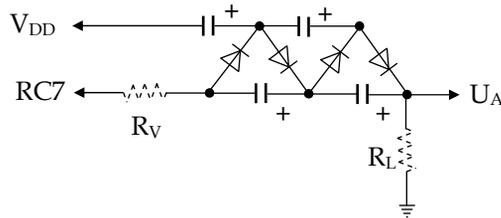
When the **CorrT** counter underflows in the ISR, the **SetAdjust** macro is called to set the **Adjust flag**. In this case, the value defined for **ISRCorr** is added to the default **reti w** return value of -200, i.e. in our example, the ISR will be called after 199 clock cycles now. Because the **ClrAdjust macro** is called before exiting the ISR, subsequent ISR calls will again return -200 in w before the **Adjust** flag is set anew after the ISR has been called **CorrInit** times.

In case the clock is too fast, the ISR call period must be extended from time to time, i.e. in this case, **ISRCorr** must be defined to represent a negative value.

If you like, you may enhance the program so that the clock speed correction can be entered by the user instead of using the “hard coded” approach, as shown before.

4.8 Voltage Converters

4.8.1 A Simple Voltage Converter



In some applications, a voltage higher than V_{DD} is required for external components. When only little power is required, and a free port pin is available at the SX, this voltage can be generated using a Villard voltage multiplier circuit. To drive the voltage multiplier, the SX controller must generate a square wave signal with a 50% duty cycle at the port pin.

The resistor R_V might be necessary to limit the maximum current drawn from the SX output. A value of $180\ \Omega$ limits the current to less than 30 mA.

The size of the capacitors depends on the frequency of the square wave signal. They should be large enough to filter out that signal.

In the test circuit, four silicon diodes, and capacitors of 33 nF each were used. Without load, the resulting voltage U_A was approximately 13 V, and with a load of 18 k Ω , the voltage dropped down to about 12 V, i.e. the output current was approx. 0.6 mA.

This is the simple program to drive the circuit:

```

; =====
; Programming the SX Microcontroller
; APP017.SRC
; =====
include "Setup28.inc"
RESET   Main

org     $08
Timer   ds 1

org     $000

ISR
  xor rc, #%10000000          ; Toggle rc.7
  mov  w, #-100              ; Call the ISR every 1 us

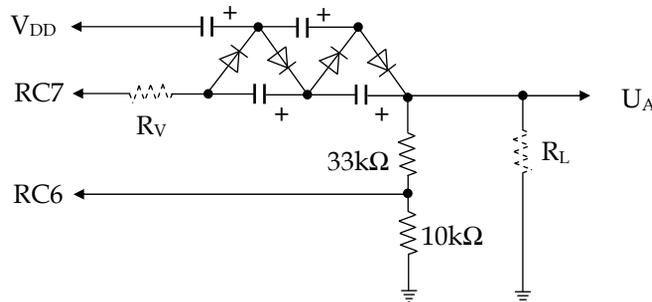
```

Programming the SX Microcontroller

```
retiw
org    $100
Main
mov    Timer, #250
clr    rc
mov    !rc, #%01111111
mov    !option, #%10011111    ; Enable the RTCC interrupt
Loop
jmp    Loop
```

4.8.2 A Regulated Voltage Converter

If another port pin is available, the SX can be used to stabilize the output voltage to a certain extend. The schematic shows the additional components that are required:



Here, we use a voltage divider ($33\text{ k}\Omega / 10\text{ k}\Omega$) to feed part of the output voltage into port pin RC6 that is configured as a CMOS input.

To drive the voltage multiplier, we now use a PWM VP in order to control the output voltage by changing the pulse width of the square wave signal.

As long as the voltage at RC6 is below $V_{DD}/2$ the port bit reads 0, and at voltages above that value, the port bit reads 1.

Using this information, the pulse width is increased or decreased, and this logic builds a closed loop that stabilizes the output voltage (within certain limits) to a value that depends on the ratio of the voltage divider.

Using the values above, the test circuit generated an output voltage of 10 V that changed by less than 0.1% when an output current of 2 mA was drawn.

This is the program:

```

; =====
; Programming the SX Microcontroller
; APP018. SRC
; =====
include "Setup28.inc"
RESET    Main

LVL      = $0d
TRIS    = $0f
SensePin = rc.6                ; Port pin for sense voltage

org      $08
PwmAcc   ds 1                  ; Current PWM value
PwmVal   ds 1                  ; Contents defines pulse width
rcBuff   ds 1                  ; Buffer for port C output

org      $000

Stabilization
    sb SensePin                ; Read the divided voltage
    jmp :TooLow                 ; -> too low
    dec PwmVal                  ; Too high, reduce pulse width
    snz                                     ; If PwmVal is 0, force it
    inc PwmVal                  ; to 1
    jmp PWM

:TooLow                                     ; Voltage too low,
    inc PwmVal                  ; increase pulse width
    clrb PwmVal.7               ; Don't allow values above 127 for
    ; a maximum duty cycle of 50%

PWM
    clrb rcBuff.7               ; Clear the PWM bit in advance
    add PwmAcc, PwmVal          ; Set current PWM value
    snc                                     ; Need to toggle the output ?
    setb rcBuff.7

    mov rc, rcBuff              ; Output port data
    mov w, #-100
    reti

org      $100

Main
    clr PwmVal
    clr PwmAcc
    mode LVL
    mov !rc, #%10111111         ; rc.6 is CMOS input
    mode TRIS
    mov !rc, #%01111111         ; rc.7 is PWM output
    mov !option, #%10011111     ; Enable RTCC interrupt

Loop    ; Nothing to do here
    jmp Loop                    ; for now...

```

Programming the SX Microcontroller

To generate the PWM signal, we use the PWM VP that we already have discussed.

In the Stabilizer VP, we test **rc. 6**. If this bit reads 0, the output voltage is below the desired value. In this case, we need to increase the pulse width. On the other hand, if **rc. 6** reads 1, the output voltage is too high, so that we must reduce the pulse width.

The maximum output voltage can be achieved when the square wave signal has a duty cycle of 50%. Therefore, it only makes sense to change the duty cycle in the range from 1% up to 50%, and this is why **PWMVal** is limited to that range in the program.

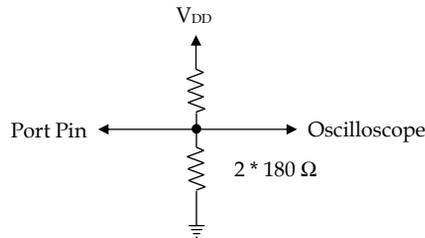
4.9 Testing Port Outputs

While “bread boarding” SX applications, it may happen that you short-circuit a port pin that currently is configured as an output with high level by accident, or that you connect two “offending” outputs (one on high level, and the other on low level).

During many experiments with the SX that were done to prepare this book, the outputs always survived such “stress situations” (it even “survived” a wrong power supply polarity).

Nevertheless, there are chances to “kill” an output, and when you don’t notice this, it is most likely that you blame the software first for the problem. After some hours of debugging, you might consider exchanging the SX, and – voila – now the system works as expected.

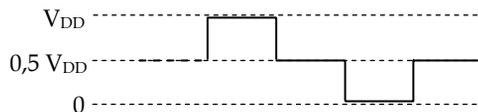
This simple test circuit helps to find out if the port outputs are working properly:



When a port pin is configured as an input, i.e. Hi-Z, the two resistors make up a voltage divider, and the oscilloscope input “sees” $V_{DD}/2$.

When the port pin is configured as output, the oscilloscope input is either pulled up to almost V_{DD} when the output has high level, or pulled down close to 0 Volts when the output has low level.

When a port pin is set to the various states in the order high – hi-Z – low – hi-Z, the oscilloscope will display the following signal:



The program below sends this pattern to all port pins from Port A through Port C:

```
;
; =====
; Programming the SX Microcontroller
; APP019. SRC
; =====
```

Programming the SX Microcontroller

```
include "Setup28.inc"
RESET    Main

org      $08
PortId   ds 1           ; Address of the port under test
PinMask  ds 1           ; Mask for port pin under test
Timer    ds 2           ; Delay counter
Temp     ds 1           ; Temporary storage
TrisMask ds 1           ; Mask for TRIS register

; ** Main program *****
;
Main
    mov PortId,    #5           ; Start the test at port ra

:PortLoop
    mov PinMask,  #%00000001 ; Start the test with pin 0

:PinLoop
    mov fsr, PortId           ; Indirectly address the port
    mov w, PinMask           ; register, and write
    mov ind, w               ; the pin mask
    not w                    ; Invert the pin mask for the
    mov TrisMask, w          ; TRIS register and save it
    call SetDir              ; Set port as output, high level
    ; and delay
    call SetInput           ; Set port as input,
    ; and delay
    mov fsr, PortId         ; Indirectly address the port
    mov w, /PinMask         ; register, and write the
    mov ind, w              ; inverted pin mask
    mov TrisMask, w         ; Save inverted mask in the
    ; TRIS register
    call SetDir             ; Set port as output, low level
    ; and delay
    call SetInput           ; Set port as input,
    ; and delay
    clc
    rl PinMask              ; Rotate left the
    sc                      ; after 8 RLs, the C flag is set
    jmp :PinLoop            ; Not yet 8 tests, select next port
    ; pin for test in this port
    inc PortId              ; Next port
    sb PortId, 3            ; PortId > 7 ?
    jmp :PortLoop          ; If not, test this port,
    ; else repeat the test
    jmp Main                ; for port ra

; ** Subroutine sets all pins of the port <PortId> to inputs (hi-Z)
;
SetInput
    mov TrisMask, #$ff

; ** Subroutine sets the !R? register of port <PortId> to the value
; contained in <TrisMask>
```

```

;
; Call: : PortId = port address, TrisMask = data
; Uses: Temp
;
SetDir
mov Temp, PortId      ; PortId -> Temp
sub Temp, #4          ; Make PortId 1-based
mov w, TrisMask      ; TrisMask -> w
dec Temp
snz ; If Temp = 0,
    mov !ra, w        ; Port A, else
dec Temp
snz ; if Temp = 0,
    mov !rb, w        ; Port B, else
dec Temp
snz ; if Temp = 0,
    mov !rc, w        ; Port C
mov Timer+1, #20      ; Time delay
:Loop
decsz Timer
    jmp :Loop
decsz Timer+1
    jmp :Loop
ret

```

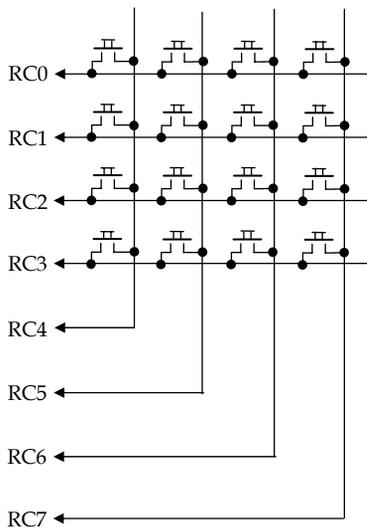
This program can be used to test the SX 18/20/28 controllers. if you want to test the SX 48/52 devices, you can easily enhance the program to address ports RD and RE (addresses \$08 and \$09) as well.

It is an idea to build a special probe with the two 180 Ω resistors built in, that is connected to the oscilloscope through a coaxial cable, and connected to V_{DD} and V_{SS} via two wires with two plugs or crocodile clips at the other end.

4.10 Reading Keyboards

In the examples before, we have only used two pushbuttons, each with a make and a brake contact to make de-bouncing easy. In many cases, it is necessary to read a lot more keys, and it is not possible to assign two port pins for each key.

Similar to scanning a 7-Segment LED matrix, the keys to be read can be arranged in a matrix:



The diagram shows a matrix of four rows and four columns. With an SX 28, you can extend the matrix up to 8 by 8 to read 64 keys if the remaining four I/O pins are enough to do the rest of the job, like serial communications, etc.

When designing a system, it is important that hardware and software designers work together. Unfortunately, there are cases that the “Hardware Specialist” wants to assign the port pins in a way that makes the PCB design easier, but if the port pins are arbitrarily assigned, the software design may become a nightmare.

The schematic shows that the column lines and the row lines are both assigned to Port C pins. On the first glance, it might be better to have one group of lines assigned to another port (e.g. the columns to Port B). But this means that 4 bits of Port B, and 4 bits of Port C would be assigned to the key matrix where the remaining 4 bits on each port are free for other purposes, and in order to “sort out” the right bits, it is necessary to mask out the other bits for both ports.

If you use one port nibble for inputs, and the other nibble for outputs, it often makes sense to use the lower nibble for the inputs, and the higher one for the outputs as the input data are directly available as binary value when you mask out the higher nibble.

To control the column lines that are connected to outputs, one zero bit must “rotate” through the higher nibble, a task that can easily be achieved.

Usual pushbutton keys require de-bouncing in most cases, and therefore, some timing considerations are in order:

To avoid a delay between pressing a key and the action following on that key press, it is important to read all the keys often enough. A save value is to activate each column every 1 to 10 ms.

To de-bounce a key, a delay between 20 to 40 ms between the first registered key press and the actual reading of the key is sufficient for most button types. There are two options how the software can handle the de-bouncing: Either it pauses further key scans until the de-bouncing for the registered key is finished, or it continues scanning the other keys in the meantime.

The first option is easier to program because the second option requires separate de-bounce timer counters for each single key.

De-bouncing is necessary when a key is pressed, but also when the key is released again.

Another design question is if it is sufficient to inform the “rest of the world” about a key-down event only, or if it is also important to report a “key up” event as well. This might be necessary, if the time between key-down, and key-up plays a role, e.g. to initiate an auto-repeat.

In addition, it is important how to react if two or more keys are pressed at the same time. Because most people have 10 fingers, the following rule is an extent to Murphy’s Law:



“If a device has more than one key, the user **will** press more than one key at a time!”

First, this brings us to the method how to drive the column lines. If the program would pull one column line output to low, and all the other column lines to high level, it can happen that port outputs will be damaged. If, for example, the user would press the keys RC3/RC4, and RC3/RC5 at the same time with RC4 high, and RC5 low, these two offending outputs are connected via the two pressed buttons. (This is a good reason why you should build the output tester described before.)

To avoid this situation, all inactive columns should be set to the hi-Z state, and only the active column line should have low level.

Note that this requires pull-up resistors between V_{DD} and the row lines.

Programming the SX Microcontroller

Besides the fact that multiple pressed keys may cause short circuits, another question is of importance: How should the software deal with two or more keys pressed at the same time?

Actually, it is almost impossible that two or more keys are really pressed at the same time. Usually, there is a slight time difference between the key-presses. However, it is of course possible that the user presses and holds down two or more keys for a certain period.

The easiest solution is just to report the key that was recognized first, and to ignore all other keys until all keys have been released again. Especially for keyboards that are designed in a typewriter style, this is not acceptable in most cases because “fast fingers” tend to hit the next key before the previously pressed key is completely released.

As an example, if you press two keys on a PC keyboard at the same time, and hold them down, you will notice that both characters assigned to these keys are displayed, and that after a while, an auto-repeat starts for the character that was detected last. When you try to press more than two keys at the same time you will notice that to a certain number of keys the assigned characters are displayed in the order, the PC has recognized the single key-down events. On the PC used to write this book, this worked for up to five keys, but six keys held down together did not cause any screen output, and the system speaker generated a beep.

The feature to handle more than one key is called n-key rollover. When a system can handle just two overlapping key-presses, we can talk about a 2-key rollover behavior, and this is the minimum, a typewriter-style keyboard should be able to handle.

To support an n-key rollover (with say n being 5), it is important that no keystroke gets lost, and that the system maintains the correct order of the key-down events. For this purpose, a temporary storage with FIFO (first in, first out) characteristic is required (we will discuss a FIFO later in this book).

Let’s start with a simple program version first. Later, we will enhance this version by some additional features.

Because scanning of the keyboard matrix should be performed in fixed periods, it makes sense to use the SX’s RTCC interrupt in order to obtain a correct timing. In general, it is possible to execute the code that is required to scan and to decode the key matrix within the ISR if the number of available ISR clock cycles is large enough, but it often makes more sense to install a timer VP in the ISR, and to handle the keyboard tasks in the mainline program. This leaves enough “room” in the ISR for other VPs, e.g. for an UART that sends the key events to another device.

4.10.1 Scanning a Key Matrix, First Version

```

; =====
; Programming the SX Microcontroller
; APP020.SRC
; =====
include "Setup28.inc"
RESET    Main

TRIS     = $0f
PLP      = $0e

TickOn   MACRO
    setb Flags.0                ; Macro to set the tick flag
ENDM

TickOff  MACRO
    clrb Flags.0                ; Macro to clear the tick flag
ENDM

SkipIfTick MACRO
    sb Flags.0
ENDM

org      $08
Flags   ds 1                ; Register for various flags

org      $30
Keys    equ $
Column  ds 1                ; Current scan column
DebCnt  ds 1                ; De-bounce counter
KeyId   ds 1                ; Id of pressed key

org      $50
Timers  equ $
us5     ds 1                ; Counter for 5 us
Msec    ds 1                ; Counter for 1 ms

org      $000

; ** Timer-VP *****
;
Clock
Bank Timers
mov     w, #250                ; 5us * 250 = 1,25 ms
dec     us5
snz
    mov us5, w
mov     w, #2
snz
    dec Msec
snz
    mov MSec, w
snz

```

Programming the SX Microcontroller

```
    Tick0n    ; Every 2,5 ms

    mov    w, #-250                ; Call ISR every 5 us
    reti w

; ** Subroutine reads the Column variable and returns the column mask
;   in w.
;
;
Col2Mask
    mov w, Column
    jmp pc+w
    retw %11101111
    retw %11011111
    retw %10111111
    retw %01111111

; ** This subroutine would send the key code in "real live"
;
;
SendKey
    mov w, KeyId                ; This is just an anchor to set
                                ; a breakpoint
    ret

org    $100

; ** Mainline program *****
;
Main
include "Clr2x.inc"

; ** Initialize the data ports
;
;
mode PLP
mov !rc, #%11110000            ; Pull-up at rc.3...0
mode TRIS
mov !rc, #$ff                ; All lines Hi-Z for now
clr rc                        ; pre-initialize rc with all zeros
bank Keys
clr Column
mov !option, #%10011111      ; Enable the RTCC interrupt

; ** Main program loop
;
;
Loop
    SkipIfTick                ; Wait for the 2.5 ms tick
    jmp Loop

    TickOff                    ; Clear the tick flag
    inc Column                ; Next column, but don't
    clrb Column.2             ; allow values > 3
    call Col2Mask             ; Get the column mask,
    mov !rc, w                ; and set the Port C TRIS register
    mov w, /rc                ; Complement the row data, and
    and w, #$0f                ; mask out the rows
```

```

SZ ; 0, if no keys are pressed in that
    ; column
    jmp :Key ; Key(s) pressed in the column,
            ; go, and decode the key
    jmp Loop ; Continue waiting for a tick and
            ; a key

; ** Decode a key
;
; Key
    mov KeyId, w ; Save column information
    mov DebCnt, #4 ; De-bounce = 4 * 5 ms = 20 ms

: Debounce
    SkipIfTick ; Wait for the 2,5 ms tick
    jmp :Debounce
    TickOff ; Clear the tick flag
    decsz DebCnt ; De-bounce counter - 1
    jmp :Debounce

    mov w, /rc ; Read, invert, and
    and w, #$0f ; mask out the rows again
    mov w, KeyId-w ; Same as before de-bouncing ?
    sz ; If yes, continue decoding,
    jmp Loop ; else, we had a bounce

; ** Convert row and column info into key id
;
    clc
    rr KeyId ; Convert 8, 4, 2, 1 to 4, 2, 1, 0
    mov w, #3
    snb KeyId.2; Convert 4 to 3 and end up in
    ; 3, 2, 1, 0
    mov KeyId, w

    clc
    rl KeyId ; KeyId = KeyId * 4 + Column
    rl KeyId
    or KeyId, Column

    call SendKey ; Send key Id

; ** Wait for key release
;
: WaitRelease
    mov DebCnt, #4 ; De-bounce time = 20 ms

: Rel Debounce
    SkipIfTick ; Wait for 2,5 ms tick
    jmp :RelDebounce
    TickOff ; Clear the tick flag

    mov w, /rc ; Read inverted row data, and
    and w, #$0f ; mask out the rows

```

Programming the SX Microcontroller

```
SZ
  jmp :WaitRelease           ; It's a bounce, de-bounce again
decsz DebCnt                 ; De-bouncing time (20 ms) ended?
  jmp :RelDebounce          ; No, keep on de-bouncing

  jmp Loop                   ; Wait for next key down
```

This first program version continues scanning the keyboard matrix until it recognizes a key down in a column. In this case, scanning is suspended until the recognized key has been de-bounced, decoded and sent via **SendKey**, and is finally released.

The **SendKey** subroutine in this sample program just contains one instruction so that a breakpoint can be set there – in a real application, this routine would send the id of a pressed key to the “outside world”.

Incrementing the column number is done at the beginning of the main loop **Loop**, to re-enter the loop from any place within the loop (when a key-bounce has been detected).

4.10.1.1 Decoding the Key Number

SendKey expects a number from 0 through 15 in **KeyId** that identifies the pressed key. The column code is contained in **Column** (0...3), but reading the row lines returns a value of 1, 2, 4, or 8 in case just one key is pressed in the active column. These numbers must be converted into 0, 1, 2, or 3 in order to be combined with the column number. When we rotate right the values 1, 2, 4, or 8 the result is 0, 1, 2, or 4, i.e. all values are already correctly adjusted except the 4. The instructions

```
mov w, #3
snb KeyId, 2
mov KeyId, w
```

provide the necessary adjustment.

Finally, this value is multiplied by 4, and the column number is added to yield the key id.

The table below shows the resulting values for **KeyId**:

	RC4	RC5	RC6	RC7
RC0	0	4	8	12
RC1	1	5	9	13
RC2	2	6	10	14
RC3	3	7	11	15

The method to decode the key rows shown above has one drawback because it will return wrong results when more than one key is pressed in one column. If, for example, the keys in rows RC1

and RC2 were held down at the same time when the column line RC4 is active, the decoding would result in 3. On the other hand, this can only happen if both keys were pressed within the 10 ms period while the other columns are scanned because otherwise this would be interpreted as a key bounce.

The change shown below makes sure that multiple key-presses are handled correctly:

```

; ** Convert row and column info into key id
;
mov DebCnt, #-1           ; DebCnt is used as temporary storage
                           ; here
:Decode
inc DebCnt                ; First row number (0)
rr KeyId                  ; Rotate row info ->
sc                         ; If C=1, a key is pressed in this row
    jmp :Decode

mov KeyId, DebCnt         ; DebCnt contains row number
clc
rl KeyId                  ; KeyId = KeyId * 4 + Column
rl KeyId
or KeyId, Column

```

To convert a column number into a bit pattern that is stored in the **rc** TRIS control register, we call the **Col2Mask** subroutine here, instead of rotating a register contents, as we did for the 7-segment display. The method used here adds some extra safety because the pattern is derived from the column number variable.

4.10.1.2 Initial “Quick Scan”

The previous version of the program keeps scanning the key matrix until a key-down is recognized, i.e. the instructions

```

Loop
SkipIfTick                ; Wait for the 2.5 ms tick
    jmp Loop

TickOff                    ; Clear the tick flag
inc Column                 ; Next column, but don't
clrb Column.2              ; allow values > 3
call Col2Mask              ; Get the column mask,
mov !rc, w                  ; and set the Port C TRIS register
mov w, /rc                  ; Complement the row data, and
and w, #$0f                ; mask out the rows
sz                           ; 0, if no keys are pressed in that
                           ; column
    jmp :Key                ; Key(s) pressed in the column,
                           ; go, and decode the key
jmp Loop                    ; Continue waiting for a tick and
                           ; a key

```

Programming the SX Microcontroller

and the subroutine **Col2Mask** are executed all the time, even if no key is pressed, wasting clock cycles.

To improve the program by a “Quick-Scan Mode”, the ISR could check if any key in any column is pressed, and “inform” the mainline program about that fact by setting a flag. To allow the ISR to do that check, all column lines must be pulled to low level. Now, when a key is down, no matter in what row or what column, reading the row port bits results in a value other than \$0f. Keeping the column lines “quiet” as long as no key is down also helps to reduce unnecessary noise.

4.10.2 Quick-Scan and 2-Key Rollover

The next program below uses the “Quick-Scan Mode”, and allows for a 2-key rollover. It also saves the last 16 key Ids in a buffer for debugging purposes.

```

; =====
; Programming the SX Microcontroller
; APP021. SRC
; =====
include "Setup28.inc"
RESET   Main

TRIS    = $0f
PLP     = $0e

TimerOn MACRO                ; Turn the timer on
    setb Flags. 0
ENDM

TimerOff MACRO               ; Turn the timer off
    clrb Flags. 0
ENDM

SkipIfTimeout MACRO         ; Skip if timer is done
    snb Flags. 0
ENDM

SkipIfTimer MACRO           ; Skip, if timer is active
    sb  Flags. 0
ENDM

QuickScanOn MACRO          ; Turn on Quick Scan
    setb Flags. 1
ENDM

QuickScanOff MACRO        ; Turn off Quick Scan
    clrb Flags. 1
ENDM

SkipIfQuickScan MACRO     ; Skip if Quick Scan active
    sb  Flags. 1
ENDM

```

```

SkipIfKeyDown MACRO ; Skip if key is pressed
    snb Flags.1
ENDM
org      $08
Flags   ds 1           ; Register for various flags
KeyId   ds 1           ; Id of pressed key
Ix      ds 1           ; Index for KeyId buffer

org      $30
Keys    equ $
Column  ds 1           ; Current scan column
Row     ds 1           ; Current scan row
RowMask ds 1           ; Mask for the row decoded last

org      $50
Timers  equ $
us5     ds 1           ; Counter for 5 us
Msec    ds 1           ; Counter for Milliseconds

org      $70
Buffer  ds 16          ; Buffer for key IDs (debug only)

org      $000

; ** Timer and Quick Scan *****
;
Clock
    SkipIfQuickScan ; If Quick Scan is off, go ahead
    jmp :Timer       ; with the Timer
    mov w, /rc       ; Read inverted row bits, and
    and w, #$0f      ; mask out the row lines
    sz               ; If a key is pressed,
    QuickScanOff    ; turn off Quick Scan

:Timer
    SkipIfTimer     ; If timer is off,
    jmp :ISRExit    ; no action
    Bank Timers
    mov w, #250     ; 5us * 250 = 1,25 ms
    dec us5
    snz
    mov us5, w
    mov w, #16      ; 1,25 ms * 16 = 20 ms
    snz
    dec Msec
    snz
    mov MSec, w
    snz
    TimerOff        ; After 20 ms

:ISRExit
    mov w, #-250    ; Call ISR every 5 us

```

Programming the SX Microcontroller

```
    reti w
; ** Subroutine reads the Column variable and returns the column mask
;   in w.
;
Col2Mask
    mov w, Column
    jmp pc+w
    retw %11101111
    retw %11011111
    retw %10111111
    retw %01111111
;
; ** This subroutine will send the recognized key IDs to some external
;   device in "real life". Here, we save 16 Key IDs in a buffer
;   for debugging purposes. As the buffer contains all zeros at
;   program start, we increment all key IDs by one, i.e. 1..16
;   are the "transformed" key IDs in order to make a difference
;   between an empty buffer register and the lowest key ID.
;
SendKey
    mov w, #Buffer          ; Buffer base address +
    add w, Ix               ; Index =
    mov fsr, w              ; indirect address
    mov ind, KeyId          ; Save the key ID, and
    inc ind                 ; increment it
    inc Ix                  ; Next buffer register index,
    clrb Ix.4               ; but not above 15
    bank Keys
    ret
;
org    $100
; ** Mainline program *****
;
Main
include "Clr2x.inc"
; ** Initialize the data ports
;
mode PLP
mov !rc, #%11110000        ; Pull-up at rc.3..0
mode TRIS
mov !rc, #$ff              ; All lines Hi-Z for now
clr rc                     ; pre-initialize rc with all zeros
bank Timers
mov us5, #250
mov MSec, #16
bank Keys
mov !option, #%10011111    ; Enable the RTCC interrupt
;
; ** Main program loop
;
```

```

Loop
  mov  !rc, #%00001111      ; All columns low
  QuickScanOn              ; Turn on Quick Scan, and
: WaitForKey                ; wait for a key
  SkipIfKeyDown
  jmp  :WaitForKey
  clr  Column              ; Start with column 0
: Scan
  call Col2Mask            ; Get the column mask,
  mov  !rc, w              ; and set the Port C TRIS register
  clr  KeyId               ; The contents of KeyId are not
                          ; important here, therefore, we
                          ; use it as a counter
: Delay
  decsz KeyId              ; Wait until signals are stable,
                          ; i.e. do a pre-de-bounce here
  jmp  :Delay
  mov  w, /rc              ; Complement the row data
  and  w, #$0f             ; and mask them out
  sz   ; 0 if no key is down in this col.
  jmp  :Key                ; A key is down, decode it
  inc  Column              ; Next column
  snb  Column.2            ; If column > 3, we have a bounce,
  jmp  Loop                ; therefore, no action
  jmp  :Scan

; ** De-bounce and decode the key
;
: Key
  mov  KeyId, w            ; Save row info
  TimerOn                  ; Turn the timer on
: Debounce
  SkipIfTimeout            ; Wait for the 20 ms tick
  jmp  :Debounce

  mov  w, /rc              ; Read inverted row info again,
  and  w, #$0f             ; and mask it out
  mov  w, KeyId-w          ; Still same value ?
  sz   ; If yes, go ahead, and decode it
  jmp  Loop                ; If no, we have a bounce

; ** Convert row and column info into the key ID
;
  mov  Row, #-1
: Decode
  inc  Row                 ; First row number (0)
  rr   KeyId               ; Rotate row info -> C
  sc   ; If C=1, a key is down in that row
  jmp  :Decode

  mov  KeyId, Row          ; Row contains row number
  clc
  rl   KeyId               ; KeyId = KeyId * 4 + Column
  rl   KeyId
  or   KeyId, Column

```

Programming the SX Microcontroller

```
inc Row          ; Set the bit in RowMask that
clr RowMask     ; corresponds to the detected
stc ; key
: SetupMask
rl RowMask
decsz Row
jmp : SetupMask

call SendKey    ; Send KeyId

; ** Wait for key release
;
: WaitRelease
TimerOn        ; Turn timer on
: WaitForTick
SkipIfTimeout  ; Wait for 20 ms tick
jmp : WaitForTick

mov w, /rc     ; Read inverted port bits, and
and w, RowMask ; mask the bit where the current
               ; key was detected
sz            ; If this key is still down,
jmp : WaitRelease ; continue waiting for release
jmp Loop     ; Wait for next key press, or
             ; decode the second key that is
             ; still down.
```

Here, the ISR usually does not perform any actions until the mainline program has not turned on the timer, or the Quick Scan mode.

When one of the two modes is active, they are automatically turned off again, when the timer is done, or a key down has been detected. The mainline program can test the “On” flags to find out when the ISR has turned the assigned mode off again.

In the main loop, the Quick Scan mode is turned on, and no other actions are taken, until the ISR “reports” a key down.

Then the key matrix is scanned as fast as possible after an initial delay for each column which is necessary to allow the signals on the matrix lines to become stable. If a pressed key has been detected, the program enables the timer, and waits until 20 ms have elapsed in order to de-bounce the key. If the key is still down after the de-bounce delay, it will be decoded, and **SendKey** is called to “send” the key ID.

To de-bounce the key release, the timer is enabled again for a 20 ms delay. This is repeated until the key that was detected last is no longer down.

For debugging and testing purposes, the SendKey subroutine in this program version now saves up to 16 Key IDs in the **Buffer** registers. To make a difference between an empty buffer register

containing \$00, and the lowest **KeyId** that would be \$00 as well, the key IDs are incremented once before they are stored in the buffer.

While waiting for a button release, we now do not wait until all row lines are on high level. Instead, we only test the row line that was low before due to the pressed key that was most recently detected. The **RowMask** variable contains the necessary information. If another key is down in the same column, it will be detected as soon as the first key is released. This is how a 2-key rollover is achieved here even if both keys are located in the same column.

In case you also want to send the button up information to the “rest of the world”, you can do this by adding two more instructions at the end of the main program loop:

```

jmp :WaitRelease           ; continue waiting for release
setb KeyId.7
call SendKey
jmp Loop                   ; Wait for next key press, or
                             ; decode the second key that is
                             ; still down.

```

Here, we set bit 7 in **KeyId**, and call **SendKey** again. **SendKey** must check this bit to determine what message it must send in a real application.

You can follow up the results with the debugger by clicking the “Poll” button after hitting some keys in the matrix. Note how the contents of bank \$70 changes in that case.



Because the key matrix is scanned at a very high speed when the system clock frequency is 50 MHz or above, it is necessary to add a time delay between activating a new column and reading the row lines. This allows the signals on the lines to become stable when using the internal weak pull-up resistors of about 40 kΩ.

You may use external pull-up resistors (2.2 kΩ or less) to decrease the settling time, but even then it is a good idea to insert a short delay between the **mov !rc, w** and **mov w, /rc** instructions, e.g. some **nop** instructions.

While testing the first version of this program, we did not insert that delay, and it took a while to find out why sometimes no key or the wrong key was detected in “Run” mode while in “Single Step” mode keys were never lost.

4.10.3 Need more Port Pins for the Keyboard Matrix?

If you intend to scan a relatively large key matrix, like one that is required for an alphanumeric keyboard, you will most likely run out of available port pins when using an SX 28 device. Before ordering an SX48 or SX52 controller (sorry, Ubicom), you should consider to use a multiplexer like the 75ACT138 chip in order to extend the available matrix column lines, as shown in the next schematic.

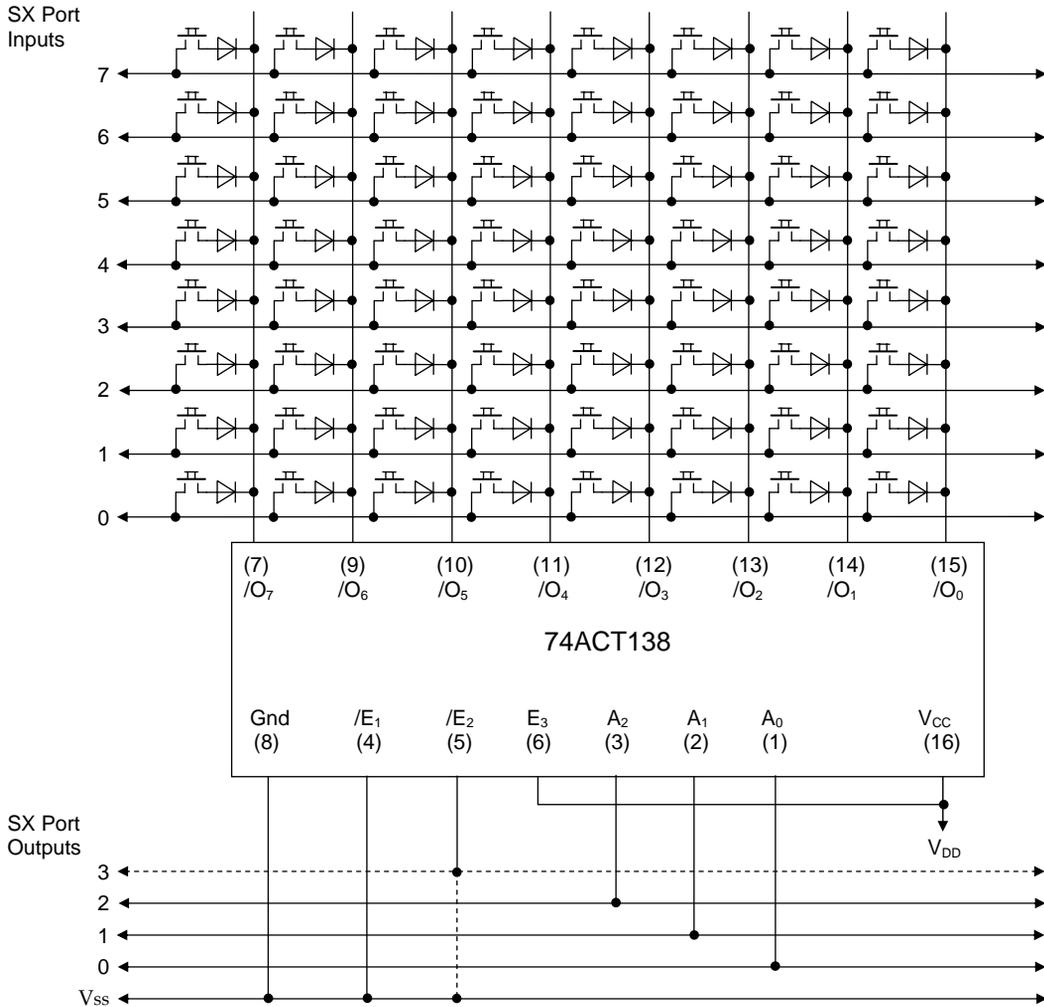
The SX program requires just a few modifications to drive the 75ACT138 instead of the column lines directly. Here, three or more SX port lines need to output the column address, and not a low for the active column. The multiplexer will take care of decoding the address, and pulling down the associated column line.

When the multiplexer does not allow to set non-active column lines to Hi-Z (i.e. if these outputs are not three-state) you will have to add one diode for each push button to avoid short circuits when the user (according to the first extend of Murphy's Law) holds down more than one key at the same time.

You can cascade two or more 75ACT138 multiplexers when you drive the enable inputs /E1, /E2, and E3 accordingly to scan 128 or even more keys.

Because multiplexers usually don't allow setting all output lines to low, you cannot use the Quick-Scan mode here.

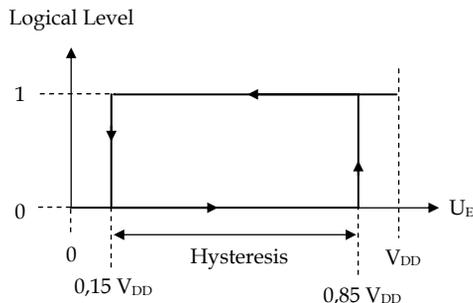
The schematic below shows how to connect an 74ACT138 multiplexer to an SX 28, to scan 64 keys.



When you use just one 74ACT138, you may connect its /E₂ input to V_{SS}, and use the SX port bit 3 for some other purpose, but if you want to cascade another 74ACT138 multiplexer, connect the /E₂ input of the first 74ACT138 to the SX port bit 3. The second 74ACT138's input E₃ should be connected to the SX port bit 3, and the /E₂ input to V_{SS}.

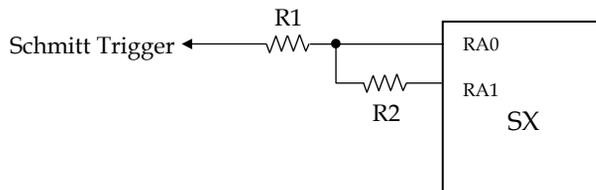
4.11 An “Artificial” Schmitt Trigger Input

With the exception of Port A, all SX inputs can be configured to Schmitt Trigger, having the following characteristics:



When the input voltage increases, starting at 0 Volts, at a level of 85% V_{DD} , the port bit's logical level “jumps” from 0 to 1. When the input voltage decreases starting at above 85% V_{DD} , the port bit's logical level “jumps” from 1 to 0 when the input voltage goes below 15% V_{DD} . The difference between the two voltage levels that cause a change of the logical level is called “Hysteresis”.

When you need all pins at Port B and Port C for other purposes, but still need another Schmitt Trigger input, or if you need a Schmitt Trigger input with a different hysteresis, use the following setup:



Here, RA0 is a CMOS input, and RA1 is configured as an output. The software sets RA1's output level to follow the logical level the SX “sees” at RA0.

When RA0 reads 0, RA1 outputs low level, and pulls RA0 to low across R2.

When the input voltage increases, the voltage at RA0 also increases, but due to the voltage divider R1/R2, the input voltage must increase to a value greater than 50% V_{DD} before the voltage at RA0 reaches the CMOS level of 50% V_{DD} .

Vice versa, when RA0 reads logical 1, it is additionally pulled up to V_{DD} through R2 by RA1 that outputs high level now. This means that the input voltage must drop to a value that is a certain

amount below 50% V_{DD} before the voltage at RA0 reaches 50% V_{DD} which makes RA0 read 0 again.

This is the typical Schmitt Trigger behavior, and the hysteresis can be adjusted to a certain extent by changing the ratio R1/R2.

Please note that here the input impedance is not Hi-Z, but equal to R1.

```

; =====
; Programming the SX Microcontroller
; APP022. SRC
; =====
include "Setup28.inc"
RESET    Main

TRIS     = $0f
LVL      = $0d

org      $100

Main
    mode LVL
    mov  !ra,  #%11111110    ; Set CMOS for ra.0
    mode TRIS
    mov  !ra,  #%11111101    ; ra.1 is output for "Mr. Schmitt"
    mov  !rc,  #%11111110    ; rc.0 is output for LED
Loop
    movb ra.1, ra.0          ; Copy input to output
    movb rc.0, /ra.0         ; Copy the inverted input level
                                ; to the LED output
    jmp  Loop                 ; Do it again as long as VDD is
                                ; there...

```

This simple program allows you to test the “artificial” Schmitt Trigger input. Try 22 k Ω for R1, and 39 k Ω for R2. When you connect an LED between RC.0 and V_{DD} (don’t forget the current limiter resistor), the LED will give you an optical feedback..

Now connect a variable voltage (0..5 V) to the Schmitt Trigger input. As you change the voltage, you can tell from the LED going on or off, how far the two levels are apart. When you use a potentiometer to obtain the variable voltage, make sure that the potentiometer value is relatively small compared to R1 otherwise, the input current would add an error when you are going to measure the hysteresis.

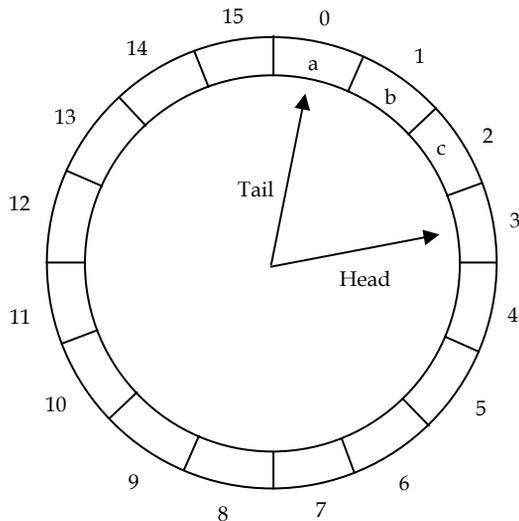
4.12 A Software FIFO

“FIFO” means “First In, First Out”, i.e. other than with a stack memory where the value saved last is read first (LIFO – Last In, First Out), a FIFO buffer always returns the value that was stored first when read (provided that at least one value was stored before).

A FIFO is useful to temporarily buffer some data that “pile up” faster as a system can process them. For example, for a while, “Mr. Fastfinger” might hit a keyboard faster than the send routine is able to transfer the key data to another device. Here, a FIFO helps to buffer the data “peaks” until the transfer routine can follow up. Of course, this only works fine if the FIFO is large enough to buffer all the excessive data that might occur in worst case.

A PC, for example, can buffer up to 16 keystrokes until they are read by the running application. If you try to type in more characters, the PC will generate a warning beep, and excessive keystrokes are lost.

It is quite easy to “build” a software FIFO for the SX controller. A FIFO is like a number of registers linked together in a ring, or circle:



This figure shows a “circular buffer” made up of 16 registers. The two pointers “Head” and “Tail” each address one register.

Head points to an empty register. When a new value is stored in the FIFO, this value goes into the register addressed by the "Head" pointer which is incremented after the new value has been stored in order to address the next empty register.

"Tail" points to the register that was read last. In order to retrieve a value from the FIFO, "Tail" first is incremented to address the next register containing valid data, and then this register is read.

When an increment of one of the pointers sets them to a value that is greater than 15 (in our example), the pointers are reset to 0 to make them "turn around in a circle".

When "Tail" has reached "Head" while reading the FIFO, i.e. both pointers address the same register, the FIFO is "empty", and all buffered data has been read. On the other hand, if "Head" reaches "Tail" while saving data, the FIFO is "full", and no more values can be stored.

The figure above shows three registers that contain valid data (a, b, and c in registers 0..2). "Head" points to the next free register, and "Tail" points to the register that has been read last, i.e. registers 1 and 2 contain buffered data that has not been read yet.

When implementing a FIFO in software, it is important to test if the FIFO is "full", or "empty", and the condition Head = Tail is not enough to tell which of the two conditions is true.

The easiest method to keep track of the FIFO status is to maintain an additional variable that contains the number of current items in the FIFO. If this variable contains 0, this indicates that the FIFO is "empty", and if it contains 16 (in our example), this indicates that the FIFO is "full".

The program below contains two subroutines that write and read values to/from a FIFO buffer, and the main program contains some instructions to test the FIFO with a debugger:

```

; =====
; Programming the SX Microcontroller
; APP023.SRC
; =====
include "Setup28.inc"
RESET   Main

org     $08
Head    ds 1           ; FIFO head pointer
Tail    ds 1           ; FIFO tail pointer
FIFOCnt ds 1           ; Current number of items the FIFO
Temp     ds 2           ; Temporary storage
FsrSave ds 1           ; Temporary storage for FSR
TestVal ds 1           ; Utility variables for testing

org     $30
FIFO    = $             ; 16 bytes for FIFO memory
        ds 16

org     $000

```

Programming the SX Microcontroller

```
;** Subroutine writes the contents of w into the FIFO
;
; Input:   w = value
;
; Changes: Temp, FSRSave, Head, and FIFOCnt
;
WriteFIFO
    snb FIFOCnt.4      ; If FIFO is "full", no action
    ret
    mov Temp, w        ; Save the value for later
    mov FsrSave, fsr   ; Save the FSR
    mov w, #FIFO       ; Indirectly address the FIFO-Puffer
    add w, Head        ; using the Head pointer
    mov fsr, w
    mov ind, Temp      ; Save the value to the FIFO
    inc Head           ; Point Head to next free storage
    clrb Head.4        ; If Head = 16, "circle" around to 0
    inc FIFOCnt        ; Increment the item count
    mov fsr, FsrSave   ; Restore the FSR
    ret

;** Subroutine reads the FIFO and returns the value in w
;
; Returns: Value in w
;
; Changes: Temp+1, FSRSave, Tail, FIFOCnt
;
; NOTE:    Temp+1 is used for temporary storage of the retrieved
;          value here to avoid conflicts when ReadFIFO is called
;          from the mainline program and gets interrupted by the ISR,
;          calling WriteFIFO to store a new value.
;
ReadFIFO
    test FIFOCnt      ; If FIFOCnt = 0, the
    snz               ; FIFO is "empty",
    ret              ; no action
    mov FsrSave, fsr  ; Save the FSR
    mov w, #FIFO      ; Indirectly address the FIFO-Puffer
    add w, Tail       ; using the Head pointer
    mov fsr, w
    mov Temp+1, ind   ; Read the value from the FIFO
    inc Tail          ; Set Tail to next location
    clrb Tail.4       ; If Tail = 16, "circle" around to 0
    dec FIFOCnt       ; Decrement the item count
    mov fsr, FsrSave  ; Restore the FSR
    mov w, Temp+1     ; Copy the value to w
    ret

org    $100

;** Mainline program to test the FIFO *****
;
Main
```

```

include "Clr2x.inc"

Loop
  inc TestVal          ; Generate test data
  mov w, TestVal      ; Write to FIFO
  call WriteFIFO      ; Write to FIFO
  inc TestVal          ; Generate more test data
  mov w, TestVal      ; Write to FIFO again
  call WriteFIFO

  clr w
  call ReadFIFO        ; Read from FIFO
  jmp Loop             ; Repeat forever...

```

When you execute the program in single steps, you will see how the FIFO buffer is filled. As the main program calls **WriteFifo** twice, but **ReadFifo** only once, the FIFO will become “full” after some program loops. You will then notice that the second **WriteFifo** call does not perform an action.

A “real” application should check if the content of **FIFOCnt** is greater than zero (e.g. in the ISR) and then read and process a value from the FIFO (e.g. transmit it to a peripheral using a UART VP).

It also makes sense to test if the contents of **FIFOCnt** is less than 16 before trying to store a new value into the FIFO to see if there is still “room” for more data in the FIFO buffer.

You can easily increase the size of the FIFO buffer if necessary, and it is possible to store larger units of data in a FIFO too. For example, if you want to save 16-bit values, you can reserve two buffers with 16 bytes each, and use them in “parallel”. In this case, you can no longer pass the value to be stored or to be retrieved through the W register. The next program is an example how to handle 16-bit FIFO data:

```

; =====
; Programming the SX Microcontroller
; APP024.SRC
; =====
include "Setup28.inc"
RESET  Main

org    $08
FIFOCnt ds 1          ; Number of items in the FIFO
FsrSave ds 1         ; Temporary storage for FSR
FIFODat ds 2         ; FIFO parameter buffer
Value  ds 2          ; Test variable for the main program

org    $30
FIFO   = $
Head   ds 1          ; FIFO head pointer

```

Programming the SX Microcontroller

```
Tail    ds 1                ; FIFO tail pointer
; NOTE: Use two subsequent banks for FIFO1 and FIFOh !
;
org     $50
FIFO1  = $                   ; 16 bytes for FIFO data (low byte)
        ds 16

org     $50
FIFOh  = $                   ; 16 Bytes for FIFO data (high byte)
        ds 16

WATCH  FIFOCnt, 8, UDEC
WATCH  Head,    8, UDEC
WATCH  Tail,    8, UDEC

org     $000

; ** Subroutine saves the contents of FIFOData in the FIFO
;
; Entry:  FIFOData = Value (low byte)
;         FIFOData+1 = Value (high byte)
;
; Changes: FSRSave, Head, FIFOCnt
;
WriteFIFO
    snb  FIFOCnt, 4          ; If FIFO is "full", no action
        ret
    mov  FsrSave, fsr       ; save FSR
    bank FIFO               ; Switch bank for Head
    mov  w, #FIFO1          ; Indirectly address the FIFO
    add  w, Head            ; using the Head
    mov  fsr, w             ; pointer
    mov  ind, FIFOData      ; Save the value (low byte)
    add  fsr, #32           ; Switch to FIFOh bank
    mov  ind, FIFOData+1    ; Save the value (high byte)
    bank FIFO               ; Switch bank for Head
    inc  Head              ; Point Head to next empty storage
    clrb Head, 4           ; If Head = 16, reset it to 0
    inc  FIFOCnt           ; Increment the item count
    mov  fsr, FsrSave      ; Restore the FSR
    ret

; ** Subroutine reads the FIFO and returns the value in FIFOData
;
; Returns: Value in FIFOData (low byte) and
;          FIFOData+1 (high byte)
;
; Changes: Temp, FSRSave, Tail, and FIFOCnt
;
ReadFIFO
    test FIFOCnt           ; If FIFOCnt = 0, the
    snz  ret               ; FIFO is "empty", i. e.
        ret               ; no action
    mov  FsrSave, fsr     ; Save FSR
```

```

bank FIFO          ; Switch bank for Tail
mov w, #FIFO1      ; Indirectly address the FIFO
add w, Tail        ; using the Tail
mov fsr, w         ; pointer
mov FIFODat, ind   ; Read the value (low byte)
add fsr, #32       ; Switch the FIFOh bank
mov FIFODat+1, ind ; Read the value (high byte)
bank FIFO          ; Switch bank for Tail
inc Tail           ; Point Tail to next storage
clrb Tail.4        ; If Tail = 16, reset it to 0
dec FIFOCnt        ; Decrement the item count
mov fsr, FsrSave   ; Restore the FSR
ret

org $100

; ** Main program to test the FIFO *****
;
Main
include "Clr2x.inc"

Loop
inc Value          ; Generate test data
inc Value+1        ;
mov FIFODat, Value ; Set the FIFO "input value"
mov FIFODat+1, Value+1 ;
call WriteFIFO     ; Save the value
inc Value          ; Generate test data
inc Value+1        ;
mov FIFODat, Value ; Set the FIFO "input value"
mov FIFODat+1, Value +1 ;
call WriteFIFO     ; Save the value
clr FIFODat
clr FIFODat+1
call ReadFIFO      ; Read one FIFO value
jmp Loop           ; Do it again...

```

As you can see, the “FIFO-internal” variables have been moved into another memory bank in order to leave more free space in the global bank for other variables, but **FIFOCnt**, and **FIFODat** are located in the global bank to allow access to these variables without the need to switch the bank.

The FIFO buffer now occupies 16 bytes in two banks, and it is important that two subsequent banks are used because the FIFO routines simply add 32 to the FSR register to indirectly address the upper part of the FIFO buffer.

4.13 I²C Routines

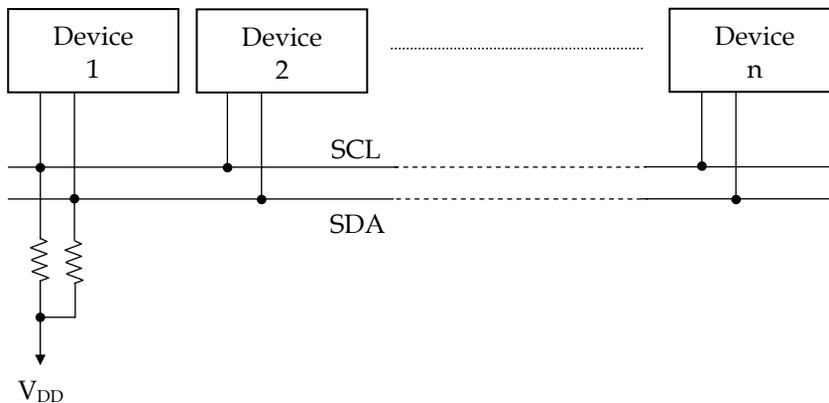
The I²C Bus (Inter Integrated circuits Communications) was developed by Philips, and it offers an interesting method to exchange bi-directional data between two or more components across two signal lines.

Ubicom, Parallax, and other companies have published various I²C routines for the SX that are useful to communicate with I²C components, like EEPROMS, A/D converters, etc. Besides explaining some I²C-basics, this chapter presents a concept how a modified I²C bus protocol, or 2-wire serial protocol can be used to exchange data between various function groups of a larger system. The following information is presented for educational purposes. I²C is a patented technology of Philips; it is the responsibility of the user to determine whether his or her application requires licensing from Philips.

Let's briefly address the I²C-basics first:

4.13.1 The I²C Bus

All components that "talk to each other" via the I²C bus are connected by two lines, as shown in the figure below:

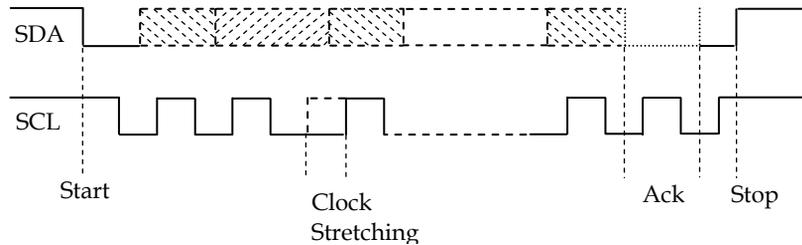


The two lines are named SDA (Serial DAta), and SCL (Serial CLock). Each device can either read the two lines, or drive the lines. When a device drives the lines, it may only pull down a line to low level, but it never may impose high level to any of the two lines. In other words, this means that all device outputs connected to the I²C bus must be (or behave like) open collector outputs. The two pull-up resistors connected between SDA, SCL, and V_{DD} provide high level on the two bus lines in case none of the connected devices pulls these lines down to low level.

4.13.2 The Basic I²C Protocol

Generally, you should keep in mind that the SDA line may only change its current level while the SCL line is at low level; in other words, data on SDA is valid only while SCL is high.

However, there are two exceptions from that rule: the Start and the Stop condition. The diagram below shows the general I²C communication:



4.13.2.1 “Master“ and “Slave“

When a communication takes place across the bus, one device takes over the role as a “Master”, i.e. this device controls the bus, while one or more other devices act as “Slaves”, i.e. they react on the “Master”, and usually one (the addressed) device will finally answer to the “Master’s voice”.

There may be devices on the bus that can only act as slave, others that can only act as master, and other devices may be able to act as master or slave as well.

4.13.2.2 The Start Condition

To start a communication, the master pulls down SDA to low while SCL is high. This is the first exception to the rule mentioned above, and this special state indicates that a master wants to start communications.

4.13.2.3 Data Transfer, and Clock Stretching

Next, the master pulls SCL low, and releases SDA, or pulls SDA low, depending on what data bit shall be sent. When the master releases SCL again, this indicates that SDA is now stable, and the slave(s) read(s) SDA now.

This process is repeated for all the data bits, the master wants to send to the slave(s). Again, note that the state of SDA only changes while SCL is low.

Common transfer rates via the I²C bus are 100 kBit/s or 400 kBit/s.

Programming the SX Microcontroller

A special case is the enlarged low phase of SCL, that is called “Clock Stretching”, and this is controlled by a slave, and not by the master. The master releases SCL when the default clock-low time has elapsed, but the receiving slave is free to hold low the SCL line as long as required in order to signal the master that it requires more time to process the data received so far. The master will continue data transmission only after the slave has released the SCL line.

Clock Stretching may occur after each bit sent, after each byte sent, or at arbitrary times, whenever the slave desires to “slow-down” the communications. This makes it possible that a master, designed for 400 kBit/s can communicate with a slave that is designed to handle 100 kBit/s only (provided that the slave “knows about its right” to stretch SCL).

4.13.2.4 Acknowledge Message from the Slave

When the master has transferred all data bits, it releases the SDA line, and then expects that the slave pulls down SDA while SCL is low, and that it keeps SDA low until SCL has made the next high-low transition.

If this is the case, the master interprets it as acknowledge from the slave. In case SDA is not pulled low in this state, the master must interpret this as transmission failure, or that the addressed slave does not exist on the bus because it did not acknowledge.

4.13.2.5 The Stop Condition

Finally, the master releases SCL first and then SDA. This is the second exception to the rule that SDA may only change its state while SCL is low, and this indicates the “Stop Condition”, i.e. the master “tells” the slaves that no more communication will follow at this time.

4.13.2.6 The Idle State

The state when SDA, and SCL both are high, and no more data bits are sent via the I²C bus is called the “Idle state”. A device that wants to become active as a master should monitor the I²C bus for the idle state before putting a start condition on the bus.

4.13.2.7 Bus Arbitration

As soon as more than one device is connected to the I²C bus that can take over a master function, it might happen that two or more masters want to start communications at the same time. This means, all masters find the bus in idle state, and “think” that it’s a good time for a new message to a slave. Don’t worry – the inventors of the I²C bus were clever enough to handle that special case as well.

Let’s assume that two masters begin communications at exactly the same time. Both masters would set up the start condition on the bus (set SDA low while SCL is high, and then set SCL

low). As long as both masters want to send the same sequence of data bits, they would continue to pull down or release the SDA/SCL lines in “complete harmony”.

In the very rare case that both masters really want to send the same bit sequence to a slave, this is fine, and both masters will see the slave’s acknowledge at the end of the transmission. Usually during that process, the time will come that one master wants to release the SDA line in order to send a 1-bit while the other master will pull down SDA because it wants to send a 0-bit.

When each master checks the SDA line for high level in case it has released it for sending a 1-bit, the masters are able to recognize that another master on the bus has set SDA to the offending level (i.e. to low). The master that has pulled SDA to low has “won” in this case, and the master that “wanted” SDA to be high, has “lost”, i.e. this master should immediately release both bus lines, and possibly try another communication when the bus is idle later.

When a master “gives” up the bus control, we can also say that this master has lost arbitration, and that the master who continues communications has “won” the bus arbitration.

In case a device can act as both, master or slave, it should continue reading the I²C bus when it has lost arbitration because there is a chance that the current bus message is addressed to the device’s slave.

4.13.2.8 Repeated Transmissions

In case a master loses arbitration, or does not receive an acknowledge from the slave, it makes sense to allow the master to repeat the current message several times until it has successfully sent the data. The send retries should be limited to a maximum number to avoid that a master keeps the bus busy when, for example, the addressed slave is not available at all.

How often a master should try a repeat, and in what time intervals depends on the needs of the specific system environment, so a general rule of thumb cannot be given here, but in most cases, it is a good idea to let the master start a retry after a random time period instead of fixed time periods.

4.13.3 The I²C Data Format

The document “The I²C-Bus and how to use it (including specifications)” published by Philips Semiconductors defines in detail the format of the I²C data packets, and you will have to respect that format when an application shall communicate with components that expect this format, like integrated circuits.

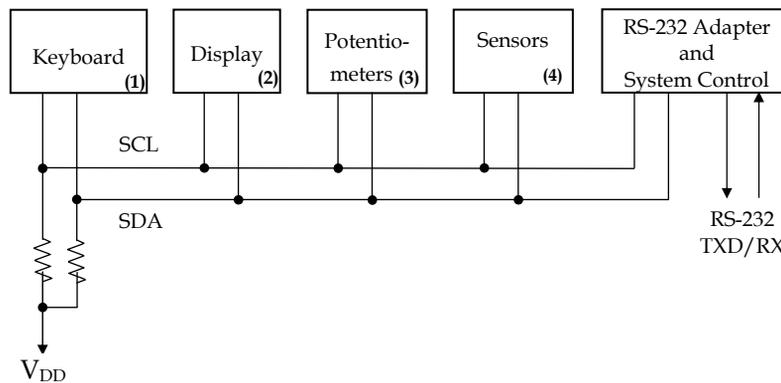
For your own systems, on the other hand, you are free to respect these specifications, or not. Possibly a different data format might be more suitable or faster for your special needs.

Programming the SX Microcontroller

In general, if more than two devices are connected via the I²C bus, it is important that each device that can act as a slave has its own unique address. If two or more devices exist in a system that can act as masters, it is also necessary in most cases that these send a unique device Id, so that the receiving slaves can determine the origin of a message.

Usually, the slave devices monitor the bus, waiting for a start condition. After detecting the start condition, the slaves enter the receive mode, and read the data packets sent from the master. If the first bits of a data packet contain the address of the destination, the receiving slaves can compare this address against their own Id. Slaves with a non-matching Ids can immediately stop processing the received data, waiting for the next stop condition, or a bus idle.

Lets have a look at an example what devices might be linked together via the I²C bus:



This fictive system is equipped with one device called the “RS-232 Adapter and System Control”. This unit receives messages from the other devices on the bus, and sends them via an RS-232 interface to another system (a computer, for example). On the other hand, this device also receives messages from another system via RS-232, interprets the messages, and sends them to one of the other devices on the bus. Therefore, this unit can act as master and slave.

The Keyboard device (1) is used for data entry, and it sends a message to the System Control unit in case a key is pressed. This device only needs to have master functionality, as it does not make sense to send output data to an input device.

On the other hand, the Display Device (2) needs to have slave functionality only in order to receive the information from the control unit, what LED should be turned on or off, or what character should be displayed on an LCD screen, etc.

The Potentiometer Device (3) might have master and slave functionality. It would automatically send an information to the control unit in case one of the potentiometer settings has changed, on the other hand is it necessary that the control unit can request the current potentiometer settings, e.g. at system start to find out the initial settings.

The same is true for the Sensors device – it should notify the Control unit about sensor value changes, but the Control unit might request certain sensor values from time to time.

Important to note is the fact that devices 1 to 4 (let's call them peripherals) send messages to the control unit only, but not to other peripherals. Vice versa, the control unit sends messages to one specific peripheral per transmission only. Nevertheless, systems are possible where the control unit sends some kind of a broadcast message to all peripherals (a master reset, for example).

We can further assume that messages originated from the peripherals shall have a higher priority than the messages from the control unit, in order to notify the Control unit about events, like key presses as fast as possible.

When the data packets originated from peripherals always begin with a zero bit, and the data packets from the Control unit always have a leading one bit, the peripherals already win bus arbitration “against” the control unit when sending the first byte. As mentioned before, the Control unit should continue acting as a slave in that case because in this system, the message is definitely dedicated to the Control Unit.

In addition, other peripheral units can stop reception immediately, when they notice that the leading bit of a data packet is 0 because then the message is originated from another peripheral that can only be directed to the Control unit.

The next bits in each message should contain the address information. Messages sent from the Control unit will have the address of the peripheral in this section that shall receive the message, and messages originated from one peripheral have the peripheral's Id in that section to allow the Control unit to figure out which peripheral device is sending the message.

In our example, together with the leading one or zero bit, two address bits are sufficient to uniquely identify all devices on the bus.

The data bits that follow the address section in each message depend on the type of device that has originated a message. In case of the keyboard device, this might be the number of a key, eventually together with an additional bit that indicates if the key was currently pressed or released.

Although a the number of data bits might be varying, depending on the device, in most cases, it makes sense to use the same number of data bits in all messages, filling sections that are not needed with zero bits.

Programming the SX Microcontroller

As mentioned before, the Control unit may send a request to the potentiometers peripheral for the setting of a specific potentiometer. The I²C protocol provides the possibility, that during one communication the master sends a command to the slave first, that is answered by the slave before a stop condition is set up on the bus, i.e. the direction of data flow is reversed during communication.

In a proprietary system, like in our example, it is often easier to split the request and the answer into separate messages, i.e. first the Control unit acts as master, sending the request to the potentiometers peripheral which acts as slave then. Next, the potentiometer peripheral acts as master, sending the answer as a new message to the control unit, which then acts as slave.

Of course, it might happen, that another peripheral device initiates a message to the control unit before the potentiometer device could send the requested answer, possibly leading to a bus arbitration. This can be easily handled by the Control unit as each message contains a unique device address information. In case a device has lost arbitration it should re-send the message at a later time.

4.13.4 Bus Lines and Pull-up Resistors

When you use the I²C bus for intra-device communications, as shown in the example before, the physical lengths of the bus lines is much greater than usual, when components on one PCB communicate via the I²C bus.

The example system is a bit similar to the ACCESS.bus System that was presented by Philips together with Digital Equipment Corporation. However, today, this bus system has lost importance compared to the USB system.

To connect the components of a system, you should use shielded cables with low capacity for the two bus lines SDA and SCL, and the pull-up resistors should be as low as the device outputs can drive. It also is a good idea to have a set of pull-up resistors on each device in order to reduce reflections. In this case, you must take care that the paralleled pull-up resistors do not put a load on the bus the device outputs cannot drive.

These provisions help to increase the noise immunity, make sure that the signals have a short rise time, and therefore allow for a high transmission speed, and to reduce EMI and RFI.

In practical I²C applications, the USB full speed cables have been used with success. These cables are shielded twice, have low capacity, are cost-effective, and come with two additional leads that can be used to power the devices. It is a good idea to attach an A- and B-type USB plug to each device. This allows “daisy-chaining” the devices through standard full speed detachable USB cables.

With such cables, SCL rates of up to 400 kHz have been successfully tested. When high transfer rates are not needed, it may be a good idea to reduce the rate to obtain a better noise immunity, and to reduce EMI and RFI.

4.13.5 I²C Routines for the SX Controller

As mentioned in the beginning of this chapter, there are various VPs available for the SX that offer I²C master and slave functionality, but these VPs have been mainly designed for communications between SXes, or the SX and other I²C components like EEPROMS.

The routines shown here have been designed for a system of peripheral units, similar to the one described above:

```

; =====
; Programming the SX Microcontroller
; APP025.SRC
; =====
include "Setup28.inc"

;-- Modify these definitions as needed -----
;
; SELF_TEST      equ 1          ; Activate this line for self-test!
; SCOPE          equ 1          ; Activate this line for a scope
;                ; trigger pulse at rc.0!
DEV_TYPE        equ 1          ; Device type
DEV_ID          equ 3          ; Device address
BIT_COUNT       equ 16         ; Number of bits/I2C packet
WAIT_IDLE       equ 4          ; Number of I2C clock periods that
;                ; must elapse while SCL/SDA are
;                ; both high until an idle state is
;                ; assumed
REPEAT_SEND     equ 5          ; Number of re-tries in case of a
;                ; missing acknowledge
I2C_PORT        equ ra         ; Port that controls SDA and SCL
SCL_BIT         equ 0          ; Port bit for SCL
SDA_BIT         equ 1          ; Port bit for SDA
I2C_BIT_MASK    equ %11111100; In this mask, the bits must be
;                ; clear that correspond to the
;                ; port's SDA and SCL bits.
INT_PERIOD      equ 125        ; At 50 MHz system clock, the ISR
;                ; is called every 2,5 µs. Per I2C
;                ; clock, four ISR calls are
;                ; required, resulting in an SCL
;                ; period of 10 µs, or a SCL
;                ; frequency of 100 kHz
;-----

;-- Internal definitions -----
;
ifndef SELF_TEST ; Internal device address

```

Programming the SX Microcontroller

```
DEV_ADDR equ ((DEV_TYPE * 8) + DEV_ID) * 16
else
DEV_ADDR equ (((1-DEV_TYPE) * 8) + DEV_ID) * 16
endif

TRIS equ 00fh
LVL equ 00dh
INT_ON equ %10011110; Enable RTCC interrupt
SCL equ I2C_PORT.SCL_BIT ; Definition for SCL port bit
SDA equ I2C_PORT.SDA_BIT ; Definition for SDA port bit

;-- Macro definitions -----
SDA_LOW macro; Pull SDA low
    clrb TrisMask.SDA_BIT
endm

SDA_HIGH macro ; Release SDA to high
    setb TrisMask.SDA_BIT
endm
SCL_LOW macro; Pull SCL low
    clrb TrisMask.SCL_BIT
endm

SCL_HIGH macro ; Release SCL to high
    setb TrisMask.SCL_BIT
endm

SkipIfNotSda macro ; Skip if SDA is low
    snb SDA
endm

SkipIfScl macro ; Skip if SCL is high
    sb SCL
endm

SkipIfNotScl macro ; Skip if SCL is low
    snb SCL
endm

StartTx macro ; Start the I2C master
    setb Flags.0
endm

StopTx macro ; Stop the I2C master
    clrb Flags.0
endm

SkipIfTxNotBusy macro ; Skip if I2C master is not active
    snb Flags.0
endm

SetTxError macro ; Set the I2C master error flag
    setb Flags.1
endm
```

```

ClrTxError macro          ; Clear the I2C master error flag
  clrb Flags. 1
endm

SkipIfNoTxError macro    ; Skip if no I2C master error
  snb Flags. 1
endm

SetRxData macro          ; Set "received data" flag
  setb Flags. 2
endm

ClrRxData macro          ; Clear "received data" flag
  clrb Flags. 2
endm

SkipIfNoRxData macro     ; Skip if no new data have been
  ; received
  snb Flags. 2
endm

;-- Global variables -----
;
org      $08
;-----
Flags    ds 1    ; Various flags (see the macros)
DevAddr  ds 1    ; Internal device address
Timer    ds 3    ; For testing purposes only

;-- Variables for the I2C master -----
org      $50
I2C_Tx   equ $
;-----
TxData   ds 2    ; Transmit data
TxBuffer ds 2    ; Transmit buffer
TxState  ds 1    ; Main state
TxSubState ds 1  ; Sub-state
TxRepeat ds 1    ; Repeat counter
TxBitCount ds 1 ; Bit counter
TxTimer  ds 1    ; Time counter
TrisMask ds 1    ; Port mask
TxTimeout ds 1  ;

;-- Variables for the I2C slave -----
;
org      $70
I2C_Rx   equ $
;-----
RxData   ds 2    ; Receive data
RxBitCount ds 1 ; Bit counter
RxState  ds 1    ; Main state
RxSubState ds 1 ; Sub-state
RxTimeout ds 1  ; Counter for timeout

```

Programming the SX Microcontroller

```
-----  
; ISR  
-----  
    org    $00  
    call   @I2CTX           ; Call the I2C master  
    call   @I2CRX          ; Call the I2C slave  
  
    bank   I2C_Tx  
    and    I2C_PORT, #I2C_BIT_MASK ; Clear bits in r?  
    mov    !I2C_PORT, TrisMask      ; Output bit mask for !r?  
  
    mov    w, #-INT_PERIOD  
    retiw  
  
-----  
InitPorts    ; Initialize the ports  
-----  
    mode   LVL  
    mov    w, #%11111100  
    mov    !I2C_PORT, w           ; Set I2C lines to CMOS  
    mode   TRIS  
  
    ifdef SCOPE                   ; Configure trigger output for  
        mov    !rc, #%11111110    ; an oscilloscope  
    endif  
    ret  
  
    org    $100  
  
-----  
Start ; Mainline program  
-----  
include "Clr2x.inc"  
  
    call   InitPorts             ; Configure the ports  
    mov    DevAddr, #DEV_ADDR  
    bank   I2C_Tx  
    mov    TxData, DevAddr      ; The upper four bits of the trans-  
                                ; mit data contain the device ID  
    mov    TrisMask, #I2C_BIT_MASK ; Initialize the port mask  
    setb   TrisMask.SCL_BIT  
    setb   TrisMask.SDA_BIT  
    mov    !I2C_PORT, TrisMask  ; Set SDA and SCL high  
    mov    !option, #INT_ON     ; Enable the RTCC interrupt  
    mov    Timer+1, #5         ; For testing purposes only  
  
-----  
Main ; Main program loop  
-----  
    SkipIfNoRxData             ; If received data, re-start the  
        ClrRxData              ; slave  
    SkipIfTxNotBusy           ; Wait when Master is busy
```

```

        jmp    Main
: Delay
    decsz   Timer                ; Generate a delay
    jmp     : Delay
StartTx
    decsz   Timer+1              ; Start the master
    jmp     : Delay              ; Generate a delay
    jmp     Main
    add     TxData+1, #1          ; Increment send data
    addb    TxData, c
    and     TxData, #$0f         ; Set the upper 4 bits to the
    or      TxData, DevAddr      ; device ID
    jmp     Main                 ; Repeat sending data

org    $200

;-- I2C master -----
bank   I2C_Tx

;-----
; Routines called from various sub-states
;-----

ClockHigh                ; Set SCL high and select next
                        ; sub-state
    SCL_HIGH
    inc     TxSubState
    retp

ClockLow                  ; SCL low
    ifdef  SCOPE
        clrb rc.0              ; Trigger pulse for oscilloscope
    endif
    SCL_LOW
    inc     TxSubState
    retp

DataHigh                  ; SDA high and next sub-state
    SDA_HIGH
    inc     TxSubState
    retp

WaitClockHigh; Wait until SCL is high
SkipIfNotScl
    inc     RxSubState          ; and next sub-state
    retp

;-----
I2CTX                      ; I2C-Master
;
; Data to be sent must be stored in TxData (HOB) und TxData+1 (LOB)
; before starting the master.
;
;
; The StartTx macro starts the master

```

Programming the SX Microcontroller

```
; The SkipIfTxNotBusy macro tests if the master is busy
; The SkipIfNoTxError macro tests if there was an error during the
; last send (no acknowledge from slave or timeout).
;-----
    mov     w, TxState           ; Jump table for main states
    jmp     pc+w
:TxIdle
    jmp     :Idle                ; Nothing to send
:TxInit
    jmp     :InitSend           ; Initialize the master
    jmp     :SetStart           ; Set the start condition
    jmp     :SendData           ; Send data
    jmp     :GetAck             ; Read the acknowledge
    jmp     :SetStop            ; Set stop condition
:TxError
    jmp     :HandleError        ; Handle errors

;-- Nothing to send -----
;
:Idle
    mov     TxRepeat, #REPEAT_SEND+1 ; Initialize the repeat counter
    SDA_HIGH ; Release SDA and SCL for safety
    SCL_HIGH ; reasons
    SkipIfTxNotBusy ; If the mainline has turned on the
    inc TxState ; master, next state is InitSend
    retp

;-- Initialize the master -----
;
:InitSend
    mov     TxBuffer, TxData ; Copy send data to the send
    mov     TxBuffer+1, TxData+1 ; buffer
    ifdef SCOPE ; Generate trigger pulse for an
        setb rc.0 ; oscilloscope
    endif
    mov     TxBitCount, #BIT_COUNT ; Initialize the bit counter
    ClrTxError ; Clear the error flag
    clr     TxTimeout ; Clear the timeout counter
    clr     TxSubState ; Clear the sub-state
    inc     TxState ; Next state is SetStart
    retp

;-- Set the start condition -----
;
:SetStart
    mov     w, TxSubState ; Jump table for sub-states
    jmp     pc+w
    jmp     :StartInit ; Initialization
    jmp     :WaitIdle ; Wait for the stop condition

:StartInit
    SCL_HIGH; Release SCL and
    SDA_HIGH; SDA to high
    mov     TxTimer, #WAIT_IDLE ; Initialize the time counter
    inc     TxSubState ; Next sub-state is WaitIdle
```

```

    retp
: WaitIdle
    dec    TxTimeout
    snz
        jmp    : HandleError
    sb     SCL
        jmp    : NotIdle           ; If SCL is low, no stop state
    sb     SDA
        jmp    : NotIdle           ; If SDA is low, no stop either
    dec    TxTimer
        ; Maybe, we have a stop now:
        ; decrement the counter, and stay
        ; in this sub-state if TxTimer
        ; > 0
    sz
        retp
    clr    TxTimeout
        ; We have a stop, clear timeout
    SDA_LOW
        ; Set the start condition
    clr    TxSubState
        ; Clear the sub-state
    inc    TxState
        ; Next state is SendData
    retp
: NotIdle
    mov    TxTimer, #4
        ; If no stop state, re-init the
    retp
        ; timer, and stay in this sub-
        ; state
; -- Send data -----
;
: SendData
    mov    w, TxSubState
        ; Jump table for sub-states
    jmp    pc+w
    jmp    ClockLow
    jmp    : SetDataBit
    jmp    ClockHigh
    jmp    : CheckClockHigh
: SetDataBit
    SDA_HIGH; Prepare SDA mask to be high
    sb     TxBuffer.7
        ; If transmit bit is low,
        SDA_LOW
        ; clear the SDA mask

    inc    TxSubState
        ; Next sub-state is ClockHigh
    retp
: CheckClockHigh
        ; Check if SCL is high in order
        ; to allow for clock-stretching
    dec    TxTimeout
    snz
        jmp    : HandleError
    sb     SCL
        retp
        ; If SCL is low, stay in this
        ; sub-state
    sb     TxBuffer.7
        ; If the send bit is high, we need
        ; to test for arbitration, else
    jmp    : PrepareNext
        ; prepare next bit
    SkipIfNotSda
        ; If SDA is low we have lost
        ; arbitration, else

```

Programming the SX Microcontroller

```
    jmp :PrepareNext          ; prepare next bit
    mov TxState, #(:TxError-:TxIdle) ; State is HandleError
    retp

:PrepareNext
    clr TxTimeout            ; Reset timeout
    rl  TxBuffer+1          ; Next bit to TxBuffer.7
    rl  TxBuffer             ;
    clr TxSubState          ; Clear the sub-state
    dec TxBitCount          ; Decrement the bit counter, if
    sz  ; 0, we're all done
    retp
    inc TxState ; Next state is GetAck
    retp

;-- Read acknowledge -----
;
: GetAck
    mov w, TxSubState        ; Jump table for sub-states
    jmp pc+w
    jmp ClockLow
    jmp DataHigh
    jmp ClockHigh
    jmp :CheckClockHighAck

: CheckClockHighAck
    dec TxTimeout
    snz
    jmp :HandleError
    sb SCL
    retp ; If SCL is low, stay in that state
    inc TxState ; Next state is SetStop
    clr TxSubState
    SkipIfNotSda ; If SDA is low, we have an Ack,
    SetTxError ; else set the error flag
    retp

;-- Set stop condition -----
;
: SetStop
    mov w, TxSubState        ; Jump table for sub-states
    jmp pc+w
    jmp ClockLow
    jmp :DataLow
    jmp ClockHigh
    jmp DataHigh
    jmp DataHigh
    jmp :TxFinish

: DataLow
    SDA_LOW
    inc TxSubState
    retp

: TxFinish
```

```

inc    TxState          ; Prepare state for HandleError
SkipIfNoTxError        ; If there is an error, keep that
    retp                ; state, else clear the state
clr    TxState          ; next state is Idle
StopTx                  ; Clear the "Master Busy" flag
retp

;-- Error handling -----
;
: HandleError
SCL_HI GH              ; Release the bus
SDA_HI GH
clr    TxSubState      ; Clear the states
clr    TxState
dec    TxRepeat        ; If a repeats are allowed, set
mov    w, #(:TxInit - :TxIdle) ; state to TxInit, else
sz
mov    TxState, w
StopTx                  ; clear the "Master Busy" flag,
SetTxError              ; and set the error flag
retp

    org    $400
-----
I2CRX ; I2C-Slave
;
; Received data is stored in RxData (HOB) and RxData+1 (LOB)
;
; The SkipIfNoRxData macro tests if new data is available
; The ClrRxData macro enables the slave to receive more data. Call
; this macro after processing the recently received data.
;-----
    bank    I2C_Rx

SkipIfNoRxData          ; No action if the mainline pro-
    retp                ; gram has not enabled the slave

dec    RxTimeout       ; If timeout,
snz
    jmp    :RxError     ; try to receive again

mov    w, RxState      ; Jump table for mains states
jmp    pc+w
jmp    :RxDetectStart
jmp    :RxGetBits
jmp    :RxSendAck

;-- Wait for start condition
;
: RxDetectStart
mov    w, RxSubState   ; Jump table for sub-states
jmp    pc+w
jmp    :WaitSdaHigh
jmp    :WaitSdaLow

```

Programming the SX Microcontroller

```
: WaitSdaHigh
  SkipIfNotSda          ; If SDA is high, next sub-state
  inc RxSubState        ; is WaitSdaLow
  retp

: WaitSdaLow
  SkipIfNotSda          ; Wait until SDA is low
  retp                  ; (possibly a start condition)
  SkipIfNotScl          ; If SCL is not low, this is not
  jmp : RxError         ; a start condition
  mov RxBitCount, #BIT_COUNT ; Init the bit counter
  clr RxSubState        ; Clear the sub-state
  inc RxState           ; Next state is RxGetBits
  retp

; -- Receive data -----
;
: RxGetBits
  mov w, RxSubState     ; Jump table for sub-states
  jmp pc+w
  jmp : WaitClockHigh
  jmp : GetDataBit

: GetDataBit
  clc                   ; Receive a bit
  SkipIfNotSda         ; Clear C, and if
  stc ; set C          ; SDA is high,
  rl RxData+1          ; Shift C into the received data
  rl RxData ;
  decsz RxBitCount     ; If there are more bit to receive,
  jmp : SetupNext      ; prepare next bit, else
  clr RxSubState       ; clear the sub-state, and
  inc RxState          ; next state is SendAck
  retp

: SetupNext
  dec RxSubState        ; Set sub-state to WaitClockHigh
  retp

; -- Send an acknowledge -----
;
: RxSendAck
  mov w, RxSubState
  jmp pc+w
  jmp : WaitClockLowAck
  jmp : WaitClockHigh
  jmp : WaitClockLow

: WaitClockLowAck
  SkipIfNotScl         ; Wait for SCL low before sending
  retp                 ; the acknowledge
  mov w, RxData        ; Test the received device ID and
  ifndef SELF_TEST     ;
  xor w, #%10000000    ; invert the device type bit
  endif
```

```

bank I2C_Tx
and    w, #Sf0                ; Mask Id bits, and compare against
mov    w, DevAddr-w          ; DevAddr
snz    SDA_LOW                ; If equal, pull
bank I2C_Rx                    ; SDA to low for acknowledge
inc    RxSubState            ; Next sub-state is WaitClockHigh
retp

: WaitClockHigh                ; Wait until SCL is high
SkipIfNotScl
inc    RxSubState            ; Next sub-state is WaitClockLow
retp

: WaitClockLow                ; Wait until SCL is low
SkipIfNotScl
retp
#ifdef SELF_TEST
bank I2C_Tx
SDA_HI GH                    ; Release SDA to high
bank I2C_Rx
endif
SetRxData; Set the "Data received" flag

: RxError
clr    RxState                ; Clear the states, and
clr    RxSubState
clr    RxTimeout              ; The timeout counter

retp

```

This relatively large program contains an I²C Master VP (**I2CTx**), a Slave VP (**I2CRx**), and a small mainline program used to test the routines.

Some definitions are located at the beginning of the program code, which help to configure the program to your own needs:

If **SELF_TEST** is defined, the Slave VP does not invert the device type bit, i.e. it accepts the data received from its “in-program” master as valid (we’ll discuss this in more detail later).

If **SCOPE** is defined, a trigger pulse for an oscilloscope is generated at **rc. 0** in order to get a stable display of the SDA and SCL signals.

DEV_TYPE defines the device type. As we already mentioned before, if two devices start a transmission at the same time, the device that pulls SDA low wins bus arbitration, and the device that wants SDA to be high loses bus arbitration. **DEV_TYPE** defines the first bit that is sent, therefore devices with **DEV_TYPE = 0** have a higher priority (because they win arbitration) than those with **DEV_TYPE = 1**.

Programming the SX Microcontroller

As both, the Master and the Slave VP are executed “simultaneously”, the Slave VP also receives what the Master VP sends. Under normal conditions, the Slave VP only accepts data packets that begin with a device type bit that is not equal to its own device type, and therefore the Slave VP ignores the messages, its “companion” master sends. For testing purposes, you may define **SELF_TEST**. In this case, the slave ignores the device type bit, and receives the “companion” master’s messages.

DEV_ID defines the unique address of a specific device in the system. This program allows for Ids from 0 through 7. Together with the definition of **DEV_TYPE**, the program builds an address that is stored in the upper nibble of the **DevAddr** variable. The mainline program should copy these upper four bits into the upper nibble of the high order byte to be transmitted.

The Slave VP only accepts messages that have the same address stored in bits 14, 13, and 12 of a received 16-bit word.

BIT_COUNT defines the number of bits to be sent or received in a message. The maximum value is 16, and it may be smaller. If you need more than 16 bits per message, the program must be enhanced to use more than two bytes for the send and receive data buffers.

WAIT_IDLE defines the time (in numbers of I²C clock cycles) the Master shall monitor the bus for an idle state before setting the start condition.

REPEAT_SEND defines how often the Master shall try to re-send a data packet in case of an error (no acknowledge from the receiver, or timeout).

I2C_PORT defines the SX port where SDA and SCL are connected, and the definitions **SCL_BIT** and **SDA_BIT** define the pins of this port that drive SCL and SDA. Together with the **I2C_BIT_MASK** definition, you can make the entire configuration here, and there is no need to change any statement somewhere else in the code. In **I2C_BIT_MASK**, the bits that correspond to the SDA and SCL port bits must be reset, and all other bits should be set.

INT_PERIOD defines the number of system clock cycles that shall elapse before the ISR is invoked again. If you use the specified value 125, the SCL frequency will be 100 kHz. Note that changing this value might influence other VPs in the program.

4.13.5.1 Common Program Modules

The definitions above are followed by some macro definitions that make the program more readable, and then the variable definitions follow.

The global registers hold the **Flags** variable. Some of its bits are used to store the flags in this program (see the macro definitions). **Timer** is used by the mainline program for demonstration purposes, and **DevAddr** contains the combined address information in its upper nibble that the mainline program should copy into bits 15, 14, 13, and 12 of the 16-bit send data word.

Two banks are used for the I²C VPs, one for the master, and one for the slave. The comments in the source code explain the meaning of these variables.

Please note the special meaning of the **TrisMask** variable in the **I²C_TX** bank. Because the SDA and SCL lines must be turned into Hi-Z inputs when the bus lines shall be released to high, it is necessary to set or reset the bits in the TRIS configuration register of the I²C port. Because the **setb** instruction cannot be used for an **!r?** configuration register, the I²C VPs set and clear the corresponding bits in **TrisMask** instead, and the ISR copies the contents of this variable into the TRIS port configuration register each time it is called.

If you want to make use of the free I²C port pins, you must set or clear the corresponding bits in **TrisMask** in the mainline program, when the variables are initialized instead of using a **mov !r?, w** instruction because the ISR would override this setting at the next invocation. Should it be necessary to change the direction of any of these port pins at run-time, you can do this by changing the bits in **TrisMask** but you should be aware that the change will not take place before the next ISR call.

The ISR calls the two I²C VPs each time it is invoked. These I²C VPs are located in two separate program pages at \$200 and \$400. Because both VPs are subroutines, they must begin in the first half of a page. In addition, both VPs contain various jump tables using the **jmp pc+w** instruction whose targets must also be located in the first half of a page. The total size of both VPs is 214 words, i.e. both would “fit” together within one first half of a page, but a few enhancements can easily break the “magic” barrier of 256 words, and – as you know – **jmp pc+w** instructions can take you to “Nirvana” if the targets are not located in the first half of the page!

In addition, the ISR takes over the task to update the TRIS register of the I²C port regularly. To make sure that the SDA or SCL are pulled low, when the bits in **TrisMask** are clear, it clears the bits in the associated port register as well.

4.13.5.2 The Mainline program

In this example, the mainline program is used to test the I²C VPs. After clearing the data memory, initializing the ports and variables, the program enters into a loop.

In this loop, it checks whether the Slave has received data. If this is the case, the data words are ignored in this sample program, and the Slave is enabled anew.

When the Master is not busy, it gets started after a short delay.

After another delay, the contents of **TxDatA** and **TxDatA+1** are incremented, and the higher nibble of **TxDatA** is set to the high nibble in **DevAddr**.

Thus, the Master keeps sending the contents of **TxDatA** and **TxDatA+1** as it increments at a slower rate via the I²C bus.

Programming the SX Microcontroller

When you activate the **SELF_TEST** mode, and **SCOPE**, you can display the signals on SDA and SCL on an oscilloscope (don't forget to connect the "External Trigger" input with **RC. 0** for stable displays. You can also monitor how the transmitted data byte is incremented.

4.13.5.3 The I²C Master VP

I2CTx is periodically called from the ISR, four times per I²C clock cycle, i.e. when sending a data bit, SCL has performed the sequence high-high-low-low after four calls, and the SDA line has eventually changed its level in the middle of the two SDA low steps.

Before initiating a transmission, the Master first must check if the bus is available. Therefore, it checks if for a certain time both, SDA and SCL are high (**WAIT_IDLE** defines the number of ISR calls it should wait). If this is the case, i.e. when the bus is idle, the master puts the start condition on the bus, i.e. it pulls SDA low while SCL is high, and then continues sending the data bits.

Finally, the master must check if the receiving slave sends an acknowledge, i.e. if it pulls SDA low. To complete the transmission, the master releases SDA while SCL is low to establish the stop condition.

Again, implementing a state engine is a good method to handle the various steps the master must perform. This means that the master code for the master VP has just one entry point, and from there, branches to different sections of the code are performed, according to the contents of the **TxState** variable. The various sections of the code change the contents of **TxState** if necessary in order to select another state (usually the next one).

In case of the Master VP, the states are further divided into sub-states, controlled by the **TxSubState** variable.

You can follow the program flow in the source code listing quite easily by keeping track of the **TxState** and **TxSubState** variables.

State engines can cause a problem when a specific external status is expected that – by some reason – might not occur. In this case, the engine's state does not change, and it "hangs around" in one state forever.

For example, the Master VP might remain "stuck" in the **CheckClockHigh** sub-state where it waits for SCL returning to high level while other devices hold SCL low for clock-stretching. If a faulty external device holds that line low forever, the master can't continue either.

To avoid such situations, the I²C VPs presented here, have an additional timeout feature. In the Master VP, routines that are "candidates" for such problems, the variable **TxTimeout** is decremented each time the routines are executed, and when **TxTimeout** underflows, the engine's state is forced to the error state.

The jump tables that select the states and sub-states have been sorted in “logical” order, i.e. usually, a routine simply needs to increment the state variable to activate the state that logically follows next, or to decrement the state variable to select the previous state.

However, there are some necessary exceptions to that rule, for example, in case of an error situation, where **TxState** must be set to a fixed value. You could achieve this by adding a **mov TxState, #???** instruction. But the drawback of this method is that you will have to change the “hard coded” constant part of such instructions later, should there be a need to add one or more states to the routine. You can be sure, big surprises will come up in case you forget to make the necessary corrections.

Therefore, it is a better idea to let the assembler calculate such values automatically. Therefore, some jump tables have some additional labels, like in

```

    mov    w, TxState      ; Jump table for main states
    jmp    pc+w
:TxIdle
    jmp    :Idle          ; Nothing to send
:TxInit
    jmp    :InitSend      ; Initialize the master
    jmp    :SetStart      ; Set the start condition
    jmp    :SendData      ; Send data
    jmp    :GetAck        ; Read the acknowledge
    jmp    :SetStop       ; Set stop condition
:TxError
    jmp    :HandleError   ; Handle errors

```

For example, you could reach the **HandleError** state by “hard-coding” a **mov TxState, #6** instruction. Because both, the first **jmp :Idle** instruction, and the **jmp :HandleError** instruction are marked with the additional labels **:TxIdle**, and **:TxError**, you can write **mov TxState, #(:HandleError - :TxIdle)** instead, to let the assembler make the necessary calculation, giving 6 in this case. If you would later insert another **jmp :????** instruction in between the two labels, the assembler would automatically calculate the new value of 7 that is now the correct value for **TxState** to reach the error handler.

In case of an error (timeout, or no acknowledge from the slave), the **:HandleError** routine is executed. If **REPEAT_SEND** has been defined with a value greater 0, the VP tries to resend the data packet, as often as has been defined. In this sample program, retries will be immediately following an error situation. When the overall busload is relatively high, it might be better to insert a random delay time before the master performs the next retry.

4.13.5.4 The I²C Slave VP

Again, this slave has been designed as a state engine, and some states are divided into sub-states similar to the master. The slave also has a timeout feature. Here, the timeout counter is decremented, each time the slave is called in case it is enabled to receive data.

Programming the SX Microcontroller

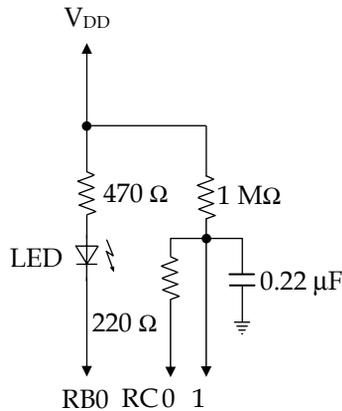
This means that a timeout error will also occur when the slave has been called 256 times while the bus stays idle. As this situation does not set the “Data Received” flag, this means that the slave simply re-starts, monitoring the bus. Only if a timeout situation occurs while the slave is waiting for a SCL high during reception of data, the error handler discards whatever has been received so far, not notifying the mainline program.

Other than the Master VP, that sends the data “unchecked”, i.e. the mainline program must take care of setting up the device type bit and the device address, the Slave VP checks both values. The “Data received” flag will only be set, and an acknowledge only be sent if the information in the first four received bits matches the configured values.

When you want to test just one device without connecting it to other devices, you can activate the **SELF_TEST** definition. In this case, the device type bit is not inverted, and the slave receives and acknowledges the data sent from its “companion” master.

4.14 A “Hardware Timer“

When an application requires the SX registers for other purposes than for counting long time periods, or if the ISR is more than busy with other tasks, you can generate a relatively long time delay with just a couple of external components, and two port pins. The schematic below shows the details (where the LED and the 470 Ω resistor are for demonstration purposes only).



The LED connected to RB0 is used to display the status in our demonstration program. An RC network (1 M Ω /0.22 μ F) is connected between V_{DD} and V_{SS}, and port pin RC0 is connected to the capacitor across a 220 Ω resistor that is used to limit the port output current.

This is the demonstration program:

```

; =====
; Programming the SX Microcontroller
; APP026. SRC
; =====
TRIS    = $0f
ST      = $0c

DEVICE  SX28L
DEVICE  TURBO, STACKX, OPTIONX
FREQ    50_000_000
RESET   Main

org     $100

; ** Main program *****
;
Main

```

Programming the SX Microcontroller

```
mode ST          ; Configure RC1 as
mov !rc, #%1111101 ; Schmitt Trigger input
mode TRIS       ; RB0 is the LED
mov !rb, #%1111110 ; output

Loop
  clrb rc.0      ; Configure RC0 as an output
  mov !rc, #%1111110 ; with low level
Wai tLow
  snb rc.1       ; Wait until the capacitor
  jmp WaitLow    ; is discharged
  mov !rc, #%1111111 ; Set RC0 to Hi-Z
Wai tHigh
  sb rc.1        ; Wait until the capacitor
  jmp WaitHigh   ; is charged
  xor rb, #%00000001 ; Toggle the LED
  jmp Loop       ; Do it again...
```

The main program loop instructions first discharge the capacitor by pulling its upper plate across the current-limiting resistor to low level, and then waits until the voltage across the capacitor has dropped below the lower level of the Schmitt Trigger input at **rc. 1**.

Then **rc. 0** is switched to Hi-Z, i.e. the capacitor is now charged across the 1 M Ω resistor. The program loop **Wai tHigh** is executed until the voltage across the capacitor, has reached the upper level of the Schmitt Trigger input at **rc. 1**.

Then, the LED port bit is toggled, and the main loop is repeated again.

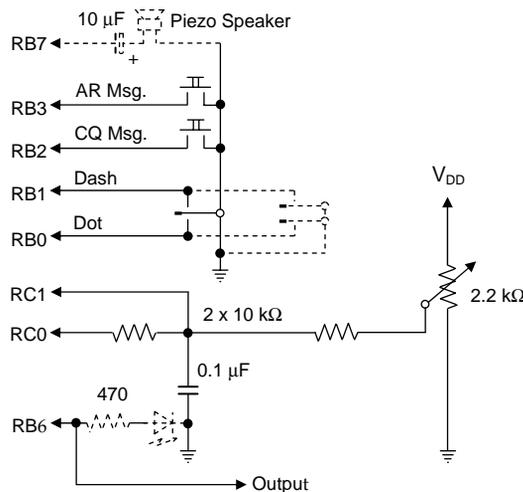
By varying the time constant RC, you can achieve all kinds of delay times. Although the delay time precision cannot be compared to a system-clock-derived time, it may be sufficient for many purposes.

4.15 A Morse Code Keyer

This application makes use of an SX Controller to build a Morse Code Keyer with the following features:

- Accepts “Paddle-Type” or “Squeeze-Type” input devices
- Automatically sends two pre-defined messages, a repeating message like “cq cq cq de <callsign> <callsign> <callsign>”, and a terminating message, like “ar pse k”.
- The tempo can be adjusted via a potentiometer.

The schematic below shows the required external components:



If you are using a SX-Key Demo Board from Parallax, most of the required components are already in place.

To read the speed potentiometer, this application makes use of the Bitstream ADC VP that was already described in this book before. The LED at RB6 is optional; it gives optical feedback as it is toggled according to the generated Morse code. RB6 is also used to drive an external circuit that keys the transmitter.

If connected to RB7, the piezo speaker provides an acoustic feedback. As most transmitters generate their own monitor tones, you might consider to not install the speaker. You may also add a switch to turn the speaker on or off.

Programming the SX Microcontroller

RB0 is the “Dot” input, i.e. when this pin is pulled low, a short “Dot” signal will be generated, followed by a pause of the same length. As long as the line is held low, “Dots” and pauses will be repeated.

When RB1, the “Dash” input is pulled low, a “Dash” signal will be generated, that is three times longer than a “Dot”. Again, a pause of one dot-length follows each “Dash”. As long as the line is held low, “Dashes” and pauses will be repeated.

When you use a “Paddle-Type” input device, either the “Dot” or the “Dash” input can be low. If you use a “Squeeze-Type” device instead, either the “Dot”, the “Dash”, or both inputs can be low at a time. When both inputs are low, the “Dot” input has higher priority, i.e. “Dots” will be generated as long as it is low. When the line is released while the “Dash” line is still low, the system will continue sending “Dashes”. If you pull the “Dot” line low, while the “Dash” line is low, “Dots” will be generated again.

If you press the “CQ” button, the CQ message, e.g. “cq cq cq de dk4tt dk4tt dk4tt” will be continuously sent and repeated.

When you press the “AR” button while the CQ message is being sent, this message will be completed, and then the AR message, e.g. “ar pse k” will be sent once, before the system enters into idle mode.

Pressing the “AR” button when the CQ message is not currently being sent, starts the transmission of one AR message before the system goes back to idle.

You can stop any automated message by pulling low the “Dot” or “Dash” lines. In this case, the message is interrupted after having completed the current character, the system sends a “Dot” or “Dash” (depending on what line went low), and returns to idle mode, i.e. it continues monitoring the RB3...0 input lines.

This is the program:

```

; =====
; Programming the SX Microcontroller
; APP027. SRC
; =====
include "Setup28.inc"
reset    Main

TRIS    equ $0f
LVL     equ $0d
PLP     equ $0e

Freque  equ 125                ; Beep frequency

org     $08                    ; Global registers
FsrSave ds 1                   ; Storage for FSR
Speed   ds 1                   ; Morse code speed
State   ds 1                   ; ISR Morse handler state

```

```

Flags      ds 1          ; Flags to control the mainline
                    ; program
Ix         ds 1          ; Message table index
Count      ds 1          ; Storage for "Morse Bitcount"
Char       ds 1          ; Storage for Morse character

SendCq     = Flags. 0    ; State flags for mainline program
SendAr     = Flags. 1
SendDot    = Flags. 2
SendDash   = Flags. 3
Tone       = State. 7    ; When this flag is set, a Dot or Dash
                    ; is generated instead of a pause,
                    ; and the speaker pin RB7 is toggled
                    ; so generate a tone (Frequ deter-
                    ; mines the tone frequency)

org        $10          ; Registers used by the ISR
IsrVars    = $
PortBBuff  ds 1          ; Buffer for Port B data
Beep       ds 1          ; Beep timer for speaker output
Length     ds 1          ; Determines the length of the
                    ; current signal or pause in
                    ; multiples of a dot length
SpeedTimer ds 1          ; This timer controls the length
                    ; of a dot, i.e. the CPS of the
                    ; Morse code
BaseTimer  ds 1          ; This is the basic timer used to
                    ; divide down the ISR call frequency

org        $30          ; ADC registers
ADC        = $
AdcCount   ds 1          ; Overflow counter
AdcAcc     ds 1          ; Accumulator
PortCBuff  ds 1          ; Buffer for Port C data

org        $000

;-----
ISR ; The Interrupt Service Routine
;-----
    mov     FsrSave, FSR          ; Save the original FSR for later
                    ; restore

;-----
; ADC VP      rc. 1 = ADC in, rc. 0 = charge/discharge
;-----

bank      ADC
mov       PortCBuff, rc
and       PortCBuff, #%11111100 ; Mask ADC pins
mov       w, >>rc
not       w
and       w, #%00000011
or        PortCBuff, w

```

Programming the SX Microcontroller

```
sb      PortCBuff. 0
  incsz AdcAcc
inc     AdcAcc
dec     AdcAcc
mov     w, AdcAcc
inc     AdcCount
snz
  call  PotAdjust          ; Transform ADC value (0...255)
                          ; into 32...160

snz
  mov   Speed, w          ; Save the transformed value
snz
  clr   AdcAcc
mov    rc, PortCBuff     ; Set the ADC pins
page   ISR               ; Reset the page (changed
                          ; by PotAdjust)

;-----
; Morse code handler
;-----

bank   IsrVars

; If a button is down, set the associated flag for the
; mainline program.
;
sb     rb. 0              ; Dot line
  setb SendDot
sb     rb. 1              ; Dash line
  setb SendDash
sb     rb. 2              ; CQ button
  setb SendCq
sb     rb. 3              ; AR button
  setb SendAr

; Morse timer states
;
Idle      = 0
Pause1    = 1
Pause3    = 2
Pause5    = 3
Dot       = 4
Dash      = 5
Delay     = 6
DelayS    = Delay | $80

mov      w, State        ; Get the current state
and      w, #%01111111   ; and ignore bit 7
jmp      pc+w
jmp      :ExitIsr
jmp      :InitPause1
jmp      :InitPause3
jmp      :InitPause5
jmp      :InitDot
jmp      :InitDash
jmp      :Delay
```

```

: InitPause1                                ; Init for 1 dot-length pause
  mov     Length, #1
  jmp     : EndInitP

: InitPause3                                ; Init for 3 dots-lengths pause.
  mov     Length, #2                        ; As a one dot-length pause is
                                           ; automatically appended to each
                                           ; dot and dash, the actual pause
                                           ; length is 2 dots here.
  jmp     : EndInitP

: InitPause5                                ; Init for a 5 dots-lengths pause,
  mov     Length, #4                        ; (see above).

: EndInitP
  mov     SpeedTimer, Speed                ; Initialize the speed timer
  mov     State, #Delay                    ; Set next state
  jmp     : ExitIsr

: InitDot
  mov     Length, #1                        ; Setup for a dot
  jmp     : EndInitD

: InitDash
  mov     Length, #3                        ; Setup for a dash (3 dots-lengths)

: EndInitD
  mov     SpeedTimer, Speed                ; Initialize the speed timer
  mov     State, #DelayS                    ; Set next state
  jmp     : ExitIsr

: Delay ; Cause a delay
  decsz   BaseTimer                        ; Decrement the base timer
  jmp     : EndDelay
  decsz   SpeedTimer                       ; Decrement the speed timer
  jmp     : EndDelay
  mov     SpeedTimer, Speed                ; Re-initialize the speed timer
  decsz   Length                           ; Decrement the length counter
  jmp     : EndDelay
  sb      Tone                              ; If a dot or dash is finished,
  jmp     : EndPause
  mov     State, #Pause1                    ; automatically add a 1 dot-length
  jmp     : EndDelay                        ; pause

: EndPause
  mov     State, #Idle                      ; When a pause has been finished, idle

: EndDelay
  clrb   PortBBuff. 6                      ; Prepare the LED bit
  sb     Tone                              ; When the Tone flag is clear,
  jmp     : ExitIsr                         ; no LED and no sound, else
  setb   PortBBuff. 6                      ; turn LED on and

```

Programming the SX Microcontroller

```
    decsz    Beep                ; decrement the Beep timer
    jmp     :ExitIsr            ;
    xor     PortBBuff, #S80      ; Toggle the beeper pin
    mov     Beep, #Frequ        ; Re-initialize the Beep timer

:ExitIsr
    mov     rb, PortBBuff       ; Set the port pins
    mov     FSR, FsrSave        ; Restore the FSR
    mov     w, #-200           ; Call the ISR every 4 us
    reti w

;-----
; This routine reads the value indexed by w from Table.
; The table is used to transform the ADC values from 0 to 255
; into a range from 32 to 160, and to provide a better pot
; resolution for higher speed values.
;-----
PotAdjust
    page    Table
    jmp     w

org        $100

;-----
; The mainline program
;-----

Main
; Initialize the ports
;
mode PLP
mov !rb, #%11110000        ; Enable pull-ups on port B inputs
mode LVL                   ; Set cmos input levels
mov !rc, #0                ; on port C inputs
mode TRIS                   ; Setup inputs/outputs
clr rc
mov !rc, #%11111110        ; rc. 1 = ADC in
                                ; rc. 0 = charge/discharge

clr rb
mov !rb, #%00111111        ; rb. 7: beeper output,
                                ; rb. 6: LED output

clr PortBBuff              ; Clear some registers
clr Flags
clr State

mov !option, #10011111     ; Enable RTCC interrupt

; The bits in Flags are used to control the states of the main
; loop.

:MainLoop
    snb     SendDot            ; When the dot contact is closed, go
    jmp     :SendDot          ; and send a dot (highest priority)
    snb     SendDash          ; When the dash contact is closed,
```

```

    jmp    :SendDash          ; go and send a dash
test    Flags                ; If no flags are set at all, we are
snz     ;                   ; idle
    jmp    :MainLoop

mov     w, #ArTab           ; Prepare the address to send the
;                               ; "+ pse k" message
sb      SendAr              ; If not SendAr, prepare the address
mov     w, #CqTab          ; to send the "cq cq cq de..."
;                               ; message
mov     Ix, w               ; Setup the message pointer
snb     SendAr              ; If the user has pushed the AR
;                               ; button, while sending the "cq cq..."
;                               ; message, stop the CQ message after
;                               ; it is completed.
    clrb  SendCQ

:Loop
mov     w, Flags            ; Get the flags
and     w, #%00001100      ; Mask the SendAr and SendCq flags
sz      ;                   ; If no flags are set,
    jmp    :MainLoop        ; don't send a message
snb     SendCQ              ; While sending the "cq..." message,
    jmp    :NoDebounce      ; AR button pushes are accepted.
snb     SendAr              ; While sending the "+ pse k" message,
    clr   Flags             ; no more AR button pushes are
;                               ; accepted.

:NoDebounce
mov     m, #CqTab >> 8    ; Setup m: w for iread
mov     w, Ix
iread   ;                   ; Read the table item indexed by Ix
mov     Char, w            ; Save the dash/dot pattern
mov     Count, m           ; Save the dash/dot count
test    Count              ; When Count = 0, we have either a
snz     ;                   ; pause, or an end of table. We
    jmp    :TestEnd         ; go to :TestEnd to find out.

:Next
rl      Char               ; Output a Morse character
;                               ; Next bit -> c
mov     w, #Dash           ; Prepare for a dash
sc      ;                   ; If c is set, it is a dash,
;                               ; else a dot
    mov   w, #Dot
    mov   State, w         ; Setup the state

:Send
test    State              ; Wait until the ISR has sent the
sz      ;                   ; dash or dot
    jmp    :Send           ;
dec     Count              ; Decrement the dash/dot count
sz      ;                   ; If there are more dashes/dots to be
;                               ; sent, loop back
    jmp    :Next           ;
mov     State, #Pause3     ; Generate a 3 dots-length pause

:Pause

```

Programming the SX Microcontroller

```
test    State          ; Wait until the ISR has generated
sz      ; the pause
    jmp    :Pause

inc     Ix              ; Next character in the message table
jmp     :Loop

:TestEnd ; When a table item has a Count of 0,
test    Char           ; it is either a Pause5 (Char != 0),
snz     ; or the end of the table (Char == 0)
    jmp    :MainLoop
mov     State, #Pause5 ; Setup state for Pause5
jmp     :Pause         ; Go and wait until the pause is done

:SendDot
mov     State, #Dot    ; Send a dot when the user has pressed
jmp     :WaitSend     ; the dot button

:SendDash
mov     State, #Dash   ; Send a dash when the use has pressed
                    ; the dash button

:WaitSend
test    State          ; Wait until the ISR has sent the
sz      ; dash or the dot
    jmp    :WaitSend
clr     Flags          ; Clear the flags in order to stop
jmp     :MainLoop     ; any message being currently sent.

org     $200

; The message tables
;
; The first four bits in each item specify the number of
; dashes and dots. The next eight bits specify from "left
; to right" the dashes (1) and dots (0).
;
; A value of $001 specifies a 5 dot-lengths pause, and
; a value of $000 specifies the end of a message.
;
CqTab
dw      $001           ; Pause 5
dw      $400 + %10100000 ; C
dw      $400 + %11010000 ; Q
dw      $001           ; Pause 5
dw      $400 + %10100000 ; C
dw      $400 + %11010000 ; Q
dw      $001           ; Pause 5
dw      $400 + %10100000 ; C
dw      $400 + %11010000 ; Q
dw      $001           ; Pause 5
dw      $300 + %10000000 ; D
dw      $100 + %00000000 ; E
dw      $001           ; Pause 5
dw      $300 + %10000000 ; D
dw      $300 + %10100000 ; K
```

```

dw      $500 + %00001000      ; 4
dw      $100 + %10000000      ; T
dw      $100 + %10000000      ; T
dw      $001                    ; Pause 5
dw      $300 + %10000000      ; D
dw      $300 + %10100000      ; K
dw      $500 + %00001000      ; 4
dw      $100 + %10000000      ; T
dw      $100 + %10000000      ; T
dw      $001                    ; Pause 5
dw      $300 + %10000000      ; D
dw      $300 + %10100000      ; K
dw      $500 + %00001000      ; 4
dw      $100 + %10000000      ; T
dw      $100 + %10000000      ; T
dw      $001                    ; Pause 5
dw      $000                    ; End
ArTab
dw      $001                    ; Pause 5
dw      $500 + %01010000      ; AR
dw      $001                    ; Pause 5
dw      $400 + %01100000      ; P
dw      $300 + %00000000      ; S
dw      $100 + %00000000      ; E
dw      $001                    ; Pause 5
dw      $300 + %10100000      ; K
dw      $000                    ; End
org
Table  $400
retw   32, 32, 32, 32
retw   33, 33, 33, 33
retw   34, 34, 34, 34
retw   35, 35, 35, 35
retw   36, 36, 36, 36
retw   37, 37, 37, 37
retw   38, 38, 38
retw   39, 39, 39
retw   40, 40, 40
retw   41, 41, 41
retw   42, 42, 42
retw   43, 43, 43
retw   44, 44, 44
retw   45, 45, 45
retw   46, 46, 46
retw   47, 47, 47
retw   48, 48, 48
retw   49, 49, 49
retw   50, 50
retw   51, 51
retw   52, 52
retw   53, 53
retw   54, 54
retw   55, 55

```

Programming the SX Microcontroller

```
retw 56, 56
retw 57, 57
retw 58, 58
retw 59, 59
retw 60, 60
retw 61, 61
retw 62, 62
retw 63, 63
retw 64, 64
retw 65, 65
retw 66, 66
retw 67, 67
retw 68, 68
retw 69, 69
retw 70, 70
retw 71, 71
retw 72, 72
retw 73, 73
retw 74, 74
retw 75, 75
retw 76, 76
retw 77, 77
retw 78, 78
retw 79, 79
retw 80, 80
retw 81, 81
retw 82, 82
retw 83, 83
retw 84, 84
retw 85, 85
retw 86, 86
retw 87, 87
retw 88, 88
retw 89, 89
retw 90, 90
retw 91, 91
retw 92, 92
retw 93, 93
retw 94, 94
retw 95, 95
retw 96
retw 97, 97
retw 98, 98
retw 99, 99
retw 100, 100
retw 101, 101
retw 102, 102
retw 103, 103
retw 104, 104
retw 105, 105
retw 106, 106
retw 107, 107
retw 108, 108
retw 109, 109
retw 110, 110
```

```
retw 111, 111
retw 112, 112
retw 113, 113
retw 114, 114
retw 115, 115
retw 116, 116
retw 117, 117
retw 118, 118
retw 119, 119
retw 120, 120
retw 121, 121
retw 122, 122
retw 123, 123
retw 124, 124
retw 125, 125
retw 126, 126
retw 127, 127
retw 128, 128
retw 129, 129
retw 130, 130
retw 131, 131
retw 132, 132
retw 133, 133
retw 134, 134
retw 135, 135
retw 136
retw 137
retw 138
retw 139
retw 140
retw 141
retw 142
retw 143
retw 144
retw 145
retw 146
retw 147
retw 148
retw 149
retw 150
retw 151
retw 152
retw 153
retw 154
retw 155
retw 156
retw 157
retw 158
retw 159
retw 160
```

In this application, the mainline program initializes the ports and some registers, and then enters into a loop that handles sending pre-defined messages. It also handles the button-down and

Programming the SX Microcontroller

Morse-key device events but it does not read the associated input lines. Instead, it evaluates the states of the flags eventually set by the ISR which actually reads the input lines.

The ISR runs the ADC VP that is used to read the current setting of the speed potentiometer. The ISR also polls the input lines at Port B, and takes care of the necessary timing to generate “Dots”, “Dashes”, pauses, and the audio signal for the piezo speaker.

The timer part of the ISR is designed as a state engine. Depending on the value of the **State** variable, different parts of the ISR code are executed.

As pauses, “Dots”, and “Dashes” all require a time delay with the only difference that “Dots” and “Dashes” must control the output signal, and also must generate the audio signal for the speaker, bit **Status. 7** has a special meaning. If this bit is set, “Dots” or “Dashes” are sent, i.e. the speaker and the output lines will be activated in this case.

The two pre-defined messages (the CQ and the AR message) are stored in program memory. The program uses the **iread** instruction to read items from those tables.

As Morse code characters have variable lengths, the first four bits of a table entry are used to specify the total number of “Dots” and “Dashes” for each character. The remaining eight bits contain the Morse code pattern. Each “Dot” is represented by a 0-bit, and each “Dash” is represented by a 1-bit. The “Dash” and “Dot” codes are arranged from “left” to “right”, i.e. the first code element is located at bit 7, the second one at bit 6, etc. When a Morse character has less than eight “Dashes” and “Dots”, the remaining lower bits are cleared.

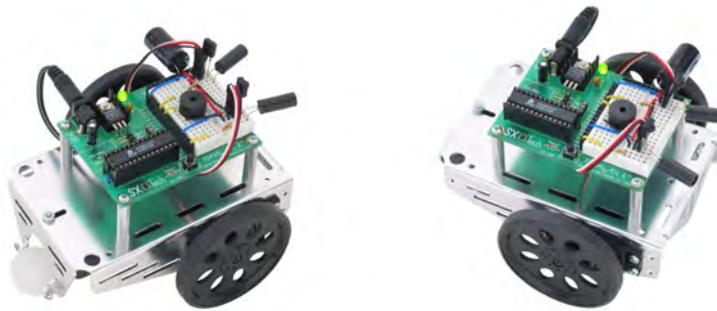
To either indicate a pause of five dot-lengths or the end of a message, the upper four bits of a table entry are cleared. When the lower eight bits are also cleared, this means that the end of the table has been reached. If the lower eight bits contain %00000001, a pause of five dot-lengths will be generated instead.

The ADC VP used here returns a result from 0 through 255 for an input voltage between 0 and 5 V. Using this full range to setup the **Speed** timer would result in extremely high and low Morse speeds when the potentiometer is turned to its two end-positions. In addition, when you use a linear potentiometer, the speed variation is very sensitive at higher speeds. Therefore, we use a table in order to convert the ADC values (0...255) into a range from 32 through 160 and to “flatten” the potentiometer curve at higher speed values.

4.16 Robotics - Controlling the Parallax SX Tech Bot

4.16.1 Introduction

The Parallax SX Tech Bot shown below is a small battery-powered autonomous robot with two drive wheels in front and a 1" polyethylene ball at its tail. This design is based on the popular Parallax Boe-Bot™ robot, which uses the BASIC Stamp® 2 module for its programmable controller on the Board of Education® prototyping platform. The SX Tech Bot uses the SX microcontroller for its programmable brain, and the SX Tech board for its prototyping platform. If you are interested in experimenting with this robot, it consists of two parts kits, the SX Tech Tool Kit (Part #45180) and the Boe-Bot Parts Kit (Part #28124).



The SX Tech board comes with all the components to run an SX-28 controller, which is inserted into the board's LIF (low insertion force) socket. The board has a 5V voltage regulator, header sockets for all SX I/O pins plus some other signals, and a small breadboard prototyping area. We will use this prototyping area to build and test sensors and indicators for the SX Tech Bot.

The chassis also has a battery holder installed for four 1.5 V AA batteries, so you can run the robot without an external power source. Simply plug the power connector leading from the battery pack into the SX-Tech Board's 6-9 VDC power jack. Nevertheless, while doing the first tests, it might be a good idea to use an external power supply. A DC supply rated for an output of 7.5 V, 1000 mA with a center-positive, 2.1 mm plug is recommended. NOTE: The supply's output rating can be from 6 to 9 VDC with a capacity of 600 mA or more.

The SX Tech Bot's two front wheels are driven by two premodified modified RC hobby servos, called Parallax Continuous Rotation servos. Both Standard and Continuous Rotation servos have an output shaft which is controlled by the circuitry inside the servo. A Standard servo is designed to rotate its output shaft to particular position and hold that position. The position is dictated by the control signal it receives. In contrast, a Parallax Continuous Rotation servo makes

Programming the SX Microcontroller

its output shaft rotate at a particular speed in a particular direction. The speed and direction is dictated by the same type of signal that is used to control the standard servo's position.

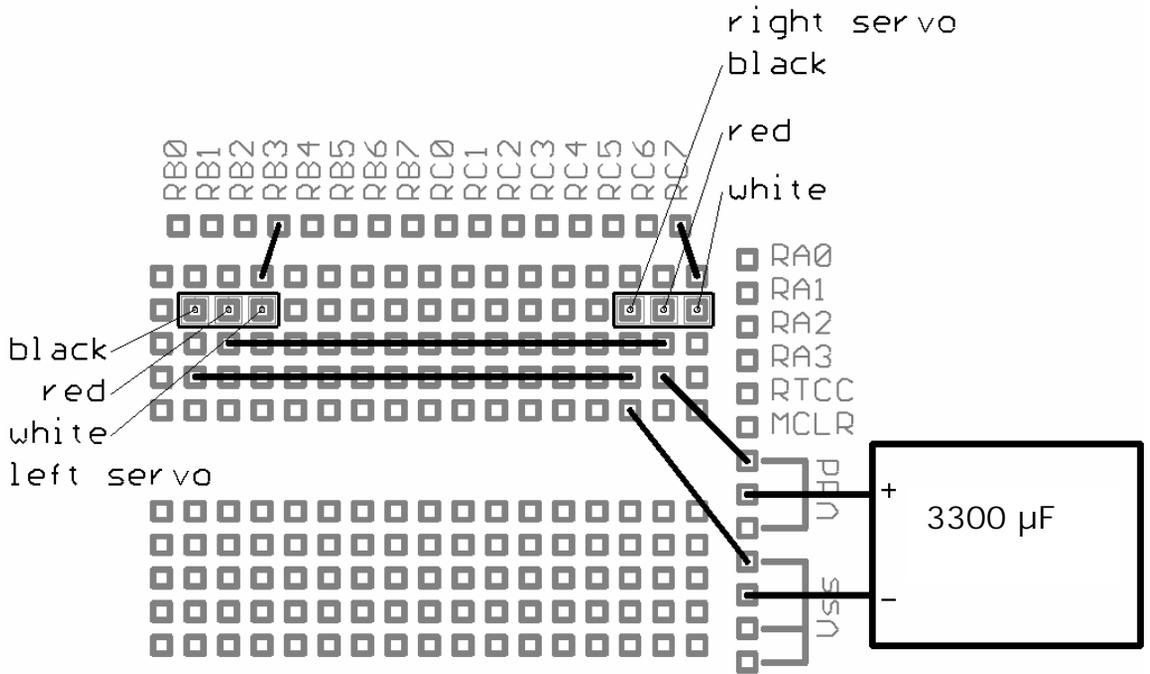
All previous code examples in this book were designed for SX controllers clocked at 50 MHz. The sample code in this chapter assumes an SX clocked at 4 MHz using an external, 4 MHz resonator. Since the SX Tech Bot relies on AA batteries for its power, the 4 MHz clock rate is a better choice since the SX draws significantly less current at lower clock rates. The drawback of lower clock rates, of course, is that the resolution of the incremental timing changes that can be made to output signals and sampling rates is much lower.



A precise external clock, such as the 4 MHz (accurate to +/- 0.3 %) ceramic resonator included in the SX-Tech Toolkit, is recommended for autonomous SX Tech Bot applications. In contrast, the SX microcontroller's internal oscillator is not recommended for these applications. Although it can supply a clock rate of 4 MHz, the IRC calibration can only guaranty an error within +/- 8 % at a given temperature. Additional programming techniques can be used to reduce this variation to around 1 %. Even so, this variation will still be noticeable if you are attempting to recalibrate the servos without the aid of the SX-Key, and the differences will be accentuated by changes in temperature.

Before assembling your SX Tech Bot, you will probably need to perform some tests and mechanical adjustments on the servos. The servos are connected to the SX Tech Board and a test and adjustment program is run. After the mechanical adjustments and potentially some software adjustments are made, you can then assemble your SX Tech Bot without having to worry about having to disassemble it again. Follow the instructions through Section 4.16.3.1 first. After that, you can then move on to the mechanical assembly instructions available from the www.parallax.com web site.

The drawing, below, shows how the two servos can be connected to power, ground, and SX I/O pins for control signals. Place two three-pin headers into the second row of the breadboarding area as shown, and also place the jumper wires as shown, to connect the servo inputs to the SX port pins RB3, and RC7, and to the power supply lines Vdd (+5V) and Vss (Ground). Use only insulated jumper wires, and as always, only make changes to circuits when power is disconnected. Also, make sure to correctly follow the color coding indications in the figure when connecting your servos to the three-pin headers.



As the servos consume quite an amount of starting current, it is important to connect an electrolytic capacitor (3300 μF , 6 V or higher) across Vdd and Vss, to avoid that the SX resets due to supply-voltage drops.



WARNING: When connected properly, these capacitors store the additional charge required by the servo motors during starts and sudden direction changes. However, when connected incorrectly, in reverse polarity, these capacitors can rupture or even explode. So, follow these connection instructions carefully. The capacitor's positive lead is denoted by a longer lead, and the negative lead is denoted by a stripe on the metal canister with negative signs. Make sure to verify that capacitor's positive lead is connected to Vdd, and that the negative lead is connected to Vss before connecting power to the system.

Finally, plug the servo connectors on to the two three-pin headers, and make sure that the orientation of the connectors is correct, i.e. that the color order of the wires follows the one shown in the drawing. The white wire (input) from the left servo should be connected to RC7, and the one from the right servo should go to RB3.

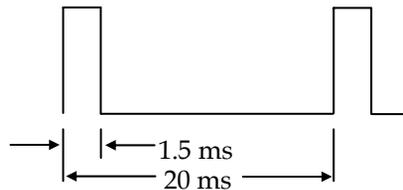
In the next section, we will address how the servos are controlled and discuss some general concepts for a basic SX Tech Bot application. We will then discuss an introductory SX program

Programming the SX Microcontroller

that takes care of the functions, necessary to control the SX Tech Bot, and we'll use this code as a "skeleton" for more examples in following sections.

4.16.2 Controlling the SX Tech Bot Servos

The servos that are used as the SX Tech Bot's "motors" are quite similar to the servos commonly used for RC models. In an RC model, such servos – for example – are used to move a rudder to any position between two left and right end positions. The servos have an input where they expect a PWM signal with a certain high time to control the servo position. Here is the typical timing diagram for a servo signal that instructs the servo to hold its "center" position:



As you can see, the time between two pulses is 20 ms. This duration is not critical for servo control, and any time between 5 and 45 ms will suffice. The pulse width is the signal that must be precise for accurate servo control.

A pulse width of 1.5 ms means that a Standard servo moves to, and holds at its center (or zero) position. This is the mid-point position in a servo's range of motion. When the pulse width becomes larger than 1.5 ms, the servo's output shaft rotates to a position counterclockwise of center, and vice versa, when the pulse width is less than 1.5 ms, the servo's output shaft rotates to a position clockwise of the center.

Instead of holding a particular position, a Parallax Continuous Rotation servo responds by rotating its output shaft counterclockwise when it receives pulses that last longer than 1.5 ms. Likewise, its output shaft rotates clockwise when it receives pulses less than 1.5 ms. The speed of rotation depends on the difference between the current pulse-width, and the 1.5 ms "stop value". The greater the difference, the faster the rotation speed will be. At pulse-widths of 1.7 ms, or 1.3 ms, the Parallax Continuous Rotation servos rotate at their maximum counterclockwise or clockwise speeds. So, it does not make sense to send PWM signals to the servo inputs with pulse widths above, or below these values.



Parallax Continuous Rotation servos are actually Standard servos that have been modified. The reason a continuous rotation servo's output shaft turns instead of holding a particular position is because the link between the its output shaft and the feedback potentiometer its circuitry uses to determine the output shaft's position has been severed. When a continuous rotation servo receives a signal that would tell a standard servo to rotate to and hold a particular position, the missing link between the continuous rotation servo's output shaft and feedback potentiometer fools its circuitry into thinking that it never arrives at the proper position. Thus, the servo's circuitry continues to drive its built-in DC motor in an attempt to reach a position it never gets to. The end result is "continuous rotation".

4.16.3 The Basic Control Program

The program listing, below, shows the basic servo control program:

```

; =====
; Programming the SX Microcontroller
; APP028.SRC
; =====
device SX28L, oscxt2, turbo, stackx
freq 4_000_000
IRC_CAL IRC_FAST

reset Main

LServo = RC.7 ; Output to servo - Left.
RServo = RB.3 ; Output to servo - Right.

LStop = 115 ; Adjust values so that the servos don't move
RStop = 115 ; when Speed = 0 and Turn = 0

org 8
Timer20L ds 1 ; Counters for
Timer20H ds 1 ; 20 ms timer
Ltimer ds 1 ; Counter for left servo timer
Rtimer ds 1 ; Counter for right servo timer
LSpeed ds 1 ; Left servo speed
RSpeed ds 1 ; Right servo speed
Speed ds 1 ; The "Bot's" speed
Turn ds 1 ; The "Bot's" turn factor

org 0
ISR
sb LServo ; Is left servo still on?
jmp :Right ; no - handle right servo
dec LTimer ; yes - count down
sz ; Left timeout?
jmp :Right ; no - handle right servo
clrb LServo ; yes - left servo off
mov LTimer, LSpeed ; Init left timer for next pulse

```

Programming the SX Microcontroller

```
: Ri ght
    sb      RServo          ; Is right servo still on?
    jmp    :Timer20       ; no - handle 20 ms timer
    dec    Rtimer         ; yes - count down
    sz     ; Right timeout?
    jmp    :Timer20       ; no - handle 20 ms timer
    clrb   RServo         ; yes - right servo off
    mov    RTimer, Rspeed ; Init right timer for next pulse
:Timer20
    dec    Timer20L       ; Handle the 20 ms timer
    sz     ; Count down low order byte
    jmp    :ExitISR      ; Is it zero?
    mov    Timer20L, #171 ; no - exit
    dec    Timer20H       ; yes, initialize and
    sz     ; count down high order byte
    jmp    :ExitISR      ; Is it zero?
    mov    Timer20H, #9   ; no - exit
    setb   LServo         ; yes, initialize and
    setb   RServo         ; turn the servos
:ExitISR
    mov    w, #-52        ; on again
    retiw                  ; ISR is invoked every 13 µs at
                          ; 4 MHz system clock

; Subroutine calculates the required values for LSpeed and RSpeed based upon
; the calibration factors, and the Speed and Turn parameters.
;
; Note: The routine does not check if the resulting values for LSpeed and
; RSpeed are out of limits (100...130).
;
Cal cVal ues
    mov    LSpeed, #LStop ; Initialize left speed to stop
    mov    RSpeed, #RStop ; Initialize right speed to stop

    add    LSpeed, Speed  ; Add the Speed value
    sub    RSpeed, Speed  ; Subtract the Speed value

    add    LSpeed, Turn   ; Add the turn value
    add    RSpeed, Turn   ; to both speeds
    ret

Mai n
    clrb   LServo         ; Clear servo outputs
    clrb   RServo         ;
    mov    !rb, #%11110111 ; RB.3 is output for left servo
    mov    !rc, #%01111111 ; RC.7 is output for right servo

    mov    !option, #%00001000 ; Enable interrupts

    mov    Speed, #0
    mov    Turn, #0
    call   Cal cVal ues

    jmp    $              ; Main program loops forever
```

As we will have to generate precisely timed PWM signals, it is obvious that we make use of an ISR that is periodically invoked on RTCC overflows. Therefore, some calculations are in order first:

The SX is clocked with 4 MHz here, and so the clock period is 250 ns. When we return from the ISR with the RETIW instruction, we have loaded -52 into w before, i.e. the ISR will be called every 13 μ s. You may wonder why we use such an “odd” timing here. This will become obvious later, when we discuss a SX Tech Bot with infrared obstacle detection, for now, just accept this value.

As you know, the PWM signal for each servo should have a positive edge every 20 ms, and a negative edge 1.3 to 1.7 ms later, depending on the desired servo speeds and directions.

For a delay of 20 ms, approximately 1,539 ISR calls (20 ms/13 μ s) are required. A division factor of 1,539 can be achieved by two nested decrementing counters, where one is initialized to 9, and the other to 171 (171 * 9 = 1,539).

A delay of 1.5 ms (this is the pulse width for a centered servo) requires approximately 115 ISR calls (1.5 ms/13 μ s = 115.38).

At the beginning of the program, we have defined some variables:

```

Timer20L    ds 1          ; Counters for
Timer20H    ds 1          ; 20 ms timer
Ltimer      ds 1          ; Counter for left servo timer
Rtimer      ds 1          ; Counter for right servo timer
LSpeed     ds 1          ; Left servo speed
RSpeed     ds 1          ; Right servo speed

```

Two variables, **Timer20L**, and **Timer20H** are the low and high counters for the 20 ms timing, **Ltimer**, and **Rtimer** are the counters for the pulse widths for the left and right servos. **LSpeed** and **RSpeed** contain the current speed values for the two servos. Let’s assume for now, that they are both are initialized to 115. **LServo** and **RServo** are symbolic names for RB.3 and RC.7, the SX output pins, where the two servo inputs are connected.

The first instructions in the ISR code

```

ISR
  sb    LServo          ; Is left servo still on?
  jmp   :Right          ; no - handle right servo
  dec   LTimer          ; yes - count down
  sz           ; Left timeout?
  jmp   :Right          ; no - handle right servo
  clrb  LServo          ; yes - left servo off
  mov   LTimer, LSpeed ; Init left timer for next pulse

```

handles the pulse for the left servo. If the servo output is still high, **LTimer** is decremented each time the ISR is invoked until **LTimer** becomes zero. In this case (after 1.5 ms when **LTimer** was

Programming the SX Microcontroller

initialized to 115 before), the servo output is set to low, and **LTimer** is re-initialized with the contents of **LSpeed** (115 for now). In case the servo output is already low, execution continues at:

```
: Right
    sb      RServo          ; Is right servo still on?
    jmp    :Timer20        ; no - handle 20 ms timer
    dec    Rtimer          ; yes - count down
    sz     ; Right timeout?
    jmp    :Timer20        ; no - handle 20 ms timer
    clrb   RServo          ; yes - right servo off
    mov    Rtimer, Rspeed  ; Init right timer for next pulse
```

This code performs the similar actions for the right servo, as described for the left servo, before. When this code is done, or when the right servo output is already low, execution continues with:

```
: Timer20
    dec    Timer20L        ; Handle the 20 ms timer
    sz     ; Count down low order byte
    jmp    :ExitISR        ; Is it zero?
    mov    Timer20L, #171  ; no - exit
    dec    Timer20H        ; yes, initialize and
    sz     ; count down high order byte
    jmp    :ExitISR        ; Is it zero?
    mov    Timer20H, #9    ; no - exit
    setb   LServo          ; yes, initialize and
    setb   RServo          ; turn the servos
: ExitISR
    mov    w, #-52         ; on again
    reti w                  ; ISR is invoked every 13 µs at
                           ; 4 MHz system clock
```

Here, the low-order counter of the 20 ms timer is decremented first. When it is not yet zero, the ISR is terminated. In case it is zero, it is re-initialized to 171, and the high-order counter is decremented. When this one becomes zero, it will be re-initialized to 9, and both servo inputs are set to high level then. This happens every 20 ms.

We will discuss the remaining parts of this program later. Let's first use the program "as is" to calibrate the servos.

Enter the program code using the SX-Key Editor, or open a copy from the Parallax CD, and assemble it. For now, you should use the SX-Key debugger to load and run the code on the SX. Calibration is easier when there is only one servo connected, so leave the connected to RC7, but disconnect the other servo from RB3.

4.16.3.1 Calibrating the Servos

If no errors are reported by the assembler, start the debugger, and run the program at full speed. The servo is likely to respond one way if it is labeled Parallax Continuous Rotation and another way if it is labeled Parallax PM (pre-modified).

If the servo is labeled Parallax Continuous Rotation, it will most likely rotate in an arbitrary direction because it is not yet been manually calibrated. With this newer type of Parallax servo, calibration is quite easy: Locate the small hole at the side of the servo housing near where the cable comes out. Behind this hole is a trim potentiometer. Insert a small Philips screwdriver, and slowly turn the potentiometer in one direction. If the servo rotates faster, turn the potentiometer in the other direction until you find the setting where the servo completely stops, and no longer produces a humming sound. This setting is quite critical, so turn the potentiometer very slowly and in small increments.

If the servo is labeled Parallax, and the letters PM are highlighted, the internal potentiometer is pre calibrated. The servo should either stay still, or rotate very slowly. If it rotates slowly, it's usually easier to make a small adjustment to the program to make the servo stay still. This is a more attractive option than disassembling the servo to correct the small adjustment error in its potentiometer.

Let's assume that the PM servo rotates slowly clockwise. The program can compensate for the small potentiometer offset by sending slightly wider pulses. Therefore, stop the debugger, and increase the initial value for **RStop** from 115 to 116. Re-assemble the program, and run it again. Should the servo still turn right, but at a slower speed, you need to further increase the initial value of **RStop**. If the servo starts turning in the opposite direction, the offset is too large, so decrease the value. If you are lucky, you will find the correct initial value for a complete stop after a while. Depending on the tolerances of the servo, and the relatively coarse timing of our program (we will discuss this later), you might not be able to exactly match that value, so at least find a value that slows down the servo as much as possible.

Next, disconnect the servo from RC7, and connect the other servo to RB3. Repeat the same calibration procedure just discussed for the second servo.

After you have calibrated your servos, you can construct your SX Tech Bot by following the instructions in *Robotics with the Boe-Bot*, available as a free download from www.parallax.com. While assembling your SX Tech Bot, there are two differences to keep in mind. First, when attaching the standoffs to the chassis, use the four holes that have the same pattern and dimensions as the hole pattern on the SX-Tech Board. Second, the SX Tech Bot's left servo should be connected to RC7, and its right servo should be connected to RB3.

4.16.3.2 More Parts of the Control Program

For the SX Tech Bot to operate autonomously, the 4 MHz ceramic resonator supplied with the SX Tech Toolkit should be inserted into the 3-socket header on the SX-Tech Board. The SX should then be programmed (CTRL-P), and finally, the SX-Key should be disconnected from the SX-Tech board. Since the resonator is supplying the clock signal, the `FREQ 4_000_000` for the SX-Key is no

Programming the SX Microcontroller

longer in effect. Instead, the **OSCXT2** directive sets the appropriate feedback and drive settings for the SX Tech Toolkit's ceramic resonator.



IMPORTANT: A common mistake is to unplug the SX-Key and wonder why the SX Tech Bot is not functioning. The SX Tech Bot will not function until the resonator is plugged-in. Also, when you are using the external resonator, always remember to remove before using the SX-Key for debugging. You can program the SX chip while the resonator is plugged in, but the Debugging tools will not work until the resonator is unplugged.

The main program code looks like this:

Main

```
clr b    L Servo          ; Clear servo outputs
clr b    R Servo          ;
mov     !rb, #01110111 ; RB. 3 is output for left servo
mov     !rc, #01111111 ; RC. 7 is output for right servo

mov     !option, #00001000 ; Enable interrupts

mov     Speed, #0
mov     Turn, #0
call    Cal cValues

jmp     $                ; Main program loops forever
```

At the very beginning, the servo output bits are cleared to avoid any “glitches” at startup, and then, the two port pins RB.3 and RC.7 are configured as outputs to control the two servo inputs.

Next, RTCC interrupts are enabled, and the two variables, **Speed** and **Turn** are both initialized to zero. The idea here is, to make the interface to the servo control code in the ISR as simple as possible. Instead of defining the initial values for the two variables **LSpeed** and **RSpeed** to control the pulse widths of the PWM signals for various SX Tech Bot moves and turns, we use **Speed** to control the forward/backward speed, and **Turn** to control the left/right turn rate.

When **Speed** is 0, the SX Tech Bot shall stop. For **Speed** > 0, the SX Tech Bot should move forward, and for **Speed** < 0, the SX Tech Bot should move backwards. When **Turn** is 0, the SX Tech Bot shall not turn at all; it will either go straight forward or straight backward, or stop, depending on the value of speed. When is **Turn** > 0, it should turn right, and when **Turn** is < 0, it should turn left. The greater the absolute values of **Speed** and **Turn** are, the faster the SX Tech Bot moves or turns in the specified directions. For now, the Main routine initializes both, **Speed** and **Turn** to 0, i.e. the SX Tech Bot should not move or turn at all. So in this mode, we can calibrate the servos.

Following the initialization of **Speed** and **Turn**, **Cal cValues** is called. This subroutine performs the necessary calculations to convert **Speed** and **Turn** into the initialization values that are stored

in **LSpeed** and **RSpeed**. We'll discuss this subroutine in a moment. Finally, the program enters into an endless loop because the remaining tasks are handled by the ISR for now.

Before discussing the **Cal cValues** routine, let's discuss some general considerations on the values for **Speed** and **Turn**, and the resulting settings of **LSpeed** and **RSpeed**. As mentioned before, both servos stop when **LSpeed** and **RSpeed** both contain 115 (or the values you have determined during "software servo calibration"). When the values are below 115, the servos turn clockwise, and on values above 115, they turn counterclockwise.

In order to have the SX Tech Bot move straight forward, the left servo must turn right at a certain counterclockwise speed, and the right servo must turn clockwise at the same speed. This means that **LSpeed** must be $115+v$, and **RSpeed** must be $115-v$. For a straight backward direction, the left servo must turn clockwise, and the right servo must turn counterclockwise. Therefore, **LSpeed** must be $115-v$, and for **RSpeed** it is $115+v$.

The SX-Tech Bot can also rotate in place to perform turns. When viewed from above, the SX-Tech Bot must rotate counterclockwise to perform a left turn, and clockwise to perform a right turn. For the SX-Tech Bot to perform a left turn, both its left and right wheels must rotate clockwise. Thus, **RSpeed** and **LSpeed** both are $115+t$. For the SX-Tech bot to turn right, both its wheels must turn counterclockwise, so **RSpeed** and **LSpeed** should both be $115+t$. The table, below, summarizes the various combinations, where v and t are now replaced by **Speed** and **Turn**, where both can be positive or negative:

Movement	Speed	Turn	Lspeed	RSpeed
Stop	0	0	$115 + \text{Speed} + \text{Turn}$	$115 - \text{Speed} + \text{Turn}$
Straight forward	> 0	0	$115 + \text{Speed} + \text{Turn}$	$115 - \text{Speed} + \text{Turn}$
Straight backward	< 0	0	$115 + \text{Speed} + \text{Turn}$	$115 - \text{Speed} + \text{Turn}$
Turn right	0	> 0	$115 + \text{Speed} + \text{Turn}$	$115 - \text{Speed} + \text{Turn}$
Turn left	0	< 0	$115 + \text{Speed} + \text{Turn}$	$115 - \text{Speed} + \text{Turn}$
Right and curve fwd	> 0	> 0	$115 + \text{Speed} + \text{Turn}$	$115 - \text{Speed} + \text{Turn}$
Left and curve fwd	> 0	< 0	$115 + \text{Speed} + \text{Turn}$	$115 - \text{Speed} + \text{Turn}$
Curve backward and right	< 0	< 0	$115 + \text{Speed} + \text{Turn}$	$115 - \text{Speed} + \text{Turn}$
Curve backward and left	< 0	> 0	$115 + \text{Speed} + \text{Turn}$	$115 - \text{Speed} + \text{Turn}$

The code for the **Cal cValues** subroutine converts these terms (115 , **Speed** and **Turn**) into the equivalent SX instructions:

Programming the SX Microcontroller

Cal cVal ues

```
mov    LSpeed, #LStop ; Initialize left speed to stop
mov    RSpeed, #RStop ; Initialize right speed to stop

add    LSpeed, Speed  ; Add the Speed value
sub    RSpeed, Speed  ; Subtract the Speed value

add    LSpeed, Turn   ; Add the turn value
add    RSpeed, Turn   ; to both speeds
ret
```

LSpeed and **RSpeed** first are initialized with the two stop constants (usually, 115), then **Speed** is added to **LSpeed**, and subtracted from **RSpeed**, and finally **Turn** is added to both, **LSpeed**, and **RSpeed**.

4.16.4 Some Timing Considerations

As already mentioned, the Parallax servos make maximum speeds at pulse widths of 1.3 ms (clockwise), and 1.7 ms (counterclockwise). This means that the values of **LSpeed** and **RSpeed** should not go above 130 (for 1.7 ms) or below 100 (for 1.3 ms). As **Cal cVal ues** does not check if the resulting values are out of limits you will need to take care of that when assigning values to **Speed** and **Turn**.

With the timing provided by the ISR, there are only 15 increments for **LSpeed** and **RSpeed** to control the servo speed into each direction from 0% to 100%. In other words, each increment corresponds to about 6.7%. This is a relatively coarse resolution, but for the next experiments, this is fine enough.

If you are using the older Parallax PM servos together with the “software calibration” you should now understand why you possibly could not find a value for **LStop** or **RStop** to completely stop the servos.

In order to increase the resolution, you might consider invoking the ISR more often, e.g. every 7.5µs by replacing the line

```
mov    w, #-52          ; ISR is invoked every 13 µs
```

with

```
mov    w, #-26          ; ISR is invoked every 7.5 µs
```

Besides adjusting the 20 ms timer (which is quite easy – simply initialize **Timer20H** with 18), this also means that the possible values for **LSpeed**, and **RSpeed** would range from 200 to 260 then. As an 8-bit counter can only handle a maximum divide-by 256 factor, this means that you would have to use two-byte counters for **LTimer** and **RTimer**, so more code in the ISR would be required to handle them. On the other hand, invoking the ISR every 26th clock cycle does not allow for total ISR execution times above 26 clock cycles. We are already close to that value, so no more code in the ISR would be possible (which we plan to add later). Besides this, the **Main** code would

not have much time to execute its own instructions because most of the time, it would be interrupted to service the ISR. For now, this is not a problem because **Main** just performs an endless loop, but we plan to add more instructions there later.

While increasing the SX clock frequency, say to 10 MHz, or even more, would allow for finer timing resolution, it would also cause the SX to consume more power. If you are interested in experimenting with higher clock rates, consider also replacing the four 1.5 V AA batteries with five or six 1.2 V AA rechargeable batteries, or a 7.5 V rechargeable battery pack. For now, however, let's be happy with the 4 MHz clock, and keep in mind that precision is not a major feature here.

4.16.5 The SX Tech Bot's First Walk (in the Park)

At this point, you should have completed the necessary servo calibrations and mechanical assembly of the SX Tech Bot, so now the time has come to send out the SX Tech Bot to make its first steps.

In the **Main** section of our program, simply assign values other than 0 to **Speed** to make the SX Tech Bot walk. Remember that positive values result in a forward movement, where negative values make it back up. You may also assign values other than 0 to **Turn** in order to let the SX Tech Bot turn around, or perform curves when both, **Speed** and **Turn** are other than 0. Feel free to experiment with various combinations of values for **Speed** and **Turn** (positive, or negative), but keep an eye on the resulting values for **LSpeed** and **RSpeed** – they should not exceed the limit from 100 to 130.

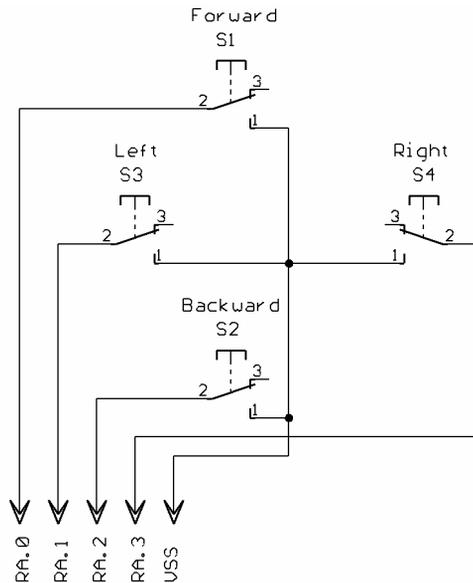
Keep in mind, after each modification to **Speed** or **Turn**, it will be necessary to re-load the program into the SX. This will not be necessary when the SX is making decisions based on sensor inputs.

4.16.6 Adding a “Joystick” to the SX Tech Bot

You may find it annoying to re-program the SX each time you want to make changes to the **Speed** and **Turn** assignments in the Main program, so here comes an improvement:

If you have one of the “antiquarian” joysticks on hand that came with four micro switches for “forward”, “backward”, “left”, and “right”, grab it from your junk box, and connect it to the SX Tech Bot. As an alternative, you could install four pushbuttons on a breadboard, according to the schematic, below:

Programming the SX Microcontroller



Connect the “Joystick Assembly” via a cable (the length depends on how much “freedom” you want to allow for the SX Tech Bot) to the four header sockets on the SX Tech board marked RA0 through RA3, and to one of the free Vss header sockets. Maybe, it is a good idea to solder the four leads going to RA.0 through RA.4 to a four-pin header, and the fifth lead going to Vss to a single header pin before plugging them in.

Change the **Main** section in our basic program to look like this:

```

Main
mode    $0e                ; Select PLP
mov     !ra, #%11110000    ; Activate pull-ups on port A
clrb   LServo
clrb   RServo
mode    $0f                ; Select TRIS
mov     !rb, #%11110111    ; RB. 3 is output for left servo
mov     !rc, #%01111111    ; RC. 7 is output for right servo
mov     !option, #%00001000 ; Enable interrupts

CheckSwitches
cjne   Timer20H, #9, $     ; Wait until after servo pulses have
cjne   Timer20L, #9, $     ; been delivered to check buttons.
mov     Speed, #0
mov     Turn, #0
snb    ra. 0               ; Forward button pressed?
      jmp :TestBack
mov     Speed, #7          ; Positive speed = forward
      jmp :TestLeft
:TestBack

```

```

        snb    ra. 2          ; Backward button pressed?
        jmp    :TestLeft
        mov    Speed, #-7    ; Negative speed = backward
:TestLeft
        snb    ra. 1          ; Left button pressed?
        jmp    :TestRight
        mov    Turn, #-7     ; Negative turn = left
        jmp    :TestEnd
:TestRight
        snb    ra. 3          ; Right button pressed?
        jmp    :TestEnd
        mov    Turn, #7      ; Positive turn = right
:TestEnd
        call   CalcValues
        jmp    CheckSwitches

```

In the beginning of this program version, we activate the internal pull-up resistors for all port A pins, so that we don't need to connect external pull-up resistors to the pushbuttons. Instead of an endless "do-nothing" loop, the main program executes code that checks to find out which pushbutton or pair of pushbuttons are pressed, and sets the values for **Speed** and **Turn** accordingly. You can press single buttons, such as forward, backward, left or right. You can also press and hold combinations of buttons such as forward and right, backward and left, etc.

Please note the order of how the buttons are checked. This makes it impossible to let the SX Tech Bot go "crazy" in case you push two opposite buttons, like Forward and Backward at the same time. When the Forward button is pressed, no check for the Backward button will be performed, and so it does not matter if you have pressed it, or not. The same is true for the Left and Right buttons.

For **Speed**, we use a value of 7 here, and 7 for **Turn** as well. In the event that one speed and one turn button are pressed at the same time, it causes the SX-Tech Bot to perform a pivot-turn, where one wheel stays still while the other turns the bot. This is different from a rotating turn, where both wheels turn in the same directions at the same speeds. This gives you eight possible maneuvers with four buttons.

4.16.7 The SX Tech Bot "Learns" to Detect Obstacles

So far, we did not really build a robot, but just a very simple "toy". Usually, robots are supposed to have some kind of "brainpower", and this is what we are going to add now.

The SX Tech Bot Kit comes with two infrared (IR) LEDs, and two infrared sensors. In this experiment, we will attach these components plus two resistors to the breadboarding area on the SX Tech board. The two IR LEDs are used like headlights of a car to illuminate any obstacles that might occur along the SX Tech Bot's path. The two sensors are used to detect the infrared light that will be reflected from such obstacles, and cause a change in the SX Tech Bot's maneuver.

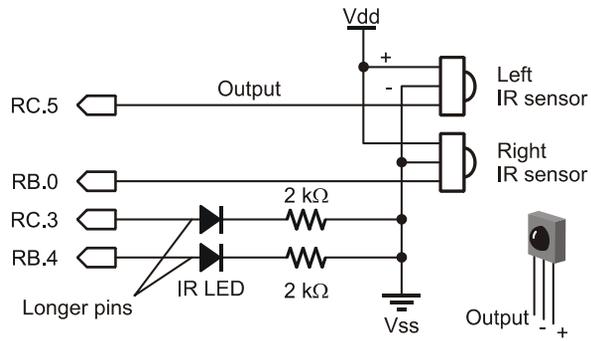
Programming the SX Microcontroller

The infrared sensors used here, have a built-in filter that makes them only sensitive for infrared light that is pulsed with a frequency of approximately 38.5 kHz. That is, the sensors will only react on infrared light that is turned on and off 38,500 times per second. This prevents interference with other infrared sources, like the sunlight (which is turned on and off only once per day), and other light sources powered by mains power. Such lights usually flash at 100 or 120 Hz, depending on the country where you live (with 50 or 60 Hz mains frequency).

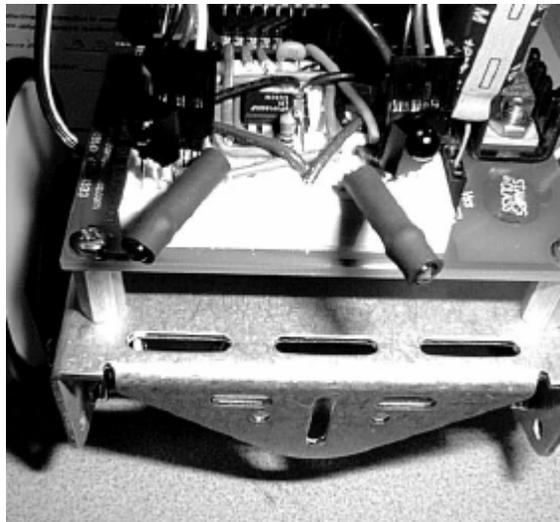
The schematic, below, shows the necessary wiring of the two infrared sensors, the infrared LEDs, and the two resistors that are necessary to limit the LEDs current. The photograph, further below, gives you an idea where, and how to position the IR sensors and LEDs on the breadboard. Adjust the left LED and sensor, to that they are “looking” about 30° to the left, and let the right LED and sensor “look” about 30° to the right.

Be careful to correctly connect both, the LEDs and the sensors. The longer LED pins go to the SX ports, and the shorter pins to the resistors. Please refer to the front view drawing of the sensor to correctly identify the three pins. Make sure that the SX Tech Bot's left sensor output (shown at the right side of the picture) is connected to RC.5, and the right sensor output (shown on the left side of the picture) is connected to RB.0. The same applies to the IR LEDs. The one on the SX Tech Bot's left side should be connected to RC.3, and the one on its right side should be connected to RB.4.

Also be careful that there are no short circuits between the leads as the breadboarding area is much more “crowded” now. If necessary, cover the leads with insulating tube, or use isolated wires.



Wiring the IR Sensors and LEDs



Aligning the IR Sensors and LEDs

Programming the SX Microcontroller

4.16.7.1 The Control Program for the Obstacle-Detecting SX Tech Bot

The program is an enhanced version of the basic version we have used to calibrate the servos. Here comes the program listing:

```

; =====
; Programming the SX Microcontroller
; APP029.SRC
; =====
device SX28L, oscxt2, turbo, stackx
freq 4_000_000
IRC_CAL IRC_FAST

reset Main

LServo = RC.7 ; Output to servo - left
RServo = RB.3 ; Output to servo - right
LSensor = RC.5 ; Input for left IR sensor
RSensor = RB.0 ; Input for right IR sensor
LLED = RC.3 ; Output for left IR LED
RLED = RB.4 ; Output for right IR LED
Calibrate = RA.0 ; Input for calibrate jumper

LStop = 115 ; Adjust values so that the servos don't move
RStop = 115 ; when Speed = 0 and Turn = 0

org 8
Timer20L ds 1 ; Counters for
Timer20H ds 1 ; 20 ms timer
Ltimer ds 1 ; Counter for left servo timer
Rtimer ds 1 ; Counter for right servo timer
LSpeed ds 1 ; Left servo speed
RSpeed ds 1 ; Right servo speed
Speed ds 1 ; The "Bot's" speed
Turn ds 1 ; The "Bot's" turn factor

org $30
Sensors ds 1

ISR org 0
sb LServo ; Is left servo still on?
jmp :Right ; no - handle right servo
dec LTimer ; yes - count down
sz ; Left timeout?
jmp :Right ; no - handle right servo
clrb LServo ; yes - left servo off
mov LTimer, LSpeed ; Init left timer for next pulse
:Right
sb RServo ; Is right servo still on?
jmp :Timer20 ; no - handle 20 ms timer
dec Rtimer ; yes - count down
sz ; Right timeout?
jmp :Timer20 ; no - handle 20 ms timer
clrb RServo ; yes - right servo off
```

```

: Timer20    mov    RTimer, Rspeed ; Init right timer for next pulse
            dec    Timer20L ; Handle the 20 ms timer
            sz     ; Count down low order byte
            jmp    :ExitISR ; Is it zero?
            mov    Timer20L, #171 ; no - exit
            dec    Timer20H ; yes, initialize and
            sz     ; count down high order byte
            jmp    :ExitISR ; Is it zero?
            mov    Timer20H, #9 ; no - exit
            setb   LServo ; yes, initialize and
            setb   RServo ; turn the servos
            ; on again

: ExitISR
            movb   LLED, Timer20L.0 ; Toggle both
            movb   RLED, Timer20L.0 ; IR LEDs
            mov    w, #-52 ; ISR is invoked every 13 µs at
            reti w ; 4 MHz system clock

; Subroutine calculates the required values for LSpeed and RSpeed based upon
; the calibration factors, and the Speed and Turn parameters.
;
; Note: The routine does not check if the resulting values for LSpeed and
; RSpeed are out of limits (100..130).

CalcValues
            mov    LSpeed, #LStop ; Initialize left speed to stop
            mov    RSpeed, #RStop ; Initialize right speed to stop

            add    LSpeed, Speed ; Add the Speed value
            sub    RSpeed, Speed ; Subtract the Speed value

            add    LSpeed, Turn ; no, add the turn value
            add    RSpeed, Turn ; to both speeds
            ret

Main
            mode   $0e ; Select PLP
            mov    !ra, #%11111110 ; Activate pull-up on pin 0
            clrb   LServo
            clrb   RServo
            mode   $0f ; Select TRIS
            mov    !rb, #%11110111 ; RB.3 is output for left servo,
            mov    !rc, #%01111111 ; RC.7 is output for right servo,
            mov    !option, #%00001000 ; Enable interrupts
            mov    Speed, #0
            mov    Turn, #0
            bank   Sensors

: Loop
            clr    Speed
            clr    Turn
            clr    Sensors
            sb     Calibrate ; Do nothing when the Calibrate
            jmp    :Loop ; jumper is in position
            cjne   Timer20H, #3, $ ; Wait for Timer20H = 3

```

Programming the SX Microcontroller

```
    mov    !rb, #%11100111          ; Right IRLED to output
    cjne   Timer20H, #2, $          ; Wait for Timer20H = 2
    mov    !rb, #%11110111          ; Right IRLED to input
    movb   Sensors.0, RSensor       ; Copy right IR detect bit
    mov    !rc, #%011110111        ; Left IRLED to output
    cjne   Timer20H, #1, $          ; Wait for Timer20H = 1
    movb   Sensors.1, LSensor       ; Copy left IR detect bit
    mov    !rc, #%011111111        ; Left IRLED to input
    mov    w, Sensors               ; Depending on the sensor states,
    jmp    pc+w                     ; jump to the state handler
:Both   jmp    :Both
        jmp    :Right
:Right  jmp    :Right
        jmp    :Left
:Left   jmp    :Left
        jmp    :None
:None   jmp    :None
:Done   mov    Speed, #-10          ; Both sensors detect, so
        jmp    :Done              ; back up
:Right  mov    Turn, #10           ; Right sensor detects, so
        jmp    :Done              ; turn left
:Left   mov    Turn, #-10          ; Left sensor detects, so
        jmp    :Done              ; turn right
:None   mov    Speed, #10          ; No obstacles at all, so
:Done   jmp    :Done              ; go forward
        call   CalcValues          ; Calculate LSpeed and RSpeed
        jmp    :Loop              ; Repeat it forever
```

Compared to the servo calibration program, we have added some more definitions for the port I/O pins that are connected to the IR sensors and LEDs now, and for a “Calibrate” input. We also have introduced a new variable, **Sensors** in bank \$30, two additional instructions in the ISR following the **:ExitISR** label, and added some code to the **Main** program.

As mentioned before, the IR sensors have built-in filters that let pass infrared light only that is pulsed at a frequency of approximately 38.5 kHz. This means that we need to turn the two IR LEDs on and off at that rate. This happens in the ISR due to the two new instructions:

```
:ExitISR
    movb   LLED, Timer20L.0 ; Toggle both
    movb   RLED, Timer20L.0 ; IR LEDs
```

Timer20L is decremented on each ISR call, i.e. every 13 μ s, so the LEDs are repeatedly turned on for 13 μ s, and turned off for another 13 μ s, or the on-off period is 26 μ s which is equivalent to a frequency of 38.46 kHz. This is close enough to 38.5 kHz. Now it becomes clear why we are using such an “odd” interrupt period of 13 μ s; it makes it really easy to “flash” the LEDs.

At the beginning of the **Main** section, we configure a pull-up resistor on pin 0 of port A, two additional port pins as outputs for the LEDs (RB.4 and RC.3), and select the bank for the **Sensors** variable.

The SX Tech Bot's "brainpower" lies in the new **: Loop** code with the **Main** section:

```

: Loop
  clr   Speed
  clr   Turn
  clr   Sensors
  sb    Calibrate           ; Do nothing when the Calibrate
  jmp   : Loop             ; jumper is in position
  cjne  Timer20H, #3, $    ; Wait for Timer20H = 3
  mov   !rb, #%11100111   ; Right IRLED to output
  cjne  Timer20H, #2, $    ; Wait for Timer20H = 2
  mov   !rb, #%11110111   ; Right IRLED to input
  movb  Sensors.0, RSensor ; Copy right IR detect bit
  mov   !rc, #%01110111   ; Left IRLED to output
  cjne  Timer20H, #1, $    ; Wait for Timer20H = 1
  movb  Sensors.1, LSensor ; Copy left IR detect bit
  mov   !rc, #%01111111   ; Left IRLED to input
  mov   w, Sensors        ; Depending on the sensor states,
  jmp   pc+w              ; jump to the state handler
  jmp   :Both
  jmp   :Right
  jmp   :Left
  jmp   :None
: Both
  mov   Speed, #-10       ; Both sensors detect, so
  jmp   :Done             ; back up
: Right
  mov   Turn, #10         ; Right sensor detects, so
  jmp   :Done             ; turn left
: Left
  mov   Turn, #-10        ; Left sensor detects, so
  jmp   :Done             ; turn right
: None
  mov   Speed, #10        ; No obstacles at all, so
  jmp   :Done             ; go forward
: Done
  call  CalcValues        ; Calculate LSpeed and RSpeed
  jmp   :Loop             ; Repeat it forever

```

At each entry into **: Loop**, we clear **Speed**, **Turn**, and **Sensors** for a clean start. We then test the port bit (RA.0) that is assigned to **Calibrate**. When this bit is clear, RA.0 has been connected to Vss in order to activate the calibration mode. In this case, we do nothing else. It is a good idea to make this mode available because normally, the SX Tech Bot would always be in motion. It may, from time to time, be necessary to stop the servos in order to re-calibrate them, especially in situations where the vibration from prolonged operation causes the manually adjusted potentiometer calibration setting to drift.

When calibrate mode is inactive, the first step is to wait for a full cycle of servo pulses to complete. For the sake of navigation, the effective sampling rate of checking the detectors 40 to 50 times per second is ample. This also prevents sampling and adjustment while the actual pulse is delivered, which could cause instability in a given pulse width.

Programming the SX Microcontroller

Although the IR LED I/O ports (RB4 and RC3) are toggled each time through the ISR, the IR LEDs do not flash on/off because the output bits for these ports are disabled since !RB.4 and !RC.3 are set to 1 (input). The program waits until the value of Timer20H has counted down to 3. At this point, the next step is to broadcast infrared to SX Tech Bot's right IR LED. This is accomplished by clearing !RB.4 (setting it to output). The signal is allowed until the ISR decrements the Timer20H variable, at which point the right IR detector's output is tested and stored in bit-0 of the **Sensors** variable. Then, !RB.4 is set, restoring RB4 to input and in turn stopping the right IR LED from broadcasting IR. This process is repeated for the SX Tech Bot's left IR LED and detector, and the result is stored in bit-1 of the sensors variable.

A key feature of this IR detection algorithm is that the detections are mutually exclusive. If both IR LEDs were to broadcast at the same time, both detectors might "see" an object that is really only on one side of the SX Tech Bot. So broadcasting and sampling on one side, then moving on to the other side ensures accurate detection of an object's position relative to the SX Tech Bot. Bit-1 of Sensors is 1 if an object on the SX Tech Bot's left is not detected, or 0 if an object is detected. Likewise, bit-0 of Sensors is 1 if an object on the right is not detected, or 1 if it is. As a result, the **Sensors** variable can only contain the values shown in the table, below:

Sensors	Obstacle to the...
0	left and right
1	right
2	left
3	no obstacles

After bits for both detectors are stored in the **Sensors** variable, navigation decisions can be made. We take this value as offset for a jump table that directs the program-flow to the right state handler. Depending on the current **Sensors** state, we set the according values for **Speed** and **Turn**, so that the SX Tech Bot changes its direction of movement, in order not to bump into the obstacle.

Please enter the program code into the SX-Key IDE, assemble, and transfer it into the SX Tech Bot's SX for "stand-alone" execution, because this type of SX Tech Bot would not be too agile with the SX-Key "umbilical cord" still connected. Note that the sample code contains the device configuration **OSCXT2**, i.e. the external ceramic resonator must be used. Remember to insert the resonator into the SX Tech Board's 3-socket header so that the program executes after the SX-Key is unplugged from the SX Tech Board.

Should you prefer to use the Debugging environment, make sure to set the SX Tech Bot on something that will prevent its wheels from touching the ground. You can then run the program, place obstacles at various positions in front of the SX Tech Bot, and verify that the wheel rotations indicate it is performing the correct maneuver.



IMPORTANT: This SX Tech Bot detection system performs well with reflective obstacles. For best results, use white cardboard boxes. Most colors of cardboard or paper boxes will work, as will your hand or foot so long as you are not wearing black shoes. Many black surfaces absorb infrared light so well that the SX Tech Bot will be blind to them. You can increase the SX Tech Bot's sensitivity to IR absorbing objects by reducing the value of the resistors in series with the IR LEDs. This also makes the SX Tech Bot more farsighted with reflective objects.

Keep in mind that you can test and recalibrate your SX Tech Bot's servos by connecting a jumper across the Vss and RA0 header sockets on the SX Tech board. When this connection is made, the servos should stay still. If they do rotate, or produce humming sounds, it is time for recalibration. When you are done with it, remove the jumper, and off you go...the SX Tech Bot will autonomously roam and avoid obstacles it detects.

In the sample code, we have used a relatively high speed setting for the continuous rotation servos, with **Speed** and **Turn** ranging from -10 to 10 depending on each maneuver. Feel free to experiment with other values, and combination of values. You may find that slower settings perform better in crowded areas. This in combination with adjustments to the IR LED series resistors can prepare your SX Tech Bot for a variety of obstacle courses.

4.16.7.2 Some More Thoughts About the Obstacle-Detecting SX Tech Bot

Let's think about what should happen (at least in theory) when the SX Tech Bot performs a forward move, perpendicular to a flat obstacle, like a wall:

When the wall comes into "sight" of the sensors, both will report an obstacle, and according to the program logic, the SX Tech Bot would back up, until the sensors no longer report this obstacle. Then, it will again move forward closer to the wall until the sensors "see" the wall again, which will be the case after a short while. This means that the SX Tech Bot would move back and forth forever, until its batteries are dead.

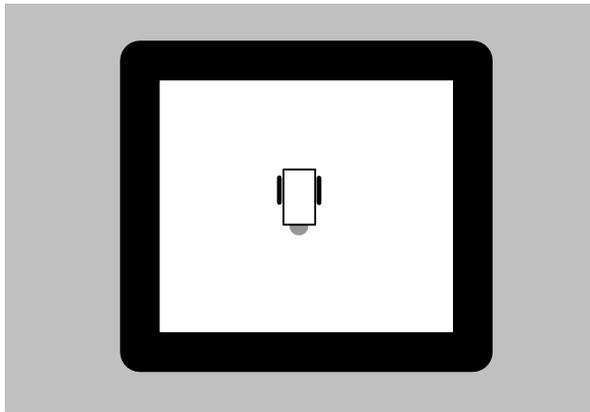
But this is only theory. Due to the coarse resolution of the PWM signals, the servos will never run completely synchronized. This means that the SX Tech Bot will soon leave the straight perpendicular line to the wall, and one sensor will detect the wall earlier than the other, making the SX Tech Bot turn. In other words, the lack of precision is an advantage here, adding some "fuzziness" to our system.

Programming the SX Microcontroller

Equipped with two infrared “headlights”, and “eyes”, the SX Tech Bot can easily be configured to react on “non-existing” obstacles. What does that mean at all?

You could align the IR LEDs and sensors to “look” at points on the surface, the SX Tech Bot is moving on. The **Main** code must be modified in a way that the SX Tech Bot moves straight forward as long as it “sees” an obstacle, that is the surface on which it is moving. In this case, it must react accordingly, when it detects an “abysm”, like the corner of the table it is moving on.

We leave it up to you, modifying the SX Tech Bot software to fulfill that task. When you try this, it is a good idea to first put a larger square of white cardboard, covered with black tape, some inches wide, on the floor, like this:



Here, the black tape acts as “abysm”, or “restricted area” without the danger that the SX Tech Bot drops down, in case it would ignore it. When you are sure that the program works as expected, and have verified that the SX Tech Bot does not “overshoot” the “restricted area”, you may actually try this experiment on a table. For safety reasons, it may be a good idea to run the SX Tech Bot at reduced speeds this time.

4.17 More Ideas for SX Tech Bot Applications

The examples shown here, are intended to give you a basic idea on how to control the SX Tech Bot servos using an SX controller, and how to “automate” the SX Tech Bot’s behavior.

You will certainly have other ideas in mind that could be realized, so the only limits are your imagination, and the precision of the SX Tech Bot.

Here are some tips for more experiments:

- Replace the IR components by two photo resistors (LDRs) that are “looking” at the floor, some inches in front of the SX Tech Bot at angles of about 30° to the right and to the left. You can then modify the SX Tech Bot program in a way that it follows a flashlight beam that you direct to the floor in front of the SX Tech Bot.
- Instead of optical devices, you could also attach some mechanical “whiskers” to the SX Tech Bot that pull two port pins low, when the right or left “whisker” touches an obstacle.
- Think of an SX Tech Bot with two sensors (IR sensors/LEDs, or mechanical “whiskers”), one “looking” to the right, and one “looking” forward. This could be used to help the SX Tech Bot find its way through a maze. Unfortunately, the SX does not have enough memory to store the shortest trace, but what about adding a serial EEPROM for additional storage capacity?
- As mentioned several times before, the resolution of the PWMs controlling the servos at 4 MHz system clock is quite coarse. Therefore, you may consider increasing the system clock, and modify the ISR code to handle shorter clock cycles. When designing a new timing concept, you should keep in mind to provide a “source” for the 38.5 kHz signal which is required to drive the LEDs for an infrared-based sensor system.

For more ideas on robotics, please visit the Parallax site at www.parallax.com, where you can find a lot of robotics-related material, including the text *“Robotics With the Boe-Bot, Student Guide”*. Although this text is intended to be used with the BASIC Stamp-controlled version of the SX Tech Bot, it contains many hints, concepts and ideas that you may port to the “World of SX Robotics”.

Programming the SX Microcontroller

A Complete Guide by Günther Daubach

2ND EDITION

Section V - Index

5 Index

–	
_mode macro	221
=	
= directive	53
7	
7-Segment display	314
A	
A/D Conversion	293
ADC	293
ADC, Bitstream calibration	301
ADD instruction	67, 253, 256
ADDB instruction	253, 256, 257
Addition	67
Addition, multi-byte	70
Addresses, symbolic	20
Addressing, direct	204, 206
Addressing, indirect	204, 206
Addressing, semi-direct	206
Alarm clock	338
ALU	228
Analog comparator	148, 217
Analog comparator and interrupts	150
Analog comparator and sleep mode	151
AND instruction	76, 253, 256
Application Examples	271
Arithmetic instructions	67
Assembler directives	28
Assembly, conditional	125
B	
BANK instruction	253, 259
Bank, saving in a subroutine	58
Banks	204
BCD display	323
Boe-Bot	425
Branches, conditional	31
BREAK directive	146
Breakpoints	26
Brown Out detection	227
C	
C flag	68, 229
CALL instruction	33, 253, 258
Capture/Compare mode	239
Carry flag	68
CJA instruction	172, 253, 258
CJAE instruction	172, 253, 258
CJB instruction	173, 253, 258
CJBE instruction	173, 253, 258
CJE instruction	173, 253, 258
CJNE instruction	174, 253, 258
CLC instruction	68, 253, 256
Clock generation	152
Clock Timer	307
Clock, crystal/ceramic resonator	153
Clock, digital	338
Clock, external	154
Clock, external R-C network	153
Clock, internal	152
Clock, PLL	154
Clock, selecting the frequency	155
CLR !WDT instruction	226
CLR instruction	253, 256
Clr2x.inc	52
CLRB instruction	253, 257
CLZ instruction	68, 253, 256
Comparator, analog	148, 217
Compare instructions	172
Compound instructions	17
Conditional assembly	125
Conditional branches	31

Programming the SX Microcontroller

Configuration directives.....	28	DW directive.....	169
Constants, symbolic.....	141	E	
Contacts, debouncing.....	99	ELSE directive.....	126
CSA instruction.....	175, 253, 258	END directive.....	147
CSAE instruction.....	175, 253, 258	ENDIF directive.....	126
CSB instruction.....	176, 253, 258	ENDM directive.....	135
CSBE instruction.....	176, 253, 258	ENDR directive.....	127
CSE instruction.....	176, 253, 258	EQU directive.....	53
CSNE instruction.....	176, 253, 258	ERROR directive.....	147
D		Event counter mode.....	238
D/A Converter.....	272	EXITM directive.....	137
Data direction register.....	212	Expressions.....	140
Data memory, addressing the.....	41	F	
Data memory, clearing the.....	45	FIFO.....	382
Data memory, organization of.....	203	frequ directive.....	146
Data types.....	141	FREQU directive.....	29
DC flag.....	87, 229	FSR stack.....	60
Debouncing contacts.....	99	Function generator.....	271
Debugger.....	15	Function generator, ramp.....	273
Debugger, run mode.....	21	Function generator, sine wave.....	282
Debugging, single step.....	16	Function generator, triangle wave.....	277
DEC instruction.....	73, 254, 256	FUSE register.....	230, 231, 234
Decimal display.....	323	FUSEX register.....	232, 236
Decrement.....	73	H	
DECSZ instruction.....	74, 254, 258	Hexadecimal display.....	320
DECZ instruction.....	256	I	
Delay loops.....	23	I/O registers.....	210
device directive.....	144	I ² C bus.....	388
DEVICE directive.....	28	I ² C master.....	389
DEVICE register.....	234, 237	I ² C protocol.....	389
DEVICE WATCHDOG directive.....	121	I ² C routines.....	388
Digital clock.....	338	I ² C slave.....	389
Digital clock, adjusting the.....	355	I ² C, Acknowledge.....	390
Direct addressing.....	204, 206	I ² C, Bus arbitration.....	390
Directives, configuration.....	28	I ² C, Clock stretching.....	390
Display, 7-Segment.....	314	I ² C, Data format.....	391
Display, BCD.....	323	I ² C, Example.....	392
Display, decimal.....	323	I ² C, Example routines.....	395
Display, hexadecimal.....	320	I ² C, Idle state.....	390
Division.....	81		
DJNZ instruction.....	174, 254, 256, 258		
DS directive.....	29		

I ² C, Master VP.....	408	Keyboard, Quick Scan	371
I ² C, Pull up resistors	394	Keyboards, reading	364
I ² C, Slave VP	409	Keyboards, scanning.....	364
I ² C, Start condition	389		
I ² C, Stop condition	390	L	
I ² C, Transmission repeat.....	391	Labels.....	20
ID directive.....	146	Labels, local.....	36
IF directive.....	127	Level register	213
IFDEF directive	125	Light-following SX Tech Bot.....	449
IFNDEF directive	126	Local labels.....	36
IJNZ instruction.....	174, 254, 256, 258	Logical operations	76
INC instruction	73, 254, 256		
INCLUDE directive.....	24, 52	M	
Include file.....	24, 28, 52	m register	220
Increment	73	M register.....	218
INCSZ instruction.....	74, 254, 256, 258	Macro arguments	135
Indirect addressing.....	204, 206	MACRO directive.....	135
Infrared sensors	440	Macros	135
Instructions, arithmetic	67	MODE instruction	218, 254, 259
Instructions, compound	17	MODE register.....	218, 220, 261
Instructions, construction of	202	MOV instruction.....	90, 254, 257
Instructions, read-modify-write	210	MOV instructions with arithmetic	87
Interrupt modes, configuring	224	MOVB instruction	254, 257, 258
Interrupts.....	102, 222	MOVSZ instruction	177, 254, 258, 259
Interrupts by analog comparator.....	150	Multi-Byte addition	70
Interrupts by counter overflow	224	Multi-Byte counters	74
Interrupts, timed.....	223	Multi-Function timer	217
Interrupty by signal edges.....	222	Multi-Funtion timers.....	237
IREAD instruction.....	169, 254, 259	Multiplication	81
J		N	
JB instruction.....	254, 258	NOP instruction	177, 255, 259
JC instruction.....	254, 258	NOT instruction.....	80, 255, 256
JMP instruction.....	254, 259	Numbers, signed	72
JNB instruction.....	174, 254, 259		
JNC instruction.....	175, 254, 259	O	
JNZ instruction	175, 254, 259	Obstacles, detecting with the Boe-Bot.....	439
JZ instruction	254, 259	OPTION register.....	226
		OR instruction.....	78, 255, 256
K		ORG directive.....	29
Keyboard matrix with multiplexer.....	378		
Keyboard, 2-Key rollover.....	372		

Programming the SX Microcontroller

P	
PAGE instruction	158, 255, 259
PD flag	228
Port outputs, testing	361
Port signals, recognizing	94
Port, block diagram	211
Ports	210
Potentiometer settings, reading	293
Prescaler	110, 123
Program memory, organization of	207
Program memory, organizing the	157, 163
Prototyping system, “homebrew”	6
Prototyping systems	5
Pull-Up enable register	213
Pulse Width Modulation	288
PWM	288, 428
PWM mode	238
PWM, constant frequency	290
R	
Read-modify-write instructions	210
Recognizing port signals	94
REPT directive	127
RESET directive	29
Reset reasons	124, 227
RET instruction	33, 255, 259
RETI instruction	103, 255, 259
RETIW instruction	103, 109, 255, 259
RETP instruction	255, 259
Return stack	34, 229
RETW instruction	164, 255, 259
RL instruction	80, 255, 256
Robotics	425
Rotate instruction	80
RR instruction	80, 255, 256
RTCC register	107
S	
SB instruction	255, 257, 259
SC instruction	69, 255, 259
Schmitt trigger enable register	214
Schmitt Trigger, Hysteresis	380
Schmitt Trigger, simulated	380
SCL	388
SDA	388
Semi-direct addressing	206
Sensors, infrared	440
Servo	425
Servo control program	429
Servos, calibrating	432
Servos, controlling	428
SET directive	53
SETB instruction	255, 257
Setup28.inc	24
Signed numbers	72
Sine wave	282
Sine wave, 1 st quadrant	284
Sine waves, superimposed	283
SKIP instruction	177, 255, 259
Skip instructions	69
SLEEP instruction	255, 259
Sleep mode	129
SNB instruction	255, 257, 259
SNC instruction	69, 255, 259
SNZ instruction	69, 255, 259
Software timer mode	238
Soubroutine calls across pages	161
Stack	34
Stack for FSR	60
Stack memory	229
State Engine	327, 338
State Machine	327, 338
STATUS register	68, 208, 228
STC instruction	68, 255, 256
Stopwatch	326
STZ instruction	68, 255, 256
SUB instruction	71, 255, 257
SUBB instruction	255, 257
Subroutines	33
Subtraction	71
SWAP instruction	87, 255, 257
SX features	197
SX Tech board	425
SX Tech Bot	425
SX Tech Bot with whiskers	449

SX Tech Bot, light-following	449	TRIS registers	264, 265
SX Tech Bot, maze-following	449	<i>U</i>	
SX-Key	8	UART	179
SX-Key IDE	8	<i>V</i>	
Symbolic addresses.....	20	Variable names, symbolic.....	52
Symbolic constants	141	Virtual peripherals	179
Symbolic variable names.....	52	Voltage converter.....	357
System clock generation.....	152	Voltage converter, stabilized	358
SZ instruction	69, 255, 259	<i>W</i>	
<i>T</i>		W register	209
T1CNTA register.....	239	Wakeup.....	129
T1CNTB register	240	Wakeup by signal edges	129
T2CNTA register	241	Wakeup by watchdog timer.....	133
T2CNTB register	242	Wake-Up configuration register	214
TEST instruction	69, 255, 259	Watchdog timer	121, 225
Test of port outputs.....	361	<i>X</i>	
Time delays	23	XOR instruction.....	79, 255, 256
Timer control registers.....	239	<i>Z</i>	
Timer, 'Hardware'	411	Z flag	229
Timer, multi-function	217		
Timer, software mode.....	238		
Timer, Virtual Peripheral.....	307		
Timers, multi-function.....	237		
TO flag	228		