User's Manual

SX Microcontrollers

SCENIX

Scenix™ SX18AC/SX20AC/SX28AC User's Manual

Revision History

REVISION	RELEASE DATE	SUMMARY OF CHANGES
1.0	October 16, 1998	First Release

Scenix Semiconductor, Inc., 3160 De la Cruz Boulevard, Suite 200, Santa Clara, CA 95054 USA
Telephone: +1 408 327 8888, Web site: hhtp://www.scenix.com

Contents

Chapter 3 Instruction Set

List of Figures

List of Tables

Chapter 1

Overview

1.1 Introduction

The Scenix SX18AC, SX20AC, and SX28AC are members of the SX family of high-performance 8-bit microcontrollers fabricated with an advanced CMOS process technology. The advanced process, combined with a RISC-based architecture, allows high-speed computation, flexible I/O control, and efficient data manipulation. Throughput is enhanced by operating the device at frequencies up to 50 MHz and by optimizing the instruction set to include mostly single-cycle instructions.

On-chip functions include a general-purpose 8-bit timer with prescaler, an analog comparator, a brown-out detector, a watchdog timer, a power-save mode with multi-source wakeup capability, an internal R/C oscillator, user-selectable clock modes, and high-current outputs.

1.2 Key Features

These are the key features of the SX18AC, SX20AC, and SX28AC devices:

- 50 MIPS performance at 50 MHz oscillator frequency
- 2048 x 12 bits EE/Flash program memory rated for 10,000 rewrite cycles
- 136 x 8 bits SRAM
- In-system programming capability through OSC pins
- User-selectable clock modes using an internal oscillator, external clock signal, or external oscillator (crystal or RC)
- Analog comparator
- Brown-out detector (4.2V, on/off)
- Multi-InputWakeup (MIWU) on eight pins
- Fast lookup capability through run-time readable code
- Complete development tool support available from Parallax, Inc.

1.3 CPU Features

These are the key features of the device CPU:

- Fully static design – DC to 50 MHz operation
- 20 ns instruction cycle time at 50 MHz
- Mostly single-cycle instructions
- Selectable 8-level deep hardware subroutine stack
- Single-level interrupt processing
- Fixed interrupt response time: 60 ns internal, 100 ns external at 50 MHz
- Hardware context save/restore for interrupt processing
- Designed to be pin-compatible and upwardly code-compatible with the PIC16C5x® microcontroller

1.4 I/O Features

These are the key features of the device I/O ports:

- Software-selectable I/O configuration
- Each pin programmable as an input or output
- TTL or CMOS level selection on inputs
- Internal weak pull-up selection on inputs (~20 kΩ to V_{DD})
- Schmitt trigger inputs on Port B and Port C
- All outputs capable of sinking/sourcing 30 mA
- Symmetrical drive on Port A outputs (same V_{drop} +/-)

1.5 Architecture

The SX device uses a modified Harvard architecture. This architecture is based on having two separate memories with separate address buses, one for the program and one for data, while allowing transfer of data from program memory to SRAM. This ability allows accessing data tables from program memory. The advantage of this architecture is that instruction fetch and memory transfers can be overlapped in a multi-stage pipeline, which means the next instruction can be fetched from program memory while the current instruction is being executed.

The SX family implements a four-stage pipeline (fetch, decode, execute, and write back), which results in a throughput of one instruction per clock cycle. At the maximum operating frequency of 50 MHz, instructions are executed at the rate of one per 20-ns clock cycle.

1.6 Programming and Debugging Support

The SX devices are currently supported by the SX-Key™ development package offered by Parallax, Inc. This package provides an integrated development environment including editor, macro assembler, debugger, and device programmer. The available package components include the SX-Key programmer/emulator unit, a demonstration board, a prototyping board, a serial interface cable, development software that runs under Windows 95, and complete documentation.

1.7 Applications

Emerging applications and advances in existing ones require higher performance while maintaining low cost and fast time-to-market.

The SX devices provide solutions for many familiar applications such as process controllers, electronic appliances/tools, security/monitoring systems, and personal communication devices. In addition, the enhanced throughput of the device allows efficient development of software modules called "Virtual Peripherals" to replace on-chip hardware peripherals. The concept of Virtual Peripherals provides benefits such as using a more simple device, reduced component count, fast time to market, increased flexibility in design, and ultimately overall system cost reduction.

Here are some examples of Virtual Peripheral applications:

- Serial, Parallel, I2C™, Microwire™ (µ-Wire), Dallas µ-Wire, SPI, DMX-512, X-10, IR transceivers
- Frequency generation and measurement
- Spectrum analysis
- Multi-tasking, interrupts, and networking
- Resonance loops
- DRAM drivers
- Music and voice synthesis
- PPM/PWM output
- Delta/Sigma ADC
- DTMF generation/detection
- PSK/FSK generation/detection

- Quadrature encoder/decoder
- Peripheral Interface Device (PID) and servo control
- Video controller

1.8 Part Numbers and Pinout Diagrams

This user's guide describes the Scenix SX18AC, SX20AC, and SX28AC microcontrollers, which are available in the four pin configurations shown in Figure 1-1. All of these devices are functionally the same, except that the 18-pin and 20-pin devices do not have the port pins RC0 through RC7. Therefore, Port C cannot be used in the smaller devices.

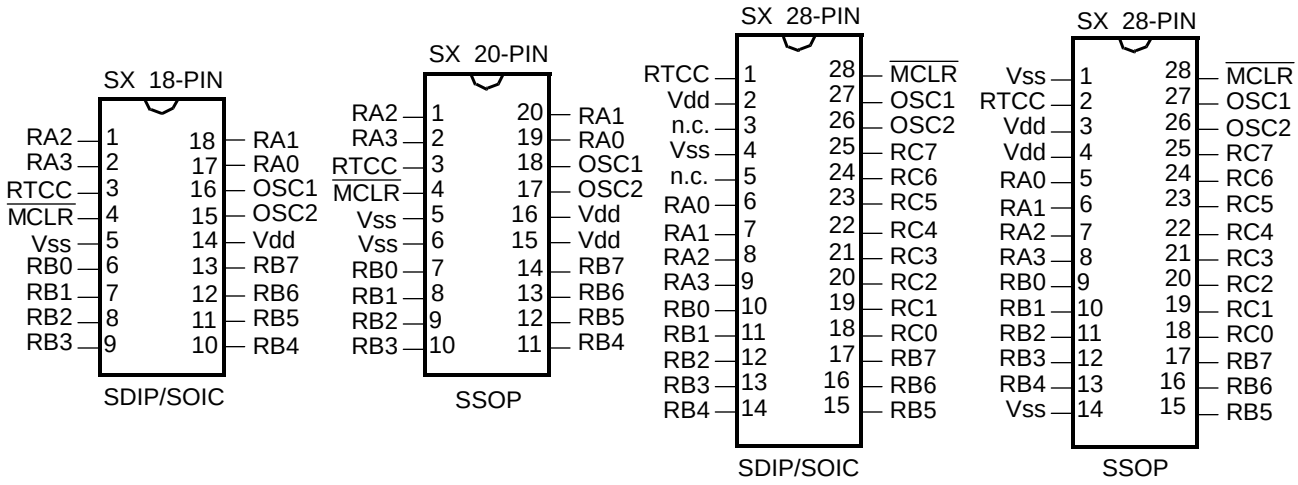

Figure 1 -1 Device Pin Assignments

Table 1-1 is a list of the available SX device packages and the corresponding number of pins, number of I/O pins, program (flash) memory size, and general-purpose RAM size. Use this table as a guide for ordering the parts that fit your requirements.

Table 1-1 Device Package Names

Device	Pins	I/O	EE/Flash (Words)	RAM (Bytes)
SX18AC/SO	18	12	2K	136
SX18AC/DP	18	12	2K	136
SX20AC/SS	20	12	2K	136
SX28AC/SO	28	20	2K	136
SX28AC/DP	28	20	2K	136
SX28AC/SS	28	20	2K	136

Figure 1-2 is a diagram showing the general naming conventions for SX family devices. The part number consists of several fields that specify the manufacturer, pin count, feature set, memory size, supply voltage, operating temperature range, and package type, as indicated in Figure 1-2.

Throughout this manual, the term “SX” refers to all the devices listed in Table 1-1, except where indicated otherwise.

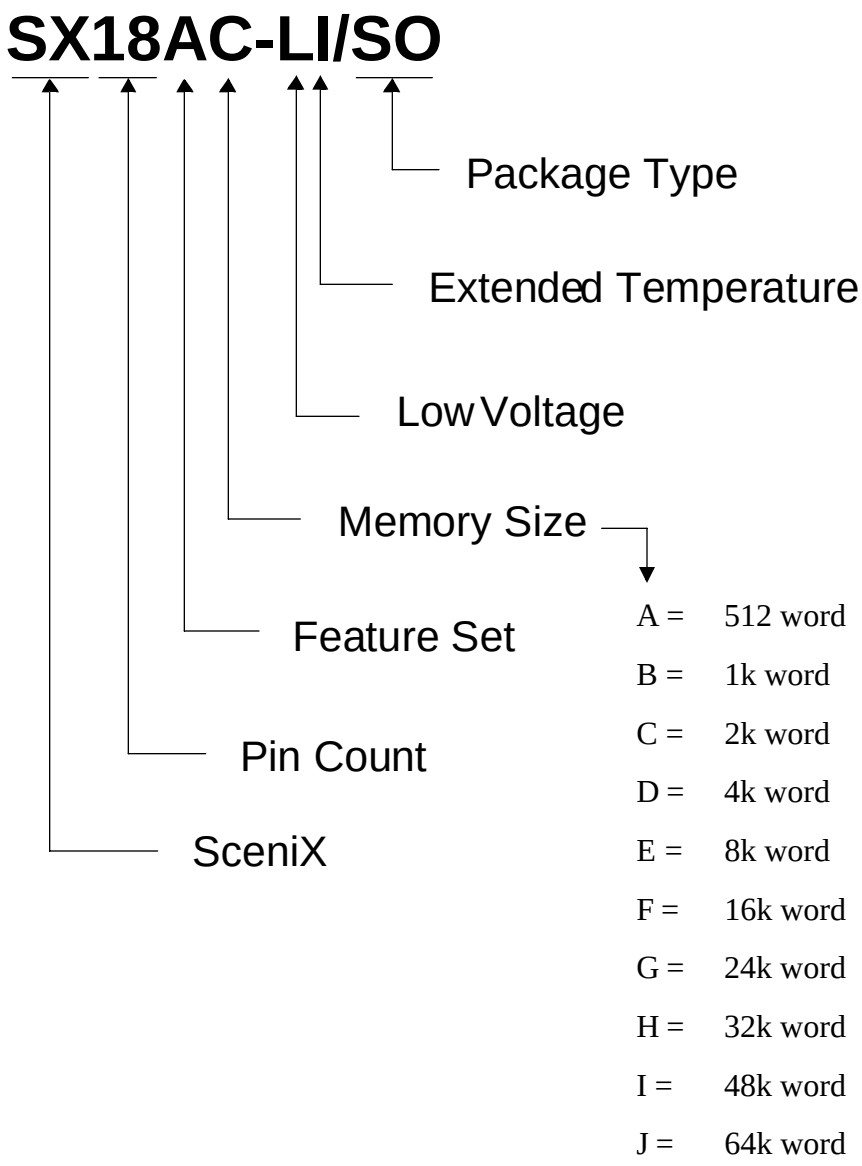

Figure 1-2 Part Numbering Reference Guide

1.9 Pin Descriptions

Table 1-2 describes the SX28AC device pins. For each pin, the table shows the pin type (input, output, or power), the input voltage levels (TTL, CMOS, or Schmitt trigger), and the pin function. The following abbreviations are used in the table:

- I = device input
- O = device output
- I/O = bidirectional I/O pin
- P = power supply pin
- NA = not applicable
- TTL = TTL input levels
- CMOS = CMOS input levels
- ST = Schmitt trigger input
- MIWU = Multi-Input Wakeup

Table 1-2 Pin Descriptions

Name	Pin Type	Input Levels	Description
RA0	I/O	TTL/CMOS	Bi-directional I/O Pin; Symmetrical Source / Sink Capability
RA1	I/O	TTL/CMOS	Bi-directional I/O Pin; Symmetrical Source / Sink Capability
RA2	I/O	TTL/CMOS	Bi-directional I/O Pin; Symmetrical Source / Sink Capability
RA3	I/O	TTL/CMOS	Bi-directional I/O Pin; Symmetrical Source / Sink Capability
RB0	I/O	TTL/CMOS/ST	Bi-directional I/O Pin; Comparator Output; MIWU Input
RB1	I/O	TTL/CMOS/ST	Bi-directional I/O Pin; Comparator Negative Input; MIWU Input
RB2	I/O	TTL/CMOS/ST	Bi-directional I/O Pin; Comparator Positive Input; MIWU Input
RB3	I/O	TTL/CMOS/ST	Bi-directional I/O Pin; MIWU Input
RB4	I/O	TTL/CMOS/ST	Bi-directional I/O Pin; MIWU Input
RB5	I/O	TTL/CMOS/ST	Bi-directional I/O Pin; MIWU Input
RB6	I/O	TTL/CMOS/ST	Bi-directional I/O Pin; MIWU Input
RB7	I/O	TTL/CMOS/ST	Bi-directional I/O Pin; MIWU Input
RC0	I/O	TTL/CMOS/ST	Bi-directional I/O Pin
RC1	I/O	TTL/CMOS/ST	Bi-directional I/O Pin
RC2	I/O	TTL/CMOS/ST	Bi-directional I/O Pin
RC3	I/O	TTL/CMOS/ST	Bi-directional I/O Pin
RC4	I/O	TTL/CMOS/ST	Bi-directional I/O Pin
RC5	I/O	TTL/CMOS/ST	Bi-directional I/O Pin
RC6	I/O	TTL/CMOS/ST	Bi-directional I/O Pin
RC7	I/O	TTL/CMOS/ST	Bi-directional I/O Pin
RTCC	I	ST	Input to Real Time Clock/Counter
MCLR	I	ST	Master Clear Reset Input – Active Low
OSC1/In/Vpp	I	ST	Crystal Oscillator Input - External Clock Source Input
OSC2/Out	O	CMOS	Crystal Oscillator Output – in R/C mode, internally pulled to Vdd through weak pullup
Vdd	P	NA	Positive Supply Pin
Vss	P	NA	Ground Pin

Chapter 2

Architecture

2.1 Introduction

The SX28AC is a complete 8-bit RISC microcontroller with an electrically erasable (flash) program memory and in-system programming capability. The device can operate with a clock rate of up to 50 MHz and can execute instructions at a rate of up to 50 million instructions per second.

The SX28AC device has three multi-pin I/O ports, an internal oscillator, a Watchdog timer, a Real-Time Clock/Counter, an analog comparator, power-on and brownout reset control, and Multi-Input Wakeup capability. Figure 2-1 is a block diagram of the device. The SX18AC and SX20AC have the same features, except that they have two rather than three I/O ports.

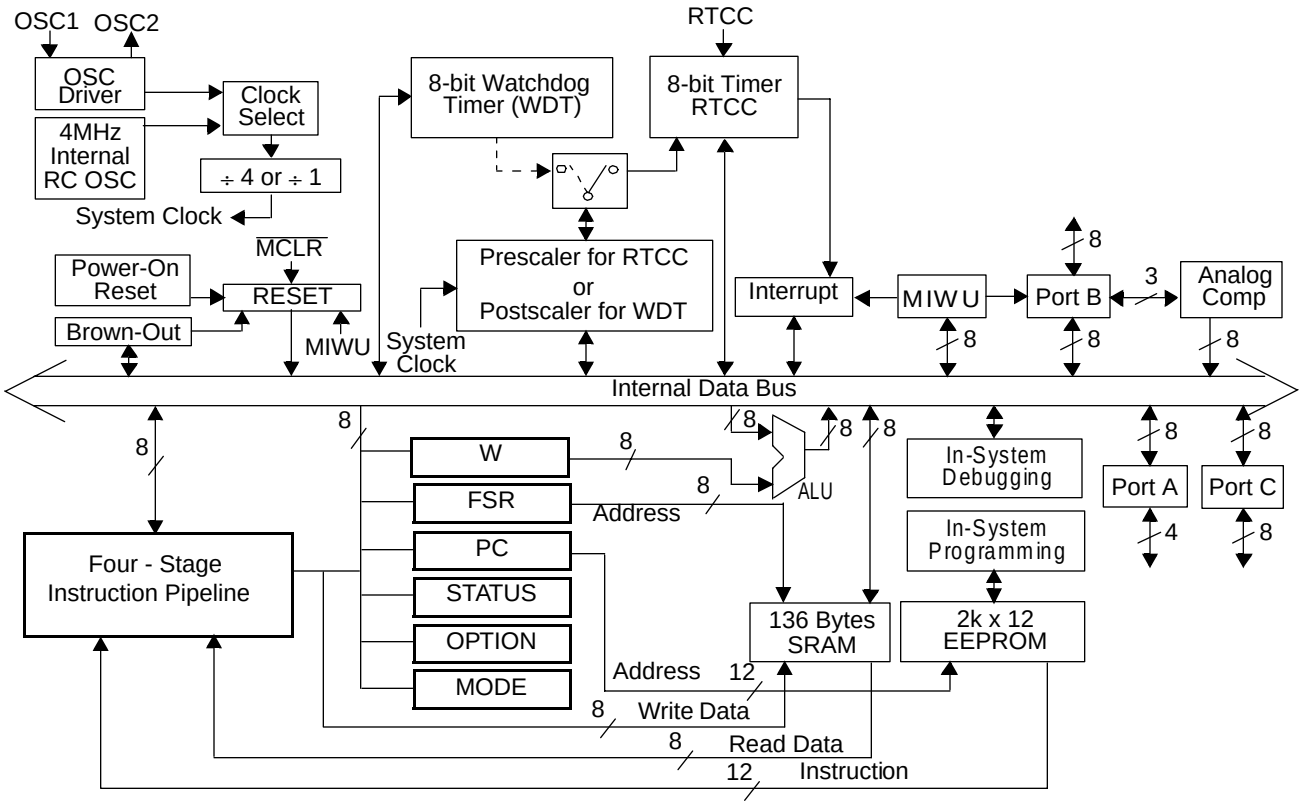

Figure 2 -1 SX28AC Block Diagram

The SX device uses a modified Harvard architecture, in which the program and data are stored in separate memory spaces. The advantage of this architecture is that instruction fetches and data transfers can be overlapped with a multi-stage pipeline, which means the next instruction can be

fetched from program memory while the current instruction is being executed uses data from the data memory. This device has a "modified" Harvard architecture because instructions are available for transferring data from the program memory to the data memory.

2.2 Program Memory

The program memory holds the application program for the device. It is an electrically erasable, flash-programmed memory containing 2,048 words of 12 bits per word. Each memory location holds a single 12-bit instruction opcode or 12 bits of fixed data that can be accessed by the program. The memory can be programmed and reprogrammed through the device oscillator pins, even with the device installed in the target system.

The program memory is addressed by the program counter, an 11-bit register. Operation of the program counter is described in detail in Section .

2.3 Data Memory

The data memory is a RAM-based register set consisting of 136 general-purpose registers and eight dedicated-purpose registers. All of these registers are eight bits wide. The registers are organized into eight banks, designated Bank 0 through Bank 7. This organization allows the SX instructions to address the registers using just five bits of the 12-bit instruction opcode.

Because the registers are organized into banks or "files," these memory-mapped registers are called "file registers." In the descriptions of the SX instructions inChapter 3, the abbreviation "fr" represents a 5-bit register address encoded into the instruction opcode.

2.3.1 Banks

The SX device can be programmed to use any one of the eight banks at any given time. The three high-order bits in the File Select Register (FSR) specify the current bank number. To change from one bank to another, the program can either write an eight-bit value to the FSR register or use the "bank" instruction. The "bank" instruction writes the three bank-selection bits in the FSR register without affecting the other bits in the register. Bank 0 is selected by default upon power-up or reset.

Within each bank, there are 32 available addresses, ranging from 00h to 1Fh. Table 2-1 shows the organization of file registers in the memory-mapped address space. The numbers along the left side the table (ranging from $00 to $1F) show the 32 possible register addresses that can be specified an instruction. The bank numbers listed across the top (ranging from 0 to 7) are the numbers that can be programmed into the three high-order bits of the FSR register. The entries inside the table shows the registers accessed by each combination of register address and bank selection.

The 5-bit register addresses along the left side are shown as they are written in the syntax of the SX assembly language, using a dollar sign ($) indicating the beginning of a hexadecimal value. Inside the table, the register addresses are shown as 8-bit hexadecimal values.

Table 2-1 RAM Register Map

	Bank 0	Bank 1	Bank 2	Bank 3	Bank 4	Bank 5	Bank 6	Bank 7
$00	INDF	INDF	INDF	INDF	INDF	INDF	INDF	INDF
$01	RTCC	RTCC	RTCC	RTCC	RTCC	RTCC	RTCC	RTCC
$02	PC	PC	PC	PC	PC	PC	PC	PC
$03	Status	Status	Status	Status	Status	Status	Status	Status
$04	FSR	FSR	FSR	FSR	FSR	FSR	FSR	FSR
$05	RA	RA	RA	RA	RA	RA	RA	RA
$06	RB	RB	RB	RB	RB	RB	RB	RB
$07	RC	RC	RC	RC	RC	RC	RC	RC
$08	08h	08h	08h	08h	08h	08h	08h	08h
$09	09h	09h	09h	09h	09h	09h	09h	09h
$0A	0Ah	0Ah	0Ah	0Ah	0Ah	0Ah	0Ah	0Ah
$0B	0Bh	0Bh	0Bh	0Bh	0Bh	0Bh	0Bh	0Bh
$0C	0Ch	0Ch	0Ch	0Ch	0Ch	0Ch	0Ch	0Ch
$0D	0Dh	0Dh	0Dh	0Dh	0Dh	0Dh	0Dh	0Dh
$0E	0Eh	0Eh	0Eh	0Eh	0Eh	0Eh	0Eh	0Eh
$0F	0Fh	0Fh	0Fh	0Fh	0Fh	0Fh	0Fh	0Fh
$10	10h	30h	50h	70h	90h	B0h	D0h	F0h
$11	11h	31h	51h	71h	91h	B1h	D1h	F1h
$12	12h	32h	52h	72h	92h	B2h	D2h	F2h
$13	13h	33h	53h	73h	93h	B3h	D3h	F3h
$14	14h	34h	54h	74h	94h	B4h	D4h	F4h
$15	15h	35h	55h	75h	95h	B5h	D5h	F5h
$16	16h	36h	56h	76h	96h	B6h	D6h	F6h
$17	17h	37h	57h	77h	97h	B7h	D7h	F7h
$18	18h	38h	58h	78h	98h	B8h	D8h	F8h
$19	19h	39h	59h	79h	99h	B9h	D9h	F9h
$1A	1Ah	3Ah	5Ah	7Ah	9Ah	BAh	DAh	FAh
$1B	1Bh	3Bh	5Bh	7Bh	9Bh	BBh	DBh	FBh
$1C	1Ch	3Ch	5Ch	7Ch	9Ch	BCh	DCh	FCh
$1D	1Dh	3Dh	5Dh	7Dh	9Dh	BDh	DDh	FDh
$1E	1Eh	3Eh	5Eh	7Eh	9Eh	BEh	DEh	FEh
$1F	1Fh	3Fh	5Fh	7Fh	9Fh	BFh	DFh	FFh

For the first 16 addresses that can be specified in an instruction (00h through 0Fh), the same 16 registers are accessed, irrespective of the bank setting. Therefore, these 16 “global” registers are always accessible. The first eight are dedicated-purpose registers (INDF, RTCC, PC, and so on), and the next eight are general-purpose registers. In Table 2-1, these registers are shown shaded in Bank 1 through Bank 7 to indicate that they are the same registers as in Bank 0.

For the upper 16 addresses that can be specified in an instruction (10h through 1Fh), a different set of registers is accessed in each bank. This allows as many as 128 different registers to be accessed in this memory range, although only 16 are accessible at any given time.

The total number of general-purpose registers is 24 in Bank 0 (from 08h to 1Fh) and 16 in each of the remaining seven banks (from 10h to 1Fh in each bank), for a total of 136 registers. In the SX18AC and SX20AC, an additional general-purpose register is available at address 08h because there is no Port C register occupying that address.

2.4 Special-Function Registers

The SX instructions can access the eight dedicated file registers from 00h to 07h and the 136 general-purpose file registers at higher addresses. Many instructions can also access certain non-memory-mapped registers: the Working register (W), the port control registers, the MODE register, and the OPTION register. All of these registers are eight bits wide.

Table 2-2 lists and briefly describes the dedicated file registers and non-memory-mapped registers that are accessible to SX instructions.

Table 2-2 Register Summar

Register Name	Description
W	Working Register. This is the main working register used by many instructions as the source or destination of the operation.
INDF (00h)	Indirect through FSR. There is no actual register at this memory location. When this address (00h) is specified as the source or destination of an operation, the register location pointed to by FSR is accessed.
RTCC (01h)	Real-Time Clock/Counter. This register can be used to keep track of elapsed time or occurrences of transitions on the RTCC input pin.
PC (02h)	Program Counter. Only the lower eight bits of the 11-bit program counter are available at this register location.
STATUS (03h)	Status. This register contains the status bits for the device such as the C flag, Z flag, and program memory page selection bits.
FSR (04h)	File Select Register. This register specifies the bank number for direct addressing or the full 8-bit address for indirect addressing.
RA (05h)	Port A Data Register. This register is used to control output signals and read input signals on the RA0-RA3 I/O pins.
RB (06h)	Port B Data Register. This register is used to control output signals and read input signals on the RB0-RB7 I/O pins.
RC (07h)	Port C Data Register. In the SX28AC, this register is used to control output signals and read input signals on the RC0-RC7 I/O pins. In the SX18AC and SX20AC, the register at 07h is a general-purpose register.
Port Control Registers	The port control registers are used to control the configuration of the port I/O pins (RA0-RA3, RB0-RB7, and RC0-RC7). These registers are accessed by a special-purpose instruction, “mov !rx,W”.
MODE	MODE Register. This register controls access to the port control registers when you use the “mov !rx,W” instruction.
OPTION	Option Register. This register sets some device configuration options such as the Real-Time Clock/Counter incrementing mode.

2.4.1 W (Working Register)

The W register is the main working register used by many instructions as the source or destination of the operation. It is often used as a temporary storage area for intermediate operations. For example, to add the contents of two file registers, you must first move the contents of one file register to W and then execute an "add" instruction to perform an addition between W and the other file register.

In the default device configuration, W is not memory-mapped and can only be accessed by instructions that work specifically with W as the source or destination. However, you can optionally make the W available as a memory-mapped register at address 01h. To do this, first program the $\overline{\text{OPTIONX}}$ bit to 0 in the FUSE word in the program memory. Then have your program clear the RTW bit in the OPTION register. If you do this, the RTCC register normally at address 01h becomes unavailable.

2.4.2 FSR (Indirect through FSR)

The INDF register location (address 00h) is used for indirect addressing. Whenever this address is specified as the source or destination of an operation, the device uses the register pointed to by the FSR register (address 04h). There is no actual register or data stored at address 00h.

For more information on indirect addressing, see Section 3.3.3.

2.4.3 RTCC (Real-Time Clock/Counter)

The RTCC register (address 01h) is an 8-bit Real-Time Clock/Counter used to keep track of elapsed time or to keep a count of transitions on the RTCC input pin. The timer operating configuration is determined by control bits in the OPTION register.

To keep track of time, you configure the timer register to be incremented once per instruction cycle or once per multiple of the instruction cycle. To count external events, you configure the timer register to be incremented once per rising edge or falling edge on the RTCC input pin.

The program can read or write the register at any time. A rollover from FFh to 00h generates an interrupt to the CPU if that condition is enabled as an interrupt.

For more information on the operation of the timer, see Section 6.2.

If you do not need to use the RTCC register, you can optionally make the working register (W) available as a memory-mapped register at address 01h. For details, see the description of the W register.

2.4.4 PC (Program Counter)

The PC register (address 02h) contains the lower eight bits of the 11-bit program counter. The 11-bit program counter is a pointer register that points to the current instruction being executed in the 2,048-word program memory. During regular program execution, the program counter is incremented automatically once per instruction cycle. This regular sequence is altered in order to perform skips, jumps, and subroutine calls in the application program.

For detailed information on program counter operation, see Section 2.6.

2.4.5 STATUS (Status Register)

The STATUS register (address 03h) contains the device status bits, which are automatically set or cleared by the device when certain events occur. The program can read this register at any time to determine the status of the device. The format of the register is shown below and Table 2-3 briefly describes each of the register bit fields.

Table 2-3 STATUS Register Bits

Status Bits	Description
PA2:PA0	Program memory page selection bits. You set or clear these bits to specify the program memory page number for a jump or call instruction.
TO	Watchdog timeout flag. This bit is set to 1 upon power-up and cleared to 0 when a Watchdog timeout occurs.
PD	Power Down flag. This bit is set to 1 upon power-up and cleared to 0 when the “sleep” instruction is executed.
Z	Zero flag. This bit is set when the result of an operation is zero.
DC	Digit Carry flag. This bit is set when there is a carry out from bit 3 to bit 4 in an addition operation and cleared when there is a borrow out from bit 3 to bit 4 in a subtraction operation.
C	Carry flag. This bit is set when there is a carry out of bit 7 in an addition operation and cleared when there is a borrow out of bit 7 a subtraction operation. It is also affected by the rotate-through-carry instructions.

The STATUS register is a read/write register except for the TO and PD bits, which are read-only bits. Those two bits cannot be changed by writing to the STATUS register address.

When you write to the STATUS register, it is recommended that you use the “setb” (set bit) and “clrb” (clear bit) instructions to control the individual flags rather than “mov” (move) instructions to move whole register values. This is because the CPU often modifies the STATUS register bits, possibly resulting in register values that are different from what you expect.

The individual bits of the STATUS register are described below.

PA2:PA0 (Program Memory Page Selection Bits)

PA2:PA0 are the program memory page selection bits. They are used to set the high-order bits of the 11-bit program counter for jump and call instructions. You can set them without affecting the other bits in the STATUS register by using the “page” instruction. For details, see Section 2.6.

T0 (Watchdog Timeout Flag)

T0 is the Watchdog Timeout flag. It is set to 1 upon power-up and cleared to 0 when a watchdog timeout occurs. It is set back to 1 upon execution of the "clrwdt" (clear Watchdog timer) instruction or "sleep" instruction. For details, see Section .

PD (Power Down Flag)

PD is the Power Down flag. It is set to 1 upon power-up and cleared to 0 upon execution of the "sleep" instruction. It is set back to 1 upon execution of the "clrwdt" (clear Watchdog timer) instruction. For details, see Section 4.3.

Z (Zero Flag)

Z is the Zero flag. This bit is affected by the execution of many types of instructions (add, subtract, increment, decrement, move, logic operations, and so on). When one of these instructions is executed, the Z flag is set to 1 if the result is zero or cleared to 0 if the result is nonzero.

DC (Digit Carry Flag)

DC is the digit carry flag. This bit is affected by the execution of instructions that add or subtract. For an instruction that performs addition, the C flag is set to 1 if a carry occurs out of bit 3 to bit 4, or is cleared to 0 otherwise. For instructions that perform subtraction, the C flag is cleared to 0 if a borrow occurs out of bit 3 to bit 4, or is set to 1 otherwise. This flag can be used to implement carry-bit functions with single hexadecimal digits.

C (Carry Flag)

C is the carry flag. This bit is affected by the execution of the addition, subtraction, and rotate-through-carry instructions. For an instruction that performs addition, the C flag is set to 1 if overflow occurs (a carry out of bit 7), or is cleared to 0 otherwise. For instructions that perform subtraction, the C flag is cleared to 0 if underflow occurs (a borrow out of bit 7), or is set to 1 otherwise.

The device can be configured either to use or not use the C flag as an implicit input to addition and subtraction operations. This option is controlled by the $\overline{CF}$ bit in the FUSEX Word (a word that is programmed at the same time as the program memory). An implicit addition of the C flag can be used to implement multiple-byte addition and subtraction algorithms.

In the default configuration, the carry flag is not used as an input to addition and subtraction operations. In that case, the carry flag can still be added or subtracted explicitly by using a separate "test carry bit and skip" instruction in conjunction with an "increment" or "decrement" instruction.

2.4.6 FSR (File Select Register)

The FSR register (address 04h) is the File Select Register used to specify the 3-bit bank number for direct addressing of file registers, or the full 8-bit address for indirect addressing of file registers. The file registers are addressed as follows:

For direct addressing, the three high-order bits of FSR specify the bank number, and the instruction opcode specifies the 5-bit address of the register within the selected bank. The five low-order bits of FSR are ignored in this addressing mode.

For indirect addressing, the FSR register specifies the full 8-bit address of the register being accessed. To invoke this mode, the instruction specifies address 00h (INDF) as the source or destination of the operation.

For more information on direct and indirect addressing, see Section 3.3.2 and Section 3.3.3, respectively.

2.4.7 RA, RB, and RC (Port Data Registers)

The RA, RB, and RC registers (addresses 05h, 06h, and 07h) are the I/O port data registers for Port A, Port B, and Port C. When a port is configured to operate as an output, writing to its port data register sets the output values of the port pins. Reading from one of these register locations reads the port pins directly (not necessarily returning the values contained in the port data register).

For detailed information on configuring and using the I/O ports, see Chapter 5.

2.4.8 Port Control Registers and MODE Register

The MODE register controls access to the port control registers for subsequent uses of the "MOV !rx,W" instruction. For example, there are three registers for controlling Port A: the RA Direction register, the PLP_A (pullup enable A) register, and the LVL_A (level selection A) register. One of these three registers is written by the "mov !RA,W" instruction, depending on the value contained in the MODE register: 0Fh, 0Eh, or 0Dh, respectively for the RA Direction, PLP_A, and LVL_A registers.

Upon reset, the MODE register is initialized to 0Fh. This makes the port direction registers accessible to the "MOV !rx,W" instructions. In order to write to the other port control registers, you first need to write the appropriate value into the MODE register, as indicated in Table 2-4. MODE register values not listed in the table are reserved for future expansion.

After you write a value to the MODE register, that setting remains in effect until you change it by writing to the MODE register again. For example, you can write the value 0Eh to the MODE register just once, and then write to each of the three pullup configuration registers using the three "mov !rx,W" instructions shown at the top of Table 2-4.

For detailed information on configuring and using the I/O ports, see Chapter 5.

2.4.9 OPTION (Device Option Register)

The OPTION register sets several device configuration options, mostly related to operation of the Real-Time Clock/Counter. The format of the register is shown below. Upon reset, all bits in this register are set to 1.

Table 2-4 MODE Register Settings

MODE Register Setting	Reg. Written by mov !RA,W	Reg. Written by mov !RB,W	Reg. Written by mov !RC,W
08h	none	CMP_B	none
09h	none	WKPND_B	none
0Ah	none	WKED_B	none
0Bh	none	WKEN_B	none
0Ch	none	ST_B	ST_C
0Dh	LVL_A	LVL_B	LVL_C
0Eh	PLP_A	PLP_B	PLP_C
0Fh	RA Direction	RB Direction	RC Direction

RTW Bit: RTCC or W at address 01h

Clear the RTW bit to 0 to make W available as a memory-mapped register at address 01h. Set the RTW bit to 1 for the default register configuration, with RTCC at address 01h. Before you can clear the RTW bit, the option must be enabled by programming the $\overline{\text{OPTIONX}}$ bit to 0 in the FUSE word in the program memory.

RTE_IE Bit: RTCC Rollover Interrupt Enable

Clear the RTE_IE bit to 0 to enable the interrupt that occurs upon rollover of the RTCC counter, or set this bit to 1 to disable the interrupt. Before you can clear the RTE_IE bit, the option must be enabled by programming the $\overline{\text{OPTIONX}}$ bit to 0 in the FUSE word in the program memory.

RTS Bit: RTCC Trigger Selection

Clear the RTS bit to 0 to have the RTCC counter incremented automatically with each instruction cycle (or a specified number of instruction cycles). This mode can be used to implement a real-time clock. Set the RTS bit to 1 to have the RTCC counter incremented once each time a transition is detected on the RTCC input pin (or a specified number of transitions). This mode can be used as an external event counter.

RTE_ES: RTCC Input Edge Select

When the RTCC counter is configured to count transitions received on the RTCC pin (when RTS=1), the RTCC bit specifies the type of signal edges detected on the RTCC pin. Clear RTE_ES to 0 to detect low-to-high transitions on the RTCC pin. Set RTE_ES to 1 to detect high-to-low transitions on the RTCC pin.

PSA Bit: Prescaler Assignment

Clear the PSA bit to 0 to have the internal prescaler operate with the Real-Time Clock/Counter. In that case, the RTCC counter is incremented once every n instruction cycles, with the number n determined by the PS2:PS0 bits; and the Watchdog timer operates at the default rate.

Set the PSA bit to 1 to have the internal prescaler operate with the Watchdog timer. In that case, a Watchdog reset is generated after n timeouts of the Watchdog timer register, with the number n determined by the PS2:PS0 bits; and the RTCC register is incremented once per instruction cycle or external event.

PS2:PS0 Field: Prescaler Divide-By Factor

Use this bit field in conjunction with the PSA bit to specify an operating rate for the RTCC timer or Watchdog timer that is lower than the default rate. Table 2-5 shows the clock divide-by factors determined by these bits. Note that for a given setting, the divide-by factor depends on whether you use the prescaler register with the RTCC timer (PSA=0) or with the Watchdog timer (PSA=1). For the RTCC timer, the timer is incremented once every 2, 4, 8, ... or 256 instruction cycles or external events. For the Watchdog timer, a Watchdog reset is triggered after 1, 2, 4, ... or 128 overflows of the Watchdog timer register.

Table 2-5 Prescaler Divide-By Factors

PS2:PS0	RTCC Timer Input Divide-By Factor (PSA=0)	Watchdog Timer Output Divide-By Factor (PSA=1)
000	2	1 (timeout = 0.018 sec)
001	4	2 (timeout = 0.037 sec)
010	8	4 (timeout = 0.073 sec)
011	16	8 (timeout = 0.15 sec)
100	32	16 (timeout = 0.29 sec)
101	64	32 (timeout = 0.59 sec)
110	128	64 (timeout = 1.17 sec)
111	256	128 (timeout = 2.34 sec)

For detailed information on the Real-Time Clock/Counter and Watchdog timer, see Chapter 6.

2.5 Instruction Execution Pipeline

The CPU executes in program in a 4-stage pipeline consisting of the following stages:

- Fetch the instruction from program memory.
- Decode the instruction opcode.
- Execute the operation.

Write the result to destination register.

Each execution stage requires one instruction cycle. Although it takes four cycles to complete the execution of each instruction, an overall throughput of one instruction per clock cycle is achieved by overlapping successive operations in the pipeline. For example, Table 2-6 shows the sequence of operations carried out as the CPU executes the first six instructions of a program.

Table 2-6 Pipeline Execution Sequence

Program Instruction	Clock Cycle 1	Clock Cycle 2	Clock Cycle 3	Clock Cycle 4	Clock Cycle 5	Clock Cycle 6	etc.
1st instruction	Fetch	Decode	Execute	Write			
2nd instruction		Fetch	Decode	Execute	Write		
3rd instruction			Fetch	Decode	Execute	Write	
4th instruction				Fetch	Decode	Execute	...
5th instruction					Fetch	Decode	...
6th instruction						Fetch	...

As long as the normal flow of the program is not interrupted, the device performs four pipeline operations in parallel, thus achieving an overall throughput of one instruction per clock cycle, or 50 MIPS with a 50 MHz clock in the “turbo” clocking mode.

2.5.1 Clocking Modes

The SX device can be configured to operate in either the “turbo” or “compatible” mode. In the “turbo” mode, instructions are executed at the rate of one per clock cycle, and one clock cycle is the same as one instruction cycle. In the “compatible” mode, instructions are executed at the rate of one per four clock cycles, and four device clock cycles are required for each instruction cycle. For more information on these clocking modes, see Section 4.2.1.

2.5.2 Pipeline Delays

Any instruction or interrupt condition that alters the normal program flow will take at least one additional instruction cycle. For example, when a test-and-skip instruction is executed and the tested condition is true, the next instruction in the program is skipped. The next instruction occupies space and takes up time in the pipeline whether or not it is skipped. As a result, a skipped instruction causes a delay of one instruction cycle when a skip occurs. The test-and-skip instruction is described as taking one cycle if the tested condition is false or two cycles if the tested condition is true.

The call, jump, and return-from-interrupt instructions reload the program counter and cause the program to jump to an entirely new location in program memory. As a result, the instructions in the pipeline are discarded, causing a multi-cycle delay in program execution. Each call, jump, and return-from-interrupt instruction takes two, three, or four cycles for execution, depending on the specific instruction and the device clocking mode. For details, see the instruction descriptions in Chapter 3.

For the same reason, the triggering of an interrupt causes a pipeline delay. For an RTCC interrupt, the delay is three cycles. For a Multi-Input Wakeup interrupt, the delay is five cycles (two cycles for interrupt synchronization and a three-cycles pipeline delay).

2.5.3 Read-Modify-Write Considerations

A “read-modify-write” instruction is an instruction that operates by reading a register, modifying the value, and writing the result back to the register. Any instruction that writes a new value to a register that depends on the existing value is a read-modify-write instruction. Some examples are “clrb fr.bit” (clear bit), “setb fr.bit” (set bit), “add fr,w” (add W to file register), and “dec fr” (decrement file register). The “set bit” instruction, for example, does not simply set one bit and ignore the others. Instead, it reads the whole register, clears the specified bit, and writes the whole result back to the register.

When you use successive read-modify-write instructions on a port data register, you might get unexpected results at very high clock rates (such as 50 MHz). When you write to an I/O port, you write to the port data register; but when you read a port, you read the actual voltage on the I/O port pin. There is a slight delay from the time that the data port is written and the time that the output voltage changes to the programmed level.

When you use two successive read-modify-write instruction on the same I/O port, the “write” part of one instruction might not occur soon enough before the “read” part of the very next instruction, resulting in getting “old” data for the second instruction. (Remember that successive instructions are executed in parallel, one behind the next in the pipeline.)

To ensure predictable results, avoid using two successive read-modify-write instructions that access the same port data register. For example, you can insert a “nop” instruction between two such instructions in the program.

2.6 Program Counter

The program counter is an 11-bit register that points to the current instruction being executed in the 2,048-word program memory. The eight low-order bits of the program counter are directly accessible as a file register called the PC register, at address 02h. The three high-order bits are not directly accessible.

During regular program execution, the 11-bit program counter is incremented automatically once per instruction cycle. This regular sequence is altered in order to perform skips, jumps, subroutine calls, and interrupt processing.

Upon power-up or reset, the program counter is loaded with 7FFh, the top program address. This memory location typically contains an instruction to jump to an initialization routine.

All interrupts cause the program counter to be loaded with 000h, the bottom program address. Therefore, if interrupts are used, the bottom memory segment must contain the interrupt service routine.

2.6.1 Test and Skip

There are several instructions that test a condition and cause the next instruction to be skipped if the condition is true. For example, the “SB fr.bit” instruction tests a bit in a file register and skips the next instruction if that bit is set to 1.

When a skip occurs, the program counter is incremented by two rather than one upon conclusion of the test-and-skip instruction, and the skipped instruction (which is already being processed in the pipeline) is canceled. There is a delay of one clock cycle caused by the skip operation.

2.6.2 Jump Absolute

The “JMP addr9” instruction causes the program to jump to a new location by loading a new value into the 11-bit program counter. The lower nine bits of the new value come from a 9-bit field in the instruction opcode. The upper two bits of the new value come from the PA1 and PA0 bits of the STATUS register. Therefore, the PA1 and PA0 bits of the STATUS register must be pre-loaded with the desired 512-word page number before the jump instruction is executed.

For example, if the jump destination is address 7E0h in the program memory, the PA1 and PA0 bits in the STATUS register must be set to 1 before you execute the “JMP addr9” instruction. If those two bits are not already set to 1, you can use the following sequence of instructions to perform the jump:

```
setb $03.5   ;set bit 5 in STATUS register (PA0)
setb $03.6   ;set bit 6 in STATUS register (PA1)
clrb $03.7   ;clear bit 7 in STATUS register (PA2)
jmp  $1E0    ;jump to program memory address 7E0h
```

In this example, the jump address is 7E0h, an 11-bit address. The lower nine bits of this address are specified by the “JMP addr9” instruction as 1E0h, and the upper two bits are obtained from the PA1 and PA0 bits (bits 6 and 5) in the STATUS register, both of which are set to 1 prior to the “jmp” instruction.

In this example, there is also an instruction to clear the PA2 bit (bit 7) in the STATUS register. Although this is not necessary for proper operation in the SX device, it ensures that the software will be compatible with any future devices that have a program memory larger than 2,048 locations. In such devices, the PA2 bit will specify the high-order bit of a 12-bit jump address.

Another way to achieve the same effect faster and with fewer instructions is to use the "page" instruction to set the PA2:PA0 bits in the STATUS register:

```
page $600    ;set page to 600h (PA2:PA0 = 011 binary)
jmp  $1E0    ;jump to program memory address 7E0h
```

The "page" instruction sets the values of the PA2:PA0 bits without affecting other bits in the STATUS register. It does this in just one clock cycle. You specify a 12-bit value in the instruction and the assembler encodes the three high-order bits of the value into the instruction (and ignores the lower-order bits). When you execute the instruction, it sets the PA2:PA0 bits in the STATUS register accordingly.

Note that is necessary to set the PA2:PA0 bits prior to the "jmp" instruction only if they do not already contain the desired page number. You can set them just once and then use any number of "jmp" instructions as long as you stay within the same 512-word page in the program memory.

A "JMP addr9" instruction takes two clock cycles in the "compatible" clocking mode or three clock cycles in the "turbo" clocking mode. (For information on clocking modes, see Section 4.2).

2.6.3 Jump Indirect and Jump Relative

Instead of using the "JMP addr9" to specify an absolute jump destination, you can cause a jump by modifying the PC register (file register address 02h), which holds the lower eight bits of the program counter.

For example, to perform an indirect jump, you can move a new value from W to PC, as in the following example:

```
mov  W,$0B   ;load W with 8-bit jump address from file reg.
mov  $02,W   ;load PC with new address (lower 8 bits only)
```

To perform an indirect relative jump (a jump of a certain number of memory locations forward or backward from the next instruction), you can add W to PC or subtract W from PC, as in the following example:

```
mov  W,#$04  ;load W with the immediate value 04h
add  $02,W   ;increase PC by 4 (jump forward 5 instructions)
```

You can use an indirect jump to implement a multiple-branch conditional jump (for example, to jump to one of four different routines based on a calculation result).

If you perform a jump by modifying the PC register, you can only jump to a location within the same 256-word segment in the program memory. This is because you can only modify the lower eight bits of the 11-bit program counter. To jump across a 256-word boundary, use the "PAGE addr12" and "JMP addr9" instructions.

A jump performed by modifying the PC register with a “mov” or “add” instruction takes four clock cycles in the “compatible” clocking mode or three clock cycles in the “turbo” clocking mode.

2.6.4 Call

The “CALL addr8” instruction calls a subroutine. It works just like a “JMP addr9” instruction, with the following differences:

- The “call” instruction saves the 11-bit program counter value, incremented by one, on the program stack. This allows the program to later return from the subroutine and continue execution with the instruction immediately following the call.
- The “call” instruction only specifies the lower eight bits (rather than the lower nine bits) of the jump address. The ninth bit (bit 8) of the jump address is always 0. Therefore, the subroutine must start in the bottom half of a 512-word page in the program memory (000h to 0FFh, 200h to 2FFh, etc.).

Figures 2-2 and 2-3 show how the program counter is loaded for a “jmp” instruction and for “call” instruction, respectively. In either case, the PA1 and PA0 bits must contain the desired 512-word page of the program memory before the “jmp” or “call” instruction is executed. These bits can be easily changed with the “page” instruction. Note that the PA2 bit in the status register is not used, but is reserved for future devices that have a 12-bit program counter .

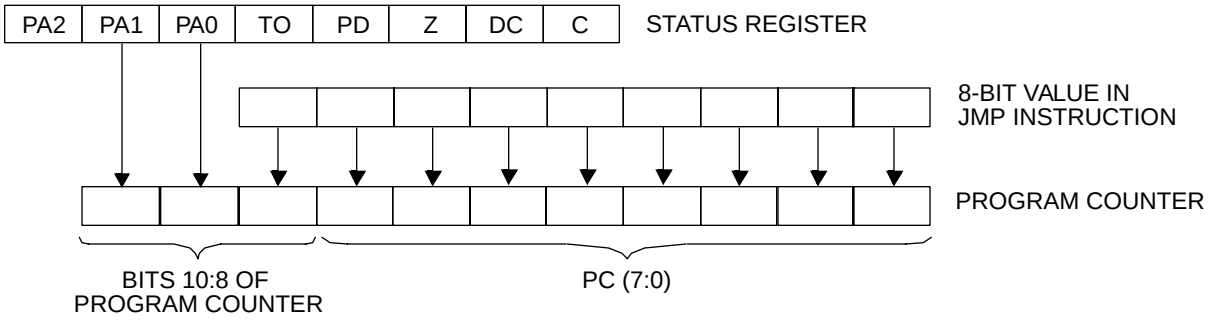

Figure 2 -2 Program Counter Loading for Jump Instruction

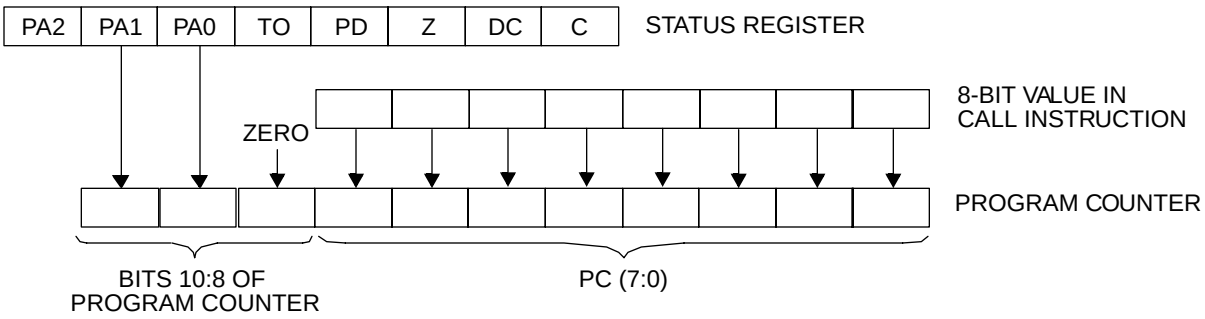

Figure 2-3 Program Counter Loading for Call Instruction

When a “call” instruction is executed, the CPU does the following:

- Increments the stack pointer and stores the 11-bit value on the program stack.
- Loads the lower eight bits of the program counter (the PC register) with the 8-bit value specified in the instruction opcode.
- Clears the ninth bit (bit 8) of the program counter to 0.
- Copies the PA1 and PA0 bits into the two high-order bits of the 11-bit stack pointer (bit 10 and bit 9).

Like the “jmp” instruction, the “call” instruction takes two clock cycles in the “compatible” mode or three clock cycles in the “turbo” clocking mode.

2.6.5 Return

A subroutine called by the “call” instructions is terminated by a “return” instruction. The “return” instruction restores the 11-bit value stored on the stack to the program counter. This causes the program to jump back to the instruction immediately following the “call” instruction that called the subroutine.

It is not necessary to set the PA2:PA0 bits in the STATUS register in order to return to the correct place in the program. This is because the full 11-bit program address is saved on the stack in a “call” instruction and fully restored by a “return” instruction. Therefore, the program always returns to the instruction immediately following the “call” instruction, even for a subroutine call across page boundaries. The PA2:PA0 bits are ignored by “return” instructions.

There are several different “return” type instructions available in the instruction set. Some are for returning from subroutines and other are for returning from interrupts. All of them are listed and described in Table 2-7. For more information on interrupts, see Chapter 6.

Table 2-7 Return-from-Subroutine/Interrupt Instructions

Option Bits	Description
RET	Return from Subroutine. This is an ordinary return from subroutine. It does not affect any registers or flags.
RETP	Return from Subroutine Across Page Boundary. This instruction works like the RET instruction, but also writes bit 10 and bit 9 of the return address (the address of the instruction immediately following the CALL instruction) to the PA1:PA0 bits of the STATUS register. This automatically configures the PA1:PA0 bits to select the current page, allowing a subsequent same-page jump or call to be executed without another "page" instruction.
RETW #lit	Return from Subroutine with Literal in W. This instruction works like the RET instruction, except that it loads a literal value into W before returning from the subroutine. A sequence of these instructions can be used in conjunction with a PC-adjustment instruction to implement a data-lookup table.
RETI	Return from Interrupt. This instruction restores the program counter and the W, STATUS, and FSR registers that were saved upon occurrence of the interrupt. (Note that the program stack is not used for interrupt processing.)
RETIW	Return from Interrupt and Adjust RTCC with W. This instruction works like the RETI instruction, but also subtracts W from the RTCC register. This can be used to adjust the RTCC counter back to the value in contained upon occurrence of the interrupt.

2.7 Stack

When a "call" instruction is executed, the 11-bit address of the instruction immediately following the "call" instructions is pushed onto the program stack. Upon return from the subroutine, the 11-bit address is popped from the stack and restored to the program counter, causing execution to resume with the instruction immediately following the "call" instruction.

The stack is a last-in, first-out (LIFO) data buffer, 11 bits wide and eight levels deep. The eight levels of the stack allow subroutines be nested, one within another, up to eight levels deep.

In the default device configuration, the stack is limited to two levels. In general, however, the stack should be configured to eight levels because there is no reason to limit the stack size. This option is controlled by the $\overline{\text{STACKX}}$ bit in the FUSE word register (a register programmed at the same time as the program memory).

The stack is not memory-mapped and there are no "push" or "pop" instructions in the instruction set. Therefore, the program stack is not directly accessible to the program and is not used for any purpose other than to save and restore program memory addresses, which is done implicitly by "call" and "return" instructions.

There is no “stack pointer” for this stack. Instead, the device simply moves all the data words down or up the stack for each “call” or “return” instruction executed, as indicated in Figures 2-4 and 2-5.

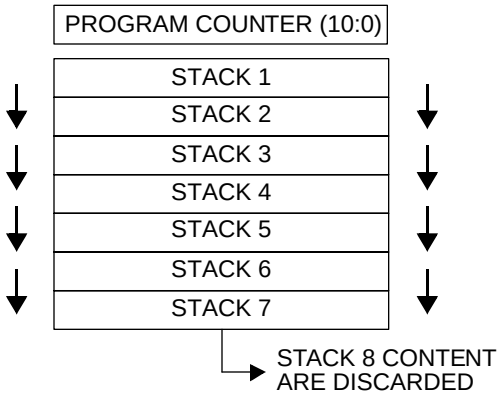

Figure 2-4 Stack Operation for a “Call” Instruction

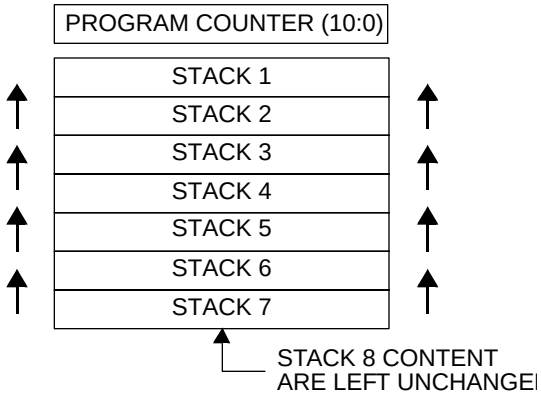

Figure 2 -5 Stack Operation for a “Return” Instruction

For a “call” instruction, the device copies the contents of the 11-bit program counter to the top stack location, and existing words in the stack are moved down by one stack location. Any data word in the bottom stack location is lost.

For any type of return-from-subroutine instruction (RET, RETP, or RETW lit), the device copies the contents of the top-level stack location into the program counter, and existing words in the stack are moved up by one stack location. The bottom stack location is left unchanged.

If you attempt to nest subroutines beyond eight levels, or if you execute a return-from-subroutine instruction without a prior corresponding “call” instruction, unpredictable results will occur because an incorrect address will be copied to the program counter.

The stack is not used for interrupt processing and is therefore not involved in the return-from-interrupt instructions (RETI and RETIW). For information on interrupt processing, see Chapter 6.

2.8 Device Configuration Options

The SX device has three 12-bit configuration registers that can be read or written at the same time that the instruction memory is programmed:

- FUSE word register, memory-mapped in the program memory at FFFh
- FUSEX word register, accessible by a device programming command
- DEVICE word register, a read-only word accessible by a device programming command

These registers are not accessible to the application program at run time. They can only be read or written when the device is set up for programming the instruction memory.

The register formats are shown in Figure 2-6 and the configuration fields within the registers are explained in Table 2-8, Table 2-9, and Table 2-10 for the three registers.

FUSE Word (Read/Program at FFFh main memory map)

$\overline{\text{TURBO}}$	$\overline{\text{SYNC}}$	$\overline{\text{OPTIONX}}$	$\overline{\text{STACKX}}$	$\overline{\text{IRC}}$	DIV2	DIV1	DIV0	$\overline{\text{CP}}$	WDTE	FOSC1	FOSC0
Bit 11											**Bit 0**

FUSEX Word (Read/Program via Programming Command)

Preset	Preset	Preset	Preset	Preset	$\overline{\text{CF}}$	BOR1	BOR0	RAM1	RAM0	MEM1	MEM0
Bit 11											**Bit 0**

DEVICE Word (Hard-Wired Read-Only)

Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	RAM1	RAM0	MEM1	MEM0
Bit 11											**Bit 0**

Figure 2 -6 Device Configuration Register Formats

Table 2-8 FUSE Word Register Configuration Bits

Option Bits	Description
$\overline{\text{TURBO}}$	Turbo Mode. Set to 1 for "compatible" mode, in which the instruction rate operates at one-fourth the oscillator clock rate. Set to 0 for the turbo mode, in which the instruction rate is equal to the oscillator clock rate.
$\overline{\text{SYNC}}$	Synchronous Input Mode (for turbo mode operation). Set to 1 to disable or clear to 0 to enable synchronous inputs (for debugging purposes).
$\overline{\text{OPTIONX}}$	OPTION Register Extension. Set to 1 to disable the programmability of bit 6 and bit 7 in the OPTION register, the RTW and RTE_IE bits (in other words, to force these two bits to 1). Clear to 0 to enable programming of the RTW and RTE_IE bits in the OPTION register.
$\overline{\text{STACKX}}$	Stack Extension. Set to 1 to limit the stack size to two locations. Clear to 0 to extend the stack size to eight locations.
$\overline{\text{IRC}}$	Internal RC Oscillator. Set to 1 to disable the internal oscillator and have the OSC1 and OSC2 pins operate as defined by the FOSC1:FOSC0 bits. Clear to 0 to enable the internal oscillator, and to have the OSC1 pin weakly pulled low and the OSC2 pin weakly pulled high.
DIV2:DIV0	Internal RC Oscillator Divider. This field sets the divide-by factor for generating the instruction clock from the internal oscillator when the internal oscillator is enabled ($\overline{\text{IRC}}$ = 0). The nominal instruction rate is determined by DIV2:DIV0 as follows: 000 = 4 MHz 001 = 2 MHz 010 = 1 MHz 011 = 500 kHz 100 = 250 kHz 101 = 125 kHz 110 = 62.5 kHz 111 = 31.25 kHz
$\overline{\text{CP}}$	Code Protection. Set to 1 for no code protection. Clear to 0 for code protection. With code protection, the program code and configuration registers read back as scrambled data. This prevents reverse-engineering of your proprietary code and configuration options.
WDTE	Watchdog timer enable. Set to 1 to enable the Watchdog timer. Clear to 0 to disable the Watchdog timer.
FOSC1: FOSC0	External Oscillator Configuration. This field sets up the device to operate with a particular type of external oscillator when the device is configured to operate with an external oscillator ($\overline{\text{IRC}}$ = 1). The type of external oscillator is determined by FOSC1:FOSC0 as follows: 00 = LP – low-power crystal 01 = HS – high-speed crystal 10 = XT – normal crystal 11 = RC network – OSC2 is weakly pulled high and no CLKOUT output

Table 2-9 FUSEX Word Register Configuration Bits

Option Bits	Description
Preset	Preset factory-configured bits. Do not alter these bits when you program the register.
$\overline{CF}$	Carry Flag Input. Set to 1 to ignore the carry flag as an input to addition and subtraction operations. Clear to 0 to add the carry flag into all addition operations (ADD fr,W means fr = fr + W + C); and to subtract the complement of the carry flag from all subtraction operations (SUB fr,W means fr = fr – W – /C).
BOR1: BOR0	Brown-Out Reset. These two factory-configured bits should not be changed unless you want to disable the brown-out functionality of the device. To disable the brown-out feature, set both of these bits to 1.
RAM1: RAM0	Configured Number of RAM Banks. These two factory-configured bits should not be changed unless you want to reduce the configured amount of RAM in the device. To do so, use one the following RAM1:RAM0 settings: 00 = 1 bank 01 = 2 banks 10 = 4 banks 11 = 8 banks (default)
MEM1: MEM0	Configured Memory Size. These two factory-configured bits should not be changed unless you want to reduce the configured amount of program memory in the device. To do so, use one the following MEM1:MEM0 settings: 00 = 1 page, 1 bank 01 = 1 page, 2 banks 10 = 4 pages, 4 banks 11 = 4 pages, 8 banks (default)

Table 2-10 DEVICE Word Register Configuration Bits (Read-Only)

Option Bits	Description
Res.	Reserved bits. These bits are for identifying different versions of the device when those versions become available. In this device, these bits are set to 1.
RAM1: RAM0	Absolute Number of RAM Banks. These two factory-configured bits identify the amount of RAM in the device: 00 = 1 bank 01 = 2 banks 10 = 4 banks 11 = 8 banks (actual setting for this device)
MEM1: MEM0	Absolute Memory Size. These two factory-configured bits identify the amount of program memory in the device: 00 = 512 words 01 = 1024 words 10 = 2048 words (actual setting for this device) 11 = 4096 words

Chapter 3

Instruction Set

3.1 Introduction

The Scenix SX microcontrollers use a RISC (Reduced Instruction Set Computer) architecture. In this type of architecture, the instruction set is limited in complexity and diversity, but the instructions can be executed very fast, typically at a rate of one instruction per clock cycle. High performance is achieved by executing many simple instructions very fast.

The instruction set consists entirely of single-word (12-bit) instructions, most of which can be executed at a rate of one instruction per clock cycle, for a total throughput of up to 50 MIPS (million instructions per second) when the device operates with a 50 MHz clock. The only common instructions that take more than one clock cycle to execute are those that control program flow, such as call and return instructions, and test-and-skip instructions that result in a skip.

3.2 Registers

An SX program consists of a sequence of instructions stored in the device program memory. Each instruction, when executed, changes the data contained in one or more device registers. All data registers are eight bits wide.

Most of the device registers are memory-mapped. Each memory-mapped register occupies an address in the data memory address space, and can be accessed by the “mov” instructions of the SX instruction set. An instruction refers to a memory-mapped register by specifying its 5-bit address in the instruction. Multiple sets or “banks” of registers are available at addresses 10h thorough 1Fh, as specified by the File Select Register (FSR).

The W (Working) register is used in many of the instructions but is not memory-mapped. It is often used as the source or destination of an operation. The letter “W” represents this register in the syntax of the assembly language.

There are eight dedicated-purpose registers and 136 general-purpose registers in the data memory address space, organized as shown in Tables 2-1. The numbers along the left side the table (ranging from $00 to $1F) show the 32 possible register addresses that can be specified an instruction. The bank numbers listed across the top (ranging from Bank 0 to Bank 7) are the numbers that can be programmed into the three high-order bits of the FSR register. The entries inside the table shows the registers accessed by each combination of register address and bank selection. For more information on how to access these registers, see Section .

3.3 Addressing Modes

The device supports the following addressing modes:

- Immediate
- Data Direct
- Data Indirect

3.3.1 Immediate Addressing

In the immediate addressing mode, the source data for an operation is provided in the instruction itself. In the syntax of the assembly language, the “number” or “pound” character (#) indicates an immediate value. Here is one example:

```
mov  W,#$0F    ;move immediate value 0Fh into W
```

The immediate value 0Fh is loaded into the W register. The 8-bit immediate value occupies an eight-bit field in the instruction opcode.

3.3.2 Direct Addressing

The 136 general-purpose registers and eight dedicated-purpose registers shown in Table 2-1 are memory-mapped file registers. Each register is readily accessible to the application program by specifying its 5-bit address in the instruction. This type of register access is called “direct addressing.” For example, to increment register 0Fh, you would use the following instruction:

```
inc  $0F       ;increment file register 0Fh
```

In this example, it is not necessary to be concerned about the bank selection because the register at this address is always accessible.

To increment a register in the range of 10h through 1Fh using direct addressing, you first need to specify the bank number if it is not already set to the desired bank. The bank is selected by the three high-order bits of the FSR register. For example, to increment the register at FFh (register 1Fh in Bank 7), you could use the following instructions:

```
mov  W,#$E0    ;load W with E0h
mov  FSR,W     ;set upper 3 bits of FSR register (Bank 7)
inc  $1F       ;increment file register 1Fh in Bank 7
```

All eight FSR register bits are used for “indirect addressing” (described in the next section). With indirect addressing, you specify the whole eight-bit address of the file register in FSR, including the 3-bit bank number and the 5-bit address within the bank. For this reason, the conventional practice is to set bit 4 in FSR whenever you set it to select Bank 1 through Bank 7, so that FSR contains a valid register address (in the upper half of the bank). For example:


```
mov  W,#$F0    ;load W with F0h
mov  FSR,W     ;write a valid Bank 7 address (F0h) to FSR
inc  $1F       ;increment file register 1Fh in Bank 7
```

The "bank" instruction modifies the upper three bits of FSR without affecting the other bits in that register. In the syntax of the assembly language, you specify the desired bank number as an 8-bit address. The assembler encodes the three high-order bits of the specified value into the instruction opcode and ignores the low-order bits. For example, to perform the same function as the example above, you can use the following code`:

```
bank $F0       ;select Bank 7 in FSR (set 3 high-order bits)
inc $1F        ;increment file register 1Fh in Bank 7
```

3.3.3 Indirect Addressing

With indirect addressing, you specify the full 8-bit address of a register using FSR as a pointer. This addressing mode provides the flexibility to access different registers or multiple registers using the same instruction in the program.

There are two addresses used for indirect addressing: FSR (File Select Register) at address 04h and INDF (Indirect through FSR) at address 00h. There is no real register located at address 00h. Instead, when you specify 00h as the source or destination of an operation, the device accesses the register pointed to by the FSR register. For example:

```
mov  W,#$F5    ;load W with F5h
mov  $04,W     ;move value F5h into FSR
mov  W,#$01    ;load W with 01h
mov  $00,W     ;move value 01h into register F5h
```

In the second "mov" instruction, FSR is loaded with the desired 8-bit register address. In the fourth "mov" instruction, address 00h is specified as the destination, so the device looks at FSR and moves the result to the register addressed by FSR, which is the register at F5h (register 15h in Bank 7).

A practical example that uses indirect addressing is the following program, which sets all the registers from 10h to 1Fh in all eight banks to FFh:

A practical example that uses indirect addressing is the following program, which clears all the registers from 10h to 1Fh in all eight banks:

```
  clr   FSR        ;clear FSR to 00h (at address 04h)
:loop   setb FSR.4 ;set bit 4: address 10h-1Fh, 30-3Fh, etc.
  clr   INDF       ;clear register pointed to by FSR
  incsz FSR        ;increment FSR and test, skip jmp if 00h
  jmp   :loop      ;jump back and clear next register
```

This program initially clears FSR to 00h. At the beginning of the loop, it sets bit 4 of FSR so that it starts at 10h. The "mov" instruction writes FFh to the register pointed to by FSR (initially, the file register at 10h in Bank 0). Then the program increments FSR and writes FFh to consecutive file registers, always in the upper half of each bank: (10h, 11h, 12h ... 1Fh, 30h, 31h ... FFh). The loop ends when FSR wraps back to 00h.

3.4 Instruction Types

The instructions are divided into the following categories:

- Logic Instructions
- Arithmetic and Shift Instructions
- Bitwise Operation Instructions
- Data Movement Instructions
- Program Control Instructions
- System Control Instructions

The following subsections describe the characteristics of the instructions in these categories.

3.4.1 Logic Instructions

Each logic instruction performs a standard logical operation (AND, OR, exclusive OR, or logical complement) on the respective bits of the 8-bit operands. The result of the logic operation is written to W or to a file register.

All of these instructions take one clock cycle for execution.

3.4.2 Arithmetic and Shift Instructions

Each arithmetic or shift instruction performs an operation such as add, subtract, rotate left or right through carry, increment, decrement, clear to zero, or swap high/low nibbles.

The device can be configured either to use or not use the carry flag as an implicit input to addition and subtraction operations. This option is controlled by the $\overline{CF}$ bit in the FUSEX Word (a word that is programmed at the same time as the program memory). In the default configuration, the carry flag is not used as an input to these operations. In that case, the carry flag can still be added or subtracted explicitly by using a separate “test carry bit” instruction in conjunction with an “increment” or “decrement” instruction.

There are instructions are available that increment or decrement a register and simultaneously test the result. If the 8-bit result is zero, the next instruction in the program is skipped. These instructions can be used to make program loops.

All of the arithmetic and shift instructions take one clock cycle for execution, except in the case of the test-and-skip instructions when the tested condition is true and a skip occurs.

3.4.3 Bitwise Operation Instructions

There are four bitwise operation instructions:

- “setb” sets a single bit to 1 in a data register without affecting other bits

- "clrb" clears a single bit to 0 in a data register without affecting other bits
- "sb" tests a single bit in a data register and skips the next instruction if the bit is set to 1
- "snb" tests a single bit in a data register and skips the next instruction if the bit is cleared to 0

Any bit in any memory-mapped register can be set, cleared, or tested individually, including bits in the program counter, FSR register, and STATUS register. These instructions are often used to set, clear, and test bits in the STATUS register.

All of the bitwise operation instructions take one clock cycle for execution, except in the case of the test-and-skip instructions when the tested condition is true and a skip occurs.

3.4.4 Data Movement Instructions

Each data movement instruction moves a byte of data from one register to another, or performs an operation on the contents of a source register and simultaneously moves the result into W (without affecting the source register). The following operations can be performed simultaneously with data movement into W: add, subtract, complement, increment, decrement, rotate left, rotate right, and swap high/low nibbles.

Instructions are also available that simultaneously increment or decrement the contents of a register, move the result into W, and test the result. If the 8-bit result is zero, the next instruction in the program is skipped.

Additional data movement instructions are provided to access the port control registers, the MODE register, and the OPTION register, which are not accessible as ordinary file registers.

All of the data movement instructions take one clock cycle for execution, except in the case of the test-and-skip instructions when the tested condition is true and a skip occurs.

3.4.5 Program Control Instructions

Each program control instruction alters the flow of the program by changing the contents of the program counter. Included in this category are the jump, call, and return-from-subroutine instructions.

The "jmp" instruction has a single operand that specifies the new address at which to resume execution. The new address is typically specified as a label, as in the following example:

```
snb  STATUS.0     ;check carry bit and skip next if C=0
jmp  do_carry     ;jump to do_carry routine if C=1
...
do_carry          ;jump destination label
...               ;program execution continues here
```

If the carry bit is set to 1, the "jmp" instruction is executed and program execution continues where the "do_carry" label appears in the program.

The "call" instruction works in a similar manner, except that it saves the contents of the program counter before jumping to the new address. Therefore, it calls a subroutine that can be terminated by any of several "return" instructions, as shown in the following example:

```
...
call add_2bytes   ;call subroutine add_carry
...               ;subroutine results used here
add_2bytes        ;subroutine label
...               ;subroutine code here
ret               ;return from subroutine
```

Returning from a subroutine restores the saved program counter contents, which causes program to resume execution with the instruction immediately following the "call" instruction.

A program memory address contains 11 bits. The "jmp" instruction specifies only the lowest nine bits of the jump address and the "call" instruction specifies only the lowest eight bits of the call address. For information on how the device handles the higher-order program address bits, see Section .

An indirect (register-specified) jump can be accomplished by moving the desired jump address from W to the PC register (mov $02,W). An indirect relative jump can be accomplished by adding W to the PC register (add $02,W).

Program control instructions such as "jmp," "call," and "ret" alter the normal program sequence. Therefore, when one of these instructions is executed, the execution pipeline is automatically cleared of pending instructions and refilled with new instructions, starting at the new program address. Because the pipeline must be cleared, multiple clock cycles are required for execution. The typical execution time for one of these instructions is two or three clock cycles, depending on the specific instruction and the device configuration mode ("compatible" or "turbo" clocking mode). For the exact number of clock cycles required, see the instruction set summary tables or the detailed instruction descriptions.

3.4.6 System Control Instructions

A system control instruction performs a special-purpose operation that sets the operating mode of the device or reads data from the program memory. Included in this category are the following instructions:

- "bank" loads a bank number into the FSR register
- "iread" reads a word from the program memory
- "page" writes the page number bits in the STATUS register
- "sleep" places the device in the power down mode

All of these instructions take one clock cycle for execution, except in the case of the "iread" instruction in the "turbo" device clocking mode, which takes four clock cycles.

3.5 Instruction Summary Tables

Tables 3-1 through 3-6 list all of the instructions, organized by category. For each instruction, the table shows the instruction mnemonic (as written in assembly language), a brief description of what the instruction does, the number of instruction cycles required for execution, the binary opcode, and the status flags affected by the instruction.

The "Cycles" column typically shows a value of 1, which means that the overall throughput for the instruction is one per clock cycle. In some cases, the exact number of cycles depends on the outcome of the instruction (such as the test-and-skip instructions) or the clocking mode (Compatible or Turbo). In those cases, all possible numbers of cycles are shown in the table.

The instruction execution time is derived by dividing the oscillator frequency be either one (Turbo mode) or four (Compatible mode). The divide-by factor is selected through the FUSE Word register

The detailed instruction descriptions in Section 3.6 fully explain the operation of each instruction, including the flags affected, the number of cycles required for execution, and usage examples.

Table 3-1 Logic Instructions

Syntax	Description	Cycles		Opcode	Flags
		Comp.	Turbo		
AND fr, W	AND of fr and W into fr	1	1	0001 011f ffff	Z
AND W, fr	AND of W and fr into W	1	1	0001 010f ffff	Z
AND W,#lit	AND of W and Literal into W	1	1	1110 kkkk kkkk	Z
NOT fr	Complement of fr into fr	1	1	0010 011f ffff	Z
OR fr,W	OR of fr and W into fr	1	1	0001 001f ffff	Z
OR W,fr	OR of W and fr into fr	1	1	0001 000f ffff	Z
OR W,#lit	OR of W and Literal into W	1	1	1101 kkkk kkkk	Z
XOR fr,W	XOR of fr and W into fr	1	1	0001 010f ffff	Z
XOR W,fr	XOR of W and fr into W	1	1	0001 100f ffff	Z
XOR W,#lit	XOR of W and Literal into W	1	1	1111 kkkk kkkk	Z

Table 3-2 Arithmetic and Shift Instructions (Sheet 1 of 2)

Syntax	Description	Cycles		Opcode	Flags
		Comp.	Turbo		
ADD fr,W	Add W to fr	1	1	0001 111f ffff	C, DC, Z
ADD W,fr	Add fr to W	1	1	0001 110f ffff	C, DC, Z
CLR fr	Clear fr	1	1	0000 011f ffff	Z
CLR W	Clear W	1	1	0000 0100 0000	Z
CLR !WDT	Clear Watchdog Timer	1	1	0000 0000 0100	TO, PD
DEC fr	Decrement fr	1	1	0000 111f ffff	Z
DECSZ fr	Decrement fr and Skip if Zero	1 or 2 (skip)	1 or 2 (skip)	0010 111f ffff	none
INC fr	Increment fr	1	1	0010 101f ffff	Z
INCSZ fr	Increment fr and Skip if Zero	1 or 2 (skip)	1 or 2 (skip)	0011 111f ffff	none
RL fr	Rotate fr Left through Carry	1	1	0011 011f ffff	C

Table 3-2 Arithmetic and Shift Instructions (Sheet 2 of 2)

Syntax	Description	Cycles		Opcode	Flags
		Comp.	Turbo		
RR fr	Rotate fr Right through Carry	1	1	0011 001f ffff	C
SUB fr,W	Subtract W from fr	1	1	0000 101f ffff	C, DC, Z
SWAP fr	Swap High/Low Nibbles of fr	1	1	0011 101f ffff	none

Table 3-3 Bitwise Operation Instructions

Syntax	Description	Cycles		Opcode	Flags
		Comp.	Turbo		
CLRB fr.bit	Clear Bit in fr	1	1	0100 bbbf ffff	none
SB fr.bit	Test Bit in fr and Skip if Set	1 or 2 (skip)	1 or 2 (skip)	0111 bbbf ffff	none
SETB fr.bit	Set Bit in fr	1	1	0101 bbbf ffff	none
SNB fr.bit	Test Bit in fr and Skip if Clear	1 or 2 (skip)	1 or 2 (skip)	0110 bbbf ffff	none

Table 3-4 Data Movement Instructions (Sheet 1 of 2)

Syntax	Description	Cycles		Opcode	Flags
		Comp.	Turbo		
MOV fr,W	Move W to fr	1	1	0000 001f ffff	none
MOV W,fr	Move fr to W	1	1	0010 000f ffff	Z
MOV W,fr-W	Move (fr-W) to W	1	1	0000 100f ffff	C, DC, Z
MOV W,#lit	Move Literal to W	1	1	1100 kkkk kkkk	none
MOV W,/fr	Move Complement of fr to W	1	1	0010 010f ffff	Z
MOV W,--fr	Move (fr-1) to W	1	1	0000 110f ffff	Z
MOV W,++fr	Move (fr+1) to W	1	1	0010 100f ffff	Z

Table 3-4 Data Movement Instructions (Sheet 2 of 2)

Syntax	Description	Cycles		Opcode	Flags
		Comp.	Turbo		
MOV W,<<fr	Rotate fr Left through Carry and Move to W	1	1	0011 010f ffff	C
MOV W,>>fr	Rotate fr Right through Carry and Move to W	1	1	0011 000f ffff	C
MOV W,<>fr	Swap High/Low Nibbles of fr and move to W	1	1	0011 100f ffff	none
MOV W,M	Move MODE Register to W	1	1	0000 0100 0010	none
MOVSZ W,--fr	Move (fr-1) to W and Skip if Zero	1 or 2 (skip)	1 2 (skip)	0010 110f ffff	none
MOVSZ W,++fr	Move (fr+1) to W and Skip if Zero	1 or 2 (skip)	1 2 (skip)	0011 110f ffff	none
MOV M,W	Move W to MODE Register	1	1	0000 0100 0011	none
MOV M,#lit	Move Literal to MODE Register	1	1	0000 0101 kkkk	none
MOV !rx,W	Move W to Port Rx Control Register	1	1	0000 0000 0fff	none
MOV !OPTION, W	Move W to OPTION Register	1	1	0000 0000 0010	none
TEST fr	Test fr for Zero	1	1	0010 001f ffff	Z

Table 3-5 Program Control Instructions

Syntax	Description	Cycles		Opcode	Flags
		Comp.	Turbo		
CALL addr8	Call Subroutine	2	3	1001 kkkk kkkk	none
JMP addr9	Jump to Address	2	3	101k kkkk kkkk	none
NOP	No Operation	1	1	0000 0000 0000	none
RET	Return from Subroutine	2	3	0000 0000 1100	none
RETP	Return from Subroutine Across Page Boundary	2	3	0000 0000 1101	none
RETI	Return from Interrupt	2	3	0000 0000 1110	all Status
RETIW	Return from Interrupt and Adjust RTCC with W	2	3	0000 0000 1111	all Status
RETW lit	Return from Subroutine with Literal in W	2	3	1000 kkkk kkkk	none

Table 3-6 System Control Instructions

Syntax	Description	Cycles		Opcode	Flags
		Comp.	Turbo		
BANK addr8	Load Bank Number into FSR(7:5)	1	1	0000 0001 1nnn	none
IREAD	Read Word from Instruction Memory	1	4	0000 0100 0001	none
PAGE addr12	Load Page Number into STATUS(7:5)	1	1	0000 0001 0nnn	none
SLEEP	Power Down Mode	1	1	0000 0000 0011	TO, PD

3.6 Equivalent Assembler Mnemonics

Some assemblers support additional instruction mnemonics that are special cases of existing instructions or alternative mnemonics for standard ones. For example, an assembler might support the mnemonic “CLC” (clear carry), which is interpreted the same as the instruction “clrb $03.0” (clear bit 3 in the STATUS register). Some of the commonly supported equivalent assembler mnemonics are described in Table 3-7.

Table 3-7 Equivalent Assembler Mnemonics

Syntax	Description	Equivalent	Cycles
CLC	Clear Carry Flag	CLRB $03.0	1
CLZ	Clear Zero Flag	CLRB $03.2	1
JMP W	Jump Indirect W	MOV $02,W	4 or 3 (note 1)
JMP PC+W	Jump Indirect W Relative	ADD $02,W	4 or 3 (note 1)
MODE imm4	Move Immediate to MODE Register	MOV M,#lit	1
NOT W	Complement W	XOR W,#$FF	1
SC	Skip if Carry Flag Set	SB $03.0	1 or 2 (note 2)
SKIP	Skip Next Instruction	SNB $02.0 or SB $02.0	4 or 2 (note 3)

NOTES:
1. The JMP W or JMP PC+W instruction takes 4 cycles in the “compatible” clocking mode or 3 cycles in the “turbo” clocking mode.
2. The SC instruction takes 1 cycle if the tested condition is false or 2 cycles if the tested condition is true.
3. The assembler converts the SKIP instruction into a SNB or SB instruction that tests the least significant bit of the program counter, choosing SNB or SB so that the tested condition is always true. The instruction takes 4 cycles in the “compatible” clocking mode or 2 cycles in the “turbo” clocking mode.

3.7 Detailed Instruction Descriptions

Each instruction in the SX instruction set is described in detail in the following pages. The instructions are described in alphabetical order by mnemonic name.

Each description starts on a new page of the manual. The heading at the top of the page shows the syntax of the command and a brief description of what the command does.

In the syntax description, the parts that are to be used literally are shown in upper case and the variable parts are shown in lower case and. For example, the “add W to file register” command is shown as follows:

```
add  fr,W
```

The “ADD” and “W” should be used exactly as shown in the command syntax, whereas the lower-case notation “fr” means that you should use a file register address, which can be any value from $00 to $1F, or an equivalent symbol. In an actual program, you can use either upper-case or lower-case characters. Here is an example of an actual “add W to register” command:

```
add  $0F,W     ;add contents of W to file register 0Fh
```

The text after the semicolon is a comment, which is ignored by the assembler.

Each instruction description includes the following information:

- Operation. This section describes the effects of the command in equation form. For example, the “add W to file register” command shows the operation as “fr = fr + W” (fr is set equal to the sum of fr plus W).
- Flags Affected. This is a list of the status flags that are affected by execution of the command, such as the carry (C) and zero (Z) flags.
- Opcode. This is the 12-bit opcode of the encoded instruction, shown in binary format. Bits that depend on variables are shown as letters rather than 0 or 1. For example, the opcode for the “ADD fr,W” instruction is shown as `0001 110f ffff`. The sequence of five “f” characters represents the five-bit file register address specified in the instruction. The letter “k” or “n” is similarly used to represent the constant or number specified in the instruction.
- Description. This is a verbal description of what the instruction does.
- Cycles. This is the number of clock cycles required to execute the instruction. In cases where this number depends on certain conditions, those conditions and the resulting numbers are explained. In some cases, the number depends on the clocking mode (“turbo” or “compatible” mode). In the “compatible” mode, the number shown is the number of regular instruction cycles required for execution, each cycle consisting of four device clocks.
- Example. At least one example of the instruction is provided, together with an explanation of how the example operates.

In some cases, there is an additional section called “Config. Option,” which explains how the behavior of the instruction is affected by the device configuration.

Some assemblers support additional instruction mnemonics that are special cases of existing instructions. Also, some assemblers support “macro” mnemonics, which are assembled into multiple instructions. These additional assembler mnemonics are beyond the scope of this section. For more information, see the documentation provided with the assembler.

Table 3-8 is a quick reference to the abbreviations and symbols used in the instruction descriptions.

Table 3-8 Key to Abbreviations and Symbol

Symbol	Description
W	Working register
fr	File register (memory-mapped register in the range of 00h to FFh)
PC	Lower eight bits of program counter (file register 02h)
STATUS	STATUS register (file register 03h)
FSR	File Select Register (file register 04h)
C	Carry flag in STATUS register (bit 0)
DC	Digit Carry flag in STATUS register (bit 1)
Z	Zero flag in STATUS register (bit 2
PD	Power Down flag in STATUS register (bit 3)
TO	Watchdog Timeout flag in STATUS register (bit 4)
PA2:PA0	Page select bits in STATUS register (bits 7:5)
OPTION	OPTION register (not memory-mapped)
WDT	Watchdog Timer register (not memory-mapped)
MODE	MODE register (not memory-mapped)
rx	Port control register pointer (RA, RB, or RC)
!	Non-memory-mapped register designator
f	File register address bit in opcode
k	Constant value bit in opcode
n	Numerical value bit in opcode
b	Bit position selector bit in opcode
.	File register / bit selector separator in assembly language instruction
#	Immediate literal designator in assembly language instruction
lit	Literal value in assembly language instruction
addr8	8-bit address in assembly language instruction
addr9	9-bit address in assembly language instruction
addr12	12-bit address in assembly language instruction
/	Logical 1’s complement
|	Logical OR
^	Logical exclusive OR
&	Logical AND
<>	Swap high and low nibbles (4-bit segments)
<<	Rotate left through carry flag
>>	Rotate right through carry flag
- -	Decrement file register
++	Increment file register

3.7.1 ADD fr,W — Add W to fr

Operation: fr = fr + W

Flags affected: C, DC, Z

Opcode: `0001 111f ffff`

Description: This instruction adds the contents of W to the contents of the specified file register and writes the 8-bit result into the same file register. W is left unchanged. The register contents are treated as unsigned values.

If the result of addition exceeds FFh, the C flag is set and the lower eight bits of the result are written to the file register. Otherwise, the C flag is cleared.

If there is a carry from bit 3 to bit 4, the DC (digit carry) flag is set. Otherwise, the flag is cleared.

If the result of addition is 00h, the Z flag is set. Otherwise, the flag is cleared. An addition result of 100h is considered zero and therefore sets the Z flag.

Config. Option: If the $\overline{CF}$ bit in the FUSEX configuration register has been programmed to 0, this instruction also adds the C flag as a carry-in input:

fr = fr + W + C

Cycles: 1

Example:
```
add   $12,W
```

This example adds the contents of W to file register 12h. For example, if the file register contains 7Fh and W contains 02h, this instruction adds 02h to 7Fh and writes the result, 81h, into the file register; and clears the C and Z flags. It sets the DC flag because of the carry from bit 3 to bit 4.

3.7.2 ADD W,fr — Add fr to W

Operation: W =W + fr

Flags affected: C, DC, Z

Opcode: `0001 110f ffff`

Description: This instruction adds the contents of the specified file register to the contents of W and writes the 8-bit result into W. The file register is left unchanged. The register contents are treated as unsigned values.

If the result of addition exceeds FFh, the C flag is set and the lower eight bits of the result are written to W. Otherwise, the C flag is cleared.

If there is a carry from bit 3 to bit 4, the DC (digit carry) flag is set. Otherwise, the flag is cleared.

If the result of addition is 00h, the Z flag is set. Otherwise, the flag is cleared. An addition result of 100h is considered zero and therefore sets the Z flag.

Config. Option: If the $\overline{CF}$ bit in the FUSEX register has been programmed to 0, this instruction also adds the C flag as a carry-in input:

W = W + fr + C

Cycles: 1

Example:

```
add   W,$12
```

This example adds the contents of file register 12h to W. For example, if the file register contains 81h and W contains 82h, this instruction adds 81h to 82h and writes the lower eight bits of the result, 03h, into W. It sets the C flag because of the carry out of bit 7, and clears the DC flag because there is no carry from bit 3 to bit 4. The Z flag is cleared because the result is nonzero.

3.7.3 AND fr,W — AND of fr and W into fr

Operation: fr = fr & W

Flags affected: Z

Opcode: 0001 011f ffff

Description: This instruction performs a bitwise logical AND of the contents of the specified file register and W, and writes the 8-bit result into the same file register. W is left unchanged. If the result is 00h, the Z flag is set.

Cycles: 1

Example:
```
and   $10,W    ;perform logical AND and overwrite fr
```

This example performs a bitwise logical AND of the working register W with a value stored in file register 10h. The result is written back to the file register 10h.

For example, suppose that the file register 10h is loaded with the value 0Fh and W contains the value 13h. The instruction takes the logical AND of 0Fh and 13h and writes the result, 03h, into the same file register. The result is nonzero, so the Z flag is cleared.

3.7.4 AND W,fr — AND of W and fr into W

Operation: W = W & fr

Flags affected: Z

Opcode: `0001 010f ffff`

Description: This instruction performs a bitwise logical AND of the contents of W and the specified file register, and writes the 8-bit result into W. The file register is left unchanged. If the result is 00h, the Z flag is set.

Cycles: 1

Example:
```
and   W,$0B    ;perform logical AND and overwrite W
```

This example performs a bitwise logical AND of the value stored in file register 0Bh with W. The result is written back to W.

For example, suppose that the file register 0Bh is loaded with the value 0Fh and W contains the value 13h. The instruction takes the logical AND of 0Fh and 13h and writes the result, 03h, into W. The result is nonzero, so the Z flag is cleared.

3.7.5 AND W,#lit — AND of W and Literal into W

Operation: W = W & lit

Flags affected: Z

Opcode: `1110 kkkk kkkk`

Description: This instruction performs a bitwise logical AND of the contents of W and an 8-bit literal value, and writes the 8-bit result into W. If the result is 00h, the Z flag is set.

Cycles: 1

Example:
```
and   W,#$0F   ;mask out four high-order bits of W
```

This example performs a bitwise logical AND of W with the literal value #0Fh. The result is written back to W.

For example, suppose that W contains the value 50h. The instruction takes the logical AND of this value with 0Fh and writes the result, 00h, into W. The result is zero, so the Z flag is set.

3.7.6 BANK addr8 Load Bank Number into FSR(7:5)

Operation: FSR(7:5) = addr8(7:5)

Flags affected: none

Opcode: `0000 0001 1nnn`

Description: This instruction loads the three high-order bits of the File Select Register (FSR). These bits specify the data memory bank number for subsequent memory access instructions. You can specify any bank number from 0 to 7. The five low-order bits of FSR are not affected.

In the syntax of the assembly language, you specify the bank using a full 8-bit data memory address. The assembler encodes the three high-order bits of this address into the instruction opcode and ignores the five low-order bits.

Cycles: 1

Example:
```
bank  $F0      ;select bank 7
```

This example writes the three high-order bits of FSR with 111, which selects Bank 7 for subsequent data memory access instructions.

3.7.7 CALL addr8 Call Subroutine

Operation: top-of-stack = program counter + 1

PC(7:0) = addr8

program counter (8) = 0

program counter (10:9) =PA1:PA0

Flags affected: none

Opcode: `1001 kkkk kkkk`

Description: This instruction calls a subroutine. The full 11-bit address of the next program instruction is saved on the stack and the program counter is loaded with a new address, which causes a jump to that program address.

Bits 7:0 come from the 8-bit constant value in the instruction, bit 8 is always 0, and bits 10:9 come from the PA1:PA0 bits in the STATUS register. Therefore, the subroutine must start in the bottom half of a 512-word page in the program memory (000h to 0FFh, 200h to 2FFh, etc.).

The subroutine is terminated by any one of the "return" instructions, which restores the saved address to the program counter. Execution proceeds from the instruction following the "call" instruction.

Cycles: 2 in "compatible" mode, or 3 in "turbo" mode

Example:

```
page  $600     ;set page of subroutine in STATUS reg.
call  addxy    ;call subroutine addxy
mov   $0C,W    ;use addxy subroutine results
...            ;more of program (not shown)
addxy          ;subroutine address label
mov   W,$0E    ;subroutine instructions start here
add   W,$0F
...
ret            ;return from subroutine
```

The "call" instruction in this example calls a subroutine called "addxy." When the "call" instruction is executed, the address of the following instruction (the "mov $0C,W" instruction) is pushed onto the stack and the program jumps to the "addxy" routine. When the "ret" instruction is executed, the 11-bit program address saved on the stack is popped and restored to the program counter, which causes the program to continue with the instruction immediately following the "call" instruction.

The “addxy” routine must start in the lower half of a 512-word page of the program memory. This is because bit 8 of the subroutine address must be 0. The PA1:PA0 bits of the STATUS register must contain the two high-order bits of the subroutine address prior to the “call” instruction. This is the purpose of the “page” instruction.

3.7.8 CLR fr — Clear fr

Operation: fr = 0

Flags affected: Z

Opcode: `0000 011f ffff`

Description: This instruction clears the specified file register to zero. It also sets the Z flag unconditionally.

Cycles: 1

Example:
```
clr   $0A
```
This example clears file register 0Ah to 00h and sets the Z flag.

3.7.9 CLR W — Clear W

Operation: W = 0

Flags affected: Z

Opcode: `0000 0100 0000`

Description: This instruction clears W, the working register. It also sets the Z flag.

Cycles: 1

Example:
```
clr   W
```
This example clears W to 00h and sets the Z flag.

3.7.10 CLR !WDT Clear Watchdog Timer

Operation: Clears Watchdog timer counter and prescaler counter

Flags affected: Z

Opcode: `0000 011f ffff`

Description: This instruction clears the Watchdog Timer counter to zero. It also clears the Watchdog prescaler register to zero, and sets the Z, TO, and PD flags to 1 (the Zero, Watchdog Timeout, and Power Down flags).

If the Watchdog circuit is enabled, the application software must execute this instruction periodically in order to prevent a Watchdog reset.

Cycles: 1

Example:
```
clr   !WDT
```

This example clears the Watchdog Timer counter and the Watchdog prescaler register to zero; and sets the Z, TO, and PD flags.

3.7.11 CLRB fr,bit — Clear Bit in fr

Operation: Clear a specified bit in fr

Flags affected: none

Opcode: `0100 bbbf ffff`

Description: This instruction clears a bit in the specified file register to 0 without changing the other bits in the register. The file register address (00h through 1Fh) and the bit number (0 through 7) are the instruction operands.

Cycles: 1

Example:

```
clrb  $1F.7
```

This example clears the most significant bit of file register 1Fh.

3.7.12 DEC fr — Decrement fr

Operation: fr = fr -1

Flags affected: Z

Opcode: `0000 111f ffff`

Description: This instruction decrements the specified register file by one.

If the file register contains 01h, it is decremented to 00h and the Z flag is set. Otherwise, the flag is cleared.

If the file register contains 00h, it is decremented to FFh.

Cycles: 1

Example:
```
dec    $18
```
This example decrements file register 18h.

3.7.13 DECSZ fr — Decrement fr and Skip if Zero

Operation: fr = fr - 1

if 0, then skip next instruction

Flags affected: none

Opcode: `0010 111f ffff`

Description: This instruction decrements the specified register file by one and tests the new register value. If that value is zero, the next program instruction is skipped. Otherwise, execution proceeds normally with the next instruction.

Cycles: 1 if tested condition is false, 2 if tested condition is true

Example:

```
decsz $18
jmp   back1
mov   $19,W
```

The “decsz” instruction decrements file register 18h. If the result is nonzero, execution proceeds normally with the “jmp” instruction. If the result is zero, the device skips the “jmp” instruction and proceeds with the “mov” instruction.

3.7.14 INC fr — Increment fr

Operation: fr = fr +1

Flags affected: Z

Opcode: `0010 101f ffff`

Description: This instruction increments the specified register file by one.

If the file register contains FFh and is incremented to 00h, the Z flag is set. Otherwise, the flag is cleared.

Cycles: 1

Example:
```
inc   $18
```

This example increments file register 18h.

3.7.15 INCSZ fr — Increment fr and Skip if Zero

Operation: fr = fr + 1

if 0, then skip next instruction

Flags affected: none

Opcode: `0011 111f ffff`

Description: This instruction increments the specified register file by one and tests the new register value. If that value is zero, the next program instruction is skipped. Otherwise, execution proceeds normally with the next instruction.

Cycles: 1 if tested condition is false, 2 if tested condition is true

Example:

```
incsz $18
jmp   back1
mov   $17,W
```

The "incsz" instruction increments file register 18h. If the result is nonzero, execution proceeds normally with the "jmp" instruction. If the result is zero, the device skips the "jmp" instruction and proceeds with the "mov" instruction.

3.7.16 IREAD Read Word from Instruction Memory

Operation: MODE:W = data at (MODE:W)

Flags affected: none

Opcode: `0000 0100 0001`

Description: This instruction allows the device to transfer data from instruction memory into data memory. It concatenates the lower three bits of the MODE register with W to make an 11-bit address, using the MODE register bits for the high-order part and W for the low-order part. It reads the 12-bit word from program memory at that address. Then it writes the four high-order bits of the word into the lower four bits of the MODE register, and writes the eight low-order bits of the word into W. The four high-order bits of the MODE register are cleared to zero.

Figure 3-1 shows how the MODE and W register are used to specify the program memory address and to contain the 12-bit result.

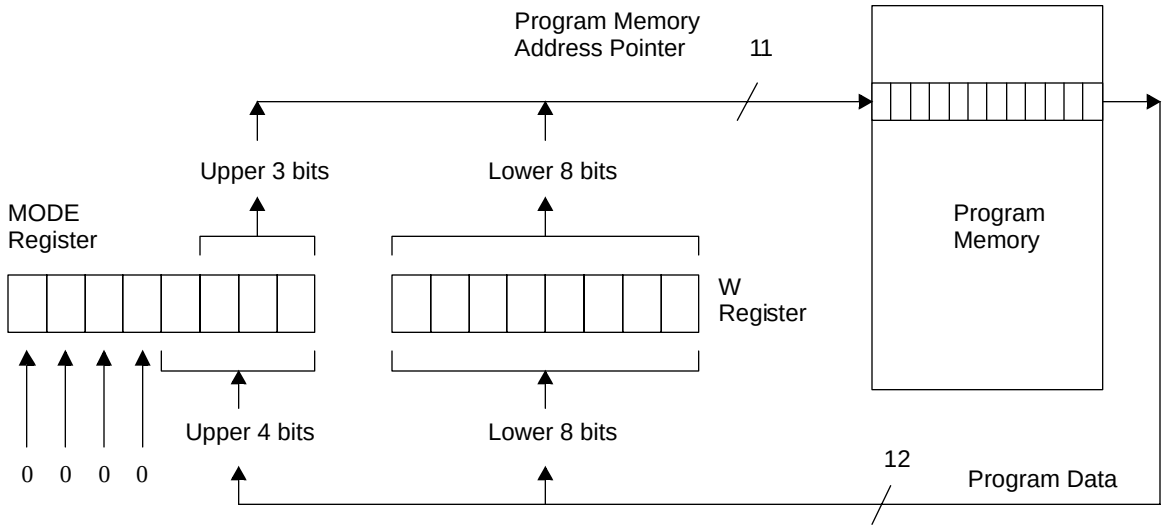

Figure 3-1 Program Counter Loading for Call Instruction

Cycles: 1

Example:

```
mov   W,#$03   ;load W with the value 03h
mov   M,W      ;move value 03h into MODE register
mov   W,#$80   ;load W with the value 80h
iread          ;read program address 380h into W & MODE
mov   $0E,W    ;move lower byte of data to reg 0Eh
mov   W,M      ;move upper 4 bits of data to W
mov   $0F,W    ;move upper 4 bits of data to reg 0Fh
```

This example reads the 12-bit data stored the program address 380h. The program first loads the MODE register and W with the 11-bit program address, 380h. After the “iread” instruction, the MODE register and W contain the 12-bit value stored in the program memory at address 380h. The program then stores the lower eight bits of the result into file register 0Eh and the upper four bits of the result into file register 0Fh.

3.7.17 JMP addr9 Jump to Address

Operation: PC(7:0) = addr9(7:0)

program counter (8) = addr9(8)

program counter (10:9) =PA1:PA0

Flags affected: none

Opcode: `101k kkkk kkkk`

Description: This instruction causes the program to jump to a specified address. It loads the program counter with the new address. The new 11-bit address is generated from two different sources. Bits 8:0 come from the 9-bit constant value in the instruction and bits 10:9 come from the PA1:PA0 bits in the STATUS register. The STATUS register must contain the appropriate value prior to the jump instruction.

Cycles: 2 in “compatible” mode, or 3 in “turbo” mode

Example:

```
page  $600     ;set page of jump addr. in STATUS reg.
snb   $03.0    ;test carry flag and skip if clear
jmp   overflo  ;jump to overflo routine if C=1
...            ;more of program (not shown)
overflo        ;
mov   W,$09    ;routine executed if C=1
...
```

This example shows one way to implement a conditional jump. The “jmp” instruction, if executed, causes a jump to the address of the “overflo” program label. The “snb” instruction (test bit and skip if clear) causes the “jmp” instruction to be either executed or skipped, depending on the state of the carry flag.

The PA1:PA0 bits of the STATUS register must contain the two high-order bits (bits 10 and 9) of the “overflo” routine address prior to the “jump” instruction. This is the purpose of the “page” instruction.

3.7.18 MOV fr,W — Move W to fr

Operation: fr = W

Flags affected: none

Opcode: `0000 001f ffff`

Description: This instruction moves the contents of W into the specified file register. W is left unchanged.

Cycles: 1

Example:

```
bank  $F0      ;select Bank 7
mov   $10,W    ;move W to reg. 10h in Bank 7
```

This example moves the contents of W into file register 10h in Bank 7.

3.7.19 MOV M,#lit — Move Literal to MODE Register

Operation: MODE = lit

Flags affected: none

Opcode: `0000 0101 kkkk`

Description: This instruction writes a 4-bit value to the lower four bits of the MODE register. These bits select the port control registers accessed by the MOV !rx,W instructions.

The following table shows the port control registers accessed for each MODE register setting.

MODE Reg.	mov !RA,W	mov !RB,W	mov !RC,W
08h	not used	CMP_B	not used
09h	not used	WKPND_B	not used
0Ah	not used	WKED_B	not used
0Bh	not used	WKEN_B	not used
0Ch	not used	ST_B	ST_C
0Dh	LVL_A	LVL_B	LVL_C
0Eh	PLP_A	PLP_B	PLP_C
0Fh	RA Direction	RB Direction	RC Direction

Cycles: 1

Example:

```
mov   M,#$0E   ;MODE=0Eh to access port pullup regs
mov   W,#$03   ;W = 0000 0011
mov   !RA,W    ;disable pullups for A0 and A1
mov   W,#$FF   ;W = 1111 1111
mov   !RB,W    ;disable all pullups for B0-B7
mov   W,#$00   ;W = 0000 0000
mov   !RC,W    ;enable all pullups for C0-C7
```

This example sets the pullup configuration for Ports A, B, and C.

3.7.20 MOV M,W — Move W to MODE Register

Operation: MODE = W

Flags affected: none

Opcode: `0000 0100 0011`

Description: This instruction moves the contents of W into the MODE register. W is left unchanged. The four least significant bits of the MODE register operate as a pointer to the device port registers for subsequent accesses to those registers using the MOV !rx,W instruction.

Cycles: 1

Example:

```
mov   W,$0B    ;move value from file reg 0Bh to W
mov   M,W      ;move W into MODE register
```

This example moves a value from file register 0Bh to W, and then from W into the MODE register.

3.7.21 MOV !OPTION,W — Move W to OPTION Register

Operation: OPTION = W

Flags affected: none

Opcode: `0000 0000 0010`

Description: This instruction moves W to the OPTION register. W is left unchanged. The OPTION register sets the Real-Time Clock/Counter (RTCC) configuration options such as RTCC interrupt enable, RTCC increment event control, and prescaler assignment.

Cycles: 1

Example:

```
mov   W,#$3F     ;load W with 3Fh
mov   !OPTION,W  ;write value to OPTION register
```

This example moves programs the OPTION register with the value 3Fh.

3.7.22 MOV !rx,W — Move W to Port Rx Control Register

Operation: rx <=> W (exchange W and WKPND_B or CMP_B) or

rx = W (move W to rx for all other port control registers)

Flags affected: none

Opcode: `0000 0000 0fff`

Description: This instruction moves W to one of the port control registers (rx). The port control register is specified in the instruction mnemonic as !RA, !RB, or !RC. This corresponds to a 3-bit field in the opcode that is set to 5, 6, or 7 to write a port control register for Port A, Port B, or Port C.

To access the port data register rather than the control register, use the "MOV fr,W" or similar instruction, addressing the port as "fr" rather than "!rx".

Each port has a set of control registers: one each for setting the data direction, the pullup configuration, the Schmitt trigger configuration, and so on. The MODE register setting determines the type of port control register accessed by the "MOV !rx,W" instruction.

For accesses to the WKPND_B or CMP_B register using this instruction, the device exchanges the data between the register and W. In other words, the device moves the contents of the port register into W, in addition to moving the contents of W into the port register. This provides a way for the program to read the contents of those registers.

Cycles: 1

Example 1:

```
mov   M,#$0F   ;write MODE reg to select data direction
mov   W,#$3F   ;pins 0-5 Hi-Z inputs, pins 6-7 outputs
mov   !RB,W    ;configure Port B pin data directions
mov   W,#$FF   ;all pins Hi-Z inputs
mov   !RC,W    ;configure Port C pin data directions
```

This example configures the data direction for each pin of Port B and Port C. The first instruction programs the MODE register to allow access to the port data direction registers. The second instruction loads W with the value 3Fh. The third instruction writes this value to the RB direction register, which configures pins 0 through 5 to operate as high-impedance inputs and pins 6 and 7 to operate as outputs. The last two instructions configure all Port C pins to operate as inputs.

Example 2:

```
mov   M,#$08   ;load MODE register to select CMP_B
clr   W        ;clear W
mov   !RB,W    ;00h into CMP_B and old CMP_B into W
               ;enables comparator and its output pin
```

This example enables the comparator and its output pin. The “mov !RB,W” instruction does an exchange of data between the CMP_B register and W.

3.7.23 MOV W,fr — Move fr to W

Operation: W = fr

Flags affected: Z

Opcode: `0000 001f ffff`

Description: This instruction moves the contents of the specified file register into W. The file register is left unchanged.

If the value is 00h, the Z flag is set. Otherwise, the flag is cleared.

Cycles: 1

Example:

```
bank  $30      ;select Bank 1
mov   W,$1E    ;move reg 1Eh in Bank 7 to W
```

This example moves the contents of file register 1Eh in Bank 1 into W. The Z flag is set if the value is zero or cleared if the value is nonzero.

3.7.24 MOV W,/fr — Move Complement of fr to W

Operation: W = fr ^ FFh

Flags affected: Z

Opcode: `0000 000f ffff`

Description: This instruction loads the one's complement of the specified file register into W The file register is left unchanged.

If the value loaded into W is 00h, the Z flag is set. Otherwise, the flag is cleared.

Cycles: 1

Example:

```
mov   W,/$0F
```

This example moves the one's complement of file register 0Fh into W. For example, if the file register contains 75h, the complement of this value, 8Ah, is loaded into W, and the Z flag is cleared. The file register still contains 75h after execution of the instruction.

3.7.25 MOV W,fr-W — Move (fr-W) to W

Operation: W = fr - W

Flags affected: C, DC, Z

Opcode: `0000 100f ffff`

Description: This instruction subtracts the contents of W from the contents of the specified file register and writes the 8-bit result into W. The file register is left unchanged. The register contents are treated as unsigned values.

If the result of subtraction is negative (W is larger than fr), the C flag is cleared to 0 and the lower eight bits of the result are written to W. Otherwise, the C flag is set to 1.

If there is a borrow from bit 3 to bit 4, the DC (digit carry) flag is cleared to 0. Otherwise, the flag is set to 1.

If the result of subtraction is 00h, the Z flag is set. Otherwise, the flag is cleared.

Config. Option: If the $\overline{CF}$ bit in the FUSEX configuration register has been programmed to 0, this instruction also subtracts the complement of the C flag as a borrow-in input:

W = fr - W - /C

Cycles: 1

Example:
```
mov   W,$0D-W
```

This example subtracts the contents of W from file register 0Dh and moves the result into W. For example, if the file register contains 35h and W contains 06h, this instruction subtracts 06h from 35h and writes the result, 2Fh, into W. It also sets the C flag, clears the DC flag, and clears the Z flag. The file register still contains 35h after execution of the instruction.

3.7.26 MOV W,--fr — Move (fr-1) to W

Operation: W = fr -1

Flags affected: Z

Opcode: `0000 110f ffff`

Description: This instruction decrements the value in the specified register file by one and moves the 8-bit result into W. The file register is left unchanged.

If the file register contains 01h, the value moved into W is 00h and the Z flag is set. Otherwise, the flag is cleared.

Cycles: 1

Example:
```
mov   w,--$18
```

This example decrements the value in file register 18h and moves the result into W. For example, if the file register contains 75h, the value 74h is loaded into W, and the Z flag is cleared. The file register still contains 75h after execution of the instruction.

3.7.27 MOV W,++fr — Move (fr+1) to W

Operation: W = fr + 1

Flags affected: Z

Opcode: `0010 100f ffff`

Description: This instruction increments the value in the specified register file by one and moves the 8-bit result into W. The file register is left unchanged.

If the file register contains FFh, the value moved into W is 00h and the Z flag is set. Otherwise, the flag is cleared.

Cycles: 1

Example:
```
mov   w,++$18
```

This example increments the value in file register 18h and moves the result into W. For example, if the file register contains 75h, the value 76h is loaded into W, and the Z flag is cleared. The file register still contains 75h after execution of the instruction.

3.7.28 MOV W,<<fr Rotate fr Left through Carry and Move to W

Operation: W = << fr

Flags affected: C

Opcode: `0011 010f ffff`

Description: This instruction rotates the bits of the specified file register left using the C flag bit and moves the 8-bit result into W. The file register is left unchanged.

The bits obtained from the register are shifted left by one bit position. C is shifted into the least significant bit position and the most significant bit is shifted out into C, as shown in the diagram below.

Cycles: 1

Example 1:
```
mov   W,<<$18
```

This example rotates the bits of file register 18h left through the C bit and moves the result into W. If the file register contains 14h and the C flag is set to 1, after this instruction is executed, W will contain 29h and the C flag will be cleared to 0. The file register will still contain 14h after execution of the instruction.

3.7.29 MOV W,>>fr Rotate fr Right through Carry and Move to W

Operation: W = >> fr

Flags affected: C

Opcode: `0011 000f ffff`

Description: This instruction rotates the bits of the specified file register right using the C flag bit and moves the 8-bit result into W. The file register is left unchanged.

The bits obtained from the register are shifted right by one bit position. C is shifted into the most significant bit position and the least significant bit is shifted out into C, as shown in the diagram below.

Cycles: 1

Example 1:
```
mov   W,>>$0F
```

This example rotates the bits of file register 0Fh right through the C bit and moves the result into W. If the file register contains 12h and the C flag is set to 1, after this instruction is executed, W will contain 89h and the C flag will be cleared to 0. The file register will still contain 12h after execution of the instruction.

3.7.30 MOV W,<>fr Swap High/Low Nibbles of fr and Move to W

Operation: W = <> fr

Flags affected: none

Opcode: `0011 100f ffff`

Description: This instruction exchanges the high-order and low-order nibbles (4-bit segments) of the value in the specified file register and moves the result to W. The file register is left unchanged.

Cycles: 1

Example:
```
mov   W,<>$0B
```

This example swaps the high-order and low-order nibbles of the value in file register 0Bh and move the result into W. For example, if the file register contains A5h, after executing this instruction, W will contain 5Ah.

3.7.31 MOV W,#lit — Move Literal to W

Operation: W = lit

Flags affected: none

Opcode: `1100 kkkk kkkk`

Description: This instruction loads an 8-bit literal value (a value specified within the instruction) into W.

Cycles: 1

Example:
```
mov   W,#$75
```
This example loads the immediate value 75h into W.

3.7.32 MOV W,M — Move MODE Register to W

Operation: W = MODE

Flags affected: none

Opcode: `0000 0100 0010`

Description: This instruction moves the contents of the MODE register into W. The MODE register is left unchanged. The four least significant bits of the MODE register operate as a pointer to the device port registers for subsequent accesses to those registers using the MOV !rx,W instruction.

Cycles: 1

Example:

```
mov   W,M      ;get MODE register contents
mov   $10,W    ;save value to file register 10h
```

This example moves the contents of the MODE register into W, and then stores that value into file register 10h.

3.7.33 MOVSZ W, --fr Move (fr-1) to W and Skip if Zero

Operation: W = fr - 1; if 0, then skip next instruction

Flags affected: none

Opcode: `0010 110f ffff`

Description: This instruction decrements the value in the specified file register and moves the result to W. The file register is left unchanged.

If the result is zero, the next instruction in the program is skipped. Otherwise, program execution proceeds normally with the next instruction.

Cycles: 1 if tested condition is false; 2 if tested condition is true

Example:

```
movsz W,--$1F  ;move register 1Fh -1 into W
ret            ;return from subroutine if 0
nop            ;execution continues here otherwise
```

This example takes the contents of file register 1Fh, decrements that value, and moves the result to W. If the result is zero, the device skips the “ret” instruction and proceeds with the “nop” instruction. If the result is nonzero, the device executes the “ret” instruction.

3.7.34 MOVSZ W, ++fr Move (fr+1) to W and Skip if Zero

Operation: W = fr + 1; if 0, then skip next instruction

Flags affected: none

Opcode: `0011 110f ffff`

Description: This instruction increments the value in the specified file register and moves the result to W. The file register is left unchanged.

If the result is zero, the next instruction in the program is skipped. Otherwise, program execution proceeds normally with the next instruction.

Cycles: 1 if tested condition is false; 2 if tested condition is true

Example:

```
movsz W,++$1F  ;move register 1Fh +1 into W
ret            ;return from subroutine if 0
nop            ;execution continues here otherwise
```

This example takes the contents of file register 1Fh, increments that value, and moves the result to W. If the result is zero, the device skips the “ret” instruction and proceeds with the “nop” instruction. If the result is nonzero, the device executes the “ret” instruction.

3.7.35 NOP No Operation

Operation: none

Flags affected: none

Opcode: `0000 0000 0000`

Description: This instruction does nothing except to cause a one-cycle delay in program execution.

Cycles: 1

Example:

```
sb    $05.4    ;set bit 4 in Port A
nop            ;no operation, 1-cycle delay
sb    $05.6    ;set bit 5 in Port A
```

This example shows how a “nop” instruction can be used as a one-cycle delay between two successive read-modify-write instructions that modify the same I/O port. This delay ensures reliable results at high clock rates.

3.7.36 NOT fr — Complement of fr into fr

Operation: fr = fr ^ FFh

Flags affected: Z

Opcode: `0010 011f ffff`

Description: This instruction complements each bit of the specified file register and writes the result back into the same register. If the result is 00h, the Z flag is set.

Cycles: 1

Example:
```
not   $11      ;complement file register 11h
```

Suppose that W contains the value 1Ch. This instruction takes the complement of 1Ch and writes the result, E3h, into the same register. The result is nonzero, so the Z flag is cleared.

3.7.37 OR fr,W — OR of fr and W into fr

Operation: fr = fr | W

Flags affected: Z

Opcode: `0001 001f ffff`

Description: This instruction performs a bitwise logical OR of the contents of the specified file register and W, and writes the 8-bit result into the same file register. W is left unchanged. If the result is 00h, the Z flag is set.

Cycles: 1

Example:
```
or    $10,W    ;perform logical OR and overwrite fr
```

This example performs a bitwise logical OR of the working register W with a value stored in file register 10h. The result is written back to the file register 10h.

For example, suppose that the file register 10h is loaded with the value 0Fh and W contains the value 13h. The instruction takes the logical OR of 0Fh and 13h and writes the result, 1Fh, into the same file register. The result is nonzero, so the Z flag is cleared.

3.7.38 OR W,fr — OR of W and fr into W

Operation: W = W | fr

Flags affected: Z

Opcode: `0001 000f ffff`

Description: This instruction performs a bitwise logical OR of the contents of W and the specified file register, and writes the 8-bit result into W. The file register is left unchanged. If the result is 00h, the Z flag is set.

Cycles: 1

Example:

```
or    W,$0B    ;perform logical OR and overwrite W
```

This example performs a bitwise logical OR of the value stored in file register 0Bh with W. The result is written back to W.

For example, suppose that the file register 0Bh is loaded with the value 0Fh and W contains the value 13h. The instruction takes the logical OR of 0Fh and 13h and writes the result, 1Fh, into W. The result is nonzero, so the Z flag is cleared.

3.7.39 OR W,#lit — OR of W and Literal into W

Operation: W = W | lit

Flags affected: Z

Opcode: `1101 kkkk kkkk`

Description: This instruction performs a bitwise logical OR of the contents of W and an 8-bit literal value, and writes the 8-bit result into W. If the result is 00h, the Z flag is set.

Cycles: 1

Example:

```
or    W,#$0F   ;set four low-order bits of W
```

This example performs a bitwise logical OR of W with the literal value #0Fh. The result is written back to W.

For example, suppose that W contains the value 50h. The instruction takes the logical OR of this value with 0Fh and writes the result, 5Fh, into W. The result is nonzero, so the Z flag is cleared.

3.7.40 PAGE addr12 Load Page Number into STATUS(7:5)

Operation: STATUS(7:5) = addr12(11:9)

Flags affected: none

Opcode: `0000 0001 0nnn`

Description: This instruction writes a three-bit value into the PA2:PA0 bits of the STATUS register (bits 7:5). These bits select the program memory page for subsequent “jump” and “call” instructions. The 3-bit value specified in the instruction must be 0, 1, 2, or 3. (The values 4 through 7 are reserved for future expansion. The PA2 bit is not used in this device.)

In the syntax of the assembly language, you specify the page using a full 12-bit program memory address. The assembler encodes the three high-order bits of this address into the instruction opcode and ignores the nine low-order bits. The three high-order bits are written into the PA2:PA0 field of the STATUS register.

Cycles: 1

Example:

```
page  $400     ;set page bits PA2:PA0 to 010 binary
jump  home1    ;jump to address in page 2
```

This example sets the PA2:PA0 bits in the STATUS register to 010. This means that the subsequent “call” instruction calls a subroutine that starts in page 2 of program memory (somewhere in the address range of 400h to 5FFh).

3.7.41 RET Return from Subroutine

Operation: program counter = top-of-stack

Flags affected: none

Opcode: `0000 0000 1100`

Description: This instruction causes a return from a subroutine. It pops the 11-bit value previously stored on the stack and restores that value to the program counter. This causes the program to jump to the instruction immediately following the "call" instruction that called the subroutine.

It is not necessary to set the PA2:PA0 bits in the STATUS register in order to return to the correct place in the program. This is because the full 11-bit program address is restored from the stack. The "ret" instruction does not use (and does not affect) the PA2:PA0 bits. It also does not affect the W register.

If you want to automatically configure the PA1:PA0 bits to select the current page (the page of the instruction following the call instruction), use RETP instead of RET.

Cycles: 2 in "compatible" mode, or 3 in "turbo" mode

Example:

```
page  $000      ;set page of subroutine in STATUS reg.
call  addxy     ;call subroutine addxy
mov   $0C,W     ;use addxy subroutine results
...             ;more of program (not shown)
addxy           ;subroutine address label
mov   W,$0E     ;subroutine instructions start here
add   W,$0F
...
ret             ;return from subroutine
```

The "call" instruction in this example calls a subroutine called "addxy." When the "call" instruction is executed, the address of the following instruction (the "mov $0C,W" instruction) is pushed onto the stack and the program jumps to the "addxy" routine. When the "ret" instruction is executed, the saved program address is popped from the stack and restored to the 11-bit program counter, which causes the program to continue with the instruction immediately following the "call" instruction.

3.7.42 RETI Return from Interrupt

Operation: restore W, STATUS, FSR, and program counter from shadow registers

Flags affected: STATUS register restored, which affects all flags

Opcode: `0000 0000 1110`

Description: This instruction causes a return from an interrupt service routine. It restores the 11-bit program counter value that was saved when the interrupt occurred. This causes the program to return to the point in the program where the interrupt occurred. The instruction also restores the contents of W, STATUS, and FSR registers that were saved when the interrupt occurred.

Cycles: 2 in “compatible” mode, or 3 in “turbo” mode

Example:

```
org   0         ;interrupt routine at address 000h
mov   M,#$09    ;set up MODE register to access WKPND_B
clr   W         ;clear W
mov   !RB,W     ;exchange W and WKPND_B contents
and   W,#$0F    ;mask out unused bits of WKPND_B
mov   $1A,W     ;move pending bits to register 1Ah
...             ;test pending bits perform service
reti            ;return from interrupt
```

This is an example of an interrupt service routine that services interrupts triggered on the RB0, RB1, RB2, and RB3 pins. When an interrupt occurs, the device saves the 11-bit contents of the program counter and the contents of the W, STATUS, and FSR registers into a set of shadow registers. The program then jumps to the interrupt service routine, which starts at address 000h. The interrupt service routine determines the cause of the interrupt, clears the applicable interrupt pending bit, performs the required task, and ends with the “reti” instruction.

The “reti” instruction restores the contents of the program counter and the W STATUS, and FSR registers. This causes the device to continue program execution at the point where the program was interrupted.

3.7.43 RETIW Return from Interrupt and Adjust RTCC with W

Operation: RTCC = RTCC + W

restore W, STATUS, FSR, and program counter from shadow registers

Flags affected: STATUS register restored, which affects all flags

Opcode: `0000 0000 1111`

Description: Like the RETI instruction, the RETIW instruction causes a return from an interrupt service routine. It restores the 11-bit program counter value that was saved when the interrupt occurred. This causes the program to return to the point in the program where the interrupt occurred.

Before it returns from the interrupt service routine, the RETIW instruction first adds W to the Real-Time Clock Counter (RTCC). Then it restores the contents of the W, STATUS, and FSR registers and the program counter that were saved when the interrupt occurred.

Adding W to RTCC allows the interrupt service routine to restore the RTCC to the value it contained at the time the main program was interrupted. To use this feature, the interrupt service routine should check the RTCC at the beginning of the routine and again at the end of the routine, and then put the adjustment value into W before returning from the interrupt.

Cycles: 2 in “compatible” mode, or 3 in “turbo” mode

Example:

```
...     ;interrupt service routine at address 000h
...     ;check RTCC
...     ;check interrupt pending bits
...     ;perform interrupt service
...
...     ;check RTCC
...     ;put adjustment value into W
retiw   ;return from interrupt and adjust RTCC
```


3.7.44 RETP Return from Subroutine Across Page Boundary

Operation: STATUS(PA1:PA0) = top-of-stack (10:9)
program counter = top-of-stack

Flags affected: none

Opcode: `0000 0000 1101`

Description: Like the RET instruction, the RETP instruction causes a return from a subroutine. It pops the 11-bit value previously stored on the stack and restores that value to the program counter. This causes the program to jump to the instruction immediately following the "call" instruction that called the subroutine.

Unlike the RET instruction, the RETP instruction also writes bit 10 and bit 9 of the return address (the address of the instruction immediately following the "call" instruction) into the PA1:PA0 bits of the STATUS register. This automatically configures the PA1:PA0 bits to select the current page, allowing a subsequent same-page jump or call to be executed without using another "page" instruction.

Cycles: 2 in "compatible" mode, or 3 in "turbo" mode

Example:

```
org   $050     ;start of program in page 0
...
page  $200     ;set PA1:PA0 bits to 01 (different page)
call  subxy    ;call subroutine in different page
...
call  ddxy     ;call subroutine in same page
...
addxy          ;subroutine in same page as call
...
ret
...
org   $200     ;new memory segment at 200h
subxy          ;subroutine address label at 200h
...
retp           ;return from subroutine (different page)
```

The first call crosses a 512-word page boundary (PA1:PA0 = 01). Upon return from that subroutine, the PA1:PA0 bits are automatically returned to their original values (PA1:PA0 = 00), allowing a subsequent same-page call to be done without using the "page" instruction again.

3.7.45 RETW lit — Return from Subroutine with Literal in W

Operation: W = lit

program counter = top-of-stack

Flags affected: none

Opcode: `1000 kkkk kkkk`

Description: This instruction causes a return from a subroutine and also puts an 8-bit literal value into W. It pops the 11-bit value previously stored on the stack and loads that value into the program counter. This causes the program to jump to the instruction immediately following the "call" instruction that called the subroutine.

You can use multiple "RETW lit" instructions to implement a data lookup table.

Cycles: 2 in "compatible" mode, or 3 in "turbo" mode

Example:

```
mov   W,$0A    ;load W with value to be squared (0-7)
call  square   ;call lookup-table subroutine
mov   $0B,W    ;use subroutine results (in W)
...            ;more of program (not shown)
square         ;subroutine entry point
and W,#$07     ;ensure that W is less than 8
add   $02,W    ;add W to PC to jump to applicable retw
retw  0        ;0 squared = 0, beginning of data table
retw  1        ;1 squared = 1
retw  4        ;2 squared = 4
retw  9        ;3 squared = 9
retw  16       ;4 squared = 16
etw   25       ;5 squared = 25
retw  6        ;6 squared = 36
retw  9        ;7 squared = 49, end of data table
```

The "square" subroutine calculates the square of W and returns the result in W. To use the subroutine, the program first loads W with the value to be squared, which must be a value from 0 to 7. The subroutine adds the contents of W to the program counter, which advances the program to the applicable "RETW lit" instruction. The "RETW lit" instruction returns from the subroutine with the appropriate result in W.

3.7.46 RL fr — Rotate fr Left through Carry

Operation: fr = << fr

Flags affected: C

Opcode: `0011 011f ffff`

Description: This instruction rotates the bits of the specified file register left using the C flag bit. The bits inside the register are shifted left by one bit position. C is shifted into the least significant bit position and the most significant bit is shifted out into C, as shown in the diagram below.

Cycles: 1

Example 1:

```
rl $18
```

This example rotates the bits of file register 18h. If the register initially contains 14h and the C flag is set to 1, after executing this instruction, the register will contain 29h and the C flag will be cleared to 0.

Example 2:

```
clrb  $03.0    ;clear carry flag
rl    $18      ;rotate left, reg=reg*2
rl    $18      ;rotate left, reg=reg*2
```

This example multiplies file register 18h by 4. The initial “clrb” instruction clears the C flag, which ensures that 0 will be shifted into the least significant bit position. The two “rl” instructions perform two successive multiply-by-2 operations.

3.7.47 RR fr — Rotate fr Right through Carry

Operation: fr = >> fr

Flags affected: C

Opcode: `0011 001f ffff`

Description: This instruction rotates the bits of the specified file register right using the C flag bit. The bits inside the register are shifted right by one bit position. C is shifted into the most significant bit position and the least significant bit is shifted out into C, as shown in the diagram below.

Cycles: 1

Example 1:

```
rr $0F
```

This example rotates the bits of file register 0Fh. If the register initially contains 12h and the C flag is set to 1, after executing this instruction, the register will contain 89h and the C flag will be cleared to 0.

Example 2:

```
clrb  $03.0    ;clear carry flag
rr    $0F      ;rotate right, reg=reg/2
clrb  $03.0    ;clear carry flag
rr    $0F      ;rotate right, reg=reg/2
```

This example divides file register 0Fh by 4. The “clrb” instructions ensure that 0 will be shifted into the most significant bit positions. The two “rr” instructions perform two divide-by-2 operations.

3.7.48 SB fr,bit Test Bit in fr and Skip if Set

Operation: Test a specified bit in fr; if 1, skip next instruction

Flags affected: none

Opcode: `0111 bbbf ffff`

Description: This instruction tests a bit in the specified file register. The file register address (00h through 1Fh) and the bit number (0 through 7) are the instruction operands. If the bit is 1, the next instruction in the program is skipped. Otherwise, program execution proceeds normally with the next instruction.

Cycles: 1 if tested condition is false, 2 if tested condition is true

Example:

```
sb    $1F.7    ;test bit 7 of file register
inc   $1F      ;increment if bit=0
mv    W,$1F    ;move file register to W
```

This example tests the most significant bit of file register 1Fh. If that bit is 1, the "inc" instruction is skipped. Otherwise, program execution proceeds normally with the "inc" instruction.

3.7.49 SETB fr,bit — Set Bit in fr

Operation: Set a specified bit in fr

Flags affected: none

Opcode: `0101 bbbf ffff`

Description: This instruction sets a bit in the specified file register to 1 without changing the other bits in the register. The file register address (00h through 1Fh) and the bit number (0 through 7) are the instruction operands.

Cycles: 1

Example:
```
setb  $1F,7
```
This example sets the most significant bit of file register 1Fh.

3.7.50 SLEEP Power Down Mode

Operation: WDT = 00h

STATUS(TO) = 1, STATUS(PD) = 0

stop oscillator

Flags affected: none

Opcode: `0000 0000 0011`

Description: This instruction places the device in the power down mode. If the Watchdog timer is enabled, the WDT register is cleared, the TO (timeout) bit in the STATUS register is set to 1, and the PD (power down) bit in the STATUS register is cleared to 0.

There are three types of events that can cause an exit from the power down mode: a Watchdog timer overflow, a transition on a Multi-Input Wakeup pin, or an external reset on the $\overline{\text{MCLR}}$ pin.

Cycles: 1

Example:
```
sleep
```

This example puts the device into the power down mode until a wakeup event occurs.

3.7.51 SNB fr,bit Test Bit in fr and Skip if Clear

Operation: Test a specified bit in fr; if 0, skip next instruction

Flags affected: none

Opcode: `0110 bbbf ffff`

Description: This instruction tests a bit in the specified file register. The file register address (00h through 1Fh) and the bit number (0 through 7) are the instruction operands. If the bit is 0, the next instruction in the program is skipped. Otherwise, program execution proceeds normally with the next instruction.

Cycles: 1 if tested condition is false, 2 if tested condition is true

Example:

```
snb   $1F,5    ;test bit 5 of file register
dec   $1F      ;decrement if bit=1
mov   W,$1F    ;move file register to W
```

This example tests bit number 5 of file register 1Fh. If that bit is 0, the "dec" instruction is skipped. Otherwise, program execution proceeds normally with the "dec" instruction.

3.7.52 SUB fr,W — Subtract W from fr

Operation: fr = fr - W

Flags affected: C, DC, Z

Opcode: `0000 101f ffff`

Description: This instruction subtracts the contents of W from the contents of the specified file register and writes the 8-bit result into the same file register. W is left unchanged. The register contents are treated as unsigned values.

If the result of subtraction is negative (W is larger than fr), the C flag is cleared to 0 and the lower eight bits of the result are written to the file register. Otherwise, the C flag is set to 1.

If there is a borrow from bit 3 to bit 4, the DC (digit carry) flag is cleared to 0. Otherwise, the flag is set to 1.

If the result of subtraction is 00h, the Z flag is set. Otherwise, the flag is cleared.

Config. Option: If the $\overline{CF}$ bit in the FUSEX configuration register has been programmed to 0, this instruction also subtracts the complement of the C flag as a borrow-in input:

fr = fr - W - /C

See Example 2 below for a program example of multiple-byte subtraction with borrow.

Cycles: 1

Example 1:
```
sub   $0D,W
```

This example subtracts the contents of W from file register 0Dh. For example, if the file register contains 35h and W contains 06h, this instruction subtracts 06h from 35h and writes the result, 2Fh, into the file register. It also sets the C flag, clears the DC flag, and clears the Z flag.

Example 2:
```
set $03.0  ;set carry flag for no borrow in
mov W,$0A  ;load W from 0Ah (low-order byte)
sub $0C,W  ;low-order subtraction, C=0 for borrow out
mov W,$0B  ;load W from 0Bh (high-order byte)
sub $0D,W  ;high-order subtraction, borrow in & out
```

This example performs 16-bit subtraction of file registers 0Ah-0Bh from file registers 0Ch-0Dh. For this example, the $\overline{CF}$ bit in the FUSEX configuration register must be programmed to 0 in order to implement subtraction with borrow.

The first "sub" instruction subtracts the contents of 0Ah from 0Ch and clears the C flag if a borrow occurs out of bit 7, or sets the C flag otherwise. The second "sub" instruction subtracts the contents of 0Bh from 0Dh with borrow-in using the C flag.

This algorithm can also be implemented with the device in the default configuration (with the $\overline{CF}$ bit set to 1 in the FUSEX register), although not as efficiently. For example, you can do the low-order subtraction, test the carry bit, decrement file register 0Dh if the carry flag is 0, and then do the high-order subtraction.

3.7.53 SWAP Swap High/Low Nibbles of fr

Operation: fr = <> fr

Flags affected: none

Opcode: `0011 101f ffff`

Description: This instruction exchanges the high-order and low-order nibbles (4-bit segments) of the specified file register.

Cycles: 1

Example:
```
swap  $0B
```

This example swaps the high-order and low-order nibbles of file register 0Bh. For example, if the register contains A5h, after executing this instruction, the register will contain 5Ah.

3.7.54 TEST fr — Test fr for Zero

Operation: fr = fr

Flags affected: Z

Opcode: `0010 001f ffff`

Description: This instruction moves the contents of the specified file register into the same register. There is no net effect except to set or clear the Z flag. If the register contains 00h, the flag is set. Otherwise, the flag is cleared.

Cycles: 1

Example:

```
test  $1B      ;test file register 1Bh
sb    STATUS.2 ;test Z bit and skip if set
inc   $1B      ;increment file reg 1Bh if nonzero
mov   W,$1B    ;move file reg 1Bh to W
```

This example tests the contents of file register 1Bh. The “test” instruction sets or clears the Z flag based on the contents of the file register. The “sb” instruction tests the Z flag. The “inc” instruction is executed if the file register contains zero or is skipped if the file register contains a nonzero value.

3.7.55 XOR fr,W — XOR of fr and W into fr

Operation: fr = fr ^ W

Flags affected: Z

Opcode: `0001 101f ffff`

Description: This instruction performs a bitwise exclusive OR of the contents of the specified file register and W, and writes the 8-bit result into the same file register. W is left unchanged. If the result is 00h, the Z flag is set.

Cycles: 1

Example:
```
xor   $10,W    ;perform logical XOR and overwrite fr
```

This example performs a bitwise logical XOR of the working register W with a value stored in file register 10h. The result is written back to the file register 10h.

For example, suppose that the file register 10h is loaded with the value 0Fh and W contains the value 13h. The instruction takes the logical XOR of 0Fh and 13h and writes the result, 1Ch, into the same file register. The result is nonzero, so the Z flag is cleared.

3.7.56 XOR W,fr XOR of W and fr into W

Operation: W = W ^ fr

Flags affected: Z

Opcode: `0001 100f ffff`

Description: This instruction performs a bitwise exclusive OR of the contents of W and the specified file register, and writes the 8-bit result into W. The file register is left unchanged. If the result is 00h, the Z flag is set.

Cycles: 1

Example:
```
xor   W,$0B    ;perform logical XOR and overwrite W
```

This example performs a bitwise logical XOR of the value stored in file register 0Bh with W. The result is written back to W.

For example, suppose that the file register 0Bh is loaded with the value 0Fh and W contains the value 13h. The instruction takes the logical XOR of 0Fh and 13h and writes the result, 1Ch, into W. The result is nonzero, so the Z flag is cleared.

3.7.57 XOR W,#lit XOR of W and Literal into W

Operation: W = W ^ lit

Flags affected: Z

Opcode: `1111 kkkk kkkk`

Description: This instruction performs a bitwise exclusive OR of the contents of W and an 8-bit literal value, and writes the 8-bit result into W. If the result is 00h, the Z flag is set.

Cycles: 1

Example:
```
xor   W,$#0F   ;complement four low-order bits of W
```

This example performs a bitwise logical XOR of W with the literal value #0Fh. The result is written back to W.

For example, suppose that W contains the value 51h. The instruction takes the logical XOR of this value with 0Fh and writes the result, 5Eh, into W. The result is nonzero, so the Z flag is cleared.

Chapter 4

Clocking, Power Down, and Reset

4.1 Introduction

The SX device can be configured to operate in any one of several clocking modes. You can use the built-in oscillator, an external oscillator circuit, or an external clock signal to drive the device. Each type of clock has its advantages and disadvantages with respect to clock rate choices, rate accuracy, and cost.

The SX device supports a "power down" mode, which reduces power consumption to a very low level during periods of inactivity. This mode is invoked by executing the "sleep" instruction. During power down, the device is completely inactive (except for the Watchdog timer, if enabled). Upon "wakeup" from the power down mode, the device is reset.

A reset occurs for any of the following conditions: initial power-up, wakeup from the power down mode, brown-out, Watchdog timeout, or assertion of the $\overline{\text{MCLR}}$ input signal (Master Clear Reset). When a reset occurs, the program counter is initialized to the last program address (7FFh), where the application program should have a "jump" instruction to its initialization routine.

4.2 Clocking Options

You can configure the SX device to use an on-chip RC oscillator, an external RC oscillator, an external crystal/resonator, or an externally generated clock signal. This choice depends on the required speed and precision of the clock, as well as cost considerations.

There are two device pins used for clocking, called OSC1 and OSC2. The functions of these pins depend on the device configuration and the chosen clocking mode.

You select the desired clocking mode by programming the FUSE word register, a 12-bit register mapped into the program memory at address FFFh. This register is accessible only when you are programming the instruction memory of the device, not a run time. For information on the specific bit fields in the register and the corresponding clocking modes, see Section 2.8.

4.2.1 Clock/Instruction Rate Option (Compatible or Turbo Mode)

When you select the clock type, you need to consider the clock/instruction rate option. This option lets you select one of two instruction clocking modes, called the "compatible" mode and the "turbo" mode.

In the "compatible" mode, the instruction rate is one-fourth of the clock rate. In this configuration, you need to select a clock rate four times higher than the intended instruction rate. For example, if you want

to execute instructions at a rate of 1 MHz (one instruction per microsecond), you need to select a clock rate of 4 MHz. This mode is designed for compatibility with the PIC16C5x series of microcontroller devices.

In the "turbo" mode, the instruction rate is equal to the clock rate. For example, if you want to execute instructions at a rate of 50 MHz (one instruction per 20 nanoseconds), you use a 50 MHz clock. This is the preferred operating mode for new designs because you can use a slower clock to achieve a given instruction rate, thus reducing electromagnetic interference (EMI) in the system and the cost of the oscillator.

4.2.2 Internal RC Oscillator

Using the on-chip, built-in RC (resistor-capacitor) oscillator for the device clock is the lowest-cost option because no external components are required. This mode is suitable for lower-speed applications (4 MHz or less) where high accuracy is not needed. For this mode, you leave the OSC1 and OSC2 pins unconnected.

The internal RC oscillator operates at a nominal rate of 4 MHz and has an accuracy of plus or minus 8% over the allowed temperature range. The device can be configured to divide this clock down to produce a lower-rate clock for device operation, with the divide-by factor set to of any power-of-2 from 2 to 128. This selection is made by programming the DIV2:DIV0 bits in the FUSE word as follows:

000 for 4 MHz operation
001 for 2 MHz operation
010 for 1 MHz operation
011 for 500 kHz operation
100 for 250 kHz operation
101 for 125 kHz operation
110 for 62.5 kHz operation
111 for 31.25 kHz operation

4.2.3 External RC Oscillator

Using an external RC oscillator network is a low-cost option suitable for applications that do not require high precision. The only external components required are a resistor and a capacitor. Unlike the internal RC oscillator, you can choose any operating frequency for which the device is rated, not just certain frequencies between 31.25 kHz and 4 MHz.

The RC oscillator operating rate is a function of the resistor and capacitor values, the supply voltage, and the operating temperature. The operating rate will vary from unit to unit due to normal variations in component values, and from time to time due to fluctuations in temperature and voltage. Therefore, an application that requires high precision (for example, a system with a real-time clock) should use an external resonator or crystal rather than an RC oscillator.

Figure 4-1 shows how the resistor and capacitor are connected to the device. The operating frequency can be adjusted by choosing the values for R and C.

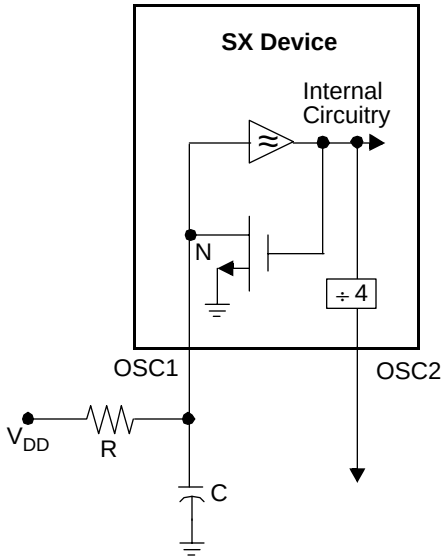

Figure 4-1 External RC Oscillator Connections

In this operating mode, OSC1 is the clock input and OSC2 operates as a clock output. The input signal is amplified and fed back to the pin through a transistor to stabilize the clock signal. The output clock signal is created by dividing the internal clock by 4. For example, if the RC clock operates at 8 MHz, the OSC2 output clock will operate at 2 MHz.

A resistor value between 3 kΩ and 100 kΩ is recommended. For resistor values below this range, the oscillator might become unstable or stop completely. For resistor values higher than this range, the oscillator becomes sensitive to noise, humidity, and capacitor leakage.

Although the device will operate without a capacitor (C = 0 pF), a capacitor of at least 20 pF is recommended for noise immunity and stability. For capacitance values lower than this, the oscillator frequency can vary significantly due to reliance on the small parasitic capacitance associated with the PCB traces and device package.

4.2.4 External Crystal/Resonator (XT, LP, or HSMode)

Using an external crystal or ceramic resonator to generate the clock is suitable for a system that requires precise and accurate timing (for example, for a real-time clock). These types of oscillators cost more than RC oscillators.

The SX device can be configured to operate in any one of the following external crystal/resonator modes:

- LP (Low Power Crystal)
- XT (Crystal/Resonator)
- HS (High-Speed Crystal/Resonator)

With the SX device configured in one of these modes, the crystal or ceramic resonator is connected to the OSC1 and OSC2 pins as shown in Figure 4-2.

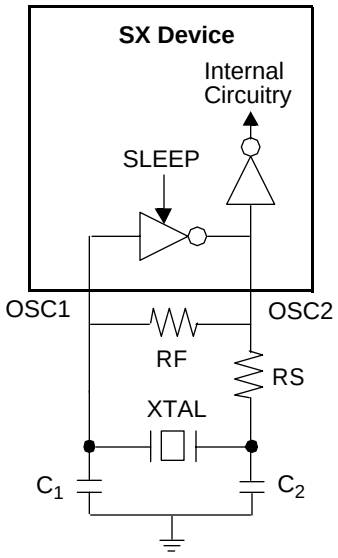

Figure 4-2 Crystal or Ceramic Resonator Connections

For clock frequencies above 1 MHz, the series resistor R_S is not needed; use a direct connection instead. For frequencies at or below 1 MHz, use R_S = 10 MΩ.

If you use a crystal, a parallel resonant type crystal is recommended. Using a series resonant type crystal may result in a frequency that is outside of the crystal manufacturer's recommended range.

Table 4-1 shows the recommended operating modes and component values to be used with ceramic resonators at various clock frequencies. Table 4-2 shows the same information for crystal oscillators. Note that the LP (low-power) mode is not shown in the tables because it is used only at lower frequencies, in the range of 32 kHz to 100 kHz. For operation at these lower frequencies, contact Scenix Semiconductor for more information.

Table 4-1 Clock Modes and Component Values (Murata Ceramic Resonators)

Clock Mode	Resonator Frequency	C1	C2	R_F	R_S
XT	455 kHz	220 pF	220 pF	1 MΩ	6.8 kΩ
XT	1 MHz	100 pF	100 pF	1 MΩ	6.8 kΩ
XT	2 MHz	100 pF	100 pF	100 kΩ	680 Ω
HS	4 MHz	100 pF	100 pF	100 kΩ	0
HS	4 MHz	Internal (47 pF)	Internal (47 pF)	100 kΩ	470 Ω
HS	8 MHz	30 pF	30 pF	1 MΩ	0
HS	8 MHz	Internal (47 pF)	Internal (47 pF)	1 MΩ	470 Ω
HS	12 MHz	30 pF	30 pF	1 MΩ	0
HS	12 MHz	Internal (22 pF)	Internal (22 pF)	1 MΩ	0
HS	16 MHz	15 pF	15 pF	1 MΩ	0
HS	16 MHz	Internal (15 pF)	Internal (15 pF)	1 MΩ	0
HS	20 MHz	10 pF	10 pF	1 MΩ	0
HS	33 MHz	10 pF	10 pF	33 kΩ	0
HS	50 MHz	Internal (5 pF)	Internal (5 pF)	33 kΩ	0

Table 4-2 Clock Modes and Component Values (Crystal Oscillators)

Clock Mode	Resonator Frequency	C1	C2	R_F
XT	4 MHz	20 pF	47 pF	1 MΩ
HS	8 MHz	20 pF	47 pF	1 MΩ
HS	12 MHz	20 pF	47 pF	1 MΩ
HS	16 MHz	15 pF	30 pF	1 MΩ
HS	20 MHz	15 pF	30 pF	1 MΩ
HS	25 MHz	5 pF	20 pF	10 kΩ
HS	30 MHz	5 pF	20 pF	4.7 kΩ
HS	36 MHz	5 pF	15 pF	3.3 kΩ
HS	40 MHz	5 pF	15 pF	3.3 kΩ
HS	50 MHz	5 pF	10 pF	3.3 kΩ

RS = 0

4.2.5 External Clock Signal

You can use an externally generated clock signal to drive the SX device. This mode is suitable for systems in which there is already a clock signal available (used to drive other chips in the system) that can also be used to drive the SX device. The clock signal must meet the clock specifications of the SX device, including the duty cycle, rise time, fall time, and voltage levels.

To use this mode, configure the device to operate in the XT, LP, or HS mode. It does not matter which one of these modes you select. Then connect the clock signal to the OSC1 input and leave the OSC2 pin unconnected, as shown in Figure 4-3.

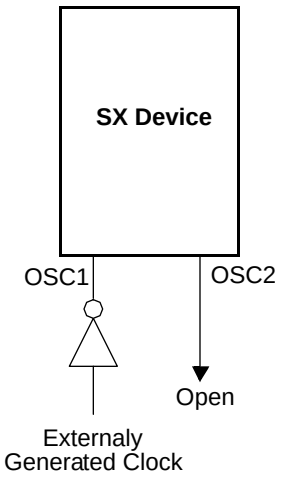

Figure 4-3 External Clock Signal Connection

4.3 Power Down Mode

In the SX power down mode, the device is shut down and the clock is stopped to all parts of the device. The Watchdog timer, if enabled, continues to operate because it uses its own independent on-chip oscillator. Upon wakeup from the power down mode, the device is reset and the program jumps to the last program address (7FFh).

4.3.1 Entering the Power Down Mode

For the lowest possible power consumption in the power down state, disable the Watchdog timer. This eliminates the power consumption of the Watchdog oscillator and counter. In that case, however, you will not be able to use a Watchdog timeout to wake up the device.

The device enters the power down mode upon execution of the “sleep” instruction. Program execution stops and the device is powered down until a wakeup event occurs.

If the Watchdog timer is enabled, the “sleep” instruction sets the TO (Watchdog Timeout) bit to 1 and clears the PD (Power Down) bit to 0 in the STATUS register. The Watchdog timer continues to operate

while the device is powered down. AWatchdog timeout will then wake up the device from the power down state.

4.3.2 Waking Up from the Power Down Mode

Any one of the following events will cause a wakeup from the power down state:

- a timeout signal from the Watchdog timer, generated when the Watchdog timer overflows
- a valid transition on any Port B pin configured to operate as a Multi-Input Wakeup pin
- a low voltage on the $\overline{\text{MCLR}}$ input pin (Master Clear Reset)

When a wakeup event occurs, the TO and PD bits are both set to 1 in the STATUS register, and program execution resumes at the highest program memory address (7FFh), just like an ordinary reset operation. The highest program memory address should contain an instruction to jump to the device initialization routine.

4.4 Multi-Input Wakeup/Interrupt

The Multi-Input Wakeup circuit allows the Port B pins to be used as device inputs to wake up the device from the power down state, or to trigger an interrupt from an external source.

The same Multi-InputWakeup circuit is used for both wakeups and interrupts. In the power down state, a wakeup signal on a Port B pin wakes up and resets the device, causing the program to jump to the highest program memory address (7FFh). The same signal received on a Port B pin during normal operation triggers an interrupt, which causes the device to save the program context (program counter, W, STATUS, and FSR) and then jump to the lowest memory address (000h). For more information on interrupts, see Chapter 6.

4.4.1 Port B Configuration for Multi-Input Wakeup/Interrupt

Figure 4-4 is a block diagram of the Multi-Input Wakeup circuit. The circuit uses the I/O pins of Port B for the wakeup inputs. Port B must be properly configured for Multi-Input Wakeup operation. The eight Port B pins can be individually configured for this purpose. You control the port configuration by writing to its configuration registers using the “mov !RB,W” instruction. Selection of those registers is controlled by the MODE register.

These are the configuration registers that you must program in order to prepare Port B pins for Multi-Input Wakeup/Interrupt operation:

- RB Data Direction register
- WKEN_B (Port B Wakeup Enable register)
- WKED_B (Port B Wakeup Edge Select register)
- WKPND_B (Port B Wakeup Pending Flag register)

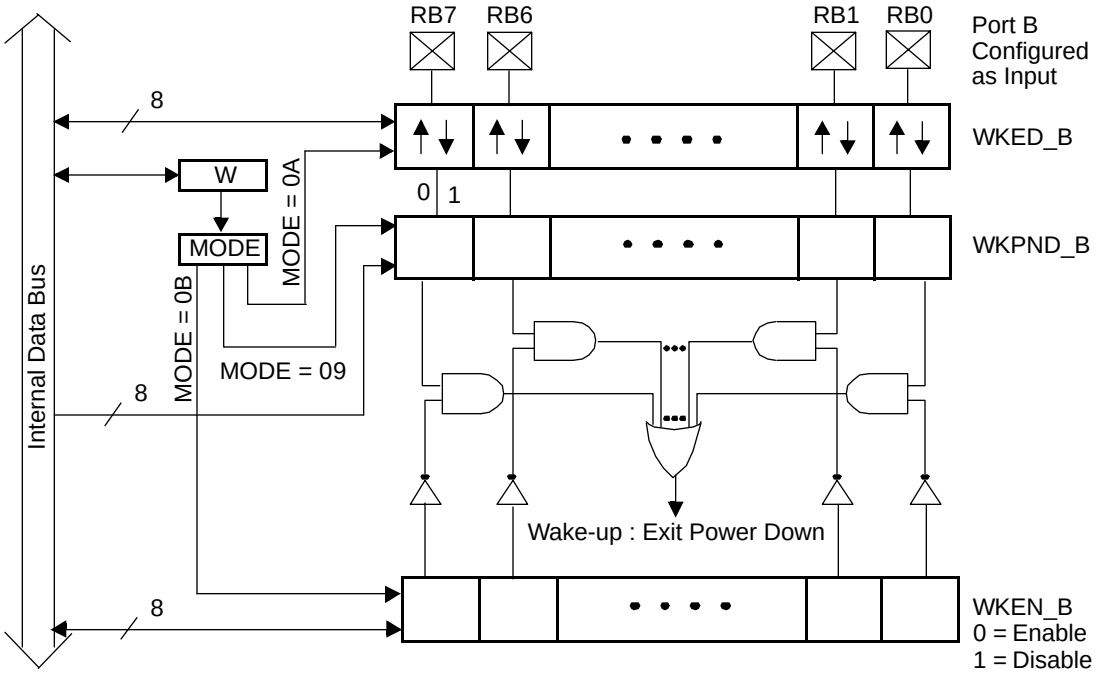

Figure 4-4 Multi-Input Wakeup/Interrupt Block Diagram

To make a Port B pin operate as a high-impedance input (not as an output), set the corresponding bit to 1 in the RB data direction register.

To enable a Port B pin to operate as a Multi-Input Wakeup input, clear the corresponding bit to 0 in the WKEN_B register

To specify the edge sensitivity of the pin, set or clear the corresponding bit in the WKED_B register. Set the bit to 1 to sense falling edges (high-to-low transitions) or clear the bit to 0 to sense rising edges (low-to-high transitions). An edge of the specified type on the wakeup-enabled pin will trigger a wakeup or interrupt.

The WKPND_B register contains flag bits that indicate occurrences of wakeup/interrupt events on the Port B pins. When a valid edge is received on a wakeup-enabled pin, it sets the corresponding flag bit is set to 1 in the WKPND_B register and triggers the wakeup or interrupt. The program can read the WKPND_B register to determine which Port B pin received the wakeup/interrupt signal.

Upon power-up or reset, the WKPND_B register contains unknown data. Therefore, the program should clear this register to zero before it enables the Multi-Input Wakeup function in the WKEN_B register. Otherwise, the program will not be able to determine which pin received the wakeup signal.

Upon power-up or reset, the WKEN_B register is set to FFh. This disables the wakeup interrupts by default. You must explicitly enable any pins that you want to use as wakeup/interrupt pins.

Here is an example of a program segment that configures the RB0, RB1, and RB2 pins to operate as Multi-InputWakeup/Interrupt pins, sensitive to falling edges:

```
mov  M,#$09    ;prepare to access WKPND_B (pending) register
mov  W,#$00    ;clear W
mov  !RB,W     ;clear all wakeup pending flags

mov  M,#$0F    ;prepare to write port data direction registers
mov  W,#$07    ;load W with the value 07h
mov  !RB,W     ;configure RB0-RB2 to be inputs

mov  M,#$0A    ;prepare to write WKED_B (edge) register
               ;W contains the value 07h
mov  !RB,W     ;configure RB0-RB2 to sense falling edges

mov  M,#$0B    ;prepare to write WKEN_B (enable) register
mov  W,#$F8h   ;load W with the value F8h
mov  !RB,W     ;enable RB0-RB2 to operate as wakeup inputs
```

To prevent false interrupts, the enabling step (clearing bits in WKEN_B) should be done as the last step in a sequence of Port B configuration steps.

After this program segment is executed, the device can receive interrupts on the RB0, RB1, and RB2 pins. If the device is put into the power down mode (by executing a “sleep” instruction), the device can then receive wakeup signals on those same pins.

4.4.2 Reading the Wakeup Pending Bits

The interrupt service routine or initialization code can determine which pin received the wakeup signal by reading the WKPND_B register, as in the following example:

```
mov  M,#$09    ;set MODE register to access WKPND_B
mov  W,#$00    ;clear W
mov  !RB,W     ;exchange contents of W and WKPND_B
```

When the MODE register is set to provide access to WKPND_B or CMP_B, the instruction “mov !RB,W” performs an exchange between the contents of W and the port control register, rather than a simple move from W to the port control register. This feature provides a way for the program to read the WKPND_B and CMP_B registers. In the example above, the “mov !RB,W” instruction simultaneously loads W with the current WKPND_B pending flags and clears the WKPND_B register. Then the program can test the bits in W to determine which Port B pin caused the wakeup or interrupt event.

4.5 Reset

A reset operation puts the SX device into a known initial state. A reset occurs upon any one of the following conditions:

- initial power-up
- wakeup from the power down mode

- recovery from brown-out, as determined by the brown-out detection circuit
- Watchdog timeout
- assertion of the $\overline{\text{MCLR}}$ input signal (Master Clear Reset)

When a reset occurs, the program counter is initialized to the last program address (7FFh), where the application program should have a “jump” instruction to its initialization routine.

Figure 4-5 shows the internal logic of the SX reset circuit. This circuit senses the voltage supply on the V_{DD} pin, the state of the $\overline{\text{MCLR}}$ (Master Clear Reset) input pin, the output of the on-chip RC oscillator, and signals from the Multi-Input Wakeup circuit and Watchdog timer. Based on these inputs, the circuit generates a chip-internal $\overline{\text{RESET}}$ signal. This signal goes low to put the device into the reset state and then goes high to allow the device to begin operating from a known state.

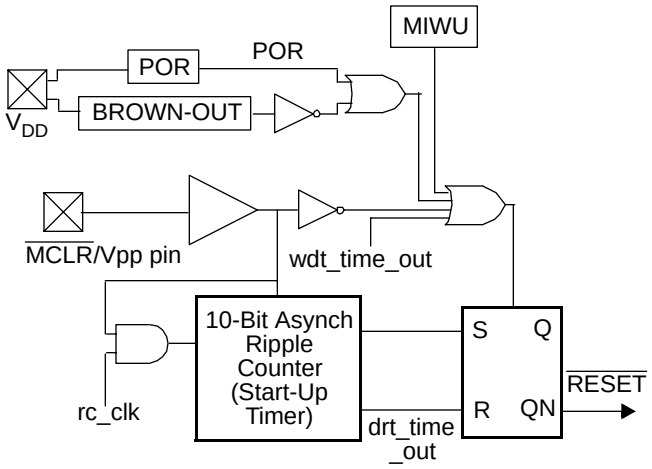

Figure 4-5 On-Chip Reset Circuit Block Diagram

4.5.1 Register States Upon Reset

The effect of a reset operation on a register depends on the register and the type of reset operation. Some registers are initialized to specific values, some are left unchanged (for wakeup and brown-out resets), and some are initialized to an unknown value. A register that starts with an unknown value should be initialized by the software to a known value; you cannot simply test the initial state and rely on it starting in that state consistently.

Table 4-3 lists the SX registers and shows the state of each register upon reset. The column on the left lists the register names, and the first row shows the various types of reset operations. Each entry in the table shows the state of the register just after the applicable reset operation.

Table 4-3 Register States Upon Reset

Register	Power-On	Wakeup	Brown-out	Watchdog Timeout	$\overline{\text{MCLR}}$
W	Undefined	Unchanged	Undefined	Unchanged	Unchanged
OPTION	FFh	FFh	FFh	FFh	FFh
MODE	0Fh	0Fh	0Fh	0Fh	0Fh
RTCC (01h)	Undefined	Unchanged	Undefined	Unchanged	Unchanged
PC (02h)	0Fh	0Fh	0Fh	0Fh	0Fh
STATUS (03h)	Bits 0-2: Undefined Bits 3-4: 1 Bits 5-7: 0	Bits 0-2: Undefined Bits 3-4: Unch. Bits 5-7: 0	Bits 0-4: Undefined Bits 5-7: 0	Bits 0-2: Undefined Bits 3-4: (Note 1) Bits 5-7: 0	Bits 0-2: Undefined Bits 3-4: (Note 2) Bits 5-7: 0
FSR (04h)	Undefined	Bits 0-6: Undefined Bit 7: 1	Bits 0-6: Undefined Bit 7: 1	Bits 0-6: Undefined Bit 7: 1	Bits 0-6: Undefined Bit 7: 1
RA/RB/RC Direction	FFh	FFh	FFh	FFh	FFh
RA/RB/RC Data	Undefined	Unchanged	Undefined	Unchanged	Unchanged
Other File Registers - SRAM	Undefined	Unchanged	Undefined	Unchanged	Unchanged
CMP_B	Bits 0, 6-7: 1 Bits 1-5: Undefined	Bits 0, 6-7: 1 Bits 1-5: Undefined	Bits 0, 6-7: 1 Bits 1-5: Undefined	Bits 0, 6-7: 1 Bits 1-5: Undefined	Bits 0, 6-7: 1 Bits 1-5: Undefined
WKPND_B	FFh	Unchanged	Undefined	Unchanged	Unchanged
WKED_B	FFh	FFh	FFh	FFh	FFh
WKEN_B	FFh	FFh	FFh	FFh	FFh
ST_B/ST_C	FFh	FFh	FFh	FFh	FFh
LVL_A/LVL_B/LVL_C	FFh	FFh	FFh	FFh	FFh
PLP_A/PLP_B/PLP_C	FFh	FFh	FFh	FFh	FFh
Watchdog Counter	Undefined	Unchanged	Undefined	Unchanged	Unchanged

NOTE: 1. Watchdog reset during SLEEP mode: 00
Watchdog reset during Active mode: 01

NOTE: 2. External reset during SLEEP mode: 10
External reset during Active mode: Unchang

4.5.2 Power-On Reset

In a typical power-on situation, the supply voltage takes a known (approximate) amount of time to rise from zero volts to the final operating voltage. The SX device has an on-chip power-on reset circuit that holds the device in the reset state until the supply voltage rises to a stable operating level, thus ensuring reliable operation upon power-up.

The power-on reset circuit uses an asynchronous ripple counter to hold the device in the reset state for a period of time as the supply voltage rises. This counter, called the Delay Reset Timer (DRT), provides the device start-up delay. It is used only for a power-on reset, not for a reset caused by another event such as a wakeup from the power down mode or a brown-out.

Upon power-up, the internal reset latch is set, which asserts the internal $\overline{\text{RESET}}$ signal and holds the device in the reset state. The DRT counts clock pulses generated by the on-chip RC oscillator. It starts counting when the RC oscillator starts working and a valid logic high signal is detected on the $\overline{\text{MCLR}}$ input pin. When the DRT reaches the end of its timeout period (typically 72 msec), it clears the internal reset latch, which releases the device from the reset state.

The $\overline{\text{MCLR}}$ (Master Clear Reset) input pin must be held low upon power-up of the device. If you do not need to use the $\overline{\text{MCLR}}$ pin as a hardware reset input, you can simply tie it together with the V_{DD} power supply pin. This will work reliably only if the power supply rise time is significantly less than the DRT delay of 72 msec.

Figure 4-6 shows the power-on reset timing in this situation. The supply voltage and $\overline{\text{MCLR}}$ pin voltages rise together, and the DRT counter allows the device to begin operating after a delay of about 72 msec.

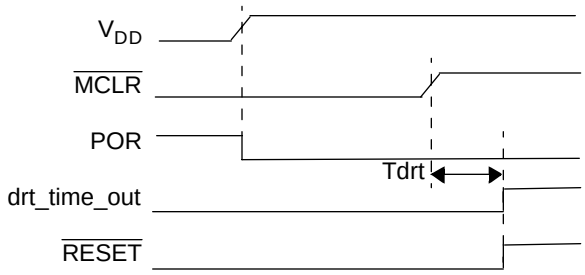

Figure 4 -6 Power-On Reset Timing, Fast V_{DD} Rise Time

Figure 4-7 shows the unacceptable situation where the supply voltage rises too slowly, and the device is allowed to begin operating when the supply voltage has not yet reached a reliable level.

One solution to the situation shown in Figure 4-7 is to use an external RC delay circuit like the one shown in Figure 4-8. This circuit holds the $\overline{\text{MCLR}}$ input low while the supply voltage rises. The values of R and C should be chosen to cause a delay that exceeds the supply voltage rise time. R should be less than 40 kΩ to ensure a sufficiently high voltage. The diode helps to discharge the capacitor quickly when the power is turned off.

The power-on timing with the external RC network is shown in Figure 4-9. In this case, the device comes out of reset about 72 msec after the $\overline{\text{MCLR}}$ input goes high.

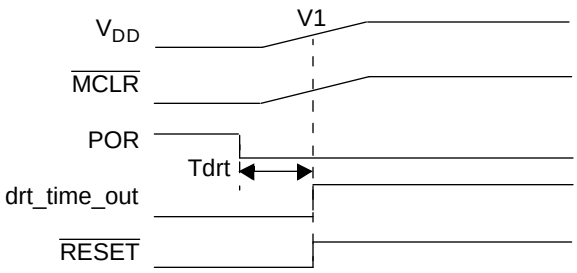

Figure 4 -7 Power-On Reset Timing, V_{DD} Rise Time Too Slow

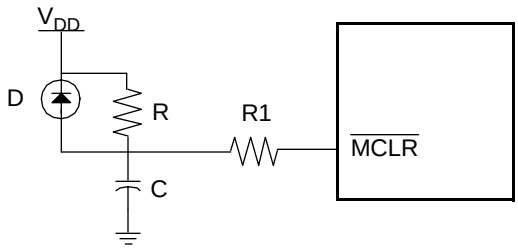

Figure 4 -8 External Power-On MCLR Signal

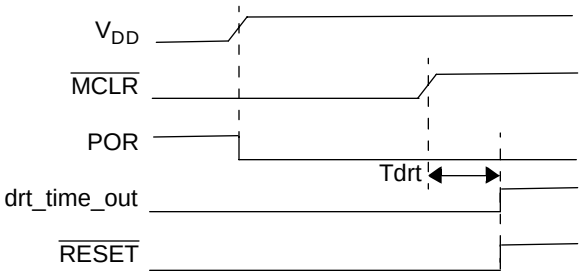

Figure 4 -9 Power-On Reset Timing, Separate MCLR Signal

4.5.3 Wakeup from the Power Down Mode

A wakeup from the power down mode (described in Section 4.3) causes a device reset. The device is designed to not have start-up delay as there is with a power-on reset. This is because the operating supply voltage is already stable when a wakeup occurs.

The device initialization routine can determine the Port B pin that caused the wakeup to occur by reading the WKPND_B register, as described in Section .

4.5.4 Brown-Out Reset

When the supply voltage to the SX device drops below a specified value but remains above zero volts, it is called a “brown-out” condition. The SX device has a brown-out detection circuit that puts the

device into the reset state when a brown-out occurs, and allows the device to re-start when the brown-out condition ends. This feature prevents the device from producing abnormal results when the supply voltage falls to unreliable levels.

The brown-out threshold voltage is factory-set to approximately 4.2 volts. If the supply voltage drops below this level but remains above zero, the brown-out circuit holds the SX device in the reset state. When the voltage rises above this threshold, the device starts operating again, starting at the reset address (7FFh).

You can optionally disable the brown-out detection circuit by setting the BOR0 and BOR1 bits to 1 in the FUSEX word register (a register programmed along with the instruction memory). In that case, the device will still operate below the brown-out threshold voltage, but will produce unreliable results if the supply voltage falls too low.

4.5.5 Watchdog Timeout

A Watchdog timeout occurs if the Watchdog circuit is enabled and the Watchdog timer overflows. This feature provides an escape mechanism from an infinite loop or other abnormal program condition. When a Watchdog timeout occurs, it resets the device just like assertion of the $\overline{\text{MCLR}}$ input.

4.5.6 $\overline{\text{MCLR}}$ Input Signal (Master Clear Reset)

A reset occurs whenever the $\overline{\text{MCLR}}$ (Master Clear Reset) input pin goes low. The device is held in the reset state as long as the $\overline{\text{MCLR}}$ pin is held low. When the input goes high, the program jumps to the reset address at the top of the program memory (7FFh). If you do not intend to use the $\overline{\text{MCLR}}$ pin as a hardware reset input, you should connect tie it together with the power supply pin (Vdd) or to a power-on RC network, as described in Section 4.5.2.

Chapter 5

Input/Output Ports

5.1 Introduction

The SX device has a set of Input/Output (I/O) ports. Each port consists of a set of pins on which the device can read logic signals from other devices, or send logic signals to other devices. Each port pin can be individually software-configured to operate as an input or as an output, to accept TTL or CMOS voltage levels, and to use or not use an internal pullup resistor. Some ports allow the selection of Schmitt-trigger input characteristics.

The SX28AC has three ports, designated Port A, Port B, and Port C. The SX18AC and SX20AC have Port A and Port B, but no Port C. The three ports share many of the same features, but have some characteristics that vary from port to port:

- Port A offers symmetrical drive capability, which means that the same voltage drop occurs across the external load whether the output pin is sourcing or sinking current. There are four pins in this port.
- The Port B pins can be software-configured to operate as general-purpose I/O pins, interrupt/ wakeup inputs, or comparator I/O signals. There are eight pins in this port.
- The Port C pins are general-purpose I/O pins available in the SX28AC (not in the SX18AC or SX20AC). There are eight pins in this port.

5.2 Reading and Writing the Ports

The three ports are memory-mapped into the data memory address space. To the CPU, the three ports are available as the RA, RB, and RC file registers at data memory addresses 05h, 06h, and 07h, respectively. Writing to a port data register sets the voltage levels of the corresponding port pins that have been configured to operate as outputs. Reading from a register reads the voltage levels of the corresponding port pins that have been configured as inputs.

For example, suppose that you want to use all four Port A pins as outputs and you want RA0 and RA1 to be high, and RA2 and RA3 to be low. You would first configure all four pins to operate as outputs, and then you would execute code such as the following:

```
mov  W,#$03    ;load W with the value 03h (bits 0 and 1 high)
mov  $05,W     ;write 03h to Port A data register
```

The second “mov” instruction in this example writes the Port A data register (RA), which controls the output levels of the four Port A pins, RA0 through RA3. Because Port A has only four I/O pins, only

the four least significant bits of this register are used. The four high-order register bits are “don’t care” bits. Port B and Port C are both eight bits wide, so the full widths of the RB and RC registers are used.

To use all four Port A pins as inputs, you would first configure them to operate as inputs and then read them using code such as the following:

```
mov  W,$05   ;move data from Port A into W
```

Note that pins can be individually configured within a port. For example, you could use three pins of Port A as inputs and the remaining pin as an output. For information on configuring the port pins, see Section 5.3.

When you write to a bit position for a port that has been configured as an input, you still write to the port data register, but it has no immediate effect on the pin. If you later configure that pin to operate as an output, it will reflect the value that has been written to the data register.

When you read from a bit position for a port, you are actually reading the voltage level on the pin itself, not necessarily the bit value stored in the port data register. This is true whether the pin is configured to operate as an input or an output. Therefore, with the pin configured to operate as an input, the data register contents have no effect on the value that you read. With the pin configured to operate as an output, what you read generally matches what has been written to the register.

When you use two successive read-modify-write instruction on the same I/O port with a very high clock rate, the “write” part of one instruction might not occur soon enough before the “read” part of the very next instruction, resulting in getting “old” data for the second instruction. To ensure predictable results, avoid using two successive read-modify-write instructions that access the same port data register if the clock rate is high. For more information on this topic, see Section 2.5.3.

5.3 Port Configuration

Each port pin offers the following configuration options:

- data direction
- input voltage levels (TTL or CMOS)
- pullup type (pullup resistor or open collector)
- Schmitt trigger input (for Port B and Port C only)

Port B offers the additional option to use the port pins for the Multi-Input Wakeup/Interrupt function and/or the analog comparator function.

5.3.1 Accessing the Port Control Registers

You set the configuration of a port by writing to a set of control registers associated with the port. A special-purpose instruction is used to write these control registers:

- mov !RA,W (move W to Port A control register)

- mov !RB,W (move W to Port B control register)
- mov !RC,W (move W to Port C control register)

Each one of these instructions writes a port control register for Port A, Port B, or Port C. There are multiple control registers for each port. To specify which one you want to access, you use another register called the MODE register.

5.3.2 MODE Register

The MODE register controls access to the port configuration registers. Because the MODE register is not memory-mapped, it is accessed by the following special-purpose instructions:

- mov M, #lit (move literal to MODE register)
- mov M,W (move W to MODE register)
- mov W,M (move MODE register to W)

The value contained in the MODE register determines which port control register is accessed by the "mov !rx,W" instruction as indicated in Table 5-1. MODE register values not listed in the table are reserved for future expansion and should not be used. Therefore, the MODE register should always contain a value from 08h to 0Fh. Upon power-up, the MODE register is initialized to 0Fh, which enables access to the port direction registers.

Table 5-1 MODE Register and Port Control Register Access

MODE Reg.	mov !RA,W	mov !RB,W	mov !RC,W
08h	not used	CMP_B	not used
09h	not used	WKPND_B	not used
0Ah	not used	WKED_B	not used
0Bh	not used	WKEN_B	not used
0Ch	not used	ST_B	ST_C
0Dh	LVL_A	LVL_B	LVL_C
0Eh	PLP_A	PLP_B	PLP_C
0Fh	RA Direction	RB Direction	RC Direction

After you write a value to the MODE register, that setting remains in effect until you change it by writing to the MODE register again. For example, you can write the value 0Eh to the MODE register just once, and then write to each of the three pullup configuration registers using the three "mov !rx,W" instructions shown at the top of Table 5-1.

5.3.3 Port Configuration Example

The following code example shows how to program the pullup control registers.

```
mov  M,#$0E    ;MODE=0Eh to access port pullup registers

mov  W,#$03    ;W = 0000 0011
mov  !RA,W     ;disable pullups for A0 and A1

mov  W,#$FF    ;W = 1111 1111
mov  !RB,W     ;disable all pullups for B0-B7

mov  W,#$00    ;W = 0000 0000
mov  !RC,W     ;enable all pullups for C0-C7
```

First you load the MODE register with 0Eh to select access to the pullup control registers (PLP_A, PLP_B, and PLP_C). Then you use the MOV !rx,W instructions to specify which port pins are to be connected to the internal pullup resistors. Setting a bit to 1 disconnects the corresponding pullup resistor, and clearing a bit to 0 connects the corresponding pullup resistor.

5.3.4 Port Configuration Registers

The port configuration registers that you control with the MOV !rx,W instruction operate as described below.

RA, RB, and RC Data Direction Registers (MODE=0Fh)

Each register bit sets the data direction for one port pin. Set the bit to 1 to make the pin operate as a high-impedance input. Clear the bit to 0 to make the pin operate as an output.

PLP_A, PLP_B, and PLP_C: Pullup Enable Registers (MODE=0Eh)

Each register bit determines whether an internal pullup resistor is connected to the pin. Set the bit to 1 to disconnect the pullup resistor or clear the bit to 0 to connect the pullup resistor.

LVL_A, LVL_B, and LVL_C: Input Level Registers (MODE=0Dh)

Each register bit determines the voltage levels sensed on the input port, either TTL or CMOS, when the Schmitt trigger option is disabled. Program each bit according to the type of device that is driving the port input pin. Set the bit to 1 for TTL or clear the bit to 0 for CMOS.

ST_B and ST_C: Schmitt Trigger Enable Registers (MODE=0Ch)

Each register bit determines whether the port input pin operates with a Schmitt trigger. Set the bit to 1 to disable Schmitt trigger operation and sense either TTL or CMOS voltage levels; or clear the bit to 0 to enable Schmitt trigger operation.

WKEN_B: Wakeup Enable Register (MODE=0Bh)

Each register bit enables or disables the Multi-Input Wakeup/Interrupt (MIWU) function for the corresponding Port B input pin. Clear the bit to 0 to enable MIWU operation or set the bit to 1 to disable MIWU operation. For more information on using the Multi-Input Wakeup/Interrupt function, see Section 4.4.

WKED_B: Wakeup Edge Register (MODE=0Ah)

Each register bit selects the edge sensitivity of the Port B input pin for MIWU operation. Set the bit to 1 to sense rising (low-to-high) edges. Clear the bit to 0 to sense falling (high-to-low) edges.

WKPND_B: Wakeup Pending Flag Register (MODE=09h)

When you access the WKPND_B register using MOV !RB,W, the CPU does an exchange between the contents of W and WKPND_B. This feature lets you read the WKPND_B register contents. Each bit indicates the status of the corresponding MIWU pin. A bit set to 1 indicates that a valid edge has occurred on the corresponding MIWU pin, triggering a wakeup or interrupt. A bit set to 0 indicates that no valid edge has occurred on the MIWU pin.

CMP_B: Comparator Register (MODE=08h)

When you access the CMP_B register using MOV !RB,W, the CPU does an exchange between the contents of W and CMP_B. This feature lets you read the CMP_B register contents. Clear bit 7 to enable operation of the comparator. Clear bit 6 to place the comparator result on the RB0 pin. Bit 0 is a result flag that is set to 1 when the voltage on RB2 is greater than RB1, or cleared to 0 otherwise. (For more information using the comparator, see Chapter 8.)

5.3.5 Port Configuration Upon Power-Up

Upon power-up, all the port control registers are initialized to FFh. Thus, each pin is configured to operate as a high-impedance input that senses TTL voltage levels, with no internal pullup resistor connected. The MODE register is initialized to 0Fh, which allows immediate access to the data direction registers using the “MOV !rx,W” instruction.

5.3.6 Port Block Diagram

Figure 5-1 is a block diagram showing the internal device hardware for one pin of Port B. This diagram will help you understand how the port operates and how to use it. Note that pin features related to the Multi-Input Wakeup/Interrupt function and the analog comparator function are not shown in this diagram.

The boxes labeled RB Direction, PLP_B, LV_B, and ST_B represent individual control bits within the respective port control registers. The data registers and control registers are all mapped into the data memory space at address 06h. The control registers are accessed with the “mov !RB,W” instruction, with access controlled by the value in the MODE register; while the RB Data register bit is accessed by ordinary file register instructions such as “mov fr,W”.

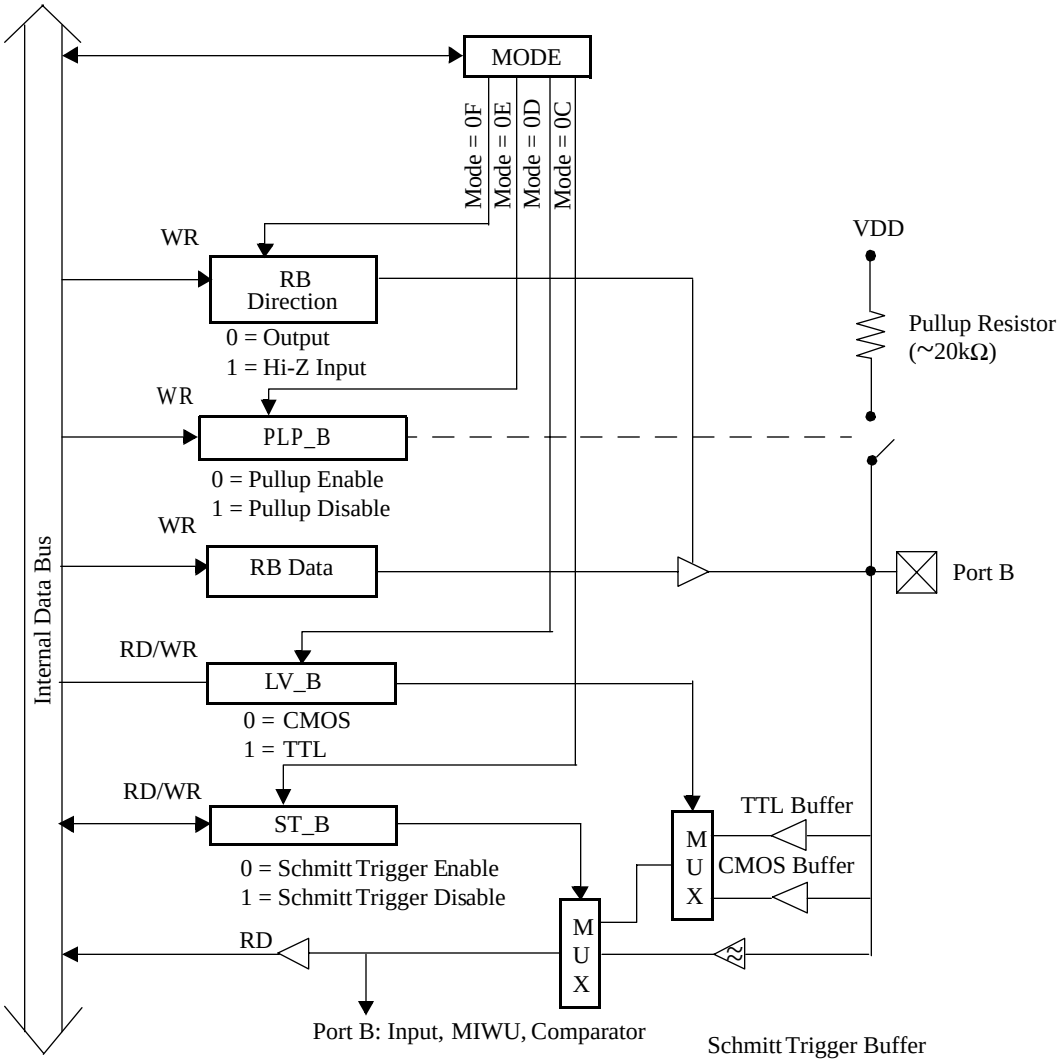

Figure 5 -1 Port B Pin Block Diagram

The port pin is configured to operate as either a high-impedance input or an output, as determined by the RB data register bit. When the pin is configured to operate as an input, the ST_B and LV_B bits determine the type of input buffer used. The ST_B bit either enables or disables the Schmitt trigger buffer. If the Schmitt trigger buffer is disabled, the LV_B bit selects either the TTL or CMOS buffer for sensing the input voltage levels on the pin.

When the device is configured to operate as an output, the bit in the RB data register is buffered and placed on the output pin. Reading from the port data address returns the actual logic level on the pin, even when the pin is configured to operate as an output.

The PLP_B bit either connects or disconnects the internal pullup resistor. If the pullup resistor is disconnected, an external pullup will be required.

The block diagram in Figure 5-1 is for a Port B data pin, but the same diagram also applies to pins of Port A and Port C, with one exception. Port A does not offer a Schmitt trigger input option, so it lacks the control register bit and logic associated with the Schmitt trigger buffer.

Chapter 6

Timers and Interrupts

6.1 Introduction

The SX device has two different timers: the Real-Time Clock/Counter (RTCC) and the Watchdog timer. The RTCC timer can be used to keep track of elapsed time or to count external events. The Watchdog timer provides an automatic escape route from infinite loops and other program errors. An RTCC timer overflow triggers an interrupt, whereas a Watchdog timer overflow triggers a device reset.

The SX device supports interrupts from the RTCC circuit and from up to eight Multi-Input Wakeup pins in Port B. An interrupt causes a jump to the bottom of the program memory (address 0000h), where the interrupt service routine is located. The service routine is terminated by an RETI or RETIW instruction, which causes the device to jump back to the point in the program where the interrupt occurred and restore the program context at that point.

6.2 Real-Time Clock/Counter

The Real-Time Clock/Counter is a general-purpose timer that can be used to keep track of elapsed time or to keep a count of pulses received on the RTCC input pin. The RTCC register is a memory-mapped, 8-bit register that can keep a count up to 256. An 8-bit prescaler register can be used to extend the maximum count to 65,536.

Figure 6-1 is a block diagram showing the RTCC circuit, including the RTCC register, the 8-bit prescaler register, the Watchdog timer (WDT) register, and the supporting multiplexers and configuration bits that control the RTCC timer.

The RTCC register is clocked (incremented) either by the internal instruction clock or by pulses received on the RTCC input pin. The choice is controlled by the RTS bit in the OPTION register. Set this bit to 1 to count pulses on the RTCC input pin, or clear this bit to 0 to count instruction cycles.

If you select the RTCC input pin as the clock source, the RTE_ES bit in the OPTION register specifies the type of transition sensed on the pin. Set the bit to 1 to sense falling edges (high-to-low transitions) or clear the bit to sense rising edges (low-to-high transitions) on the RTCC pin.

6.2.1 Prescaler Register

The 8-bit prescaler register is shared between the Watchdog timer and RTCC circuit. It can be configured to operate as a prescaler for the RTCC circuit or as a postscaler for the Watchdog timer, but

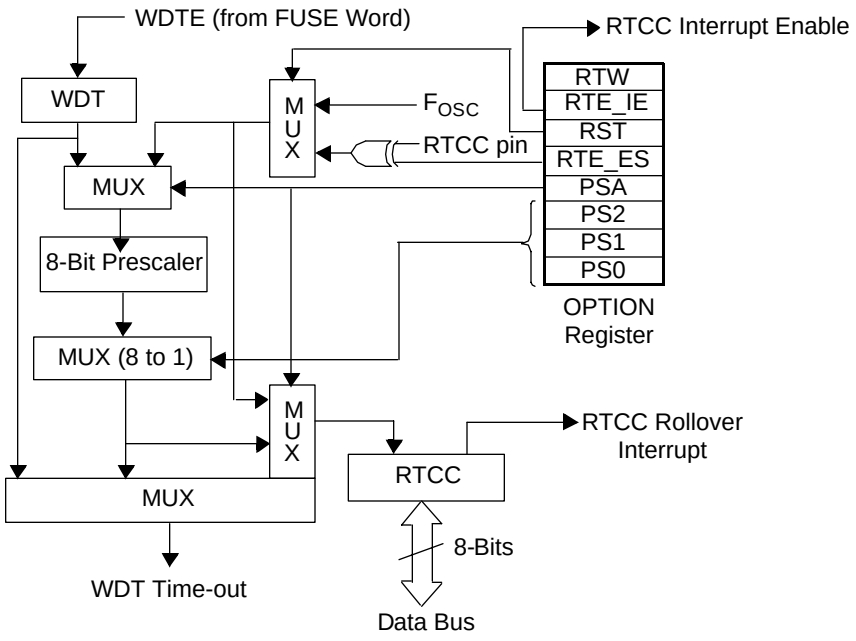

Figure 6 -1RTCC Block Diagram

it cannot be used for both purposes at the same time. The selection between the two possible functions is controlled by the PSA (Prescaler Assignment) bit in the OPTION register.

If the prescaler register is used with the RTCC clock, it reduces the rate at which the RTCC register is incremented. The instruction cycles or external events being counted are used to increment the prescaler register, and transitions of a specified bit in the prescaler register are used to increment the RTCC register.

The rate at which the RTCC register is incremented is reduced by a factor determined by the PS2:PS0 bits in the OPTION register:

6.2.2 Maximum Count

The RTCC counter register is eight bits wide, so it can count up to 256 instruction cycles or external events. If you use the prescaler register and select a divide-by factor of 256, you can count up to 65,536 instruction cycles or events because the RTCC register is incremented only once per 256 instruction cycles or events.

The RTCC counter can be configured to trigger an interrupt each time it overflows from FFh to 00h. To enable this interrupt, clear the RTE_IE bit in the OPTION register. You can have the interrupt service routine increment a file register (or a set of cascaded file registers), and thereby keep track of any number of instruction cycles or events.

6.2.3 RTCC Operation as a Real-Time Clock or Timer

To use the RTCC circuit as a real-time clock, configure it to be incremented by the instruction clock. In that case, the RTCC counter is incremented at a fixed rate of once per instruction clock cycle. For example, if the instruction rate is 4 MHz, The RTCC counter is incremented at 4 MHz, or once per 250 nsec. The accuracy of the timer depends only on the accuracy of the clock that drives the SX device.

To increment the RTCC counter at slower rate, enable the prescaler register and specify the divide-by factor using the PS2:PS0 bits in the OPTION register.

To operate the RTCC as a count-down timer, initialize the RTCC register to the appropriate value and let the counter run. For example, to count 100 instruction cycles, load RTCC with the value 156 (decimal) using an instruction such as "mov $01,#156". The RTCC counter will then increment the register 100 times before it reaches the maximum value and rolls over back to zero, triggering an interrupt (if enabled by the RTE_IE bit in the OPTION register).

If you power down the device using the "sleep" instruction, the device clock is stopped, the RTCC counter stops operating, and the RTCC register contents are lost. Upon wakeup from the power down mode, the RTCC register contains unknown data.

6.2.4 RTCC Operation as an Event Counter

To use the RTCC circuit as an external event counter, configure it to be incremented by pulses on the RTCC input pin. Design the system to generate a pulse for each occurrence of the event, and feed that signal into the RTCC pin. Then the RTCC counter is incremented once for each occurrence of the external event. Use the RTE_ES bit in the OPTION register to specify the type of transition to be sensed on the RTCC pin (rising or falling edges).

To increment the RTCC counter at slower rate, enable the prescaler register and specify the divide-by factor using the PS2:PS0 bits in the OPTION register.

The RTCC circuit can count no more than one event per instruction cycle. Multiple edges received within a single instruction cycle are counted as a single event.

6.2.5 RTCC Overflow Interrupts

The device can be configured to generate an interrupt each time the RTCC register rolls over from FFh to 00h. To do this requires the following actions:

- Clear the OPTION_X bit in the FUSE word register when you program the device. This enables operation of the RTW and RTE_IE bits in the OPTION register.
- Have the software clear the RTE_IE bit in the OPTION register.
- Have the interrupt service routine check the contents of the RTCC register to determine whether an RTCC rollover caused the interrupt, and process the interrupt when that is the cause.

There is no interrupt pending flag associated with RTCC rollover interrupts like there is for each of the Multi-Input Wakeup interrupts. Therefore, the interrupt service routine should read the RTCC register

to determine whether an RTCC rollover caused the interrupt. A register value of 00h (or a very low value) is an indicator that a rollover has just occurred.

6.3 Watchdog Timer

The Watchdog timer is a circuit that provides an automatic escape route from infinite loops and other abnormal program conditions. It can be enabled or disabled by the WDTE (Watchdog Timer Enable) bit in the FUSE word register. In the default configuration, the Watchdog timer is enabled.

The timer has an 8-bit register that is incremented by an independent on-chip oscillator, completely separate from the on-chip RC oscillator that can be used to drive the rest of the device. The counter counts up from 00h to FFh. When the counter rolls over from FFh to 00h (or rolls over the number of times programmed into the prescaler register), it generates a device reset and clears the TO (Timeout) flag in the STATUS register to indicate that a Watchdog timeout has occurred.

To prevent this automatic reset, the application program must periodically set the timer back to zero. This is accomplished by executing the "CLR !WDT" (clear Watchdog Timer) instruction, which clears the Watchdog timer register and prescaler register to zero. Executing this instruction is called "servicing" the Watchdog. The Watchdog timer register is not memory-mapped and is not accessible by any other means.

If the program gets stuck in an infinite loop, it is unlikely to servicethe Watchdog in that loop. In that case, when the Watchdog counts up to FFh and rolls over to 00h (or rolls over a specified number of times), the device is reset automatically, thus providing an escape from the infinite loop. A rollover also clears the TO (Timeout) flag.

The "CLR !WDT" instruction, in addition to clearing the Watchdog timer register, also sets the TO and PD flags to 1 in the STATUS register. The TO flag is cleared to 0 to indicate the occurrence of a Watchdog timeout. The PD flag is cleared to 0 by the "sleep" instruction to indicate that the device has been put into the power down mode.

6.3.1 Watchdog Timeout Period

The Watchdog oscillator has a nominal operating frequency of 14 kHz, or a period of 714 microseconds. At this rate, the 8-bit counter counts from 00h to FFh in 18 milliseconds. This amount of time is the default Watchdog timeout period. The application program needs to execute a "CLR !WDT" instruction at least once every 18 milliseconds to prevent a Watchdog reset.

The Watchdog timeout period can be increased by using the 8-bit prescaler register. This register can be configured to operate with either the Watchdog timer or RTCC circuit, but not both at the same time. This selection is controlled by the PSA (Prescaler Assignment) bit in the OPTION register.

If the prescaler register is used with the Watchdog timer, it actually operates as a postscaler that causes a device reset to occur after the 8-bit Watchdog register overflows a certain number of times. This increases the Watchdog timeout period by a factor determined by the PS2:PS0 bits in the OPTION register. Table 6-1 lists the PS2:PS0 settings and the corresponding divide-by factors and timeout periods.

Table 6-1 Watchdog Timeout Settings

PS2:PS0 (with PSA=1)	Watchdog Timer Output Divide-By Factor	Watchdog Timeout Period
000	1	0.018 sec
001	2	0.037 sec
010	4	0.073 sec
011	8	0.15 sec
100	16	0.29 sec
101	32	0.59 sec
110	64	1.17 sec
111	128	2.34 sec

6.3.2 Watchdog Operation in the Power Down Mode

The Watchdog timer operates even during the power down mode. This feature causes an automatic wakeup from the power down mode after the Watchdog timeout period has elapsed. The Watchdog circuit can continue to operate in power down mode because it is driven by its own on-chip oscillator.

If you do not need to usethe Watchdog timer, you can disable it by clearing the WDTE bit in the FUSE word register. Doing so reduces power consumption in the power down mode because the Watchdog oscillator and counter no longer operate.

6.4 Interrupts

An interrupt is a condition that causes a CPU to stop its normal program execution and perform a separate “service” routine that handles the cause of the interrupt condition. An interrupt can occur at any point in the program and is typically triggered by an event that can happen at any time.

An interrupt causes the CPU to save the program context (program counter, W, STATUS, and FSR) and then jump to address 000h, where the interrupt service routine should be located. The service routine is terminated by a return-from-interrupt instruction, which restores the program context and causes the program to resume execution at the point where it was interrupted.

In the SX device, there are two possible causes of an interrupt:

- a rollover of the Real-Time Clock/Counter (RTCC)
- an interrupt signal received on a Port B input pin that has been configured for Multi-Input Wake-up/Interrupt operation

RTCC interrupts can be used to keep track of elapsed time (for example, to maintain a real-time clock that changes the displayed time once per second). Port B interrupts can be used to handle any type of external device that needs service, such as a hardware peripheral or a serial interface.

6.4.1 Single-Level Interrupt Operation

In the SX device, all interrupts are global in nature and have the same priority. Only one interrupt can be processed at a time. As soon as an interrupt is received, all other interrupts are disabled for the duration of the interrupt service routine.

When an interrupt occurs, all further interrupts are disabled internally, without any effect on the interrupt enable bits (namely, bit 6 of the OPTION register and all bits of the WKEN_B register). Upon return from an interrupt service routine, interrupts are automatically re-enabled. Any interrupt condition that occurred during the service routine is immediately serviced at that time.

The Multi-Input Wakeup/Interrupt circuit continues to operate during an interrupt service routine. It senses valid edges on the enabled wakeup/interrupt input pins and sets the WKPND_B pending flags accordingly. However, these interrupt events are not serviced until the current service routine is completed.

If more than one interrupt condition occurs during an interrupt service routine, the pending interrupts can be serviced in any order upon completion of the current interrupt service routine. There is no “priority” associated with different interrupt sources.

6.4.2 Interrupt Sequence

The following sequence takes place in processing an interrupt:

- The interrupt condition occurs (either an RTCC rollover or a Multi-Input Wakeup/Interrupt signal on Port B). An interrupt is generated if the applicable condition is enabled to operate as an interrupt.
- The CPU automatically saves the current contents of the program counter (all 11 bits) and the W, STATUS, and FSR registers. It saves these register contents in a set of independent shadow registers, not in the program stack. All further interrupts are disabled.
- The program jumps to address 000h, where the interrupt service routine should be located.
- If the device is configured to accept different interrupts, the interrupt service routine should read the applicable registers (WKPND_B and/or RTCC) to determine the cause of the interrupt.
- The interrupt service routine should perform the required task.
- The interrupt service routine should end with a return-from-interrupt instruction, either RETI or RETIW.
- The CPU automatically restores the contents of the program counter, W, STATUS, and FSR registers; and then resumes normal program execution at the point of interruption. If an interrupt condition occurred during the service routine, it immediately triggers a new interrupt at this time.

The interrupt response time is three instruction cycles for an RTCC interrupt or five instruction cycles for a Multi-Input Wakeup interrupt. This is the amount of time it takes from detection of the interrupt condition to execution of the first instruction in the interrupt service routine.

Figure 6-2 is a block diagram showing the internal logic of the interrupt generation circuit. An interrupt can be generated by either an RTCC rollover or a wakeup/interrupt signal on a Port B pin, if enabled by the appropriate bit in the OPTION register or STATUS register. A signal on a wakeup/interrupt pin of Port B generates an interrupt only during normal operation of the device, not in the power down mode.

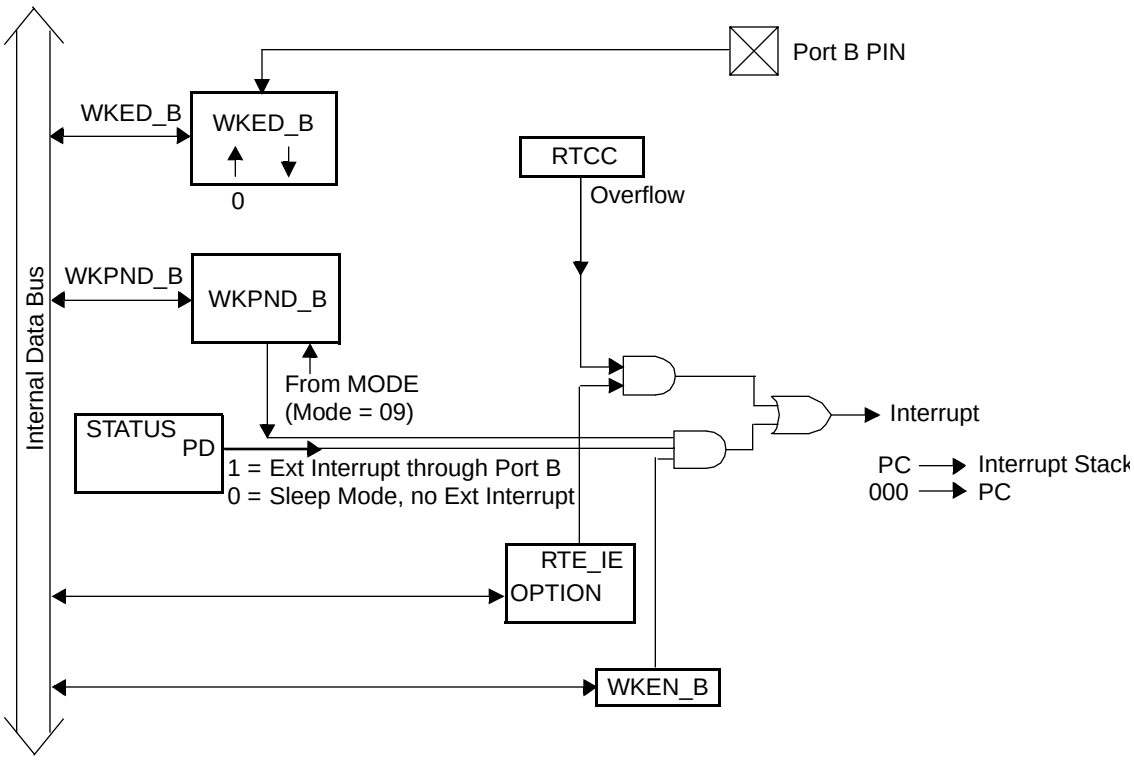

Figure 6-2 Interrupt Logic Block Diagram

6.4.3 RTCC Interrupts

The Real-Time Clock/Counter is a general-purpose timer that can be used to keep track of elapsed time or to keep a count of pulses received on the RTCC input pin. To enable RTCC interrupts, clear the RTE_EI bit in the OPTION register. In that case, the RTCC counter generates an interrupt each time it rolls over from FFh to 00h.

There is no pending flag associated with an RTCC rollover event. Therefore, in order to determine whether an RTCC rollover was the cause of an interrupt, the interrupt service routine should read the RTCC register at address 01h. A register value of zero (or a very low number) is an indicator that a rollover has just occurred.

You can configure the RTCC circuit to count instruction cycles or external events, and you can specify the number of cycles or events that cause the RTCC counter to be incremented. For details, see Section 6.2.

6.4.4 Port B Interrupts

The Multi-Input Wakeup/Interrupt circuit allows the Port B pins to be used as device inputs to trigger an interrupt from an external source. The same circuit is used for both wakeups and interrupts. In the power down state, a wakeup signal on a Port B pin wakes up the device and causes a device reset. The same signal received during normal device operation triggers an interrupt.

You can configure any of the eight Port B pins to operate as wakeup/interrupt input pins and individually enable or disable the corresponding interrupt. On each enabled pin, you can choose to sense either rising or falling edges from the external interrupt source.

Each wakeup/interrupt pin has as associated pending flag to indicate whether a wakeup/interrupt signal has been detected. When Port B has been configured to use multiple Port B interrupt pins, the interrupt service routine should read the wakeup pending register to determine which Port B pin caused the interrupt.

For more information on using the Multi-Input Wakeup/Interrupt pins and the associated Port B registers, see Section 4.4.

6.4.5 Return-from-Interrupt Instructions

There are two return-from-interrupt instructions available:

- RETI (Return from Interrupt)
- RETIW (Return from Interrupt and Adjust RTCC with W)

Both of these instructions cause a return from the current interrupt service routine by restoring W, STATUS, FSR, and the program counter. The RETI instruction is a “plain” return from interrupt, whereas the RETIW also makes an adjustment to the RTCC register prior to the return.

The RETIW instruction subtracts W from RTCC before it restores W, STATUS, FSR, and the program counter. This allows RTCC to be restored to the value it contained at the time the main program was interrupted. To use this feature, the interrupt service routine should check the RTCC register at the beginning of the routine and again at the end of the routine, and then put the adjustment value into W before executing the RETIW instruction.

6.4.6 Interrupt Example

The following code example shows the part of an interrupt service routine that determines the cause of an interrupt and jumps to a processing routine based on the cause. In this example, the RB0 and RB1 pins are configured to operate as interrupt inputs and the RTCC counter is enabled to generate interrupts.

```
   org  0          ;interrupt routine starts at address 000h
   mov  M,#$09     ;set up MODE register to read WKPND_B
   clr  W          ;clear W to zero
   mov  !RB,W      ;exchange contents of W and WKPND_B
   and  W,#$03     ;mask out unused bits from WKPND_B
                   ;W now indicates cause of interrupt:
                   ;00h = RTCC, 01h = RB0, or 02h = RB1
   add  $02,W      ;add W to program counter for indirect jump
   jmp  rtcc_i     ;W=00h, jump to RTCC interrupt service routine
   jmp  rb0_i      ;W=01h, jump to RB0 interrupt service routine
   jmp  rb1_i      ;W=02h, jump to RB1 interrupt service routine
   rtcc_i
...                ;RTCC interrupt service routine here
   reti            ;return from interrupt
   rb0_i
...                ;RB0 interrupt service routine here
   reti            ;return from interrupt
   rb1_i
...                ;RB1 interrupt service routine here
   reti            ;return from interrupt
```


Chapter 7

Analog Comparator

7.1 Introduction

The SX has an analog voltage comparator. The comparator circuit, when properly enabled and configured, compares the analog voltages supplied to two Port B input pins. The comparator determines which voltage is higher and reports the logical result in an internal register and also on a Port B output pin (if enabled for that purpose). The application program can read the result from the internal register, and an external device can read the result from the Port B output pin.

The comparator uses Port B pins RB2, RB1, and RB0. RB2 and RB1 are the comparator inputs, with RB2 operating as the positive input and RB1 operating as the negative input. If the voltage on RB2 is greater than the voltage on RB1, the result of a comparison operation is logic 1. Otherwise, the result is logic 0.

This result is reported on the RB0 pin, which is configured to operate as an output. If the result is only need by the SX software and not by an external device, then RB0 does not need to be used for the comparator function. Instead, it can be used as a general-purpose I/O pin or a Multi-Input Wakeup input pin.

7.2 Comparator Enable/Status Register (CMP_B)

The Comparator Enable/Status Register (CMP_B) is a Port B control register used to enable operation of the comparator, to enable the comparator output pin, and to read the comparison results. The register format is shown below.

There are three non-reserved bits in this register:

- $\overline{\text{CMP\_EN}}$ (Comparator Enable). To enable operation of the comparator, clear this bit to 0. You must also configure RB2 and RB1 to operate as inputs by setting bit 2 and bit 1 in the RB Data Direction register.
- $\overline{\text{CMP\_OE}}$ (Comparator Output Enable). Using the RB0 pin as a comparator output is optional. To do this, clear this bit to 0. You must also configure that pin to operate as an output by clearing bit 0 in the RB Data Direction register.
- CMP_RES (Comparator Result). To determine the comparator result, look at this bit the CMP_B register. A "1" indicates that the voltage on RB2 is greater than the voltage on RB1, and a "0" indicates the opposite. The comparator must be already enabled ($\overline{\text{CMP\_EN}}$ bit cleared to 0) in order to read a valid result.

Upon power-up or reset, the $\overline{\text{CMP\_EN}}$ and $\overline{\text{CMP\_OE}}$ bits are both set to 1. This means that the comparator starts in the disabled state.

7.2.1 Accessing the CMP_B Register

Like all port configuration registers, the CMP_B register is accessed by the "mov !rx,W" instruction in conjunction with the MODE register setting. For example, you can access the CMP_B register using the following commands:

```
mov  M,#$08    ;set MODE register to access CMP_B
mov  W,#$00    ;clear W
mov  !RB,W     ;enable comparator and its output
...            ;delay after enabling comparator for response

mov  M,#$08    ;set MODE register to access CMP_B
mov  W,#$00    ;clear W
mov  !RB,W     ;enable comparator and its output and
               ;also read CMP_B (exchange W and CMB_B)
and  W,#$01    ;set/clear Z flag based on comparator result
snb  $03.2     ;test Z flag in STATUS reg (0 => RB2<RB1)
jmp  rb2_hi    ;jump only if RB2>RB1
...
```

When you use the "MOV !RB,W" instruction to access the CMP_B register, it performs an exchange of data between W and port control register rather than a simple write to the port control register. (An exchange of this type is performed only when you access the CMP_B or WKPND_B register.) In the programming example above, the "MOV !RB,W" instruction writes 00h into the CMP_B register, and simultaneously reads the contents of CMP_B into W. This feature provides a way for the software to read the CMP_B register.

7.3 Comparator Operation

Figure 7-1 is a block diagram showing the internal hardware of the comparator circuit. The two analog inputs to the comparator are the RB2 and RB1 pins. Operation of the comparator is enabled by the CMP_RES bit and operation of the RB0 pin as the comparator output is enabled by the $\overline{\text{CMP\_OE}}$ bit. The comparator result appears in the CMP_RES bit position, whether or not the RB0 output pin is used with the comparator. Read/write access to the CMP_B register is enabled when the MODE register contains 08h.

As long as the comparator is enabled, it operates continuously and reports its result in the CMP_RES bit of the CMP_B register and on the RB0 pin (if enabled for that purpose). To reduce unnecessary power consumption during the power down state, you should disable the comparator before using the "sleep" instruction.

The comparator takes some time to respond after it is enabled and after a change in the analog input voltages. For details, see the comparator DC and AC specifications in Appendix C.

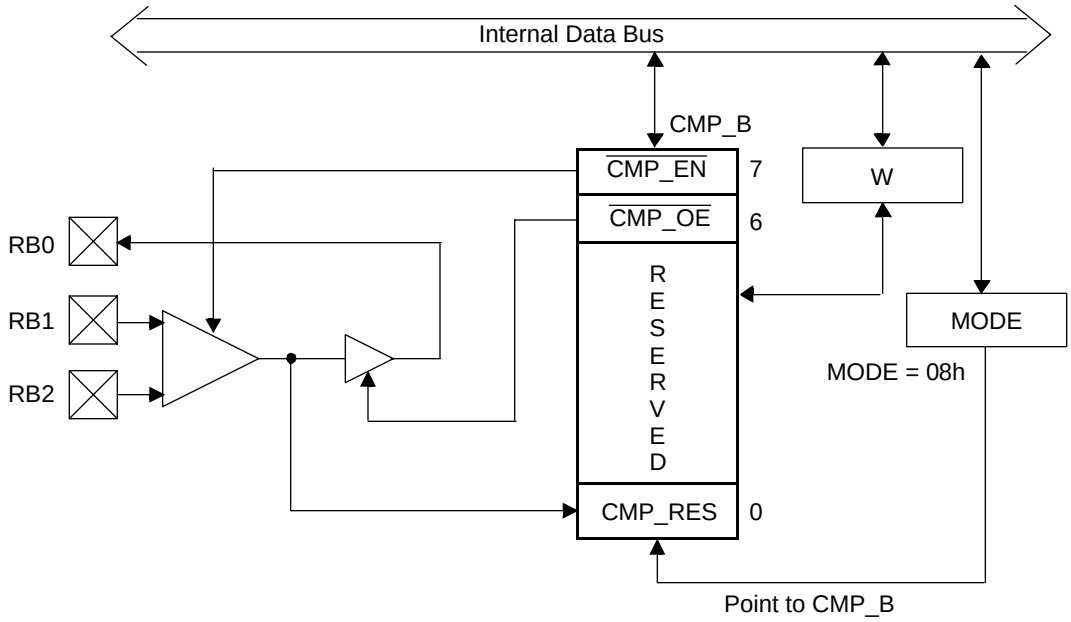

Figure 7 -1 Comparator Block Diagram

Chapter 8

Device Programming

8.1 Introduction

The SX device has a program memory consisting of 2,048 words of 12 bits per word, plus some additional 12-bit words that specify the device configuration. This memory is a non-volatile, electrically erasable (EEPROM) flash memory, rated for 10,000 rewrite cycles.

Before you can use the SX device, you must write the application code into the program memory. You do this by placing the device into a programming mode and following the protocol for accessing the program memory. You can write to the program memory only in the programming mode, not when the device is executing the application software.

8.1.1 Erasure and Reprogramming

When you erase the program memory, you automatically erase the entire memory, including the DEVICE word, FUSE word, and FUSEX word. An erased memory has all bits set to 1. When you program the device, you clear some of the bits to 0. If you want to reprogram a memory location and clear some more bits to 0, you can "overwrite" the memory location without erasing. However, if you want to program a bit to 1 that has already been cleared to 0, the only way to do so is to erase and reprogram the whole EEPROM memory.

8.1.2 Standard and Custom Programming Tools

The task of programming an SX device is most easily handled by using a third-party tool. One such tool is the SX-Key made by Parallax, Inc. The Parallax tool is simple and easy to use because is takes care of all of the programming details. You only need to write the source code and specify the device options. The tool compiles the code, erases the existing program in the device, and writes the new configuration settings and program code into the device.

Instead of using an existing tool such as the SX-Key, you might want to build your own programmer unit to accommodate the special requirements of your application. This chapter describes the device programming interfaces and protocols so that you can build your own programming unit.

8.1.3 In-System and Parallel Programming Modes

There are two basic device programming modes, called the "In-System Programming" (ISP) mode and the "parallel" mode. The In-System Programming mode uses just two device pins, OSC1, and OSC2, and writes the data to the device serially, one bit at a time. This mode lets you program devices that are

already installed in the target system. The parallel mode uses a larger set of pins and writes data 12 bits at a time, in parallel. This mode is a little faster but can only be used to program free-standing SX parts.

8.2 In-System Programming (ISP) Mode

The In-System Programming (ISP) mode lets you program or re-program an SX device that has been installed and soldered into the target system. Using the ISP mode has many advantages over traditional programming methods, in all stages of the product life: development, manufacturing, and customer service.

In the product development cycle, a separate "emulation" type device is not required. The controller device used for development is the same as the one used for final production, including the package type and pinout. The SX device can be soldered into the target system, and then programmed and reprogrammed any number of times, without removing and reinstalling it. No special socket or support circuitry is required, so the system can be debugged accurately, even in timing-sensitive and noise-sensitive applications.

For manufacturing, circuit boards can be pre-built with the controller installed and soldered on the board, even before the software has been finalized, to meet short time-to-market requirements. Additional information such as vendor numbers and serial numbers can be programmed into the device just prior to shipment. There is no risk of stocking out-dated, pre-programmed units because the software can be corrected or updated at any time.

Even after the product is received by the customer, it can be quickly and easily revised or patched by field service personnel. Customers can even reprogram their products themselves if they have the necessary programming equipment. This equipment is relatively inexpensive and easy to use.

8.2.1 Scenix In-System Programming Implementation

The Scenix ISP method is a proprietary system that uses just two device pins: the clock input pin, OSC1, and the clock output pin, OSC2 (VDD, and GND, and $\overline{\text{MCLR}}$ pins should be connected properly). This system eliminates the need for dedicated programming pins, thus reducing the total device pin count. There is no need for a JTAG tester, an expensive device required by some other programming systems.

OSC1 is used to supply the higher voltage necessary for programming the flash memory (12.5 V), while OSC2 is used to issue commands, to write data to the EEPROM, and to read data back from the EEPROM. The external programming device writes a data stream to OSC2 to specify the ISP programming operations, and to supply the data written into the program memory. When the specified operation is a request to read a program memory location, the SX returns the results as a data stream on the same pin, OSC2.

The OSC1 and OSC2 pins are usually connected to passive components such as resistors, capacitors, and crystals. In typical systems, these components do not interfere with the programming signals and are not harmed by the higher voltages used for programming. In these cases, they can be left connected to the SX device during programming. It is usually not necessary to install additional hardware to isolate the ISP circuit from the rest of the system.

There are three stages to the Scenix ISP protocol:

- Entering the ISP Mode
- Programming in ISP Mode
- Exiting the ISP Mode

8.2.2 Entering the ISP Mode

For normal operation of the SX device, the OSC2 pin is either left unconnected, connected to passive components, or used as a clock output pin, depending on the chosen clock configuration. To put the device into the ISP mode, you pull the OSC2 pin low for at least nine consecutive clock cycles on the OSC1 pin (or nine internal clock cycles in the internal clocking mode). This action is a signal to the SX to go into the programming mode.

The exact procedure for entering the ISP mode depends on whether you are using external or internal components for normal clocking of the device. If you are using an external crystal or resonator (including the XT, LP, or HS mode), an external RC oscillator, or an external clock signal for normal device operation, then you need to use the control signals and timing shown in Figure 8-1 to enter the programming mode. If you are using the internal RC oscillator for normal device operation, then you need to use the control signals and timing shown in Figure 8-2 to enter the programming mode.

3

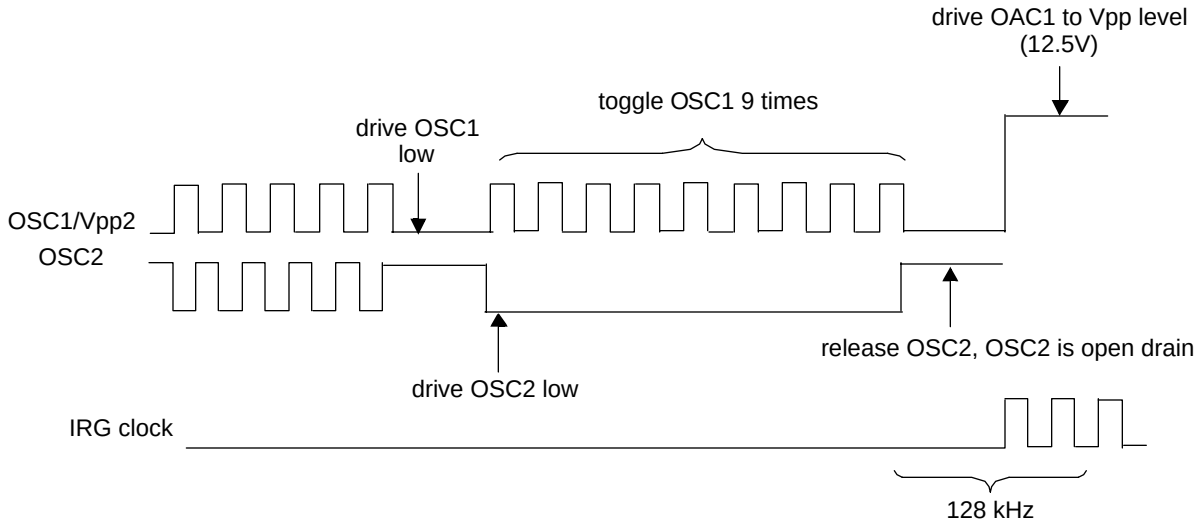

Figure 8 -1ISP Mode Entry with External Clocking

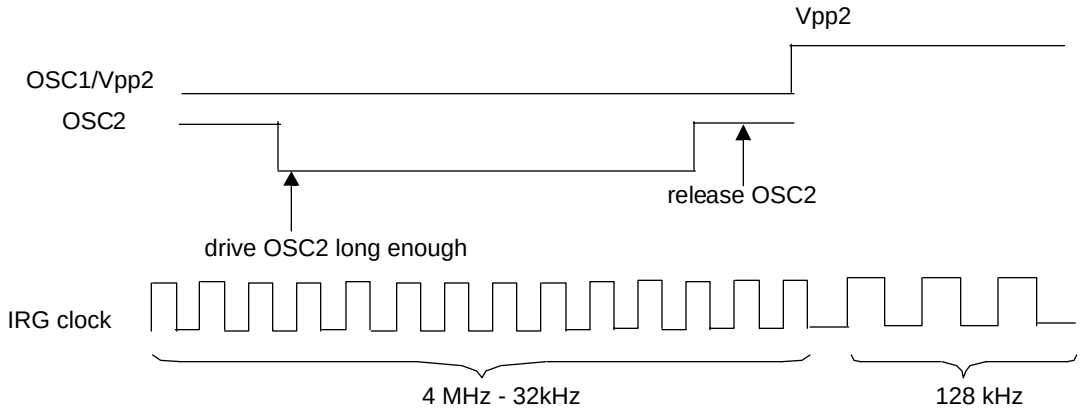

Figure 8-2 ISP Mode Entry with the Internal RC Oscillator

External Clocking

When the device is clocked by external components or an external clock signal, the programmer unit should use the following procedure to place the SX device in the ISP programming mode:

1. Drive the OSC1 pin low to stop the clock.
2. Drive the OSC2 pin low and toggle the OSC1 pin at least nine times. This is the signal to enter the ISP mode.
3. Release the OSC2 pin.
4. Apply the VPP programming voltage to the OSC1 pin. The SX internal RC oscillator starts operating at 128 kHz. This clock drives the SX device during ISP mode programming.

Internal RC Oscillator

When the device is clocked by the internal RC oscillator, the programmer unit should use the following procedure to place the SX device in the ISP programming mode:

1. Drive the OSC2 pin low for at least nine internal clock cycles. The internal clock frequency can be any one of eight values ranging from 31.25 kHz to 4 MHz, depending on the divide-by rate programmed into the FUSE word.
2. Release the OSC2 pin.
3. Apply the VPP programming voltage to the OSC1 pin. The SX internal RC oscillator starts operating at 128 kHz. This clock drives the SX device during ISP mode programming.

8.2.3 Programming in ISP Mode

Upon entry into the ISP mode, the SX device could be in the middle of executing a program, possibly with some I/O ports configured as outputs and driving other devices in the system. The first action of the ISP logic is to reset the SX device. This puts the device into a known logic state and configures the I/O ports to operate as inputs, thus preventing possible damage to other components in the system.

After the device is reset, the ISP logic executes the ISP protocol. This is a "self-aligned" serial communication protocol that uses the OSC2 pin for both synchronization and for serial I/O. No separate clock pin is needed in this protocol. The OSC2 pin is implemented with an open drain and an internal pullup, allowing it to operate as an input or output.

Frames, Cycles, and Internal Clocks

Communication is carried out in packets called "frames." Each frame consists of 17 cycles, and each cycle consists of four internal clocks. The period of the internal clock is 7.81 microseconds, so each cycle is 31.3 microseconds and each frame is 531 microseconds.

Figure 8-3 shows the timing of an ISP frame. The frame consists of 17 cycles. The first cycle is the "sync" cycle, used to synchronize the programmer unit to the ISP frame. This is followed by four "command" cycles, designated C3 through C0. The programmer unit drives the OSC2 pin during these cycles to specify a programming operation such as "erase," "read," or "write." The command cycles are followed by 12 "data" cycles, designated D11 through D0. During these cycles, the programmer unit drives the OSC2 pin for a "write" operation, or the SX device drives the pin for a "read" operation.

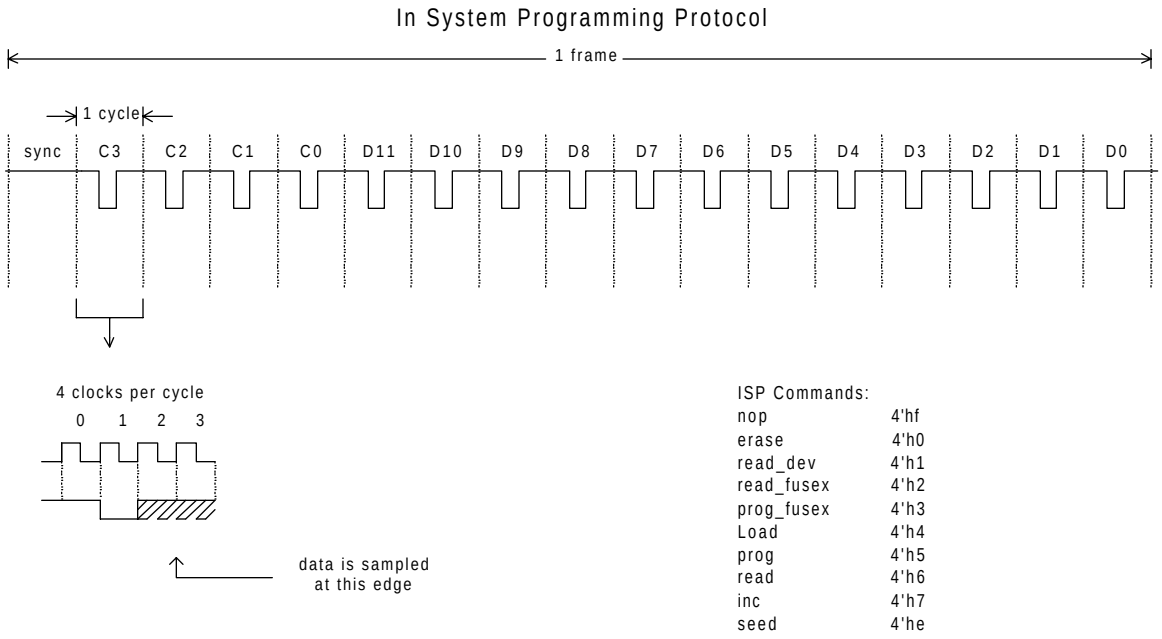

Figure 8 -3 ISP Frame

Each of the 17 cycles consists of four internal clock periods.

In the first clock period, nothing drives the OSC2 pin, so the pin is pulled high by an internal pullup resistor.

In the second clock period, the SX device drives the OSC2 pin low. This is the synchronization pulse. The external programming unit uses the leading edge of this pulse to synchronize itself to the SX device. The pulse is omitted in the sync cycle (the first of 17 cycles in a frame) so that the programming unit can determine where the frame starts.

In the third and fourth clock periods, the programmer unit writes a data bit to the SX device or reads a data bit from the SX device, depending on the cycle and the type of command issued. The data bit is placed on the OSC2 pin during these two clock periods, either by the programmer unit or by the SX device, and then sampled by on the rising edge of the fourth clock period.

In the four command cycles (C3-C0), the programmer writes a four-bit command to the ISP logic, which tells the ISP logic what to do during the data cycles. In the 12 data cycles (D11-D0), for a "write" operation, the programmer writes the 12 bits that are to be written to a memory location. For a "read" operation, the programmer reads 12 bits supplied by the SX device from a memory location.

Internal Hardware

Figure 8-4 is a simplified block diagram of the chip-internal ISP hardware.

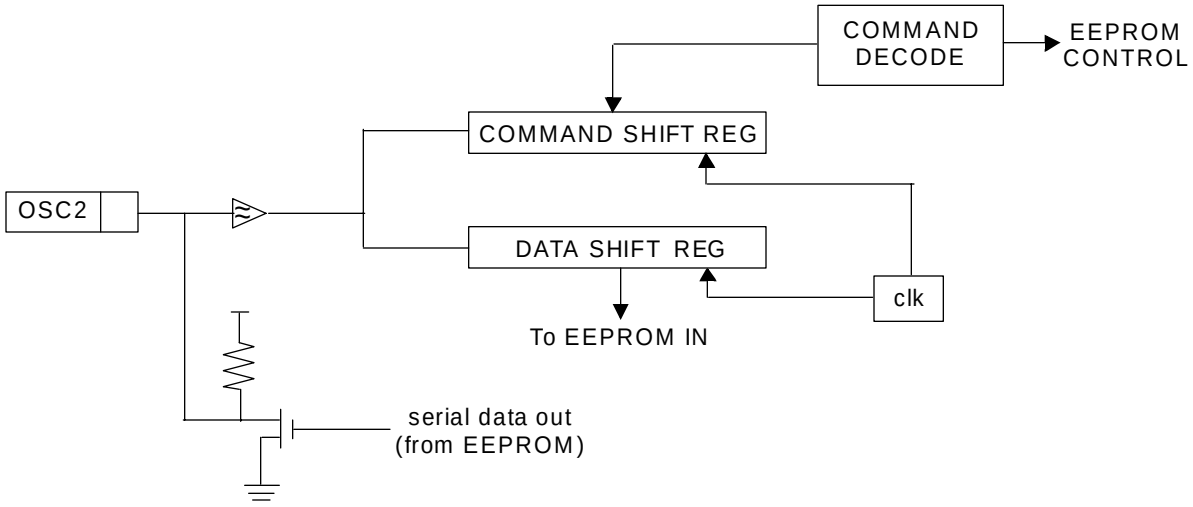

Figure 8-4 ISP Circuit Block Diagram

Serial data written to the OSC2 pin is shifted into the command shift register or data shift register, depending on whether command bits or data bits are being processed within a frame. Command bits are decoded and used to control the flash EEPROM block, while data bits are written to the flash EEPROM.

When the command is to read data from the program memory, the data bits are read from the EEPROM block and shifted out on the OSC2 pin during the data cycles. An open-drain transistor and a pullup

resistor pull the OSC2 pin low or high for each bit. This same transistor is used to pull the OSC2 pin low during the second clock within each cycle (except in the sync cycle).

Commands

The programmer unit writes a 4-bit command during the four command cycles at the beginning of each frame, just after the sync cycle. This 4-bit command tells the ISP logic what to do during the remaining 12 cycles of the frame. Table 8-1 lists and describes the programming commands. Codes not listed in the table are reserved for future expansion.

Table 8-1 ISP Commands

Name	Code	Description
Erase	0000	Erase all EEPROM locations.
Read DEVICE Word	0001	Read DEVICE word (memory size configuration).
Read FUSEX Word	0010	Read FUSEX word (configuration options)
Program FUSEX Word	0011	Program FUSEX word (configuration options)
Load Data	0100	Load data word to be programmed into memory.
Program Data	0101	Program previously loaded data word into memory.
Read Data	0110	Read data word from memory.
Increment Address	0111	Increment the program memory pointer by one.
NOP	1111	No operation.

The commands that erase or program the EEPROM registers must be repeated consecutively for a certain frames in order to work reliably. To determine the minimum required number of repetitions of a command, look in the Electrical Characterization appendix and find the minimum time requirement for the operation. Divide this value by the frame period, 0.53 milliseconds, and round up to the nearest whole number.

For example, if you find that the minimum time requirement for an "Erase" operation is 100 msec, divide 100 by 0.53 and round up, and the result is 189. This means that you must repeat the "Erase" command for at least 189 consecutive frames in order to complete the "Erase" operation reliably.

No repetition is necessary to read a register or to increment the memory address pointer. You can complete one of these operations in just a single frame.

NOP Command

The NOP (no-operation) command causes the ISP logic to do nothing and wait for the next command. The NOP command has a code of 1111 binary. Whenever the programmer unit is not driving to the OSC2 pin, the internal pullup resistor pulls the pin high, which produces 1111 as the command string and invokes the NOP command by default.

This is an important feature because the programmer unit needs some time to synchronize itself to the pulses generated by the ISP logic, and cannot begin driving the OSC2 until synchronization is achieved. In the meantime, the NOP command is executed by default, causing the ISP logic to wait for the first active command.

Reading the DEVICE Word

The DEVICE word is a hard-wired, read-only register containing device information such as the number of register banks and the size of the program memory and SX version number. To read the DEVICE register, the programmer unit issues the "Read DEVICE Word" command and reads the 12 bits of data in the data cycle portion of the frame.

Reading and Programming the FUSEX Word

The FUSEX word is a read/write register that controls device options such as carry flag operation and the brown-out reset function. The five highest-order bits of this register (bits 11:7) are factory-set to certain values that must not be changed. Therefore, the programmer must always read these bits before erasure and reprogram them to the same values after erasure. Bits 11, 9, and 8 are used to calibrate the internal clock. Bit 10 specifies the package size (1 for a 28-pin device or 0 for an 18-pin or 20-pin device). Bit 7 is reserved for future expansion.

To read the FUSEX register, the programmer unit issues the "Read FUSEX Word" command and reads the 12 bits of data in the data cycle portion of the frame.

To program the FUSEX register, first issue the Load command,, the programmer unit issues the "Program FUSEX Word" command and writes the 12 bits of data in the data cycle portion of the frame. This command must be repeated consecutively for a certain number of frames in order to program the register reliably, as explained earlier.

Erasing the Memory

The "Erase" command erases all of the EEPROM memory, including the DEVICE word and FUSEX word. The command must be repeated consecutively for a certain number of frames in order to complete the operation reliably, as described earlier.

The programmer unit should always read the FUSEX word before erasure and restore the five highest-order bits of that register after erasure. These bits have factory-set values that must be maintained.

Reading the Memory

To read the EEPROM program memory, you use two commands: "Read Data" to read the current memory location and "Increment Address" to change an internal memory address pointer from one location to the next.

Upon entry into the ISP mode, the ISP logic is set to access address FFFh, which is the address of the FUSE word. The FUSE word controls many of the device configuration options such as the clocking, stack size, and Watchdog options. To read this initial memory location, the programmer unit issues the "Read Data" command and reads the 12 bits of data in the data cycle portion of the frame.

To read the word at the next address, the programmer unit issues the "Increment Address" command. This increments an internal pointer to the program memory, allowing access to address 000h. It does not matter what the programmer unit does during the 12 data cycles of the "Increment Address" frame. Following this frame, the programmer issues another "Read Data" command and reads the 12 bits of data in the data cycle portion of the frame.

This sequence is repeated to read consecutive memory locations. The first memory location is FFFh (the FUSE word register), followed by 000h, 001h, 002h, and so on up to the top memory address, 7FFh. The programmer can skip over any number of memory locations by repeating the "Increment Address" command consecutively, without using the "Read Data" command. The "Increment Address" command must be used 2,048 times to traverse the whole program memory.

Programming the Memory

To program the EEPROM program memory, you use three commands: "Load Data" to load the a word to be written to a memory location, "Program Data" to write the word into memory, and "Increment Address" to change the memory address pointer from one location to the next.

Upon entry into the ISP mode, the ISP logic is initially set to access the FUSE word. To program this memory location, the programmer unit issues the "Load Data" command and writes the 12 bits of data in the data cycle portion of the frame. Then it issues a "Program Data" command to write the loaded word. It does not matter what the programmer unit does during the 12 data cycles of the "Program Data" frame. This command must be repeated a certain number of times in order to program the register reliably, as explained earlier.

To program the word at the next address, the programmer unit issues the "Increment Address" command, which increments the internal pointer to access the words at address 000h. Following the "Increment Address" frame, the programmer issues another "Load Data" command and writes the 12 bits of data in the data cycle portion of the frame. Then it issues the "Program Data" command consecutively for a certain number of frames.

This sequence is repeated to program consecutive memory locations. The first memory location is FFFh (the FUSE word register), followed by 000h, 001h, 002h, and so on up to the top memory address, 7FFh. The programmer can skip over any number of memory locations by repeating the "Increment Address" command consecutively

8.2.4 Exiting the ISP Mode

Exiting from the ISP mode must be done according to the following protocol to prevent possible damage to system components:

1. The programmer drops the voltage on the OSC1 pin from the programming voltage (VPP = 12.5 V) to logic zero. This is a signal to exit from the ISP mode.
2. On the next rising clock edge after the sync cycle, the SX device exits from the ISP mode and generates an internal reset signal that resets the device (programmer has to observe protocal until stage 2).
3. The programmer releases the OSC1 pin, allowing the SX device to begin normal operation.

8.3 Parallel Programming Mode

The parallel programming mode is faster than the ISP mode because you read and write data in parallel, 12 bits at a time, rather than serially. However, the parallel mode uses a larger number of pins, which means that you can use it only to program free-standing devices, or devices whose programming pins can be isolated from the rest of the system.

The parallel programming modes uses the following device pins:

- Vss (ground)
- Vdd to supply the normal operating voltage
- $\overline{\text{MCLR}}$/Vpp to supply the programming voltage (12.5 V)
- RA0-RA3 (Port A pins) to read and write the four low-order data bits
- RB0-RB7 (Port B pins) to read and write the eight high-order data bits
- RTCC to control programming operations
- OSC1 to increment the address pointer

8.3.1 Parallel Programming Operations

To use the parallel programming mode, you first power up the device in the normal operating mode, keeping the device in the reset state by holding the $\overline{\text{MCLR}}$ input low. You apply a 12-bit command to the Port A and Port B pins, and then apply the programming voltage (Vpp = 12.5 V) to the $\overline{\text{MCLR}}$ pin. This puts the device into the parallel programming mode and latches the command into the programming logic. The rise time of the signal on the $\overline{\text{MCLR}}$ pin should be at least 1.0 microsecond.

After the device has been put in the parallel programming mode, you use the port pins to read and write data, the RTCC pin to control the timing of programming operations, and the OSC1 pin to increment the address pointer.

To exit from the parallel programming mode, you can bring the $\overline{MCLR}$ back down to zero volts, which puts the device back into the reset state; or bring the Vdd pin down to zero volts, which shuts off power to the device.

It is not necessary to shut off the power between successive programming operations. For example, you can erase the memory and then proceed directly to programming the memory while leaving the power supplied to the Vdd pin. You only need to use the $\overline{MCLR}$ pin to latch the next command.

To protect the internal circuitry of the device, use a 100 (resistor between the $\overline{MCLR}$ pin and the Vpp power supply.

8.3.2 Commands

The programmer unit writes a 12-bit command to tell the SX programming logic what to do. Table 8-2 lists and describes the programming commands. Command codes are shown in hexadecimal format. Codes not listed in the table are reserved for future expansion.

Table 8-2 ISP Commands

Name	Code	Description
Erase	010h	Erase all EEPROM locations.
Read DEVICE Word	001h	Read DEVICE word (memory size configuration).
Read FUSEX Word	002h	Read FUSEX word (configuration options)
Program FUSEX Word in SX28 device	003h	Program FUSEX word (configuration options) in a 28-pin SX device
Program FUSEX Word in SX18 device	020h	Program FUSEX word (configuration options) in an 18-pin SX device
Read Data	004h	Read data words from memory.
Program Data	FFEh	Program data words into memory.

The RTCC pin is used to control the timing of programming operations. For each operation, the RTCC signal must be asserted for at least a specific period of time in order to work reliably. To determine the minimum required time, look in the Electrical Characterization appendix and find the time requirement for that particular operation.

8.3.3 Erasing the Memory

The "Erase" command erases all of the EEPROM memory, including the DEVICE word, FUSE word, and FUSEX word.

Before you perform an erase operation, you should read and save the FUSEX register so that you can restore the five high-order bits when you reprogram the device. These bits have factory-set values that must be maintained by the programming unit.

Figure 8-5 shows the signals used and the timing requirements for an "Erase" operation. The "Erase" signal at the bottom of the diagram is a chip-internal signal that is asserted during the erase operation. It becomes active as soon as the "Erase" command is decoded and is deactivated by the falling edge of the RTCC signal. The duration of this signal must be at least the value specified for this operation in the Electrical Characterization appendix.

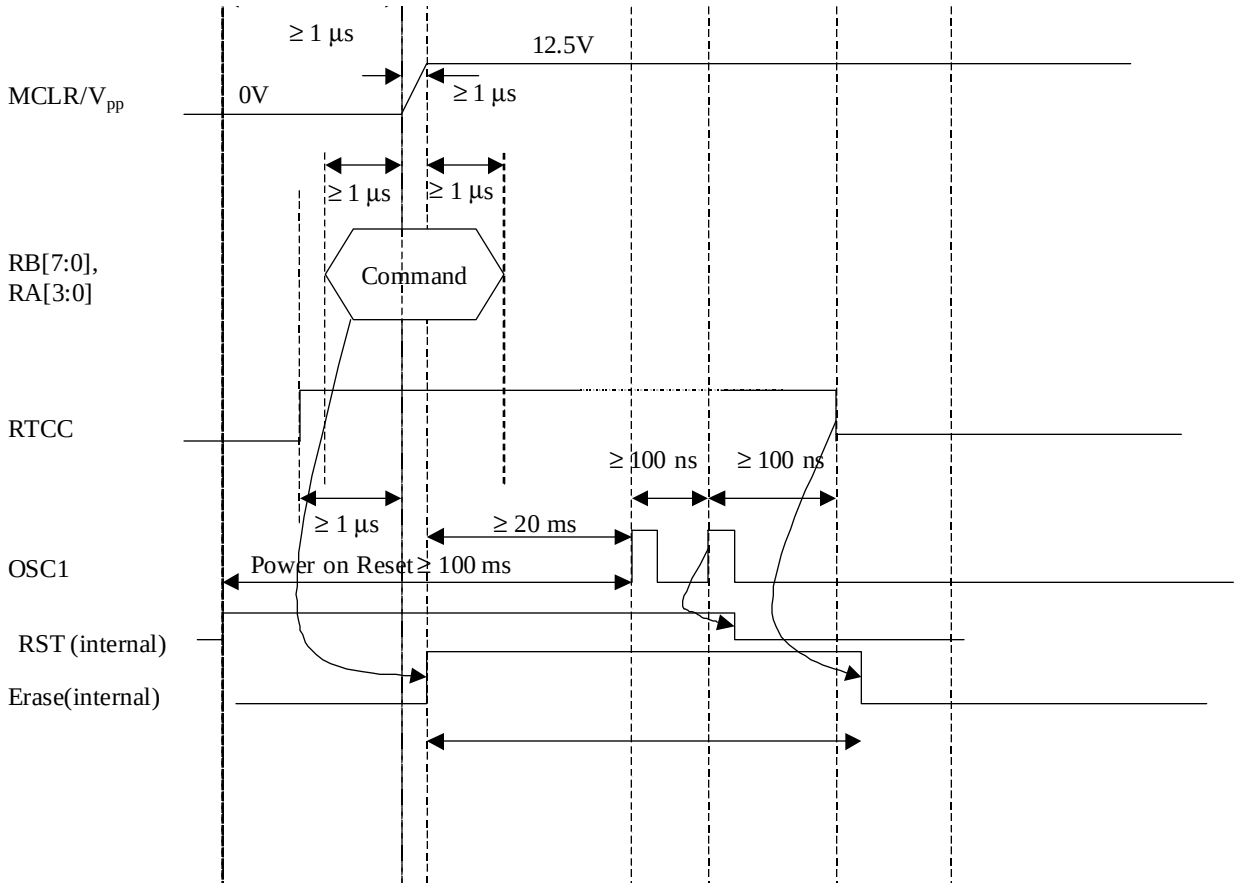

Figure 8 -5 Erase Timing in Parallel Mode

This is the procedure shown in Figure 8-5:

1. If the power is not already on, apply normal power to the Vdd pin while holding the $\overline{\text{MCLR}}$ pin low.
2. Apply a logic high signal to the RTCC pin.

3. Apply the "Erase" command to the Port A and Port B pins.
4. Apply the programming voltage to the $\overline{MCLR}$ pin. This latches the "Erase" command and starts the erase operation.
5. After the erase time has elapsed, apply a logic low signal to the RTCC pin.
6. Bring the $\overline{MCLR}$ pin back down to zero.

Reading the Memory

To read the EEPROM program memory, you use the "Read Data" command in conjunction with the OSC1 pin, which operates as a device input. You use the OSC1 pin to increment the internal pointer for each successive memory address.

Figure 8-6 shows the signals used and the timing requirements for a "Read Data" operation. The "RST" and "Address" signals at the bottom of the diagram are chip-internal signals. The "RST" signal is an internal reset signal that is deactivated by the second pulse on the OSC1 pin. The "Address" waveform represents the internal address bus used to access the program memory. The address is incremented by each pulse on the OSC1 pin, starting with the second pulse.

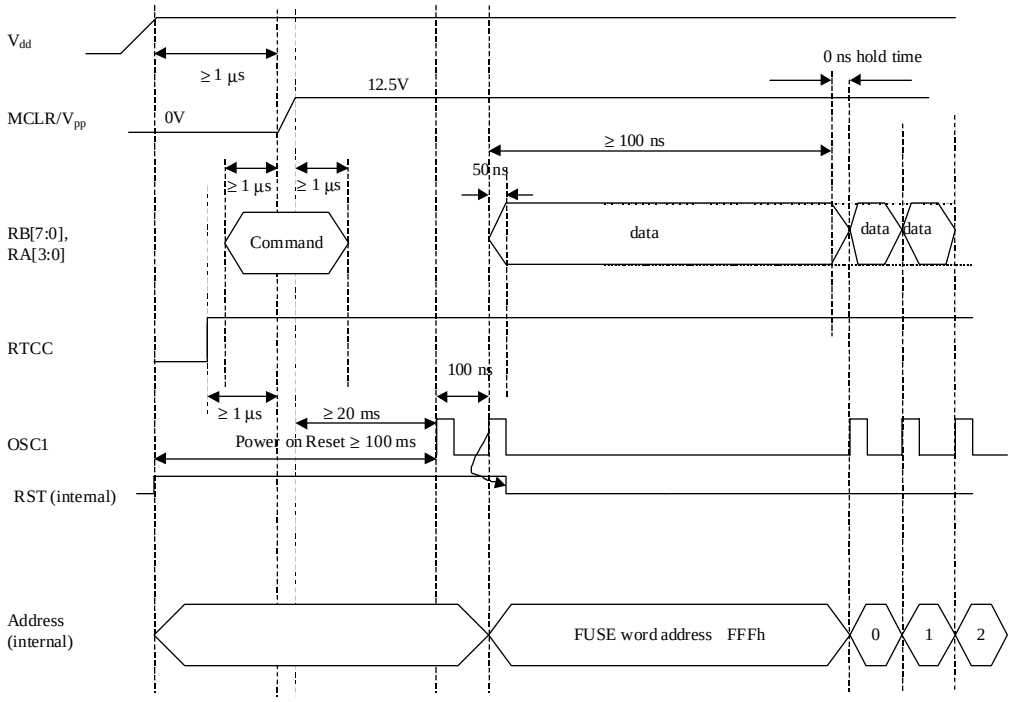

Figure 8 -6 Read Timing in Parallel Mode

This is the procedure shown in Figure 8-6:

1. If the power is not already on, apply normal power to the Vdd pin while holding the $\overline{MCLR}$ pin low.
2. Apply a logic high signal to the RTCC pin.

3. Apply the "Read Data" command to the Port A and Port B pins.
4. Apply the programming voltage to the $\overline{MCLR}$ pin. This latches the "Read Data" command.
5. After the required time has elapsed (100 msec from power-up or 20 msec from latching the command), generate two pulses on the OSC1 pin. The second pulse takes the device out of the reset mode and generates the first device address, FFFh (the address of the FUSE word). The device reads the data from that address and places the 12-bit result on the Port A and Port B pins, allowing the programmer unit to read the data.
6. Generate a single pulse on the OSC1 pin. This advances the address to the next memory location (000h comes after FFFh) and reads the data from that memory location.
7. Repeat step 6 to read successive memory locations. Do this step 2,048 times to read all memory-mapped program locations from 000h through 7FFh.
8. Bring the $\overline{MCLR}$ pin back down to zero.

Programming the Memory

To write to the EEPROM program memory, you use the "Program Data" command in conjunction with the RTCC and OSC1 pins, which operate as device inputs. You use the RTCC pin to tell the device when to write the data and read back the data, and the OSC1 pin to increment the internal memory address pointer.

Figure 8-7 shows the signals used and the timing requirements for a "Program Data" operation. The "RST" and "Address" signals at the bottom of the diagram are chip-internal signals.

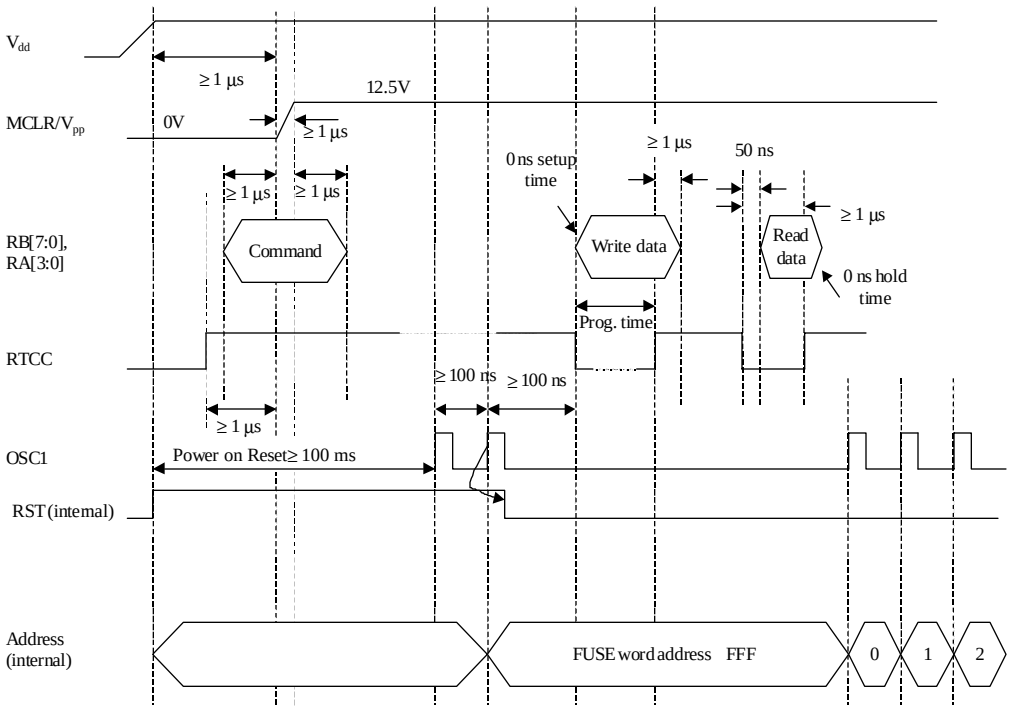

Figure 8-7 Program Timing in Parallel Mode

This is the procedure shown in Figure 8-7:

1. If the power is not already on, apply normal power to the Vdd pin while holding the $\overline{\text{MCLR}}$ pin low.
2. Apply a logic high signal to the RTCC pin.
3. Apply the "Program Data" command to the Port A and Port B pins.
4. Apply the programming voltage to the $\overline{\text{MCLR}}$ pin. This latches the "Program Data" command.
5. After the required time has elapsed, generate two pulses on the OSC1 pin. The second pulse takes the device out of the reset mode and generates the first device address, FFFh (the address of the FUSE word).
6. Apply the 12-bit data word to the Port A and Port B pins
7. Pull the RTCC pin low. This causes the device to read the data on the port pins and write that data to the program memory. After the required time has elapsed, pull the RTCC pin high.
8. Pull the RTCC pin low. This puts the device through a read cycle (as described in the previous section), allowing the programmer unit to verify that the data word has been written correctly to the program memory location. After reading is done, drive the RTCC pin high again.
9. Generate a single pulse on the OSC1 pin. This advances the address to the next memory location (000h comes after FFFh). Repeat to skip over multiple memory locations.
10. Repeat steps 6 through 9 to program successive memory locations. Do these steps 2,048 times to program all memory-mapped program locations, from 000h to 7FFh.
11. Bring the $\overline{\text{MCLR}}$ pin back down to zero.

At each new address, the first active-low pulse on the RTCC pin writes a 12-bit data word, and the second such pulse on the RTCC pin reads back the written data word. You can simply write and read the data and then immediately proceed to the next / skip n memory location by generating another pulse/pulses on the OSC1 pin.

Reading and Writing the FUSEX and DEVICE Words

The DEVICE word and FUSEX word are not memory-mapped in the program address space. They can be accessed by using the following special-purpose programming commands:

- Read DEVICE Word, code 001h
- Read FUSEX Word, code 002h
- Program FUSEX Word in SX28 device, code 003h
- Program FUSEX Word in SX18 device, code 020h

The procedures for reading and writing these registers is the same as for reading or writing the main program memory, except for the command code and OSC1 signal. Because you access only one register with each command, you just leave the OSC1 pin low during the read or write procedure.

Sales and Tech Support Contact Information

DEVELOPMENT TOOLS

Parallax Inc.
3805 Atherton Road
Rocklin, CA 95765
(916) 624-8333
(916) 624-8003 (fax)
http://www.parallaxinc.com

PRODUCT DISTRIBUTION

EBV Electronics, Inc.
9980 Huennekens Street
San Diego, CA 92121
(800) 677-5664
(800) 556-0225 TECHNICAL SUPPORT HOTLINE
(619) 677-7950 (fax)

MANUFACTURERS REPRESENTATIVES

Ciponic
Room 1608, 16/F
Profit Industrial building
1-15 Kwai Fung St.
Hong Kong
011-2482-2981
011-852-2419-7874 (fax)

Com-File
30-1 Singae-Dong Yougsan-Gu
Seoul, Korea
011-822-711-2592
011-82-2-711-2593 (fax)

Com-Tek Sales
3502 Shoreline Drive
Navarre, MN 55392-0017
(612) 471-7181
(612) 471-7238 (fax)

Dynamic Technical Sales
416 E. State Parkway,
Suite 212
Schaumburg, IL 60173
(847) 755-5490
(847) 755-0532 (fax)

553 Industrial Drive, Suite 6
Hartland, WI 53029
(414) 367-1821
(414) 367-1930 (fax)

M-Rep
12801 Stemmons Freeway,
Suite 825
Dallas, TX 75234
(972) 484-5711
(972) 484-0634 (fax)

Nelco
9725 E. Hampden Avenue,
Suite 100
Denver, CO 80231
(303) 671-7677
(303) 671-7994 (fax)

6032 Caprock Court,
Apt. 1807
El Paso, TX 79912
(915) 833-7300
(915) 833-7300 (fax)

Optima Sales, Inc.
901 Campisi Way
Campbell, CA 95008
(408) 558-0655
(408) 558-0658 (fax)

Quality Components, Inc.
116 Fayette Street
Manlius, NY 13104
(315) 682-8885
(315) 682-2277 (fax)

ERA
354 Veterans Memorial High-way
Commack, NY 11725
(516) 543-0510
(516) 543-0758 (fax)

Schillinger Associates, Inc.
2297 E. Boulevard
Kokomo, IN 46902
(765) 457-7241
(765) 457-7732 (fax)

Spectrum Sales
31368 Via Colinas, Suite 101
Westlake Village, CA 91362
(818) 706-2919
(818) 706-2978 (fax)

30 Fairbanks, Suite 115
Irvine, CA 92618
(949) 461-5280

STG
101 Washington Street, Suite 6
Huntsville, AL 35801
(205) 534-2376
(205) 534-2384 (fax)

207 New Edition Court
Cary, NC 27511
(919) 468-1524
(919) 468-1534 (fax)

6045 Atlantic Boulevard
Norcross, GA 30071
(770) 239-7576
(770) 239-7589 (fax)

Sumisho Electronic Devices Corp.
1 Kandamitoshirocho
Chiyoda-ku, Tokyo, 101 Japan
(03) 5282-7225
(03) 5282-7235 (fax)

Venture
12503 Bel-Red Road,
Suite 101
Bellevue, WA 98005
(425) 454-4594
(425) 454-9003 (fax)
VentureWA@aol.com

7165 S.W. Fir Loop, Suite 103
Portland, OR 97223
(503) 624-0617
(503) 620-4682
VentureOR@aol.com

VISTAssociates, Inc.
237 Cedar Hill Street
Marlbourgh, MA 01752
(508) 481-9277
(508) 460-1869 (fax)

For the latest contact and support information on SX devices, please visit the Scenix Semiconductor website at www.scenix.com. The site contains technical literature, local sales contacts, tech support and many other features.

Scenix Semiconductor, Inc. **3160 De La Cruz Blvd., Suite #200**
Santa Clara, CA 95054
(408) 327-8888
http://www.scenix.com