TEXAS INSTRUMENTS

MSC1210 Analog-to-Digital Converter

with 8051 Microcontroller and Flash Memory

User's Guide

December 2002

Mailing Address:

Texas Instruments
Post Office Box 655303
Dallas, Texas 75265

Contents

Figures

Tables

Chapter 1

Introduction to the MSC1210

This chapter describes the basic function of the MSC1210 analog-to-digital converter (ADC).

Topic **Page**

1.1 MSC1210 Description

The MicroSystem family of devices is designed for high-resolution measurement applications in smart transmitters, industrial process control, weigh scales, chromatography, and portable instrumentation. They provide high-performance mixed signal solutions. The MicroSystem family not only includes high-end analog features and digital processing capability, but also integrates high-performance peripherals to offer a unique system solution.

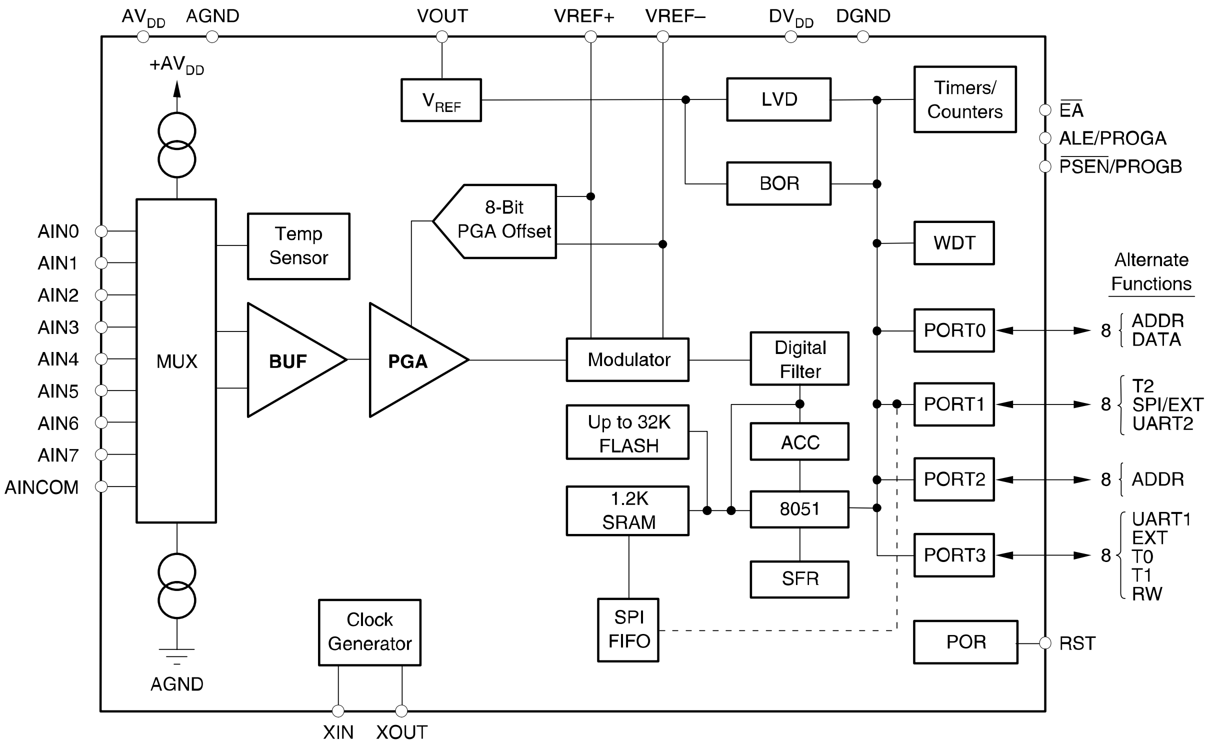
The main components of a MicroSystem product include:

- Enhanced 8051 microcontroller core
- FLASH memory
- High-performance analog functions
- High-performance peripherals

The enhanced 8052 microcontroller core includes dual data pointers and executes instructions three times faster than the standard 8052 core. This MIPS capability allows you to optimize speed, power, and noise tradeoffs based on specific requirements.

A block diagram of the MSC1210 ADC is shown in Figure 1–1.

Figure 1–1. MSC1210 Block Diagram



The on-chip FLASH memory is programmable in a variety of modes over a wide temperature and operating voltage range. This greatly simplifies programming at both the manufacturing level and in the field.

The on-chip high-performance analog features are state-of-the-art. The performance and features of the analog functions rival the best of the industry. The low-noise ADC and the precision voltage reference along with the integration of other analog features greatly simplify achieving high-end analog performance.

The on-chip high-performance peripherals not only reduce the cost, design time, and board space required for external circuitry, but also blend analog and digital functions that simplify the system design. The high-performance peripherals are designed from a system perspective, thereby decreasing the processing requirements on the CPU and providing greater system throughput.

1.2 MSC1210 Pin-Out

The names and functions of these pins are similar to those found on a traditional 8052 core, but the MSC1210 includes additional pin assignments to support the additional functions specific to the part.

Figure 1–2. Pin Configuration of the MSC1210

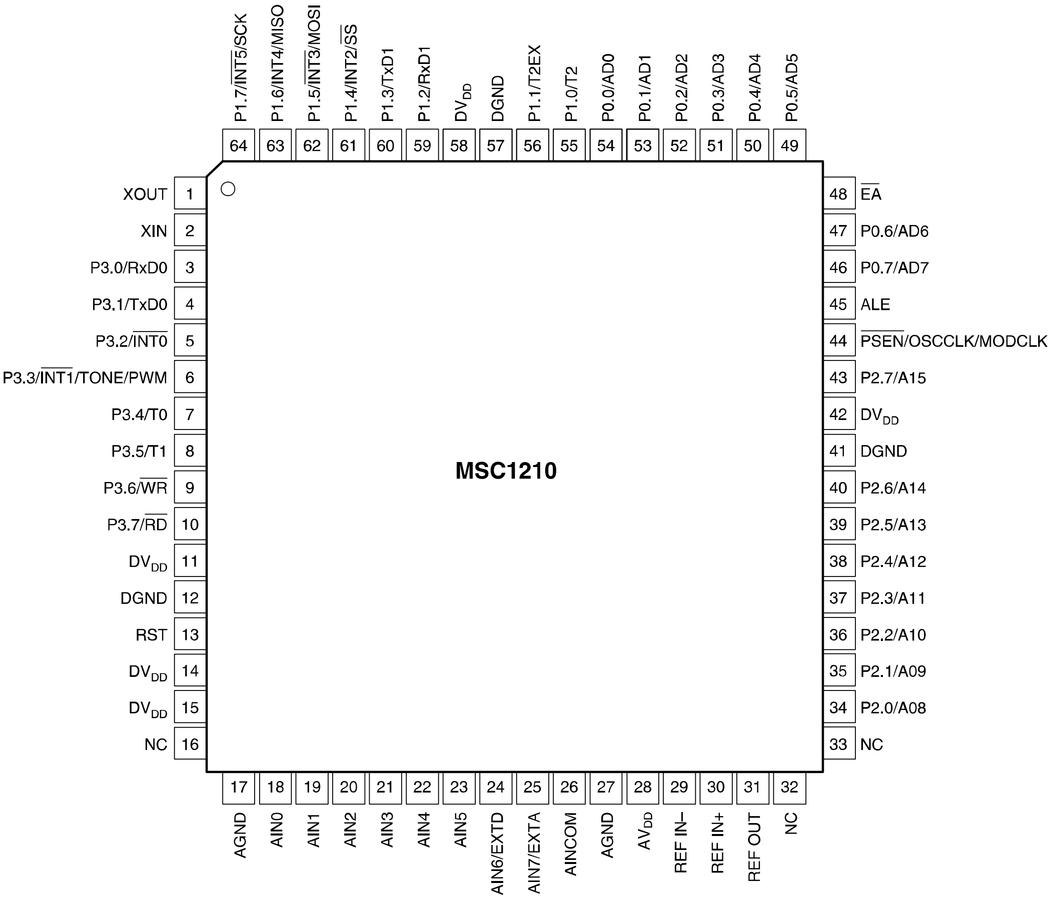


Table 1–1. Pin Descriptions of the MSC1210

Pin #	Name	Description		
1	XOUT	The crystal oscillator pin XOUT supports parallel resonant AT cut crystals and ceramic resonators. XOUT serves as the output of the crystal amplifier.		
2	XIN	The crystal oscillator pin XIN supports parallel resonant AT cut crystals and ceramic resonators. XIN can also be an input if there is an external clock source instead of a crystal.		
3-10	P3.0-P3.7	Port 3 is a bidirectional I/O port. The alternate functions for Port3 are listed below. Port 3—Alternate Functions:		
		PORT	ALTERNATE	MODE
		P3.0	RxD0	Serial Port 0 Input
		P3.1	TxD0	Serial Port 0 Output
		P3.2	$\overline{\text{INT0}}$	External Interrupt 0
		P3.3	$\overline{\text{INT1}}$/TONE/PWM	External Interrupt 1/TONE/PWM Output
		P3.4	T0	Timer 0 External Input
		P3.5	T1	Timer 1 External Input
		P3.6	$\overline{\text{WR}}$	External Data Memory Write Strobe
		P3.7	$\overline{\text{RD}}$	External Data Memory Read Strobe
11, 14, 15, 42, 58	DV_{DD}	Digital Power Supply		
12, 41, 57	DGND	Digital Ground		
13	RST	A HIGH on the reset input for two instruction clock cycles will reset the device.		
16, 32, 33	NC	No Connection		
17, 27	AGND	Analog Ground		
28	AV_{DD}	Analog Power Supply		
18	AIN0	Analog Input Channel 0		
19	AIN1	Analog Input Channel 1		
20	AIN2	Analog Input Channel 2		
21	AIN3	Analog Input Channel 3		
22	AIN4	Analog Input Channel 4		
23	AIN5	Analog Input Channel 5		
24	AIN6, EXTD	Analog Input Channel 6, Digital Low Voltage Detect Input		
25	AIN7, EXTA	Analog Input Channel 7, Analog Low Voltage Detect Input		
26	AINCOM	Analog Common for Single–Ended Inputs		
29	REF IN–	Voltage Reference Negative Input		
30	REF IN+	Voltage Reference Positive Input		
31	REF OUT	Voltage Reference Output		

Table 1–1 Pin Descriptions of the MSC1210 (Continued)

Pin #	**Name**	**Description**		
34-40, 43	P2.0-P2.7	Port 2 is a bidirectional I/O port. The alternate functions for Port 2 are listed below. Port 2—Alternate Functions:		
34-40, 43	P2.0-P2.7	PORT	ALTERNATE	MODE
		P2.0	A8	Address Bit 8
		P2.1	A9	Address Bit 9
		P2.2	A10	Address Bit 10
		P2.3	A11	Address Bit 11
		P2.4	A12	Address Bit 12
		P2.5	A13	Address Bit 13
		P2.6	A14	Address Bit 14
		P2.7	A15	Address Bit 15
44	PSEN, OSCCLK, MODCLK	Program Store Enable: Connected to optional external memory as a chip enable. $\overline{\text{PSEN}}$ will provide an active low pulse. In programming mode, $\overline{\text{PSEN}}$ is used as an input along with ALE to define serial or parallel programming mode. $\overline{\text{PSEN}}$ is held HIGH for parallel programming and tied LOW for serial programming. This pin can also be selected (when not using external program memory) to output the Oscillator clock, Modulator clock, HIGH, or LOW.		
		ALE	$\overline{\text{PSEN}}$	Program Mode Selection
		NC	NC	Normal Operation
		0	1	Parallel Programming
		1	0	Serial Programming
		0	0	Reserved
45	ALE	Address Latch Enable: Used for latching the low byte of the address during an access to external memory. ALE is emitted at a constant rate of 1/2 the oscillator frequency, and can be used for external timing or clocking. One ALE pulse is skipped during each access to external data memory. In programming mode, ALE is used as an input along with $\overline{\text{PSEN}}$ to define serial or parallel programming mode. ALE is held HIGH for serial programming and tied LOW for parallel programming.		
48	$\overline{\text{EA}}$	External Access Enable: $\overline{\text{EA}}$ must be externally held LOW to enable the device to fetch code from external program memory locations starting with 0000_H.		
46, 47, 49-54	P0.0–P0.7	Port 0 is a bidirectional I/O port. The alternate functions for Port 0 are listed below. Port 0—Alternate Functions:		
		PORT	ALTERNATE	MODE
		P0.0	AD0	Address/Data Bit 0
		P0.1	AD1	Address/Data Bit 1
		P0.2	AD2	Address/Data Bit 2
		P0.3	AD3	Address/Data Bit 3
		P0.4	AD4	Address/Data Bit 4

Table 1–1 Pin Descriptions of the MSC1210 (Continued)

Pin #	Name	Description		
46, 47, 49-54	P0.0–P0.7	P0.5	AD5	Address/Data Bit 5
		P0.6	AD6	Address/Data Bit 6
		P0.7	AD7	Address/Data Bit 7
55, 56, 59–64	P1.0–P1.7	Port 1 is a bidirectional I/O port. The alternate functions for Port 1 are listed below. Port 1—Alternate Functions:		
		PORT	ALTERNATE	MODE
		P1.0	T2	T2 Input
		P1.1	T2EX	T2 External Input
		P1.2	RxD1	Serial Port Input
		P1.3	TxD1	Serial Port Output
		P1.4	INT2/$\overline{\text{SS}}$	External Interrupt/Slave Select
		P1.5	$\overline{\text{INT3}}$/MOSI	External Interrupt/Master Out–Slave In
		P1.6	INT4/MISO	External Interrupt/Master In–Slave Out
		P1.7	$\overline{\text{INT5}}$/SCK	External Interrupt/Serial Clock

1.2.1 I/O Ports (P0, P1, P2, and P3)

Of the 64 pins on the MSC1210, 32 of them are dedicated to I/O lines that have a one-to-one relation with SFRs P0, P1, P2, and P3. The developer may raise and lower these lines by writing 1s or 0s to the corresponding bits in the SFRs. Likewise, the current state of these lines may be found by reading the corresponding bits of the SFRs.

All of the ports have optional pull-up resistors that are enabled when the port is in 8051 mode, as configured by the PxDDRL/H SFRs. The pull-up resistors are disabled when the port is configured in any other mode, or when accessing external memory.

1.2.1.1 Port 0

Port 0 is dual-function: in some designs port 0 I/O lines are available to the developer to access external devices, while in other designs it is used to access external memory. If the circuit requires external RAM, the microcontroller will use port 0 to latch in/out the 8-bit data word, as well as the low eight bits of the address in response to a MOVX instruction, as long as the hardware configuration registers are set up correctly. Port 0 I/O lines may be used for other functions as long as external data memory is not being accessed at the same time and the hardware configuration registers are set up correctly. If the circuit requires external code memory, the microcontroller will use port 0 I/O lines to access each instruction to be executed. In this case, port 0 cannot be used for other purposes, because the state of the I/O lines are constantly being modified to access external code memory.

1.2.1.2 Port 1

Port 1 consists of eight I/O lines that may be used to interface to external parts. Port 1 is commonly used to interface to external hardware such as LCDs, keypads, and other devices. As opposed to a standard 8052 core, all I/O lines of the MSC1210 serve optional alternate functions, as described below. These lines can still be used for the developing purposes, if the functions described below are not needed.

P1.0 (T2): If T2CON.1 is set (C/$\overline{\text{T}}$2), then timer 2 is incremented whenever there is a 1-0 transition on this line. With C/$\overline{\text{T}}$2 set, P1.0 is the clock source for timer 2.

P1.1 (T2EX): If timer 2 is in auto-reload mode and T2CON.3 (EXEN2) is set, a 1-0 transition on this line causes timer 2 to be reloaded with the auto-reload value. This also causes the T2CON.6 (EXF2) external flag to be set, which may cause an interrupt, if so enabled.

P1.2 (RxD1): If the secondary USART is being used, P1.2 (RxD1) is the pin that receives serial data. Data received via this pin is read using the SBUF1 SFR.

P1.3 (TxD1): If the secondary USART is being used, P1.3 (TxD1) is the pin that transmits serial data. Data written to the SBUF1 SFR is sent via this pin.

P1.4 (INT2/$\overline{\textbf{SS}}$): This pin has two dual functions. It may be used to trigger an external 2 interrupt when a 0-1 transition is detected on this line. It is also used as slave select in SPI applications.

P1.5 ($\overline{\textbf{INT3}}$/MOSI): This pin may be used to trigger an external 3 interrupt when a 1-0 transition is detected. It is also used as Master Out/Slave In in SPI applications.

P1.6 (INT4/MISO): This pin may be used to trigger an external 4 interrupt when a 0-1 transition is detected. It is also used as Master In/Slave Out in SPI applications.

P1.7 (INT5/SCK): This pin may be used to trigger an external 5 interrupt when a 1-0 transition is detected. It is also used as serial clock in SPI applications.

1.2.1.3 Port 2

Like port 0, port 2 is dual-function. In some circuit designs, it is available for accessing external devices, while in others it is used to address external RAM or external code memory. When more than 256 bytes of external RAM are used, port 2 is used to output the high byte of the address that is to be accessed in a MOVX operation. Whether port 2 is used to address external memory or as general I/O lines is defined by the EGP23 bit in hardware configuration Register 1.

Note:

When the EGP23 bit of hardware configuration Register 1 is set, Port 2 assumes the value of the high byte of DPTR when using the MOVX @DPTR instruction. When using the MOVX @Rx instructions, port 2 assumes the value of the MPAGE SFR.

If the circuit requires external code memory, the microcontroller automatically uses port 2 I/O lines to access each instruction to be executed, but only if bit EGP23 of HCR1 equals one. In this case, port 2 cannot be used for other purposes because the state of the I/O lines are constantly being modified to access external code memory.

1.2.1.4 Port 3

Port 3 consists entirely of dual-function I/O lines. While you can access all these lines from the software by reading/writing to the P3 SFR, each pin has a predefined function that the microcontroller handles automatically when configured to do so and/or when necessary.

P3.0 (RxD0): The primary USART/serial port uses P3.0 as the receive line. For in-circuit designs that are using the microcontroller internal serial port, this is the line into which the serial data is clocked.

Note:

When interfacing an 8052 to an RS-232 port, you cannot connect this line directly to the RS-232 pin; you must pass it through a part such as the MAX233 to obtain the correct voltage levels.

You can assign any function to this pin as long as the circuit has no need to receive data via the integrated serial port.

P3.1 (TxD0): The primary USART/serial port uses P3.1 as the transmit line. For in-circuit designs that is using the microcontroller internal serial port, this is the line used by the microcontroller to clock out all data written to the SBUF SFR.

Note:

When interfacing an 8052 to an RS-232 port, you cannot connect this line directly to the RS-232 pin; you must pass it through a part such as the MAX233 to obtain the correct voltage levels.

You can assign any function to this pin as long as the circuit has no need to transmit data via the integrated serial port.

P3.2 ($\overline{\text{INT0}}$): When so configured, this line is used to trigger an external 0 Interrupt. This may either be low-level triggered or may be triggered on a 1-0 transition (see Chapter 10, *Interrupts,* for details). You can assign any function to this pin as long as the circuit has no need to trigger an external 0 interrupt.

P3.3 ($\overline{\text{INT1}}$/TONE/PWM): When so configured, this line is used to trigger an external 1 Interrupt. This may either be low-level triggered or may be triggered on a 1-0 transition (see Chapter 10, *Interrupts,* for details). This pin is also used for outputting PWM, if so configured.

P3.4 (T0): When so configured, this line is used as the clock source for timer 0. Timer 0 is incremented either every instruction cycle that T0 is high, or every time there is a 1-0 transition on this line, depending on how the timer is configured (see Chapter 8, *Timers*, for details). You can assign any function to this pin as long as the circuit has no need to control timer 0 externally.

P3.5 (T1): When so configured, this line is used as the clock source for timer 1. Timer 1 is incremented either every instruction cycle that T1 is high, or every time there is a 1-0 transition on this line, depending on how the timer is configured (see Chapter 8, *Timers*, for details). You can assign any function to this pin as long as the circuit has no need to control timer 1 externally.

P3.6 ($\overline{\text{WR}}$): This is the external memory write strobe line when bit EGP23 is set in hardware configuration Register 1. This line is asserted low by the microcontroller whenever a MOVX instruction writes to external RAM. This line should be connected to the RAM write ($\overline{\text{W}}$) line. You can assign any function to this pin as long as the circuit does not write to external RAM using MOVX.

P3.7 ($\overline{\text{RD}}$): This is the external memory read strobe line when bit EGP23 is set in hardware configuration Register 1. This line is asserted low by the microcontroller whenever a MOVX instruction is read from external RAM. This line must be connected to the RAM read ($\overline{\text{R}}$) line. You can assign any function to this pin as long as the circuit does not read from external RAM using MOVX.

1.2.2 Oscillator Inputs (XTAL1 and XTAL2)

The MSC1210 is typically driven by a crystal connected to pins 1 (XOUT) and 2 (XIN). Common crystal frequencies are 11.0592MHz as well as 12MHz, although the MSC1210 is capable of accepting frequencies as high as 33MHz.

While a crystal is the normal clock source, this is not always the case. A digital clock source may also be attached to XIN and XOUT to provide the clock for the microcontroller.

1.2.3 Reset Line (RST)

Pin 13 is the master reset line for the microcontroller. When this pin is brought high for two instruction cycles, the microcontroller is effectively reset. SFRs, including the I/O ports, are restored to their default conditions and the program counter is reset to 0000_H. Keep in mind that Internal RAM is *not* affected by a reset. The microcontroller begins executing code at 0000_H when pin 13 returns to a low state.

The reset line is often connected to a reset button/switch that you can press to reset the circuit. It is also common to connect the reset line to a watchdog IC or a supervisor IC (such as MAX707). Traditional resistor-capacitor networks attached to the reset line also work well because the RST input is a Schmitt trigger input.

1.2.4 Address Latch Enable (ALE)

The ALE at pin 45 is an output-only pin that is controlled entirely by the microcontroller and allows the microcontroller to multiplex the low-byte of a memory address and the 8-bit data itself on port 0. This is because, while the high byte of the memory address is sent on port 2, port 0 is used both to send the low byte of the memory address and the data itself. This is accomplished by placing the low byte of the address on port 0, exerting an ALE high-to-low transition to latch the low byte of the address into a latch IC (such as the 74HC573), and then placing the 8 data bits on port 0. In this way, the MSC1210 is able to output a 16-bit address and an 8-bit data word with 16 I/O lines instead of 24.

The ALE line is used in this fashion both to access external RAM with MOVX @DPTR, as well as to accessi instructions in external code memory. When the program is executed from external code memory, ALE pulses at a rate that is ¼ that of the oscillator frequency. Thus, if the oscillator operates at 11.0592MHz, ALE pulses at a rate of 2 764 800 times per second. When the MOVX instruction is executed, one $\overline{\text{PSEN}}$ pulse is missed in lieu of a pulse on $\overline{\text{WR}}$ or $\overline{\text{RD}}$.

This pin is also used when programming the part, along with $\overline{\text{PSEN}}$, as an input during reset to indicate whether programming will occur in serial or parallel mode. If this line is held high when in programming mode, programming will occur in serial mode.

1.2.5 Program Store Enable ($\overline{\text{PSEN}}$)

The program store enable ($\overline{\text{PSEN}}$) line at pin 44 is exerted low automatically by the microcontroller whenever it accesses external code memory. This line should be attached to the output enable ($\overline{\text{OE}}$) pin of the device that contains your code memory. The $\overline{\text{PSEN}}$ signal is applied for both internal and external memory access.

This pin is also used when programming the part, along with ALE, as an input to indicate whether programming will occur in serial or parallel mode. If this line is held high when in programming mode, programming will occur in parallel mode.

1.2.6 External Access ($\overline{\text{EA}}$)

The external access ($\overline{\text{EA}}$) line at pin 48 is used to determine whether the MSC1210 will execute your program from external code memory or from internal code memory. If $\overline{\text{EA}}$ is tied high (connected to supply), the microcontroller will execute the program it finds in internal/on-chip code memory. If $\overline{\text{EA}}$ is tied low (to ground), it will attempt to execute the program that it finds in the attached external program memory. Of course, the external program memory must be properly connected for the microcontroller to be able to access the program in external program memory.

The $\overline{\text{EA}}$ pin is ignored during serial or parallel flash programming modes.

Note:

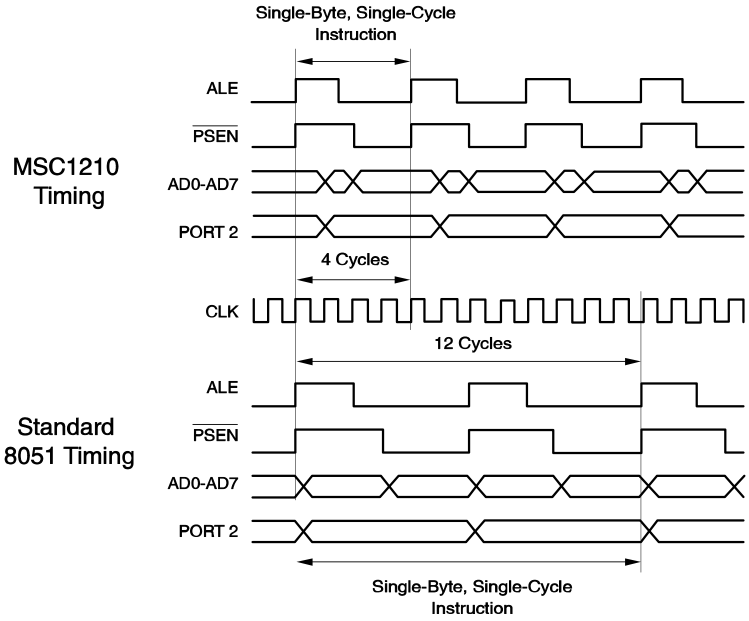
Even when $\overline{\text{EA}}$ is tied high (indicating that the microcontroller should execute from internal code memory), the microcontroller will attempt to execute from external code memory if the program counter references an address not available for the chip you are using, or if you are accessing program memory in excess of the amount of flash memory that you have partitioned for program memory. For example, if you have partitioned 4k of flash memory to be program memory and you tie $\overline{\text{EA}}$ high, the derivative starts executing the program it finds on-chip. However, if your on-chip program attempts to execute code above $0FFF_H$ (that is, exceeding 4k), then the MSC1210 will attempt to execute that code at that address from external code memory. Thus, it is possible to have a split design, in which some of the code is found on-chip and the rest is found off-chip.

1.3 Enhanced 8051 Core

The MSC1210 is an 8052-based family of high-performance, mixed-signal controllers. All instructions in the MSC1210 family perform exactly the same function as they would in a standard 8052 core. Although the effect on bits, flags, and registers is the same, the timing is different.

The MSC1210 family uses an efficient 8052 core that results in an improved instruction execution speed of three times faster than the original core for the same external clock speed (4 clock cycles per instruction versus 12 clock cycles per instruction, as shown in Figure 1–3). This allows you to run the device at slower external clock speeds, which reduces system noise and power consumption, but provides greater throughput.

Figure 1–3. MSC1210 Timing Compared to Standard 8051 Timing



The timing of software loops is faster with the MSC1210 than with the standard 8052. However, the timer/counter operation of the MSC1210 may be maintained at 12 clocks per increment or optionally run at 4 clocks per increment.

You can develop software for the MSC1210 with the existing 8052 development tools because the MSC1210 is fully compatible with the standard 8052 instruction set. Additionally, a complete integrated development environment is provided with each demonstration board.

1.4 Family Device Compatibility

The hardware functionality and pin outs across the MSC1210 family are fully compatible. The only difference between family members is the memory configuration and this enables simple migration between family members. Code written for the 4K bytes program memory version of the MSC1210 can be executed directly on the 8K, 16K, or 32K versions. This allows you to add or delete software functions and to freely migrate between family members.

The MSC1210 can become a standard device used across several application platforms.

1.5 Flash Memory

The MSC1210 features flexible flash memory that allows you to uniquely configure the program and non-volatile data memory maps to meet the needs of the application. The flash memory is programmable over the entire operating voltage range and temperature range using both serial and parallel programming methods.

1.6 High Performance Analog Functions

The analog functionality is state-of-the-art. The ADC is extremely low noise, which enables you to meet even the most stringent analog requirements. The integrated programmable gain amplifier (PGA) further improves the performance of the ADC. This effectively provides for resolution into the nanovolt range.

The on-chip voltage reference provides for low drift and high accuracy, thus eliminating the need for an external voltage reference.

These features are integrated with other analog functions, such as a programmable filter, multiplexer, temperature sensor, burnout current sources, analog input buffer, and an offset correction digital-to-analog converter (DAC).

1.7 High-Performance Peripherals

High-performance peripherals are included on-chip, which offload CPU processing and control functions from the core to further improve the overall device efficiency and throughput. On-chip peripherals include additional SRAM, a 32-bit accumulator, an SPI-compatible serial port with a FIFO buffer, dual USARTs, on-chip power-on reset, brownout reset, low-voltage detect, multiple digital ports with configurable I/O, a 16-bit pulse width modulator (PWM), a watchdog timer, and three timer/counters.

For instance, the SPI interface uses a FIFO buffer, which allows for the serial transmission and reception of data with virtually no CPU overhead. The FIFO buffer function allows for the transfer of large amounts of data at faster transfer rates than more conventional methods.

Additionally, the 32-bit accumulator significantly reduces the processing overhead for the multiple byte data from the ADC or other sources. This allows for 24-bit addition, subtraction, and shifting to be accomplished without using CPU resources. This can reduce both the code size and code execution time.

Chapter 2

MSC1210 Memory Organization

This chapter defines the Memory Organization of MSC1210 ADC.

Topic **Page**

2.1 Description

The MCS1210 has three very general types of memory. To program the MCS1210 effectively, it is necessary to have a basic understanding of these memory types:

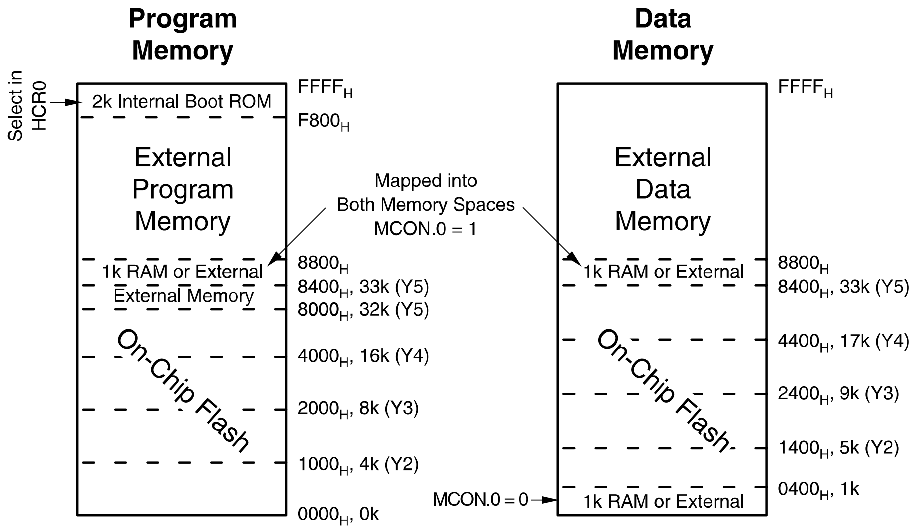
- **Special Function Registers** refer to 128 bytes that control the operation of the MSC1210.
- **Program Memory** is used to store the actual program that may reside on-chip, off-chip, or both.
- **Data Memory** is static random access memory (SRAM) that can reside on-chip, off-chip, or both. The MSC1210 has four types of data memory:
  - On-chip extended SRAM
  - Off-chip external SRAM
  - On-chip Flash Data memory
  - Internal RAM

2.2 Program Memory

Program memory holds the actual program that is to be run. This memory includes the on-chip flash memory designated as program memory and/or external memory.

The MSC1210 family offers a maximum of 32k of on-chip flash program memory. The exact amount of on-chip program memory depends on the specific MSC1210 version selected and how the flash memory of that chip has been partitioned between program and data memory. Figure 2–1 illustrates how the flash memory may be distributed between these two types of memory.

Figure 2–1. MSC1210 Memory Map



For example, in the Y5 model there is 32k flash memory available. This 32k may be configured as either program memory, data memory, or both. This configuration is set at the moment the firmware is loaded onto the MSC1210 by setting hardware configuration register HCR0 as per Table 2–1. This table indicates the total amount of program and data memory available for each part revision given a specific HCR0 setting.

Table 2–1. Program and Data Memory Size.

HCR0	**MSC1210Y2**		**MSC1210Y3**		**MSC1210Y4**		**MSC1210Y5**	
DFSEL	PM	DM	PM	DM	PM	DM	PM	DM
000	0kB	4kB	0kB	8kB	0kB	16kB	0kB	32kB
001	0kB	4kB	0kB	8kB	0kB	16kB	0kB	32kB
010	0kB	4kB	0kB	8kB	0kB	16kB	16kB	16kB
011	0kB	4kB	0kB	8kB	8kB	8kB	24kB	8kB
100	0kB	4kB	4kB	4kB	12kB	4kB	28kB	4kB
101	2kB	2kB	6kB	2kB	14kB	2kB	30kB	2kB
110	3kB	1kB	7kB	1kB	15kB	1kB	31kB	1kB
111 (default)	4kB	0kB	8kB	0kB	16kB	0kB	32kB	0kB

Note: When a 0kB program memory configuration is selected, program execution is external

For example, setting the DFSEL bits to 110 with a MSC1210Y5 would cause 31kb of on-chip flash memory to be partitioned as program memory and 1kb of flash memory to be partitioned as data memory.

Table 2–2 indicates where the assigned memory will be located in address space. This table provides essentially the same information as Table 2–1, but also indicates where the memory will be located. For example, the DFSEL = 110 example in the previous paragraph (31kb of on-chip flash program memory, 1k of on-chip flash data memory) appears in Table 2–2 as flash program memory from 0000_H to $7BFF_H$ (which is 31k) and flash data memory from 0400_H to $07FF_H$ (which is 1k).

Note that the Data memory address starts at 0400_H because the first 1k (0000_H-$03FF_H$) is, by default, used to address the on-chip extended SRAM. The location of on-chip extended SRAM may be changed by using the Memory Control (MCON) SFR. By setting bit 0 of MCON, the on-chip extended SRAM may be moved from 0000_H-$03FF_H$ to 8400_H-$87FF_H$. However, on-chip extended flash data memory always begins at 0400_H regardless of whether or not SRAM is located at 0000_H or 8400_H.

Table 2–2. Program and Data Memory Addresses.

HCR0	**MSC1210Y2**		**MSC1210Y3**		**MSC1210Y4**		**MSC1210Y5**	
DFSEL	PM	DM	PM	DM	PM	DM	PM	DM
000 (reserved)	—	—	—	—	—	—	—	—
001	—	—	—	—	—	—	0000	0400-83FF
010	—	—	—	—	0000	0400-43FF	0000-3FFF	0400-43FF
011	—	—	0000	0400-23FF	0000-1FFF	0400-23FF	0000-5FFF	0400-23FF
100	0000	0400-13FF	0000-0FFF	0400-13FF	0000-2FFF	0400-13FF	0000-6FFF	0400-13FF
101	0000-07FF	0400-00BF	0000-17FF	0400-0BFF	0000-37FF	0400-0BFF	0000-77FF	0400-0BFF
110	0000-00BF	0400-07FF	0000-1BFF	0400-07FF	0000-3BFF	0400-07FF	0000-7BFF	0400-07FF
111 (default)	0000-0FFF	0000	0000-1FFF	0000	0000-3FFF	0000	0000-7FFF	0000

Note: Program accesses above the highest listed address will access external Program memory.

Program memory addressing beyond the on-chip address range is accessed externally via ports 0 and 2. The total amount of code memory, on-chip and off, is limited to 64k due to limitations of the 8052 architecture.

Note:

MSC1210 programs are limited to 64k because code memory is restricted to 64k. Some compilers offer ways to get around this limit when used with specially wired hardware. However, without such special compilers and hardware, programs are limited to 64k.

The MSC1210 includes 2k of boot ROM code that controls operation during serial or parallel programming. In program mode, the boot ROM is located in the first 2kB of program memory.

The boot ROM is available to your program as long as EBR (hardware configuration register 0, bit 4) is set, which is the default. When enabled, the boot ROM routines will be located at program memory addresses $F800_H$-$FFFF_H$. The boot ROM includes a number of functions such as flash memory access, and serial routines including data transmission, reception, and auto-baud.

2.3 Data Memory

Data memory is divided into four types of memory, depending on its location and volatility: internal RAM, on-chip extended SRAM, off-chip external SRAM, and on-chip flash data memory. However, data memory (regardless of its location or volatility) is accessed using the MOVX instruction, except for internal RAM, which is accessed using the MOV instruction.

2.3.1 On-Chip Extended Static RAM (SRAM)

The MSC1210 includes 1024 bytes of on-chip extended static RAM (SRAM). Even though this memory resides on-chip, it is accessed using the MOVX instruction as if it were external data memory. Whenever a program accesses data memory addresses 0000_H through $03FF_H$, the on-chip external SRAM is used.

On-chip extended static RAM provides 1k of data memory that requires no external circuitry and is available regardless of how the MSC1210's flash memory is designated. This makes it a convenient memory area for purposes such as temporary buffers, calculation scratchpads, or any other purpose that requires 1k or less of memory, but does not require it to survive a power failure.

2.3.2 On-Chip Flash Data Memory

In addition to the on-chip extended SRAM described in the previous section, the MSC1210 also has the capability of offering on-chip flash data memory. Flash memory is slower than SRAM, but has the advantage of being nonvolatile: its contents will not be lost when the power source is removed.

All of the parts in the MSC1210 family come with some amount of on-chip flash memory, ranging from 4k for the MSC1210Y2 all the way up to 32k for the MSC1210Y5. This flash memory may be configured such that it can be used as either program memory, data memory, or both.

When configured as data memory, on-chip flash data memory is accessed starting at address 0400_H—immediately after on-chip SRAM.

For example, if the MSC1210Y5 is configured to use 2k as on-chip flash data memory, addresses 0000_H through $03FF_H$ will access on-chip extended SRAM while addresses 0400_H through $0BFF_H$ will access on-chip flash data memory. Any attempts to read data memory with addresses $0C00_H$ and higher will result in the part attempting to fetch that data off-chip from external data memory (see the next section), except when the internal 1kB SRAM is configured as Von Neumann type, which occupies from 8400_H~$87FF_H$.

2.3.3 External Data Memory

The MSC1210 is capable of addressing up to 64k of data memory, however, a maximum of 33k of that may be on-chip: 1k of SRAM and up to 32k of flash data memory. If additional data memory is necessary, it must be added to the circuit as external data memory.

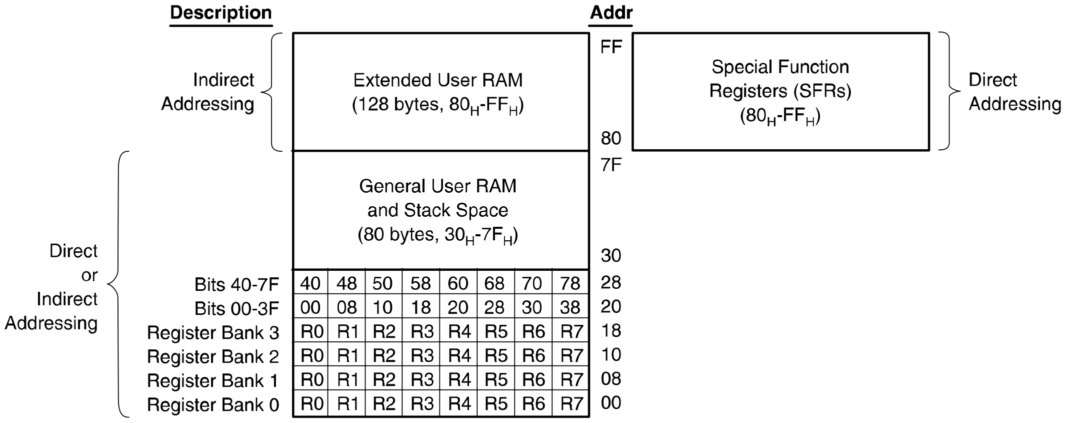
External data memory is any off-chip data memory that is connected to the MSC1210 via ports 0 and 2 and uses control pins ALE, $\overline{\text{RD}}$, and $\overline{\text{WR}}$. These two ports combined with these three control lines allow the MSC1210 to address external RAM.

External data memory can also be used to access "memory mapped devices," which are devices that appear to the MSC1210 to be external data memory but in reality are external components such as LCDs, buttons, keypads, etc.

Note:

The MSC1210 must only address 64kB of RAM. To expand RAM beyond this limit requires programming and hardware tricks. It may be necessary to do this "by hand" because many compilers and assemblers, while providing support for programmers in excess of 64kB, do not support more than 64kB of RAM. If more than 64kB of RAM is necessary, the compiler must be checked to verify that the excess RAM is supported. If not, it will be necessary to do it "by hand."

Figure 2–2. MSC1210 Memory Map Register Bank.



2.4 Internal RAM

As shown in Figure 2–2, the MSC1210 has a bank of 256 bytes of internal RAM. This internal RAM is found on-chip within the IC, so it is the fastest RAM available and is also the most flexible in terms of reading, writing, and modifying its contents. internal RAM is volatile, so when the MSC1210 is powered up, the contents of this memory bank is random.

The 256 bytes of internal RAM are subdivided as shown in the memory map of Figure 2–2. The first eight bytes (00_H–07_H) are register bank 0. By manipulating certain SFRs, a program may choose to use register banks 0, 1, 2, or 3. These alternative register banks are located in internal RAM at addresses 08_H through $1F_H$. Register banks are described in greater detail in chapters 3 and 4. For now it is sufficient to know that they reside in and are part of internal RAM.

Bit memory also resides in and is part of internal RAM. Bit memory will be described more in section 2.4.3, but for now just keep in mind that bit memory actually resides in internal RAM at addresses 20_H through $2F_H$.

The 208 bytes remaining of internal RAM, from addresses 30_H through FF_H, may be used by user variables that need to be accessed frequently or at high-speed. This area is also used by the microcontroller as a storage area for the operating stack. This fact limits the MSC1210 stack because, as illustrated in the memory map of Figure 2–2, the area reserved for the stack is only 208 bytes—and usually it is less because these 208 bytes have to be shared between the stack and user variables.

Note:

Internal RAM addresses 00_H through $7F_H$ may be accessed either via direct addressing or indirect addressing, whereas internal RAM addresses 80_H through FF_H may only be accessed via indirect addressing. This will be discussed completely in Chapter 5, *Addressing Modes*.

2.4.1 The Stack

The stack is a “last in, first out” (LIFO) storage area that exists in internal RAM. It is used by the MSC1210 to store values that the user program manually pushes onto the stack, as well as to store the return addresses for CALLs and interrupt service routines (ISRs)—more on these topics later.

The stack is defined and controlled by an SFR called SP. As a standard 8–bit SFR, SP holds a value between 0 and 255 that represents the internal RAM address of the *end* of the current stack. If a value is removed from the stack, it is taken from the internal RAM address pointed to by SP, and SP will subsequently be decremented by 1. If a value is pushed onto the stack, SP is first incremented and then the value is inserted in internal RAM at the address now pointed to by SP.

SP is initialized to 07_H when the MSC1210 is first powered up. This means the first value to be pushed onto the stack is placed at internal RAM address 08_H (07_H + 1), the second is placed at 09_H, etc.

Note:

By default, the MSC1210 initializes the stack pointer (SP) to 07_H when the microcontroller is reset. This means that the stack will start at address 08_H and expand upwards. If using the alternate register banks (banks 1, 2, or 3) the stack pointer must be initialized to an address above the highest register bank being used. Otherwise, the stack will overwrite the alternate register banks. Similarly, if using bit variables, it is usually a good idea to initialize the stack pointer to some value greater than $2F_H$ to ensure that the bit variables are protected from the stack. Following is more information about the register banks and bit memory.

2.4.2 Register Banks

The MSC1210 uses eight R registers, which are used in many of its instructions. These R registers are numbered from 0 through 7 (R0, R1, R2, R3, R4, R5, R6, and R7) and are generally used to assist in manipulating values and moving data from one memory location to another. For example, to add the value of R4 to the accumulator, the following assembly language instruction would be executed:

```
ADD A,R4
```

Thus, if the accumulator (A) contains the value 6, and R4 contains the value 3, the accumulator will contain the value 9 after this instruction is executed.

However, as the memory map of Figure 2–2 illustrates, R Register R4 is really part of internal RAM. Specifically, R4 is address 04_H of internal RAM. This can be seen in the bright green section of the memory map. The above instruction, therefore, accomplishes the same thing as the following operation:

```
ADD A,04h
```

This instruction adds the value found in internal RAM address 04_H to the value of the accumulator, leaving the result in the accumulator. The above instruction effectively accomplishes the same thing as the previous ADD instruction because R4 is really internal RAM address 04_H.

But watch out! As the memory map shows, the MSC1210 has four distinct register banks. When the MSC1210 is first reset, register bank 0 (addresses 00_H through 07_H) is used by default. However, the MSC1210 may be instructed to use one of the alternate register banks (i.e., register banks 1, 2, or 3). In this case, R4 will no longer be the same as internal RAM address 04_H. For example, if the program instructs the 8052 to use register bank 1, register R4 is now synonymous with internal RAM address $0C_H$. If register bank 2 is selected, R4 is synonymous with 14_H, and if register bank 3 is selected, it is synonymous with address $1C_H$.

The concept of register banks adds a great level of flexibility to the 8052, especially when dealing with interrupts (see chapter 10, *Interrupts*, for details). However, always remember that the register banks really reside in the first 32 bytes of internal RAM.

Note:

If only the first register bank (i.e. bank 0) is used, internal RAM locations 08_H through $1F_H$ can be used by the program for its own use. If register banks 1, 2, or 3 are to be used, be very careful about using addresses below 20H to avoid overwriting the value of “R” registers from other register banks.

2.4.3 Bit Memory

The MSC1210, being a communications and control-oriented microcontroller that often has to deal with on and off situations, gives you the ability to access a number of bit variables directly with simple instructions to set, clear, and compare these bits. These variables may be either 1 or 0.

There are 128 bit variables available to the user, numbered 00_H through $7F_H$. The user may make use of these variables with commands such as SETB and CLR. For example, to set bit number 24_H (hex) to 1, the user would execute the instruction:

```
SETB 24h
```

It is important to note that Bit memory, like the register banks in section 2.4.2, is really a part of internal RAM. In fact, the 128-bit variables occupy the 16 bytes of internal RAM from 20_H through $2F_H$. Thus, if the value FF_H is written to internal RAM address $20_{H,}$ bits 00_H through 07_H have been effectively set. That is to say that the instruction:

```
MOV 20h,#0FFh
```

is equivalent to the instructions:

```
SETB 00h
SETB 01h
SETB 02h
SETB 03h
SETB 04h
SETB 05h
SETB 06h
SETB 07h
```

As shown, bit memory is not really a new type of memory, it is just a subset of internal RAM. However, because the MSC1210 provides special instructions to access these 16 bytes of memory on a bit-by-bit basis, it is useful to think of it as a separate type of memory. Always keep in mind that it is just a subset of internal RAM, and that operations performed on internal RAM can change the values of the bit variables.

Note:

If your program does not use bit variables, you may use internal RAM locations 20_H through $2F_H$ for your own use. When using bit variables, be very careful about using addresses from 20_H through $2F_H$, as you may end up overwriting the value of your bits.

Note:

By default, the MSC1210 initializes SP to 07_H when the microcontroller is booted. This means that the stack will start at address 08_H and expand upwards. If using the alternate register banks (banks 1, 2 or 3), SP must be initialized to an address above the highest register bank being used. Otherwise the stack will overwrite the alternate register banks. Similarly, if using bit variables, it is usually a good idea to initialize SP to some value greater than $2F_H$ to ensure that the bit variables are protected from the stack.

Bit memory 00_H through $7F_H$ is for user-defined functions in their programs. Bit memory 80_H and above are used to access certain SFRs (see section 2.4.4) on a bit-by-bit basis. For example, if output lines P0.0 through P0.7 are all clear (0), to turn on the P0.0 output line, either execute:

```
MOV P0,#01h
```

or execute:

```
SETB 80h
```

Both these instructions accomplish the same thing. However, using the SETB command will turn on the P0.0 line without affecting the status of any of the other P0 output lines. The MOV command effectively turns off all the other output lines that, in some cases, may not be acceptable.

When dealing with bit addresses of 80_H and above, remember that the bits refer to the bits of corresponding SFRs that are divisible by eight. This is a complicated way of saying that bits 80_H through 87_H refer to bits 0 through 7 of SFR 80_H, bits 88_H through $8F_H$ refer to bits 0 through 7 of SFR 88_H, bits 90_H through 97_H refer to bits 0 through 7 of 90_H, etc.

2.4.4 Special Function Register (SFR) Memory

SFRs are areas of memory that control specific functionality of the MSC1210. For example, four SFRs permit access to the 32 input/output lines (eight lines per SFR) of the MSC1210. Another SFR allows a program to read or write to the MSC1210 serial port. Other SFRs allow the user to set the serial baud rate, control and access timers, and configure the MSC1210 interrupt system.

When programming, SFRs have the illusion of being internal memory. For example, if writing the value 1 to internal RAM location 50_H, execute the instruction:

```
MOV 50h,#01h
```

Similarly, if writing the value 1 to the MSC1210 serial port, write this value to the SBUF SFR, which has an SFR address of 99_H. Thus, to write the value 1 to the serial port, execute the instruction:

```
MOV 99h,#01h
```

As shown, it appears as if the SFR is part of internal memory. *This is not the case*. When using this method of memory access (it is called direct addressing—more on that soon), any instruction that has an address of 00_H through $7F_H$ refers to an internal RAM memory address; any instruction with an address of 80_H through FF_H refers to an SFR control register.

Note:

SFRs are used to control the way the MSC1210 functions. Each SFR has a specific purpose and format that will be discussed later. Not all addresses above 80_H are assigned to SFRs. However, this area may *not* be used as additional RAM memory, even if a given address has not been assigned to an SFR.

Note:

Direct access addressing cannot be used to access internal RAM addresses 80_H through FF_H because direct access to addresses 80_H through FF_H refers to SFRs. The upper 128 bytes of internal RAM must be accessed using indirect addressing, which is explained in Chapter 5, *Addressing Modes*.

Chapter 3

Special Function Registers (SFRs)

Chapter 3 defines the MSC1210 SFRs.

Topic **Page**

3.1 Description

The MSC1210 is a flexible microcontroller with a relatively large number of modes of operation. Your program may inspect and/or change the operating mode of the MSC1210 by manipulating the values of its SFRs.

SFRs are accessed as if they were normal internal RAM. The only difference is that internal RAM is addressed in direct mode with addresses 00_H through $7F_H$, whereas SFR registers are accessed in the range of 80_H through FF_H.

Each SFR has an address (80_H–FF_H) and a name. Table 3–1 provides a graphical presentation of the 8052’s SFRs, their names, and their address.

Although the address range of 80_H through FF_H offers 128 possible addresses, there are 24 addresses that are not assigned to an SFR, as shown in Table 3–1.

Note:

Reading an unassigned SFR will get 00_H, and writing to an unassigned SFR is ignored.

Table 3–1. SFR Names and Addresses.

80	**P0**	SP	DPL0	DPH0	DPL1	DPH1	DPS	PCON	87
88	**TCON**	TMOD	TL0	TL1	TH0	TH1	CKCON	MWS	8F
90	**P1**	EXIF	MPAGE	CADDR	CDATA	MCON			97
98	**SCON0**	SBUF0	SPICON	SPIDATA	SPIRCON	SPITCON	SPISTART	SPIEND	9F
A0	**P2**	PWMCON	PWMLOW	PWMHI		PAI	AIE	AISTAT	A7
A8	**IE**	BPCON	BPL	BPH	P0DDRL	P0DDRH	P1DDRL	P1DDRH	AF
B0	**P3**	P2DDRL	P2DDRH	P3DDRL	P3DDRH				B7
B8	**IP**								BF
C0	**SCON1**	**SBUF1**					EWU		C7
C8	**T2CON**		RCAP2L	RCAP2H	TL2	TH2			CF
D0	**PSW**	OCL	OCM	OCH	GCL	GCM	GCH	ADMUX	D7
D8	**EICON**	ADRESL	ADRESM	ADRESH	ADCON0	ADCON1	ADCON2	ADCON3	DF
E0	**ACC**	SSCON	SUMR0	SUMR1	SUMR2	SUMR3	ODAC	LVDCON	E7
E8	**EIE**	HWPC0	HWPC1	HDWVER	Reserved	Reserved	FMCON	FTCON	EF
F0	**B**	PDCON	PASEL				ACLK	SRST	F7
F8	**EIP**	SECINT	MSINT	USEC	MSECL	MSECH	HMSEC	WDTCON	FF

3.2 Referencing SFRs

When writing code in assembly language, SFRs may be referenced either by their name or their address.

For example, the SBUF0 SFR is at address 99_H (see Table 3–1). In order to write the value 24_H to the SBUF SFR in assembly language, it would be written in code as:

```
MOV 99h,#24h
```

This instruction moves the value 24_H into address 99_H. The value 99_H is in the range of 80_H to FF_H, and, therefore, refers to an SFR. Furthermore, because 99_H refers to the SBUF0 SFR, this instruction will accomplish the goal of writing the value 24_H to the SBUF0 SFR.

Although the above instruction certainly works, it is not necessarily easy to remember the address of each SFR when writing software. Thus, all 8052 assemblers allow the name of the SFR to be used in code rather than its numeric address. The above instruction would more commonly be written as:

```
MOV SBUF0,#24h
```

The instruction is much easier to read because it is obvious the SBUF0 SFR is being accessed. The assembler will automatically convert this to its numeric address at assemble time.

Note:

Many of the SFRs that the MSC1210 uses are MSC1210-specific; only 26 are recognized by the original 8052. It is usually necessary to include a header file or an include file in your program to define the additional SFRs supported by the MSC1210. Failing to do so may result in the assembler or compiler reporting compile errors. Please refer to the documentation for the compiler or assembler to discover how new SFRs of the MSC1210 must be defined in the development platform to be used.

3.2.1 Referencing Bits of SFRs

Individual bits of SFRs are referenced in one of two ways. The general convention is to name the SFR followed by a period and the bit number. For example, SCON0.0 refers to bit 0 (the least significant bit) of the SCON0 SFR. SCON0.7 refers to bit 7 (the most significant bit) of SCON0.

These bits also have names: SCON0.0 is RI and SCON0.7 is SM0_0. It is also acceptable to refer to the bits by their name, although in this document they will usually be referred to in the SCON0.0 format, because that defines which bit is in which SFR.

3.3 Bit–Addressable SFRs

All SFRs that have addresses divisible by eight (i.e., 80_H, 88_H, 90_H, 98_H, etc.) are bit-addressable. This means that individual bits of these SFRs can be set or cleared using the SETB and CLR instruction.

Note:

The SFRs whose names appear **BOLD** in Table 3–1 are SFRs that may be accessed via bit operations; these also happen to be the first column of SFRs on the left side of the chart. The other SFRs cannot be accessed using bit operations such as SETB or CLR.

3.4 SFR Types

Four of the SFRs are related to the I/O ports. The MSC1210 has four I/O ports of eight bits, for a total of 32 I/O lines. Whether a given I/O line is high or low, and the value read from the line, is controlled by these SFRs. Refer to Section 15.1 for the detailed control of the port usages.

SFRs control the operation or the configuration of the MSC1210. For example, TCON controls the timers and SCON controls the serial port.

The remaining SFRs can be thought of as auxiliary SFRs, in the sense that they do not directly configure the MSC1210, but obviously the MSC1210 cannot operate without them. For example, once the serial port has been configured using SCON0, the program can read or write to the serial port using the SBUF0 register.

3.5 SFR Definitions

This section will endeavor to quickly overview each of the SFRs found in the SFR chart map of Table 3–1. It is not the intention of this section to fully explain the functionality of each SFR—this information will be covered in separate chapters. This section is to just give a general idea of what each SFR does.

P0 (Port 0, Address 80_H, Bit-Addressable): This is input/output port 0. Each bit of this SFR corresponds to one of the pins on the microcontroller. For example, bit 0 of port 0 is pin P0.0, bit 7 is pin P0.7. Writing a value of 1 to a bit of this SFR sets a high level on the corresponding I/O pin, whereas a value of 0 brings it to a low level.

Note:

Even though the MSC1210 has four I/O ports (P0, P1, P2, and P3), if the hardware uses external RAM or external code memory (i.e., if the program is stored in an external ROM or EPROM chip, or if external RAM chips are being used), P0 or P2 may not be used. This is because the MSC1210 uses ports P0 and P2 to address the external memory (refer to Section 15.1 for the detailed control of the port usages). Thus, if external RAM or code memory is being used, only ports P1 and P3 (except P3.6 and P3.7) may be used by the application.

SP (Stack Pointer, Address 81_H): This is the stack pointer of the microcontroller. This SFR indicates where the next value to be taken from the stack will be read from Internal RAM. If a value is pushed onto the stack, the value will be written to the address of SP + 1. That is to say, if SP holds the value 07_H, a PUSH instruction will push the value onto the stack at address 08_H. This SFR is modified by all instructions that modify the stack, such as PUSH, POP, LCALL, RET, RETI, and whenever interrupts are triggered by the microcontroller.

Note:

The SP SFR, on startup, is initialized to 07_H. This means the stack will start at 08_H and will grow to larger addresses of internal RAM. It is necessary to initialize SP in the program to some other value if alternate register banks and/or bit memorywill be used because alternate register banks 1, 2, and 3, as well as the user bit variables, occupy internal RAM from addresses 08_H through $2F_H$. It is not a bad idea to initialize SP to $2F_H$ as the first instruction of every one of the programs, unless there is complete confidence that the program will not be using register banks and bit variables.

DPL0/DPH0 (Data Pointer 0 Low/High, Addresses 82_H/83_H): The SFRs DPL0 and DPH0 work together to represent a 16-bit value called Data Pointer 0. The data pointer is used in operations regarding external RAM and some instructions involving code memory. It can represent values from 0000_H to $FFFF_H$ (0 through 65,535 decimal) because it is an unsigned 2-byte integer value,

Note:

DPTR is really DPH0 and DPL0 taken together as a 16-bit value. In reality, DPTR must almost always be dealt with one byte at a time. For example, to push DPTR onto the stack, first push DPL0 and then DPH0. It is not possible to simply push DPTR onto the stack as a single value. Additionally, there is an instruction to increment DPTR. When this instruction is executed, the two bytes are operated upon as a 16-bit value. However, there is no instruction which decrements DPTR. If it is necessary to decrement the value of DPTR, special code must be written to do so. DPTR is a useful storage location for occasional 16-bit values that are being manipulated by your program—especially if those values need to be incremented frequently.

DPL1/DPH1 (Data Pointer 1 Low/High, Addresses 84_H/85_H): These two SFRs work together to form a 16-bit value called Data Pointer 1. Its purpose and function is the same as DPL0/DPH0 just described. The existence of two distinct data pointers allows a program to quickly copy data from one area of memory to another.

DPS (Data Pointer Select, Address 86_H): Bit 0 of this SFR determines whether instructions that refer to DPTR will use Data Pointer 0 or Data Pointer 1. If bit 0 is clear, Data Pointer 0 will be used (DPH0/DPL0). If bit 1 is set, Data Pointer 1 will be used (DPH1/DPL1).

PCON (Power Control, Address 87_H): This SFR is used to control the MSC1210 CPU power control modes. Certain operation modes allow the MSC1210 to go into a type of sleep mode that requires much less power. These modes of operation are controlled through PCON. Additionally, one of the bits in PCON is used to double the effective baud rate of the MSC1210 primary serial port. Do not confuse it with PDCON, which controls peripheral power-down.

TCON (Timer Control, Address 88_H, Bit-Addressable): This SFR is used to configure and modify the way in which the two timers of the 8052 operate. This SFR controls whether each of the two timers is running or stopped, and contains a flag to indicate whether each timer has overflowed. Additionally, some non-timer related bits are located in the TCON SFR. These bits are used to configure the way in which the external interrupts are activated and also contain the external interrupt flags that are set when an external interrupt has occurred.

T2CON (Timer Control 2, Address $C8_H$, Bit-Addressable): This SFR is used to configure and control the way in which timer 2 operates. This SFR is only available on 8052s, and not on 8051s.

TMOD (Timer Mode, Address 89_H): This SFR is used to configure the mode of operation of each of the two timers. Using this SFR, the program may configure each timer to be a 16-bit timer, an 8-bit auto-reload timer, a 13-bit timer, or two separate timers. Additionally, the timers may be configured to only count when an external pin is activated or to count events that are indicated on an external pin.

TL0/TH0 (Timer 0 Low/High, Addresses $8A_H$/$8B_H$): These two SFRs, taken together, represent timer 0. Their exact behavior depends on how the timer is configured in the TMOD SFR, however, these timers always count up. How and when they increment in value is configurable.

TL1/TH1 (Timer 1 Low/High, Addresses $8C_H$/$8D_H$): These two SFRs, taken together, represent timer 1. Their exact behavior depends on how the timer is configured in the TMOD SFR, however, these timers always count up. How and when they increment in value is configurable.

CKCON (Clock Control, Address $8E_H$): This SFR is used by the MSC1210 to provide you with a number of timing controls that allow the MSC1210 to mimic standard 8052 timing, or to fully exploit the high-speed nature of the MSC1210. This SFR allows timers 0, 1, and 2 to be clocked at a rate of 1/12th the crystal frequency (just like an 8052), or to be clocked at the rate of 1/4th the crystal frequency such that the clocks will be incremented once every instruction cycle. Additionally, the CKCON SFR allows you to modify how long the MSC1210 takes to access external data memory.

MWS (Memory Write Select, Address $8F_H$): This SFR contains a single bit (bit 0) that enables writing to program flash memory. If this bit is clear, MOVX @DPTR or MOVX @Ri write to data flash memory or data SRAM memory. If this bit is set, MOVX @DPTR or MOVX @Ri write to program flash memory.

TL2/TH2 (Timer 2 Low/High, Addresses CC_H/CD_H): These two SFRs, taken together, represent timer 2. Their exact behavior depends on how the timer is configured in the T2CON SFR.

RCAP2L/RCAP2H (Timer 2 Capture Low/High, Addresses CA_H/CB_H): These two SFRs, taken together, represent the timer 2 capture register. It may be used as a reload value for timer 2, or to capture the value of timer 2 under certain circumstances. The exact purpose and function of these two SFRs depends on the configuration of T2CON.

P1 (Port 1, Address 90_H, Bit–Addressable): This is input/output port 1. Each bit of this SFR corresponds to one of the pins on the microcontroller. For example, bit 0 of port 1 is pin P1.0, bit 7 is pin P1.7. Writing a value of 1 to a bit of this SFR will set a high level on the corresponding I/O pin, whereas a value of 0 will bring it to a low level.

EXIF (External Interrupt Flag, Address 91_H): This SFR contains the interrupt trigger flags for external interrupts 2 through 5. When these bits are set, the corresponding interrupt will be triggered, as long as that interrupt is enabled.

MPAGE (Memory Page, Address 92_H): This SFR contains the high byte of the address to access when using the MOVX @Ri instructions. A normal 8052 requires the high byte of the address be written to P2; the MSC1210, however, requires that the byte be written to the MPAGE SFR.

CADDR (Configuration Address Register, Address 93_H): This SFR is used to read the 128 bytes of Flash hardware configuration data. The contents of the Flash configuration data at the address pointed to by this SFR will be loaded into CDATA (see the following SFR definition).

CDATA (Configuration Data Register, Address 94_H): The contents of the Flash hardware configuration data pointed to by CADDR will be readable in this SFR. This SFR is read-only. Also note that attempting to read the Flash configuration data while executing the program from flash memory will return invalid data. Internal Boot ROM routines or external program memory user routines may access this memory correctly.

MCON (Memory Configuration, Address 95_H): This SFR is used to control the memory configuration. It determines breakpoints, as well as where the internal static RAM will be mapped to in memory.

SCON0 (Serial Control 0, Address 98_H, Bit-Addressable): This SFR is used to configure the behavior of the MSC1210 primary onboard serial port. This SFR controls the baud rate of the serial port, whether the serial port is activated to receive data, and also contains flags that are set when a byte is successfully sent or received.

Note:

To use the MSC1210 onboard serial port, it is generally necessary to initialize the following SFRs: SCON0, TCON, and TMOD. This is because SCON0 controls the serial port, but in most cases the program must use one of the timers to establish the serial port baud rate. In this case, it is necessary to configure timer 1 or timer 2 by initializing TCON and TMOD, or T2CON.

SBUF0 (Serial Buffer 0, Address 99_H): This SFR is used to send and receive data via the primary serial port. Any value written to SBUF0 will be sent out the serial port TXD pin. Likewise, any value which the MSC1210 receives via the serial port RXD pin will be delivered to your program via SBUF0. In other words, SBUF0 serves as the output port when written to and as an input port when read from.

SPICON (SPI Control, Address $9A_H$): This SFR controls the basic configuration of the SPI interface, including clocking rate, master/slave, and polarity. Note that writing to or updating this SFR will reset the SPI interface.

SPIDATA (SPI Data, Address $9B_H$): This SFR acts in a fashion similar to SBUF0 in that data written to this SFR will be sent out the SPI port and incoming data received by the SPI port will be readable at this SFR address.

SPIRCON (SPI Receive Control, Address 9C$_H$): This SFR is dual-purpose: when read, it will return the number of bytes currently in the SPI receive buffer; when written, it can be used to clear the receive buffer and/or indicate how many characters should accumulate in the receive buffer before triggering an SPI interrupt.

SPITCON (SPI Transmit Control, Address 9D$_H$): This SFR, like SPIRCON, is dual-purpose: when read, it will return the number of bytes currently in the SPI transmit buffer; when written, it can be used to clear the transmit buffer and/or configure whether the SCLK driver is enabled (when in master mode).

SPISTART (SPI Buffer Start Address, Address 9E$_H$): This SFR indicates where the SPI buffer begins. A value of between 128 and 255 must be written to this SFR, and the buffer is situated in internal RAM in the upper 128 bytes.

SPIEND (SPI Buffer End Address, Address 9F$_H$): This SFR indicates where the SPI buffer ends. It must be a value between 128 and 255, and must be larger than SPISTART.

P2 (Port 2, Address A0$_H$, Bit-Addressable): This is input/output port 2. Each bit of this SFR corresponds to one of the pins on the microcontroller. For example, bit 0 of port 2 is pin P2.0, bit 7 is pin P2.7. Writing a value of 1 to a bit of this SFR will set a high level on the corresponding I/O pin, whereas a value of 0 will bring it to a low level.

Note:

Even though the MSC1210 has four I/O ports (P0, P1, P2, and P3), if the hardware uses external RAM or external code memory (i.e., the program is stored in an external ROM or EPROM chip, or if external RAM chips are being used), P0, P2, P3.6, or P3.7 may not used. This is because the MSC1210 uses ports P0 and P2 to address the external memory. Thus, if external RAM or code memory is being used, only P1 and P3 (except P3.6 and P3.7) are available to the application for I/O.

PWMCON (PWM Control, Address A1$_H$): This SFR controls the PWM that can be generated automatically by the MSC1210.

PWMLOW/PWMHIGH (PWM Low/High-Byte, Addresses A2$_H$/A3$_H$): This SFR works together with the PWMCON SFR to determine the length and shape of the PWM. This SFR contains the low byte.

PAI (Pending Auxiliary Interrupt, Address A5$_H$): This SFR contains information regarding which of the various possible conditions triggered an auxiliary interrupt. This SFR is normally used by the ISR to determine the highest priority pending auxiliary interrupt.

AIE (Auxiliary Interrupt Enable, Address A6$_H$): This SFR enables and disables the various interrupts that were described in the previous paragraph regarding PAI. The interrupts mentioned in PAI will only be triggered if they are enabled in this SFR and if EAI (in EICON) is enabled. When read, the AIE SFR provides the status of the interrupt, regardless of the state of the EAI bit.

AISTAT (Auxiliary Interrupt Status, Address A7$_H$): This is a read-only SFR that will provide you with the current status of all the enabled (not masked by AIE) auxiliary interrupts. Those interrupts that have been disabled (masked) by AIE will not be available in AISTAT.

IE (Interrupt Enable, Address A8$_H$): This SFR is used to enable and disable specific interrupts. The low seven bits of the SFR are used to enable or disable the specific interrupts, whereas the highest bit is used to enable or disable ALL interrupts. Therefore, if the high bit of IE is 0, all interrupts are disabled regardless of whether an individual interrupt is enabled by setting a lower bit.

BPCON (Breakpoint Control, Address A9$_H$): This SFR controls whether or not breakpoints are enabled and, if they are, what the source of the breakpoint is.

BPL/BPH (Breakpoint Address Low/High Byte, Addresses AA$_H$/AB$_H$): These two SFRs hold a 16-bit address at which a breakpoint will be triggered. Which breakpoint (0 or 1) the SFRs reference depends on the configuration of the MCON SFR.

P0DDRL/P0DDRH (Port 0 Data Direction Low/High Byte, Addresses AC$_H$/AD$_H$): These two SFRs, together, configure the state of each port 0 pin: standard 8051 (pull-up), CMOS output, ppen-drain output, or input.

P1DDRL/P1DDRH (Port 1 Data Direction Low/High Byte, Addresses AE$_H$/AF$_H$): These two SFRs, together, configure the state of each port 1 pin: standard 8051 (pull-up), CMOS output, open-drain output, or input.

P3 (Port 3, Address B0$_H$, Bit-Addressable): This is input/output port 3. Each bit of this SFR corresponds to one of the pins on the microcontroller. For example, bit 0 of port 3 is pin P3.0, bit 7 is pin P3.7. Writing a value of 1 to a bit of this SFR will set a high level on the corresponding I/O pin, whereas a value of 0 will bring it to a low level.

P2DDRL/P2DDRH (Port 2 Data Direction Low/High Byte, Addresses B1$_H$/B2$_H$): These two SFRs, together, configure the state of each port 2 pin: standard 8051 (pull-up), CMOS output, open-drain output, or input.

P3DDRL/P3DDRH (Port 3 Data Direction Low/High Byte, Addresses B3$_H$/B4$_H$): These two SFRs, together, configure the state of each port 3 pin: standard 8051 (pull-up), CMOS output, open-drain output, or input.

IP (Interrupt Priority, Addresses B8$_H$, Bit-Addressable): This SFR is used to specify the relative priority of each interrupt. An interrupt may either be of low (0) priority or high (1) priority. An interrupt may only interrupt interrupts of lower priority. For example, if we configure the MSC1210 so that all interrupts are of low priority except the serial interrupt, the serial interrupt will always be able to interrupt the system, even if another interrupt is currently executing. However, if a serial interrupt is executing, no other interrupt will be able to interrupt the serial interrupt routine, because the serial interrupt routine has the highest priority.

SCON1 (Serial Control 1, Address C0$_H$, Bit-Addressable): This SFR is used to configure the behavior of the MSC1210 secondary onboard serial port. SCON1 controls the baud rate of the serial port, whether the serial port is activated to receive data, and also contains flags that are set when a byte is successfully sent or received.

SBUF1 (Serial Buffer 1, Address C1$_H$): This SFR is used to send and receive data via the secondary onboard serial port. Any value written to SBUF1 will be sent out the serial port TXD1 pin. Likewise, any value that the MSC1210 receives via the serial port RXD1 pin will be delivered to the user program via SBUF1. In other words, SBUF1 serves as the output port when written to, and as an input port when read from.

EWU (Enable Wake–up, Address C6$_H$): The EWU SFR controls under what conditions the MSC1210 will wake up from idle mode: external 1 interrupt, external 0 interrupt, and watchdog interrupt. Idle wakeup from Auxint is controlled via EAI bit of EICON SFR.

PSW (Program Status Word, Address D0$_H$, Bit-Addressable): This SFR is used to store a number of important bits that are set and cleared by instructions. The PSW SFR contains the carry flag, the auxiliary carry flag, the overflow flag, and the parity flag. Additionally, the PSW SFR contains the register bank select flags that are used to select which of the R register banks are currently selected.

Note:

When writing an interrupt handler routine, it is a very good idea to always save the PSW SFR on the stack and restore it when the interrupt is complete. Many instructions modify the bits of the PSW. If the interrupt routine does not ensure that the PSW is the same upon exit as it was upon entry, the program is bound to behave rather erratically and unpredictably, and it will be tricky to debug because the behavior may not make any sense.

OCL/OCM/OCH (Offset Calibration Low/Middle/High Byte, Addresses D1$_H$/D2$_H$/D3$_H$): These three SFRs make up a 24-bit value that sets the ADC offset calibration.

GCL/GCM/GCH (Gain Low/Middle/High Byte, Addresses D4$_H$/D5$_H$/D6$_H$): These three SFRs make up a 24-bit value that sets ADC gain calibration.

ADMUX (ADC Multiplexer Register, Address D7$_H$): This SFR selects the positive input for the ADC and/or selects the temperature sensor option.

EICON (Enable Interrupt Control, Address D8$_H$, Bit-Addressable): This SFR controls whether or not the additional interrupts provided by the MSC1210 will cause an interrupt to occur when their corresponding conditions are enabled.

ADRESL/ADRESM/ADRESH (ADC Conversion Results, Addresses D9$_H$/DA$_H$/DB$_H$): These three SFRs make up a 24–bit value which holds the results of an ADC conversion.

ADCON0/ADCON1 (ADC Control 0 and 1, Addresses DC_H/DD_H): These two SFRs allow the user program to configure various aspects of the ADC.

ADCON2/ADCON3 (ADC Controls 2 and 3, Addresses DE_H/DF_H): These two SFRs control the decimation rate of the ADC; in other words, they control the frequency at which sampled data will be provided to the user program via the ADRES SFRs.

ACC (Accumulator, Addresses $E0_H$, Bit–Addressable): The accumulator is one of the most-used SFRs, because it is involved in so many instructions. The accumulator resides as an SFR at $E0_H$, which means the instruction *MOV A,#20h* is the same as *MOV E0h,#20h*. However, it is a good idea to use the first method because it only requires two bytes, whereas the second option requires three bytes.

SSCON (Summation/Shift Control, Address $E1_H$): This SFR controls what action is taken in regards to summation registers SUMR0/SUMR1/SUMR2/SUMR3.

SUMR0/SUMR1/SUMR2/SUMR3 (Summation Registers 0/1/2/3, Addresses $E2_H$/$E3_H$/$E4_H$/$E5_H$): These four registers, together, make up a 32-bit summation value for the ADC. Writing a value to the least significant byte (SUMR0) will cause the values in the other three summation registers to be added to the summation result.

ODAC (Offset DAC Register, Address $E6_H$): This SFR allows the MSC1210 to shift the input by up to half of the ADC input range.

LVDCON (Low-Voltage Detection Control, Address $E7_H$): The LVDCON SFR configures the low-voltage detection on both the analog and digital supplies. In both cases, the LVDCON allows the user program to specify the *trip* voltage below which the low-voltage detection will be triggered.

EIE (Extended Interrupt Enable, Address $E8_H$, Bit-Addressable): This SFR configures whether or not the extended interrupts are enabled, including the watchdog and external interrupts 2 through 5.

HWPC0/HWPC1 (Hardware Product Code, Addresses $E9_H$/EA_H): These two SFRs are read-only and can provide the user program with information regarding the part number version and how much flash memory is available on the part.

FMCON (Flash Memory Control, Address EE_H): This SFR controls certain aspects of the flash memory, including page erase and byte write operation. FRCM controls power saving for flash memory read operations when the MSC1210 is running at a low clock frequency. It also includes a bit that indicates whether or not flash memory is currently idle or busy with a prior memory access operation.

FTCON (Flash Memory Timing Control, Address EF_H): This SFR controls the timing and period of flash memory, specifically for writing and erasing flash memory. The period of writing to flash memeory is determined by USEC and the low four bits of FTCON, and should produce a write period of 30µs to 40µs. Meanwhile, the period of erasing flash memory is determined by MSECH/MSECL and the high four bits of FTCON, and should produce an erase period of 4ms to 11ms.

B (B Register, Address $F0_H$, Bit-Addressable): The B register is used in two instructions: multiply and divide. The B register is also commonly used by programmers as an auxiliary register to store temporary values.

PDCON (Power-Down Control, Address $F1_H$): This SFR allows the user program to power down specific on-chip peripherals that the program may not need at a given moment, thus contributing to a more energy-efficient design. This SFR allows the user to power down (or power up) the PWM generator, ADC, watchdog, SPI system, and the system timer.

PASEL ($\overline{\text{PSEN}}$/ALE select, Address $F2_H$): This SFR allows for a user program that runs entirely in internal flash memory to control the ALE and $\overline{\text{PSEN}}$ lines. The PASEL allows you to configure both ALE and $\overline{\text{PSEN}}$ such that they either behave normally or may be forced high or low. In this manner, $\overline{\text{PSEN}}$ and ALE may be used as two additional output lines if they are not needed for their normal functions.

Note:

When these two lines are used as output lines, they should only drive light capacitive loads to avoid triggering serial or parallel flash programming modes.

ACLK (Analog Clock, Address $F6_H$): This SFR is used to determine the analog clock for the ADC. The value of ACLK, plus 1, multiplied by 64 represents the number of instruction cycles between each analog sample. For example, if an instruction cycle lasts 100ns and ACLK is 9, then ACLK + 1 = 10, so 10 • 100ns = 1µs, multiplied by 64 would result in a sample being made every 64µs. A sample every 64µs is equivalent to 1 000 000 / 64 = 15 625 samples per second.

SRST (System Reset Register, Address $F7_H$): Setting this SFR to 1 and then 0 will cause a system reset to occur. This provides an easy way to reset the system via software without the need for external circuitry.

EIP (Extended Interrupt Priority, Address $F8_H$): This is the pnterrupt priority register for the extended interrupts that are enabled/disabled using the EIE SFR ($E8_H$).

SECINT (Seconds Timer Interrupt, Address $F9_H$): This SFR can be set to cause an interrupt to occur after the specified number of fractions of a second. Specifically, this SFR can cause an interrupt every 100 milliseconds to every 12.8 seconds, assuming the HMSEC is set to a value that represents 100ms. The precise frequency at which SECINT will cause an interrupt depends on the system clock and the values of the MSECH, MSECL, HMSEC, and SECINT SFRs.

MSINT (Milliseconds Interrupt, Address FA$_H$): This SFR can be set to cause an interrupt to occur after the specified number of milliseconds. This assumes that the millisecond registers FC$_H$ and FD$_H$ are set to generate a cycle every millisecond. The precise frequency at which MSINT will cause an interrupt depends on the system clock and the value of the MSECH, MSECL, and MSINT SFRs.

USEC (Microsecond Register, Address FB$_H$): This SFR is divided into the clock speed to determine the timing of 1ms. This value is used for programming flash memory. The value in USEC, taken together with the low four bits of FTCON, should produce a timing of 30µs to 40µs, which is used for flash write operations.

MSECL/MSECH (Millisecond Low/High Registers, Addresses FC$_H$/FD$_H$): These two SFRs, together, are used by the system to determine how long a millisecond is. This value is used for erasing flash memory, millisecond interrupt, second interrupt, and watchdog time. Although it is named Millisecond Low/High, the clock speed and the value placed in these registers will determine the exact length of time measured.

HMSEC (Hundred Millisecond Clock, Address FE$_H$): This SFR is used to create a 100ms clock based on the MSECL/MSECH SFRs. However, the exact frequency generated by this SFR will depend on the system clock, the value of MSECL/MSECH, and the value placed in this register.

WDTCON (Watchdog Control, Address FF$_H$): The WDTCON SFR is used to enable, disable, and reset the watchdog timer. Once enabled, this SFR must be periodically reset in order to prevent the system from resetting.

Chapter 4

Basic Registers

Chapter 4 describes the basic register functions of the MSC1210 ADC.

Topic **Page**

4.1 Description

A number of MSC1210 registers can be considered basic. Very little can be done without them and a detailed explanation of each one is warranted to make sure the reader understands these registers before getting into more complicated areas of development.

4.2 Accumulator

The accumulator is a familiar concept when working with any assembly language.

The accumulator, as its name suggests, is used as a general register to accumulate the results of a large number of instructions. It can hold an 8-bit (1-byte) value and is the most versatile register of the MSC1210, due to the shear number of instructions that make use of the accumulator. More than half of the 255 opcodes of the MSC1210 manipulate or use the accumulator in some way.

For example, if adding the numbers 10 and 20, the resulting 30 will be stored in the accumulator. Once a value is in the accumulator, it may continue to be processed, or may be stored in another register or in memory.

4.3 R Registers

The R registers are sets of eight registers that are named R0 through R7.

These registers are used as auxiliary registers in many operations. To continue with the previous example of adding 10 and 20, the original number 10 may be stored in the accumulator, whereas the value 20 may be stored in, say, register R4. To process the addition, the following command would be executed:

```
ADD A,R4
```

After executing this instruction, the accumulator will contain the value 30.

The R registers are considered as very important auxiliary, or helper, registers. The accumulator alone would not be very useful if it were not for these R registers.

The R registers are also used to store values temporarily. For example, add the values in R1 and R2 together and then subtract the values of R3 and R4. One way to do this would be:

```
MOV A,R3  ;Move the value of R3 into the accumulator
ADD A,R4  ;Add the value of R4
MOV R5,A  ;Store the resulting value temporarily in R5
MOV A,R1  ;Move the value of R1 into the accumulator
ADD A,R2  ;Add the value of R2
SUBB A,R5 ;Subtract the value of R5 (which now contains R3 + R4)
```

As shown, R5 was used to temporarily hold the sum of R3 and R4. Of course, this is not the most efficient way to calculate (R1 + R2) – (R3 + R4), but it does illustrate the use of the R registers as a way to store values temporarily.

As mentioned previously, there are four sets of R registers: register bank 0, 1, 2, and 3. When the MSC1210 is first powered up, register bank 0 (addresses 00_H through 07_H) is used by default. In this case, for example, R4 is the same as internal RAM address 04_H. However, yours program may instruct the MSC1210 to use one of the alternate register banks (i.e., register banks 1, 2, or 3). In this case, R4 will no longer be the same as internal RAM address 04_H. For example, if your program instructs the MSC1210 to use register bank 1, register R4 will now be synonymous with internal RAM address $0C_H$. If selecting register bank 2, R4 is synonymous with 14_H, and if selecting register bank 3, it is synonymous with address $1C_H$.

The concept of register banks adds a great level of flexibility to the MSC1210, especially when dealing with interrupts (see Chapter 10, *Interrupts,* for details). However, always remember that the register banks really reside in the first 32 bytes of internal RAM.

4.4 B Register

The B register is very similar to the accumulator in the sense that it may hold an 8-bit (1-byte) value.

The B register is only used by two MSC1210 instructions: MUL AB and DIV AB. Therefore, to quickly and easily multiply or divide A by another number, the other number may be stored in B.

Aside from the MUL and DIV instructions, the B register is often used as another temporary storage register much like a 9th R register.

4.5 Program Counter (PC)

The program counter (PC) is a 2-byte address that tells the MSC1210 where the next instruction to execute is found in memory. When the MSC1210 is initialized, the PC always starts at 0000_H and is incremented each time an instruction is executed. It is important to note that the PC is not always incremented by one. The PC will be incremented by two or three in these cases because some instructions require two or three bytes.

The PC is special in that there is no way to directly modify its value. That is to say, something like PC = 2430_H cannot be done. On the other hand, by executing LJMP 2430_H, the same thing is effectively accomplished.

It is also interesting to note that although the value of the PC may be changed (by executing a jump instruction, etc.), there is no way to read the value of the PC. That is to say, there is no way to ask the 8052 “what address are you about to execute?”

4.6 Data Pointer (DPTR0/DPTR1)

The data pointer (DPTR0/DPTR1) is the user-accessible 16-bit (2-byte) register of the MSC1210. The accumulator, R registers, and B register are all 1-byte values. The PC just described is a 16-bit value, but is not directly user-accessible as a working register.

DPTR0/DPTR1, as the name suggests, are used to point to data. They are used by a number of commands that allow the MSC1210 to access data and code memory. When the MSC1210 accesses external memory, it accesses the memory at the address indicated by DPTR0/DPTR1.

Although DPTR0/DPTR1 is most often used to point to data in external memory or code memory, many developers take advantage of the fact that it is the only true 16-bit register available. It is often used to store 2-byte values that have nothing to do with memory locations. DPTR0 or DPTR1 is selected by SFR DPS.

4.7 Stack Pointer (SP)

The stack pointer (SP), like all registers except DPTR and PC, may hold an 8-bit (1-byte) value. The SP is used to indicate where the next value to be removed from the stack should be taken from.

When a value is pushed onto the stack, the MSC1210 first increments the value of the SP and then stores the value at the resulting memory location.

When a value is popped off the stack, the MSC1210 returns the value from the memory location indicated by the SP, and then decrements the value of the SP.

This order of operation is important. When the MSC1210 is initialized, SP will be initialized to 07_H. If a value is immediately pushed onto the stack, the value will be stored in internal RAM address 08_H. This makes sense, taking into account what was mentioned two paragraphs above. First the MSC1210 will increment the value of the SP (from 07_H to 08_H) and then will store the pushed value at that memory address (08_H).

The SP is modified directly by the MSC1210 by six instructions: PUSH, POP, ACALL, LCALL, RET, and RETI. It is also used intrinsically whenever an interrupt is triggered (more on interrupts in Chapter 10—do not worry about them for now).

Chapter 5

Addressing Modes

Chapter 5 describes the various addressing modes of the MSC1210.

Topic **Page**

5.1 Description

As is the case with all microcomputers from the PDP-8 onwards, the MSC1210 uses several memory addressing modes. An addressing mode refers to how you are accessing (addressing) a given memory location or data value. In summary, the addressing modes are listed in Table 5–1 with an example of each.

Table 5–1. MSC1210 Addressing Modes.

Mode	**Example**
Immediate Addressing	MOV A,#20h
Direct Addressing	MOV A,30h
Indirect Addressing	MOV A,@R0
External Direct	MOVX A,@DPTR
External Indirect	MOVX A,@R0
Code Indirect	MOVC A,@A+DPTR

Each of these addressing modes provides important flexibility to the programmer.

5.2 Immediate Addressing

Immediate addressing is so named because the value to be stored in memory immediately follows the opcode in memory. That is to say, the instruction itself dictates what value will be stored in memory. For example:

```
MOV A,#20h
```

This instruction uses immediate addressing because the accumulator (A) will be loaded with the value that immediately follows; in this case 20_H (hex).

Immediate addressing is very fast because the value to be loaded is included in the instruction. However, because the value to be loaded is fixed at compile time, it is not very flexible. It is used to load the same, known value every time the instruction executes.

5.3 Direct Addressing

Direct addressing is so named because the value to be stored in memory is obtained by directly retrieving it from another memory location. For example:

```
MOV A,30h
```

This instruction will read the data out of internal RAM address 30_H (hex) and store it in the accumulator (A).

Direct addressing is generally fast because, although the value to be loaded is not included in the instruction, it is quickly accessible due to it being stored in the MSC1210 internal RAM. It is also much more flexible than immediate addressing because the value to be loaded is whatever is found at the given address, which may change.

Additionally, it is important to note that when using direct addressing, any instruction that refers to an address between 00_H and $7F_H$ is referring to internal RAM. Any instruction that refers to an address between 80_H and FF_H is referring to the SFR control registers that control the MSC1210 itself.

The obvious question that may arise is “if direct addressing an address from 80_H through FF_H refers to SFRs, how can I acess the upper 128 bytes of internal RAM that are available with the MSC1210?” The answer is: it cannot be accessed using direct addressing. As stated, if an address of 80_H through FF_H is directly referred to, it refers to an SFR.

However, the upper 128 bytes of RAM of the MSC1210 can be accessed by using the next addressing mode, indirect addressing.

5.4 Indirect Addressing

Indirect addressing is a very powerful addressing mode that in many cases provides an exceptional level of flexibility. Indirect addressing is also the only way to access the upper 128 bytes of Internal RAM found on an 8052.

Indirect addressing appears as follows:

```
MOV A,@R0
```

This instruction causes the MSC1210 to analyze the value of the R0 register. The MSC1210 then loads the accumulator (A) with the value from Internal RAM that is found at the address indicated by R0.

For example, suppose R0 holds the value 40_H and internal RAM address 40_H holds the value 67_H. When the above instruction is executed, the 8052 checks the value of R0. The MSC1210 gets the value out of internal RAM address 40_H (which holds 67_H) and stores it in the accumulator because R0 holds 40_H. Thus, the accumulator ends up holding 67_H.

Indirect addressing always refers to internal RAM; *it never refers to an SFR*. In a prior example, it was mentioned that SFR 99_H can be used to write a value to the serial port. Therefore, one can think that the following code would be a valid solution to write the value of 1 to the serial port:

```
MOV R0,#99h  ;Load the address of the serial port

MOV @R0,#01h ;Send 01 to the serial port –– WRONG!!
```

This is not valid. These two instructions write the value 01_H to internal RAM address 99_H on the MSC1210 because indirect addressing *always* refers to internal RAM.

5.5 External Direct Addressing

External memory is accessed using a suite of instructions that use external direct addressing. It is referred to as external direct because it appears to be direct addressing, but it is used to access external memory rather than internal memory.

There are only two commands that use external direct addressing mode:

```
MOVX A,@DPTR

MOVX @DPTR,A
```

As you can see, both commands use DPTR. In these instructions, DPTR must first be loaded with the address of external memory that you wish to read or write. Once DPTR holds the correct external memory address, the first command moves the contents of that external memory address into the accumulator. For example, if you want to read the contents of external RAM address 1516_H, execute the instructions:

```
MOV DPTR,#1516h ;Select the external address to read

MOVX A,@DPTR    ;Move the contents of external RAM into
                ;accumulator
```

The second command does the opposite: it allows you to write the value of the accumulator to the external memory address pointed to by DPTR. For example, if you want to write the contents of the accumulator to external RAM address 1516_H, execute the instructions:

```
MOV DPTR,#1516h ;Select the external address to read

MOVX @DPTR,A    ;Move the contents of external RAM into
                ;accumulator
```

MOVX to data flash memory writes to the data flash memory location. To clear the flash content, page erase is needed.

5.6 External Indirect Addressing

External memory can also be accessed using a form of indirect addressing called external indirect. This form of addressing is usually only used in relatively small projects that have a very small amount of external RAM. An example of this addressing mode is:

MOVX @R0,A

Once again, the value of R0 is first read and the value of the accumulator is written to that address in external RAM, internal extended SRAM, and internal flash data memory. High address A8~A15 is provided by the MPAGE SFR because the value of @R0 can only be 00_H through FF_H—that is A0~A7 of the previous memories.

5.7 Code Indirect Adressing

The last addressing mode is called code indirect and offers two additional 8052 instructions that allow you to access the program code itself. This is useful for accessing data tables, strings, etc. The two instructions are:

```
MOVC A,@A+DPTR

MOVC A,@A+PC
```

For example, if you want to access the data stored in code memory at address 2021_H, execute the instructions:

```
MOV   DPTR,#2021h  ;Set DPTR to 2021h

CLRA               ;Clear the accumulator (set to 00h)

MOVC  A,@A+DPTR    ;Read code memory address 2021h into
                   ;the accumulator
```

The MOVC A,@A+DPTR instruction moves the value contained in the code memory address that is pointed to by adding DPTR to the accumulator.

To write to flash code memory, set the MXWS bit and MOVX will write to flash code memory (if the memory is not write protected by harware configuration bits). The same operation can be used to perform flash page erase. See section 1.5, *Flash Memory*, for more details.

Chapter 6

Program Flow

Chapter 6 describes the program flow of the MSC1210 ADC.

Topic **Page**

6.1 Description

When the MSC1210 is first initialized the PC SFR is cleared to 0000_H. The part then begins to execute instructions sequentially in memory unless a program instruction causes the PC to be otherwise altered. There are various instructions that can modify the value of the PC; specifically, conditional branching instructions, direct jumps and calls, and returns from subroutines. Additionally, interrupts (when enabled) can cause the program flow to deviate from its otherwise sequential scheme.

6.2 Conditional Branching

The MSC1210 contains a suite of instructions that, as a group, are referred to as “conditional branching” instructions. These instructions cause the program execution to follow a non-sequential path if a certain condition is true.

Let us use the JB instruction as an example. This instruction means jump if bit set. An example of the JB instruction might be:

```
          JB 45h,HELLO

          NOP

HELLO:....
```

In this case, the MSC1210 will analyze the contents of bit 45_H. If the bit is set, program execution will jump immediately to label HELLO, skipping the NOP instruction. If the bit is not set, the conditional branch fails and program execution continues as usual with the NOP instruction that follows.

Conditional branching is really the fundamental building block of program logic because all decisions are accomplished by using conditional branching. Conditional branching can be thought of as the “IF ... THEN” structure of assembly language.

Note:

Your program may only branch to instructions located within 128 bytes prior to, or 127 bytes after the address that follows the conditional branch instruction. This means that in the above example, the label HELLO must be within –128 bytes to +127 bytes of the memory address that contains the conditional branching instruction.

6.3 Direct Jumps

While conditional branching is extremely important, it is often necessary to make a direct branch to a given memory location without basing it on a given logical decision. This is equivalent to saying GOTO in Basic. In this case, the program flow will continue at a given memory address without considering any conditions.

This is accomplished with the MSC1210 using direct jump and call instructions. As illustrated in the last paragraph, this suite of instructions causes program flow to change unconditionally.

Consider the example:

```
              LJMP NEW_ADDRESS
.
.
.
NEW_ADDRESS: ....
```

The LJMP instruction in this example means “Long Jump.” When the MSC1210 executes this instruction, the PC is loaded with the address of NEW_ADDRESS and program execution continues sequentially from there.

The obvious difference between the Direct Jump and Call instructions and conditional branching is that with Direct Jumps and Calls, program flow always changes; with conditional branching, program flow only changes if a certain condition is true.

It is worth mentioning that, aside from LJMP, there are two other instructions that cause a direct jump to occur: the SJMP and AJMP commands. Functionally, these two commands perform the exact same function as the LJMP command—that is to say, they always cause program flow to continue at the address indicated by the command. However, these instructions differ from LJMP in that they are not capable of jumping to any address. They both have limitations as to the range of the jumps.

The SJMP command, like the conditional branching instructions, can only jump to an address within –128/+127 bytes of the address following the SJMP command.

The AJMP command can only jump to an address that is in the same 2k block of memory as the byte following the AJMP command. That is to say, if the AJMP command is at code memory location 650_H, it can only do a jump to addresses 0000_H through $07FF_H$ (0 through 2047, decimal).

You may ask “why use the SJMP or AJMP commands, which have restrictions as to how far they can jump, if they do the same thing as the LJMP command that can jump anywhere in memory?” The answer is simple: the LJMP command requires three bytes of code memory, whereas both the SJMP and AJMP commands require only two. When developing applications that have memory restrictions, quite a bit of memory can be saved using the 2-byte AJMP/SJMP instructions instead of the 3-byte instruction.

Note:

Some assemblers will do the above conversion automatically. That is, they will automatically change LJMPs to SJMPs whenever possible. This is a nifty and very powerful capability that may be a necessity in an assembler, if planning to develop many projects that have relatively tight memory restrictions.

6.4 Direct Calls

Another operation that will be familiar to seasoned programmers is the LCALL instruction. This is similar to a “GOSUB” command in Basic.

When the MSC1210 executes an LCALL instruction, it immediately pushes the current PC onto the stack and then continues executing code at the address indicated by the LCALL instruction.

6.5 Returns From Routines

Another structure that can cause program flow to change is the “Return from Subroutine” instruction, known as RET in Assembly language. The RET instruction, when executed, returns to the address following the instruction that called the given subroutine. More accurately, it returns to the address that is stored on the stack.

The RET command is direct in the sense that it always changes program flow without basing it on a condition, but is variable in the sense that where program flow continues can be different each time the RET instruction is executed, depending on where the subroutine was originally called from.

6.6 Interrupts

An interrupt is a special feature that allows the MSC1210 to break from its normal program flow to execute an immediate task, providing the illusion of multitasking. The word interrupt can often be substituted with the word event.

An interrupt is triggered whenever a corresponding event occurs. When the event occurs, the MSC1210 temporarily puts the normal execution of the program on hold and executes a special section of code referred to as an interrupt handler. The interrupt handler performs whatever special functions are required to handle the event and then returns control to the MSC1210, at which point program execution continues as if it had never been interrupted.

The topic of interrupts is somewhat tricky and very important. For that reason, Chapter 10 is dedicated to the topic.

Chapter 7

System Timing

Chapter 7 describes the system timing of the MSC1210 ADC.

Topic **Page**

7.1 Description

In order to understand—and better make use of—the MSC1210, it is necessary to understand some underlying information concerning timing.

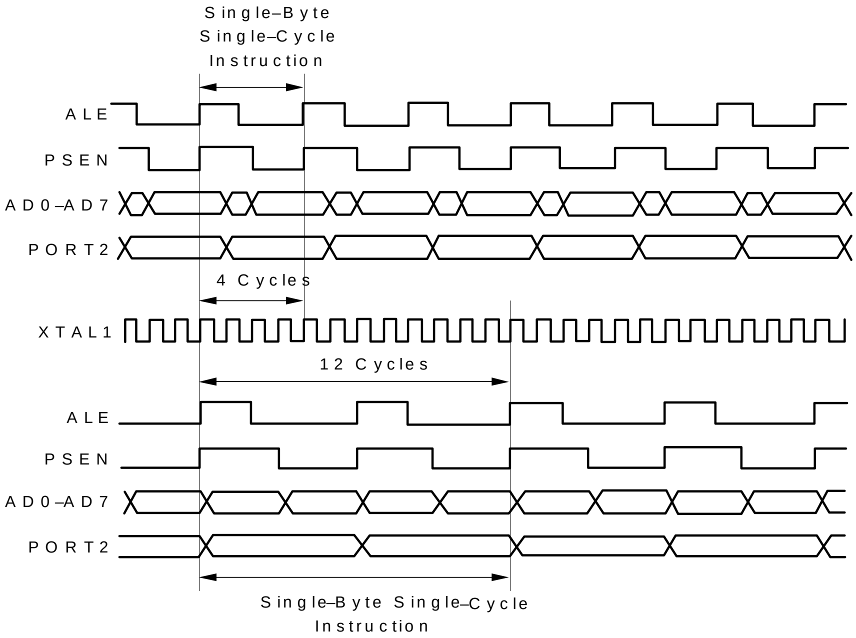
The MSC1210 operates with timing derived from an external crystal or a clock signal generated by some other system. A crystal is a mechanical oscillator that allows an electronic oscillator to run at a very precisely known frequency. One can find crystals of virtually any frequency depending on the application requirements. When using an MSC1210, a common crystal frequency is 11.0592MHz due to baud rate accuracy considerations.

Microcontrollers (and many other electrical systems) use their oscillators to synchronize operations. The MSC1210 uses its crystal or clock for precisely that—to synchronize its internal operation. The MSC1210 operates using what are called instruction cycles. A single instruction cycle is the minimum amount of time in which a single MSC1210 instruction can be executed, although many instructions take multiple cycles.

Note:

A standard 8052 executes an instruction in 12 clock cycles rather than 4, as shown in Figure 7–1. This means that, with no program changes, an MSC1210 will execute code approximately three times faster than the same program run under a traditional 8052. It also means that programs written for a standard 8052 may have to be modified if they depend on certain instructions executing in a certain amount of time. The fact that the MSC1210 executes an instruction in four cycles is not configurable.

Figure 7–1. Standard 8051 Timing.



An instruction cycle is, in reality, four clock cycles. That is to say, if an instruction takes one instruction cycle to execute, it will take four clocks from a crystal or oscillator to execute. Using the maximum crystal frequency of 33MHz, the crystal oscillates 33 000 000 times per second. Due to one instruction cycle being four clock cycles, the MSC1210 can execute the following number of instruction cycles per second:

33 000 000 / 4 = 8 250 000

This means that the MSC1210 can execute 8 250 000 single-cycle instructions per second.

It is important to emphasize that not all instructions execute in the same amount of time. The fastest instructions require one instruction cycle (four clock cycles), many others require two instruction cycles (eight clock cycles), and the two slow math operations require four instruction cycles (16 clock cycles).

Due to all the instructions requiring different amounts of time to execute, a very obvious question comes to mind: how can one keep track of time in a time-critical application if we have no reference to time in the outside world?

Luckily, the MSC1210 includes timers that allow us to time events with high precision, which is the topic of the next chapter.

7.2 System Timers

In addition to the standard 8052 timers to be described in Chapter 8, the MSC1210 includes the following system timers, both of which are capable of triggering an auxiliary interrupt (for more on interrupts, see chapter 10):

- Microseconds Timer: set via the USEC (FB_H) SFR, and is used to configure the flash writing timing and also used by the PWM module.
- Milliseconds Timer: set via the MSECH (FD_H) and MSECL (FC_H) SFRs, and is used as a base to configure the flash erase timing, as well as the milliseconds interrupt, and also as a base for the seconds interrupt and the watchdog timer.

The MSC1210 timers are illustrated in Figure 7–2. The SYS Clock is the signal that comes from the oscillator or other timing input. This signal is used as the input for all of the part’s timing logic, including the following timing circuits:

- SPI I/O (chapter 13)
- PWM/Tone generation (chapter 11).
- Flash erase/write (chapter 15).
- Milliseconds/Seconds/Watchdog interrupts (chapter 7, 14).
- A/D conversion timing (chapter 12)
- Standard 8052 timers 0, 1, and 2 (chapter 8).

Figure 7–2. MSC1210 Timing Chain and Clock Control

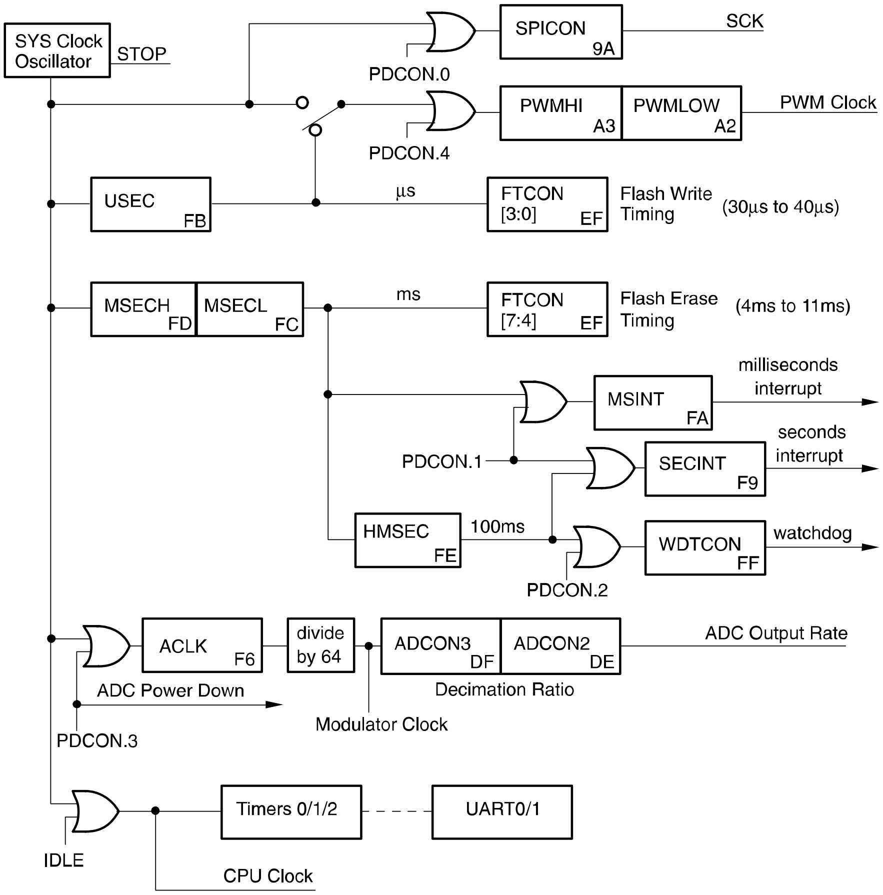
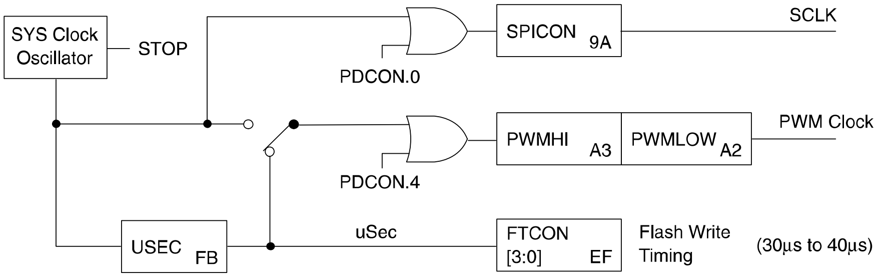


Figure 7–3. SPI/PWM/Flash Write Timing



7.2.1 Microseconds Timer

The microseconds timer is used by the MSC1210 in order to establish a 1µs clock. This clock, in turn, is used by flash memory to establish timing for flash writes, as well as by the PWM module.

The USEC (FB_H) SFR should be set to a value such that the system clock divided by the value of this SFR, plus one, generates a 1µs clock. For example, given a system clock of 12.000MHz, USEC should be set to:

12 000 000/1 000 000 = 12 – 1 = 11.

Therefore, for a 12.000MHz system clock, USEC should be set to 11 to generate a 1µs clock.

In reality, the USEC SFR may be set to a value that produces a clock that is something other than 1µs. This works fine as long as the other two timers that depend on the USEC SFR are adjusted accordingly.

7.2.1.1 PWM Clock

The PWM module may use the microseconds timer as its input clock. By clearing SPDSEL (PWMCON.3), the input clock for the PWM module will be the microsecond timer. This creates a 1MHz input clock for the PWM module, assuming the microseconds timer is correctly configured to produce a 1µs clock. In this case, the microseconds clock is further divided by the value contained in the PWMHI/PWMLOW SFRs.

7.2.1.2 Flash Write Timing

The microseconds clock is further used to establish the flash memory write timing. The flash write timing uses the microsecond clock as an input clock and then further divides it by the value of FTCON[3:0] to generate a flash write clock. The flash write clock must be between 30µs and 40µs for flash writing to operate properly.

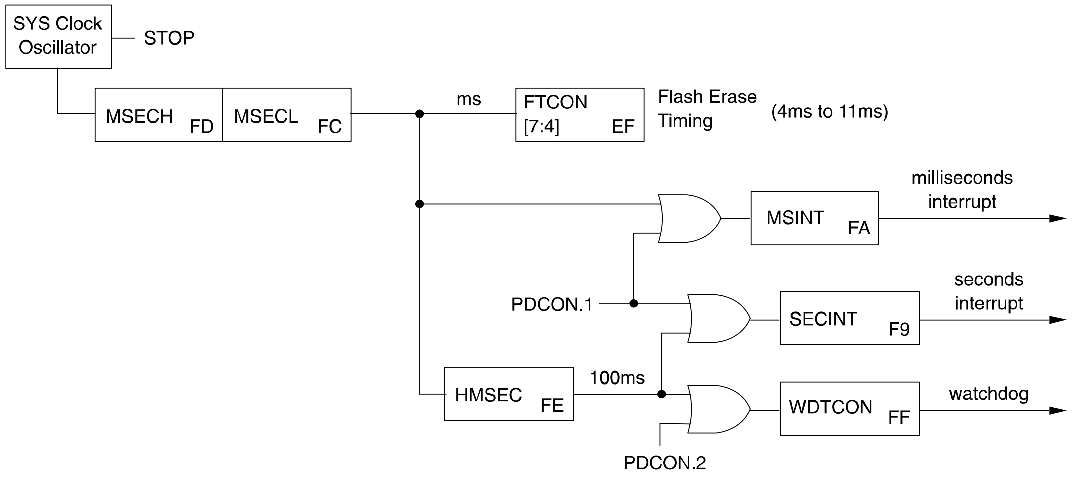
Specifically, FTCON[3:0] + 1, multiplied by five, multiplied by the frequency of the microsecond clock, should produce an appropriate flash write timer (30µs to 40µs).

Assuming USEC is set to generate a correct 1µs clock, FTCON[3:0] should be set to 5, 6, or 7. If FTCON[3:0] is 6, then (6 + 1) • 5 = 35µs, which is right in the middle of the expected range.

7.2.2 Milliseconds Timer

The milliseconds timer is used by the MSC1210 in order to establish a millisecond clock. This clock, in turn, is used as a base for establishing flash erase timing, the milliseconds interrupt, the seconds interrupt, and to establish timing for the watchdog timer.

Figure 7–4. System Timing Interrupt Control



The MSECH (FD_H) and MSECL (FC_H) SFRs should be set to a value such that the system clock divided by the value of these SFRs, plus one, generates a 1ms clock. For example, given a system clock of 12.000MHz, MSECH/MSECL should be set to 12 000 000 / 1000 = 12 000 – 1 = 11 999. Thus, for a 12.000MHz system clock, MSECH/MSECL should be set to 11 999 to generate a 1ms clock.

In reality, the MSECH/MSECL SFRs may be set to a value that produces a clock that is something other than 1ms. This works fine, as long as the other timers that depend on the MSECH/MSECL SFR are adjusted accordingly.

7.2.2.1 Milliseconds Auxiliary Interrupt

The milliseconds interrupt is one of the auxiliary interrupts that may be used by the user program. The milliseconds auxiliary interrupt is enabled by setting EMSEC (AIE.4) and enabling auxiliary interrupts via the EAI (EICON.5) bit. The frequency at which the milliseconds interrupt will be triggered is controlled by the value written to the MSINT (FA_H) SFR.

When enabled, a millisecond auxiliary interrupt will be triggered after MSINT + 1ms, assuming that MSECH/MSECL have been configured to produce a correct milliseconds clock. The value written to the MSINT SFR is a value between 0 and 127, meaning that the milliseconds interrupt may be triggered every 1ms to 128ms (assuming a correct milliseconds clock).

For example, given an accurate milliseconds clock, setting MSINT to 5 would produce a milliseconds auxiliary interrupt every 6ms.

Bit 7 of MSINT, when written, indicates whether the MSINT value being written should be written immediately, or if it should be written after the current MSINT count has expired. If bit 7 is set, MSINT will immediately be updated with the new value; if it is clear, MSINT will be updated with the new value as soon as the current milliseconds count has expired.

7.2.2.2 One Hundred Millisecond Clock

The one hundred millisecond clock is used by the MSC1210 in order to establish a 10Hz clock. This clock is not directly outputted by the MSC1210; it is used as the input into the seconds auxiliary interrupt and also is used by the watchdog timer. The 100ms clock uses the output of the millisecond clock (MSECH/ MSECL) as an input, so its correct operation assumes that the millisecond clock has been set to a value that in fact generates a millisecond clock.

The HMSEC (FE_H) SFR is used to indicate how many millisecond clocks amount to 100ms (1/10th of a second), less 1. Therefore, assuming the millisecond clock is correctly configured to generate a 1kHz clock, HMSEC would be set to 99 (decimal) in order to generate an accurate, 100ms clock.

7.2.2.3 Seconds Auxiliary Interrupt

The seconds auxiliary interrupt is one of the auxiliary interrupts that may be used by the user program. The seconds auxiliary interrupt is enabled by setting ESEC (AIE.7) and enabling auxiliary interrupts via the EAI (EICON.5) bit. The frequency at which the seconds interrupt will be triggered is controlled by the value written to the SECINT ($F9_H$) SFR.

When enabled, a seconds auxiliary interrupt will be triggered after SECINT + 100ms, assuming the MSECH/MSECL and HMSEC SFRs have been configured to produce a correct 100ms clock. The value written to the SECINT SFR is between 0 and 127, meaning that the milliseconds interrupt may be triggered every 100ms to 12.8 seconds (assuming a correct 100ms clock).

For example, given an accurate 100ms clock, setting SECINT to 15 would produce a seconds auxiliary interrupt every 1.6 seconds.

Bit 7 of SECINT, when written, indicates whether the SECINT value being written should be written immediately, or if it should be written after the current SECINT count has expired. If bit 7 is set, SECINT will immediately be updated with the new value; if it is clear, SECINT will be updated with the new value as soon as the current seconds count has expired.

7.2.2.4 Watchdog Timer

The functioning of the watchdog timer is fully described in section 14.3. However, it is important to keep in mind that the watchdog timer is dependent on the 100ms timer. The length of the watchdog timer is directly dependent on the 100ms timer being configured to a reasonable value because the watchdog timer frequency is configured in WDTCON (FF_H) using units of HMSEC.

7.3 Startup Timing

When power is turned on, or a reset is initiated, a power-on delay circuit is implemented with a 17-bit counter to guarantee that the power supply has reached a certain level, and the oscillator is stable. The delay introduced by this counter is:

24MHz System clock: $(2^{17} – 1) \bullet (1/24) \bullet 10^{–6} = 0.005461$s

1MHz System clock: $(2^{17} – 1) \bullet 10^{–6} = 0.131071$s

7.3.1 Normal-Mode Power-On Reset Timing

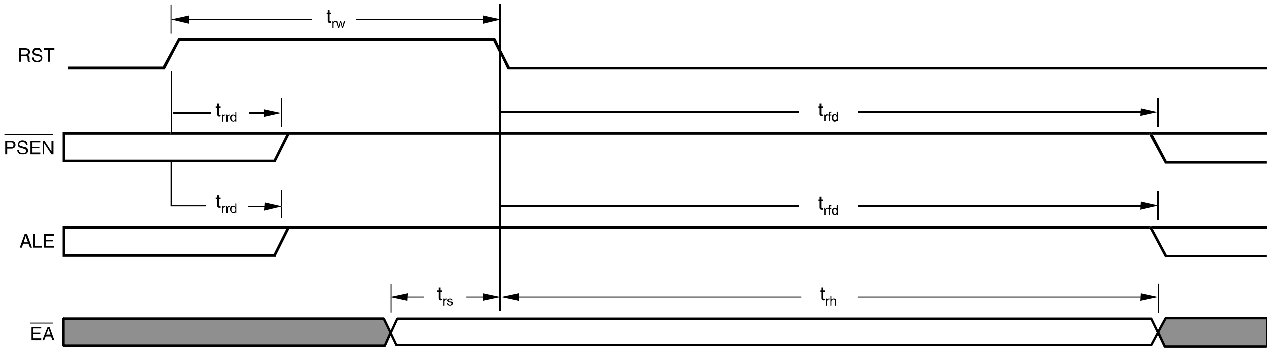
$\overline{\text{EA}}$ is sampled during power-on reset for code security purposes. $\overline{\text{PSEN}}$ and ALE are internally pulled up during reset for serial and parallel flash programming mode detection.

After the reset sequence, $\overline{\text{PSEN}}$ and ALE signals are driven by the CPU, and the internal pull up resistors are removed for saving power.

7.3.2 Flash Programming Mode Power-On Reset Timing

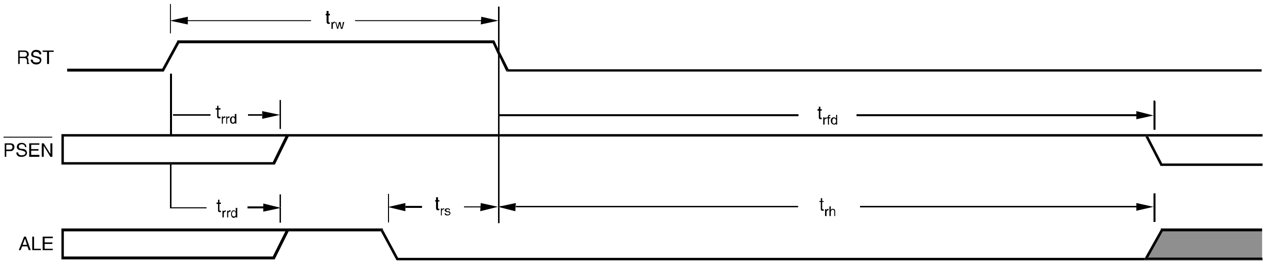
EA is ignored for serial and parallel flash programming operations.

Figure 7–5. Reset Timing



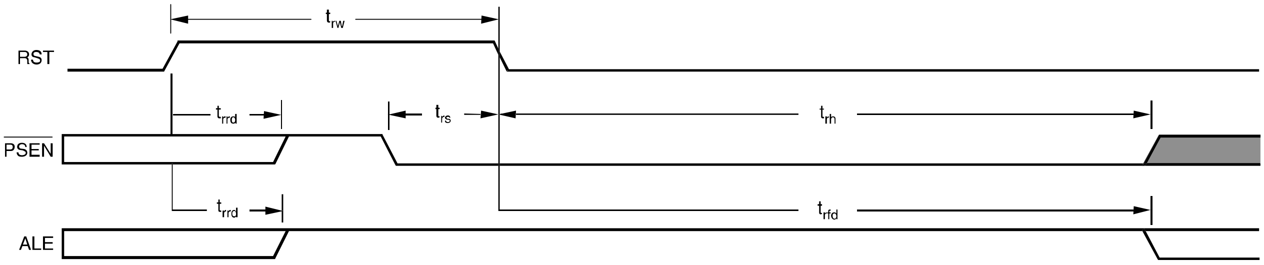
NOTE: PSEN and ALE are internally pulled up with ~9kΩ during RST high.

Figure 7–6. Parallel Flash Programming Power-On Timing ($\overline{\text{EA}}$ is ignored)



NOTE: PSEN and ALE are internally pulled up with ~9kΩ during RST high.

Figure 7–7. Serial Flash Programming Power-On Timing ($\overline{EA}$ is ignored)



NOTE: PSEN and ALE are internally pulled up with ~9kΩ during RST high.

Table 7–1. Signal Definitions for Reset Timing Diagrams

Symbol	Parameter	Min	Max	Unit
t_{rw}	RST Width	10 t_{CLK}(1)	—	ns
t_{rrd}	RST rise to $\overline{\text{PSEN}}$ ALE internal pull high	—	5	µs
t_{rfd}	RST falling to $\overline{\text{PSEN}}$ and ALE start	—	(2^{17}+512) t_{CLK}(1)	ns
t_{rs}	Input signal to RST falling setup time	t_{CLK}(1)	—	ns
t_{rh}	RST falling to input signal hold time	(2^{17}+512) t_{CLK}(1)	—	ns

Notes: 1) t_{CLK} is the Xtal clock period.

Chapter 8

Timers

Chapter 8 describes the timers of the MSC1210 ADC.

Topic **Page**

8.1 Description

The MSC1210 comes equipped with three standard timer/counters, all of which may be controlled, set, read, and configured individually. The timer/counters have three general functions:

1) Keeping time and/or calculating the amount of time between events

2) Counting the events themselves

3) Generating baud rates for the serial port

The uses of the three timer/counters are distinct, so we will talk about each of them separately. The first two uses will be discussed in this chapter, whereas the use of timers for baud rate generation will be discussed in the Chapter 9, *Serial Communication*.

8.2 How Does a Timer Count?

The answer to this question is very simple: a timer always counts up. It does not matter whether the timer is being used as a timer, a counter, or a baud rate generator. A timer is always incremented by the microcontroller.

8.3 Using Timers to Measure Time

Obviously, one of the primary uses of timers is to measure time. We will discuss this use of timers first and will subsequently discuss the use of timers to count events. When a timer is used to measure time, it is also called an interval timer, because it is measuring the time of the interval between two events.

8.3.1 How Long Does a Timer Take to Count?

Before continuing, it is worth mentioning that when a timer is in interval timer mode (as opposed to event counter mode) and correctly configured, the timer will increment by one on each instruction cycle. Therefore, a running timer in the MSC1210 will be incremented:

33 000 000 / 4 = 8 250 000 times per second

However, to maintain compatibility with existing 8052 code, the default mode for the MSC1210 timers is to increment by one every three instruction cycles (i.e., operate as if the timer increments every 12 clocks). Thus, a running timer can be configured to be incremented:

33 000 000 / 12 = 2 750 000 times per second

Using the first option, which increments the timer every four clocks, allows the user program to obtain three times higher precision than would be available by the default mode just explained. Whether the timers are incremented every four or 12 clocks is controlled by the CKCON SFR.

The individual bits of TMOD have the following functions:

	7	6	5	4	3	2	1	0	Reset Value
SFR $8E_H$	0	0	T2M	T1M	T0M	MD2	MD1	MD0	01_H

T2M (bit 5)—**Timer 2 Clock Select**. This bit controls the division of the system clock that drives Timer 2. This bit has no effect when the timer is in baud rate generator or clock output modes. Clearing this bit to 0 maintains 80C32 compatibility. This bit has no effect on instruction cycle timing.

0: Timer 2 uses a divide by 12 of the crystal frequency.

1: Timer 2 uses a divide by 4 of the crystal frequency.

T1M (bit 4)—**Timer 1 Clock Select**. This bit controls the division of the system clock that drives Timer 1. Clearing this bit to 0 maintains 8051 compatibility. This bit has no effect on instruction cycle timing.

0: Timer 1 uses a divide by 12 of the crystal frequency.

1: Timer 1 uses a divide by 4 of the crystal frequency.

T0M (bit 3)—**Timer 0 Clock Select**. This bit controls the division of the system clock that drives Timer 0. Clearing this bit to 0 maintains 8051 compatibility. This bit has no effect on instruction cycle timing.

0: Timer 0 uses a divide by 12 of the crystal frequency.

1: Timer 0 uses a divide by 4 of the crystal frequency.

MD2, MD1, MD0 (bits 2-0)—**Stretch MOVX Select 2–0**. These bits select the time by which external MOVX cycles are to be stretched. This allows slower memory or peripherals to be accessed without using ports or manual software intervention. The $\overline{\text{RD}}$ or $\overline{\text{WR}}$ strobe will be stretched by the specified interval, which will be transparent to the software except for the increased time to execute the MOVX instruction. All internal MOVX instructions on devices containing MOVX SRAM are performed at the 2 instruction cycle rate.

MD2	MD1	MD0	Stretch Value	MOVX Duration	$\overline{\text{RD}}$ or $\overline{\text{WR}}$ Strobe Width (SYS CLKs)	$\overline{\text{RD}}$ or $\overline{\text{WR}}$ Strobe Width (μs) at 12MHz
0	0	0	0	2 Instruction Cycles	2	0.167
0	0	1	1	3 Instruction Cycles (default)	4	0.333
0	1	0	2	4 Instruction Cycles	8	0.667
0	1	1	3	5 Instruction Cycles	12	1.000
1	0	0	4	6 Instruction Cycles	16	1.333
1	0	1	5	7 Instruction Cycles	20	1.667
1	1	0	6	8 Instruction Cycles	24	2.000
1	1	1	7	9 Instruction Cycles	28	2.333

Unlike instructions—some of which require one instruction cycle, others 2, and others 4—the timers are consistent. They will always be incremented once every 12 (or four) clocks. Therefore, if a timer has counted from 0 to 55 000 you may calculate:

55 000 / 2 750 000 = 0.020 seconds ($f_{OSC}/12$) or

55 000 / 8 250 000 = 0.007 seconds ($f_{OSC}/4$)

The trade off in using $f_{OSC}/12$ or $f_{OSC}/4$ as the clock source is (1) code compatibility and (2) resolution. With a 33MHz external clock, the resolution of $f_{OSC}/12$ = 364ns per increment, and the resolution of $f_{OSC}/4$ is 121ns per increment.

Thus, we now have a system that measures time. All we need to review is how to control the timers and initialize them to provide us with the information needed.

8.3.2 Timer SFRs

As mentioned before, the MSC1210 has three standard timers. Two of these timers work in essentially the same way. One timer is Timer 0 and the other is Timer 1. The two timers share two SFRs (TMOD and TCON) which control the timers, and each timer also has two SFRs dedicated solely to maintaining the value of the timer itself (TH0/TL0 and TH1/TL1). The third timer (Timer 2) functions somewhat differently and will be explained separately.

The SFRs used to control and manipulate the first two timers are presented in the Table 8–1.

Table 8–1. Timer Conrol SFRs.

SFR Name	Description	SFR Address	Bit Addressable?
TH0	Timer 0 high byte	$8C_H$	No
TL0	Timer 0 low byte	$8A_H$	No
TH1	Timer 1 high byte	$8D_H$	No
TL1	Timer 1 low byte	$8B_H$	No
TCON	Timer control	88_H	Yes
TMOD	Timer mode	89_H	No

Timer 0 has two SFRs dedicated exclusively to itself: TH0 and TL0. TL0 is the low byte of the value of the timer, while TH0 is the high byte of the value of the timer. That is to say, when Timer 0 has a value of 0, both TH0 and TL0 will contain 0. When Timer 0 has the value 1000, TH0 will hold the high byte of the value (3 decimal) and TL0 will contain the low byte of the value (232 decimal). Reviewing low/high byte notation, recall that you must multiply the high byte by 256 and add the low byte to calculate the final value. In this case:

$(TH0 \bullet 256) + TL0 = 1000$

$(3 \bullet 256) + 232 = 1000$

Timer 1 works the exact same way, but its SFRs are TH1 and TL1.

It is apparent that the maximum value a timer may have is 65,535 because there are only two bytes devoted to the value of each timer. If a timer contains the value 65,535 and is subsequently incremented, it will reset—or overflow—back to 0.

8.3.3 TMOD SFR

The TMOD SFR is used to control the mode of operation of both timers. Each bit of the SFR gives the microcontroller specific information concerning how to run a timer. The high four bits (bits 4 through 7) relate to Timer 1, whereas the low four bits (bits 0 through 3) perform the exact same functions, but for Timer 0.

The individual bits of TMOD have the following functions:

	7	6	5	4	3	2	1	0	Reset Value
	TIMER 1				TIMER 0				
SFR 89_H	GATE	$C/\overline{T}$	M1	M0	GATE	$C/\overline{T}$	M1	M0	00_H

GATE (bit 7)—**Timer 1 Gate Control.** This bit enables/disables the ability of Timer 1 to increment.

0: Timer 1 will clock when TR1 = 1, regardless of the state of pin $\overline{\text{INT1}}$.

1: Timer 1 will clock only when TR1 = 1 and pin $\overline{\text{INT1}}$ = 1.

$C/\overline{T}$ (bit 6)—**Timer 1 Counter/Timer Select.**

0: Timer is incremented by internal clocks.

1: Timer is incremented by pulses on pin T1 when TR1 (TCON.6, SFR 88_H) is 1.

M1, M0 (bits 5-4)—**Timer 1 Mode Select.** These bits select the operating mode of Timer 1.

M1	M0	Mode
0	0	Mode 0: 8-bit counter with 5-bit prescale.
0	1	Mode 1: 16 bits.
1	0	Mode 2: 8-bit counter with auto-reload.
1	1	Mode 3: Timer 1 is halted, but holds its count.

GATE (bit 3)—**Timer 0 Gate Control.** This bit enables/disables the ability of Timer 0 to increment.

0: Timer 0 will clock when TR0 = 1, regardless of the state of pin $\overline{\text{INT0}}$ (software control).

1: Timer 0 will clock only when TR0 = 1 and pin $\overline{\text{INT0}}$ = 1 (hardware control).

$C/\overline{T}$ (bit 2)—**Timer 0 Counter/Timer Select.**

0: Timer is incremented by internal clocks.

1: Timer is incremented by pulses on pin T0 when TR0 (TCON.4, SFR 88H) is 1.

M1, M0 (bits 1-0) **Timer 0 Mode Select.** These bits select the operating mode of Timer 0.

M1	M0	Mode
0	0	Mode 0: 8-bit counter with 5-bit prescale.
0	1	Mode 1: 16 bits.
1	0	Mode 2: 8-bit counter with auto-reload.
1	1	Mode 3: Timer 1 is halted, but holds its count.

As is shown in the previous chart, four bits (two for each timer) are used to specify a mode of operation. The modes of operation are shown in Table 8–2.

Table 8–2. Timer Modes and Usage

TxM1	TxM0	Timer Mode	Description of Timer Mode	Timer 1	Timer 0
0	0	0	13-bit timer/counter	Y	Y
0	1	1	16-bit timer/counter	Y	Y
1	0	2	8-bit timer/counter with auto-reload	Y	Y
1	1	3	Two 8-bit counters (split timer mode)	N	Y

The TMOD.GATE bit controls gating of the timer/counter. If TMOD.GATE is cleared, the timer/counter increments only if TCON.TRx is set. If TMOD.GATE is set, the timer/counter increments only if TCON.TRx is set *and* the corresponding INTx pin is held high. This feature can be used for pulse width measurements.

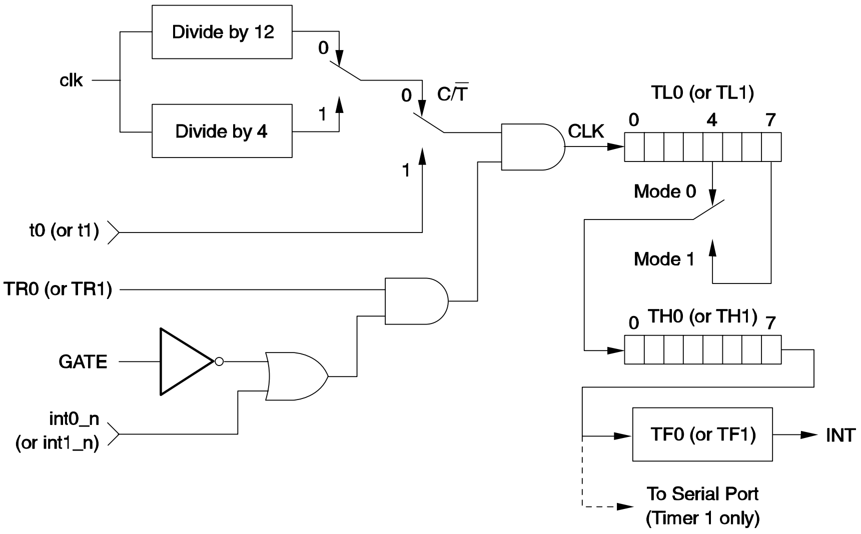
The TMOD.C$\overline{\text{T}}$ bit selects counter or timer operation. If TMOD.C$\overline{\text{T}}$ is cleared, the timer/counter register is incremented on either $f_{OSC}/4$ or $f_{OSC}/12$ (based on the state of CKCON.TxM). If TMOD.C$\overline{\text{T}}$ is set, the timer/counter register is incremented by the Tx pin.

8.3.3.1 13-Bit Time Mode (mode 0)

Timer mode 0 is a 13-bit timer. This is a relic that was kept around in the 8052 (and subsequently MSC1210) to maintain compatibility with its predecessor, the 8048. The 13-bit timer mode is not normally used in new development.

In this mode, the timer/counter uses five bits of the TLx register and all eight bits of the THx register for the 13-bit register. Therefore, the upper three bits of TLx must be masked if they are used by software. When the timer/counter rolls over on a transition from $01FFF_H$, the timer/counter interrupt flag is set (TCON.TFx).

Figure 8–1. Timer 0/1 Block Diagram for Modes 0 and 1



When the timer is in 13-bit mode, TLx will count from 0 to 31. When TLx is incremented from 31, it will roll over to 0 and overflow into THx, thus incrementing it. Therfore, only 13 bits of the two timer bytes are being used: bits 0 to 4 of TLx, and bits 0 to 7 of THx. This also means the timer can only contain 8192 values. If you set a 13-bit timer to 0, it overflows back to zero 8192 instruction cycles later.

There is very little reason to use this mode and it is only mentioned so there will be no surprise if ever analyzing archaic code that has been passed down through the generations.

8.3.3.2 16-Bit Time Mode (mode 1)

Mode 1 operates in the same manner as mode 0, except Timer 0 or Timer 1 is configured as a 16-bit timer/counter. The timer/counter uses all 8 bits of both the TLx register and THx register for the 16-bit register.

When the timer/counter rolls over on a transition from $0FFFF_H$, the timer/counter interrupt flag is set (TCON.TFx).

Timer mode 1 is a 16-bit timer. This is a very commonly used mode. It functions just like 13-bit mode, except that all 16 bits are used.

TLx is incremented from 0 to 255. When TLx is incremented from 255, it resets to 0 and causes THx to be incremented by 1. The timer may contain up to 65 536 distinct values because this is a full 16-bit timer. If a 16-bit timer is set to 0, it will overflow back to 0 after 65 536 machine cycles.

8.3.3.3 8-Bit Auto-Reload Time Mode (mode 2)

Timer mode 2 is an 8-bit auto-reload mode. When a timer is in mode 2, THx holds the reload value and TLx is the timer itself.

TLx starts counting up. When TLx reaches 255 and is subsequently incremented instead of resetting to 0 (as in the case of modes 0 and 1), it will be reset to the value stored in THx.

For example, TH0 holds the value FD_H and TL0 holds the value FE_H. Table 8–3 shows what would occur if the values of TH0 and TL0 are viewed for a few machine cycles.

Table 8–3. Example of 8-Bit Auto-Reload

Instruction Cycle	TH0 Value	TL0 Value
1	FD_H	FE_H
2	FD_H	FF_H
3	FD_H	FD_H
4	FD_H	FE_H
5	FD_H	FF_H
6	FD_H	FD_H
7	FD_H	FE_H

As shown, the value of TH0 never changed. In fact, when mode 2 is used, THx is almost always set to a known value and TLx is the SFR that is constantly incremented. THx is initialized once, and then left unchanged.

The benefit of auto-reload mode is that, perhaps, the timer may need to always have a value from 200 to 255. When using mode 0 or 1, the code would have to be checked to see if the timer had overflowed and, if so, the timer reset to 200. This takes precious amounts of execution time to check the value and/or reload it.

When mode 2 is used, the microcontroller takes care of this. Once a timer has been configured in mode 2, it does not have to be checked to see if the timer has overflowed, nor does the value need to be reset—the microcontroller hardware will do it all.

The auto-reload mode is very commonly used for establishing a baud rate, which will be discussed further in Chapter 9, *Serial Communications*.

8.3.3.4 Split-Timer Mode (mode 3)

Timer mode 3 is a split-timer mode. When Timer 0 is placed in mode 3, it essentially becomes two separate 8-bit timers. That is to say, Timer 0 is TL0 and Timer 1 is TH0. Both timers count from 0 to 255 and overflow back to 0. All the bits that are related to Timer 1 will now be tied to TH0, and all the bits related to Timer 0 will be tied to TL0.

While Timer 0 is in split mode, the real Timer 1 (i.e. TH1 and TL1) can be put into modes 0, 1, or 2 normally. However, the real Timer 1 may not be started or stopped, because the bits that do that are now linked to TH0. The real Timer 1, in this case, will be incremented every machine cycle no matter what. The real Timer 1 may be stopped by setting it to mode 3.

The only real use of note in using split-timer mode is if two separate timers are needed along with a baud rate generator. In such a case, use the real Timer 1 as a baud rate generator, and use TH0/TL0 as two separate timers.

8.3.4 TCON SFR

Finally, there is one more SFR that controls the two timers and provides valuable information about them. The TCON SFR has the structure described in Table 8–4.

Table 8–4. TCON (88_H) SFR

Bit	Name	Bit Address	Explanation of Function	Timer
7	TF1	$8F_H$	Timer 1 overflow. This bit is set by the microcontroller when Timer 1 overflows.	1
6	TR1	$8E_H$	Timer 1 run. When this bit is set, Timer 1 is turned on. When this bit is clear, Timer 1 is off.	1
5	TF0	$8D_H$	Timer 0 overflow. This bit is set by the microcontroller when Timer 0 overflows.	0
4	TR0	$8C_H$	Timer 0 run. When this bit is set, Timer 0 is turned on. When this bit is clear, Timer 0 is off.	0

So far, only four of the eight bits have been defined. That is because the other four bits of the SFR do not have anything to do with timers—they have to do with interrupts and they will be discussed in Chapter 10, *Interrupts*.

Table 8–4 contains the bit address column because this SFR is bit-addressable. That means in order to set bit TF1, which is the highest bit of TCON, you execute the command:

```
MOV TCON, #80h
```

However, because this SFR is bit-addressable, you can execute the command:

```
SETB TF1
```

This has the benefit of setting the high bit of TCON without changing the value of any of the other bits of the SFR. Usually, when starting or stopping a timer, the other values in TCON should not be modified, so take advantage of the fact that the SFR is bit-addressable.

8.3.5 Initializing a Timer

After discussing the timer-related SFRs, it is time to write code that initializes the timer and starts it running. As shown previously, the timer mode should be decided upon. In this case, a 16-bit timer that runs continuously will be used; that is to say, it is not dependent on any external pins.

We must first initialize the TMOD SFR. When working with Timer 0, the low four bits of TMOD will be used. The first two bits, GATE0 and C$\overline{T}$0 are both 0, because the timer needs to be independent of the external pins. 16-bit mode is timer mode 1, so T0M1 must be cleared and T0M0 must be set. Effectively, bit 0 of TMOD is the only bit that should be turned on. Therefore, to initialize the timer, execute the instruction:

```
MOV TMOD,#01h
```

Timer 0 is now in 16-bit timer mode, however, the timer is not running. To start the timer running, set the TR0 bit by executing the instruction:

```
SETB TR0
```

Upon executing these two instructions, Timer 0 will immediately begin counting, being incremented once every instruction cycle (every 12 crystal pulses).

8.3.6 Reading the Timer

There are two common ways of reading the value of a 16-bit timer; which one is used depends on the specific application. The actual value of the timer may be read as a 16–bit number, or the timer may be detected when overflowed.

8.3.6.1 Reading the Value of a Timer

If the timer is in 8-bit mode—that is, either 8-bit auto-reload mode, or in split-timer mode—reading the value of the timer is simple. Just read the 1-byte value of the timer and that is it.

However, when dealing with a 13-bit or 16-bit timer, the chore is a little more complicated. Consider what happens when the low byte of the timer is read as 255, then the high byte of the timer is read as 15. In this case, what actually happens is that the timer value is 14/255 (high byte 14, low byte 255) but the readout is 15/255. The reason for this is because the low byte was read as 255. However, when the next instruction is executed, enough time passes for the timer to increment again, which rolls the value over from 14/255 to 15/0. In the process, the timer is read as being 15/255 instead of 14/255. Obviously, this is a problem.

The solution is not complicated. Read the high byte of the timer, then read the low byte, then read the high byte again. If the high byte read the second time is not the same as the high byte read the first time you repeat the cycle. In code, this would appear as:

```
REPEAT:

      MOV A,TH0

      MOV R0,TL0

      CJNE A,TH0,REPEAT

   ...
```

In this case, the accumulator is loaded with the high byte of Timer 0. Then R0 is loaded with the low byte of Timer 0. Finally, the high byte we read out of Timer 0—which is now stored in the accumulator—is checked to see if it is the same as the current Timer 0 high byte. If it is not, that means it just rolled over and the timer value must be reread, which is done by going back to REPEAT. When the loop exits, the low byte of the timer is in R0 and the high byte is in the accumulator.

Another much simpler alternative is to simply turn off the timer run bit (i.e. CLR TR0), read the timer value, and then turn on the timer run bit (i.e. SETB TR0). In this case, the timer is not running, so no special tricks are necessary. Of course, this implies that the timer will be stopped for a few instruction cycles. Whether or not this is tolerable depends on the specific application.

8.3.6.2 Detecting Timer Overflow

Often it is only necessary to know that the timer has reset to 0. That is to say, there is no particular interest in the value of the timer, but rather an interest in knowing when the timer has overflowed back to 0.

Whenever a timer overflows from its highest value back to 0, the microcontroller automatically sets the TFx bit in the TCON register. This is useful because, rather than checking the exact value of the timer, you can just check if the TFx bit is set. If the TF0 bit is set, it means that Timer 0 has overflowed; if TF1 is set, it means that Timer 1 has overflowed.

This approach can be used to cause the program to execute a fixed delay. As shown earlier, we calculated that it takes the 8051 1/20th of a second to count from 0 to 46 080. However, the TFx flag is set when the timer overflows back to 0.

Therefore, to use the TFx flag to indicate when 1/20th of a second has passed, the timer must be set initially to 65 536 less 46 080, or 19 456. If the timer is set to 19 456, 1/20th of a second later the timer will overflow. Thus, the following code will execute a pause of 1/20th of a second:

```
MOV TH0,#76  ;High byte of 19,456 (76 * 256 = 19,456)

MOV TL0,#00  ;Low byte of 19,456 (19,456 + 0 = 19,456)

MOV TMOD,#01 ;Put Timer 0 in 16–bit mode

CLR TF0      ;Make sure TF0 bit is clear initially

SETB TR0     ;Make Timer 0 start counting

JNB TF0,$    ;If TF0 is not set, jump back to this same
             ;instruction
```

In the above code, the first two lines initialize the Timer 0 starting value to 19 456. The next two instructions configure Timer 0 and turn it on. Finally, the last instruction (JNB TF0,$) reads: jump back to the same instruction if TF0 is not set. The $ operand means, in most assemblers, the address of the current instruction.

As long as the timer has not overflowed and the TF0 bit has not been set, the program will keep executing this same instruction. After 1/20th of a second, Timer 0 overflows, sets the TF0 bit, and program execution then breaks out of the loop.

8.3.7 Timing the Length of Events

The MSC1210 provides another useful method to time the length of events.

For example, in order to save electricity in the office, a light switch is measured to see how long it is turned on each day. When the light is turned on, time must be measured; when the light is turned off, time is not measured. One option is to connect the light switch to one of the pins, constantly read the pin, and turn the timer on or off based on the state of that pin. Although this method works well, the MSC1210 provides an easier way of accomplishing this.

Looking again at the TMOD SFR, there is a bit called GATE0. So far, this bit has always been cleared because the timer is run regardless of the state of the external pins. However, now it would be nice if an external pin could control whether the timer was running or not. It can.

Simply connect the light switch to pin $\overline{\text{INT0}}$ (P3.2) on the MSC1210 and set the bit GATE0. When GATE0 is set, Timer 0 will only run if P3.2 is high. When P3.2 is low (i.e., the light switch is off) the timer will automatically be stopped.

Thus, with no control code whatsoever, the external pin P3.2 can control whether or not the timer is running.

8.4 Using Timers as Event Counters

We have discussed how a timer can be used for the obvious purpose of keeping track of time. However, the MSC1210 also allows the use of timers to count events.

This can be useful in many applications. For example, a sensor is placed across a road that would send a pulse every time a car passes over it. This could be used to determine the volume of traffic on the road. The sensor is attached to one of the MSC1210 I/O lines and constantly monitored, detecting when it pulses high, and the counter incremented when it goes back to a low state. This is not terribly difficult, but requires some code. If the sensor is hooked to P1.0, the code to count passing cars would look something like this:

```
JNB P1.0,$    ;If a car hasn’t raised the signal,
              ;keep waiting

JB P1.0,$     ;The line is high, car is on the sensor
              ;right now

INC COUNTER   ;The car has passed completely, so we count it
```

As shown, it is only three lines of code. However, what if other processing needs to be done at the same time? The program cannot be stuck in the JNB P1.0,$ loop waiting for a car to pass if it needs to be doing other things. What if the program is doing other things when a car passes over? It is possible that the car will raise the signal and the signal will fall low again before the program checks the line status; this would result in the car not being counted. Of course, there are ways to get around even this limitation, but the code quickly becomes big, complex, and ugly.

Luckily, the MSC1210 provides a way to use the timers to count events. It is painfully easy. Only one additional bit has to be configured.

Timer 0 can be used to count the number of cars that pass. In the bit table for the TCON SFR, there is a bit called $C/\overline{T}0$—it is bit 2 (TCON.2). Reviewing the explanation of the bit, we see that if the bit is clear, Timer 0 will be incremented every instruction cycle. This is what has already been used to measure time.

If $C/\overline{T}0$ is set, however, Timer 0 will monitor the P3.4 line. Instead of being incremented every machine cycle, Timer 0 will count events on the P3.4 line. So in this case, simply connect the sensor to P3.4 and let the 8052 do the work. Then, when the number of how many cars have passed is desired, just read the value of Timer 0—the value of Timer 0 will be the number of cars that have passed.

So what exactly is an event? What does Timer 0 actually count? Speaking at the electrical level, the MSC1210 counts 1–0 transitions on the P3.4 line. This means that when a car first runs over the sensor, it raises the input to a high (1) condition. At that point, the MSC1210 does not count anything because this is a 0–1 transition. However, when the car has passed, the sensor falls back to a low (“0”) state. This is a 1–0 transition and at that instant the counter is incremented by 1.

It is important to note that the MSC1210 checks the P3.4 line each instruction cycle (4 clock cycles). This means that if P3.4 is low, goes high, and goes back low in 3 clock cycles, it will probably not be detected by the MSC1210. This also means the MSC1210 event counter is only capable of counting events that occur at a maximum of 1/8th the rate of the crystal frequency. That is to say, if the crystal frequency is 12.000MHz, it can count a maximum of 1 500 000 events per second (12.000MHz • 1/8 = 1 500 000). If the event being counted occurs more than 1 500 000 times per second, the MSC1210 will not be able to accurately count the event without using additional external circuitry or a faster crystal.

8.5 Using Timer 2

The MSC1210 has a third timer, Timer 2, which functions slightly differently than Timers 0 and 1 and, for that reason, we are addressing this third timer separately from the first two.

8.5.1 T2CON SFR

The operation of Timer 2 (T2) is controlled almost entirely by the T2CON SFR, at address C8$_H$. This SFR is bit-addressable because this SFR is evenly divisible by eight.

The individual bits of T2CON have the following functions:

	7	**6**	**5**	**4**	**3**	**2**	**1**	**0**	**Reset Value**
SFR C8$_H$	TF2	EXF2	RCLK	TCLK	EXEN2	TR2	C/$\overline{T}$	CP/$\overline{RL2}$	00$_H$

TF2 (bit 7)—**Timer 2 Overflow Flag.** This flag will be set when Timer 2 overflows from FFFF$_H$. It must be cleared by software. TF2 will only be set if RCLK and TCLK are both cleared to 0. Writing a 1 to TF2 forces a Timer 2 interrupt if enabled.

EXF2 (bit 6) **Timer 2 External Flag.** A negative transition on the T2EX pin (P1.1) will cause this flag to be set based on the EXEN2 (T2CON.3) bit. If set by a negative transition, this flag must be cleared to 0 by software. Setting this bit in software will force a timer interrupt, if enabled.

RCLK (bit 5)—**Receive Clock Flag.** This bit determines the serial Port 0 time-base when receiving data in serial modes 1 or 3.

0 = Timer 1 overflow is used to determine receiver baud rate for serial Port 0.

1 = Timer 2 overflow is used to determine receiver baud rate for serial Port 0.

Setting this bit will force Timer 2 into baud rate generation mode. The timer will operate from a divide by 2 of the external clock.

TCLK (bit 4)—**Transmit Clock Flag.** This bit determines the serial Port 0 timer base when transmitting data in serial modes 1 or 3.

0 = Timer 1 overflow is used to determine transmitter baud rate for serial Port 0.

1 = Timer 2 overflow is used to determine transmitter baud rate for serial Port 0.

Setting this bit will force Timer 2 into baud rate generation mode. The timer will operate from a divide by 2 of the external clock.

EXEN2 (bit 3)—**Timer 2 External Enable**. This bit enables the capture/reload function on the T2EX pin if Timer 2 is not generating baud rates for the serial port.

0 = Timer 2 will ignore all external events at T2EX.

1 = Timer 2 will capture or reload a value if a negative transition is detected on the T2EX pin.

TR2 (bit 2)—**Timer 1 Run Control.** This bit enables/disables the operation of Timer 2. Halting this timer will preserve the current count in TH2, TL2.

0 = Timer 2 is halted.

1 = Timer 2 is enabled.

C/$\overline{\text{T}}$ (bit 1)—**Counter/Timer Select.** This bit determines whether Timer 2 will function as a timer or counter. Independent of this bit, Timer 2 runs at 2 clocks per tick when used in baud rate generator mode.

0 = Timer 2 functions as a timer. The speed of Timer 2 is determined by the T2M bit (CKCON.5).

1 = Timer 2 will count negative transitions on the T2 pin (P1.0).

CP/$\overline{\text{RL2}}$ (bit 0)—**Capture/Reload Select.** This bit determines whether the capture or reload function will be used for Timer 2. If either RCLK or TCLK is set, this bit will not function and the timer will function in an auto-reload mode following each overflow.

0 = Auto-reloads will occur when Timer 2 overflows or a falling edge is detected on T2EX if EXEN2 = 1.

1 = Timer 2 captures will occur when a falling edge is detected on T2EX if EXEN2 = 1.

8.5.2 Timer 2 in Auto-Reload Mode

The first mode in which Timer 2 may be used is auto-reload. The auto-reload mode functions just like Timer 0 and Timer 1 in auto-reload mode, except that the Timer 2 auto-reload mode performs a full 16-bit reload (recall that Timer 0 and Timer 1 only have 8-bit reload values). When a reload occurs, the value of TH2 is reloaded with the value contained in RCAP2H, and the value of TL2 is reloaded with the value contained in RCAP2L.

To operate Timer 2 in auto-reload mode, the CP/RL2 bit (T2CON.0) must be clear. In this mode, Timer 2 (TH2/TL2) is reloaded with the reload value (RCAP2H/RCAP2L) whenever Timer 2 overflows; that is to say, whenever Timer 2 overflows from FFFF_H back to 0000_H. An overflow of Timer 2 causes the TF2 bit to be set, which causes an interrupt to be triggered (if Timer 2 interrupt is enabled). Note that TF2 will not be set on an overflow condition if either RCLK or TCLK (T2CON.5 or T2CON.4) are set.

Additionally, by also setting EXEN2 (T2CON.3), a reload will also occur whenever a 1–0 transition is detected on T2EX (P1.1). A reload that occurs as a result of such a transition causes the EXF2 (T2CON.6) flag to be set, triggering a Timer 2 interrupt, if enabled.

8.5.3 Timer 2 in Capture Mode

A new mode, specific to Timer 2, is called capture mode. As the name implies, this mode captures the value of Timer 2 (TH2 and TL2) into the capture SFRs (RCAP2H and RCAP2L). To put Timer 2 in capture mode, CP/RL2 (T2CON.0) and EXEN2 (T2CON.3) must be set.

When configured as mentioned above, a capture occurs whenever a 1-0 transition is detected on T2EX (P1.1). At the moment the transition is detected, the current values of TH2 and TL2 is copied into RCAP2H and RCAP2L, respectively. At the same time, the EXF2 (T2CON.6) bit is set, which triggers an interrupt, if Timer 2 interrupt is enabled.

Note:

Even in capture mode, an overflow of Timer 2 results in TF2 being set and an interrupt being triggered.

Note:

Capture mode is an efficient way to measure the time between events. At the moment that an event occurs, the current value of Timer 2 is copied into RCAP2H/L. However, Timer 2 will not stop and an interrupt will be triggered. Thus the interrupt routine must copy the value of RCAP2H/L to a temporary holding variable without stopping Timer 2. When another capture occurs, the interrupt can take the difference of the two values to determine the time transpired. Again, the main advantage is that Timer 2 does not need to be stopped to have its value read, as is the case with Timer 0 and Timer 1.

8.5.4 Timer 2 as a Baud Rate Generator

Timer 2 can be used as a baud rate generator. This is accomplished by setting either RCLK (T2CON.5) or TCLK (T2CON.4).

With Timer 1, the receive and transmit baud rate must be the same. With Timer 2, however, the user can configure the serial port to receive at one baud rate and transmit at another. For example, if RCLK is set and TCLK is cleared, serial data is received at the baud rate determined by Timer 2, whereas the baud rate of transmitted data is determined by Timer 1.

Determining the auto-reload values for a specific baud rate is discussed in Chapter 9, *Serial Communication*. The only difference is that in the case of Timer 2, the auto-reload value is placed in RCAP2H and RCAP2L, and the value is 16-bit rather than 8-bit.

Note:

When Timer 2 is used as a baud rate generator (either TCLK or RCLK are set), the Timer 2 overflow flag (TF2) is not set.

Chapter 9

Serial Communication

Chapter 9 describes serial communication using the MSC1210 ADC.

Topic **Page**

9.1 Description

The MSC1210 family has three serial port interfaces: two UARTs and one SPI. This chapter will cover the UARTs, while the SPI will be covered Chapter 13, *Serial Peripheral Interface (SPI)*.

One of the many powerful features of the MSC1210 is its integrated UARTs, otherwise known as universal synchronous/asynchronous receiver/transmitters. Just as the name implies, the UART can be configured for either synchronous, half-duplex operation or asynchronous full-duplex (transmit and receive data simultaneously) operation.

The fact that the MSC1210 has integrated UARTs means that values may be very easily read from and written to the serial port. If it were not for the integrated UARTs, writing a byte to a serial line would be a rather tedious process requiring turning on and off one of the I/O lines in rapid succession to properly shift out each individual bit, including start bits, stop bits, and parity bits.

However, this does not have to be done. Instead, simply configure the serial ports operating modes and baud rates. Once configured, write to an SFR to write a value to the serial port or read the same SFR to read a value from the serial port. The MSC1210 automatically lets you know when it has finished sending the written character and also lets you know whenever it has received a byte, so that it can be processed. There is no need to worry about transmission at the bit level, which saves quite a bit of coding and processing time.

The UART serial port is asynchronous full–duplex (transmit and receive simultaneously) or synchronous half-duplex (transmit or receive). It also has a receiver buffer, to enable the UART to continue to receive a second byte before the first byte has been read in software. If the first byte has not been read when the second byte has been completely transmitted, the second byte will be lost. The serial port receive and transmit registers are both accessed through SBUF. Writing to SBUF loads the transmit buffer, and reading SBUF reads the receive register.

Note:

Although a standard 8052 has only one UART, the MSC1210 has two. This provides additional flexibility when integrating the part in a device that must communicate with more than one external serial devices. This chapter explains how to use the primary UART (Serial Port 0); using the secondary UART (Serial Port 1) is identical. Just use the SFRs that refer to port 1 instead of port 0 (i.e., SCON1 instead of SCON0, etc.). Also note that the secondary UART cannot use Timer 2 as a baud rate clock, while the primary UART can.

9.2 Setting the Serial Port Mode

The first thing to be done when using the MSC1210 integrated serial port is, obviously, to configure it. This lets you tell the MSC1210 how many data bits are needed, the baud rate to be used, and how the baud rate will be determined.

First, the Serial Control 0 (SCON0) SFR is presented and what each bit of the SFR represents is defined. Remember, SCON1 has the exact same function but relates to the secondary UART.

The individual bits of SCON0 have the following functions:

	7	**6**	**5**	**4**	**3**	**2**	**1**	**0**	**Reset Value**
SFR 98_H	SM0_0	SM1_0	SM2_0	REN_0	TB8_0	RB8_0	TI_0	RI_0	00_H

SM0–2 (bits 7–5)—**Serial Port 0 Mode.** These bits control the mode of Serial Port 0. Modes 1, 2, and 3 have one start and one stop bit in addition to the eight or nine data bits.

Mode	SM0	SM1	SM2	Function	Length	Period
0	0	0	0	Synchronous	8 bits	12 p_{CLK}(1)
0	0	0	1	Synchronous	8 bits	4 p_{CLK}(1)
1	0	1	x	Asynchronous	10 bits	Timer 1 or 2 baud rate equation
2	1	0	0	Asynchronous	11 bits	64 p_{CLK}(1) (SMOD = 0) 32 p_{CLK}(1) (SMOD = 1)
2	1	0	1	Asynchronous with multiprocessor communication	11 bits	64 p_{CLK}(1) (SMOD = 0) 32 p_{CLK}(1) (SMOD = 1)
3	1	1	0	Asynchronous	11 bits	Timer 1 or 2 baud rate equation
3	1	1	1	Asynchronous with multiprocessor communication	11 bits	Timer 1 or 2 baud rate equation

Notes: 1) p_{CLK} will be equal to t_{CLK}, except that p_{CLK} will stop for IDLE.

REN_0 (bit 4)—**Receive Enable.** This bit enables/disables the Serial Port 0 received shift register.

0: Serial Port 0 reception disabled.

1: Serial Port 0 received enabled (modes 1, 2, and 3). Initiate synchronous reception (mode 0).

TB8_0 (bit 3)—**Ninth Transmission Bit State.** This bit defines the state of the ninth transmission bit in Serial Port 0 modes 2 and 3.

RB8_0 (bit 2)—**Ninth Received Bit State.** This bit identifies the state of the ninth reception bit of received data in Serial Port 0 modes 2 and 3. In serial port mode 1, when SM2_0 = 0, RB8_0 is the state of the stop bit. RB8_0 is not used in mode 0.

TI_0 (bit 1)—**Transmitter Interrupt Flag.** This bit indicates that data in the Serial Port 0 buffer has been completely shifted out. In serial port mode 0, TI_0 is set at the end of the eighth data bit. In all other modes, this bit is set at the end of the last data bit. This bit must be manually cleared by software.

RI_0 (bit 0)—**Receiver Interrupt Flag.** This bit indicates that a byte of data has been received in the Serial Port 0 buffer. In serial port mode 0, RI_0 is set at the end of the eighth bit. In serial port mode 1, RI_0 is set after the last sample of the incoming stop bit subject to the state of SM2_0. In modes 2 and 3, RI_0 is set after the last sample of RB8_0. This bit must be manually cleared by software.

Additionally, it is necessary to define the function of SM0 and SM1, as shown in Table 9–1.

The SCON0 SFR allows us to configure the primary serial port. Go through each bit and review its function.

The low four bits (bits 0 through 3) are operational bits. They are used when actually sending and receiving data; they are not used to configure the serial port.

The TB8 bit is used in modes 2 and 3, which transmit a total of nine data bits. The first eight data bits are the eight bits of the main value, and the ninth bit is taken from TB8. If TB8 is set and a value is written to the serial port, the bits of the data will be written to the serial line followed by a set ninth bit. If TB8 is clear, the ninth bit will be clear.

The RB8 bit also operates in modes 2 and 3 and functions essentially the same way as TB8, but on the reception side. When a byte is received in modes 2 or 3, a total of nine bits are received. In this case, the first eight bits received are the data of the serial byte received, and the value of the ninth bit received will be placed in RB8.

TI means transmit interrupt. When a program writes a value to the serial port, a certain amount of time passes before the individual bits of the byte are shifted out of the serial port. If the program writes another byte to the serial port before the first byte is completely output, the data being sent is intertwined. Therefore, the MSC1210 lets the program know that it has shifted out the last byte by setting the TI bit. When the TI bit is set, the program assumes that the serial port is free and ready to send the next byte.

Finally, the RI bit means receive interrupt. It functions similarly to the TI bit, but it indicates that a byte has been received. That is to say, whenever the MSC1210 receives a complete byte, it triggers the RI bit to let the program know that it needs to read the value quickly, before another byte is read.

Table 9–1. SM0 and SM1 Function Definitions.

MODE	Sync/Async	Baud Clock	Data Bits	Start/Stop	Ninth-Bit Function
0	Sync	clk/4 or clk/12	8	None	None
1	Async	Timer 1 or Timer 2(1)	8	1 Start, 1 Stop	None
2	Async	clk/32 or clk/64	9	1 Start, 1 Stop	0, 1, Parity
3	Async	Timer 1 or Timer 2(1)	9	1 Start, 1 Stop	0, 1, Parity

Notes: 1) Timer 2 available for Serial Port 0 only.

The high four bits (bits 4 through 7) are configuration bits.

The bit REN means receiver enable. This bit is very straightforward; if data need to be received via the serial port, set this bit. This bit will almost always need to be set because leaving it cleared will prevent the MSC1210 from receiving serial data.

The function of the SM2 bit depends on the serial mode. In mode 0, the SM2 bit is used to set the baud rate. When SM2 is cleared in this mode, the baud rate is $f_{OS}/12$. When SM2 is set in this mode, the baud rate is $f_{OSC}/4$. In mode 3, the SM2 bit is a flag for multiprocessor communication. Generally, whenever a byte has been received, the MSC1210 will set the RI flag. This lets the program know that a byte has been received and that it needs to be processed. However, when SM2 is set, the RI flag will only be triggered if the ninth bit received is a 1. That is to say, if SM2 is set and a byte is received whose ninth bit is clear, the RI flag will never be set. This can be useful in certain advanced serial applications that allow multiple MSC1210s (or other hardware) to communicate amongst themselves. For now, it is safe to say that this bit should almost always be clear so that the flag is set upon reception of any character.

Bits SM0 and SM1 let the serial mode be set to a value between 0 and 3, inclusive. The four modes are defined in Table 9–1. As shown, selecting the serial mode selects the mode of operation (8-bit/9-bit, UART, or shift register) and also determines how the baud rate will be calculated.

9.2.1 Serial Mode 0: Synchronous Half-Duplex

In mode 0, serial data transfers are eight bits long, half-duplex, and synchronous. The serial data are transmitted and received through the RXD pin. The shift clock is generated on the TXD pin. Eight bits are transmitted or received on each data transfer, LSB first. The data transmission begins when data are written to SBUF.

Data reception begins when the REN_0/REN_1 bit is set and the RI_0/RI_1 bit is cleared in the corresponding SCON SFR. The shift clock is activated and the UART shifts data in on each rising edge of the shift clock until eight bits have been received. One instruction cycle after the eighth bit is shifted in, the RI_0/RI_1 bit is set, and reception stops until the software clears the RI bit. The baud rate is either $f_{OSC}/12$ (if SCONx.5 is clear) or $f_{OSC}/4$ (if SCONx.5 is set).

RXD is used for serial TX and RX of data, LSB first. TXD is used as the baud clock. Transmission is initiated by any instruction that writes to SBUF.

Figure 9–1. Serial Port 0 Mode 0 Transmit Timing—High Speed Operation.

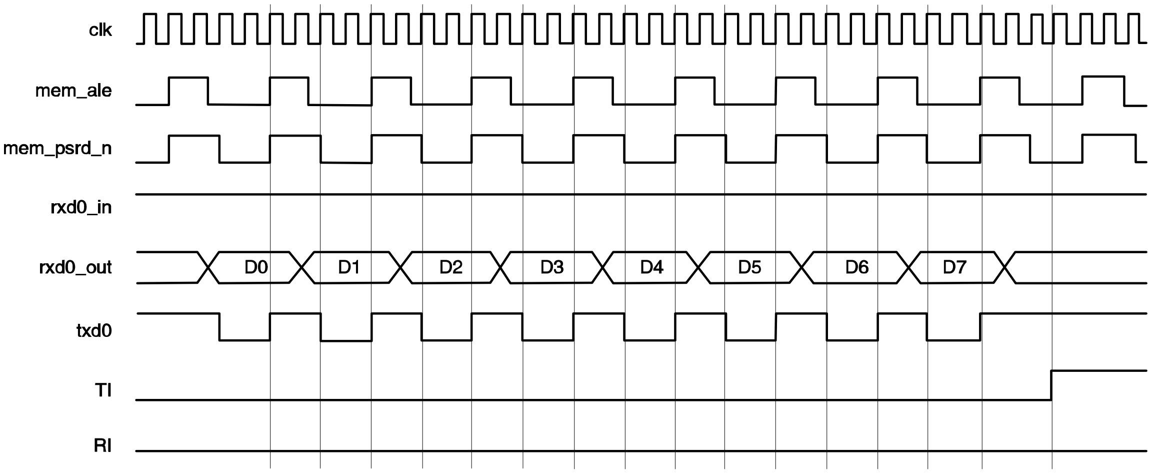
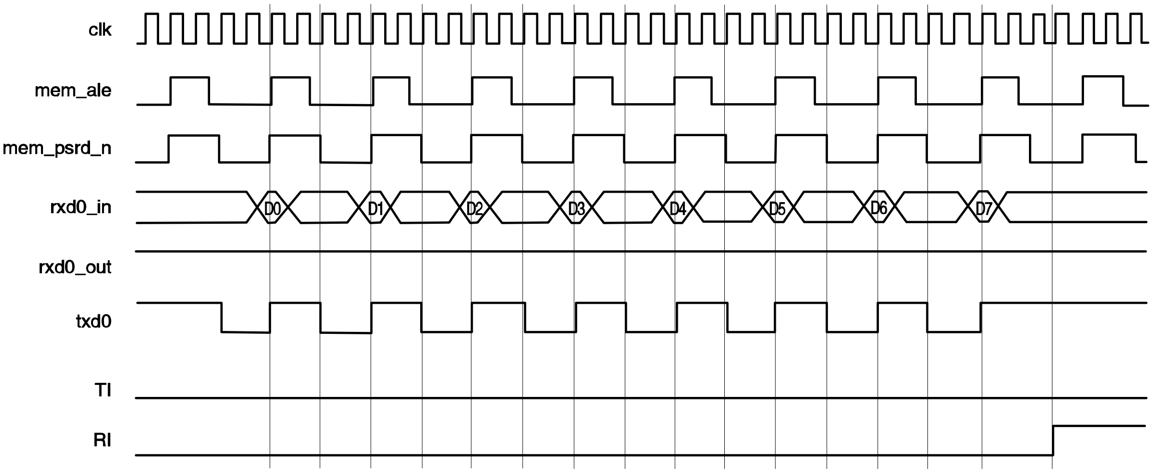


Figure 9–2. Serial Port Mode 0 Receive Timing—High Speed Operation.



9.2.2 Serial Mode 1: Asynchronous Full-Duplex

In mode 1, serial data transfers are 10 bits long, full-duplex, and asynchronous. The transfer begins with a start bit, followed by eight bits of data (LSB first), then a stop bit. On receive, the stop bit is shifted into the RB8 bit in the SCON register. The baud rate is set by Timer 1 (UART 0 or 1) or Timer 2 (UART 0).

RXD is used for receiving data and TXD is used for transmitting data, LSB first. On reception, the stop bit goes into RB8 in the SCON register. Transmission is initiated by any instruction that writes to SBUF. The transmission begins after the first rollover of the divide-by-16 counter after the write. The SCONx.TIx interrupt flag is set two f_{OSC} cycles after the stop bit has been transmitted.

Figure 9–3. Serial Port Mode 1 Transmit Timing.

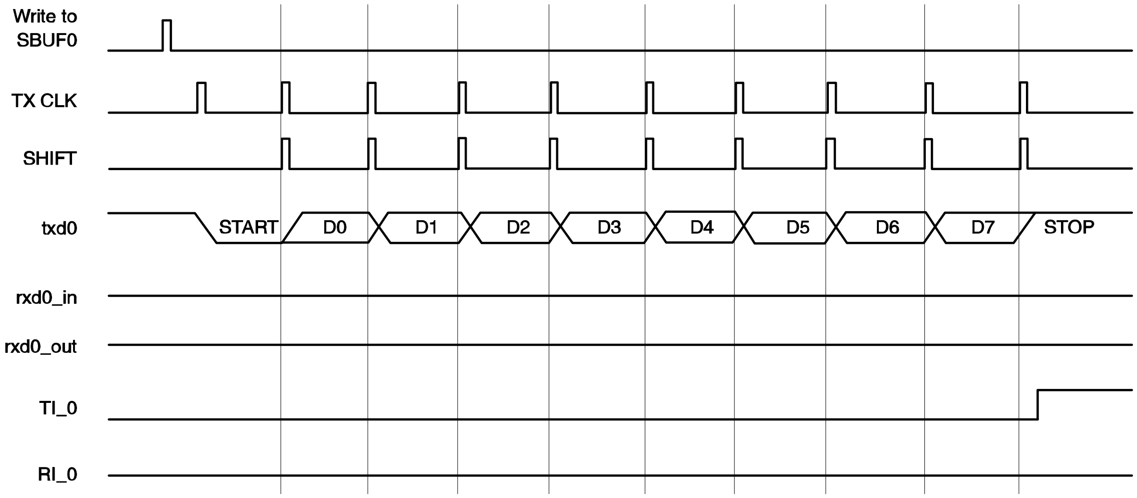
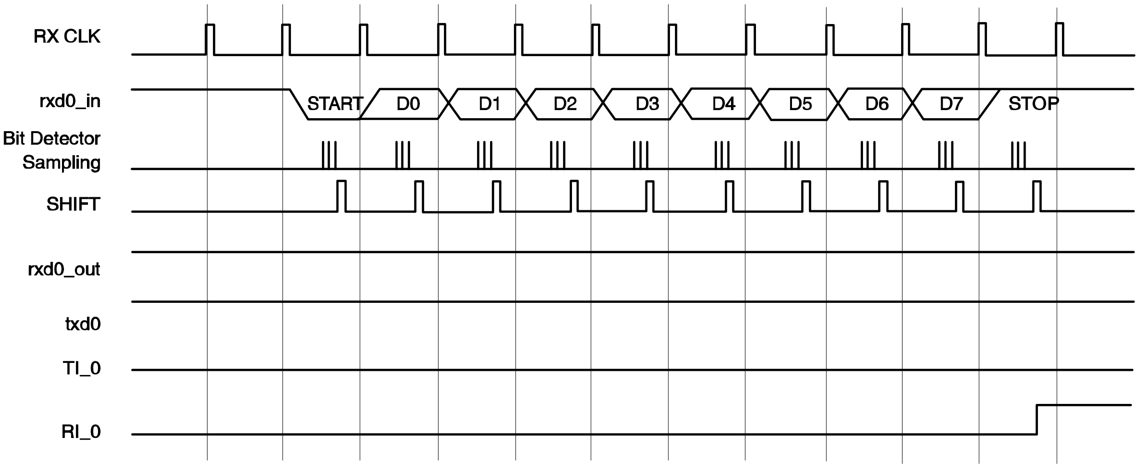


Figure 9–4. Serial Port 0 Mode 1 Receive Timing.



Reception is enabled by configuring SCON0.RBN = 1. Reception of the data begins at the falling edge of start-bit detection. The RXDx pin is sampled 16 times-per-bit for any baud rate setting. When the falling edge of the start bit is detected, the divide-by-16 counter used to generate the receive clock is reset to align the counter rollover with the bit boundaries. The state of each bit is determined by a majority detect decision on three consecutive samples in the middle of the bit; this provides an amount of noise rejection. At the middle of the stop-bit time, the serial port verifies that the status of SCONx.RI_x = 0 and SCON0.SM2_x = 1 (if SCON0.SM2_x = 0, the stop bit is a don't care). If these conditions are true, then the serial port writes the received byte to the SBUFx register, loads the stop bit into SCONx.RB8_x, and sets the SCONx.RI_x flag. If the conditions are not met, the data are ignored. After the middle of the stop bit, the serial port waits for another start-bit detection.

The baud rate is adjustable and is based on either Timer 1 or Timer 2. Serial Port 0 can use either Timer 1 or Timer 2, while Serial Port 1 can use only Timer 1. On an overflow from the timer, a clock is sent to the baud clock. The clock is divided by 16 to generate the baud clock. The PCON.SMOD0 and EICON.SMOD1 bits determine whether or not to divide Timer 1 by the rollover rate of 2. The equation for baud rate is given below:

$$BaudRate = \frac{2^{SMOD}}{32} \cdot Timer1Overflow$$

It is recommended to use Timer 1 in mode 2 (8-bit counter with auto-reload). This changes the equation to:

$$BaudRate = \frac{2^{SMOD}}{32} \cdot \frac{f_{OSC}}{12 \cdot (256 - TH1)}$$

The divide-by-12 can be changed to 4 by setting CKCON.T1M.

To determine the reload value from a given baud rate, use the equation below:

$$TH1 = 256 - \frac{2^{SMOD} \cdot f_{OSC}}{384 \cdot BaudRate}$$

You can also achieve very low baud rates from Timer 1 by enabling T1CON.TF1, configuring the timer for mode 1, and using the timer interrupt to initiate a 16-bit software reload, as shown in Table 9–2.

Table 9–2. Common Baud Rates Using Timer 1

Baud Rate	**SMODx**	**$C/\overline{T}$**	**Timer 1 Mode**	**TH1 Value for an 11.0592MHz f_{OSC}**
57.6k	1	0	2	$0FF_H$
19.2k	1	0	2	$0FD_H$
9.6k	1	0	2	$0FA_H$
4.8k	1	0	2	$0F4_H$
2.4k	1	0	2	$0E8_H$
1.2k	1	0	2	$0D0_H$

When using Timer 2 for the baud rate clock, the equation is:

$$BaudRate = \frac{Timer2Overflow}{16}$$

To use Timer 2 as the baud rate generator, configure Timer 2 in auto-reload mode and set T2CON.TCLK and T2CON.RCLK (to select Timer 2 as the baud-rate generator for the transmitter and receiver, respectively). Setting T2CON.TCLK and T2CON.RCLK will disable the setting of T2CON.TF2 and the reload on 1-to-0 on T2. The 16-bit reload value is stored in RCAP2L and RCAP2H, which gives the following equation:

$$BaudRate = \frac{f_{OSC}}{32 \cdot (65536 - (RCAP2H : RCAP2L))}$$

The divide-by-32 is a result of the f_{OSC} being divided by 2 (by setting T2CON.TCLK and T2CON.RCLK) and the Timer 2 overflow being divided by 16.

To determine the RCAP2H:RCAP2L value from a given baud rate use the equation below:

$$RCAP2H : RCAP2L = (65536 - \frac{f_{OSC}}{32 \cdot BaudRate})$$

Table 9–3. Common Baud Rates Using Timer 2

Baud Rate	$C/\overline{T}2$	RCAP2H:RCAP2L (@ 11.0592MHz f_{OSC})
57.6k	0	$0FFFA_H$
19.2k	0	$0FFEE_H$
9.6k	0	$0FFDC_H$
4.8k	0	$0FFB8_H$
2.4k	0	$0FF70_H$
1.2k	0	$0FEE0_H$

9.2.3 Serial Mode 2: Asynchronous Full-Duplex

In mode 2, serial data transfers are 11 bits long, full-duplex, and asynchronous. The transfer begins with a start bit, followed by eight bits of data (LSB first), an additional bit of data (ninth bit), then a stop bit. On transmit, the ninth data bit is set by TB8. On receive, the ninth bit is shifted into the RB8 bit in the SCON register and the stop bit is ignored. The baud rate is either $f_{OSC}/64$ or $f_{OSC}/12$.

RXD is used for receiving data, TXD is used for transmitting data, LSB first. On transmission, SCON.TB8 is used for the ninth bit. On reception the ninth bit goes into RB8 in the SCON register. The baud rate is selectable at $f_{OSC}/32$ or $f_{OSC}/64$.

Figure 9–5. Serial Port 0 Mode 2 Transmit Timing.

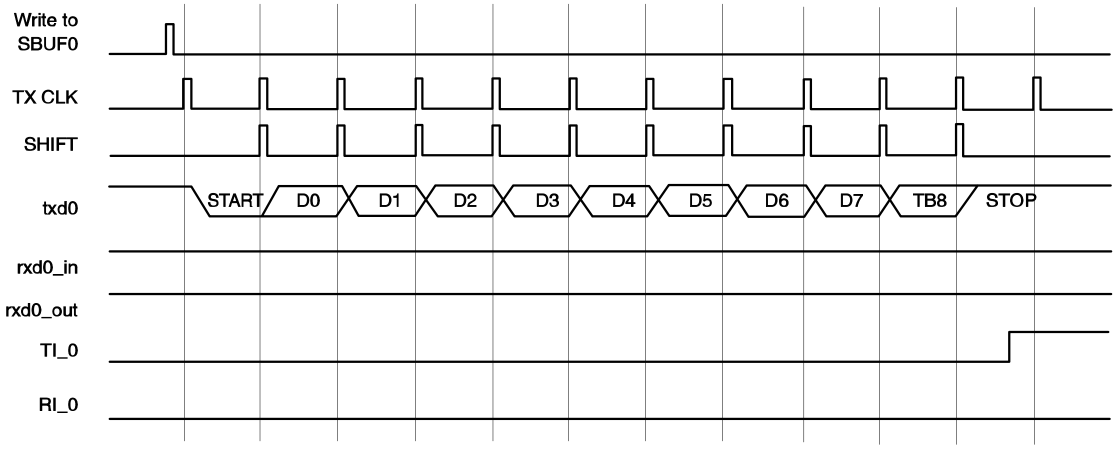
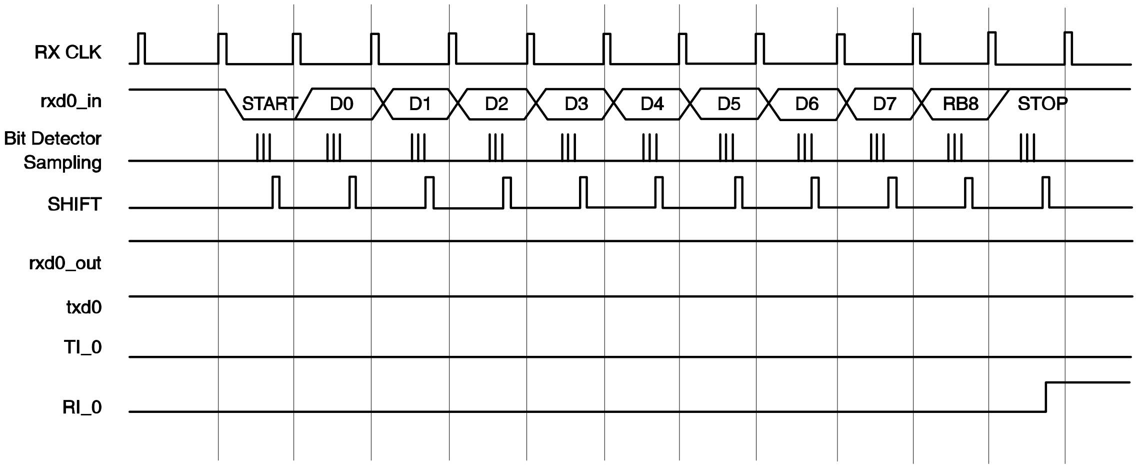


Figure 9–6. Serial Port 0 Mode 2 Receive Timing.



Transmission is initiated by any instruction that writes to SBUF. The transmission begins after the first rollover of the divide-by-16 counter after the write. The SCONx.Ti_x interrupt flag is set when the stop bit has been placed on the TXDx pin.

Reception is enabled by configuring SCON0.RBN = 1. Reception of the data begins at the falling edge of start-bit detection. The RXDx pin is sampled 16 times per bit for any baud rate setting. When the falling edge of the start bit is detected, the divide-by-16 counter used to generate the receive clock is reset to align the counter rollover with the bit boundaries. The state of each bit is determined by a majority detect decision on three consecutive samples in the middle of the bit, providing an amount of noise rejection. At the middle of the stop-bit time, the serial port verifies that the status of SCONx.RI_x = 0 and SCON0.SM2_x = 1 (if SCON0.SM2_x = 0, the stop bit is a “don’t care”). If these conditions are true, then the serial port writes the received byte to the SBUFx register, loads the stop bit into SCONx.RB8_x, and sets the SCONx.RI_x flag. If the conditions are not met, the data are ignored. After the middle of the stop bit, the serial waits for another start-bit detection.

The state of SCON0.SMODx determines the baud rate clock. The equation is:

$$BaudRate = \frac{2^{SMOD} \cdot f_{OSC}}{64}$$

Mode 2 has a special provision for multiprocessor communications. This mode is typically used when a master wants to address a specific slave device on the bus. The address of the target slave device is transmitted in the first eight data bits. The ninth bit is used to indicate to the slaves that the data was an address. If the data matches the slave address, the device can then resume normal reception. In this mode, nine data bits are received (the ninth bit is latched into SCON0.RB8). The port can be configured such that when the stop bit is received, the serial port interrupt will be generated if RB8 = 1. This feature is enabled by setting bit SCON0.SM2.

9.2.4 Serial Mode 3: Asynchronous Full-Duplex

In mode 3, serial data transfers are 11 bits, full-duplex, and asynchronous. Mode 3 is identical to mode 2, with the exception of the baud rate. The transfer begins with a start bit, followed by eight bits of data (LSB first), an additional bit of data (ninth bit), and then a stop bit. On transmit, the ninth data bit is set by TB8. On receive, the ninth bit is shifted into the RB8 bit in the SCON register and the stop bit ignored. The baud rate is set by Timer 1 (USART0 or 1) or Timer 2 (USART0).

RXD is used for receiving data, TXD is used for transmitting data, LSB first. On transmission, SCON.TB8 is used for the ninth bit. On reception, the ninth bit goes into RB8 in the SCON register. The baud rate is adjustable and is based on either Timer 1 or Timer 2.

Transmission is initiated by any instruction that writes to SBUF. The transmission begins after the first rollover of the divide-by-16 counter after the write. The SCONx.Ti_x interrupt flag is set when the stop bit has been placed on the TXDx pin.

Figure 9–7. Serial Port 0 Mode 3 Transmit Timing.

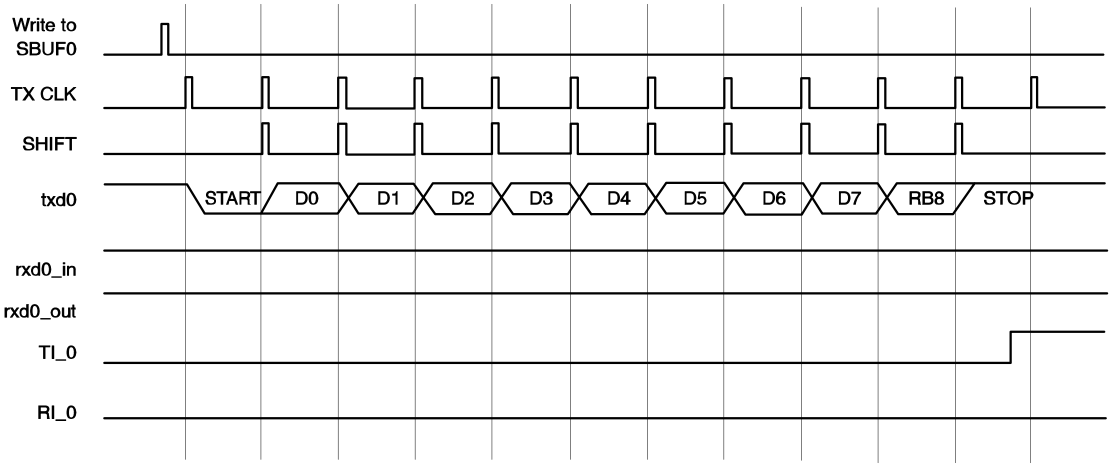
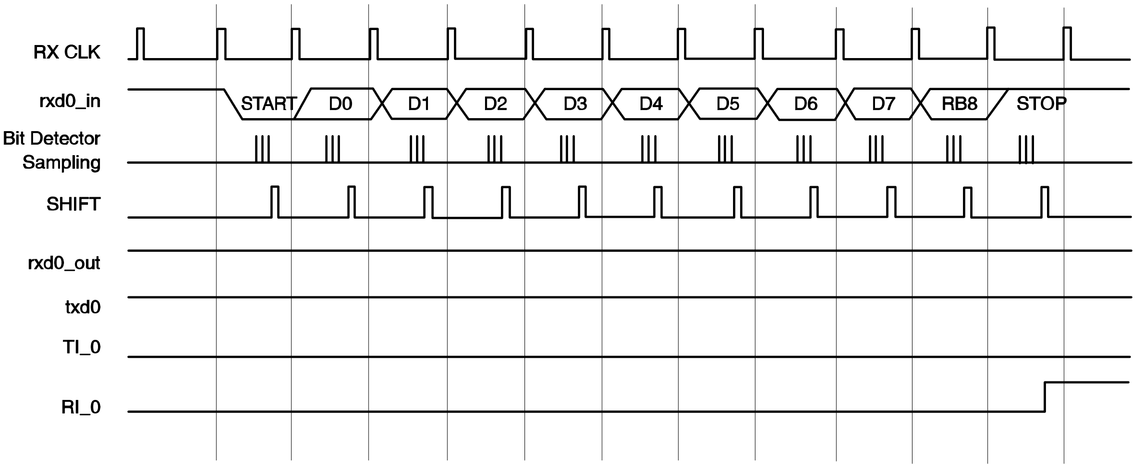


Figure 9–8. Serial Port 0 Mode 3 Receive Timing.



Reception is enabled by configuring SCON0.RBN = 1. Reception of the data begins at the falling edge of start-bit detection. The RXDx pin is sampled 16 times per bit for any baud rate setting. When the falling edge of the start bit is detected, the divide-by-16 counter used to generate the receive clock is reset to align the counter rollover with the bit boundaries. The state of each bit is determined by a majority detect decision on three consecutive samples in the middle of the bit, providing an amount of noise rejection. At the middle of the stop bit time, the serial port verifies that the status of SCONx.RI_x = 0 and SCON0.SM2_x = 1 (if SCON0.SM2_x = 0, the stop bit is a “don’t care”). If these conditions are true, then the serial port writes the received byte to the SBUFx register, loads the stop bit into SCONx.RB8_x, and sets the SCONx.RI_x flag. If the conditions are not met, the data are ignored. After the middle of the stop bit, the serial waits for another start-bit detection.

Baud rate calculation for mode 3 is identical to that of mode 1, which is fully explained in section 9.2.2.

Mode 3 has a special provision for multiprocessor communications. This mode is typically used when a master wants to address a specific slave device on the bus. The address of the target slave device is transmitted in the first eight data bits. The ninth bit is used to indicate to the slaves that the data was an address. If the data matches the slave address, the device can then resume normal reception. In this mode, nine data bits are received (the ninth bit is latched into SCON0.RB8). The port can be configured such that when the stop bit is received, the serial port interrupt will be generated if RB8 = 1. This feature is enabled by setting bit SCON0.SM2.

9.3 Setting the Serial Port Baud Rate

Once the serial port mode has been configured, as explained above, the program must configure the serial port baud rate. In mode 0, the baud rate is either the clock frequency divided by 12 or the clock frequency divided by 4, depending on the SM2 bit in the SCONx register.

Table 9–4 shows some commonly used baud rates for Mode 0.

Table 9–4. Mode 0 Commonly Used Baud Rates.

SM2	**f_{OSC} (MHz)**	**Baud Rate (kBaud)**
0	33	2750
1	33	8250
0	12	1000
1	12	3000

The mode 1 baud rate is a function of timer overflow. Serial Port 0 can use either Timer 1 or Timer 2 to generate baud rates. Serial Port 1 can only use Timer 1. The two serial ports can run at the same baud rate if they both use Timer 1, or different baud rates if Serial Port 0 uses Timer 2 and Serial Port 1 uses Timer 1.

Each time the timer increments from its maximum count (FF_H for Timer 1 or $FFFF_H$ for Timer 2), a clock is sent to the baud-rate circuit. The clock is then divided by 16 to generate the baud rate. When using Timer 1, the SMOD0 (or SMOD1) bit selects whether or not to divide the Timer 1 rollover rate by two.

In modes 1 and 3, the baud rate is determined by how frequently Timer 1 or Timer 2 overflows. The more frequently Timer 1 overflows, the higher the baud rate. There are many ways you can cause Timer 1 to overflow at a rate that determines a baud rate, but the most common method is to put Timer 1 in 8-bit auto-reload mode (Timer mode 2) and set a reload value (TH1) that causes Timer 1 to overflow at a frequency appropriate to generate a baud rate.

To determine the value that must be placed in TH1 to generate a given baud rate, the following equation can be used (assuming PCON.7 is clear):

$TH1 = 256 - ((Crystal / 384) / Baud)$

If PCON.7 is set, the baud rate is effectively doubled, thus, the equation becomes:

$TH1 = 256 - ((Crystal / 192) / Baud)$

For example, with an 11.059MHz crystal, to configure the serial port to 19 200 baud, try plugging it in the first equation:

$TH1 = 256 - ((Crystal / 384) / Baud)$

$TH1 = 256 - ((11\,059\,000 / 384) / 19\,200)$

$TH1 = 256 - ((28\,799) / 19\,200)$

$TH1 = 256 - 1.5 = 254.5$

As shown, to obtain 19 200 baud with an 11.059MHz crystal, TH1 would have to be set to 254.5. If it is set to 254, 14 400 baud is achieved and if it is set to 255, 28 800 baud is achieved. This may seem to be an impasse.

However, there is a solution. To achieve 19 200 baud, simply set PCON.7 (SMOD). When this is done, the baud rate is doubled and the second equation mentioned above is used:

$TH1 = 256 - ((Crystal / 192) / Baud)$

$TH1 = 256 - ((11\,059\,000 / 192) / 19\,200)$

$TH1 = 256 - ((57\,699) / 19\,200)$

$TH1 = 256 - 3 = 253$

Here, a nice, even TH1 value is calculated. Therefore, to obtain 19 200 baud with an 11.059MHz crystal:

1) Configure Serial Port mode 1 or 3 (for 8-bit or 9-bit serial mode).

2) Configure Timer 1 to timer mode 2 (8-bit auto-reload).

3) Set TH1 to 253 to reflect the correct frequency for 19 200 baud.

4) Set PCON.7 (SMOD) to double the baud rate.

Table 9–5 shows common settings when using Timer 1 to generate the baud rate clock.

Likewise, common settings when using Timer 2 to generate a baud rate clock are indicated in Table 9–6.

Table 9–5. Baud Rate Settings for Timer 1.

Desired Baud Rate (kb/s)	SMOD x	$C/\overline{T}$	Timer 1 Mode	TH1 Value for 33MHz clk	TH1 Value for 25MHz clk	TH1 Value for 11.0592MHz clk
57.6	1	0	2	FD_H	FE_H	FF_H
19.2	1	0	2	$F7_H$	$F9_H$	FD_H
9.6	1	0	2	EE_H	$F2_H$	FA_H
4.8	1	0	2	DC_H	$E5_H$	$F4_H$
2.4	1	0	2	$B8_H$	CA_H	$E8_H$
1.2	1	0	2	71_H	93_H	$D0_H$

Table 9–6. Baud Rate Settings for Timer 2.

Baud Rate (kb/s)	C/$\overline{T}$2	33MHz clk		25MHz clk		11.0592MHz clk	
		RCAP2H	**RCAP2L**	**RCAP2H**	**RCAP2L**	**RCAP2H**	**RCAP2L**
19.2	0	FF$_H$	EE$_H$	FF$_H$	F2$_H$	FF$_H$	FA$_H$
9.6	0	FF$_H$	CA$_H$	FF$_H$	D7$_H$	FF$_H$	EE$_H$
4.8	0	FF$_H$	95$_H$	FF$_H$	AF$_H$	FF$_H$	DC$_H$
2.4	0	FF$_H$	29$_H$	FF$_H$	5D$_H$	FF$_H$	B8$_H$
1.2	0	FE$_H$	52$_H$	FE$_H$	BB$_H$	FF$_H$	70$_H$
1.2	0	FC$_H$	A5$_H$	FD$_H$	75$_H$	FE$_H$	E0$_H$

9.4 Writing to the Serial Port

Once the serial port has been properly configured as explained previously, the serial port is ready to be used to send and receive data. If you think configuring the serial port was easy, using the serial port will be even easier.

To write a byte to the serial port, simply write the value to the SBUF0 (99$_H$) SFR. For example, sending the letter A to the serial port is accomplished as easily as:

```
MOV SBUF0,#’A’
```

Upon execution of the above instruction, the MSC1210 will begin transmitting the character via the serial port. Obviously, transmission is not instantaneous—it takes a measurable amount of time to transmit the eight data bits that make up the byte, along with its start and stop bits—and because the MSC1210 does not have a serial output buffer, you need to be sure that a character is completely transmitted before trying to transmit the next character.

The MSC1210 lets you know when it is done transmitting a character by setting the TI bit in SCON. When this bit is set, the last character has been transmitted and the next character, if any, may be sent. Consider the following code segment:

```
CLR TI          ;Be sure the bit is initially clear

MOV SBUF,#’A’   ;Send the letter ‘A’ to the serial port

JNB TI,$        ;Pause until the RI bit is set.
```

The above three instructions transmit a character and wait for the TI bit to be set before continuing. The last instruction says jump if the TI bit is not set to $. The $ character (in most assemblers), means the same address of the current instruction. Therefore, the MSC1210 will pause on the JNB instruction until the TI bit is set (upon successful transmission of the character).

9.5 Reading the Serial Port

Reading data received by the serial port is equally easy. To read a byte from the serial port, just read the value stored in the SBUF0 (99_H) SFR after the MSC1210 has automatically set the RI flag in SCON.

For example, if you want the program to wait for a character to be received and subsequently read it into the accumulator, the following code segment can be used:

```
JNB RI,$     ;Wait for the MSC1210 to set the RI flag

MOV A,SBUF   ;Read the character from the serial port
```

The first line of the above code segment waits for the MSC1210 to set the RI flag; again, the MSC1210 sets the RI flag automatically when it receives a character via the serial port. So as long as the bit is not set, the program repeats the JNB instruction continuously.

Once a character is received, the RI bit will be set automatically, the previous condition automatically fails, and program flow falls through to the MOV instruction that reads the character into the accumulator.

Chapter 10

Interrupts

Chapter 10 describes the interrupts of the MSC1210 ADC.

10.1 Description

As the name implies, an interrupt is some event that interrupts normal program execution. As stated previously, program flow is always sequential, being altered only by those instructions that expressly cause program flow to deviate in some way. However, interrupts give us a mechanism to put on hold the normal program flow, execute a subroutine, and then resume normal program flow as if we had never left it. This subroutine, called an interrupt handler or interrupt service routine (ISR), is only executed when a certain event (interrupt) occurs. The event may be one of 21 interrupt sources such as the timers overflowing, receiving a character via the serial port, transmitting a character via the serial port, or external events. The MSC1210 may be configured so that when any of these events occur, the main program is temporarily suspended and control passed to a special section of code, which presumably would execute some function related to the event that occurred. Once complete, control would be returned to the original program. The main program never even knows it was interrupted.

The ability to interrupt normal program execution when certain events occur makes it much easier and more efficient to handle certain conditions. If it were not for interrupts, the program would have to be manually checked as to whether the timers have overflowed, whether the serial port has received another character, or if some external event has occurred. Besides making the main program ugly and hard to read, such a situation makes the program inefficient because precious instruction cycles are wasted checking for events that happen infrequently.

For example, say a large 16k program is executing many subroutines and performing many tasks. Additionaly, suppose that the program is to automatically toggle the P3.0 port every time Timer 0 overflows. The code to do this is not very difficult:

```
             JNB TF0,SKIP_TOGGLE
             CPL P3.0
             CLR TF0
SKIP_TOGGLE: ...
```

The above code toggles P3.0 every time Timer 0 overflows because the TF0 flag is set whenever Timer 0 overflows. This accomplishes what is needed, but is inefficient.

Luckily, this is not necessary. Interrupts allow you to forget about checking for the condition. The microcontroller itself will check for the condition automatically, and when the condition is met, will jump to a subroutine (the interrupt handler), execute the code, and then return. In this case, the subroutine would be nothing more than:

```
CPL P3.0  ;Toggle P3.0
RETI      ;Return from the interrupt
```

First, notice that the CLR TF0 command has disappeared. That is because when the MSC1210 executes the Timer 0 interrupt routine, it automatically clears the TF0 flag. Also notice that instead of a normal RET instruction, there is a RETI instruction. The RETI instruction does the same thing as a RET instruction, but tells the 8051 that an interrupt routine has finished. Interrupt handlers must always end with RETI.

Thus, every 65 536 instruction cycles, Timer 0 overflows and the CPL and RETI instructions are executed. Those two instructions together require three instruction cycles, and accomplish the same goal as the first example. As far as the toggling of P3.0 goes, the code is 437 times more efficient! Not to mention it is much easier to read and understand because the timer 0 flag does not have to be checked in the main program. Just setup the interrupt and forget about it, secure in the knowledge that the MSC1210 will execute the code whenever it is necessary.

10.2 Events That Can Trigger Interrupts

The MSC1210 can be configured so that any of the events in Table 10–1 will cause an interrupt.

Table 10–1. Interrupt Sources

Interrupt/Event	Addr	Priority	Flag	Enable	Priority Control
DV_{DD} Low-Voltage HW Breakpoint	33_H	HIGH 0	EDLVB (AIE.0)(1) EBP (BPCON.0)(1)	EDLVV (AIE.0)(1) EBP (BPCON.0)(1)	N/A
Av_{DD} Low Voltage	33_H	0	EALV (AIE.1)(1)	EALV (AIE.1)(1)	N/A
SPI Receive	33_H	0	ESPIR (AIE.2)(1)	ESPIR (AIE.2)(1)	N/A
SPI Transmit	33_H	0	ESPIT (AIE.3)(1)	ESPIT (AIE.3)(1)	N/A
Milliseconds Timer	33_H	0	EMSEC (AIE.4)(1)	EMSEC (AIE.4)(1)	N/A
ADC	33_H	0	EADC (AIE.5)(1)	EADC (AIE.5)(1)	N/A
Summation Register	33_H	0	ESUM (AIE.6)(1)	ESUM (AIE.6)(1)	N/A
Seconds timer	33_H	0	ESEC (AIE.7)(1)	ESEC (AIE.7)(1)	N/A
External Interrupt 0	03_H	1	IE0 (TCON.1)(2)	EX0 (IE.0)(4)	PX0 (IP.0)
Timer 0 Overflow	$0B_H$	2	TF0 (TCON.5)(3)	ET0 (IE.1)(4)	PT0 (IP.1)
External Interrupt 1	13_H	3	IE1 (TCON.3)(2)	EX1 (IE.2)(4)	PX1 (IP.2)
Timer 1 Overflow	$1B_H$	4	TF1 (TCON.7)(3)	ET1 (IE.3)(4)	PT1 (IP.3)
Serial Port 0	23_H	5	RI_0 (SCON0.0) TI_0 (SCON0.1)	ES0 (IE.4)(4)	PS0 (IP.4)
Timer 2 Overflow	$2B_H$	6	TF2 (T2CON.7)	ET2 (IE.5)(4)	PT2 (IP.5)
Serial Port 1	$3B_H$	7	RI_1 (SCON1.0) TI_1 (SCON1.1)	ES1 (IE.6)(4)	PS1 (IP.6)
External Interrupt 2	43_H	8	IE2 (EXIF.4)	EX2 (EIE.0)(4)	PX2 (EIP.0)
External Interrupt 3	$4B_H$	9	IE3 (EXIF.5)	EX3 (EIE.1)(4)	PX3 (EIP.1)
External Interrupt 4	53_H	10	IE4 (EXIF.6)	EX4 (EIE.2)(4)	PX4 (EIP.2)
External Interrupt 5	$5B_H$	11	IE5 (EXIF.7)	EX5 (EIE.3)(4)	PX5 (EIP.3)
Watchdog	63_H	12 LOW	WDTI (EICON.3)	EWDI (EIE.4)(4)	PWDI (EIP.4)

Notes: 1) These interrupts set the AI flag (EICON.4) and are enabled by EAI (EICON.5).

2) If edge triggered, cleared automatically by hardware when the service routine is vectored to. If level triggered, the flag follows the state of the pin.

3) Cleared automatically by hardware when interrupt vector occurs.

4) Globally enabled by $\overline{\text{EA}}$ (IE.7).

In other words, the MSC1210 can be configured so that any of the events in Table 10–1, ranging from a simple Timer 0 overflow to a watchdog or ADC conversion event, will trigger an interrupt calling the appropriate interrupt handler routines.

Interrupt/Event—The first column of Table 10–1 indicates the name of the event, or interrupt, in question.

Addr—The second column indicates the address that to which the MSC1210 will jump, to service the interrupt when it occurs, assuming it has been enabled. This is where the interrupt code must be placed in code memory. It is common practice to place an LJMP at the address specified for the interrupt, which jumps to the actual code somewhere else in code memory, because there are only eight bytes of memory for each routine.

Priority—The third column indicates the natural priority of the interrupt. This is the order in which interrupts will be checked. If two or more interrupts occur simultaneously, the interrupt with a higher interrupt priority (i.e., that appears first in the list) will be serviced first.

Flag—The fourth column indicates the flag that, when set, will trigger the specified interrupt. These flags are normally set by the MSC1210 automatically to indicate an interrupt condition. You can, however, set these bits manually to trigger the corresponding interrupt, except in the case of the auxiliary interrupts, which are serviced at 33_H.

Enable—The fifth column indicates the bit that must be set in order to enable the given interrupt. If this bit is not set, the interrupt flag will not provoke an interrupt.

Priority Control—The final column indicates the bit that controls that interrupt priority as either high or low priority.

Note:

The interrupts that are serviced at 0033_H are always of the highest priority and that priority may not be modified.

10.3 Enabling Interrupts

By default, at power-up all interrupts are disabled. This means that even if, for example, the TF0 bit is set, the MSC1210 will not execute the Timer 0 interrupt. You must specify in code which interrupts you want the MSC1210 to enable. You may enable and disable interrupts by modifying the IE ($A8_H$), EICON ($D8_H$), and EIE ($E8_H$) SFRs, as shown in Table 10–2, Table 10–3, and Table 10–4.

Table 10–2. IE ($A8_H$) SFR

Bit	Name	Bit Address	Explanation of Function
7	EA	AF_H	Global interrupt enable/disable
6	ES1	AE_H	Enable Serial Port 1 interrupt
5	ET2	AD_H	Enable Timer 2 interrupt
4	ES	AC_H	Enable Serial Port 0 interrupt
3	ET1	AB_H	Enable Timer 1 interrupt
2	EX1	AA_H	Enable external interrupt 1
1	ET0	$A9_H$	Enable Timer 0 interrupt
0	EX0	$A8_H$	Enable external interrupt 0

Table 10–3. EICON ($D8_H$) SFR

Bit	Name	Bit Address	Explanation of Function
7	SMOD1	DF_H	Serial Port 1 double baud rate
6	–	DE_H	Undefined (set to 1)
5	EAI	DD_H	Enable auxiliary interrupt
4	AI	DC_H	Auxiliary interrupt flag
3	WDTI	DB_H	Watchdog interrupt flag
2	–	DA_H	Undefined (cleared to 0)
1	–	$D9_H$	Undefined (cleared to 0)
0	–	$D8_H$	Undefined (cleared to 0)

Table 10–4. EIE ($E8_H$) SFR

Bit	Name	Bit Address	Explanation of Function
7	–	EF_H	Undefined (set to 1)
6	–	EE_H	Undefined (set to 1)
5	–	ED_H	Undefined (set to 1)
4	EWDI	EC_H	Enable Watchdog interrupt
3	EX5	EB_H	Enable external interrupt 5
2	EX4	EA_H	Enable external interrupt 4
1	EX3	$E9_H$	Enable external interrupt 3
0	EX2	$E8_H$	Enable external interrupt 2

Each of the MSC1210 interrupts has its own enable bit in one of these three SFRs. Enable a given interrupt by setting the corresponding bit. For example, to enable the Timer 1 Interrupt, execute either:

```
MOV IE,#08h
```

or

```
SETB ET1
```

Both of the previous instructions set bit 3 of IE, thus enabling the Timer 1 Interrupt. Once the Timer 1 Interrupt is enabled, whenever the TF1 bit is set, the MSC1210 will automatically put on hold the main program and execute the Timer 1 interrupt handler at address $001B_H$.

However, before the Timer 1 interrupt (or any other interrupt) is truly enabled, bit 7 of IE must also be set. Bit 7, the global interrupt enable/disable, enables or disables all interrupts simultaneously (except the auxiliary interrupts). That is to say, if bit 7 is cleared, no interrupts will occur, even if all the other bits of IE are set. Setting bit 7 will enable all the interrupts that have been selected by setting one of the other enable bits in one of the three SFRs. This is useful in program execution if there is time-critical code that needs to be executed. In this case, the code may need to be executed from start to finish without any interrupts getting in the way. To accomplish this, simply clear bit 7 of IE and (CLR EA) bit 5 of EICON (CLR EAI), and then set them after the time-critical code is done.

To sum up what has been stated in this section, to enable the Timer 1 Interrupt, the most common approach is to execute the following two instructions:

```
SETB ET1  ;Enable Timer 1 Interrupt

SETB EA   ;Enable Global Interrupt flag
```

Thereafter, the Timer 1 interrupt handler at $01B_H$ will automatically be called whenever the TF1 bit is set (upon Timer 1 overflow).

10.4 Polling Sequence

The MSC1210 automatically evaluates whether an interrupt should occur after every instruction. When checking for interrupt conditions, under default conditions, it checks them in the order as they appear in Table 10–1.

This means that if a serial interrupt occurs at the exact same instant that an external 0 interrupt occurs, the external 0 interrupt will be executed first, and the serial interrupt will be executed once the external 0 interrupt has completed.

10.5 Interrupt Priorities

The MSC1210 offers three levels of interrupt priority: highest, high, and low. By using interrupt priorities, higher priority may be assigned to certain interrupt conditions. The highest priority is reserved for the auxiliary interrupt that vectors through address 0033_H—the auxiliary interrupt is always of highest priority and no other interrupt may be assigned that priority.

All other interrupts may be assigned either high or low priority. For example, assume the Timer 1 interrupt has been enabled to be automatically called every instance Timer 1 overflows. Additionally, the serial interrupt has been enabled to be called every time a character is received via the serial port. However, in this case, receiving a character is much more important than the timer interrupt. Therefore, if the Timer 1 interrupt is already executing, the serial interrupt must interrupt the Timer 1 interrupt. When the serial interrupt is complete, control passes back to the Timer 1 interrupt and finally back to the main program. This may be accomplished by assigning a high priority to the serial interrupt and a low priority to the Timer 1 interrupt.

Interrupt priorities are controlled by the IP ($B8_H$) or EIP ($F8_H$) SFRs. These SFRs have the following formats, as shown in Table 10–5 and Table 10–6.

Table 10–5. IP ($B8_H$) SFR

Bit	Name	Bit Address	Explanation of Function
7	–	BF_H	Undefined
6	–	BE_H	Undefined
5	–	BD_H	Undefined
4	PS	BC_H	Serial Interrupt Priority
3	PT1	BB_H	Timer 1 Interrupt Priority
2	PX1	BA_H	External 1 Interrupt Priority
1	PT0	$B9_H$	Timer 0 Interrupt Priority
0	PX0	$B8_H$	External 0 Interrupt Priority

Table 10–6. EIP ($F8_H$) SFR

Bit	Name	Bit Address	Explanation of Function
7	–	FF_H	Undefined (set to 1)
6	–	FE_H	Undefined (set to 1)
5	–	FD_H	Undefined (set to 1)
4	PWDI	FC_H	Watchdog Interrupt Priority
3	PX5	FB_H	External Interrupt 5 Priority
2	PX4	FA_H	External Interrupt 4 Priority
1	PX3	$F9_H$	External Interrupt 3 Priority
0	PX2	$F8_H$	External Interrupt 2 Priority

When considering interrupt priorities, the following rules apply:

1) Nothing can interrupt the highest-priority auxiliary interrupt, not even another auxiliary interrupt.

2) Only an auxiliary interrupt (highest priority) can interrupt a high-priority interrupt.

3) A high-priority interrupt may interrupt a low-priority interrupt.

4) A low-priority interrupt may only occur if no other interrupt is currently executing.

5) If two interrupts occur at the same time, the interrupt with higher priority will execute first. If both interrupts are of the same priority, the interrupt that is serviced first by the polling sequence will be executed first.

10.6 Interrupt Triggering

When an interrupt is triggered, the following actions are taken automatically by the microcontroller:

1) The current program counter is saved on the stack, low byte first and high byte second.

2) Interrupts of the same and lower priority are blocked.

3) In the case of timer and external interrupts, the corresponding interrupt flag is cleared.

4) Program execution transfers to the corresponding interrupt handler vector address.

5) The interrupt handler routine, written by the developer, is executed.

Take special note of the third step. If the interrupt being handled is a timer or external interrupt, the microcontroller automatically clears the interrupt flag before passing control to the interrupt handler routine. This means it is not necessary that the bit be cleared in code.

10.7 Exiting Interrupts

An interrupt ends when your program executes the RETI (return from interrupt) instruction. When the RETI instruction is executed, the following actions are taken by the microcontroller:

1) Two bytes are popped off the stack into the program counter to restore normal program execution, high byte first and low byte second.

2) Interrupt status is restored to its pre-interrupt status. This means interrupts of the same and higher level may once again be executed.

10.8 Types of Interrupts

Each interrupt can be categorized as one these types: serial, external, timer, watchdog, or auxiliary.

10.8.1 Serial Interrupts

There are two interrupt flags that provoke a serial interrupt: receive interrupt (RI) and transmit interrupt (TI). If either flag is set, a serial interrupt is triggered. As discussed in section 9.2, the RI bit is set when a byte is received by the serial port and the TI bit is set when a byte has been sent.

This means that when the serial interrupt is executed, it may have been triggered because the RI flag was set, the TI flag was set, or both flags were set. Thus, your routine must check the status of these flags to determine what action is appropriate. Additionally, because the MSC1210 does not automatically clear the RI and TI flags, you must clear these bits in the interrupt handler.

A brief code example is in order:

```
INT_SERIAL:
  JNB RI,CHECK_TI  ;If RI flag is not set, we jump to check TI
  MOV A,SBUF       ;If we got here, the RI bit *was* set
  CLR RI           ;Clear the RI bit after we’ve processed it
CHECK_TI:
  JNB TI,EXIT_INT  ;If TI flag not set, we jump to exit point
  CLR TI           ;Clear TI bit before we send next character
  MOV SBUF,#’A’    ;Send another character to the serial port
EXIT_INT:
  RETI             ;Exit interrupt handler
```

As shown, the code checks the status of both interrupts flags. If both flags were set, both sections of code will be executed. Also note that each section of code clears its corresponding interrupt flag. If the interrupt bits are not cleared, the serial interrupt will be executed over and over until the bit is cleared. For this reason, it is very important that the interrupt flags in a serial interrupt always be cleared.

10.8.2 External Interrupts

The MSC1210 microcontroller has six external interrupt sources. These include the standard two interrupts of the 8052 architecture and four new sources. The standard 8052 interrupts are $\overline{\text{INT0}}$ and $\overline{\text{INT1}}$. These are active low, but can be configured to be edge- or level-triggered by modifying the value of IT0 and IT1 (TCON, 88_H). If ITx is assigned a logic 0, the interrupt is level-triggered. The interrupt condition remains in force as long as the pin is low. If ITx is assigned a logic 1, the interrupt is pseudo edge-triggered.

The pin driver of an edge-triggered interrupt must hold both the high, then the low condition for at least one machine cycle (each) to ensure detection because the external interrupts are sampled. This means maximum sampling frequency on any interrupt pin is 1/8th of the main oscillator frequency.

Note:

Level-sensitive interrupts are not latched. If the interrupt is level-sensitive, the condition must be present until the processor can respond to it. This is most important if other interrupts are being used with a higher or equal priority. If the device is currently processing another interrupt of higher priority, the condition must be present until the current interrupt is complete. This is because the level-sensitive interrupt is not sampled until the RETI instruction is executed. Upon returning, if the level-triggered interrupting signal is not there, it is as though the interrupt request was never issued.

The remaining four external interrupts are similar in nature, with one difference: INT2 and INT4 are positive edge detect only, while $\overline{\text{INT3}}$ and $\overline{\text{INT5}}$ are negative edge detect only. These interrupts do not have level-detect modes. All associated bits and flags operate the same and have the same polarity as the first two interrupts. A logic 1 on an interrupt flag indicates the presence of an interrupt condition, not the logic state of the input pin.

The flags that trigger external interrupts 2 through 5 are found in the EXIF (91_H) SFR, as shown in Table 10–7. When the appropriate condition (falling-edge or rising-edge) is detected, the corresponding flag is set and the interrupt is triggered, if enabled.

Note:

The bits in EXIF are set to 1 to indicate that the condition is true—the bits do not represent the current level of the pin. That is, IE5 will be set to 1 when a falling edge is detected on $\overline{\text{INT5}}$, even though $\overline{\text{INT5}}$ is at a logic 0 level at that point.

Table 10–7. EXIF (91_H) SFR

Bit	Name	Explanation of Function
7	IE5	External interrupt 5 flag – falling edge detected on $\overline{\text{INT5}}$
6	IE4	External interrupt 4 flag – rising edge detected on INT4
5	IE3	External interrupt 3 flag – falling edge detected on $\overline{\text{INT3}}$
4	IE2	External interrupt 2 flag – rising edge detected on INT2
3	–	Reserved (cleared to 1)
2	–	Undefined (cleared to 0)
1	–	Undefined (cleared to 0)
0	–	Undefined (cleared to 0)

There are three interrupts that can wake up the processor if it is in the low-power IDLE mode: the external interrupts ($\overline{\text{INT0}}$ and $\overline{\text{INT1}}$), and the Watchdog (when used as an interrupt). In order to be used to wake up the processor, they must be enabled in the Wake Up Enable register, WUEN ($C6_H$).

10.8.3 Timer Interrupts

The MSC1210 microcontroller incorporates three 16-bit programmable timers, each of which can generate an interrupt. In addition, there are three other sources for timer interrupts: the milliseconds timer, seconds timer, and watchdog timer. Each timer has an independent interrupt enable, flag, vector, and priority.

Timers 0, 1, and 2 set their respective flags when their individual timer overflows. These flags will be set regardless of the interrupt enable status. If the interrupt is enabled, this event will also cause the processor to vector into the corresponding ISR routine, provided it has the highest priority. For Timers 0 and 1, the flags are cleared when the processor jumps to the interrupt vector. Thus, these flags are not available for use by the ISR, but are available outside of the ISR and in applications that do not acknowledge the interrupt (i.e., jump to the vector). If the interrupt is not acknowledged, then software must manually clear the flag bit. In Timer 2, jumping to the interrupt vector does not clear the flag, therefore, software must always clear it manually. Timer 0 and 1 flag bits reside in the TCON register. The Timer 2 flag bit resides in the T2CON register. The interrupt enables and priorities for Timers 0, 1, and 2 reside in the IE and IP registers, respectively.

10.8.4 Watchdog Interrupt

The watchdog interrupt usually has a different connotation than the timer interrupts. Unless the watchdog is being used as a very long timer, the completion of the watchdog count means the software has failed to reset the counter and may be lost. Like other sources, the watchdog timer has a flag bit, an enable, and a priority. It also has its own vector. These are summarized in Table 10–1. For the watchdog timer to perform the processor reset function, it must be enabled in the flash configuration register during serial or parallel programming.

10.8.5 Auxiliary Interrupts

The auxiliary interrupt allows the MSC1210 to offer additional interrupts without requiring additional ISR vectors. A number of distinct interrupts, when enabled, all provoke the auxiliary interrupt. The ISR then examines the flags to determine which auxiliary interrupt was the source of the interrupt.

The auxiliary interrupt has the highest priority, which means all of the interrupts that are handled by the auxiliary interrupt will always have precedence over non-auxiliary interrupts. Although the interrupt may be disabled if required, the priority level (highest) cannot be altered by the user.

Before returning from the ISR for an auxiliary interrupt, the interrupt source must be cleared and then EICON.4 (AI) must be cleared. The interrupt sources are cleared as shown in Table 10–8.

Table 10–8. Clearing Auxiliary Interrupts

Aux Interrupt Type	Method to Clear Interrupt
Seconds interrupt	Read SECINT SFR
Summation interrupt	Read SUMR0 SFR
ADC conversion interrupt	Read ADRESL SFR
Millisecond interrupt	Read MSINT SFR
SPI transmit interrupt	Write SPIDATA SFR
SPI receive interrupt	Read SPIDATA SFR
Analog low-voltage interrupt	Remove low-voltage condition
Digital low-voltage interrupt	Remove low-voltage condition
Breakpoint interrupt	Set BP = 1, bit 7 of BPCON SFR

To enable Auxiliary interrupts, the EICON.5 (EAI) bit must be set, which enables auxiliary interrupts. When so configured, the MSC1210 will be configured to respond to those auxiliary interrupts that are enabled in the AIE ($A6_H$) SFR.

The Auxiliary Interrupt Enable (AIE) SFR controls which of the auxiliary interrupts are enabled and which are disabled (masked). If auxiliary interrupts are enabled, as described in the previous paragraph, and the specific auxiliary interrupt is enabled in AIE, that condition will set the EICON.4 (AI) flag to indicate an auxiliary interrupt and vector through 0033_H. The ISR must clear the AI flag before returning, or the auxiliary interrupt will be triggered again.

Table 10–9. AIE ($A6_H$) SFR

Bit	Name	Explanation of Function
7	ESEC	Enable Seconds Auxiliary Interrupt
6	ESUM	Enable Summation Auxiliary Interrupt
5	EADC	Enable ADC Conversion Auxiliary Interrupt
4	EMSEC	Enable Millisecond Auxiliary Interrupt
3	ESPIT	Enable SPI Transmit Auxiliary Interrupt
2	ESPIR	Enable SPI Receive Auxiliary Interrupt
1	EALV	Enable Analog Low-Voltage Auxiliary Interrupt
0	EDLVB	Enable Digital Low-Voltage or Breakpoint Auxiliary Interrupt

Note:

Reading from the AIE SFR will return the current state of the corresponding condition, regardless of whether or not an interrupt is enabled. For example, if an ADC conversion has been completed and an interrupt would be triggered if it were enabled, reading the EADC bit will return a 1, regardless of whether or not the interrupt was actually enabled.

The AISTAT ($A7_H$) is a read-only SFR that returns the current state of interrupt conditions that are enabled. Any condition that is configured to provoke an interrupt and is currently true will return a 1. Any condition that is not currently true or was not configured to provoke an interrupt will return a 0.

Table 10–10. AISTAT ($A7_H$) SFR

Bit	Name	Explanation of Function	Clear Interrupt
7	SEC	Detect seconds auxiliary interrupt	Read SECINT
6	SUM	Detect summation auxiliary interrupt	Read SUMR0
5	ADC	Detect ADC conversion auxiliary interrupt	Read ADRESL
4	MSEC	Detect millisecond auxiliary interrupt	Read MSINT
3	SPIT	Detect SPI transmit auxiliary interrupt	Write SPIDATA
2	SPIR	Detect SPI receive auxiliary interrupt	Read SPIDATA
1	ALVD	Detect analog low-voltage auxiliary interrupt	Voltage above threshold
0	DLVD	Detect digital low-voltage or breakpoint auxiliary interrupt	Write BP = 1

Note:

AISTAT is read-only. A value may not be written to this SFR with the expectation of triggering the specific auxiliary interrupt. An auxiliary interrupt may be triggered by setting the EICON.4 (AI) flag, but which auxiliary interrupt will be triggered in software cannot be specified.

When an Auxiliary interrupt occurs, the MSC1210 will vector to the ISR at 0033_H. The code of the ISR may use the Pending Auxiliary Interrupt (PAI, $A5_H$) SFR to determine which of the auxiliary interrupts provoked the actual interrupt.

Table 10–11. PAI ($A5_H$) SFR

Bit	Name	Explanation of Function
7	–	Undefined
6	–	Undefined
5	–	Undefined
4	–	Undefined
3	PAI3	Bit 3 of Auxiliary Interrupt Index
2	PAI2	Bit 2 of Auxiliary Interrupt Index
1	PAI1	Bit 1 of Auxiliary Interrupt Index
0	PAI0	Bit 0 of Auxiliary Interrupt Index

The four bits, PAI0 through PAI3, make up a 4-bit value that indicates the auxiliary interrupt that triggered the actual interrupt. Because the value returned by PAI is between 0 and 8, it can be used as an index or offset to determine what ISR to execute. There is no priority to the aquxiliary interrupts, but there is a priority to how they are displayed in the PAI register.

Table 10–12. PPI Bits of PAI SFR

PAIx BITS				**Explanation of Interrupt/Event**
3	2	1	0	
0	0	0	0	No pending peripheral IRQ
0	0	0	1	Digital low-voltage/breakpoint IRQ or lower priority IRQ pending
0	0	1	0	Analog low-voltage IRQ or lower priority IRQ pending
0	0	1	1	SPI receive IRQ or lower priority IRQ pending
0	1	0	0	SPI transmit IRQ or lower priority IRQ pending
0	1	0	1	One millisecond system timer IRQ or lower priority IRQ pending
0	1	1	0	ADC conversion IRQ or lower priority IRQ pending
0	1	1	1	Accumulator IRQ or lower priority IRQ pending
1	0	0	0	One second system timer IRQ pending

10.8.5.1 Low-Voltage Detect Interrupts

There are two low-voltage detect interrupts: one for AV_{DD} and one for DV_{DD}. In addition to these, a voltage level can be selected during programming that will cause a reset. The voltage level used for the interrupts is selected by the Low Voltage Detect control register LVDCON ($E7_H$). If V_{DD} drops below the level selected, an interrupt will result (if enabled).

The breakpoint and these two interrupts have priority for encoding in the PAI SFR for the AI interrupt. The detection level can be adjusted from 2.7V to 4.7V or an external analog signal.

Note:

The EAI bit enables the AI Interrupt. This bit is not subject to the global interrupt enable (EA). The low-voltage detect interrupts are a level-sensitive interrupt and remains set as long as V_{DD} remains below the select voltage.

10.8.5.2 SPI Receive/Transmit Interrupts

The SPI receive or transmit interrupt will be triggered when the number of bytes indicated by SPIRCON have been received, or the number of bytes indicated by SPITCON have been transmitted.

10.8.5.3 Milliseconds/Seconds Interrupts

The MSC1210 includes two additional timer interrupts that may trigger an interrupt at regular intervals.

The milliseconds interrupt is triggered every n milliseconds, where n is the number of stored in the MSINT (FA_H) SFR. For example, if MSINT is set to 20, a millisecond interrupt will be provoked every 20ms. This assumes and requires that MSECH (FD_H) and MSECL (FC_H) are set to values that represent a millisecond. If MSECH and MSECL are set to other values, the frequency at which the millisecond interrupt occurs will vary proportionally.

The seconds interrupt functions in a manner similar to the millisecond interrupt, but can be used to provoke an interrupt at reduced frequencies, on the order of seconds. For the seconds interrupt to be provoked once per second, MSECH and MSECL must be set to values that represent a millisecond, and HMSEC (FE_H) must be set to a value that represents 1/100th of a second. If any of these three SFRs are assigned different values, the frequency of the seconds interrupt will vary proportionally.

10.8.5.4 ADC Conversion Interrupt

The ADC conversion interrupt is triggered whenever an ADC conversion produces a new result in the ADRESH/M/L SFRs. When an ADC conversion interrupt is triggered or signaled, the user program can read the new result from these SFRs. The interrupt is cleared by reading the LSB of the sample data (ADRESL).

10.8.5.5 Summation Register Interrupt

When the summation mode is set to modes 1 (sum values from the ADC) or 3 (sum for SCNT times, then shift SHFT times), an interrupt will occur at the end of the process. Note that an interrupt will not occur in modes 0 and 2. The interrupt is cleared by reading the LSB of the summation registers (SUMR0).

10.9 Waking Up from Idle Mode

When the MSC1210 is placed in idle mode, three events and the auxiliary interrupts can optionally wake up the microcontroller. The three events are: a watchdog interrupt, external interrupt 1, or external interrupt 0. Which interrupt(s) wakes up the MSC1210 is determined by the Enable Wake Up (EWU, $E8_H$) SFR.

Table 10–13. EWU ($C6_H$) SFR

Bit	Name	Explanation of Function
7	–	Undefined
6	–	Undefined
5	–	Undefined
4	–	Undefined
3	–	Undefined
2	EWUWDT	Wake Up on Watchdog Timer
1	EWUEX1	Wake Up on External Interrupt 1
0	EWUEX0	Wake Up on External Interrupt 0

Setting each of the bits in this SFR will allow the MSC1210 to wake up from idle mode when the corresponding interrupt occurs. If the corresponding bit is clear, the specified interrupt will not cause the MSC1210 to wake up from idle mode.

10.10 Register Protection

One very important rule applies to all interrupt handlers: interrupts must leave the processor in the same state as it was in when the interrupt initiated. Remember, the idea behind interrupts is that the main program is not aware that they are executing in the background. However, consider the following code:

```
CLR C        ;Clear carry

MOV A,#25h   ;Load the accumulator with 25h

ADDC A,#10h  ;Add 10h, with carry
```

After the above three instructions are executed, the accumulator will contain a value of 35_H.

However, what would happen if an interrupt occurred right after the MOV instruction? During this interrupt, the carry bit was set and the value of the accumulator changed to 40_H. When the interrupt finished and control was passed back to the main program, the ADDC would add 10_H to 40_H, and also add an additional 01_H because the carry bit is set. The accumulator will contain the value 51_H at the end of execution.

In this case, the main program has seemingly calculated the wrong answer. How can 25_H + 10_H yield 51_H as a result? It does not make sense. A developer that was unfamiliar with interrupts would be convinced that the microcontroller was damaged in some way, provoking problems with mathematical calculations.

What has happened, in reality, is the interrupt did not protect the registers it used. Restated: an interrupt must leave the processor in the same state as it was in when the interrupt initiated.

This means if an interrupt uses the accumulator, it must insure that the value of the accumulator is the same at the end of the interrupt as it was at the beginning. This is generally accomplished with a PUSH and POP sequence at the beginning and end of each interrupt handler. For example:

```
INTERRUPT_HANDLER:

   PUSH ACC      ;Push the initial value of accumulator
                 ;onto stack

   PUSH PSW      ;Push the initial value of PSW SFR onto stack

   MOV A,#0FFh   ;Use accumulator & PSW for whatever you want

   ADD A,#02h    ;Use accumulator & PSW for whatever you want

   POP PSW       ;Restore the initial value of the PSW from
                 ;the stack

   POP ACC       ;Restore initial value of the accumulator
                 ;from stack
```

The guts of the interrupt are the MOV instruction and the ADD instruction. However, these two instructions modify the accumulator (the MOV instruction) and also modify the value of the carry bit (the ADD instruction will cause the carry bit to be set). The routine pushes the original values onto the stack using the PUSH instruction because an interrupt routine must ensure that the registers remain unchanged by the routine. It is then free to use the registers it protected as needed. Once the interrupt has finished its task, it POPs the original values back into the registers. When the interrupt exits, the main program will never know the difference because the registers are exactly the same as they were before the interrupt executed.

In general, the ISR must protect the following registers:

1) Program Status Word SFR (PSW)
2) Data Pointer SFRs (DPH/DPL)
3) Accumulator (ACC)
4) B Register (B)
5) R Registers (R0–R7)

Remember that the PSW consists of many individual bits that are set by various instructions. Unless you are absolutely sure and have a complete understanding of what instructions set what bits, it is generally a good idea to always protect the PSW by PUSHing and POPing it off the stack at the beginning and end of the interrupts.

Also note that most assemblers will not allow the execution of the instruction:

```
PUSH R0   ;Error – Invalid instruction!
```

This is due to the fact that, depending on which register bank is selected, R0 may refer to either internal RAM address 00_H, 08_H, 10_H, or 18_H. R0, in and of itself, is not a valid memory address that the PUSH and POP instructions can use.

Thus, if using any R register in the interrupt routine, push the absolute address of that register onto the stack instead of just saying PUSH R0. For example, instead of PUSH R0, execute:

```
PUSH Reg0  ;Requires use of definition file MSC1210.INC
```

If the MSC1210.INC definition file has not been included in the project, the register must be protected with:

```
PUSH 00h   ;Pushes R0 onto stack, if using register bank 0
```

Of course, this only works if the default register bank (bank 0) has been selected. If using an alternate register set, PUSH the address that corresponds to the register in the bank being used.

10.11 Common Problems with Interrupts

Interrupts are a very powerful tool available to you, but when used incorrectly, can be a source of a huge number of debugging hours. Errors in interrupt routines are often very difficult to diagnose and correct.

If you use interrupts and your program is crashing or does not seem to be performing as expected, always review the following interrupt-related issues:

Register protection: Make sure all registers are protected, as explained previously. Forgetting to protect a register that the main program is using can produce very strange results. In the example above, failure to protect registers caused the main program to apparently calculate that $25_H + 10_H = 51_H$. If registers start changing values unexpectedly or operations produce incorrect values, it is very likely that the registers have not been protected. *Always protect the registers!*

Forgetting to restore protected values: Another common error is to push registers onto the stack to protect them, and then forget to pop them off the stack before exiting the interrupt. For example, if you push ACC, B, and PSW onto the stack in order to protect them, and subsequently pop ACC and PSW off the stack before exiting, but forget to restore the value of B. you leave an extra value on the stack. When executing the RETI instruction, the 8051 will use that value as the return address instead of the correct value. In this case, the program will almost certainly crash. *Always make sure to pop the same number of values off the stack as were pushed onto it.*

Using RET instead of RETI: Remember that interrupts are always terminated with the RETI instruction. It is easy to inadvertently use the RET instruction instead. However, the RET instruction will not end the interrupt. Usually, using a RET instead of a RETI will cause the illusion of the main program running normally, but the interrupt will only be executed once. If it appears that the interrupt mysteriously stops executing, verify that the routine is exiting with RETI.

Make interrupt routines small: Interrupt routines should be designed to do as little as possible, as quickly as possible, and leave longer processing to the main program. For example, a receive serial interrupt should read a byte from SBUF and copy it to a temporary buffer defined by the user and exit as quickly as possible. The main program must then handle the process of interpreting the data that was stored in the temporary buffer. By minimizing the amount of time spent in an interrupt, the MSC1210 spends more time in the main program, which means additional interrupts can be handled faster when they occur.

Chapter 11

Pulse Width Modulator/Tone Generator

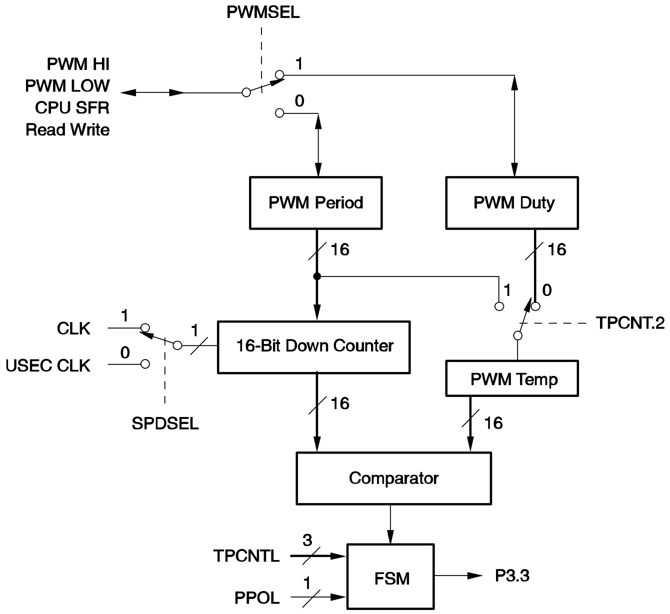
Chapter 11 describes the pulse width modulator/tone generator of the MSC1210 ADC.

Topic **Page**

11.1 Description

The pulse width modulator (PWM) has two modes: one mode functions as a tone generator and the the other mode functions as a pulse width modulator.

Figure 11–1. Block Diagram



The PWM/tone generator is controlled and configured by a number of SFRs, the primary being the PWM Configuration (PWMCON, $A1_H$) SFR.

The individual bits of PWMCON have the following functions:

	7	6	5	4	3	2	1	0	Reset Value
SFR $A1_H$	—	—	PPOL	PWMSEL	SPDSEL	TPCNTL.2	TPCNTL.1	TPCNTL.0	00_H

PPOL (bit 5)—**Period Polarity**. Specifies the level of the PWM pulse.

0: ON period. PWM Duty register programs the ON period.

1: OFF period. PWM Duty register programs the OFF period.

PWMSEL(bit 4)—**PWM Register Select**. Select which 16-bit register is accessed by PWMLOW/PWMHIGH.

0: Period.

1: Duty.

SPDSEL(bit 3)—**Speed Select**.

0: 1MHz (ONEUSEC Clock).

1: SYSCLK.

TPCNTL(bits 2-0)—**Tone Generator/Pulse Width Modulator Control**.

TPCNTL.2	TPCNTL.1	TPCNTL.0	Mode
0	0	0	Disable (default)
0	0	1	PWM
0	1	1	Tone—square
1	1	1	Tone—staircase

The three bits that together make up TPCNTL, control the function of the PWM/ tone generator. The function of the generator is determined according to the table above.

TPCNTL.0 enables or disables the PWM/tone generator. If set to ‘1’, the block will act as either a PWM or tone generator depending on the setting of TPCNTL.1. When TPCNTL.0 is ‘0’, the function block is completely disabled. This state of the block is the default state.

When TPCNTL.1 is 0, the block acts as a pulse width modulator in which a modulated pulse is generated whose duty cycle is determined by the PWM Duty and PWM Period registers. The range of frequencies that can be generated is 4kHz to 500kHz with a 1MHz clock, or up to 16MHz with sysclock.

When TPCNTL.1 is 1, the block acts as a tone generator which may generate either a staircase or square waveform, depending on further configuration. In either case, the frequency range is 60Hz to 16MHz.

11.2 Tone Generator

When TPCNTL [1:0] = 11, the block functions as a tone generator with either a square or staircase waveform that has two or three levels of 0V, high impedance, and V_{DD} volts, respectively. The widths of each step in the staircase waveform are chosen so that the error between the staircase waveform and a sinusoidal waveform of the same frequency is minimized; in staircase mode, the output is high impedance for the last 1/4 of each half period.

$$ToneFrequency = \frac{1}{2 \cdot PWMPeriod[15:0] \cdot T_{BASE}}$$

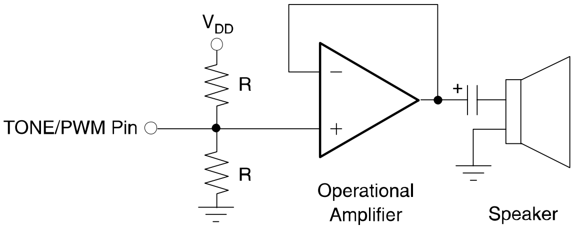
Where:

$T_{BASE} = T_{CLK}$ when $SPDSEL = 1$

$T_{BASE} = T_{USEC}$ when $SPDSEL = 0$.

The TONE/PWM output pin is fed to a circuit depending upon the application. In the Figure 11–2, the circuit of a tone generator is shown. When the output is high-impedance, the voltage value that is buffered and fed to the speaker is $V_{DD}/2$.

Figure 11–2. Tone Generator Circuit



11.2.1 Tone Generator Waveforms

When TPCNTL[1:0] = 11, the output of the tone generator may be either a staircase waveform or a square waveform depending on the configuration of TPCNTL.2.

When TPCNTL.2 is 1, a staircase waveform is generated that will have three levels: DGND, tristate, and V_{DD} volts.

When the TPCNTL.2 is 0, a square waveform of 50% duty cycle is generated that will have two levels: DGND and V_{DD} volts.

11.2.1.1 Staircase Mode

When TPCNTL.2 is 1 (i.e., TPCNTL[2:0] = 111), a staircase waveform is generated, as shown in Figure 11–3. In this figure, the value of PWM Period = $18F_H$, which is equal to 399. Therefore, the total time period is equal to 800 T_{BASE}.

11.2.1.2 Square Mode

When TPCNTL.2 is ‘0’ (i.e., TPCNTL[2:0] = ‘011’), a square waveform is generated. An example with PWM Period = 2 is shown in Figure 11–4. We get a 50 % duty cycle square wave with a period of (2 • PWM Period).

Figure 11–3. Timing Diagram of Tone Generator in Staircase Mode

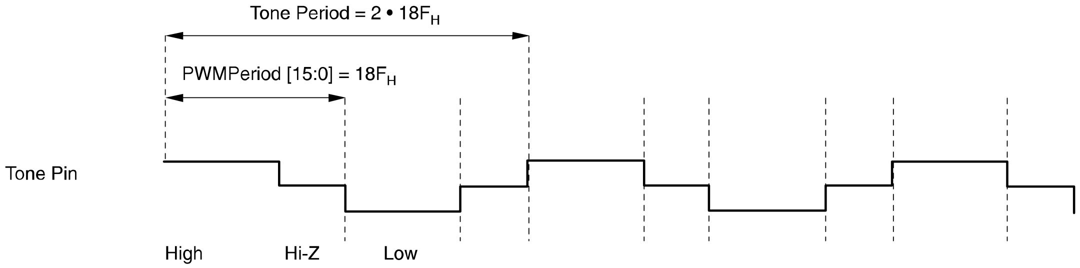
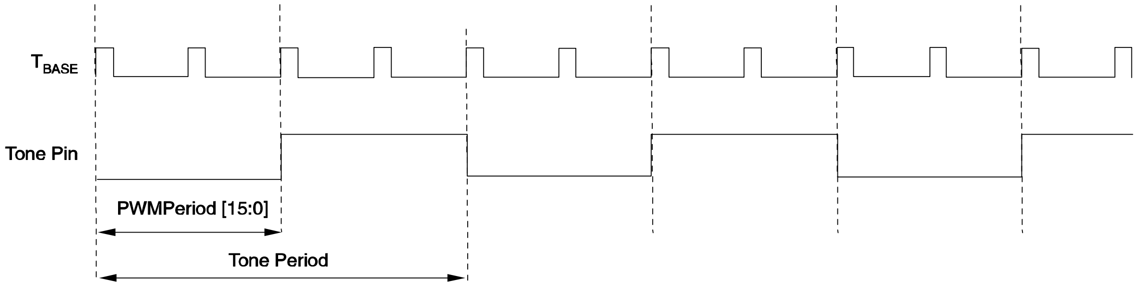


Figure 11–4. Timing Diagram of Tone Generator in Square Wave Mode



11.3 PWM Generator

The PWM generator is activated when TPCNTL[1:0] = 01. This setting allows a PWM waveform to be generated automatically by the MSC1210 with characteristics defined by the user program. The PWM is configured based on the PWMCON SFR, the PWM Period and PWM Duty settings, and the USEC SFR setting. The USEC SFR or SYS clock (defined by Speed Select) generates a tick that defines the unit period that is used by PWM Period and PWM Duty in defining the waveform.

As its name indicates, the PWM Period register gives us the period of the PWM wave, whereas the PWM Duty register defines the length of time which sets the duty cycle. We can program either the ON duty or the OFF duty depending on the bit PPOL (PWMCOM.5). If PPOL is set, then OFF duty period is programmed, and if it cleared, then ON duty period is programmed. The duty cycle is periodic with respect to the period of PWM, irrespective of the duty register. The duty cycle of the PWM wave for different configurations is shown in the following equations and in Table 11–1.

When PPOL (PWMCON.5) = 0:

$\mathit{PWM\ Frequency} = 1/T_{BASE} \bullet (\mathit{PWM\ Period}[15{:}0] + 1)$

$\mathit{PWM\ ON\ Period} = T_{BASE} \bullet \mathit{PWM\ Duty}[15{:}0]$

$\mathit{Duty\ Cycle} = \mathit{PWM\ Duty}/(\mathit{PWM\ Period}[15{:}0] + 1)$

Where:

$T_{BASE} = T_{CLK}$ when $SPDSEL = 1$,

$T_{BASE} = T_{USEC}$ when $SPDSEL = 0$.

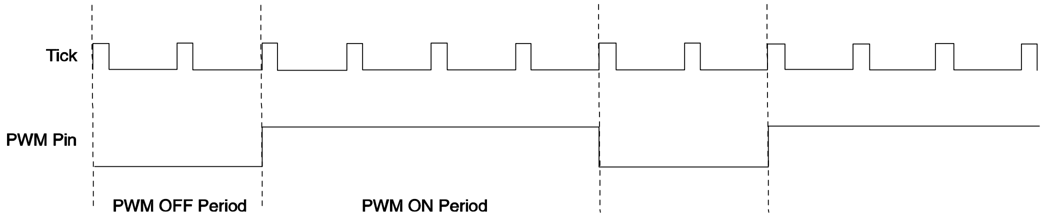
When PPOL (PWMCON.5) = 1, PWM Duty is controlling the OFF period, therefore:

$\mathit{Duty\ Cycle} = \mathit{PWM\ Period} + 1 - \mathit{PWM\ Duty}/\mathit{PWM\ Period} + 1.$

Table 11–1. PWM Polarity Conditions

PPOL	Condition	Duty Cycle
0	Period = X, Duty = 0	0% (always outputs low)
0	0 < Duty ≤ Period	Intermediate Value
0	Duty > Period	100% (always outputs high)
1	Period = X, Duty = 0	100% (logic ‘1’)
1	0 < Duty ≤ Period	Intermediate Value
1	Duty > Period	0% (logic ‘0’)

Figure 11–5. Timing Diagram of a PWM Waveform



In the timing diagram of a PWM waveform in Figure 11–5, the waveform is low for 2 ticks and high for 4 ticks. Thus, the value of PWM Period = 5 (6 ticks minus 1) and PWM Duty = 1 (2 ticks minus 1). Assuming the PPOL (PWMCON.5) bit is set, the actual length of a tick is defined by the value of USEC, or equal to the period of CLK.

Configuring the PWM generator requires that the PWM Period and PWM Duty registers be set. Both of these registers are set using the PWMLOW and PWMHI SFRs; whether the program writes to PWM Period or PWM Duty is configured by first clearing or setting PWMCON.4. When PWMSEL is clear any subsequent write to PWMLOW/HI will write to the PWM Duty register. When PWMCON.4 is set, any subsequent write to PWMLOW/HI will write to the PWM Period register. PWMLOW and PWMHI can be treated as one 16-bit register because they are adjacent SFRs.

Thus, the general process for configuring the PWM generator is as follows:

1) Configure PWMCON so that PWM mode is selected, PWMCON[2:0] = 001.

2) Set PWMCON.4 to select the PWM Duty register.

3) Write the PWM Duty – 1 value to PWMLOW and PWMHI. In the above example, PWMLOW/HI is written with the value 1 because the off period is 2 ticks long, and the value written to PWM Period is the period less 1.

4) Clear PWMCON.4 to indicate that the program will now write to PWM Period.

5) PWMCON.5 must be set to either 0 or 1. If clear, the PWM Duty value (set in step 3) is the time the signal will be high. If it is set, the PWM Duty value is the time the signal will be low. PWM Duty must be less than PWM Period or the output will stay in the state defined by PWMCON.5

6) Write the PWM Period – 1 value to PWMLOW and PWMHI. The value is the total number of USEC ticks of the period of the PWM. In the above example, PWMLOW/HI is written with the value 5, because the total period is 6 ticks long, and the value written to PWM Period is the period – 1.

This can be expressed in code as:

```
PWMCON = 0x10; // Sel PWM Duty Register

PWM = 128–1; // PWM toggle at a count of 128

PWMCON = 0x09; // Sel PWM Period access, SysClk rate, PWM mode

PWM = 512–1; // 11.0592MHz/512=21.6KHz PWM Freq, Period=512 counts
```

Note:

The port pin used for PWM (P3.3) must be configured as either standard 8051 or CMOS output for the tone generator/PWM to function.

11.3.1 Example of PWM Tone Generation

Table 11–2 illustrates configuring the PWM for tone generation, and Table 11–3 explains selected statements.

Table 11–2. Configuring the PWM for Tone Generation

Stmt	‘C’ Source Code	Assembly Source Code
1	`// PWM`	`PUBLIC main`
		`RSEG ???main?PWM`
2	`#include <reg1210.h>`	
3	`#define OneUsConst (2–1)`	
4	`sbit p33=p3^3;`	
5	`void main(void)`	`Main:`
6	`{`	
7	`PDCON &= 0xED; // turn on tone gen & sys timer`	`ANL PDCON,#0Edh`
8	`USEC = OneUsConst;`	`MOV USEC,#01h`
9	`P33 = 1; // turn on P3.3`	`SETB p33`
10	`PWMCON = 0; // select PWMPeriod`	`MOV PWMCON,#00h`
11	`PWM = 5; // Set PWMPeriod`	`MOV PWMHI,#00h`
		`MOV PWMLOW,#05h`
12	`PWMCON = 0x10; // select PWMDuty`	`MOV PWMCON,#10h`
13	`PWM = 4; // Set PWMDuty`	`MOV PWMHI,#00h`
		`MOV PWMLOW,#04h`
14	`PWMCON = 0x09; // Enable PWM`	`MOV PWMCON,#09h`
15	`While(1) {}`	`SJMP $`
16	`}`	

Table 11–3. Statement Explanations

Statement #	Explanation
7	ANDing PDCON with ED$_H$ effectively turns off bits 1 (PDST) and 4 (PDPWM). Clearing the PDST (Power Down System Timer) bit turns the system timer on, while clearing the PDPWM (Power Down PWM module) bit turns the PWM module on.
8	Sets the USEC SFR to define 1µs, which will be used for determining the PWM timing.
9	P3.3 must be set to 1 prior to using PWM or tone generator. If P3.3 is clear, the PWM or tone generator will not produce any output.
10	Clearing bit 4 by setting PWMCON to 0 selects the PWM Period register, which will be written to in the next statement.
11	Having selected PWM Period in statement 10, this statement sets the PWM Period to 5.
12	Setting bit 4 by setting PWMCON to 10$_H$ select the PWM Duty register will be written to in the next statement.
13	Having selected PWM Duty in statement 12, this statement sets the PWM Duty to 4.
14	This statements enables the PWM.

11.3.2 Example of PWM Tone Generation Idling

When PWM is idling, system requirements for the PWM output varies (idle at low or high voltage). The output of P3.3 (Tone/PWM) is internal pull-high upon power-on reset—idle high. If idle low is needed, many methods can be used to initialize P3.3 to low.

Note:

If idle low on Tone/PWM is achieved by writing 0 to P3.3 (which will suppress PWM output), subsequently, writing a 1 to P3.3 will enable PWM output at any position of the PWM cycle.

The following program, shown in Table 11–4, is very similar to the one provided in the previous section. PWM Duty is initially set to zero, which idles the PWM. It is then reset to 4, at which point the function of the program continues as the program above. Table 11–5 explains selected statements.

Table 11–4. Configuring the PWM for Tone Generation with PWM Idling

Stmt	‘C’ Source Code	Assembly Source Code
1	`// PWM`	`PUBLIC main`
		`RSEG ???main?PWM`
2	`#include <reg1210.h>`	
3	`#define OneUsConst (2–1)`	
4	`sbit p33=p3^3;`	
5	`void main(void)`	`Main:`
6	`{`	
7	`PDCON &= 0xED; // turn on tone gen & sys timer`	`ANL PDCON,#0Edh`
8	`USEC = OneUsConst;`	`MOV USEC,#01h`
9	`P33 = 1; // turn on P3.3`	`SETB p33`
10	`PWMCON = 0; // select PWMPeriod`	`MOV PWMCON,#00h`
11	`PWM = 5; // Set PWMPeriod`	`MOV PWMHI,#00h`
		`MOV PWMLOW,#05h`
12	`PWMCON = 0x10; // select PWMDuty`	`MOV PWMCON,#10h`
13	`PWM = 0; // Set PWMDuty`	`MOV PWMHI,#00h`
		`MOV PWMLOW,#04h`
14	`PWMCON = 0x09; // Enable PWM`	`MOV PWMCON,#09h`
15	`for(i=0; i < 10; i++);`	`MOV R7,#00h`
		`Loop:`
		`INC R7`
		`CJNE R7,#0Ah,Loop`
16	`PWMCON = 0x10; // select PWMDuty`	`MOV PWMCON,#10h`
17	`PWM = 4; // Set PWMDuty`	`MOV PWMHI,#00h`
		`MOV PWMLOW,#04h`
18	`PWMCON = 0x09; // Enable PWM`	`MOV PWMCON,#09h`
19	`while(1) {}`	`SJMP $`
20	`}`	

Table 11–5. Statement Explanations

Statement #	Explanation
1–12	Same as previous program in section 11.3.1.
13	Sets PWM Duty to 0, thereby configuring the PWM tone generator for idle mode.
14	Enables PWM. The PWM is enabled, but is in idle mode due to the fact that PWM Duty is 0.
15	This loops for an arbitrary number of instructions.
16–20	Same as statements 12 to 16 in previous program in section 11.3.2.

11.3.3 Example of Updating PWM

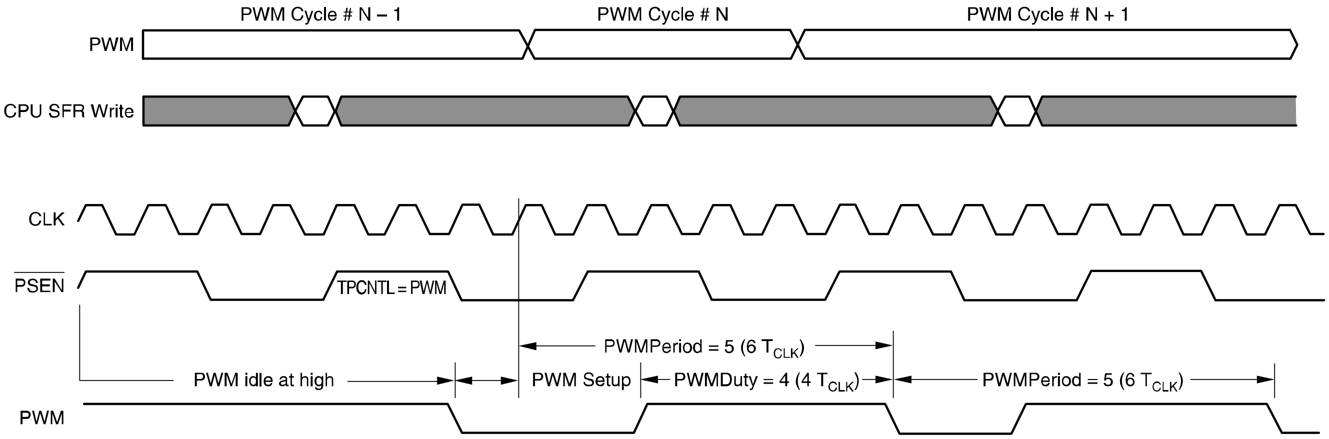
Both PWM Period and PWM Duty, set via the PWMHI and PWMLOW SFRs, are double-buffered. Their values are loaded to the 16-bit down counter and 16-bit PWMTemp register, respectively, when the counter expires.

PWM Period and PWM Duty may be renewed anytime during a PWM cycle. The newly updated values are effective on the next PWM cycle. Double–buffered operation is depicted in Figure 11–6.

PWM Period is accessed via the two 8-bit SFRs, PWMHI and PWMLOW. It is possible that while you are updating one of these two SFRs at the transition of two PWM cycles, PWM Period and PWM Duty are loaded to the counter PWMTemp. As a result, only a partial PWM Period or PWM Duty is updated. For those applications that need to avoid incomplete updates, the microcontroller could busy poll the P3.3 line to detect the transition of two PWM cycles and update the PWM SFRs after the transition is finished. However, busy polling will use up a high percentage of CPU time.

The $\overline{\text{INT1}}$ ISR can be used to detect the PWM cycle transition and update the PWM SFRs at the appropriate time because P3.3 is fed back to the CPU as $\overline{\text{INT1}}$. This is illustrated in the following program example:

Figure 11–6. PWM Timing



```
// PWM
#include <REG1210.H>
#define OneUsConst  (2–1)
#define CLEAR 0
#define SET      1
sbit p33=P3^3;
sbit p14=P1^4;
unsigned char p,d;
void pwm_isr( void) interrupt 2 //External Interrupt 1
{  p14=!p14;     // debug
   PWMCON &= 0xef;  // select PWMPeriod
   PWM=p;        // Set PWMPeriod
   PWMCON |=0x10;   // select PWMDuty
   PWM=d;
   IE1=CLEAR;       // Clear pending interrupt
   EX1=CLEAR;
}
void setpwm(period, duty)
{  p14=!p14;     // debug
   p=period; d=duty;
   IE1=CLEAR;       // Clear any pending interrupt
   EX1=SET;      // Enable *INT1 pin interrupt
}
void main(void)
{  char i;
   // Setup External INT1
   IT1=SET;      // Config *INT1 pin for falling edge trigger
   EA=SET;          // Global Int Enable
   PDCON &= 0x0ed;  //turn on tone gen & sys timer
   USEC = OneUsConst;
   p33=1;        // turn on P3.3
   PWMCON=0;     // select PWMPeriod
   PWM=500;      // Set PWMPeriod
   PWMCON=0x10; // select PWMDuty
   PWM=200;
   PWMCON=0x19; // Enable PWM
   for (i=0;i<5;i++) {;}
   setpwm(200,100); // set period/duty after current PWM cycle
   while(1) {}
}
```

Chapter 12

Analog-to-Digital Converter

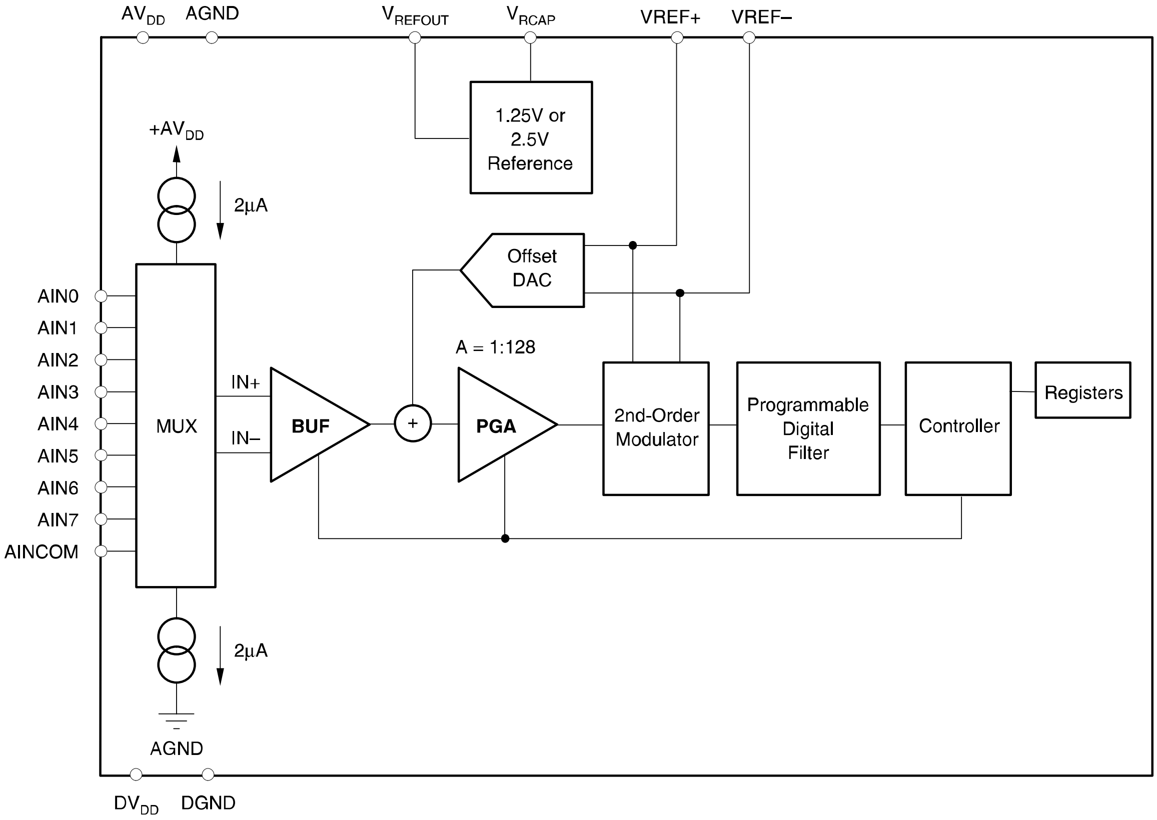
Chapter 12 describes the ADC of the MSC1210.

Topic **Page**

12.1 Description

The MSC1210 includes an ADC with 24-bit resolution. The ADC consists of an input multiplexer (MUX), an optional buffer, a programmable gain amplifier (PGA), and a digital filter. The architecture is described diagram in Figure 12–1.

Figure 12–1. MSC1210 Architecture

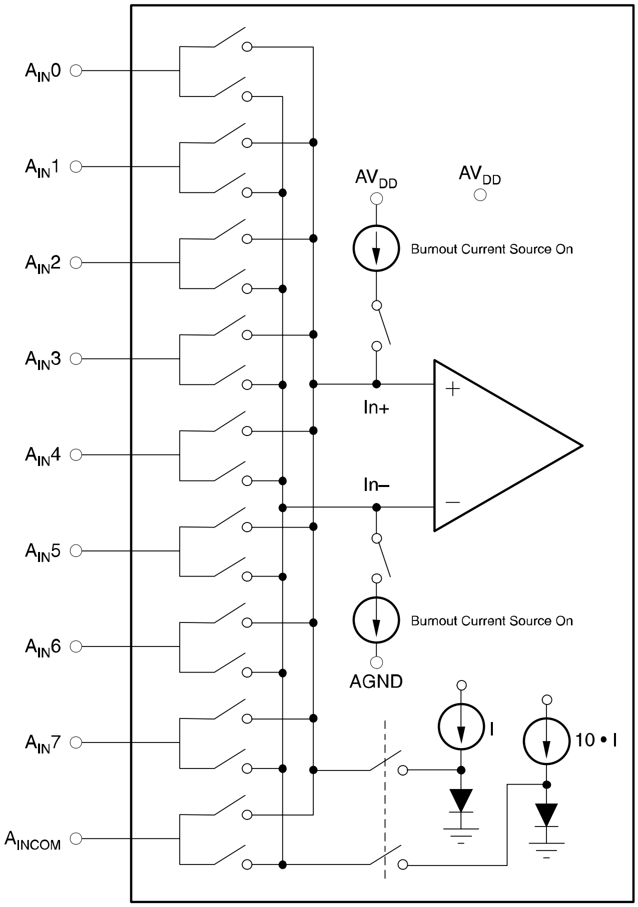


12.2 Input Multiplexer

The MSC1210 multiplexer is more flexible than a typical ADC in that each input pin can be configured as either a positive or negative input for a given measurement. While other ADC parts often define input pairs, the MSC1210 defines one pin as the negative input and the other as the positive input, thus providing complete design freedom in this respect. Any given input pin may serve as the negative input in one measurement and serve as the positive input in the next. Further, any combination of pins can be used—there are no predefined input pairs that restrict.

The input multiplexer provides for any combination of differential inputs to be selected on any of the input channels, as shown in Figure 12–2. For example, if channel 1 is selected as the positive differential input channel, any other channel can be selected as the negative differential input channel. With this method, it is possible to have up to eight fully differential input channels.

Figure 12–2. Input Multiplexer Configuration



The positive input channel and the negative input channel are selected in the ADC Multiplexer register (ADMUX, SFR D7h). The high four bits of ADMUX (bits 4 through 7) select the positive channel, while the low four bits (bits 0 through 3) select the negative channel. The ADMUX SFR has the following definition:

	7	**6**	**5**	**4**	**3**	**2**	**1**	**0**	**Reset Value**
SFR $D7_H$	INP3	INP2	INP1	INP0	INN3	INN2	INN1	INN0	01_H

INP3-0 (bits 7-4)—**Input Multiplexer Positive Channel**. This bit selects the positive signal input.

INP3	INP2	INP1	INP0	Positive Input
0	0	0	0	AIN0 (default)
0	0	0	1	AIN1
0	0	1	0	AIN2
0	0	1	1	AIN3
0	1	0	0	AIN4
0	1	0	1	AIN5
0	1	1	0	AIN6
0	1	1	1	AIN7
1	0	0	0	AINCOM
1	1	1	1	Temperature Sensor (requires ADMUX = FF_H)

INN3-0 (bits 3-0)—**Input Multiplexer Negative Channel**. This bit selects the negative signal input.

INN3	INN2	INN1	INN0	Negative Input
0	0	0	0	AIN0
0	0	0	1	AIN1 (default)
0	0	1	0	AIN2
0	0	1	1	AIN3
0	1	0	0	AIN4
0	1	0	1	AIN5
0	1	1	0	AIN6
0	1	1	1	AIN7
1	0	0	0	AINCOM
1	1	1	1	Temperature Sensor (requires ADMUX = FF_H)

Therefore, to select AIN1 as the positive channel and AIN6 as the negative channel, the following assignment would be made to the ADMUX register:

```
ADMUX = 0x16h; // 0001=AIN1, 0110=AIN6
```

By default, ADMUX defaults to 01_H at power up, so AIN0 is the default positive input and AIN1 is the default negative input.

12.3 Temperature Sensor

As shown in the chart above describing the ADMUX SFR, when all bits are set to 1 (i.e. ADMUX = FFh), all the MUX inputs (AIN0-7, AINCOM) are disconnected from the ADC, and the ADC inputs are connected to measure two diode junctions with different currents. This differential voltage will change linearly with temperature, thus providing an integrated linear temperature sensor.

When using the temperature sensor, the voltage returned by the ADC is used to determine the temperature in the following formula:

```
Float temp = α * volts – 282.14;
```

This converts the voltage into a temperature in degrees centigrade. The above temperature can, of course, be converted to Fahrenheit or Kelvin using standard conversion formulas. One value of α that gives good results is 2664.7. The value of α can vary from part to part and is determined from experimental data.

The following program is a simple example that returns the current temperature as detected by the MSC1210:

```
#include <REG1210.H>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define LSB 298.0232e–9 /* LSB=5.0/2^24 */
#define ALPHA 2664.7       /* derived for some devices */
extern void autobaud(void);
extern long bipolar(void);
void main(void)
{
   float volts, temp, resistance, ratio, lr, ave;
   int i, k, decimation = 1728, samples;

   CKCON = 0; // 0 MOVX cycle stretch
   autobaud();
   printf(”2MSC1210 ADC Temperature Test\n”);
   //Timer Setup
   USEC= 10; // 11MHz Clock
   ACLK = 9; // ACLK = 11,0592,000/10 = 1,105,920 Hz
             // modclock = 1,105,920/64 = 17,280 Hz
   // Setup ADC
   PDCON &= 0x0f7;  //turn on adc
   ADMUX = 0x0FF;   //Select Temperature Diodes
   ADCON0 = 0x30;   //Vref On, Vref Hi, Buff off, BOD off, PGA=1
   ADCON2 = decimation & 0xFF; // LSB of decimation
```

```
    ADCON3 =(decimation>>8) & 0x07; // MSB of decimation
    ADCON1 = 0x01; // bipolar, auto, self calibration, offset, gain
    printf (”Calibrating. . .\n”);
    for (k=0; k<4; k++)
            {
            // Wait for Four conversions for filter to settle
            // after calibration
            while(!(AIE & 0x20)); // Wait for data ready
            lr = bipolar(); // Dummy read to clear ADCIRQ
            }
    samples = 10; // The number of voltage samples we will average
    while(1)
            {
            ave = 0;
            for (i = 0; i < samples; i++)
                    {
                    while (!(AIE & 0x20)); // Wait for new next result
                    ave += bipolar() * LSB; // This read clears ADCIRQ
                    }
                volts = ave/samples;
                temp = ALPHA * volts – 282.14;
                printf (”V=%f, resistance=%f, Temp=%f degrees C\n”,
                            volts, resistance, temp);
        }// while
} //main
```

This program first configures the ADC, allows the ADC to self-calibrate, and then enters a loop where the temperature is sampled and reported to the user via the serial interface.

12.4 Burnout Current Sources

When the Burnout bit (BOD) is set in the ADC control register (ADCON0.6), two current sources are enabled that source approximately 2µA.

This allows for the detection of an open circuit (full-scale reading) or short-circuit (0V differential reading) on the selected input differential pair.

The following program illustrates a simple open-circuit and short-circuit detection routine.

```
#include <REG1210.H>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define LSB 298.0232e–9
extern void autobaud(void);
extern long bipolar(void);
void main(void)
{
   float sample, decimation = 1728;
   CKCON = 0;  // 0 MOVX cycle stretch
   autobaud();
   printf(”Brown–Out Detection\n”);
   //Timer Setup
   USEC= 10; // 11MHz Clock
   ACLK = 9; // ACLK = 11,0592,000/10 = 1,105,920 Hz
             // modclock = 1,105,920/64 = 17,280 Hz
   // Setup ADC
   PDCON &= 0x0f7; //turn on adc
   ADMUX = 0x01;
   ADCON0 = 0x70; // Vref On, Vref Hi, Buff off, BOD on, PGA=1
   ADCON2 = decimation & 0xFF;      // LSB of decimation
   ADCON3 =(decimation>>8) & 0x07;  // MSB of decimation
   ADCON1 = 0x01; // bipolar, auto, self calibration, offset, gain
   while(1)
   {
      while (!(AIE & 0x20));
      sample = bipolar() * LSB; // This read clears ADCIRQ
      printf (”Sample=%f”, sample);
      if(sample < 0.01)
         printf(” Short Circuit\n”);
      else if(sample > 2.4)
         printf(” Open Circuit\n”);
      else
         printf(“Normal Sensor Range\n”);
      while(!RI);
      RI = 0;
   }// while
} //main
```

The previous code detects either an open- or short-circuit situation based on the ADC sample. Also note that the comparison is less than 0.01, due to the fact that the ADC generally will not return exactly 0.

12.5 Input Buffer

The input buffer reduces the likelihood of an offset in the measurements taken by the ADC. It should be used whenever the characteristics of the input signal allow. Essentially, the only time the input buffer should not be used is if the maximum voltage on either analog input is more than 1.5V below the positive rail voltage.

The input impedance of the MSC1210 without the buffer is 5MΩ/PGA. With the buffer enabled, the impedance is typically 10GΩ, the input voltage range is reduced, and the analog power-supply current is higher. The buffer is controlled by the BUF bit in the ADC control register (ADCON0.3); setting BUF enables the input buffer, while clearing it disables the input buffer.

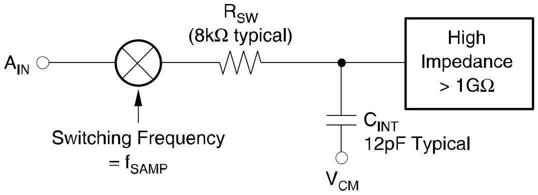
12.6 Analog Input

When the buffer is not selected, the input impedance of the analog input changes with clock frequency (ACLK F6H) and gain (PGA). The relationship is:

$$A_{IN}\ \mathit{Impedance}(\Omega) = \left(\frac{1 \cdot 10^6}{\mathit{ACLKFrequency}}\right) \cdot \left(\frac{5 \cdot 10^6}{\mathit{PGA}}\right)$$

Figure 12–3 shows the basic input structure of the MSC1210.

Figure 12–3. Basic Input Structure of the MSC1210



12.7 Programmable Gain Amplifier (PGA)

The Programmable Gain Amplifier (PGA) can be set to gains of 1, 2, 4, 8, 16, 32, 64, or 128. Using the PGA can actually improve the effective resolution of the ADC.

For example, with a PGA of 1 on a 5V full-scale range, the ADC can resolve to 1μV. With a PGA of 128 on a 40mV full-scale range, the ADC can resolve to 75nV. With a PGA of 1 on a 5V full-scale range, it would require a 26-bit ADC to resolve 76nV.

Another way of obtaining gain is by reducing the reference voltage. However, this approach quickly runs into noise limitations (at about 1V), whereby the noise itself becomes a larger component of the sample, thus reducing the benefit of the improved resolution from the lower reference voltage.

The PGA setting is set by modifying the three LSBs of the ADCON0 SFR. These three bits allow the software to set the PGA to any of the eight possible PGA settings listed in Table 12–1.

Table 12–1. PGA Settings

PGA2	PGA1	PGA0	GAIN
0	0	0	1 (default)
0	0	1	2
0	1	0	4
0	1	1	8
1	0	0	16
1	0	1	32
1	1	0	64
1	1	1	128

For example, the following instructions would have the following effects:

```
ADCON0 = 0x03; // Set PGA to 8

ADCON0 = 0x05; // Set PGA to 32
```

Notice that in both of these examples, the instruction will also clear all of the other bits of ADCON0, which may or not be desirable. To set the PGA as in the two previous examples without altering the other ADCON0 bits, the following instructions may be substituted:

```
ADCON0 = (ADCON0 & ~0x07) | 0x03; // Set PGA to 8

ADCON0 = (ADCON0 & ~0x07) | 0x05; // Set PGA to 32
```

12.8 Offset DAC

The input to the PGA can be shifted by half the full-scale input range of the PGA by using the Offset DAC (ODAC) register (SFR address: 0xE6). The ODAC register is an 8-bit value; the MSB is the sign and the seven LSBs provide the magnitude of the offset. Using the ODAC does not reduce the noise performance and increases the dynamic range of the ADC. The ODAC must be applied after any calibration is performed because the calibration will remove any offset induced by the ODAC.

$$Offset = \frac{V_{REF}}{2 \cdot PGA} \cdot \left(\frac{Code}{127}\right)$$

Note:

The input may only be shifted by half the full-scale input range. This means that if the input voltage range is 5V, it can be shifted ±2.5V. The range is divided by 256 and the LSB of the ODAC indicates an offset of that amount. Thus, given an input voltage range of 5V and an ODAC of 10_H (16), the input would be shifted by 313mV (i.e., 5.000V / 256 = 19.53mV • 16 = 312.5mV).

12.9 Modulator

The modulator is a single-loop second-order delta-sigma system. The modulator clock speed is derived from the oscillator frequency divided by the ACLK register (plus one) divided by 64. This can be summarized by the formula:

$$Analog\ Sample\ Rate = \frac{Oscillator\ Frequency/(ACLK + 1)}{64}$$

Thus, (given an oscillator frequency of 11.0592MHz), if ACLK = 8, the analog signal sample rate will be 11.0592MHz / (8 + 1) = 1.2288MHz / 64 = 19 200Hz.

The rate at which samples are made available to the user program running on the MSC1210 is less than that of the analog sample rate. The data output rate is determined by dividing the analog sample rate by the decimation value in the ADCON2 (low byte, SFR address: 0xDE) and ADCON3 (high byte, SFR address: 0xDF) registers. Therefore, in the above example that resulted in a 19 200Hz sample rate, if ADCON2 and ADCON3 together hold the value 1920, your program would be provided sample data at a rate of 10Hz (19 200Hz/1920 = 10Hz). The best noise performance is achieved with higher decimation values.

12.10 Calibration

The offset and gain errors in the MSC1210 ADC, or a complete measurement system, can be reduced with calibration. The calibration mode control bits in the ADCON1 register (SFR address: 0xDD) can select 5 different calibration processes. These include: internal (self) calibration of offset, gain, or both, and system calibration of offset or gain. Each calibration process takes seven t_{DATA} periods to complete. Therefore, it takes 14 t_{DATA} periods to complete self calibration of both offset and gain, which is represented by one mode control bit selection.

For system calibration, the appropriate signal must be applied to the inputs. The system calibration offset mode requires a zero differential input signal. It then computes an offset that will nullify the offset in the system. The system calibration gain mode requires a positive full-scale differential input signal. It then computes a value to nullify gain errors in the system. For example, in a weigh-scale application, the use of the system offset calibration could be used to null the system for a tare weight. Then the measurements that follow would only have the new weight in the output of the ADC.

Calibration should be performed after power on, a change in temperature, or a change of the PGA. For operation with a reference voltage greater than (AV_{DD} – 1.5V), the buffer must also be turned off during calibration. Calibration will remove the effects of the ODAC, therefore, changes to the ODAC register must be done after calibration, otherwise the calibration will remove the effects of the offset.

Table 12–2. Calibration Mode Control Bits

CAL2	CAL1	CAL0	Calibration Mode
0	0	0	No calibration (default)
0	0	1	Self calibration, offset and gain
0	1	0	Self calibration, offset only
0	1	1	Self calibration, gain only
1	0	0	System calibration, offset only
1	0	1	System calibration, gain only
1	1	0	Reserved
1	1	1	Reserved

The calibration is started by setting the CALx bits in the ADCON1 register. The ADC conversion interrupt will occur when the calibration is finished. If it is not masked, it will generate an interrupt or the bit can be monitored in the Peripheral Interrupt register (AISTAT.5, SFR address: 0xA7).

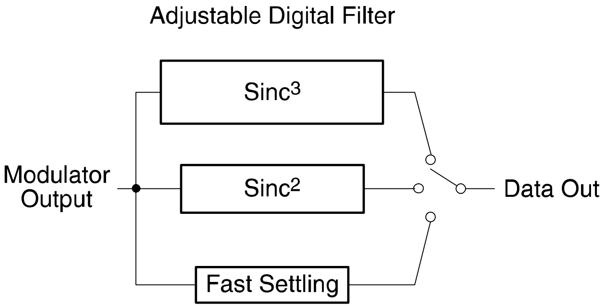
Thus, a full self-calibration, calibrating both offset and gain, may be executed in the following fashion:

```
ADCON1 = 0x01; // Initiate self–calibration, offset and gain

while(!(AISTAT & 0x20)); // Wait for interrupt to be triggered
```

12.11 Digital Filter

The digital filter can use either the fast settling, sinc^2, or sinc^3 filter, as shown in Figure 12–4. In addition, the auto mode changes the sinc filter to the best available option after the input channel or PGA is changed. When switching to a new channel, it will use the fast settling filter for the next two conversions, the first of which should be discarded. It will then use the sinc^2 followed by the sinc^3 filter to improve noise performance. This combines the low-noise advantage of the sinc^3 filter with the quick response of the fast settling time filter. The frequency response of each filter is shown in Figure 12–5.

Figure 12–4. Filter Step Responses



Filter Settling Time

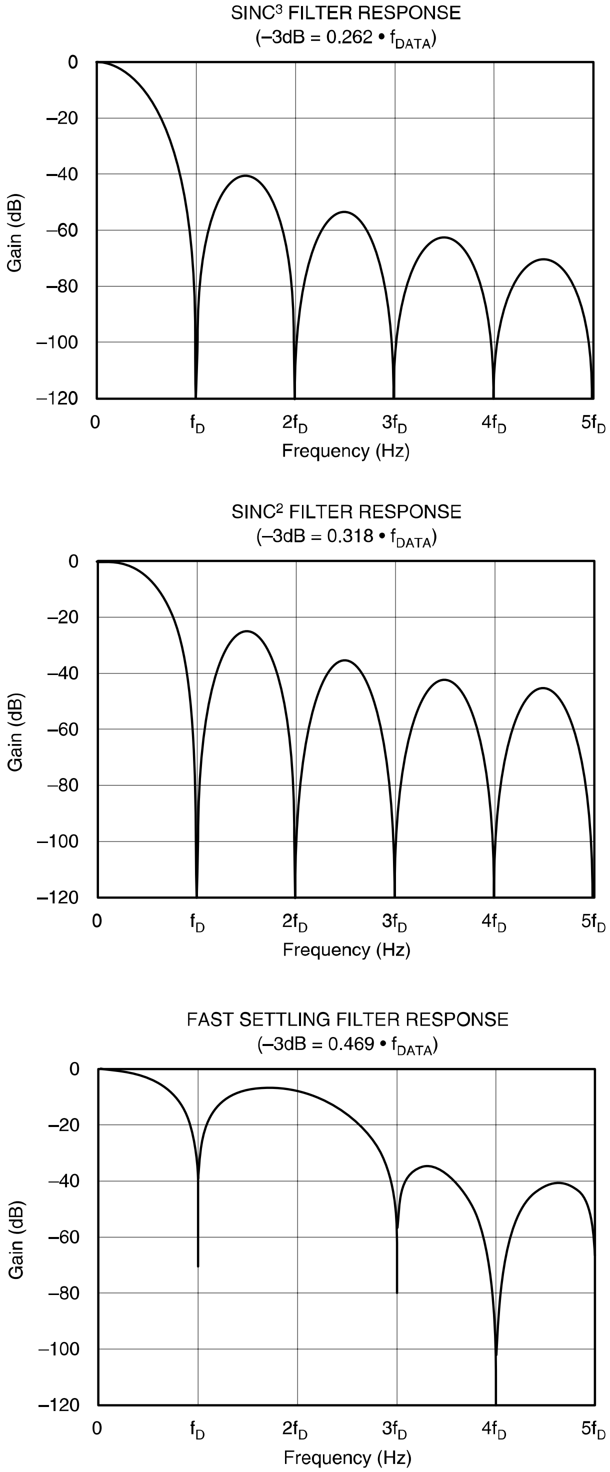
Filter	Settling Time (conversion cycles)
Sinc^3	3(1)
Sinc^2	2(1)
Fast	1(1)

NOTE: (1) With sychronized channel changes.

Auto Mode Filter Selection

Conversion Cycle			
1	2	3	4
Discard	Fast	Sinc^3	Sinc^3

Figure 12–5. Filter Frequency Responses



NOTE: f_D = Data Output Rate = $1/t_{DATA}$

12.11.1 Multiplexing Channels

When the input changes suddenly, it will take a certain amount of time for the output to correctly represent that new input. The amount of time required to correctly represent the new input depends on the type of filter being used. The filters are designed to settle in 1, 2 or 3 data output intervals. Up to an additional full period is required for an accurate sample because a change usually does not take place synchronous with the data output interval. Due to this uncertainty, as a matter of practice, one more cycle is used before the full resolution is obtained. Refer to Table 12–3 for the number of cycles that must be discarded when the input makes a significant shift.

Table 12–3. Filter Settling

Samples to Discard	Filter
1	Fast settling
2	Sinc^2
3	Sinc^3

Changing the input multiplexer usually creates the same type of step change on the input. The one significant difference is that the timing for the change is more precisely known.

Auto mode can reduce the amount of data that must be discarded, but it also reduces the resolution. Auto mode selects each of the different filter outputs after the input channel has changed. This means that the output uses the fast settling filter for 2 cycles, then sinc^2 for the next cycle and finally sinc^3 for all remaining cycles, until the channel is changed again.

When switching channels, the settling time must be factored in to determine the total throughput. For example, if the data rate is 20Hz and the filter is sinc^3, then with five channels it will give a resulting data rate on each channel of 20Hz / (4 samples per channel) / 5 channels = 1Hz data rate on each channel.

There are many trade-offs, however, that can be evaluated to determine the optimum setup. One of the first criteria is to determine the desired effective number of bits (ENOB). If 18 bits are needed, the same result could be achieved with all three types of filters. Using sinc^3, the decimation would be about 200, using sinc^2 it would be about 500, and with the fast-settling filter it would be about 1800. With a modulation clock (or sample rate) of 15 625, Table 12–4 shows the output data and channel rates.

Table 12–4. Output Data Rate and Channel Rate

Filter	Data Rate (Hz)	Channel Rate (Hz)	Synchronized (Hz)
Sinc3 (dec = 200)	78.125	/4 = 19.53	/3 = 26.04
Sinc2 (dec = 500)	31.25	/3 = 10.41	/2 = 15.625
Fast Settling (dec = 1800)	8.68	/2 = 4.34	/1 = 8.68

Notice that the speed difference for the synchronized channel changes are only different by a factor of 3, whereas the non-synchronized channel has a factor difference of 4.5.

These rates are all based on a reasonable speed for the modulation clock. In many applications, the mod clock can run as much as 10 times faster. That would make all of the times for throughput also 10 times faster, as shown in Table 12–5.

Table 12–5. Output Data Rate and Channel Rate (10x faster)

Filter	**Data Rate (Hz)**	**Channel Rate (Hz)**	**Synchronized (Hz)**
Sinc^3 (dec = 200)	780.125	/4 = 190.53	/3 = 260.04
Sinc^2 (dec = 500)	310.25	/3 = 100.41	/2 = 150.625
Fast Settling (dec = 1800)	80.68	/2 = 40.34	/1 = 80.68

12.12 Voltage Reference

The voltage reference used for the MSC1210 can either be internal or external. The power-up configuration for the voltage reference is 2.5V internal. The selection for the voltage reference is made through the ADCON0 register, bits 5 (internal/external selection) and 4 (1.25V/2.5V internal reference voltage).

Internal voltage reference is enabled by setting ADCON0.5 (EVREF, SFR address:0xDC), which is the default condition. When internal voltage reference is enabled, it may be selected as either 1.25V or 2.5V depending on the setting of ADCON0.4 (VREFH). Setting this bit sets the internal reference voltage to 2.5V, while clearing it sets the internal reference voltage to 1.25V (AVDD = 5V only).

When external voltage is selected, the external voltage reference is differential and is represented by the voltage difference between pins +VREF and –VREF. The absolute voltage on either pin (+VREF and –VREF) can range from AGND to AV_{DD}, however, the differential voltage must not exceed 5V. The differential voltage reference provides an easy means of performing ratiometric measurement.

The REFOUT pin should have a 0.1µF capacitor to AGND.

Note:

Enabling the internal V_{REF} does not eliminate the need for an external connection. The REFOUT pin must still be connected to VREF+, and VREF– must still be connected to AGND for normal operation with internal V_{REF}. The only thing that enabling internal V_{REF} does is enable the REFOUT pin.

12.13 Summation/Shifter Register

The MSC1210 includes a summation/shifter register that facilitates and increases the efficiency of certain common summation and shifting/division functions, especially those related to ADC conversions. The summation register is only active when the ADC is powered up. It is a 32-bit value that is broken into four 8-bit SFRs named SUMR0 (LSB), SUMR1, SUMR2, and SUMR3 (MSB).

The summation registers may function in one of four distinct modes:

Manual Summation—values written manually to the summation registers will be summed to the current sum (mode 0).

ADC Summation—a specified number of values returned by the ADC will automatically be summed to the current sum (mode 1).

Manual Shift/Divide—the current 32-bit value in the summation register is divided by a specified number. This division takes only four system cycles (mode 2).

ADC Summation with Shift/Divide—a specified number of values returned by the ADC will automatically be summed to the current sum, then divided by a specified number (mode 3).

The operation of the summation registers is controlled and configured with the SSCON ($E1_H$) SFR. In addition to controlling the four modes of operation, SSCON also is used to control how many samples will be taken from the ADC and by what value the final sum should be divided by, if any.

The individual bits of SSCON have the following functions:

	7	6	5	4	3	2	1	0	Reset Value
SFR $E1_H$	SSCON1	SSCON0	SCNT2	SCNT1	SCNT0	SHF2	SHF1	SHF0	00_H

The summation register is powered down when the ADC is powered down. If all zeroes are written to this register, the 32-bit SUMR3-0 registers will be cleared. The summation registers will do sign extend if bipolar is selected in ADCON1.

SSCON1-0 (bits 7-6)—**Summation Shift Control**.

Source	SSCON1	SSCON0	Mode
ADC	0	0	Values written to the SUM registers are accumulated when the SUMR0 value is written.
CPU	0	1	Summation register enabled. Source is ADC, summation count is working.
ADC	1	0	Shift enabled. Summation register is shifted by SHF Count bits. It takes four system clocks to execute.
CPU	1	1	Accumulate and shift enabled. Values are accumulated for SUM count times and then shifted by SHF count.

SSCON1 and SSCON0 (SSCON.7 and SSCON.6, respectively) control which of the four modes the summation register will operate in.

SCNT0, SCNT1, and SCNT2 (SSCON.3 through SSCON.5) are used to indicate how many ADC samples should be obtained and summed to the summation register. The number of samples that will be obtained and added are:

SCNT2	SCNT1	SCNT0	Summation Count
0	0	0	2
0	0	1	4
0	1	0	8
0	1	1	16
1	0	0	32
1	0	1	64
1	1	0	128
1	1	1	256

When the requested number of samples have been obtained and summed, a summation auxiliary interrupt will be triggered, if enabled.

SHF2, SHF1, and SHF0 (SSCON.0 through SSCON.2) are used to indicate by what value the final summation value should be divided. Specifically, the value indicates how many bits to the right the final summation value will be shifted, less one. Thus, a shift count of 0 reflects a final right shift by 1, which equates to a divide by 2. A shift count of 4 reflects a final right shift by 5, which equates to a divide by 32.

SHF2	SHF1	SHF0	Shift	Summation Count
0	0	0	1	2
0	0	1	2	4
0	1	0	3	8
0	1	1	4	16
1	0	0	5	32
1	0	1	6	64
1	1	0	7	128
1	1	1	8	256

12.13.1 Manual Summation Mode

The first mode of operation, manual summation, allows you to quickly add 32-bit values. In this mode, your program simply write the values to be added to the SUMR0, SUMR1, SUMR2, and SUMR3 SFRs. When a value is written to SUMR0, the current value of SUMR0-3 will be added to the summation register. For example, the following code will add 0x00123456 to 0x0051AB04:

```
SSCON = 0x00; // Clear summation register, manual summation
SUMR3 = 0x00; // High byte of 0x00123456
SUMR2 = 0x12; // Next byte of 0x00123456
SUMR1 = 0x034; // Next byte of 0x00123456
SUMR0 = 0x56; // Next byte of 0x0012345 – Perform addition
SUMR3 = 0x00; // High byte of 0x0051AB04
SUMR2 = 0x51; // Next byte of 0x0051AB04
SUMR1 = 0xAB; // Next byte of 0x0051AB04
SUMR0 = 0x04; // Next byte of 0x0051AB04 – Performs addition
ANSWER = (SUMR3 << 24) + (SUMR2 << 16) + (SUMR1 << 8) + SUMR0;
```

The previous code, although certainly more verbose than a simple ANSWER = 0x00123456 + 0x0051AB04 instruction in ‘C’, is much, much faster when analyzed in assembly language. In assembly language, the above solution requires just four MOV instructions for each summation, whereas the simple addition approach (which does not take advantage of the MSC1210 summation register) takes at least 8 MOV instructions and 4 ADD instructions.

12.13.2 ADC Summation Mode

The ADC summation mode functions very similarly to the manual summation mode, but instead of your program writing values to the SUMRx registers, the ADC writes values to the SUMRx registers.

In this mode, the CNT bits of SSCON are set to indicate how many ADC conversions should be summed in the summation register. The ADC will then deliver the requested number of results to the summation register and trigger a summation auxiliary interrupt, if enabled (see Chapter 10, *Interrupts*).

```
SSCON = 0x00; // Clear summation register, manual summation
SSCON = 0x50; // ADC summation, 8 samples from ADC
while(! (AISTAT & 0x40)); // Wait for 8 samples to be added
SUM = (SUMR3 << 24) + (SUMR2 << 16) + (SUMR1 << 8) + SUMR0;
```

The previous code first clears the summation registers by setting SSCON to 0, and then sets SSCON to ADC summation and requests that eight samples from the ADC be summed. The while() loop then waits for the summation auxiliary interrupt flag to be set, which indicates the requested operation was complete. The final line then takes the four individual SFRs and calculates the total summation value.

12.13.3 Manual Shift (Divide) Mode

The manual shift/divide mode provides a quick method of dividing the 32-bit number in the summation register by the value indicated by the SHF bits in SSCON. In assembly language terminology, this performs a 32-bit rotate right, dropping any bits shifted out of the least significant bit position.

For example, assuming the summation register currently holds the value 0x01516612, the following code will divide it by 8:

```
SSCON = 0x82; // Manual shift mode, divide by 8 (shift by 3)

SUM = (SUMR3 << 24) + (SUMR2 << 16) + (SUMR1 << 8) + SUMR0;
```

12.13.4 ADC Summation with Shift (Divide) Mode

The ADC summation with shift (divide) mode is a combination of ADC summation mode and manual shift mode. This mode will sum the number of ADC samples indicated by the CNT bits of SSCON, and then shift the final result to the right (divide) by the number of bits indicated by the SHF bits. This mode is useful when calculating the average of a number of ADC samples.

For example, to calculate the average of 16 ADC samples, the following code could be used (assuming the ADC had previously been correctly configured):

```
SSCON = 0x00; // Clear summation register, manual summation

SSCON = 0xDB; // ADC sum/shift, 16 ADC samples, divide by 16

while(! (AISTAT & 0x40)); // Wait for 16 samples to be added

SUM = (SUMR3 << 24) + (SUMR2 << 16) + (SUMR1 << 8) + SUMR0;
```

The previous code will clear the summation register, obtain 16 samples from the ADC, and then divide by 16, effectively calculating the average of the 16 samples.

12.14 Interrupt-Driven ADC Sampling

A useful, power-saving technique for obtaining ADC samples includes using the power-down mode of the MSC1210 between the time that a sample is requested and the time that a sample is made available to the MCU. During this time, the MSC1210 may be put into power-down mode by setting PCON.1 (PD). This will reduce power consumption significantly while the ADC sample is acquired.

The power–down mode is exited when the ADC unit triggers an interrupt. This interrupt will take the MCU out of power-down mode, execute the appropriate interrupt, and then continue with program execution.

```
#include <REG1210.H>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define LSB 298.0232e–9  /* LSB=5.0/2^24 */
extern void autobaud(void);
extern long bipolar(void);
long sample; // Hold the samples retrieved from A/D converter
void auxiliary_isr( void) interrupt 6 //AuxInt
{
   sample = bipolar() * LSB; // Read sample & clear ADCIRQ
   AI=CLEAR; // Clear Aux Int right before Aux ISR exit
}
void main(void)
{
   float volts, temp, resistance, ratio, lr, ave;
   int i, k, decimation = 1728, samples;
   CKCON = 0; // 0 MOVX cycle stretch
   autobaud();
   printf(”MSC1210 Interrupt–Driven ADC Conversion Test\n”);
   //Timer Setup
   USEC= 10; // 11MHz Clock
   ACLK = 9; // ACLK = 11,0592,000/10 = 1,105,920 Hz
             // modclock = 1,105,920/64 = 17,280 Hz
   // Setup interrupts
   EAI = 1; // Enable auxiliary interrupts
   AIE = 0x20; // Enable A/D aux. interrupt
   // Setup ADC
   PDCON &= 0x0f7; //turn on adc
   ADMUX = 0x01;  //Select AIN0/AIN1
   ADCON0 = 0x30;  // Vref On, Vref Hi, Buff off, BOD off, PGA=1
   ADCON2 = decimation & 0xFF; // LSB of decimation
   ADCON3 =(decimation>>8) & 0x07; // MSB of decimation
   ADCON1 = 0x01; // bipolar, auto, self calibration, offset, gain
   printf (”Calibrating. . .\n”);
   for (k=0; k<4; k++)
      {
      // Wait for Four conversions for filter to settle
      // after calibration.  We go to sleep.  When we wake
      // up, the interrupt will have read the sample.
      PCON |= 0x02; // Go to power–down until sample ready
      }
   samples = 10; // The number of voltage samples we will average
   while(1)
      {
      ave = 0;
      for (i = 0; i < samples; i++)
         {
         PCON |= 0x02; // Go to power–down until sample ready
         ave += bipolar() * LSB; // This read clears ADCIRQ
         }
      printf(“Average sample=%f\n”, ave / samples);
   }// while
} //main
```

12.15 Syncronizing Multiple MSC1210 Devices

In some circumstances, it may be desirable to have data conversion synchronized between several devices. In order to synchronize the MSC1210, each of the devices will need to power down their ADCs (stop the clock), and then all devices restart their ADCs at the same time.

For this explanation, we assume that one of the input port pins is defined to be the sync pin. A master device will raise the signal high when the MSC1210 should prepare for synchronization. When the MSC1210 senses the high signal on the sync input, it waits for the next ADC conversion to be completed. The ADC interrupt can be used as described in the previous section. After the ADC interrupt, the PDAD bit in the PDCON ($F1_H$) register is set to 1 to power down the ADC. The MSC1210 continues to monitor the sync input and when it goes low, the PDAD bit is set back to zero, thereby activating the ADC.

In summary, synchronizing the MSC1210 can be achieved with the following steps:

1) Start ADC operation (PDAD = 0).
2) Monitor sync input.
3) When sync = 1, wait for the ADC IRQ, then set PDAD = 1 (power down the ADC = stop clocks).
4) Wait for sync = 0, then set PDAD = 0, which restarts the ADC.
5) The ADC is now synchronized with the sync input and, therefore, with other MSC1210 devices that followed the same sync signal. They are also synchronized to within a few CPU clock cycles.

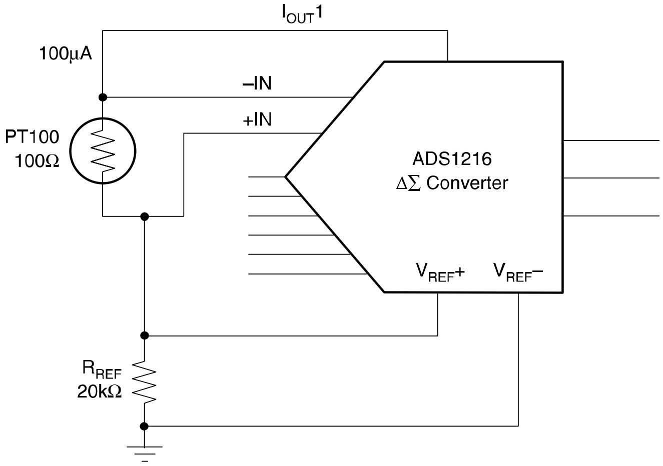
The following example program illustrates this method of syncronization:

```
#include <REG1210.H>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define LSB 298.0232e–9  /* LSB=5.0/2^24 */
extern void autobaud(void);
extern long bipolar(void);
void main(void)
{
   float volts, temp, resistance, ratio, lr, ave;
   int i, k, decimation = 1728, samples;
   autobaud();
   printf(”MSC1210 Sync Example \n”);
   //Timer Setup
   USEC = 10;    // 11MHz Clock
   ACLK = 9;        // ACLK = 11,0592,000/10 = 1,105,920 Hz
                 // modclock = 1,105,920/64 = 17,280 Hz
   // Setup ADC
   PDCON &= 0x0f7;  // turn on adc
   ADMUX = 0x01;// Select AIN0/AIN1
   ADCON0 = 0x30;   // Vref On, Vref Hi, Buff off, BOD off, PGA=1
   ADCON2 = decimation & 0xFF; // LSB of decimation
   ADCON3 =(decimation>>8) & 0x07; // MSB of decimation
   ADCON1 = 0x01; // bipolar, auto, self calibration, offset, gain
   while(sync == 0); // As long as sync is low, wait
   // Now that sync is low, shut down ADC.
   PDCON |= 0x08;
   while(sync == 1); // As long as Sync is high, wait.
   // When sync goes low, turn on ADC and continue
   PDCON = ~0x08;
   // At this point ADC is on and multiple MSC1210’s using the
   // same Sync signal will be in syncronization.
} //main
```

12.16 Ratiometric Measurements

Ratiometric measurements may be used to eliminate potential inaccuracy from the ADC process. Ratiometric measurements are obtained in a circuit similar to the one shown in Figure 12–6, where the same source used to drive the reference voltage (V_{REF}) is used to drive the ADC (–IN). This allows measurements to be taken without the accuracy of the voltage of V_{REF} being a factor in the measurement or in potential errors because the ratio between the –IN and $–V_{REF}$ will be constant, regardless of the accuracy of the voltage of +IN.

Figure 12–6. Circuit Drawing



The voltage measured is a ratio of the resistances R_{REF} and PT100 because the same current flows through the sense element (PT100) and the reference resistor (R_{REF}). Any errors in $I_{OUT}1$ do not enter into the accuracy of the measurment because, as shown in the following equations, I_{OUT} is effectively cancelled out:

$$V_{IN} = PT100 \cdot I_{OUT}$$

$$V_{REF} = R_{REF} \cdot I_{OUT}$$

$$ADC\ Result = \frac{V_{IN}}{V_{REF}}$$

$$ADC\ Result = \frac{(PT100 \cdot I_{OUT})}{(R_{REF} \cdot I_{OUT})} = \frac{PT100}{R_{REF}}$$

This eliminates both the reference voltage and the current source as sources of accuracy error and is only limited by the accuracy of the reference resistor and performance of the PT100. A high-precision reference resistor is readily obtainable. This is much easier than trying to get the same precision and accuracy from a voltage reference.

12.16.1 Differential V_{REF}

One application would be a system where the measurement and the ADC are on different grounds. Normally, you might have a voltage source that connects to a sensor, and the bottom of the sensor connects to the reference resistor. However, with two grounds, that can be different by more than 0.3V—that does not work. In such a case, you will need to connect the reference resistor from the power supply to the sensor, and then connect the sensor to GND2. Now you can still use the reference resistor to set the reference voltage, even though the voltages are between 2.5V to 4.5V.

The differential reference inputs, however, can be used for both grounded and non-grounded applications. For example, you might have a sensor that must be grounded (because of mechanical mounting). In that case the excitation could go through the reference resistor before the sensor.

Chapter 13

Serial Peripheral Interface (SPI)

Chapter 13 describes the serial peripheral interface (SPI) of the MSC1210 ADC.

Topic **Page**

13.1 Description

The MSC1210 includes a serial peripheral interface (SPI) module that allows simple and efficient access to SPI-compatible devices via a number of SFRs provided for that purpose. The SPI is an independent serial communications subsystem that allows the MSC1210 to communicate synchronously with SPI peripheral devices and other microprocessors. The SPI is also capable of interprocessor communication in a multiple master system. The SPI system can be configured as either a master or a slave device.

The maximum data transfer rates can be as high as 1/2 the f_{OSC} clock rate (12Mbits per second for a 24MHz f_{OSC} frequency).

13.2 Functional Description

The central element in the SPI system is the block containing the shift register and the read data buffer. SPI data is transmitted and received simultaneously. For every byte that is sent, a byte is also received. The system is double-buffered in the transmit direction and double-buffered in the receive direction. This means that new data for transmission can be written to the SPIDATA register before the previous transfer is complete. Additionally, received data is transferred into a parallel read data buffer, so the shifter is free to accept a second serial character. As long as the first character is read out of the SPIDATA register before the next serial character is ready to be transferred, no overrun condition occurs.

For FIFO operation, the reading of the received data can be delayed up to the length of time it takes to fill the FIFO. The SPIDATA register is used for reading data received, and for writing data to be sent, as shown in Figure 13–1.

Figure 13–1. SPI block diagram

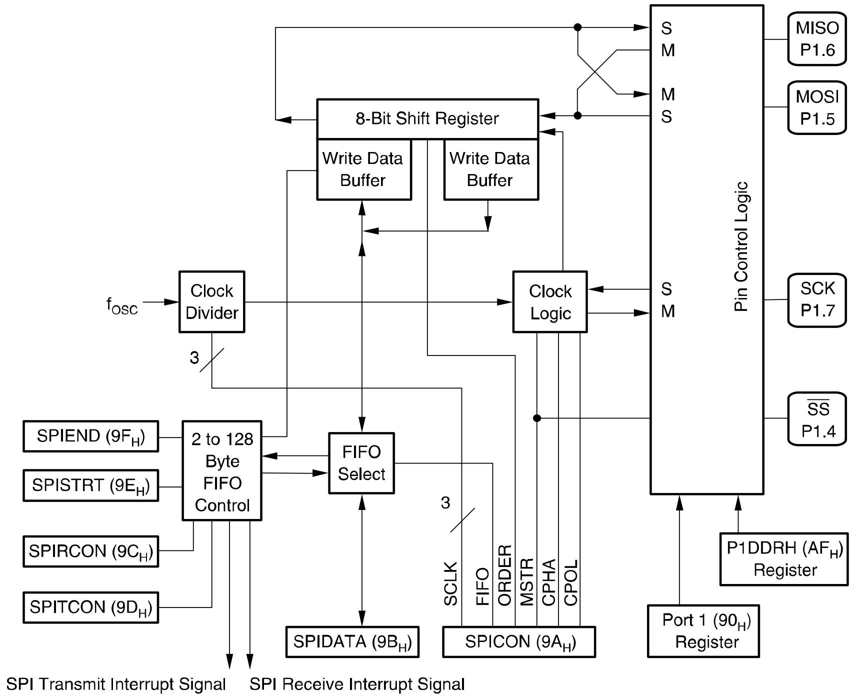
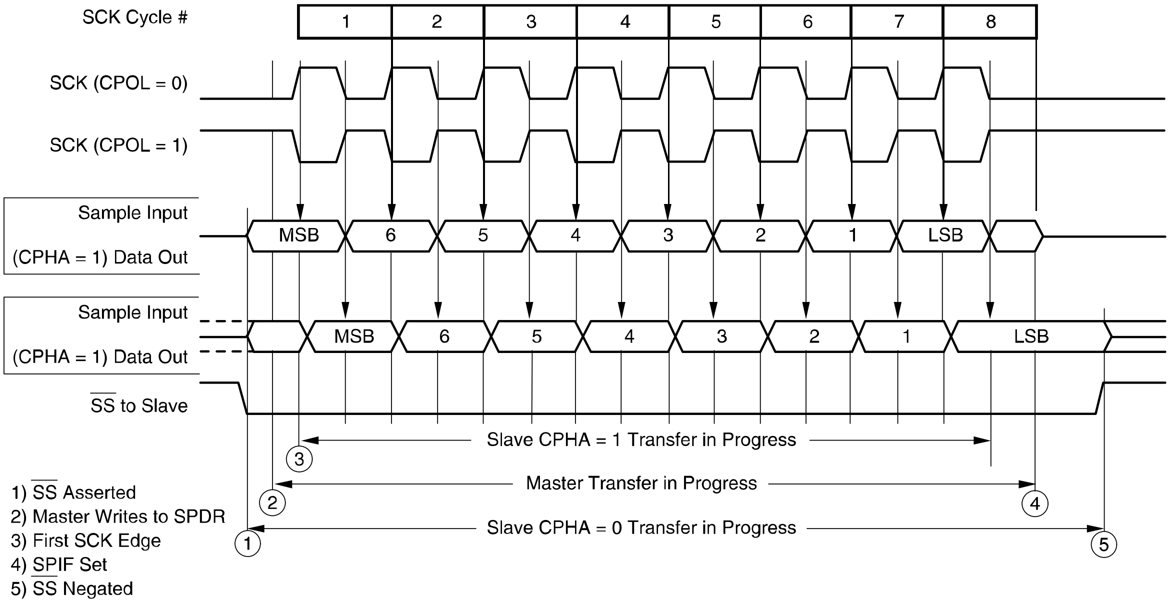


Figure 13–2. SPI Clock/Data Timing



During an SPI transfer, data is simultaneously transmitted and received. A serial clock line synchronizes shifting and sampling of the information on the two serial data lines.

A slave-select line allows individual selection of a slave SPI device; slave devices that are not selected do not interfere with SPI bus activities. On a master SPI device, the select line can optionally be used to indicate a multiple master bus contention (refer to Figure 13–2).

A section of internal RAM from 80_H to FF_H can be used as a FIFO to extend the buffering for receive and transmit. The size of the FIFO can range in size from 2 to 128 bytes.

13.3 Clock Phase and Polarity Controls

Software can select one of four combinations of serial clock phase and polarity using two bits in the SPI control register (SPICON $9A_H$). The clock polarity is specified by the CPOL control bit, which selects an active high or active low clock, and has no significant effect on the transfer format.

The clock phase (CPHA) control bit selects one of two different transfer formats. The clock phase and polarity should be identical for the master SPI device and the communicating slave device. In some cases, the phase and polarity are changed between transfers to allow a master device to communicate with peripheral slaves having different requirements.

When CPHA = 0, the SPI standard defines that the $\overline{SS}$ line must be negated and reasserted between each successive serial byte. This is more difficult when using the FIFO to transmit the bytes and cannot be done at higher clock speeds.

When CPHA = 1, the $\overline{SS}$ line can remain low between successive transfers.

13.4 SPI Signals

The following paragraphs contain descriptions of the four SPI signals: master in slave out (MISO), master out slave in (MOSI), serial clock (SCK), and slave select ($\overline{\text{SS}}$).

The port register for P1.4, P1.5, P1.6 and P1.7 must be set (P1 = Fx_H) to use the SPI functions. Additionally, the pins must be setup as inputs or outputs using the Port 1 Data Direction register (P1DDRH, AF_H). For master operation, P1DDRH = 75_H (drive $\overline{\text{SS}}$ pin), and slave P1DDRH = DF_H.

13.4.1 Master In Slave Out

MISO is one of two unidirectional serial data signals. It is an input to a master device and an output from a slave device. The MISO line of a slave device is placed in the high-impedance state if the slave device is not selected.

13.4.2 Master Out Slave In

The MOSI line is the second of the two unidirectional serial data signals. It is an output from a master device and an input to a slave device. The master device places data on the MOSI line a half-cycle before the clock edge that the slave device uses to latch the data.

13.4.3 Serial Clock

SCK, an input to a slave device, is generated by the master device and synchronizes data movement in and out of the device through the MOSI and MISO lines. Master and slave devices are capable of exchanging a byte of information during a sequence of eight clock cycles.

There are four possible timing relationships that can be chosen by using control bits CPOL and CPHA in the SPI control register (SPICON). Both master and slave devices must operate with the same timing. The SPI clock rate select bits, CLK[2:0], in the SPICON of the master device select the clock rate. In a slave device, CLK [2:0] have no effect on the operation of the SPI.

13.4.4 Slave Select

The $\overline{\text{SS}}$ input of a slave device must be externally asserted before a master device can exchange data with the slave device. $\overline{\text{SS}}$ must be low before data transactions and must stay low for the duration of the transaction.

There is no hardware support for mode fault error detection. For the master to monitor the $\overline{\text{SS}}$ line, it either needs to poll the status of the $\overline{\text{SS}}$ signal or connect it to $\overline{\text{INT}}0$ or $\overline{\text{INT1}}$, which can generate an interrupt when the line goes low. Due to this, it is reasonable for the master to drive P1.4 as the $\overline{\text{SS}}$ signal for control of the slave devices.

The state of the master and slave CPHA bits affects the operation of $\overline{\text{SS}}$. CPHA settings should be identical for master and slave. When CPHA = 0, the shift clock is the OR of $\overline{\text{SS}}$ with SCK. In this clock phase mode, $\overline{\text{SS}}$ must go high between successive characters in an SPI message. When CPHA = 1, $\overline{\text{SS}}$ can be left low between successive SPI characters. In cases where there is only one SPI slave MCU, its $\overline{\text{SS}}$ line can be tied to DGND as long as only CPHA = 1 clock mode is used.

13.5 SPI System Errors

Some SPI systems define two types of system errors: write collision and mode fault. Write collision is defined to occur when a byte is written to the transmit register before the previous byte was sent. Mode fault is an error that occurs in multiple master systems when two masters try to write at the same time.

There is no need to worry about write collision errors because the SPI transmit path is double-buffered. However, care should be taken to assure that more bytes are not written to the SPIDATA register before the previous bytes have been transferred. With the FIFO operation, when the FIFO is filled, the next writes to the SPIDATA register are ignored.

When the SPI system is configured as a master and the $\overline{SS}$ input line goes to active low, a mode fault error has occurred—usually because two devices have attempted to act as master at the same time. In cases where more than one device is concurrently configured as a master, there is a chance of contention between two pin drivers. For push-pull CMOS drivers, this contention can cause permanent damage. Care should be observed to protect against excessive currents in a multi-master system because the MSC1210 does not detect a mode fault.

13.6 Data Transfers

The transmitted and received data for SPI transfers are both double-buffered. This means that a second byte can be written for transmit before the first byte has been sent. Data that is received does not have to be read from the SPIDAT register until just before the next byte is received. The size of this buffer can essentially be extended with the FIFO mode. This adds from 2 to 128 bytes of FIFO memory.

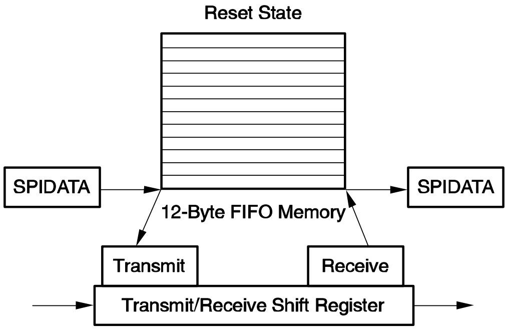
The FIFO mode uses a portion of the internal indirect RAM from 80_H to FF_H. The start and end of the FIFO portion of memory is set with the SPISTRT ($9E_H$) and SPIEND ($9F_H$) registers. The only restriction on those addresses is that the value of SPIEND must be larger than SPISTRT. The most significant bit is forced to a one.

There is no signal that switches the SPI interface on or off. It can be powered down using the PDCON ($F1_H$) register. However, if it is powered up, then it is operational. For the master, all that is necessary to transmit a byte is to write the value to SPIDATA ($9B_H$). The $\overline{SS}$ pin is not used in master mode. It can be used to drive an $\overline{SS}$ signal. For slave operation, the bytes will not transfer until $\overline{SS}$ is asserted and the clock signals are received.

For slave mode, if the $\overline{SS}$ signal goes high while a byte is being received, that byte is immediately flagged as completed and the interface is prepared for a new byte.

The SPICON ($9A_H$) register controls the SCLK frequency for master operation, and has bits to enable the FIFO, master mode, set bit order, clock polarity and phase. Any change to the SPICON register resets the SPI interface, and clears the counters and pointers, as shown in Figure 13–3.

Figure 13–3. SPI Reset State



The SPI Receive control register, SPIRCON ($9C_H$), controls the data receive operation. The receive buffer can be flushed with the write only RXFLUSH bit. A flush operation changes the SPI receive pointer so that it points to the same address as the FIFO IN pointer, and clears the receive counter. The receive counter indicates the number of bytes that have been received. An interrupt can be generated when the receive count equals or exceeds a chosen number. If the interrupt is not masked in the AISTAT register, the SPI received interrupt will cause a AI interrupt. The PPIRQ register is used in the AI interrupt routine to determine the source of the interrupt. The SPI receive interrupt can be monitored in the AISTAT register.

The SPI Transmit control register, SPITCON ($9D_H$), controls the data transmit operation. The transmit buffer can be flushed with the write only TXFLUSH bit. A flush operation changes the SPI transmit pointer so that it points to the same address as the FIFO OUT pointer, and clears the transmit counter. The transmit counter indicates the number of bytes in the transmit buffer (FIFO and buffer). An interrupt can be generated when the transmit count is less than or equal to a chosen number. If the interrupt is not masked in the AISTAT register, the SPI transmit interrupt with cause a auxiliary interrupt. The SPI transmit interrupt can be monitored in the AISTAT register.

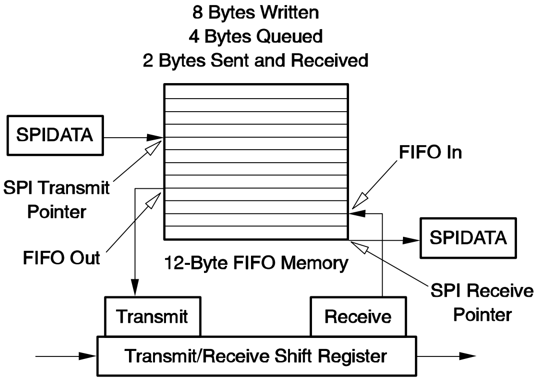
13.7 FIFO Operation

Data transmitted by the SPI interface is written to the SPIDATA register. If the FIFO is enabled, it is stored in the FIFO memory. The first two bytes are immediately written to the transmit buffer, and the SPI transmit pointer is incremented. For each byte transmitted using the SCLK signal, a byte is also received. The received bytes are immediately transferred to the FIFO. The FIFO IN pointer increments for each byte received until one less than the SPI received pointer. If the received bytes are not read or flushed, then additional SCLKs will continue to send until the last byte is sent. Therefore, if the SPI is used to only transmit bytes, the SPI receive interrupt can be used to flush the received bytes so that transmission of data is not blocked.

The SPI interrupts can be used to achieve maximum throughput. The size of the FIFO can be adjusted from 2 to 128 bytes depending on the allowable interrupt latency. For example, assume that the application has time critical operations that cannot be interrupted for 10µs. Using an 11.0592MHz crystal and if SPI clock is $f_{OSC}/2$, one byte can be shifted out in 1.46µs, or 69 bytes in 100µs. By setting the transmit IRQ level for 8, it would require that the FIFO be at least 77 bytes. If not receiving bytes, but simply flushing the receive buffer, the IRQ level for the receive interrupt has to be taken into account. For example, to allow the receive buffer to grow to 32 before generating an interrupt, add 32 to the 69 transfers. That gives a minimum buffer size of 101. A FIFO of 100 bytes would be adequate because two bytes are stored in the buffer register and shift register.

When using the FIFO, there is no mechanism to remove and reassert the $\overline{SS}$ line between each byte transferred, which is required for CPHA = 0. For slower transfer rates, it is possible for the program to monitor the SCLK using $\overline{INT5}$ and control the $\overline{SS}$ signal as needed.

Figure 13–4. SPI FIFO Operation



13.8 Code Examples

13.8.1 SPI Master Transfer in Double-Buffer Mode using Interrupt Polling

Example 13–1. SPI Master Transfer in Double-Buffer Mode using Interrupt Polling

```
1   #include ”MSC1210.H”
2   #include <Stdlib.h>
3   char spi_tx_rx ( char tx_data ) {
4       while((AIE&0x08)!=0x08){ } SPIDATA=tx_data;    // Wait until SPItx is set.
5       while((AIE&0x04)!=0x04){ } return(SPIDATA);    // Wait until SPIrx is set.
6   }
7   void main(void)
8   {
9       char j;
10      P1DDRH = 0x75;      // P1.7,P1.5,P1.4 = Outputs P1.6 = Input
11      // P1DDRH = 0xDA;   // P1.7,P1.5,P1.4 = Inputs P1.6 = Output
12      PDCON &= 0xFE;      // Turn on SPI power
13      SPICON=0xF6;        // ClkDiv=111(clk/256), DMA=0, Order=0, M/S=1, CPHA=1, CPOL=0
14      // SPICON=0x02;     // ClkDiv=Doesnt matter, DMA=0, Order=0, M/S=0, CPHA=1, CPOL=0
15      j=spi_tx_rx(0x78); // Transmit data value=0x78H, Return value is the received data
16  }
```

Example 13–1 is for a simple SPI master in double-buffer mode using interrupt polling. By changing two lines the code can also be used as a slave.

In line 10, the port direction for the pins that are used by the SPI. P1.7(SCLK pin), P1.5 (MOSI) and P1.4 ($\overline{\text{SS}}$) is configured as output, whereas pin P1.6 (MISO) is configured as input, because we are going to use the device as master. When configured as a slave, line 10 is commented out and line 11 is uncommented (line 11 is commented out in Example 13–1).

In line 12, the SPI is powered up by writing to PDCON.

In line 13, the SPICON register is set to put the SPI in master mode, double-buffer mode, with order = 0, CPHA = 1 and CPOL = 1, and the transfer clock rate at clk/256. Line 13 must be commented out and line 14 must be uncommented if the device is to be configured for slave-mode operation.

Line 15 calls the subroutine spi_tx_rx. The input to the subroutine is the data that is to be transmitted and the output is the data that is received.

Line 4 polls AIE[3] (ESPIT) to check if the interrupt is on, which indicates that the transmit buffer is empty. Once the buffer is empty, the next byte can be written for transmission.

Line 5 polls AIE[2] (ESPIR) to check if the interrupt is ON, which indicates that the receive buffer is full. Once this buffer is full, the received byte can be read.

Note:

Some applications require receive-only or transmit-only operation. In these cases, the subroutine needs to be modified accordingly.

13.8.2 SPI Master Transfer in FIFO Mode using Interrupts

Example 13–2. SPI Master Transfer in FIFO Mode using Interrupts

```
1  #include ”MSC1210.H”
2  void main(void)
3  {
4      P1DDRH = 0x75;   // P1.7,P1.5,P1.4=output P1.6=input
5      PDCON &= 0xFE;   // Turn on SPI power
6      SPIRCON=0x83;    // Flush RxBuf, RXlevel=4 or more
7      SPITCON=0xAA;    // Flush TxBuf, DrvEnb=1, SCLK Enable=1 Txlevel=4 or less
8      SPISTRT=0x00;    // Star address = 0
9      SPIEND= 0x08;    // End address = 8
10     SPICON=0x36;     // ClkDiv=001 (clk/4), dma=1, Order=0,M/S=1,CPHA=1,CPOL=0
11     AIE = 0x0C;      // SPI Transmit IRQ and SPI Receive IRQ Enabled
12     AI=0;            // Clear the external interrupt flag
13     EAI=1;       // Enable the external interrupts.
14     while(1) { }
15 }
16 void monitor_isr() interrupt 6
17 {
18     if(AISTAT==0x04) {read_4_bytes();}   // Checking for SPIRX IRQ
19     if(AISTAT==0x08) {send_4_bytes();}   // Checking for SPITX IRQ
20     AI=0;
21 }
```

Example 13–2 is for a simple SPI master in FIFO mode using interrupts.

In line 4, the the port direction is set for the pins that are used by the SPI. Pins P1.7 (SCLK pin), P1.5 (MOSI) and P1.4($\overline{SS}$) are configured as outputs and pin P1.6 (MISO) is configured as input because the device will be used in master mode.

In line 5, the SPI module is powered up by writing to the PDCON.

In line 6 the Rxlevel is set to 4 and the RXBuffer content is flushed, if any exists. Thus, SPIRXIRQ goes high whenever more than 4 bytes of data are received.

In line 7, the Txlevel is set to 4 and the TXBuffer content is also flushed, if any exists. The data and clock lines are also enabled by setting bit 3 and bit 5, respectively, of register SPITCON. The SPITXIRQ goes high if there are 4 or fewer bytes to transmit.

In line 8, the SPISTRT SFR is cleared, i.e., the start address of the buffer is at 0.

In line 9, the end address SPIEND is set to to 8, creating a buffer size of 8 bytes.

Line 10 sets the SPICON register. The clock is configured to run at clk/4 speed. The other configuration settings are master mode, FIFO mode, order = 0, CPHA = 1 and CPOL = 0.

Line 11 enables the SPIRX and SPITX interrupts, after which the AI flag is cleared and the EAI flag is enabled. There is no data to transmit, so SPITXIRQ goes up and we go to the monitor_isr() routine. The SPITX IRQ went up, so the subroutine send_4_bytes is called, where the program writes to the SPIDATA register 4 times.

In line 20, the AI flag is cleared. The interrupt SPITX goes on again immediately because the interrupt is configured to be triggered when the number of bytes to transmit is 4 or fewer. The number to bytes to transmit are 4, so the interrupt is triggered and 4 additional bytes are subsequently written to the buffer.

Thus, the buffer is completely filled with bytes to be transmitted. As one byte is transmitted, an additional byte is written. Once 4 bytes are transmitted, 4 bytes will be received, at which point both transmit and receive interrupts go high. At that point the interrupt routine is executed, first reading the 4 received bytes and then writing 4 more bytes to be transmitted. In this manner, the buffer is always used to its fullest extent without overflowing in either direction.

Chapter 14

Additional MSC1210 Hardware

Chapter 14 describes addtional hardware on the MSC1210 ADC.

Topic **Page**

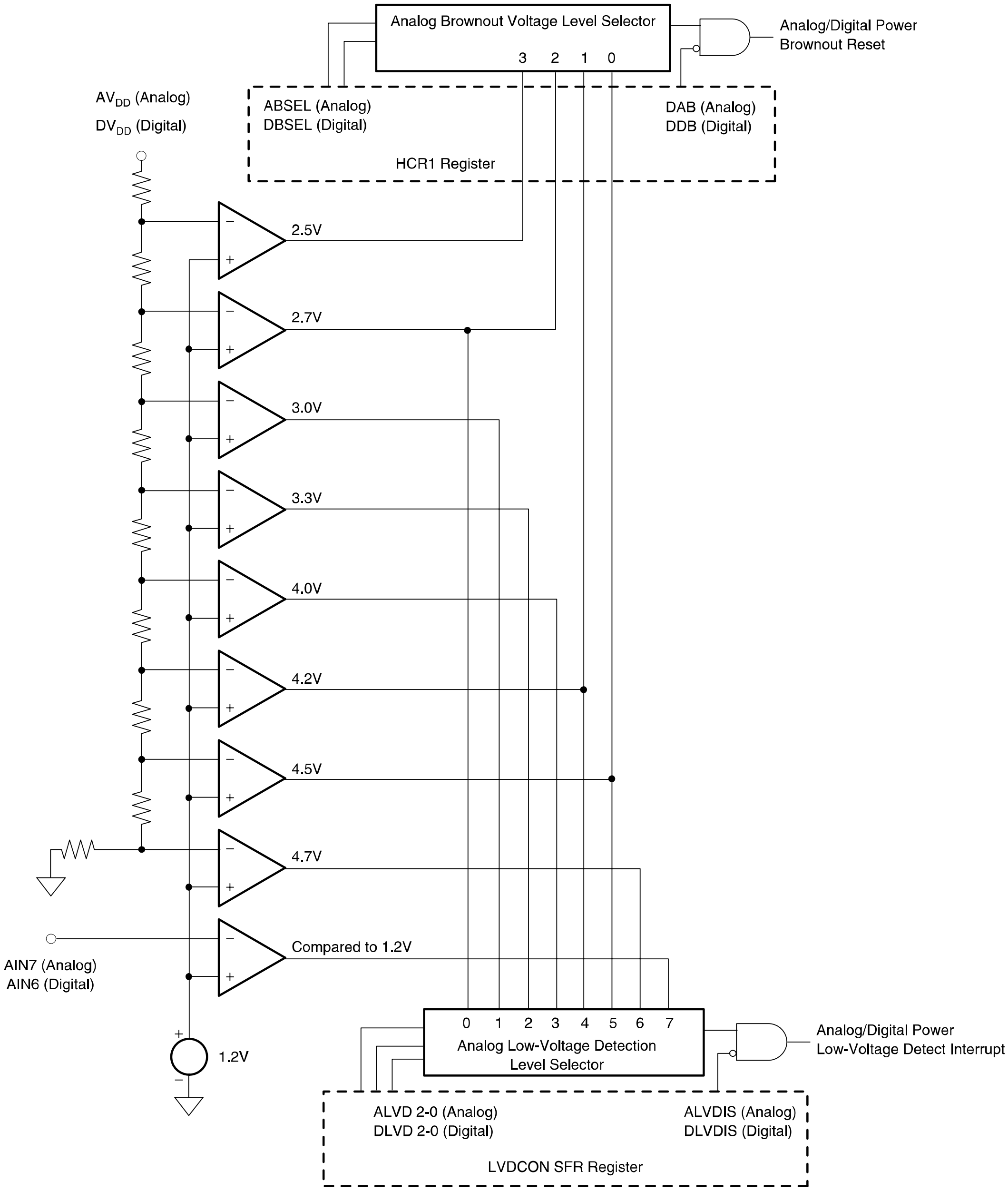
14.1 Description

The MSC1210 includes a number of special hardware features above and beyond those of a typical MCS–51 part.

14.2 Low-Voltage Detect

The MSC1210 includes low voltage and brownout detection circuits for both the analog and digital supply voltages. The voltage levels at which these circuits are tripped is programmable.

Figure 14–1. Brownout Reset and Low-Voltage Detection



The detect circuit must activate whenever the supply voltage drops below the programmed level. In order to account for temperature and process variations, the trip levels are typically higher than the specified value, to provide some margin. For example, when 4.5V is selected, the detect output will typically activate when the supply drops below 4.7V.

14.2.1 Power Supply

V_{SPD} powers the digital section resistor string and the comparators. V_{SPA} powers the analog section resistor string and the bandgap voltage. Level shifters, where needed, are included inside the block.

Table 14–1. Typical Sub-Circuit Current Consumption

Sub–Ckt Current Consumption	
Band Gap	20µA
Compartors	2µA
Resistor String	6µA
Total	40µA

Table 14–2. Comparator Specification

Comparator Parameters	
50mV ± 2mV	Hysteresis at 2.5V
100mV ± 8mV	Hysteresis at 4.7V
26mV	Hysteresis at Each Terminal
400nS	Response Time for Slow Input

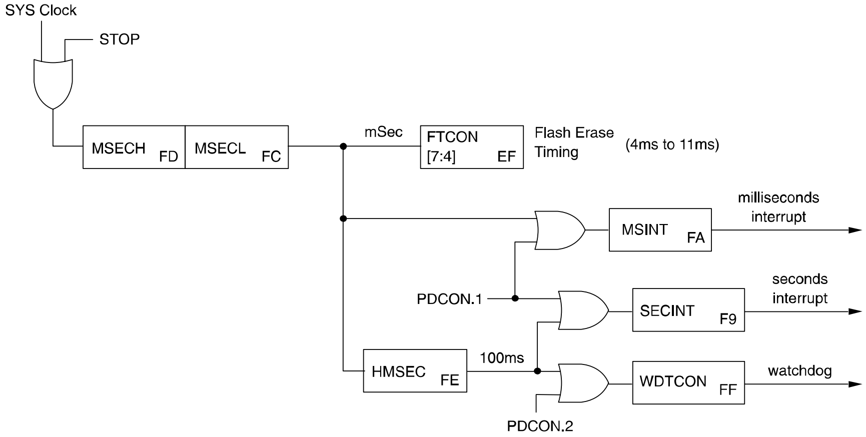
Table 14–3. Band Gap Parameters

Band Gap Parameters		
Bandgap Voltage Reference	(min)	1.00V
Bandgap Voltage Reference	(typ)	1.22V
Bandgap Voltage Reference	(max)	1.50V
Minimum Supply Voltage	(V_{SPA})	1.50V
Bandgap Startup Time	(typ)	< 16µS

14.3 Watchdog Timer

The watchdog timer is used to ensure that the CPU is executing the user program and not some random sequence of instructions provoked by a malfunction. When the watchdog timer is enabled, the user program must periodically notify the watchdog that the program is still running correctly. If the watchdog detects that the user program has not made this notification after a certain amount of time, the watchdog automatically resets the MCS1210 or executes an interrupt. This ensures that the part does not hang in an infinite loop or execute non-program code due to some malfunction or programming error.

Figure 14–2. System Timing Interrupt Control



14.3.1 Watchdog Timer Hardware Configuration

The watchdog is first configured when code is downloaded to the MSC1210. Bit 3 of hardware configuration register 0 (HCR0) is the Enable Watchdog Reset (EWDR) bit. If this bit is set, the watchdog will trigger a reset (if the watchdog is enabled by software and not reset at appropriate intervals), whereas if this bit is clear, the watchdog will trigger an interrupt (if the watchdog is enabled by software not reset at appropriate intervals). The point to remember is that the EWDR bit in the HCR0 register indicates what the watchdog will do when it is triggered: reset the MSC1210 or cause an interrupt. It does not, by itself, enable or disable the watchdog; that is done in software at execution time.

Note:

The HCR0 and HCR1 registers may be set by the TI downloader application at download time. It may also be set manually from within the source code by including the following assembly language code:

```
CSEG AT 0807EH

DB 0FCH ; Value for HCR0

DB 0FFH ; Value for HCR1
```

When the MSC1210 is in programming/download mode, code address $807E_H$ refers to the HCR0 register and $807F_H$ refers to the HCR1 register. This allows the values that are needed for HCR0/HCR1 to be hardcoded in the source code rather than having to set the registers manually via the downloader program.

14.3.2 Enabling Watchdog Timer

The watchdog timer is enabled by writing a 1 and then a 0 to the EWDT bit (WDTCON.7). This may be accomplished, for example, with the following code:

```
WDTCON = 0x80; // Set EWDT

WDTCON = 0x00; // Clear EWDT – Watchdog enabled
```

The watchdog timer then begins a countdown that, unless reset by your program, will trigger a watchdog reset or interrupt (depending on the configuration of HCR0, described previously). The time after which the watchdog will be triggered is also configured by the low five bits of the WDTCON SFR. These bits, which may represent a value from 1 to 32 (0 to 31, plus 1), multiplied by the time represented by HMSEC, defines the countdown time for the watchdog.

For example, if HMSEC is assigned a value that represents 100ms and WDTCON is assigned a value of 7, the watchdog will automatically trigger after 800ms ([7 + 1] • 100), unless the reset sequence is issued by the user program. Therefore, a better approach to enabling the watchdog timer is:

```
WDTCON = 0x80; // Set EWDT

WDTCON = 0x07; // Clear EWDT, set timeout = 7, 800ms
```

Note:

There is an uncertainty of one count in the watchdog counter. That is to say, the watchdog counter may occur a full HMSEC after the programmed time interval. In the previous example, where the watchdog is set to trigger after 800ms, the watchdog may in fact trigger as late as 900ms.

Although the watchdog timeout value (0x07 in the previous example) may be set at the same time as the EWDT bit is cleared, it may be changed after the fact. If the timeout value is changed after the watchdog has been enabled, the new timeout will take effect the next time the watchdog times out, or the next time the watchdog is reset (see next section). For example:

```
WDTCON = 0x80; // Set EWDT

WDTCON = 0x07; // Clear EWDT, set timeout = 7, 800ms

WDTCON = 0x06; // Set timeout = 6, 700ms
```

In this example, the watchdog will initially be enabled with a timeout of 800ms. The very next instruction sets the timeout to 700ms. In this case, the watchdog will time out after 800ms, unless it is reset as described in the following section. Once the watchdog has been reset, the new timeout of 700ms will take effect.

14.3.3 Resetting the Watchdog Timer

Your program, when operating properly, must reset the watchdog periodically. You can reset the watchdog as frequently or infrequently as desired, as long as it is reset more frequently than the watchdog countdown time described previously.

Your program must reset the watchdog by writing a 1 and then a 0 to the RWDT bit (WDTCON.5). This notifies the watchdog that your program is still operating correctly and that the watchdog timer should be reset.

The following code will reset the watchdog timer and notify the MSC1210 that your program is still executing correctly:

```
WDTCON |= 0x20; // Set RWDT, other bits unaffected

WDTCON &= ~0x20; // Clear RWDT—watchdog reset
```

Note:

It is generally a good idea to place the watchdog reset code in the main section of your program that is within a rapidly-executing control loop. It is not advisable to place the code within an interrupt, because the main program might be stuck in an infinite loop, although the interrupts can still trigger properly. Placing the watchdog reset code in an interrupt, in these cases, would tell the MSC1210 that the program is still executing correctly when, in fact, it is stuck in an infinite loop.

14.3.4 Disabling Watchdog Timer

Once the watchdog timer is activated, it operates continuously and your program must reset the watchdog timer regularly, as described in the previous section.

If, for some reason, you need to disable the watchdog timer (e.g., before entering idle mode), write a 1 and then a 0 to the DWDT (WDTCON.6) bit. In code, this can be accomplished with:

```
WDTCON |= 0x40; // Set DWDT, other bits unaffected

WDTCON &= ~0x40; // Clear RWDT—watchdog disabled
```

The watchdog is then disabled until it is subsequently re–enabled using the process in section 14.3.2.

14.3.5 Watchdog Timeout/Activation

If the watchdog is not reset by sending the reset sequence described previously before the watchdog counter expires, the watchdog will be activated. The watchdog will either reset the MSC1210 or trigger a watchdog interrupt, depending on the setting of the HCR0 hardware configuration register.

14.3.5.1 Watchdog Reset

In the case of a watchdog reset, the MSC1210 is reset. SFRs will assume their default values, the stack is reset, and the program starts executing again at address 0000_H. The contents of RAM is not affected.

14.3.5.2 Watchdog Interrupt

If the HCR0 register is configured to cause a watchdog interrupt, a watchdog auxiliary interrupt is flagged in the watchdog timer interrupt, WDTI (EICON.3). If the watchdog interrupt is enabled in EWDI (EIE.4) and interrupts are enabled via EA (IE.7), a watchdog interrupt is triggered and vectors to 0063_H. Your program must clear the WDTI flag before exiting the interrupt or the watchdog interrupt will be triggered again.

Note:

If the MSC1210 is in Idle mode when the watchdog interrupt is triggered, the processor will only wake up from idle mode if EWUWDT (EWU.2) is set. See section 10.9, *Waking Up from Idle Mode*, for additional details.

Chapter 15

Advanced Topics

Chapter 15 describes advanced topics associated with the MSC1210 ADC.

Topic **Page**

15.1 Hardware Configuration

In addition to whatever amount of flash memory the specific MSC1210 part contains (which may be partitioned between flash data memory and flash program memory), the MSC1210 also includes 128 bytes of hardware configuration memory. This memory is used to store two hardware configuration registers and, optionally, up to 110 bytes of configuration data that you may set at program time, and that may be used to store information such as serial numbers, product codes, etc.

Note:

Hardware configuration memory, including the hardware configuration registers and the 110 bytes of configuration data, can only be set at program time. They cannot be modified by your program at run time, once the firmware has been downloaded to the MSC1210.

15.1.1 Hardware Configuration Registers

The MSC1210 has two hardware configuration registers, HCR0 and HCR1. These registers are set at the moment the MSC1210 is programmed—be it in parallel or serial mode—and are used to set various operating parameters of the MSC1210.

When loading a program on the MSC1210, the HCR0 register is at code address 807E$_H$, whereas HCR1 is found at code address 807F$_H$. In a typical assembly language program, the HCR0 and HCR1 registers could be set by adding the following code to the program:

```
CSEG AT 0807EH  ;Address of HCR0

DB 0FCH   ;HCR0:76:DBLSEL 54:ABLSEL 3:DAB 2:DDB 1:EGP0
          ;0:EGP23

DB 0FFH   ;HCR1: 7:EPMA 6:PML 5:RSL 4:EBR 3:EWDR 210:DFSEL
```

15.1.1.1 Hardware Configuration Register 0 (HCR0)

Hardware configuration register 0 (HCR0) is used to configure the amount of flash memory partitioned as data flash memory, configure the watchdog, and set a number of security bits that restrict write access to flash memory.

The HCR0 has the following structure:

	bit 7	**bit 6**	**bit 5**	**bit 4**	**bit 3**	**bit 2**	**bit 1**	**bit 0**
CADDR $7F_H$	EPMA	PML	RSL	EBR	EWDR	DFSEL2	DFSEL1	DSEL0

EPMA (bit 7)—**Enable Program Memory Access** (Security Bit). When this bit is clear, flash memory cannot be read or written after the part is programmed. This will prevent future updates to the firmware code. When the bit is set, which is the default condition, flash memory will remain fully accessible for reprogramming.

PML (bit 6)—**Program Memory Lock.** When clear, your program may write to flash program memory. When set, flash program memory is locked and cannot be changed by your program. This may be set to ensure that the user program does not overwrite the program itself by writing to flashmemory.

RSL (bit 5)—**Reset Sector Lock.** When clear, your program may write to the reset sector (the first 4k of flash program memory). When it is set (default), your program may not write to this area of flash memory. This bit functions the same as the PML bit, but applies to only the first 4k of flash program memory. If the MSC1210 is configured such that only 4k is assigned to flash program memory, this bit has the same effect as setting PML.

EBR (bit 4)—**Enable Boot ROM.**

EWDR (bit 3)—**Enable Watchdog Reset.** When this bit is clear, a watchdog situation provokes a watchdog auxiliary interrupt that your program needs to intercept and handle. If this bit is set, a watchdog situation provokes a reset of the MSC1210.

DFSEL2/DFSEL1/DFSEL0 (bits 2-0)—**Flash Data Memory Size.** These three bits, together, select how much of the available flash memory will be assigned to data memory; the rest will be assigned to flash program memory.

DFSEL2/1/0	Amount of Flash Data Memory
001	32k
010	16k
011	8k
100	4k
101	2k
110	1k
111	No flash memory (default)

Note:

If more flash data memory is selected than flash memory exists on the actual part, all of the flash memory available will be partitioned as flash data memory, leaving nothing for flash program memory.

15.1.1.2 Hardware Configuration Register 1 (HCR1)

Hardware configuration register 1 (HCR1) is used primarily to configure the brownout detection for both the digital and analog power supplies. It is also used to configure whether ports 0, 2, and 3 are used as general I/O ports, or take part in external memory access.

The HCR1 has the following structure:

	bit 7	**bit 6**	**bit 5**	**bit 4**	**bit 3**	**bit 2**	**bit 1**	**bit 0**
CADDR $7E_H$	DBLSEL1	DBLSEL0	ABLSEL1	ABLSEL0	DAB	DDB	EGP0	EGP23

DBLSEL1/0 (bits 7-6)**—Digital Brownout Level Select.** These two bits, together, select the voltage level that triggers a digital brownout situation.

00: 4.5V

01: 4.2V

10: 2.7V

11: 2.5V (default)

ABLSEL1/0 (bits 5-4)—**Analog Brownout Level Select.** These two bits, together, select the voltage level that triggers an analog brownout situation.

00: 4.5V
01: 4.2V
10: 2.7V
11: 2.5V (default)

DAB (bit 3)—**Disable Analog Power–Supply Brownout Detection.** When this bit is set, brownout detection on the analog power supply is disabled. When clear, brownout detection operates normally.

DDB (bit 2)—**Disable Digital Power–Supply Brownout Detection.** When this bit is set, brownout detection on the digital power supply is disabled. When clear, brownout detection operates normally.

EGP0 (bit 1)—**Enable General-Purpose I/O for Port 0.** When this bit is set (default), port 0 is used as a general I/O port. When clear, P0 is used to access external memory—in this mode, P3.6 and P3.7 are used to control the $\overline{\text{WR}}$ and $\overline{\text{RD}}$ lines.

EGP23 (bit 0)—**Enable General-Purpose I/O for Port 2 and 3.** When this bit is set (default), ports 2 and 3 sre used as general I/O ports. When clear, P2 and P3 are used to access external memory—in this mode, P3.6 and P3.7 are used to control the $\overline{\text{WR}}$ and $\overline{\text{RD}}$ lines.

15.1.2 Hardware Configuration Memory

In addition to the hardware configuration registers, 116 bytes of configuration memory are available to you for your own use. This configuration memory, also set during device programming, can hold information such as unique serial numbers, parameters, or any other information that you want to record. The configuration information is available to the program when the part is operating for read operations, but cannot be changed.

Setting the configuration memory is accomplished in the same way as setting the hardware configuration registers as described above—the only difference is that the configuration memory available is located from 8002_H through $806F_H$. Thus, the first five bytes of your configuration memory could be set with assembly code such as the following:

```
CSEG AT 8002H            ;Address of user configuration memory

DB 10h,20h,30h,40h,50h    ;User configuration data, up to 116 bytes
```

Be careful that the configuration data is located at 8002_H and includes no more than 116 bytes of data. Including more than 116 bytes of configuration data causes the data to spill over into the area of configuration memory that is used to configure the actual MSC1210 hardware.

15.1.3 Accessing Configuration Memory in a User Program

The 128 bytes of flash configuration memory, which include the 116 bytes of user-defined configuration data and two bytes of hardware configuration registers, can be read by your program in normal operation. However, the configuration data is not obtained by reading the code address to which they were programmed. That is to say, although flash configuration memory is set at program time by placing it at code memory addresses 8002_H through $806F_H$ (user configuration memory) and $807E_H$ and $807F_H$ (hardware configuration), the data cannot be read by reading the data from that program memory address. Rather, two SFRs are used to read the configuration memory.

In code, your program may set the configuration address register SFR, CADDR (93_H), to the address of the byte of configuration memory that should be read. The address must be a value between 00_H and $7F_H$ reflecting the 128 bytes of configuration flash memory. Once the address is set in CADDR, the value of that address can then be read by reading the CDATA (94_H) SFR.

Note:

You may not write to the CADDR SFR if the code is executing from flash memory. This is because that would imply that the MSC1210 fetch the flash configuration memory at the same time as it is fetching instructions from flash memory.

To read flash configuration memory, a call must be made to the 2k Boot ROM that is included on the MSC1210. A call to the faddr_data_read function, passing it the address as a parameter, will return the value of the configuration memory address.

15.2 Advanced Flash Memory

Flash memory may be configured as data memory, program memory, or both.

15.2.1 Write Protecting Flash Program Memory

Flash program memory may be protected against your program overwriting it by writing to flash memory during program execution. This provides a safeguard to the integrity of the code against intentional or accidental manipulation by your program.

By setting the Program Memory Lock (PML) bit in HCR0, all of flash program memory is write-protected inasmuch as your program modifying flash program memory is concerned. When this bit is set, your program is not able to write to any area of flash memory that has been partitioned as flash program memory.

Likewise, by setting the Reset Sector Lock (RSL) bit in HCR, the first 4k of flash program memory will be write-protected inasmuch as your program modifying that area of flash program memory is concerned. This is functionality identical to the PML bit, but the RSL bit only applies to the first 4k of flash program memory, whereas the PML bit applies to all of flash program memory. By clearing PML and setting RSL, the first 4k of flash program memory is locked against all writes by your program, but the rest of flash program memory is accessible to memory writes.

If writing to flash program memory is permitted by the PML and RSL bits, your program must first set the MXWS bit of MWS ($8F_H$) prior to writing to flash program memory. If this bit is not set, writes to flash program memory are not effective.

15.2.2 Updating Interrupts with Reset Sector Lock

If the Reset Sector Lock (RSL) bit in HCR0 has been set, the user program will not be able to modify the contents of the first 4k of flash program memory. Setting RSL makes it impossible to change where the ISRs will branch to when triggered because the interrupt service routine vectors are all located in the first bytes of flash program memory.

If RSL needs to be set, but the interrupt service routines also need to be able to change , it is recommended that the ISRs in the reset sector simply branch to the same address, plus 4k. At the resulting branch address, the the user will be able to jump to wherever the ISR is actually located, and will also be able to modify that jump because it is not contained in the reset sector.

For example, given an external 0 interrupt that branches to 0003_H when triggered, the following code could be implemented:

```
CSEG AT 0003h    ;Address of External 0 Interrupt

LJMP Ext0ISR     ;Jump to the interrupt vector at 0003h + 4k

CSEG AT 1003h    ;Ext0 jump vector, outside of reset sector

LJMP Ext0Code    ;Jump to wherever the real Ext. 0 interrupt
                 ;code is
```

Thus, If the external 0 interrupt code needs to be changed later, simply change the instruction at 1003_H to jump to the new code. Both the code and the jump at 1003_H can be updated as desired because both are outside of the reset sector.

15.3 Breakpoint Generator

The purpose of the breakpoint block is to generate an interrupt whenever the desired program or data memory address is accessed. There are two kinds of memory accesses it can detect:

- ☐ Accesses to program memory (read or write)
- ☐ Accesses to data memory (read or write)

The interrupt is handled by the interrupt controller (for details, see Chapter 10, *Interrupts*). Breakpoints are useful in debugging code. You can set a breakpoint at the start of a suspect piece of code. Once the program reaches the breakpoint address, program flow can be suspended/interrupted so you can force a memory dump or a register dump. You can specify up to two 16-bit addresses for which the interrupt may be generated.

15.3.1 Configuring Breakpoints

Breakpoints are controlled by the BPCON ($A9_H$), BPL (AAh), BPH (AB_H) and MCON (95_H) SFRs. The Breakpoint Control SFR (BPCON) controls the configuration of the breakpoint. BPL and BPH together form a 16-bit breakpoint address. BPSEL (MCON.7) selects which of the two breakpoints is to be configured.

The BPCON SFR has the following structure:

	7	6	5	4	3	2	1	0	Reset Value
SFR $A9_H$	BP	0	0	0	0	0	PMSEL	EBP	00_H

BP (bit 7)—**Breakpoint Interrupt.** This bit indicates that a break condition has been recognized by a hardware breakpoint register(s).

READ: Status of breakpoint interrupt. Indicates a breakpoint match for any of the breakpoint registers.

WRITE: 0—No effect.

1—Clear Breakpoint 1 for breakpoint register selected by MCON (SFR 95_H).

PMSEL (bit 1)—**Program Memory Select.** Write this bit to select memory for the address breakpoints of the register selected in MCON (SFR 95_H).
0: Break on address in data memory.
1: Break on address in program memory.

EBP (bit 1)—**Enable Breakpoint.** This bit enables this breakpoint register. Address of breakpoint register selected by MCON (SFR 95_H).
0: Breakpoint disabled.
1: Breakpoint enabled.

To configure a breakpoint, the following steps should be taken:

1) The BPSEL (MCON.7) bit must be set to either 0 (for Breakpoint 0) or 1 (for Breakpoint 1).
2) The Program Memory Select bit, PMSEL (BPCON.1), must be either cleared if the breakpoint is to detect an access to data memory, or set if the breakpoint is to detect an access to program memory.
3) BPL and BPH should be loaded with the low and high byte, respectively, of the address at which the breakpoint should be triggered.
4) The Enable Breakpoint bit, EBP (BPCON.0), must then be set to activate the interrupt.

15.3.2 Breakpoint Auxiliary Interrupt

Once a breakpoint interrupt has been configured, the BP (BPCON.7) interrupt flag will be set and, if enabled and not masked, a breakpoint auxiliary interrupt will be triggered whenever the specified memory address is accessed. The program must write a 1 back to BPCON.7 in order to clear the interrupt after processing it.

When a breakpoint interrupt occurs, the program may read the BPSEL (MCON.7) bit to determine which breakpoint was triggered. If BPSEL is clear, Breakpoint 0 triggered the interrupt. If BPSEL is set, Breakpoint 1 triggered the interrupt.

When using breakpoints, notice that the actual breakpoint occurs after the selected address. That is because of interrupt latency on the MSC1210. It takes a few cycles for the interrupt to be recognized and serviced. During that time the processor continues for two or three more instructions, which means that the program counter will be offset from the address in the breakpoint.

Additionally, when placing a breakpoint after a jump or return instruction, the breakpoint may be triggered even though the instruction was never executed. This is because the processor pre-fetches the instructions. The breakpoint hardware cannot distinguish between pre-fetched operations or those being executed. This usually means that breakpoints should not be placed on the first instruction of a routine, because just before that instruction is the jump or return instruction from a previous routine. A workaround is to place two NOPs at the beginning of the routine and then break after those NOPs.

15.3.3 Disabling a Breakpoint

To clear a previously set breakpoint, the following steps should be taken:

1) The BPSEL (MCON.7) bit must be set to either 0 (for breakpoint 0) or 1 (for breakpoint 1).
2) The Enable Breakpoint bit, EBP (BPCON.0), must be cleared to deactivate the interrupt.

15.4 Power Optimization

The MSC1210, like a standard 8052, has the ability to operate in a power-saving mode, known as idle mode. As the name implies, idle mode shuts down most of the energy-consuming functions of the microcontroller and idles. Code execution stops in idle mode, and the only way to exit idle mode is a system reset or an enabled interrupt being triggered.

Idle mode is useful in causing the microcontroller to go to sleep until an interrupt awakens it. Instead of cycling repeatedly waiting for an interrupt condition to occur, the part may be made to go to sleep until the condition is triggered—during that time, power consumption is minimized. External interrupts, the watchdog interrupt, or the auxiliary interrupts can be made to wake up an idling MSC1210.

To enter idle mode, bit 0 of PCON must be set. This can be accomplished with the instruction:

```
PCON |= 0x01;
```

When this instruction is executed, the MSC1210 immediately drops into idle mode and remains there until an enabled interrupt occurs. When an interrupt occurs, the ISR executes and finishes, and program execution continues with the instruction following the instruction that put the MSC1210 in idle mode—in this case, the instruction mentioned previously.

15.5 Flash Memory as Data Memory

If so configured in HCR0, some portion of flash memory can be accessed by your application program as flash data memory. The amount of flash memory that is partitioned as flash data memory is controlled by the low 3 bits of HCR0. Please see Section 15.1.1.1, *Hardware Configuration Register 0*, for details.

When some amount of flash memory is partitioned as flash data memory, the program may read, update, and store information in nonvolatile memory that will survive power-off situations. That makes the flash data memory a useful area to store configuration or data logging information.

The following program illustrates how flash data memory may be read and updated.

Note:

The MSEC and USEC SFRs must be correctly set prior to erasing or writing to flash memory.

Within the infinite `while()` loop, the program first reads flash data memory. This is accomplished directly by reading XRAM memory. This is accomplished in this C program by using the pFlashPage pointer—it is accomplished in assembly language using the MOVX instruction.

After reading flash memory, it increments the value of the first byte of the memory block read by one. The call to page_erase()is a call to the routine in boot ROM that erases the requested block of memory. Thereafter, it makes repeated calls to write_flash_chk to write the buffer back to the block of flash data memory one byte at a time.

The infinite loop continues by displaying the result of the writes to flash data memory (0 = Success) and the updated value contained in the buffer as read from flash data memory via the pFlashPage pointer. It then prompts the user to hit any key, after which the loop will repeat itself and the first byte of the buffer will again be incremented.

```
#include <stdio.h>
#include <reg1210.h>
#include ”rom1210.h”
// define the page we want to modify
#define PAGE_START 0x0400
#define PAGE_SIZE  0x80
// define a pointer to this page
char xdata * pFlashPage;
// define a RAM area as a buffer to hold one page
char xdata Buffer[PAGE_SIZE];
int main()
{
   char Result;
   unsigned char i;
   // synchronize baud rate
   autobaud();
   // Set the pointer to the beginning of the page to modify
   pFlashPage = (char xdata * ) PAGE_START;
   // before writing the flash, we have to initialize
   // the usec and msec SFRs because the flash programming
   // routines rely on these SFRs
   USEC  = 12–1; // assume a 12 MHz clock
   MSEC = 12000–1;
   while(1)
      {
      // copy the page from FLASH to RAM
      for(i=0;i<PAGE_SIZE;i++)
         Buffer[i] = *pFlashPage++;
      // increment the counter
      Buffer[0] += 1;
      // now erase the page
      page_erase(PAGE_START, 0xff, DATA_FLASH);
      Result = 0;
      // and write the modified contents back into flash
      for(i=0;i<PAGE_SIZE;i++)
         Result |= write_flash_chk (PAGE_START+i, Buffer[i],
                                    DATA_FLASH);
      // re–read the counter
      pFlashPage = (char xdata *) PAGE_START;
      printf(”flash write returned %d, Reset counter is now %d,
            press any key\n”, (int) Result, (int)(*pFlashPage));
      while(RI==0);
      RI = 0;
      }
}
```

Note:

Your program must use the boot ROM routines, such as write_flash_chk, in order to modify flash data memory, if your program is itself executing from flash memory. That is because the instructions are being fetched from flash memory, and writing to flash memory simultaneously causes a conflict that results in undesired program execution. The boot ROM routines must be used to modify flash memory whenever your program itself resides on-chip in flash memory.

15.6 Advanced Topics and Other Information

15.6.1 Serial and Parallel Programming of the MSC1210

The MSC1210 flash program memory may be updated either in a serial or parallel fashion. In these cases, the process is controlled by protocol that allows the PC (or other external device) and the MSC1210 to communicate. This protocol is described in http://www–s.ti.com/sc/psheets/sbaa076a/sbaa076a.pdf.

15.6.2 Debugging Using the MSC1210 Boot ROM Routines

The MSC1210 boot ROM, in addition to facilitating the update of flash memory, can also be used to control a debugging session. This is described in http://www–s.ti.com/sc/psheets/sbaa079/sbaa079.pdf.

15.6.3 Using MSC1210 with Raisonance Development Tools

In addition to the Keil toolset, which is included with he MSC1210 EVM kit, Raisonance provides a development toolset that may be used to develop software for the MSC1210. Further details on using the Raisonance tools with the MSC1210 are provided at http://www–s.ti.com/sc/psheets/sbaa080/sbaa080.pdf.

15.6.4 Using the MSC1210 Evaluation Module (EVM)

The MSC1210 EVM is a complete evaluation module that provides significant flexibility in testing and using the features of the MSC1210. Details of using the EVM may be found at http://www–s.ti.com/sc/psheets/sbau073/sbau073.pdf.

Chapter 16

8052 Assembly Language

Chapter 16 describes the 8052 Assembly Language.

Topic **Page**

16.1 Description

Assembly language is a low-level, pseudo-English representation of the microcontroller’s machine language. Each assembly language instruction has a one-to-one relation to one of the microcontroller machine-level instructions.

High-level languages, such as C, Basic, Visual Basic, etc. are one or more steps above assembly language, in that no significant knowledge of the underlying architecture is necessary. It is possible (and common) for a developer to program a Visual Basic application in Windows without knowing much of anything about the Windows API, much less the underlying architecture of the Intel Pentium. Furthermore, a developer who has written code in C for Unix will not have significant problems adapting to writing code in C for Windows, or a microcontroller such as an 8052; although there are some variations, the C compiler itself takes care of most of the processor-specific issues.

Assembly language, on the other hand, is very processor specific. While a prior knowledge of assembly language with any given processor will be helpful when attempting to begin coding in the assembly language of another processor, the two assembly languages may be extremely different. Different architectures have different instruction sets, different forms of addressing. In fact, only general concepts may work from one processor to another.

The low-level nature of assembly language programming requires an understanding of the underlying architecture of the processor for which one is developing. This is why we explained the 8052 architecture fully before attempting to introduce the reader to assembly language programming in this document. Many aspects of assembly language may be completely confusing without a prior knowledge of the architecture.

This section of the document will introduce the reader to 8052 assembly language, concepts, and programming style.

16.2 Syntax

Each line of an assembly language program consists of the following syntax, each field of which is optional. However, when used, the elements of the line must appear in the following order:

1) **Label**—a user-assigned symbol that defines the address of this instruction in memory. The label, if present, must be terminated with a colon.

2) **Instruction**—an assembly language instruction that, when assembled, will perform some specific function when executed by the microcontroller. The instruction is a psuedo-English mnemonic which relates directly to one machine language instruction.

3) **Comment**—the developer may include a comment on each line for inline documentation. These comments are ignored by the assembler but may make it easier to subsequently understand the code. A comment, if used, must be preceded with a semicolon.

In summary, a typical 8052 assembly language line might appear as:

```
MYLABEL: MOV A,#25h ;This is just a sample comment
```

In this line, the label is MYLABEL. This means that if subsequent instructions in the program need to make reference to this instruction, they may do so by referring to MYLABEL, rather than the memory address of the instruction.

The 8052 assembly language instruction in this line is MOV A,#25h. This is the actual instruction that the assembler will analyze and assemble into the two bytes 74_H 25_H. The first number, 74_H, is the 8052 machine language instruction (opcode) “MOV A,#*dataValue*”, which means “move the value *dataValue* into the accumulator.” In this case, the value of *dataValue* will be the value of the byte that immediately follows the opcode. We want to load the accumulator with the value 25_H and the byte following the opcode is 25_H. As you can see, there is a one-to-one relationship between the assembly language instruction and the machine language code that is generated by the assembler.

Finally, the instruction above includes the optional comment “;This is just a sample comment”. The comment must always start with a semicolon. The semicolon tells the assembler that the rest of the line is a comment that should be ignored by the assembler.

All fields are optional and the following are also alternatives to the above syntax:

Label only:	`LABEL:`
Label and instruction:	`LABEL: MOV A,#25h`
Instruction and comment:	`MOV A,#25h ;This is just a comment`
Label and comment:	`LABEL: ;This is just a comment`
Comment only:	`;This is just a sample comment`

All of the above permutations are completely valid. It is up to you as to which components of the assembly language syntax are used. However, when used, they must follow the above syntax and be in the correct order.

Note:

It does not matter what column each field begins in. That is, a label can start at the beginning of the line or after any number of blank spaces. Likewise, an instruction may start in any column of the line, as long as it follows any label that is also on that line.

16.3 Number Bases

Most assemblers are capable of accepting numeric data in a variety of number bases. Commonly supported are decimal, hexadecimal, binary, and octal.

Decimal: To express a decimal number in assembly language, simply enter the number normally.

Hexadecimal: To express a hexadecimal number, enter the number as a hexadecimal value, and terminate the number with the suffix “h”. For example, the hexadecimal number 45 is expressed as 45h. Furthermore, if the hexadecimal number begins with an alphabetic character (A, B, C, D, E, or F), the number must be preceded with a leading zero. For example, the hex number E4 is written as 0E4h. The leading zero allows the assembler to differentiate the hex number from a symbol because a symbol never starts with a number.

Binary: To express a binary number, enter the binary number followed by a trailing B, to indicate binary. For example, the binary number 100010 is expressed as 100010B.

Octal: To express an octal number, enter the octal number itself followed by a trailing Q, to indicate octal. For example, the octal number 177 is expressed as 177Q.

As an example, all of the following instructions load the accumulator with 30 (decimal):

```
MOV A,#30

MOV A,#11110B

MOV A,#1EH

MOV A,#36Q
```

16.4 Expressions

You may use mathematical expressions in your assembly language instructions anywhere a numeric value may be used. For example, both of the following are valid assembly language instructions:

```
MOV A,#20h + 34h      ;Equivalent to #54h

MOV 35h + 2h,#10101B  ;Equivalent to MOV 37h,#10101B
```

16.5 Operator Precedence

Mathematical operators within an expression are subject to the following order of precedence. Operators at the same "level" are evaluated left to right.

Table 16–1. Order of Precedence for Mathematical Operators

Order	Operator
1 (Highest)	( )
2	HIGH LOW
3	* / MOD SHL SHR
4	EQ NE LT LE GT GE = <> < <= > >=
5	NOT
6	AND
7 (Lowest)	OR XOR

Note:

If you have any doubts about operator precedence, it is useful to use parentheses to force the order of evaluation that you have contemplated. It is often easier to read mathematical expressions when parentheses have been added, although the parentheses are not technically necessary.

16.6 Characters and Character Strings

Characters and character strings are enclosed in single quotes and are converted to their numeric equivalent at assemble time. For example, the following two instructions are the same:

```
MOV A,#’C’
```

```
MOV A,#43H
```

The two instructions are the same because the assembler will see the ‘C’ sequence, convert the character contained in quotes to its ASCII equivalent (43_H), and use that value. Thus, the second instruction is the same as the first.

Strings of characters are sometimes enclosed in single quotes and sometimes enclosed in double quotes. For example, Pinnacle 52 uses double quotes to indicate a string of characters and a single quote to indicate a single character. Thus:

```
MOV A,#’C’        ;Single character – ok
MOV A,#”STRING”   ;String – ERROR! Can’t load a string into the
                  ;accumulator
```

Strings are invalid in the above context, although there are other special assembler directives that do allow strings. Be sure to check the manual for your assembler to determine whether character strings should be placed within single quotes or double quotes.

16.7 Changing Program Flow (LJMP, SJMP, AJMP)

LJMP, SJMP and AJMP are used as a go to in assembly language. They cause program execution to continue at the address or label they specify. For example:

```
LJMP LABEL3  ;Program execution is transferred to LABEL3

LJMP 2400h   ;Program execution is transferred to address 2400h

SJMP LABEL4  ;Program execution is transferred to LABEL4

AJMP LABEL7  ;Program execution is transferred to LABEL7
```

The differences between LJMP, SJMP, and AJMP are:

- ❑ LJMP requires 3 bytes of program memory and can jump to any address in the program.
- ❑ SJMP requires 2 bytes of program memory, but can only jump to an address within 128 bytes of itself.
- ❑ AJMP requires 2 bytes of program memory, but can only jump to an address in the same 2k block of memory.

These instructions perform the same task, but differ in what addresses they can jump to, and how many bytes of program memory they require.

LJMP always works. You can always use LJMP to jump to any address in your program.

SJMP requires two bytes of memory, but has the restriction that it can only jump to an instruction or label within 128 bytes before or 127 bytes after the instruction. This is useful if you are branching to an address that is very close to the jump itself. You save 1 byte of memory by using SJMP instead of AJMP.

AJMP also requires two bytes of memory, but has the restriction that it can only jump to an instruction or label that is in the same 2k block of program memory. For example, if the AJMP instruction is at address 0200_H, it can only jump to addresses between 0000_H and $07FF_H$—It can not jump to 800_H.

Note:

Some optimizing assemblers allow you to use JMP in your code. While there is no JMP instruction in the 8052 instruction set, the optimizing assembler will automatically replace your JMP with the most memory-efficient instruction. That is, it will try to use SJMP or AJMP if it is possible, but will resort to LJMP if necessary. This allows you to simply use the JMP instruction and let the assembler worry about saving program memory, whenever possible.

16.8 Subroutines (LCALL, ACALL, RET)

As in other languages, 8052 assembly language permits the use of subroutines. A subroutine is a section of code that is called by a program, does a task, and then returns to the instruction immediately following that of the instruction that made the call.

LCALL and ACALL are both used to call a subroutine. LCALL requires three bytes of program memory and can call any subroutine anywhere in memory. ACALL requires two bytes of program memory and can only call a subroutine within the same 2k block of program memory.

Both call instructions will save the current address on the stack and jump to the specified address or label. The subroutine at that address will perform whatever task it needs to and then return to the original instruction by executing the RET instruction.

For example, consider the following code:

```
LCALL SUBROUTINE1   ;Call the SUBROUTINE1 subroutine
LCALL SUBROUTINE2   ;Call the SUBROUTINE2 subroutine
.
.
.
SUBROUTINE1: {subroutine code}   ;Insert subroutine code here
             RET                 ;Return from subroutine
SUBROUTINE2: {subroutine code}   ;Insert subroutine code here
             RET                 ;Return from subroutine
```

The code starts by calling SUBROUTINE1. Execution transfers to SUBROUTINE1 and executes whatever code is found there. When the MCU hits the RET instruction, it automatically returns to the next instruction, which is LCALL SUBROUTINE2. SUBROUTINE2 is then called, executes its code, and returns to the main program when it reaches the RET instruction.

Note:

It is very important that all subroutines end with the RET instruction, and that all subroutines exit themselves by executing the RET instruction. Unpredictable results will occur if a subroutine is called with LCALL or ACALL and a corresponding RET is not executed.

Note:

Subroutines may call other subroutines. For example, in the code above SUBROUTINE1 could include an instruction that calls SUBROUTINE2. SUBROUTINE2 would then execute and return to SUBROUTINE1, which would then return to the instruction that called it. However, keep in mind that every LCALL or ACALL executed expands the stack by two bytes. If the stack starts at internal RAM address 30_H and 10 successive calls to subroutines are made from within subroutines, the stack will expand by 20 bytes to 44_H.

Note:

Recursive subroutines (subroutines that call themselves) are a very popular method of solving some common programming problems. However, unless you know for certain that the subroutine will call itself a certain number of times, it is generally not possible to use subroutine recursion in 8052 assembly language. Due to the small amount of Internal RAM a recursive subroutine could quickly cause the stack to fill all of internal RAM.

16.9 Register Assignment (MOV)

One of the most commonly used 8052 assembly language instructions, and the first to be introduced here, is the MOV instruction. 57 of the 254 opcodes are MOV instructions because there are many ways data can be moved between various registers using various addressing modes.

The MOV instruction is used to move data from one register to another—or to simply assign a value to a register—and has the following general syntax:

MOV *DestinationRegister*,*SourceValue*

DestinationRegister always indicates the register or address in which *SourceValue* will be stored, whereas *SourceValue* indicates the register the value will be taken from, or the value itself if it is preceded by a pound sign (#).

For example:

```
MOV A,25h     ;Moves contents of Internal RAM address 25h
              ;to accumulator

MOV 25h,A     ;Move contents of accumulator into Internal
              ;RAM address 25h

MOV 80h,A     ;Move the contents of the accumulator to P0
              ;SFR (80h)

MOV A,#25h    ;Moves the value 25h into the accumulator
```

As shown, the first parameter is the register, internal RAM address, or SFR address that a value is being moved to. Another way of looking at it is that the first parameter is the register that is going to be assigned a new value.

Likewise, the second parameter tells the 8052 where to get the new value. Normally, the value of the second parameter indicates the Internal RAM or SFR address from which the value should be obtained. However, if the second parameter is preceded by a pound sign, the register will be assigned the value of the number that follows the pound sign (as is demonstrated in the previous example).

As already mentioned, the MOV instruction is one of the most common and vital instructions that an 8052 assembly language programmer uses. The prospective assembly language programmer must fully master the MOV instruction. This may seem simple, but it requires knowing all of the permutations of the MOV instruction and knowing when to use them. This knowledge comes with time and experience, and by reviewing Appendix A, *8052 Instruction Set Overview*.

It is important that all types of MOV instructions be understood so that the programmer knows what types of MOV instructions are available, as well as what kinds of MOV instructions are not available.

Careful inspection of the MOV commands in the instruction set reference will reveal that there is no MOV from R register to R register instruction. That is to say, the following instruction is invalid:

```
MOV R2,R1 ;INVALID!!
```

This is a logical type of operation for a programmer to implement, but the instruction is invalid. Instead, it must be programmed as:

```
MOV A,R1  ;Move R1 to accumulator
MOV R2,A  ;Move accumulator to R2
```

Another combination that is not supported is MOV indirectly from Internal RAM to another Indirect RAM address. Again, the following instruction is invalid:

```
MOV @R0,@R1  ;INVALID!!
```

This is not a valid MOV combination. Instead, it could be programmed as:

```
MOV A,@R1 ;Move contents of IRAM pointed to by R1 to accumulator
MOV @R0,A ;Move accumulator to Internal RAM address pointed to by R0
```

Also note that only R0 and R1 can be used for Indirect Addressing.

Note:

When you need to execute a type of MOV instruction that does not exist, it is generally helpful to use the accumulator. If a given MOV instruction does not exist, it can usually be accomplished by using two MOV instructions that both use the accumulator as a transfer or temporary register

With this knowledge of the MOV instruction, some simple memory assignment tasks can be performed:

1) Clear the contents of Internal RAM address FF_H:

```
MOV A,#00h    ;Move the value 00h to the accumulator
              ;(accumulator=00h)

MOV R0,#0FFh ;Move the value FFh to R0 (R0=0FFh)

MOV @R0,A     ;Move accumulator to @R0, thus clearing
              ;contents of FFh
```

2) Clear the contents of Internal RAM address FF_H (more efficient):

```
MOV R0,#0FFh  ;Move the value FFh to R0 (R0=0FFh)

MOV @R0,#00h  ;Move 00h to @R0 (FFh), clearing contents of FFh
```

3) Clear the contents of all bit memory, internal RAM addresses 20_H through $2F_H$ (this example will later be improved upon to require less code):

```
MOV 20h,#00h ;Clear Internal RAM address 20h

MOV 21h,#00h ;Clear Internal RAM address 20h

MOV 22h,#00h ;Clear Internal RAM address 20h

MOV 23h,#00h ;Clear Internal RAM address 20h

MOV 24h,#00h ;Clear Internal RAM address 20h

MOV 25h,#00h ;Clear Internal RAM address 20h

MOV 26h,#00h ;Clear Internal RAM address 20h

MOV 27h,#00h ;Clear Internal RAM address 20h

MOV 28h,#00h ;Clear Internal RAM address 20h

MOV 29h,#00h ;Clear Internal RAM address 20h

MOV 2Ah,#00h ;Clear Internal RAM address 20h

MOV 2Bh,#00h ;Clear Internal RAM address 20h

MOV 2Ch,#00h ;Clear Internal RAM address 20h

MOV 2Dh,#00h ;Clear Internal RAM address 20h

MOV 2Eh,#00h ;Clear Internal RAM address 20h

MOV 2Fh,#00h ;Clear Internal RAM address 20h
```

16.10 Incrementing and Decrementing Registers (INC, DEC)

Two instructions, INC and DEC, can be used to increment or decrement the value of a register, internal RAM, or SFR by 1. These instructions are rather self-explanatory.

The INC instruction will add 1 to the current value of the specified register. If the current value is 255, it will overflow back to 0. For example, if the accumulator holds the value 240 and the INC A instruction is executed, the accumulator is incremented to 241.

```
INC A    ;Increment the accumulator by 1

INC R1   ;Increment R1 by 1

INC 40h  ;Increment Internal RAM address 40h by 1
```

The DEC instruction will subtract 1 from the current value of the specified register. If the current value is 0, it will underflow back to 255. For example, if the accumulator holds the value 240 and the DEC A instruction is executed, the accumulator will be decremented to 239.

```
DEC A    ;Decrement the accumulator by 1

DEC R1   ;Decrement R1 by 1

DEC 40h  ;Decrement Internal RAM address 40h by 1
```

Note:

Under some assembly language architectures, the INC and DEC instructions set an overflow or underflow flag when the register overflows from 255 to 0 or underflows from 0 back to 255, respectively. This is *not* the case with the INC and DEC instructions in 8052 assembly language. Neither of these instructions affects any flags whatsoever.

16.11 Program Loops (DJNZ)

Many operations are conducted within finite loops. That is, a given code segment is executed repeatedly until a given condition is met.

A common type of loop is a simple counter loop. This is a code segment that is executed a certain number of times and then finishes. This is accomplished easily in 8052 assembly language with the DJNZ instruction. DJNZ means decrement, jump if not zero. Consider the following code:

```
      MOV R0,#08h  ;Set number of loop cycles to 8

LOOP: INC A        ;Increment accumulator (or do whatever
                   ;the loop does)

      DJNZ R0,LOOP ;Decrement R0, loop back to LOOP if R0
                   ;is not 0

      DEC A        ;Decrement accumulator (or whatever you
                   ;want to do)
```

This is a very simple counter loop. The first line initializes R0 to 8, which is the number of times the loop will be executed.

The second line labeled LOOP, is the actual body of the loop. This could contain any instruction or instructions you wishe to execute repeatedly. In this case, the accumulator is incremented with the INC A instruction.

The interesting part is the third line with the DJNZ instruction. This instruction says to decrement the R0 Register, and if it is not now zero, jump back to LOOP. This instruction decrements the R0 register, then checks to see if the new value is zero and, if not, will go back to LOOP. The first time this loop executes, R0 is decremented from 08 to 07, then from 07 to 06, and so on until it decrements from 01 to 00. At that point, the DJNZ instruction fails because the accumulator is zero. That causes the program to not go back to LOOP, and thus, it continues executing with the DEC instruction—or whatever you want the program to do after the loop is complete.

DJNZ is one of the most common ways to perform programming loops that execute a specific number of times. The number of times the loop is executed depends on the initial value of the R register that is used by the DJNZ instruction.

16.12 Setting, Clearing, and Moving Bits (SETB, CLR, CPL, MOV)

One very powerful feature of the 8052 architecture is its ability to manipulate individual bits on a bit-by-bit basis. As mentioned earlier in this document, there are 128 numbered bits (00_H through $7F_H$) that may be used by the user's program as bit variables. Additionally, bits 80_H through FF_H allow access to SFRs that are divisible by 8 on a bit-by-bit basis. The two basic instructions to manipulate bits are SETB and CLR while a third instruction, CPL, is also often used.

The SETB instruction will set the specified bit, which means the bit will then have a value of "1", or "on". For example:

```
SETB 20h  ;Sets user bit 20h (sets bit 0 of IRAM address
          ;24h to 1)

SETB 80h  ;Sets bit 0 of SFR 80h (P0) to 1

SETB P0.0 ;Exactly the same as the previous instruction

SETB C    ;Sets the carry bit to 1

SETB TR1  ;Sets the TR1 bit to 1 (turns on timer 1)
```

As illustrated by these instructions, SETB can be used in a variety of circumstances.

The first example, SETB 20h, sets user bit 20_H. This corresponds to a user-defined bit because all bits between 00_H and $7F_H$ are user bits. It is clear that bit 20_H is the 32nd user-defined bit because these 128 user bits reside in internal RAM at the addresses of 20_H through $2F_H$. Each byte of Internal RAM by definition holds 8 individual bits, so bit 20_H would be the lowest bit of Internal RAM 24_H.

Note:

It is very important to understand that bit memory is a part of internal RAM. In the case of SETB 20h, we concluded that bit 20_H is actually the low bit of internal RAM address 24_H. That is because bits 00_H-07_H are internal RAM address 20_H, bits 08_H-$0F_H$ are internal RAM address 21_H, bits 10_H-17_H are internal RAM address 22_H, bits 18_H-$1F_H$ are internal RAM address 23_H, and bits 20_H-27_H are internal RAM address 24_H.

The second example, SETB 80h, is similar to SETB 20h. Of course, SETB 80h sets bit 80_H. However, remember that bits 80_H-FF_H correspond to individual bits of SFRs, not Internal RAM. Thus, SETB 80h actually sets bit 0 of SFR 80_H, which is the P0 SFR.

The next instruction, SETB P0.0, is identical to SETB 80h. The only difference is that the bit is now being referenced by name rather than number. This makes the assembly language code more readable. The assembler will automatically convert P0.0 to 80_H when the program is assembled.

The next example, SETB C, is a special case. This instruction sets the carry bit, which is a very important bit used for many purposes. It is also special in that there is an opcode that means SETB C. Although other SETB instructions require two bytes of program memory, the SETB C instruction only requires one.

Finally, the SETB TR1 example shows a typical use of SETB to set an individual bit of an SFR. In this case, TR1 is TCON.6 (bit 6 of TCON SFR, SFR address 88_H). Due to TCON's SFR address being 88_H, it is divisible by 8 and, thus, addressable on a bit-by-bit basis.

The CLR instruction functions in the same manner, but clears the specified bit. For example:

```
CLR 20h   ;Clears user bit 20h to 0
CLR P0.0  ;Sets bit 0 of P0 to 0
CLR TR1   ;Clears TR1 bit to 0 (stops timer 1)
```

These two instructions, CLR and SETB, are the two fundamental instructions used to manipulate individual bits in 8052 assembly language.

A third bit instruction, CPL, complements the value of the given bit. The instruction syntax is exactly the same as SETB and CLR, but CPL flips (complements) the bit. If the bit is cleared, CPL sets it; likewise, if the bit is set, CPL clears it.

Note:

An additional instruction, CLR A, exists that is used to clear the contents of the accumulator. This is the only CLR instruction that clears an entire SFR, rather than just a single bit. The CLR A instruction is the equivalent of MOV A,#00h. The advantage of using CLR A is that it requires only one byte of program memory, whereas the MOV A,#00h solution requires two bytes. An additional instruction, CPL A, also exists. This instruction flips each bit in the accumulator. Therefore, if the accumulator holds 255 (11111111 binary), it will hold 0 (00000000 binary) after the CPL A instruction is executed.

Finally, the MOV instruction can be used to move bit values between any given bit—user or SFR bits—and the carry bit. The instructions MOV C,*bit* and MOV *bit*,C allow these bit movements to occur. They function like the MOV instruction described earlier, moving the value of the second bit to the value of the first bit.

Consider the following examples:

```
MOV C,P0.0    ;Move the value of the P0.0 line to the carry bit
MOV C,30h     :Move the value of user bit 30h to the carry bit
MOV 25h,C     ;Move the carry bit to user bit 25h
```

These combination of MOV instructions that allow bits to be moved through the carry flag allow for more advanced bit operations, without the need for workarounds that would be required to move bit values if it were not for these MOV instructions.

Note:

The MOV instruction, when used with bits, can only move bit values to and from the carry bit. There is no instruction that allows you to copy directly from one bit to the other bit with neither bit being the carry bit. Thus, it is often necessary to use the carry bit as a temporary bit register to move a bit value from one user bit to another user bit.

16.13 Bit-Based Decisions and Branching (JB, JBC, JNB, JC, JNC)

It is often useful, especially in microcontroller applications, to execute different code based on whether or not a given bit is set or cleared. The 8052 instruction set offers five instructions that do precisely that.

JB means jump if bit set. The MCU checks the specified bit and, if it is set, jumps to the specified address or label.

JBC means jump if bit set, and clear bit. This instruction is identical to JB except that the bit is cleared if it was set. That is to say, if the specified bit is set, the MCU jumps to the specified address or label, and also clears the bit. In some cases, this can save you the use of an extra CLR instruction.

JNB means jump if bit not set. This instruction is the opposite of JB. It tests the specified bit and jumps to the specified label or address if the bit is not set.

JC means jump if carry set. This is the same as the JB instruction, but it only tests the carry bit. An additional instruction was included in the instruction set to test for this common condition because many operations and decisions are based on whether or not the carry flag is set. Thus, instead of using the instruction JB C,label, which takes 3 bytes of program memory, the programmer may use JC label, which only takes 2.

JNC means jump if carry bit not set. This is the opposite of JC. This instruction tests the carry bit and jumps to the specified label or address if the carry bit is clear.

Some examples of these instructions are:

```
JB 40h,LABEL1  ;Jumps to LABEL1 if user bit 40h is set
JBC 45h,LABEL2 ;Jumps to LABEL2 if user bit 45h set, then clears it
JNB 50h,LABEL3 ;Jumps to LABEL3 if user bit 50h is clear
JC LABEL4      ;Jumps to LABEL4 if the carry bit is set
JNC LABEL5     ;Jumps to LABEL5 if the carry bit is clear
```

These instructions are very common, and very useful. Virtually all 8052 assembly language programs of any complexity will use them—especially the JC and JNC instructions.

16.14 Value Comparison (CJNE)

CJNE (compare, jump if not equal) is a very important instruction. It is used to compare the value of a register to another value and branch to a label based on whether or not the values are the same. This is a very common way of building a switch…case decision structure or an IF…THEN…ELSE structure in assembly language.

The CJNE instruction compares the values of the first two parameters of the instruction and jumps to the address contained in the third parameter, if the first two parameters are not equal.

```
CJNE A,#24h,NOT24   ;Jumps to the label NOT24 if accumulator isn't 24h
CJNE A,40h,NOT40    ;Jumps to the label NOT40 if accumulator is
                    ;different than the value contained in Internal
                    ;RAM address 40h
CJNE R2,#36h,NOT36  ;Jumps to the label NOT36 if R2 isn't 36h
CJNE @R1,#25h,NOT25 ;Jumps to the label NOT25 if the Internal RAM
                    ;address pointed to by R1 does not contain 25h
```

As shown, the MCU compares the first parameter to the second parameter. If they are different, it jumps to the label provided; if the two values are the same then execution continues with the next instruction. This allows for the programming of extensive condition evaluations.

For example, to call the PROC_A subroutine if the accumulator is equal to 30_H, call the CHECK_LCD subroutine if the accumulator equals 42_H, and call the DEBOUNCE_KEY subroutine if the accumulator equals 50_H. This could be implemented using CJNE as follows:

```
          CJNE A,#30h,CHECK2     ;If A is not 30h, jump to CHECK2 label
          LCALL PROC_A           ;If A is 30h, call the PROC_A subroutine
          SJMP CONTINUE          ;When we get back, we jump to CONTINUE label
CHECK2:   CJNE A,#42h,CHECK3     ;If A is not 42h, jump to CHECK3 label
          LCALL CHECK_LCD        ;If A is 42h, call the CHECK_LCD subroutine
          SJMP CONTINUE          ;When we get back, we jump to CONTINUE label
CHECK3:   CJNE A,#50h,CONTINUE   ;If A is not 50h, we jump to CONTINUE label
          LCALL DEBOUNCE_KEY     ;If A is 50h, call the DEBOUNCE_KEY subroutine
CONTINUE: {Code continues here}  ;The rest of the program continues here
```

As shown, the first line compares the accumulator with 30_H. If the accumulator is not 30_H, it jumps to CHECK2, where the next comparison is made. If the accumulator is 30_H, however, program execution continues with the next instruction, which calls the PROC_A subroutine. When the subroutine returns, the SJMP instruction causes the program to jump ahead to the CONTINUE label, thus bypassing the rest of the checks.

The code at label CHECK2 is the same as the first check. It first compares the accumulator with 42_H and then either branches to CHECK3, or calls the CHECK_LCD subroutine and jumps to CONTINUE. Finally, the code at CHECK3 does a final check for the value of 50_H. This time there is no SJMP instruction following the call to DEBOUNCE_KEY because the next instruction is CONTINUE.

Code structures similar to the one shown previously are very common in 8052 assembly language programs to execute certain code or subroutines based on the value of some register, in this case the accumulator.

16.15 Less Than and Greater Than Comparison (CJNE)

Often it is necessary not to check whether a register is or is not certain value, but rather to determine whether a register is greater than or less than another register or value. As it turns out, the CJNE instruction—in combination with the carry flag—allows us to accomplish this.

When the CJNE instruction is executed, not only does it compare parameter1 to parameter2 and branch if they are not equal, but it also sets or clears the carry bit based on which parameter is greater or less than the other.

- If parameter1 < parameter2, the carry bit will be set to 1.
- If parameter1 ≥ parameter2, the carry bit will be cleared to 0.

This is the way an assembly language program can do a greater than/less than comparisons. For example, if the accumulator holds some number and we want to know if it is less than or greater than 40_H, the following code could be used:

```
             CJNE A,#40h,CHECK_LESS ;If A is not 40h, check if  < or > 40h
             LJMP A_IS_EQUAL        ;If A is 40h, jump to A_IS_EQUAL code
CHECK_LESS:  JC A_IS_LESS           ;If carry is set, A is less than 40h
A_IS_GREATER:{Code}                 ;Otherwise, it means A is greater than 40h
```

The code above first compares the accumulator to 40_H. If they are the same, the program falls through to the next line and jumps to A_IS_EQUAL because we already know they are equal. If they are not the same, execution will continue at CHECK_LESS. If the carry bit is set, it means that the accumulator was less than the second parameter (40_H), so we jump to A_IS_LESS, which will handle the less than condition. If the carry bit was not set, execution falls through to A_IS_GREATER, at which point the code for the greater than condition would be inserted.

Note:

Keep in mind that CJNE will clear the carry bit if parameter1 is greater than or equal to parameter2. That means it is very important that the values are checked to be equal before using the carry bit to determine less than/greater than. Otherwise, the code might branch to the greater than condition when, in fact, the two parameters are equal.

16.16 Zero and Non-Zero Decisions (JZ/JNZ)

Sometimes, it is useful to be able to simply determine if the accumulator holds a zero or not. This could be done with a CJNE instruction, but because these types of tests are so common in software, the 8052 instruction set provides two instructions for this purpose: JZ and JNZ.

JZ will jump to the given address or label if the accumulator is zero. The instruction means jump if zero.

JNZ will jump to the given address or label if the accumulator is *not* zero. The instruction means jump if not zero.

For example:

```
JZ ACCUM_ZERO   ;Jump to ACCUM_ZERO if the accumulator = 0

JNZ NOT_ZERO ;Jump to NOT_ZERO if the accumulator is not 0
```

Using JZ and/or JNZ is much easier and faster than using CJNE, if all that is needed is to test for a zero/non-zero value in the accumulator.

Note:

Other non-8052 architectures have a zero flag that is set by instructions, and the zero-test instruction tests that flag, not the accumulator. The 8052, however, has no zero flag, and JZ and JNZ both test the value of the accumulator, not the status of any flag.

16.17 Performing Additions (ADD, ADDC)

The ADD and ADDC instructions provide a way to perform 8-bit addition. All addition involves adding some number or register to the accumulator and leaving the result in the accumulator. The original value in the accumulator is always overwritten with the result of the addition.

```
ADD A,#25h    ;Add 25h to whatever value is in the accumulator

ADD A,40h     ;Add contents of Internal RAM address 40h to
              ;accumulator

ADD A,R4      ;Add the contents of R4 to the accumulator

ADDC A,#22h   ;Add 22h to the accumulator, plus carry bit
```

The ADD and ADDC instructions are identical except that ADD will only add the accumulator and the specified value or register, whereas ADDC will also add the carry bit. The difference between the two, and the use of both, can be seen in the following code.

This code assumes that a 16-bit number is in Internal RAM address 30_H (high byte) and address 31_H (low byte). The code will add 1045_H to the number, leaving the result in addresses 32_H (high byte) and 33_H (low byte).

```
MOV A,31h    ;Move value from IRAM address 31h (low byte) to
             ;accumulator
ADD A,#45h   ;Add 45h to the accumulator (45h is low
             ;byte of 1045h)
MOV 33h,A    ;Move the result from accumulator to
             ;IRAM address 33h
MOV A,30h    ;Move value from IRAM address 30h (hi byte) to
             ;accumulator
ADDC A,#10h  ;Add 10h to the accumulator (10h is the high
             ;byte of 1045h)
MOV 32h,A    ;Move result from accumulator to
             ;IRAM address 32h
```

This code first loads the accumulator with the low byte of the original number from internal RAM address 31_H. It then adds 45_H to it. It does not matter what the carry bit holds because the ADD instruction is used. The result is moved to 33_H and the high byte of the original address is moved to the accumulator from address 30_H. The ADDC instruction then adds 10_H to it, plus any carry that might have occurred in the first ADD step.

Both ADD and ADDC set the carry flag if an addition of unsigned integers results in an overflow that cannot be held in the accumulator. For example, if the accumulator holds the value $F0_H$ and the value 20_H is added to it, the accumulator holds the result of 10_H and the carry bit is set. The fact that the carry bit is set can subsequently be used with the ADDC to add the carry bit into the next addition instruction.

The auxiliary carry (AC) bit is set if there is a carry from bit 3, and cleared otherwise. For example, if the accumulator holds the value $2E_H$ and the value 05_H is added to it, the accumulator then equals 33_H as expected, but the AC bit is set because the low nibble overflowed from E_H to 3_H.

The overflow (OV) bit is set if there is a carry out of bit 7 but not out of bit 6, or out of bit 6 but not out of bit 7. This is used in the addition of signed numbers to indicate that a negative number was produced as a result of the addition of two positive numbers, or that a positive number was produced by the addition of two negative numbers. For example, adding 20_H to 70_H (two positive numbers) would produce the value 90_H. However, if the accumulator is being treated as a signed number the value 90_H would represent the number $–10_H$. The fact that the OV bit was set means that the value in the accumulator should not really be interpreted as a negative number.

Note:

Many other (non-8052) architectures only have a single type of ADD instruction—one that always includes the carry bit in the addition. The reason 8052 assembly language has two different types of ADD instructions is to avoid the need to start every addition calculation with a CLR C instruction. Using the ADD instruction is the same as using the CLR C instruction followed by the ADDC instruction.

16.18 Performing Subtractions (SUBB)

The SUBB instruction provides a way to perform 8-bit subtraction. All subtraction involves subtracting some number or register from the accumulator and leaving the result in the accumulator. The original value in the accumulator is always overwritten with the result of the subtraction.

```
SUBB A,#25h  ;Subtract 25h from whatever value is in
             ;the accumulator

SUBB A,40h   ;Subtract contents of IRAM address 40h from
             ;the accumulator

SUBB A,R4    ;Subtract the contents of R4 from the
             ;accumulator
```

The SUBB instruction always includes the carry bit in the subtract operation. That means if the accumulator holds the value 38_H and the carry bit is set, subtracting 6_H will result in 31_H ($38_H – 6_H$ – carry bit).

Note:

Due to SUBB always including the carry bit in its operation, it is necessary to always clear the carry bit (CLR C) before executing the first SUBB in a subtraction operation, so that the prior status of the carry flag does not affect the instruction.

SUBB sets and clears the carry, auxiliary carry, and overflow bits in much the same way as the ADD and ADDC instructions.

SUBB sets the carry bit if the number being subtracted from the accumulator is larger than the value in the accumulator. In other words, the carry bit is set if a borrow is needed for bit 7. Otherwise, the carry bit is cleared.

The auxiliary carry (AC) bit is set if a borrow is needed for bit 3; otherwise, it is cleared.

The overflow (OV) bit is set if a borrow into bit 7 but not into bit 6, or into bit 6 but not into bit 7. This is used when subtracting signed integers. If subtracting a negative value from a positive value produces a negative number, OV is set. Likewise, if subtracting a positive number from a negative number produces a positive number, the OV flag is also set.

16.19 Performing Multiplication (MUL)

In addition to addition and subtraction, the 8052 also offers the MUL AB instruction to multiply two 8-bit values. Unlike addition and subtraction, the MUL AB instruction always multiplies the contents of the accumulator by the contents of the B register (SFR $F0_H$). The result overwrites both the accumulator and B, placing the low byte of the result in the accumulator and the high byte of the result in B.

For example, to multiply 20_H by 75_H, the following code could be used:

```
MOV A,#20h    ;Load accumulator with 20h

MOV B,#75h    ;Load B with 75h

MUL AB        ;Multiply A by B
```

The result of 20_H • 75_H is $0EA0_{H.}$ Therefore, after the above MUL instruction, the accumulator holds the low byte of the answer ($A0_H$) and B holds the high byte of the answer ($0E_H$). The original values of the accumulator and B are overwritten.

If the result is greater than 255, OV is set; otherwise, it is cleared. The carry bit is always cleared and AC is unaffected.

Note:

Any two 8-bit values may be multiplied using MUL AB and a result will be obtained that fits in the 16 bits available for the result in A and B. This is because the largest possible multiplication would be (FF_H • FF_H), which would result in $FE01_H$, which comfortably fits into the 16-bit space. It is not possible to overflow a 16-bit result space with two 8-bit multipliers.

16.20 Performing Division (DIV)

The last of the basic mathematics functions offered by the 8052 is the DIV AB instruction. This instruction, as the name implies, divides the accumulator by the value held in the B register. Like the MUL instruction, this instruction always uses the accumulator and B registers. The integer (whole-number) portion of the answer is placed in the accumulator and any remainder is placed in the B register. The original values of the accumulator and B are overwritten.

For example, to multiply $F3_H$ by 13_H, the following code could be used:

```
MOV A,#0F3h  ;Load accumulator with F3h

MOV B,#13h   ;Load B with 13h

DIV AB       ;Divide A by B
```

The result of $F3_H/13_H$ is $0C_H$ with remainder $0F_H$. Thus, after this DIV instruction, the accumulator holds the value $0C_H$, and B holds the value $0F_H$.

The carry bit and the overflow bit are both cleared by DIV, unless a division by zero is attempted, in which case the overflow bit is set. In the case of division by zero, the results in the accumulator and B after the instruction are undefined.

Note:

The MUL instruction takes two 8-bit values and multiplies them into a 16-bit value, whereas the DIV instruction takes two 8-bit values and divides it into an 8-bit value and a remainder. The 8052 does not provide an instruction that divideds a 16-bit number.

Note:

You can find source code that includes 16-bit and 32-bit division in the Code Library at http://www.8052.com/codelib.phtml.

16.21 Shifting Bits (RR, RRC, RL, RLC)

The 8052 offers four instructions that are used to shift the bits in the accumulator to the left or right by one bit: RR A**,** RRC A**,** RL A**,** RLC A. There are two instructions that shift bits to the right, RR A and RRC A, and two that shift bits to the left, RL A and RLC A. The RRC and RLC instructions are different in that they rotate bits through the carry bit, whereas RR and RL do not involve the carry bit.

```
RR A   ;Rotate accumulator one bit to right, bit 0 is rotated into
       ;bit 7

RRC A  ;Rotate accumulator to right, bit 0 is rotated into
       ;carry, carry into bit 7

RL A   ;Rotate accumulator one bit to left, bit 7 is rotated into
       ;bit 0

RLC A  ;Rotate the accumulator to the left, bit 7 is
       ;rotated into carry, carry into bit 0
```

Figure 16–1 shows how each of the instructions manipulates the eight bits of the accumulator and the carry bit.

Using the shift instructions is, obviously, useful for bit manipulations. However, they can also be used to quickly multiply or divide by multiples of two.

For example, there are two ways to multiply the accumulator by two:

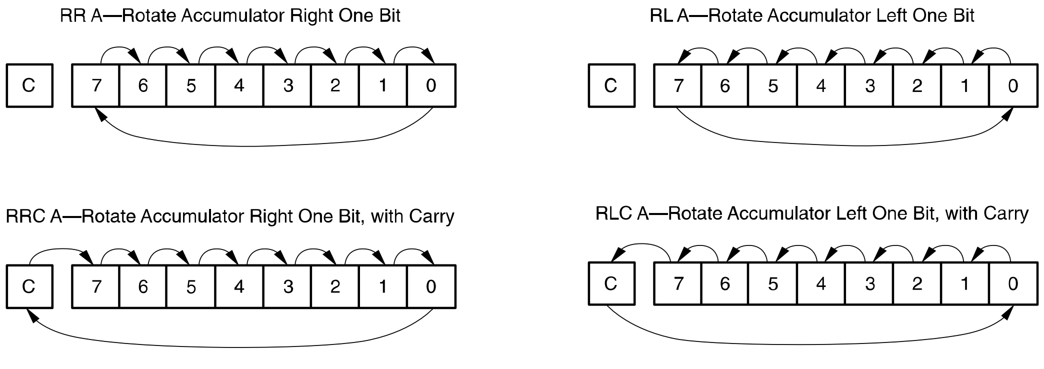
```
MOV B,#02h   ;Load B with 2

MUL AB       ;Multiply accumulator by B (2), leaving low
             ;byte in accumulator
```

Or you could simply use the RLC instruction:

```
CLR C    ;Make sure carry bit is initially clear

RLC A    ;Rotate left, multiplying by two
```

This may look like the same amount of work, but to the MCU it is not. The first approach requires four bytes of program memory and takes six instruction cycles, whereas the second approach requires only two bytes of program memory and two instruction cycles. Therefore, the RLC approach requires half as much memory and is three times as fast.

Figure 16–1. Rotate Operations


RR A—Rotate Accumulator Right One Bit
C
7
6
5
4
3
2
1
0
RL A—Rotate Accumulator Left One Bit
C
7
6
5
4
3
2
1
0
RRC A—Rotate Accumulator Right One Bit, with Carry
C
7
6
5
4
3
2
1
0
RLC A—Rotate Accumulator Left One Bit, with Carry
C
7
6
5
4
3
2
1
0

16.22 Bit-Wise Logical Instructions (ANL, ORL, XRL)

The 8052 instruction set offers three instructions to perform the three most common types of bit-level logic: logical AND (ANL), logical OR (ORL), and logical exclusive OR **(**XRL**)**. These instructions are capable of operating on the accumulator or an internal RAM address.

Some examples of these instructions are:

```
ANL A,#35h   ;Performs logical AND between accumulator
             ;and 35h, result in accumulator

ORL 20h,A    ;Performs logical OR between IRAM 20h and
             ;accumulator, result in IRAM 20h

XRL 25h,#15h ;Performs logical exclusive OR between
             ;IRAM 25h and 15h
```

ANL (logical AND) looks at each bit of parameter1 and compares it to the same bit in parameter2. If the bit is set in both parameters, the bit remains set; otherwise, the bit is cleared. The result is left in parameter1.

ORL (logical OR) looks at each bit of parameter1 and compares it to the same bit in parameter2. If the bit is set in either parameter, the bit remains set; otherwise, the bit is cleared. The result is left in parameter1.

XRL (logical exclusive OR) looks at each bit of parameter1 and compares it to the same bit in parameter2. If the bit is set in one of the two parameters, the bit is set; otherwise, the bit is cleared. That means if the bit is set in both parameters, it is cleared. If it is set in one of the two parameters, it remains set. If it is clear in both parameters it remains clear. The result is left in parameter1.

Table 16–2, Table 16–3, and Table 16–4 show the results of each of these logical instructions when applied to each possible bit combination.

Table 16–2. Results of ANL

ANL	**0**	**1**
1	0	0
0	0	1

Table 16–3. Results of ORL

ORL	**0**	**1**
0	0	1
1	1	1

Table 16–4. Results of XRL

XRL	**0**	**1**
0	0	1
1	1	0

Most of the logical bit-wise instructions affect entire 8-bit memory registers. However, the following instructions are available to perform logical operations on the carry bit. The result of these instructions is always left in the carry bit and the other bit is left unchanged.

ANL C,*bit*—this instruction will perform a logical AND between the carry bit and the specified bit. If both bits are set, the carry bit remains set. Otherwise, the carry bit is cleared.

ANL C,$\overline{bit}$—this instruction performs a logical AND between the carry bit and the complement of the specified bit. That means if the specified bit is set, the carry bit is ANDed as if it were clear. If the specified bit is clear, it is ANDed with the carry bit as if it were set.

ORL C,*bit*—This instruction will perform a logical OR between the carry bit and the specified bit. If either the carry bit or the specified bit is set, the carry bit is set. If neither bit is set, the carry bit is cleared.

ORL C,$\overline{bit}$—This instruction performs a logical OR between the carry bit and the complement of the specified bit. That means if the specified bit is set, the carry bit is ORed as if it were clear. If the specified bit is clear, it is ORed with the carry bit as if it were set.

Note:

There is no XRL that operates on the carry bit and another bit. Only the ANL and ORL logical instructions are supported with the carry bit.

16.23 Exchanging Register Values (XCH)

Very often, the value of the accumulator will need to be swapped with the value of another SFR or internal RAM address. The XCH instruction allows this to be done quickly and without using additional temporary holding variables.

XCH will take the value of the accumulator and write it to the specified SFR or internal RAM address, while at the same time writing the original value of that SFR or internal RAM address to the accumulator.

For example:

```
MOV A,#25h   ;accumulator now holds 25h

MOV 60h,#45h ;internal RAM 60h now holds 45h

XCH A,60h    ;accumulator now holds 45, IRAM 60h now holds 25h
```

16.24 Swapping Accumulator Nibbles (SWAP)

In some cases, it can be useful to swap the nibbles of the accumulator. A nibble is 4 bits, therefore, there are two nibbles in the accumulator. The high nibble consists of bits 4 through 7, whereas the low nibble consists of bits 0 through 3.

The SWAP A instruction will swap the two nibbles of the accumulator. For example, if the accumulator holds the value 56_H, the SWAP instruction converts it to 65_H. Likewise, $F7_H$ is converted into $7F_H$.

Note:

The SWAP A instruction is identical to executing four RL A instructions.

16.25 Exchanging Nibbles Between Accumulator and Internal RAM (XCHD)

The XCHD instruction swaps the low nibble of the accumulator with the low nibble of the register or internal RAM address specified in the instruction.

For example, if R0 holds 87_H and the accumulator holds 24_H, then the XCHD R0 instruction results in the accumulator holding 27_H and R0 holding 84_H. The low nibbles of the two are simply exchanged.

I personally have never used this instruction, but presumably it is useful in some situations because 11 opcodes of the 8052 instruction set are devoted to it.

16.26 Adjusting Accumulator for BCD Addition (DA)

DA A is a very useful instruction if you are doing BCD-encoded addition.

BCD stands for binary coded decimal, and is a form of expressing two decimal digits in a single 8-bit byte. When any 8-bit value is expressed in hexadecimal, it can be expressed as a number between 00 and FF. Obviously, it is possible to express all normal decimal numbers between 0 and 99 in hexadecimal format so that, printed as hexadecimal, they appear to be decimal numbers.

For example, the decimal digits 00 are represented in BCD as, not surprisingly, 00_H. The decimal digits 09 are represented in BCD as 09_H. The decimal digits 10, however, are represented in BCD as 10_H—but note that 10_H is actually 16 (decimal). That is because in BCD, the hex values A, B, C, D, E, and F are not used. Thus, 09_H jumps to 10_H.

This is all fine and good, but what happens when adding two BCD numbers together? For example, what happens when adding 38 to 25? Obviously, in normal decimal math, 38 + 25 = 63. Ideally, doing the same addition on BCD encoded values would have the same result.

However, 38 encoded as BCD is 38h and 25 encoded as BCD is $25_{H.}$ $38_H + 25_H = 5D_H$. Obviously the result no longer looks like a decimal value—and that is not surprising because BCD does not use the values A, B, C, D, E, and F.

What DA A does is automatically adjusts the accumulator after the addition of two BCD values. In the previous example, executing DA A when the accumulator holds $5D_H$ will result in the accumulator being adjusted to 63_H, thereby fixing our rather strange addition.

The details of how DA A works and why are not extremely important to this tutorial and would tend to confuse things rather than explain them. If planning on doing BCD addition, please investigate this instruction further. For the majority that will not be doing BCD addition, you can safely ignore this instruction.

16.27 Using the Stack (PUSH/POP)

The stack, as with any processor, is an area of memory that can be used to store information temporarily, including the return address for returning from subroutines that are called by ACALL or LCALL. The 8052 automatically handles the stack when making an ACALL or LCALL, as well as when returning with the RET instruction. The stack is also handled automatically when an ISR is triggered by an interrupt, and when returning from the ISR with the RETI instruction.

Additionally, the stack can be used for your purposes and for temporary storage by using the PUSH and POP instructions. The PUSH instruction puts a value onto the stack, and the POP instruction takes off the last value put on the stack. A value can be saved temporarily by PUSHing it onto the stack, and that value may be restored by POPing it.

Note:

The stack operates on a last in first out (LIFO) basis. When PUSHing the values 4, 5, and 6 (in that order), POPing them one at a time will return 6, 5, and then 4. The value most recently added to the stack is the first value that will come off when executing a POP instruction.

An example using the PUSH and POP instructions is:

```
MOV A,#35h  ;Load the accumulator with the value 35h
PUSH ACC    ;Push accumulator onto stack, accumulator still holds 35h
ADD A,#40h  ;Add 40h to the accumulator, accumulator now holds 75h
POP ACC     ;Pop the accumulator from stack, accumulator holds
            ;35h again
```

The above code is functionally useless. However, it does illustrate how to use PUSH and POP.

The code starts by assigning 35_H to the accumulator. It then PUSHes it onto the stack. Then it adds 40_H to the accumulator, just to change the accumulator to something else. At this point the accumulator holds 75_H. Finally, it POPs from the stack into the accumulator. The POP restores the value of the accumulator to 35_H because the last value pushed onto the stack was 35_H.

Note:

When PUSHing or POPing the accumulator, it must be referred to as ACC because that is the memory location of the SFR. The instructions PUSH A and POP A cannot be assembled—both of these will result in an assemble-time error in most, if not all, 8052 assemblers.

When using PUSH, the SFR or internal RAM address that follows the PUSH instruction is the value that is PUSHed onto the stack. For example, PUSH ACC pushes the value of the accumulator onto the stack. PUSH 60h pushes the value of internal RAM address 60_H onto the stack.

Likewise, the internal RAM address or SFR that follows a POP instruction indicates where the value should be stored when it is POPed from the stack. For example, POP ACC pops the next value off the stack and into the accumulator. POP 40h pops the next value off the stack and into internal RAM address 40_H.

The stack itself resides in internal RAM and is managed by the SP (stack pointer) SFR. SP will always point to the internal RAM address from which the next POP instruction should obtain the data.

- ❑ POP will return the value of the internal RAM address pointed to by SP, then decrement SP by 1.
- ❑ PUSH will increment SP by 1, then store the value at the IRAM address then pointed to by SP.

SP is initialized to 07_H when an 8052 is first powered up. That means the stack begins at 08_H and grows from there. If PUSHing 16 values onto the stack, for example, the stack occupies addresses 08_H through 17_H.

Using the stack can be both useful and powerful, but it can also be dangerous when incorrectly used. Remember that the stack is also used by the 8052 to remember the return addresses of subroutines and interrupts. If the stack is modified incorrectly, it is very easy to cause the program to crash or to behave in very unexpected ways.

When using the stack, all but advanced stack users should observe the following recommendations:

1) When using the stack from within a subroutine or ISR, be sure there is one POP instruction for every PUSH instruction. If the number of POPs and PUSHes are not the same, the program will probably end up crashing.
2) When using PUSH, be sure to always POP that value off the stack—even if not in a subroutine.
3) Be sure to not jump over the section of code that POPs a value off the stack. A common error is to PUSH a value onto the stack and then execute a conditional instruction that jumps over the instruction that POPs that value off. This results in an unbalanced stack and will probably end up crashing the program. Remember, not only must there be a POP instruction for every PUSH, but a POP instruction must be executed for every PUSH that is executed. Make sure the program does not jump over the POP instructions.
4) Always make sure to use the RET instruction to return from subroutines and RETI instruction to return from ISRs.
5) As a practice, only modify SP at the very beginning of the program in order to initialize it. Once the stack is being used or subroutine calls are being made, do not modify SP.
6) Make sure the stack has enough room. For example, the stack starts by default at address 08_H. If there is a variable at internal RAM address 20_H, then the stack has only 24 bytes available to it, from 08_H through $1F_H$. If the stack is 24 bytes long and another value is pushed onto the stack or another subroutine is called, the variable at 20_H will be overwritten.

Keep in mind, too, that the 8052 can only use internal RAM for its stack. Even if there is 64k of external RAM, the 8052 can only use its 256 bytes of internal RAM for the stack. That means the stack should be used very sparingly.

16.28 Setting the Data Pointer DPTR (MOV DPTR)

The next few instructions use the data pointer (DPTR), the only 16-bit register in the 8052. DPTR is used to point to a RAM or ROM address when used with the following instructions that are explained.

As described earlier, DPTR is really made up of two SFRs: DPH and DPL which hold the high and low bytes, respectively, of the 16-bit data pointer. However, when DPTR is used to access memory, the 8052 will treat DPTR as a single address.

To set the DPTR to a specific address, the MOV DPTR instruction is used. This instruction sets both DPH and DPL in a single instruction. However, DPTR can still be modified by accessing DPH and DPL directly, as illustrated in the following examples:

```
MOV DPTR,#1234h  ;Sets DPTR to 1234h

MOV DPTR,#0F123h ;Sets DPTR to F123h

MOV DPH,#40h     ;Sets DPTR high–byte to 40h (DPTR now 4023h)

MOV DPL,#56h     ;Sets DPTR low–byte to 56h (DPTR now 4056h)
```

As shown, the first two instructions set DPTR first to 1234_H and then to $F123_H$. The next example sets DPH to 40_H, leaving the DPTR low byte unchanged. Changing DPH to 40_H will result in DPTR being equal to 4023_H because the low byte is still 23_H from the previous example. Finally, we change the low byte to 56_H, leaving the high byte unchanged. Setting the low byte to 56_H will leave the DPTR with a value of 4056_H because the high byte was set to 40_H in the previous example.

In other words, MOV DPTR,#1567h is the same as MOV DPH,#15h and MOV DPL,#67h. The advantage to using MOV DPTR is that it uses only three bytes of memory and two instruction cycles, whereas the other method requires six bytes of memory and four instruction cycles.

16.29 Reading and Writing External RAM/Data Memory (MOVX)

The 8052 generally has 128 or 256 bytes of internal RAM that is accessed with the MOV instruction, as described previously. However, many projects will require more than 256 bytes of RAM. The 8052 has the ability of addressing up to 64k of external RAM in the form of additional, off-chip ICs.

The MOVX instruction is used to read from and write to external RAM. The MOVX instruction has four forms:

1) MOVX A,@DPTR—reads external RAM address DPTR into the accumulator.

2) MOVX A,@R#—reads external RAM address pointed to by R0 or R1 into the accumulator.

3) MOVX @DPTR,A—sets external RAM address DPTR to the value of the accumulator.

4) MOVX @R#,A—sets external RAM address held in R0 or R1 to the value of the accumulator.

The first two forms move data from external RAM into the accumulator, wheras the last two forms move data from the accumulator into external RAM.

MOVX with DPTR—when using the forms of MOVX that use DPTR, DPTR is used as a 16-bit memory address. The 8052 automatically communicates with the off-chip RAM, obtains the value of that memory address, and stores it in the accumulator (MOVX A,@DPTR), or writes the accumulator to the off-chip RAM (MOVX @DPTR,A).

For example, to add 5 to the value contained in external RAM address 2356_H, use the following code:

```
MOV DPTR,#2356h ;Set DPTR to 2356h

MOVX A,@DPTR    ;Read external RAM address 2356h into
                ;accumulator

ADD A,#05h      ;Add 5 to the accumulator

MOVX @DPTR,A    ;Write new value of accumulator back to
                ;external RAM 2356h
```

MOVX with @R0 or @R1—when using the forms of MOVX that use @R0 or @R1, R0 or R1 will be used to determine the address of external RAM to access. These forms of MOVX can only be used to access external RAM addresses 0000_H through $00FF_H$, unless actions are taken to control the high byte of the address, because both R0 and R1 are 8-bit registers.

16.30 Reading Code Memory/Tables (MOVC)

It is often useful to be able to read code memory itself from within a program. This allows for the placement of data or tables in code memory to be read at run time by the program itself. This is accomplished by the MOVC instruction.

The MOVC instruction comes in two forms: MOVC A,@A+DPTR and MOV A,@A+PC. Both instructions move a byte of code memory into the accumulator. The code memory address from which the byte is read depends on which of the two forms is used.

MOV C A,@A+DPTR reads the byte from the code memory address calculated by adding the current value of the accumulator to that of DPTR. For example, if DPTR holds the value 1234_H and the accumulator holds the value 10_H, the instruction copies the value of code memory address 1244_H into the accumulator. This can be thought of as an absolute read because the byte is always read from the address contained in the two registers, accumulator and DPTR. DPTR is initialized to point to the first byte of the table, and the accumulator is used as an offset into the table.

For example, perhaps there is a table of values that resides at 2000_H in code memory. A subroutine needs to be written that obtains one of those six values based on the value of the accumulator. This could be coded as:

```
      MOV A,#04h      ;Set accumulator to offset into the
                      ;table we want to read

      LCALL SUB       ;Call subroutine to read 4th byte of
                      ;the table

…

SUB:  MOV DPTR,#2000h ;Set DPTR to the beginning of the
                      ;value table

      MOVC A,@A+DPTR  ;Read the 5th byte from the table

      RET             ;Return from the subroutine
```

MOVC A,@A+PC will read the byte from the code memory address calculated by adding the current value of the accumulator to that of the Program Counter; that is, the address of the currently executing instruction. This can be thought of as a relative read because the address of code memory from which the byte will be read depends on where the MOVC instruction is found in memory. This form of MOVC is used when the data read immediately follows the code that read it.

For example, if the data in the previous example are located right after the routine that read it, instead of being located at code memory 2000_H, the subroutine could be changed to:

```
SUB:   INC A          ;Increment accumulator to account for
                      ;RET instruction

       MOVC A,@A+PC ;Get the data from the table

       RET            ;Return from subroutine

       DB 01h,02h,03h,04h,05h   ;The actual data table
```

Note:

In the above example, we first increment the accumulator by 1. This is because the value of PC will be that of the instruction immediately following the MOVC instruction—in this case, the RET instruction. The RET opcode is not needed, but the data that follows RET is. The accumulator needs to be INCremented by 1 byte to skip over the RET instruction because the RET instruction requires one byte of code memory.

Note:

The value that the accumulator must be incremented by is the number of bytes between the MOVC instruction and the first data of the table being read. For example, if the RET instruction above is replaced with an LJMP instruction that is 3 bytes long, the INC A instruction would be replaced with ADD A,#03h to increment the accumulator by 3.

16.31 Using Jump Tables (JMP @A+DPTR)

A frequent method for quickly branching to many different areas in a program is by using jump tables. For example, branching to different subroutines based on the value of the accumulator could be accomplished with the CJNE instruction (which has already been covered):

```
        CJNE A,#00h,CHECK1 ;If it’s not zero, jump to CHECK1
        AJMP SUB0          ;Go to SUB0 subroutine
CHECK1: CJNE A,#01h,CHECK2 ;If it’s not 1, jump to CHECK2
        AJMP SUB1          ;Go to SUB1 subroutine
CHECK2:
```

This code will work, but each additional possible value increases the size of the program by 5 bytes—3 bytes for the CJNE instruction and 2 bytes for the AJMP instruction.

A more efficient way is to create a jump table by using the JMP @A+DPTR instruction. Like the MOVC @A+DPTR, this instruction calculates an address by summing the accumulator and DPTR, and then jumps to that address. Therefore, if DPTR holds 2000_H and the accumulator holds 14_H, the JMP instruction jumps to 2014_H.

Consider the following code:

```
             RL A                   ;Rotate accumulator left, multiply by 2
             MOV DPTR,#JUMP_TABLE   ;Load DPTR with address of jump table
             JMP  @A+DPTR           ;Jump to the corresponding address
JUMP_TABLE:  AJMP SUB0              ;Jump table entry to SUB0
             AJMP SUB1
```

This code first takes the value of the accumulator and multiplies it by two by shifting the accumulator to the left by one bit. The accumulator must first be multiplied by two because each AJMP entry in JUMP_TABLE is two bytes long,

The code then loads the DPTR with the address of the JUMP_TABLE and proceeds to JMP to the address of the accumulator plus DPTR. No additional checks are necessary because we already know that we want to jump to the offset indicated by the accumulator. We jump directly into the table that jumps to our subroutine. Each additional entry in the jump table will require only two additional bytes (two bytes for each AJMP instruction).

Note:

It is almost always a good idea to use a jump table if there are two or more choices based on a zero-based index. A jump table with just two entries, like the previous example, saves one byte of memory over using the CJNE approach, and saves three bytes of memory for each additional entry.

Chapter 17

Keil Simulator

Chapter 17 describes the Keil simulator and its functions.

Topic **Page**

17.1 Description

The µVision2 is an integrated software development platform that combines a robust screen editor, and project manager with make facilities. In addition, the Keil package has an integrated source-level debugger that contains a high-speed simulator that gives you the ability to simulate the entire 8051 system. This includes the full complement of the 8051 resources, and the MSC1210 specific on-chip peripherals and external hardware peripherals. You can configure the µVision2 platform for the attributes and peripherals specific to the particular member of the TI MSC1210 family that is being targeted. The process for accomplishing this is outlined in the *Keil Software Getting Started* manual.

When the target system is properly selected, in the debug mode, one has access to the full complement of special MSC1210 peripherals. Some of the peripherals available on the simulator are common to the standard 8051 device, whereas the others are specific to the MSC1210. Following is a list of the MSC1210 simulated peripherals:

1) Interrupts
2) Port I/O: Port 0, Port1, Port2, Port3
3) Serial Ports: Serial 0, Serial 1
4) Timers: Timer 0, Timer 1, Timer 2, System Timer, Watchdog
5) SPI
6) Analog-to-Digital Converter
7) Summation/Shifter
8) Clock Control

The use and the applications of these peripherals are discussed in this section.

The graphical user interface (GUI) core of the Keil Simulator consists of a collection of individual dialog windows that represent the respective MSC1210 peripheral module that is being simulated. These dialogs facilitate interaction between the user/developer and the simulator. Facilities are provided for data to be written to, or read from, the various SFRs that control or reflect the status of the individual peripheral modules being simulated by the Keil development platform. These interactive fields include the following:

1) Various editable and noneditable text windows. The contents of these windows represent the current value that is programmed into the respective SFR or the present value of the pertinent register.

2) There are also special selection list windows that display a list of choices from which you are allowed to choose. The default settings for these list items are selected and displayed upon the activation of the peripheral module. Any selection made through this medium directly affects the setting of the pertinent SFR, on the basis of the location of the affected bit(s) within the bit pattern of the register. If the SFR is represented in the peripheral module, its value is immediately updated and displayed in the proper text display window.

3) There are also some labeled check boxes whose statuses, checked or cleared, directly affect the associated bit within the respective bit pattern of the SFR. A checked status on a check box item represents a logic 1, while a cleared status on a check box item represents a logic 0. Conversely, the current status of the corresponding bit within the associated bit pattern of the SFR is reflected in the pertinent check box.

4) In addition, there are some non-editable text field windows whose values or statuses neither represent the value of any SFR nor the status of any particular bit field within an SFR. Instead, the contents of these text fields represent the information inferred or deducted from a combination of statuses and conditions of the pertinent peripheral module(s). For instance, in the snap shot of the peripheral module depicted in Figure 17–1, Timer/Counter 0, is set for the timer option and in mode 2. Referring to Chapter 8, *Timers*, when the GATE is in an active state, and $\overline{\text{INT0}}$ is active, if the TR0 bit of the TCON SFR is also set, the timer continues running, hence, the Run status displayed in the non-editable text window labeled Status. If the TR0 bit of TCON represented by the check box is activated once, clearing the state of the TR0 check box, the state of the non-editable text window labeled Status reverts to Stop, implying that Timer 0 has stopped running.

Note:

Parameter specification through the various dialog boxes is just an alternative data entry facility for modifying the content of the corresponding SFR on the fly, during the debugging process or during the software development process. This can just as easily be accomplished by modifying the program so that the software reprograms the pertinent SFR, or by directly accessing and modifying the internal data address of the corresponding SFR. For instance, to change the mode selection of the timer/counter 0 module to mode 3, you could:

1) Assign 0x1B to the variable TMOD in the software program, recompile the program and re-execute, or

2) Perform a direct memory access to the RAM memory at address location D:0x89, and overwrite its contents with a value of 0x1B, or

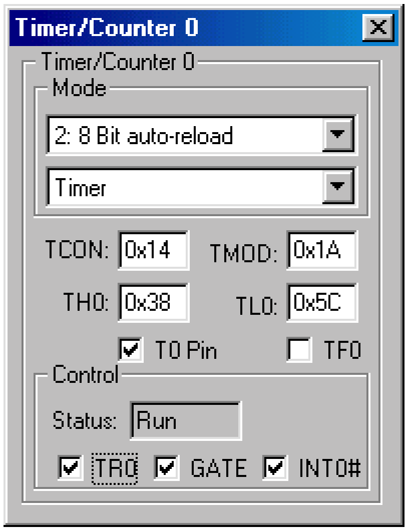
3) Place the cursor on the editable text window labeled TMOD, and replace its contents with a value of 0x1B.

4) Activate the Mode selection box, and choose the item marked “3: Two 8 Bit Timer/Cnt”.

17.2 Timers

The simulator peripheral timer has three timer/counter modules: Timers 0, 1, and 2; a system timer module; and a watchdog module. The Timer/Counter 0 module is identical to the Timer/Counter 1, so we shall only describe the operations of Timer/Counter 0.

Figure 17–1. Timer/Counter 0 – Mode 2



Timer/Counter 0

Timer/Counter 0

Mode

2: 8 Bit auto-reload

Timer

TCON: 0x14 TMOD: 0x1A

TH0: 0x38 TL0: 0x5C

☑ T0 Pin ☐ TF0

Control

Status: Run

☑ TR0 ☑ GATE ☑ INT0#

The first of the two selection boxes provides a list of four timer-operating modes upon activation, from which you are allowed to choose. The various timing modes are discussed in the Chapter 8, *Timers*. The default mode is mode 0, the 13-bit timer/counter mode. This selection properly updates the content of the TMOD on the basis of the logic statuses of the M1 and M0 bits of the Timer Mode control register (TMOD). The second selection box allows you to select between the timer and the counter modes of operation. The result of the selection is also properly reflected in the value displayed in the TMOD window, according to the status of the C/$\overline{\text{T}}$0 bit in the TMOD register. In the same vein, the TL0 and TH0 registers are properly associated with the contents of the TL0 and TH0 windows within the dialog. The logical states of the T0 pin (P3.4), TF0 (Timer/Counter 0 interrupt flag), the TR0, $\overline{\text{INT0}}$ and GATE bits of the TCON SFR for instance, are reflected in the checked/cleared statuses of the T0 pin, TF0, TR0, GATE and INT0 check box displays, respectively.

The interrupt trigger type is determined by the state of the IT0 bit within the TCON register. Clearing this bit implies level triggering, and setting it implies falling edge triggering. If the level triggering option is selected, care must be taken to make sure that the state of the $\overline{\text{INT0}}$ pin returns to a high state (non–active) before returning from an ISR, otherwise the interrupt request will be reasserted. For most intents and purposes, it is unrealistic that you would be able to switch the check box $\overline{\text{INT0}}$ on and off fast enough to avoid causing an unintended interrupt request. For this reason, an edge triggered interrupt option is recommended for simulation. Whatever the case may be, both trigger types are accommodated and implemented in this simulation package.

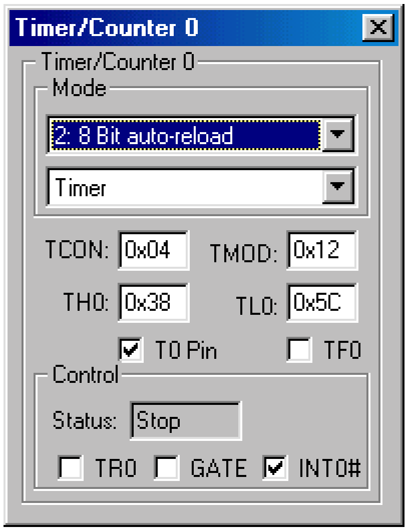
Due to the MSC1210 peripherals being modular and relatively independent, even if they share registers, each peripheral has it own unique set of bits that are associated and affiliated with it. For instance, referring to the Chapter 8, *Timers*, the status and setup bits for both Timer/Counter 0 and Timer/Counter 1 occupy separate bit positions within the same TCON SFR. The same holds for the TMOD register.

17.2.1 Timer 0 & 1 Example

Due to this sample program setting the interrupt trigger edge type for edge trigger, a transition from cleared to checked on the $\overline{\text{INT0}}$ line will induce an interrupt request.

The contents of the registers TH0, TL0, TH1 and TL1 in Figure 17–2 reflect the snapshot values of the Timer 0 MSB, Timer 0 LSB, Timer 1 MSB and Timer 1 LSB registers respectively. As explained earlier, altering the contents of any of these register displays is equivalent to altering the contents of the associated device register outside the operational confines of the program being executed.

Figure 17–2. Timer/Counter 0



Timer/Counter 0

Timer/Counter 0

Mode

2: 8 Bit auto-reload

Timer

TCON: 0x04 TMOD: 0x12

TH0: 0x38 TL0: 0x5C

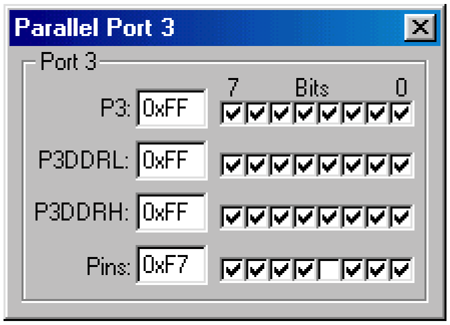
☑ T0 Pin ☐ TF0

Control

Status: Stop

☐ TR0 ☐ GATE ☑ INT0#

Figure 17–3. Parallel Port 3 Peripheral



Parallel Port 3

Port 3

7 Bits 0

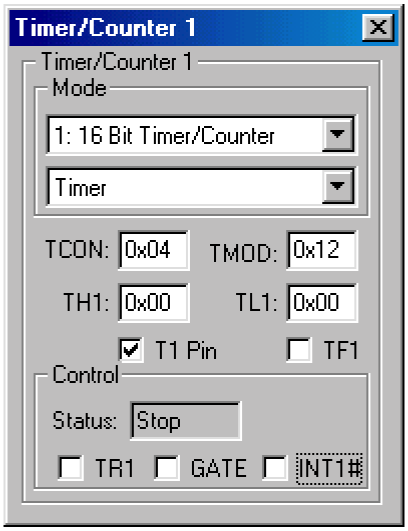
P3: 0xFF ☑☑☑☑☑☑☑☑

P3DDRL: 0xFF ☑☑☑☑☑☑☑☑

P3DDRH: 0xFF ☑☑☑☑☑☑☑☑

Pins: 0xF7 ☑☑☑☑☐☑☑☑

Figure 17–4. Timer/Counter 1 Mode 1



Timer/Counter 1

Timer/Counter 1

Mode

1: 16 Bit Timer/Counter

Timer

TCON: 0x04 TMOD: 0x12

TH1: 0x00 TL1: 0x00

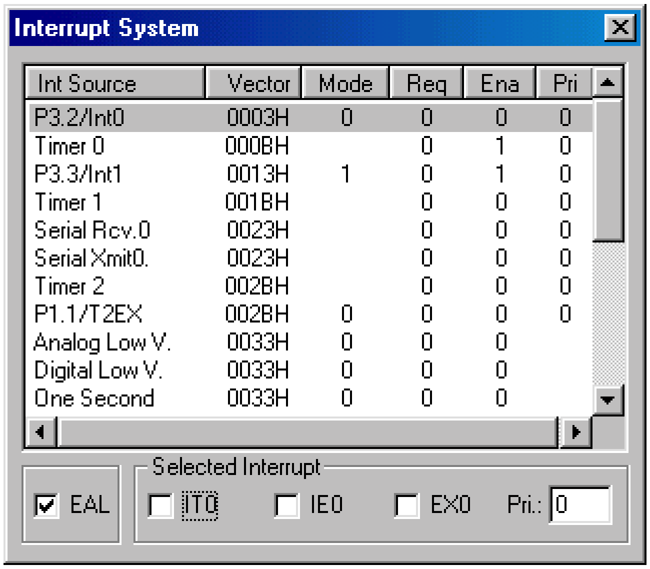
☑ T1 Pin ☐ TF1

Control

Status: Stop

☐ TR1 ☐ GATE ☐ INT1#

Figure 17–5. Interrupt System



Interrupt System

Int Source	Vector	Mode	Req	Ena	Pri
P3.2/Int0	0003H	0	0	0	0
Timer 0	000BH		0	1	0
P3.3/Int1	0013H	1	0	1	0
Timer 1	001BH		0	0	0
Serial Rcv.0	0023H		0	0	0
Serial Xmit0.	0023H		0	0	0
Timer 2	002BH		0	0	0
P1.1/T2EX	002BH	0	0	0	0
Analog Low V.	0033H	0	0	0	
Digital Low V.	0033H	0	0	0	
One Second	0033H	0	0	0	

☑ EAL

Selected Interrupt

☐ IT0 ☐ IE0 ☐ EX0 Pri.: 0

The following is a listing of the C code used to demonstrate the timing and interrupting features of Timer/Counters 0 and 1.

The included statements on the first three lines are the conventional ANSI C include statements for adding the contents of different header files to a C program. There are four routines including the main ( ) program that are needed to run this program. They are described in the following paragraphs.

```
#include ”MSC1210.h”
#include <math.h>
#include <stdio.h>
long int timer_0_overflow_count;
int    count_start;
char   end_test;
void interrupt_timer0 ( );// interrupt 1;// using 1
*********************************setport ( )*************************************
void setport (void)
{
   P3DDRL &= 0xf0;
   P3DDRL |= 0x07;  //P30 input, P31 output
   TF2 = CLEAR; T2 = CLEAR;
   CKCON |= 0x20;      // Set timer 2 to clk/4
   RCAP2 = 0xffd9; //Set Timer 2 to Generate 57690 bps
   //Initialize TH2:TL2 so that next clock generates first Baud Rate pulse
   THL2=0xffff;
   T2CON = 0x34; // Set T2 for Serial0 Tx/Rx baudgen
   //SCON: Async mode 1, 8–bit UART, enable rcvr; TI=CLEAR, RI = CLEAR
   SCON  = 0x50;
   PCON |= 0x80; // Set SMOD0 for 16X baud rate clock
}
***************************interrupt_timer 0 ( )*********************************
```

This is a type 1 interrupt, which implies that the vector address for this routine is 0x0B. If an interrupt request is issued, and there is no other interrupt request of higher priority pending, and neither is an ISR from an interrupt source of higher priority being processed, the processor makes a subroutine call to the vector address location 0x0B, from where it executes a long jump to the intended ISR routine.

There is a counter variable of type LONG within this ISR that allows the system to monitor the number of overflow interrupts serviced during the course of the test. In addition, because Timer 0 is operating in mode 1, 16 bit timer with interrupt on overflow, the original value of the TH0:TL0 register pair must be replenished at the end of each overflow cycle. The system is globally interrupt disabled at the beginning of the routine, and then globally interrupt enabled at the end of the routine.

```
void interrupt_timer0 ( ) interrupt 1 using 1
{   /*This ISR is called when a type 1 interrupt causes the processor to vector
    into the code segment address 0x0006.
    Register Bank 1 is used, as opposed to the default Register Bank 0.*/

    IE &= 0x7f;  //disable global interrupt
    timer_0_overflow_count++; //Track number of times this ISR is called
    //Reinitialize Timer 0 counters
    TH0 = count_start / 256;   //set THO for timer0
    TL0 = count_start % 256;   //set TLO for timer0
    IE |= 0x80;  //enable global interrupt
}
**************************interrupt_external 0 ( )*********************************
```

Timer 0 is set up as a gated timer. This implies that while GATE is high and if the $\overline{\text{INT0}}$ is low, there should not be any time count, regardless of the fact that the TR0 line is asserted. Hence, the Timer 0 status window displays stop. While the GATE is high and TR0 is to logic 1, if the $\overline{\text{INT0}}$ line is raised, the timer starts running, changing the status display to run. It continues running until the $\overline{\text{INT0}}$ line is dropped. This is essentially a pulse-width measurement program.

If the number of calls is odd, the TR0 bit for timer 0 is reset, which effectively stops the timer, regardless of the state of GATE and $\overline{\text{INT0}}$. The status window now displays stop. The global variable end_test is set to a value of 1. This allows the process to terminate the idle loop in the main program.

```
void interrupt_external0 (void) interrupt 2 using 1
{   /*This ISR is called when a type 2 interrupt causes the processor to vector
    into the code segment address 0x0013.
    Register Bank 1 is used, as opposed to the default Register Bank 0.*/

    static i; //declare static variable i, in order to track odd and even
              //number of calls to this ISR

    if (!(i++ % 2))
    {//even number of calls including 0
       //turn on timer0
       TCON |= 0x10; //Start timer0 by setting TR0 = 1
    }
    else
    {//odd number of calls
       TCON &= 0xef; //Stop timer0 by setting TR0 = 0
       end_test = 1;
    }
}
*********************************main ( )****************************************
```

Every time the idle loop is interrupted, the MSC1210 vectors to the ISR of the interrupting signal. If the interrupt source is the Timer 0 overflow, the processor vectors to the interrupt_timer 0 ( ) ISR, where the timer_0_overflow_count variable is updated, and the TH0:TL0 register pair is replenished with a value of 0x0200. If the external Interrupt 0 signal is the interrupt source, the interrupt_external 0 ( ) is vectored to. This ISR keeps track of even and odd ISR calls. For odd number of ISR calls, Timer 0's TR0 bit is set, preparing the timer to start running as soon as the $\overline{\text{INT0}}$ line is raised. For even number of calls, the ISR resets the TR0 bit for Timer 0. This will stop Timer 0 from timing, whether the $\overline{\text{INT0}}$ line is asserted or not. In addition, the value of end_test is changed to 1, so that upon re-entering the idle loop, the value of end_test is no longer 0, which forces the processor out of the loop.

Upon terminating the idle loop, the MSC1210 starts to compute the time lapse within the period when the $\overline{\text{INT0}}$ line was asserted and the Timer 0 TR0 bit was high, or the time between the period when TR0 and $\overline{\text{INT0}}$ were high, and TR0 went low. This is purely arithmetic. It is important to state that TH0 and TL0 never start at zero, hence the 0x0200 correction.

$$time\_lapse = \left(timer\_0\_overflow + \frac{current\_count - 0x0200}{(0x10000 - 0x0200)} \right) \cdot \frac{12 \cdot (0x10000 - 0x0200)}{24 \cdot 10^{6}}$$

The result of the pulse width computation is displayed on the Serial #1 display window.

Subsequently, the MSC1210 enters an infinite loop.

```
void main ( )
{
   float  time_lapse, time_lapse_residual, current_count;

   SP = 0x50;   //Initialize Stack Pointer
   setport ( ); //Set up UART Comm. b/w Simulator and Serial #1Window
   //Issue operation instructions
   printf (”\nMSC1210 Ver:”);
   printf (”\nTimer 0 & 1 Test\n”);
   printf (”\nActivate the Timers 0 & 1 peripherals.”);
   printf (”\nClear the Check Box for INT0. This is a Gated timer.”);
   printf (”\nTo arm the Timer 0, Clear the Check on INT1.”);
   printf (”\nTiming begins when a Check is placed on INT0.”);
   printf (”\nTiming ends either by clearing INT0 or Interrupting on INT1.”);

   //Make INT1 edge triggered
   TCON |= 0x04;
   //this global variable track the number of times the Timer 0 timed out
   timer_0_overflow_count = 0;
   //track even or odd number of calls to ISR interrupt_external0
```

```
    end_test = 0;
    //Timer 0 TH0:TL0 will always count up from 0x0200 until overflow,
    //and will be replenished with 0x0200 indefinitely
    count_start = 0x200;
    /*Timer 0 and Timer 1 in Mode 1, timer mode, Gate 0 is closed and Gate
     1 is opened. System will clock if TR0 set, only when INT0 is asserted*/
    TMOD = 0x19;
    CKCON = 0;   //Select Divide by 12
    //Enable global interrupt, timer0 overflow and external_int1 interrupts
    IE = 0x86;
    TH0 = count_start / 256;   //set THO for timer0
    TL0 = count_start % 256;   //set TLO for timer0

    /*Indefinite Idle loop.
    It breaks when interrupt_external0 ( ) ISR is called an even number of times.
    In that instance,end_test is set to ”1”,
    otherwise, it is ”0”*/
    while (!end_test);

    /*compute time elapsed, including the residual time in the 16–bit counter,
    with correction for the 0x0200 counter offset.*/
    current_count = TH0 * 256 + TL0; //current residual timer0 count
    time_lapse_residual = (float)(current_count – count_start) /
       (0x10000 – count_start);
    time_lapse = time_lapse_residual + timer_0_overflow_count;
    time_lapse *= (12 / 24000000.) * (0x10000 – count_start);
    printf (”\nThe Pulse Width for INT0 was: %f Sec.”, time_lapse);

    //enter infinite loop
    while (1);
}
```

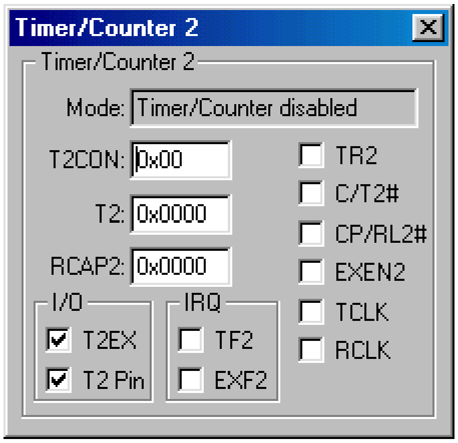
17.3 Timer 2

Timer/Counter 2 is quite different from the two other timers. The operation mode is determined by the status of one or more of the register bits displayed in Table 17–1.

Table 17–1. Timer/Counter 2 Control Bits

Register Bit	Toggle Box Name
T2CON.TR2	TR2
T2CON.C/T2	TC/T
T2CON.CP/RL2	CP/RL2
T2CON.EXEN2	EXEN2
T2CON.TCLK	TCLK
T2CON.RCLK	RCLK

Figure 17–6. Timer/Counter 2



Mode: Timer/Counter disabled

T2CON: 0x00

T2: 0x0000

RCAP2: 0x0000

☐ TR2
☐ C/T2#
☐ CP/RL2#
☐ EXEN2
☐ TCLK
☐ RCLK

☑ T2EX
☑ T2 Pin

☐ TF2
☐ EXF2

The hexadecimal equivalent of the bit pattern created by toggling the individual bits is displayed in the editable text display, T2CON. Writing a number into this window will correctly program the states of the respective bit; the same way it would program the individual bits of the device’s T2CON register.

By the same token, the checked or cleared state of the T2EX, T2 Pin, TF2 and EXF2 boxes have the same effects on, or are affected the same way by the P1.T2EX, T2CON.T2, T2CON.TF2 and T2CON.EXF2 pins respectively, as discussed earlier.

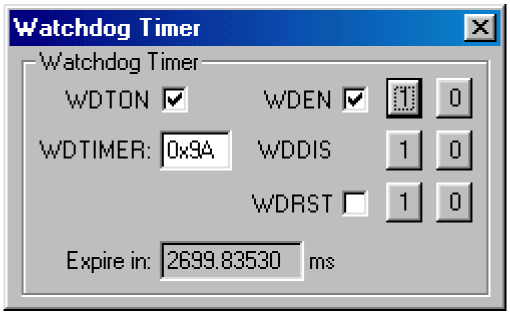
The values and the implications of the contents of the editable text windows T2 and RCAP2 are consistent with those of the actual device registers in the MSC1210 system.

17.4 Watchdog Timer

The process of setting, and the operation of the watchdog timer peripheral facility are similar to those of the other peripherals we have considered so far. However, this module has an additional feature: the special access setting or resetting of the status conditions of the EWDT, DWDT, RWDT check boxes. These boxes directly or indirectly affect, or are affected by, the status and conditions of the EWDT, DWDT, RWDT bits of the WDTCON SFR respectively. Just as these bits, for security, require special access of an application of a sequence of a logic 1, followed by a logic 0 for their respective setting and resetting in the actual MSC1210 device, so do the check boxes in this simulator peripheral. Please see section 14.3, *Watchdog Timer,* of this manual for the description of the special access programming.

To enable the watchdog timer system, the watchdog timer must be turned on either by placing a check mark on the PDWDT check box, or writing a logic 1 into the PDWDT bit of the PDCON SFR from within the software. The appropriate value must also be written into the watchdog timer register through the WDTIMER editable text window. This would properly set the WDCNT counter bits (lower five bits) of WDTIMER, through which watchdog expiration time is defined. Then the special accessed procedure is performed for the EWDT check box, involving the sequential process of activating the box marked 1 followed by the box marked 0 that are associated with the EWDT check box. While the watchdog timer is running, performing repeating the special access process for the DWDT will disable the watchdog timer. This will automatically clear the check mark in the EWDT check box. On the other hand, you could also perform the special access procedure on the RWDT check box. This places a check on the associated check box. Figure 17–7 shows the status of the watchdog peripheral mid-stride of a watchdog countdown.

Figure 17–7. Status of Watchdog Peripheral



WDTON ☑ WDEN ☑

WDTIMER: 0x9A WDDIS

WDRST ☐

Expire in: 2699.83530 ms

The non-editable Expire in: display window indicates the amount of time left (in milliseconds) before you must perform either a timed access watchdog reset or a watchdog disable, in order to avoid the watchdog timer initiating a system reset (if watchdog reset is enabled). Note that PDWDT does not enable the watchdog timer reset; it is enabled and disabled by the WDRESET bit in hardware configuration register 0 (HCR0). This register is located within the flash memory, therefore, it is not available for read or write in the execution mode. Refer to the section on flash programming for information on programming HCR0. This SFR is accessed using CADDR and CDATA. The WDRESET bit, when set, would enable the watchdog reset. This implies that upon watchdog timeout, the system automatically initiates a processor reset procedure. This also implies that even if the watchdog timer interrupt is enabled, the associated ISR will never be called. In order to be able to access the ISR, the watchdog reset must be disabled. This is achieved by clearing the watchdog reset enable bit. The default state for this bit is logic 1, i.e., watchdog reset enabled. The complete watchdog facility cannot be simulated because the configuration address and data access to the HCR0 is not implemented in this simulator version. However, an example is provided here to show how the watchdog system, with an interrupt facility, would be implemented were it possible to modify the WDRESET bit of HCR0.

17.4.1 Watchdog Reset Facility Example

```
#include ”MSC1210.H”
//unsigned char data irqen_init _at_ 0x7f ; // image of PAI
#define FWVer 0x04
#define CONVERT 0

char   watchdog_loop;

void init_watchdog ( );
void watchdog_interrupt ( );// interrupt 6;

void setport (void)
{
   P3DDRL &= 0xf0;
   P3DDRL |= 0x07;  //P30 input, P31 output
   TF2 = CLEAR; T2 = CLEAR;
   CKCON |= 0x20;      // Set timer 2 to clk/4
   RCAP2 = 0xffd9; //Set Timer 2 to Generate 57690 bps
   //Initialize TH2:TL2 so that next clock generates first Baud Rate pulse
   THL2=0xffff;
   T2CON = 0x34; // Set T2 for Serial0 Tx/Rx baudgen
   //SCON: Async mode 1, 8–bit UART, enable rcvr; TI=CLEAR, RI = CLEAR
   SCON  = 0x50;
   PCON |= 0x80; // Set SMOD0 for 16X baud rate clock
}
void init_watchdog ( )
{
   /*
   //For the actual device, the a logical AND of the content of the FRC0 SFR
   register with 0xF7 must be performed to Disable the Watchdog Reset
   so that the watchdog system can be controlled through the watchdog interrupt
   facility.
   CADDR = 0x7F;
   CDATA &= ~0x08;
   This must be done in the Parallel Programming mode, before the processor
   starts up
   */
   /*Turn Watchdog Timer On*/
   PDCON |= 0x04;
   /*Enable Watchdog Interrupt*/
   EIE |= 0x10;
   /*Set Watchdog Timer for 200 ms*/
   WDTIMER = 0x0E;
   /*Enable Watchdog Timer*/
   WDTIMER |= 0x80;
   WDTIMER &= ~0x80;
}
void watchdog_interrupt ( ) interrupt 12 using 1
{
   /*This routine cannot be tested because we cannot get around the
   watchdog reset on the simulator. The watchdog reset cannot be disabled.
   The watchdog interrupt is never activated, hence, 0x0063 is never vectored
into.*/
   static int   j;
   /*Reset Watchdog. This is the sequential process of applying
   Logic 1 followed by Logic 0*/
   WDTIMER |= 0x20;
   WDTIMER &= ~0x20;
   /*Count Number of Resets*/
   j++;
   printf (”\nWatchdog Reset %d Times”, j);
   /*Terminate watchdog Loop*/
   watchdog_loop = 0;
}
void main(void)
{
   int i, j;
   setport ( );
   init_watchdog ( );
```

```
    /*start short loop to test DWDT*/
    for (i = 0; i < 4000; i ++)
    {//idle delay
       j = (i *13) % 4000;
    }
    //Disable watchdog timer before timer expires
    WDTIMER |= 0x40;
    WDTIMER &= ~0x40;
    //Reinitialize watchdog, sinice it has just been disabled
    init_watchdog ( );
    /*start short loop to test RWDT*/
    for (i = 0; i < 400; i ++)
    {//idle delay
       j = (i *13) % 4000;
    }
    /*Reset Watchdog Timer before Watchdog Timer Expires*/
    WDTIMER |= 0x20;
    WDTIMER &= ~0x20;
    /*Infinite loop to test Watchdog Timer Time out with interrupt.
    In the case in which the Watchdog Reset cannot be disabled,
    there will not be any interrupts. The watchdog time would
    eventually run out, and a reset procedure will be activated*/
    while (1)
    {
       watchdog_loop = 1;
       while (watchdog_loop);
    }
}
```

17.5 System Timer

The MSC1210 device has many time ticks and an additional clock generator (1MHz) that are derived, and, therefore, synchronized to the system clock. Each time tick and clock generator has a set of registers that specify the value of system clock divisions required to generate it. Some of these registers are accessible through the system timer peripheral windows. The editable XTAL Freq.: window allows you to set the crystal clock frequency. Setting it is just a matter of entering the appropriate value. The ONEUSEC editable text window sets the value of the One Microsecond register. Referring to the Chapter 8, *Timers*, it is apparent that bit patterns in the fifth, sixth and seventh bit positions are ignored; they have no effect. This is also enforced in this peripheral. The One Millisecond Low and One Millisecond High registers are programmed through or displayed in the editable text windows ONEMSL and ONEMSH, respectively. The One Hundred Milliseconds register, the Millisecond Timer register and the Seconds Timer register are accessed through the HUNDMS, STIMER and MSTIMER editable text windows. The statuses of the second system timer interrupt status flag and the millisecond system timer interrupt status flag are reflected in the checked or cleared statuses of the SEC and MSEC check boxes, respectively. You can also force a second system timer iterrupt or the millisecond system timer interrupt by toggling either of these check boxes. The SYSTEM clock is turned on or turned off by toggling the SYSTON check box. Please refer to Chapter 8, *Timers*, for a more comprehensive discussion of the system timer.

The corresponding time interval for the one microsecond timer, the one millisecond timer, and the one hundred milliseconds timer, on the bases of the registered system clock frequency and the values of the pertinent timer registers, are displayed in the non-editable text windows 1µs, 1ms and 100ms, respectively.

17.6 Clock Control

The clock frequencies affecting the operations of the device timers, watchdog timers, UARTs and SPI systems depend on the states of the bit pattern of the value written into the Clock Control register (CKCON). The stretch time for the external memory access is also determined by this value. The value of the CKCON register can either be written directly into the associated editable text window, or modified or programmed by toggling the T2M, T1M and T0M check boxes, which program and set the crystal frequency divide-by-12 or divide-by-4 modes for Timers 2, 1 and 0, respectively. This allows the MSC1210 to maintain backward compatibility with the standard 8051 processor. The default setting is the divide-by-12 option. The MOVX stretch can be modified by activating the MOVX Duration (Cycles) window. This causes a display of selectable instruction cycle durations, from which, one must be chosen. Please refer to the section for Timer Control for more detailed information on the Clock Control register bit pattern.

17.7 Analog-to-Digital Converter

Data entry and bit pattern setting facilities for the ADC peripheral are similar to those of the other peripherals. Some of the text entry boxes are editable, while others are just read-only, and the check boxes respond to mouse clicks with checked and cleared status. The editable text entry windows marked ADCON0, ADCON1, ADCON2 and ADCON3 provide direct access to the ADC Control registers 0, 1, 2 and 3, respectively.

Writing to the editable text entry box marked ADCON0 sets the bit pattern for the Burnout Detect bit, the Enable Internal Voltage Reference bit, the Voltage Reference High Select bit, the Buffer Enable bit, and the bits that select the PGA. All these bits could also be set or modified by performing a check/clear activation on check boxes marked BOD, VREF, BUF and VREFH, respectively, and selecting the desired programmable gain from the gain selection list that pops up when the box marked PGA is activated.

The bit patterns for analog input data polarity, filter mode option selection, and the device’s calibration mode control option selection can be programmed or updated by entering appropriate byte data values into the editable text box marked ADCON1. Similarly, the polarity setting and the filter settling mode selection option can also be programmed, alternatively, through the check box marked POL and the selection box marked Filter. There is no data entry alternative for selecting the calibration mode control bits.

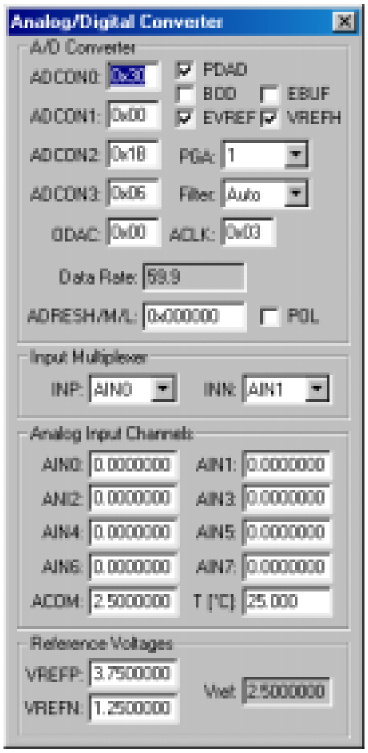
Writing data values into the ADCON2 and ADCON3 registers respectively sets the ADC decimation filter ratio values. The lower three bits of ADCON3 correspond to the most significant three bits of the converter decimation ratio, and the whole ADCON2 byte represents the LSB for the decimation ratio. It should be stressed that if the contents of either ADCON2 or ADCON3 are modified, the converter must be recalibrated.

The current ADC data conversion rate is automatically computed on the basis of the system clock setting. Its result is displayed in the non-editable text display window marked Data Rate. The data conversion completed status, is indicated by the logic state of the ADC bit of the AISTAT SFR. The 24-bit result of the most recent analog-to-digital conversion, which is a concatenation of the three result registers ADRESH, ADRESM and ADRESL, respectively, is displayed in the text display window marked ADRESH/M/L.

The MSC1210 device has an input multiplexer which facilitates the selection of any combination of any pair of differential inputs. If you select any of the input channels for the positive input of the differential input pair, any other input could be selected for the negative input. Even, the same input could be selected for both differential input pairs, if you wishe to perform the ADC conversion calibration or quantify the noise measurements of the conversion system.

There are 10 possible analog input sources, including an on-chip temperature sensor. Activating the analog input selection box associated with the pertinent differential pair input, INP or INN, and clicking on the desired source could select any of these channels. The default selection for AIN0 for INP, and AIN1 for INN.

Figure 17–8. Analog–to–Digital Converter Peripheral



Analog/Digital Converter

A/D Converter

ADCON0: 0x30
ADCON1: 0x00
ADCON2: 0x18
ADCON3: 0x06
ODAC: 0x00
☑ PDAD
☐ BOD
☐ EBUF
☑ EVREF
☑ VREFH
PGA: 1
Filter: Auto
ACLK: 0x03
Data Rate: 59.9
ADRESH/M/L: 0x000000
☐ POL

Input Multiplexer

INP: AIN0
INN: AIN1

Analog Input Channels

AIN0: 0.0000000
AIN1: 0.0000000
ANI2: 0.0000000
AIN3: 0.0000000
AIN4: 0.0000000
AIN5: 0.0000000
AIN6: 0.0000000
AIN7: 0.0000000
ACOM: 2.5000000
T [°C]: 25.000

Reference Voltages

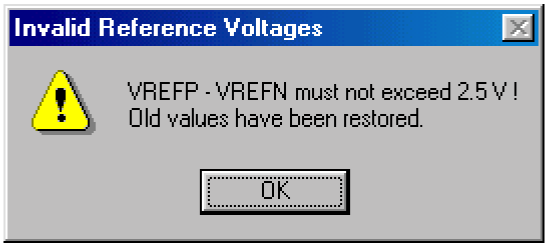
VREFP: 3.7500000
VREFN: 1.2500000
Vref: 2.5000000

For each analog input source whose editable text windows are displayed under the Analog Input Channels title, one could specify the desired analog voltages to be converted. The µVision2 simulator also provides an alternate way for entering analog voltage values by writing a script program that runs in parallel with the program being executed. A sample code is appended. Please refer to the µVision2 Debug Functions chapter in the Keil’s *Getting Started and Creative Programming* document for more information on writing scripts.

```
SIGNAL void a_to_d_sim (void)

{

   inti;

   /*Data written into the variable ain0 is automatically
   entered into the editable text window labeled AIN0 in
   the ADC peripheral dialog.*/

   ain0 = 0.5;  //specify start value for ain0

   //debug program idles for 196000 clock cycles, while
   simulation continues running in parallel*/

   twatch (196000);

   /*the following loop sends out 64 consecutive samples
   of ain0,each incremented by 0.01. Each transmittal is
   spaced 131000 clock cycles from the previous one.*/

   for (i = 0; i < 0x40; i++)

   {

      twatch (900);

      ain0 += 0.01;

      twatch (130100);

   }

}
```

The reference voltages are also specified through the VREFP and VREFN editable text windows. Checks to evaluate the validity of the values placed in these windows are also implemented. Should the difference between the value in VREFP and VREFN exceed 2.5V, the error message in Figure 17–9 is displayed.

Figure 17–9. Error Message



VREFP - VREFN must not exceed 2.5 V !
Old values have been restored.

The value of internal reference voltage, which is based on the status of the *VREFH* bit of the *ACDON0* SFR, is displayed on the non-editable V_{REF} window.

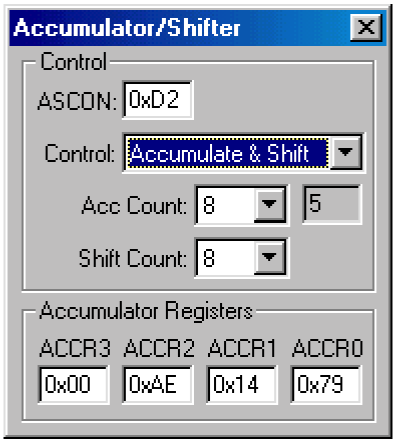
Please refer to Chapter 12, *Analog-to-Digital Converter*, for more in-depth discussions on the pertinent registers.

17.8 Summation/Shifter

The summation/shifter module implemented in this simulator package allows the developer to experience how the automatic data averaging works. It also allows the program to be tested while it is being developed.

Figure 17–10 shows a snapshot of the summation/shifter peripheral in the middle of a data acquisition cycle. The SSCON editable text field displays the programmed value of the SSCON register, which sets the bits for the summation/shifter control, the summation count, and the shift count. Please refer to the section on summation/shifter for more in depth discussions on the features of these bit pattern settings. The contents of this text field can also be updated. Like the other peripheral modules in this simulator, the items and features that this register sets are also configurable through the alternate data entry windows. The summation/shifter control setting can be alternatively made by activating the control selection window marked Control. This brings up a list of four different summation/shifter options: no source, aAccumulate, shift, and accumulate & shift. One of these options must be selected. The default is the no source option. In the accompanying example, as indicated in Figure 17–10, the accumulate & shift option was selected. Activating the Acc Count window permits the developer to determine the number of 24-bit data samples to be automatically accumulated. The count choice options are 2, 4, 8, 16, 32, 64, 128 and 256. Of these choices, one selection must be made. Figure 17–10 shows that an accumulate count of 8 was selected. In the same manner, the choice of shift count is made by activating the Shift Count selection window, and picking one of eight possible shift counts: 2, 4, 8, 16, 32, 64, 128 and 256. Updating the content or the selection choice of any of these selection window items will appropriately update the content of the SSCON editable text window.

Figure 17–10. Accumulator/Shifter Peripheral



ASCON: 0xD2

Control: Accumulate & Shift

Acc Count: 8 5

Shift Count: 8

ACCR3	ACCR2	ACCR1	ACCR0
0x00	0xAE	0x14	0x79

The intermediate result of the successive accumulations and the eventual computation of the shifting process is displayed in the Accumulate Registers editable text window sets marked ACCR3, ACCR2 ACCR1, and ACCR0. These display windows reflect the values of the contents of the ACCR3, ACCR2, ACCR1, and ACCR0 registers in the MSC1210 device.

The non-editable text window across from the acc count shows the current number of data samples accumulated into the summation registers for the current accumulate & shift cycle. The summation/shifter module depicted in Figure 17–10 shows that five samples had been accumulated, and the concatenated result of the summation registers for the freeze–framed accumulate & shift cycle, up to that point, was 0x00AE1479.

For this summation/shifter peripheral module, there is no possibility of an overflow—even in the accumulate mode or cycle—because the worst-case sample data value is 0x00FFFFFF (a 24-bit value), and the worst-case accumulate or multiply count is 256. This makes the worst-case accumulate result 0xFFFFFFFF. This worst-case scenario is comfortably accommodated because the summation register is 32 bits wide.

Please refer to Section 12.13, *Summation/Shifter Register*, for more detailed information.

17.8.1 ADC/Summation/Shifter Example

An example program has been provided to give you an insight into how to use the ADC peripheral. In order to show how the 32-bit accumulator will work with this module, a software implementation of the combination of the ADC features and the summation/shifter features have been provided. The C code is grafted into this section. In addition, a script file that runs in parallel with this C code is also provided. This script file is also written in C.

The ADC peripheral is set up with the following features: V_{REF} = 2.5V, Buff is turned on and BOD (Burn Out Detect) is turned off by assigning a value of 0x20 to ADCON0. This register setting also selects an unity gain amplification for the PGA. The bipolar option and the auto-filter options are selected through ADCON1. Setting the value of register byte also makes the calibration selection. In this case, the reserved calibration option was selected. The decimation ratio value of 0x00FF was assigned to the ACDON3:ADCON2 register pair. Please refer to Chapter 12, *Analog–to–Digital Converter*, for more information on the decimation ratio.

An ADC Conversion calibration is performed at the beginning of each data conversion session. Calibration is initiated, and the processor enters an idle state and stays there indefinitely, until the calibration process is completed. When the converter calibration is completed, the ACC flag in the AISTAT SFR is set. It is customary to discard the first 20 conversions after calibration.

The initialization of the summation/shifter is straightforward. A value of zero must be written into the SSCON register. This action clears the contents of the ACCR3, ACCR2, ACCR1 and ACCR0 SFRs. Then the proper SSCON value, in this case 0xD2, is assigned. This assignment value sets the accumulate–count, Acc_count, to eight, and the shift count value to eight. The accumulate & shift option is also selected. This process of clearing and setting the value of SSCON must be done at the beginning of every acc_count data accumulation cycle, otherwise the previous accumulate & shift result is combined with the next accumulate & shift data collection.

This setting essentially causes the 32-bit accumulator to collect eight consecutive data samples from the ADC. Upon completion, it divides the result by eight, by implementing a 3-bit position arithmetic right shift. In other words, it computes the average value of eight consecutive samples.

The following is the C code for the sample exercise described above:

```
#include ”MSC1210.H”
//unsigned char data irqen_init _at_ 0x7f ; // image of PAI
#define CONVERT 0

char   converting, averaging;

void setport (void);
void init_accumulator ();
char init_a_to_d ();
long read_a_to_d_result ();
long read_accumulator_result ();

void init_accumulator ()
{
   /*Clearing the SSCON register will always reset the concatenated string
   of ACCR3:ACCR2:ACCR1:ACCR0 registers. This must be performed prior to
   initiating a fresh set of A/D Conversion result accumulation*/
   SSCON = 0x00;
   /*Set Summation/Shifter for 8 A/D result accumulation and averaging*/
   SSCON = 0xD2;
}

void setport (void)
{
   P3DDRL &= 0xf0;
   P3DDRL |= 0x07;  //P30 input, P31 output
   TF2 = CLEAR; T2 = CLEAR;
   CKCON |= 0x20;      // Set timer 2 to clk/4
   RCAP2 = 0xffd9; //Set Timer 2 to Generate 57690 bps
   //Initialize TH2:TL2 so that next clock generates first Baud Rate pulse
   THL2=0xffff;
   T2CON = 0x34; // Set T2 for Serial0 Tx/Rx baud generation
   //SCON: Async mode 1, 8–bit UART, enable rcvr; TI=CLEAR, RI = CLEAR
   SCON  = 0x50;
   PCON |= 0x80; // Set SMOD0 for 16X baud rate clock
}
long read_accumulator_result ()
{
   /*Convert the concatenated Accumulate Result string ACCR3:ACCR2:ACCR1:ACCR0
   to a LONG integer*/
   long j;
   j = ACR3;
   j <<= 8;
   j += ACR2;
   j <<= 8;
   j += ACR1;
   j <<= 8;
   j += ACR0;

   return (j);
}
long read_a_to_d_result ()
{
   long j;

   /*Convert A/D Conversion results from the ADRESH:ADRESM:ADRESL
   register string to a LONG integer ith sign extension*/
   j = ADRESH;
   j <<= 8;
   j += ADRESM;
   j <<= 8;
   j += ADRESL;
   j &= 0x00ffffff; //eleminate upper nibble
   if (j & 0x00800000)
   {//is result negative?
      j |= 0x0ff000000;
   }
   return (j);
}

char init_a_to_d ()
{
   char i, j;

   /* Setup ADC */
// ADCON0 = 0x30;   // Vref on 2.5V, Buff on, BOD off
   ADCON0 = 0x20;   // Vref on 1.25V, Buff on, BOD off
   ADCON1 = 0X00;
   ADCON2 = 0xFF;   // decimation ratio
   ADCON3 = 0x00;
   ADCON1 = 0x05;   // bipolar, Filter = auto, self calibration, offset, gain
   //wait for the calibration to take place
   printf (”\n\nCalibrating....\n”);
   while(!(AISTAT & 0x20));
   j = ADRESL;

   for (i = 0; i < 20; i++)
   { // dump 20 conversions
      /*wait for DRBY bit*/
      while(!(AISTAT & 0x20));
   }
   /*set up Summation / Shifter*/
   /*Select Summation / Shifter option,
   Acc Count = 8, Shift Count = 8
   */
   init_accumulator ();
   //extract Accumulate–Count from SSCON SFR Register.
   j = SSCON & 0x38;
   j /= 8;
   j = 1 << (j + 1);
   return (j);
}

void a_to_d_accumulate (void) interrupt 6 using 1
{
   /*interrupt type 6 vectored to 0x33.
   Any AI type interrupt would come to this ISR.
   Evaluating the SUM and ADC bits of AISTAT will determine
   whether the ISR call was due to the A/D Converter interrupt or
   the Summation/Shifter interrupt*/
   if (AISTAT & 0x20)
   {//A/D conversion interrupt
      converting = 0;
      AISTAT &= ~0x20; /*clear ADC bit*/
   }
   if (AISTAT & 0x40)
   {//Acculullator/Shifter interrupt
      averaging = 0;
      AISTAT &= ~0x40; /*clear SUM bit*/
   }
   return;
}

void main(void)
{
   int i;
```

```
char accum_count;
long l;
float  voltage_value, vref, max_range;
char convert_accumulate;

convert_accumulate = 1; //Select data averaging option.

CKCON &= 0xf8; // 0 MOVX cycle stretch

//set Serial # 1 indow up for output display
setport ();
printf (”\nMSC1210 Ver:”);
printf (”\nA/D Res H/M/L\t”);
printf (”Acc Reg 3/2/1/0\n”);

//Enable global interrupt and enable Power Fail Interrupt
IE = 0x80;
EPFI = 1;

//initialize A/D converter and extract Accumulation–Count
accum_count = init_a_to_d ();

//Wait for conversion to be completed
while (!(AISTAT & 0x20));
//Conversion completed, then read result of the A/D converter
l = read_a_to_d_result ();

//set conversion constants max_range and vref
if (ADCON1 & 0x40) //is polarity unipolar or bipolar?
{//unipolar
   max_range = 0xFFFFFF;
}
else
{//bipolar
   max_range = 0x7FFFFF;
}

if (ADCON0 & 0x10) //is Vref = Vrefh or Vref = Vrefl?
{//Vrefh
   vref = 2.5;
}
else
{//Vrefl
   vref = 1.25;
   }

   switch (convert_accumulate)
   {
      case CONVERT: //straight A/D conversion results, no averaging
         PAI = 0x20;
         for (i = 0; i < 0x40; i++)
         {
            converting = 1;
            /*straight conversion idle loop.
            Value of ”converting” is changed in the a_to_d_accumulate ()
            ISR which is called at the end of each conversion.*/
            while (converting);
            /*Get LONG integer result and convert to a floating point
            voltage value using the proper values for vref and max_value*/
            l = read_a_to_d_result ();
            voltage_value = (float)l * vref / max_range;
            printf (”\nInstantaneous Value: %ld, i.e. %f volts”,
               l, voltage_value);
         }
         break;
      default:  //Averaged A/D conversion results
         PAI = 0x60;
         for (i = 0; i < 0x40; i++)
         {
            averaging = 1;
            /*averaged conversion idle loop.
            Value of ”averaging” is changed in the a_to_d_accumulate ()
            ISR which is called at the end of each completed
            averaging sequence.*/
            while (averaging);
            /*Get LONG integer result and convert to a floating point
            voltage value using the proper values for vref and max_value*/
            l = read_accumulator_result ();
            voltage_value = (float)l * vref / max_range;
            init_accumulator ();
            printf (”\nInstantaneous voltage Values ”);
            printf (”Averaged Over %d Samples: %ld,\n i.e. %f volts”,
               (int)accum_count, l, voltage_value);
         }
         break;
   }
   while (1);
}
```

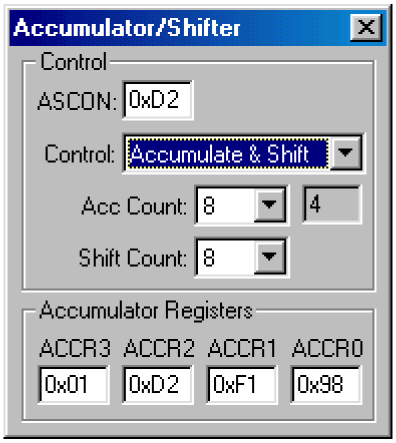
In order to demonstrate how the summation/shifter handles the incremental accumulation of sampled data, we have opted to enable both the ADC conversion interrupt enable, EADC, and the summation interrupt enable, ESUM, by assigning a value of 0x60 to the PIREG SFR. This implies that the power fail interrupt, AI, is pulsed each time the ADC completes a sample conversion on ADC, and each time the number of accumulation matches the acc_count value on SUM. This means if a breakpoint were inserted in the a_to_d_interrupt ISR routine for each data averaging cycle, eight samples in this case, the value displayed in the non-editable text window associated with acc_Count of the summation/shifter peripheral increase from 0 to 7 each time the ADC interrupt is pulsed. The data values in the summation registers vary as well, with the accumulation of conversion data results. On the eighth sample of the cycle, when the SUM interrupt has been pulsed, the value content of the non-editable display window rolls over from 7 to 0, and the contents of the summation registers will be properly adjusted for the result of the eight data sample averaging. Right after the averaged data has been successfully read, the summation registers must be reset to all zeroes so it can start the next batch of eight sample averaging with a clean slate. This can be most conveniently achieved by assigning a 0x00 value to the SSCON SFR. However, in this case, the contents of this register must be replenished with the previous value of 0xD2, in order to properly set the operating parameters for the summation/shifter module. This is equivalent to calling the init_accumulator subroutine. These processes are repeated 64 times, after which the simulator jumps into an infinite loop.

Note:

if we are not particularly interested in studying the individual data accumulation step, we can assign a value of 0x40 to the PIREG SFR. In this case, the AI interrupt ISR is called only when the SUM interrupt is triggered.

Snapshots of the summation/shifter peripheral and the ADC peripheral midstride a typical 8-sample averaging block are shown in Figure 17–11 and Figure 17–12. In the ADC Conversion peripheral, the AIN0 window shows that the input voltage value of the most recent ADC conversion is 1.399994V. This is a result of the 1.4V value set from the debugging script program. The 24-bit conversion of this AIN0 value is displayed in the editable window labeled ADRESH/M/L. The digital value of this conversion is 0x74B166. The non-editable text window associated with acc count shows that four out of eight samples have been accumulated in the summation registers, and thus far, the sum of all four accumulated 24-bit conversion values is 0x01D2F198.

Figure 17–11. summation/Shifter Peripheral



Accumulator/Shifter

Control

ASCON: 0xD2

Control: Accumulate & Shift

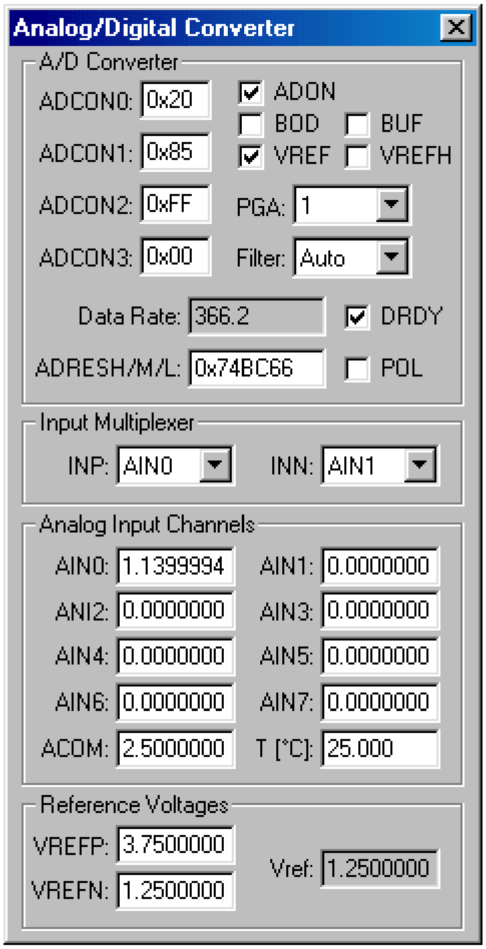
Acc Count: 8 4

Shift Count: 8

Accumulator Registers

ACCR3	ACCR2	ACCR1	ACCR0
0x01	0xD2	0xF1	0x98

Figure 17–12. The ADC Peripheral Mid-Stride a Typical 8-Sample Averaging Block



Analog/Digital Converter

A/D Converter

ADCON0: 0x20

ADCON1: 0x85

ADCON2: 0xFF

ADCON3: 0x00

☑ ADON

☐ BOD ☐ BUF

☑ VREF ☐ VREFH

PGA: 1

Filter: Auto

Data Rate: 366.2 ☑ DRDY

ADRESH/M/L: 0x74BC66 ☐ POL

Input Multiplexer

INP: AIN0 INN: AIN1

Analog Input Channels

AIN0: 1.1399994 AIN1: 0.0000000

ANI2: 0.0000000 AIN3: 0.0000000

AIN4: 0.0000000 AIN5: 0.0000000

AIN6: 0.0000000 AIN7: 0.0000000

ACOM: 2.5000000 T [°C]: 25.000

Reference Voltages

VREFP: 3.7500000

VREFN: 1.2500000

Vref: 1.2500000

In addition to the previous sample code, a sample driving code for the debugging is included below. This is a special feature of the µVision2 Simulator system that allows you to send input voltage values to the editable analog text fields in the ADC peripheral module. In the case of this example, the AIN0 input channel is selected. These input values are automatically written to the text window within a preprogrammed time interval, which is determined by the argument of the predefined function twatch (ulong states, where states is the unsigned long value of the number of CPU clock states that must elapse before the next statement in the program is executed). While this function is being executed, the debug system is placed in an idle state, and the target program continues executing. Upon expiration of this timer period (number of CPU clock states), the debugging process continues at the next statement. For all intents and purposes, the target process is oblivious to the execution or the state of the debugging program.

The debugging program is declared as one with a return value of type SIGNAL. The debugging variable AIN0 is assigned an initial value of 0.5V. This will be subsequently incremented by an incremental value of 0.01V. After the AIN0 value has been initialized, a delay of 196 000 CPU clock samples is imposed, while the main program, which started at the same time as this debug program, is performing its parameter initialization and book keeping until it is ready for the next data to be sampled. Within the For Loop, another 900 CPU clock delay is imposed, then the value of AIN0 is incremented by 0.01. It then processes another 130 100 CPU clock delay. This is repeated until the For Loop count expires. In the meantime, the AIN0 text window in the ADC peripheral is incrementally updated, and its value is periodically sampled, converted, averaged and displayed from within the main program.

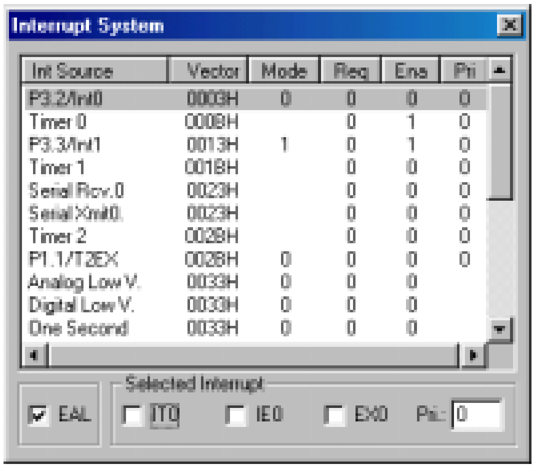
Please refer to the µVision2 Debug Functions chapter in the Keil's *Getting Started and Creative Programming* document for more information on writing, compiling and running script files.

```
SIGNAL void a_to_d_sim (void)
{
   int i;
   /*Data written into the variable ain0 is automatically entered into the
   editable text window labeled AIN0 in the A/D Converter peripheral dialog.*/
   ain0 = 0.5;  //specify start value for ain0
   //debug program idles for 196000 clock cycles, while simulation continues
running in parallel*/
   twatch (196000);

   /*the following loop sends out 64 consecutive samples of ain0,
   each incremented by 0.01. Each transmittal is spaced 131000 clock cycles
   from the previous one.*/
   for (i = 0; i < 0x40; i++)
   {
      twatch (900);
      ain0 += 0.01;
      twatch (130100);
   }
}
```

17.9 Interrupts

The list box for the interrupt peripheral is shown in Figure 17–13. The figure shows a list of interrupt sources along with their associated vector addresses that the processor automatically vectors to in the event that an enabled interrupt is triggered, there are no pending interrupt requests of higher priority, and there is no ISR being executed pertaining to an interrupt source of higher priority. As shown in Figure 17–13, the columns with headings Int Source, Vector, Mode, Req, Ena and Pri pertain, respectively, to the interrupt source name, the interrupt vector address, the interrupt edge type, (0 for level triggered and 1 for edge triggered), the interrupt request status, the interrupt enabled status and its priority level. There are some interrupt sources listed without any priority listing. All such sources have the same vector address as that of the AI interrupt. The priority levels of all interrupts affiliated with it are also unalterable and have the highest priority level setting because the AI has the highest, and an unalterable, priority.

Figure 17–13. List Box for the Interrupt Peripheral



Int Source	Vector	Mode	Req	Ena	Pri
P3.2/Int0	0003H	0	0	0	0
Timer 0	000BH		0	1	0
P3.3/Int1	0013H	1	0	1	0
Timer 1	001BH		0	0	0
Serial Rcv.0	0023H		0	0	0
Serial Xmit0.	0023H		0	0	0
Timer 2	002BH		0	0	0
P1.1/T2EX	002BH	0	0	0	0
Analog Low V.	0033H	0	0	0	
Digital Low V.	0033H	0	0	0	
One Second	0033H	0	0	0	

☑ EAL

Selected Interrupt

☐ IT0 ☐ IE0 ☐ EX0 Pri: 0

Selecting any of the itemized interrupt sources will force its pertinent associated settings and status value to be transferred to the set of check boxes in the lower part of the Interrupt display. Clicking on its corresponding check box could alter the status of each piece of information. On the Interrupt display shown in Figure 17–13, the set of check boxes in the lower section of the display indicate that the Global Enable flag EA is high (EA has a check). This implies that any interrupt source that is enabled has the potential to generate an interrupt request. If the EA button is clicked, clearing the selection, the processor's interrupts are globally disabled. Just for clarification, it should be restated that the status of EA has no bearing on the ability of a AI interrupt condition, or those of any of the peripheral interrupts tied to it, to initiate an interrupt request. Setting the EFPI bit of EICON enables the AI. One could induce an interrupt by placing a check on the corresponding Req slot of the desired interrupt source. This is equivalent to activating the interrupt through the software or the hardware.

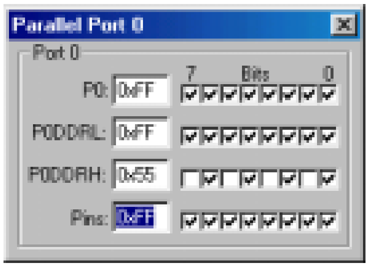
Sample routines of the interrupt peripheral module have been incorporated into sample programs for other peripheral modules that have been discussed previously. Please study those programs for more information.

17.10 Ports

There are four parallel I/O ports on this device, Port 0, Port 1, Port 2 and Port 3, and as such, there are four separate parallel port displays. We shall discuss the operation of just one I/O port display because all four of them are similar.

The Parallel Port 0 shown in Figure 17–14 depicts the value and the bit pattern of the contents on the Port 0 register (P0), the Port 0 Data Direction High register P0DDRH, and the Port 0 Data Direction Low register P0DDRL. In addition, the byte value and the bit pattern of the logic levels of the signals on the Port 0 I/O pins are also depicted. The value of any of these registers or I/O pin settings can be changed by changing either the corresponding byte value or bit pattern.

Figure 17–14. Parallel Port 0 Contents Display Window



P0: 0xFF ☑☑☑☑☑☑☑☑

P0DDRL: 0xFF ☑☑☑☑☑☑☑☑

P0DDRH: 0x55 ☐☑☐☑☐☑☐☑

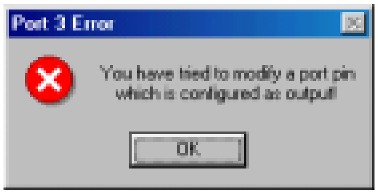
Pins: 0xFF ☑☑☑☑☑☑☑☑

For example, the bit pattern for P0DDRH could have been set either by writing a value of 0x55 into the editable text input window for P0DDRH, or clicking once on checked bit pattern toggle switches for bits 7, 5, 3, and 1, in any sequence. This, by the way, sets the Port 0 pins as inputs for port pins 0, 1, 2 and 3, and strong driver outputs for pins 4, 5, 6 and 7. Byte values of 0x55 and 0xFF could just as well have been written into the P0DDRH and P0DDRL registers respectively, through the software program, for the same effect.

Until a port read is performed, the value in the port register (P0 for instance) does not necessarily reflect the status of the port pins.

The Keil Simulator also has facilities for error detection and warning. If, for instance, we configure the upper nibble of Port 0 for inputs and the lower nibble for outputs, trying to toggle the $\overline{\text{INT1}}$ pin (P3.3) in order to simulate an interrupt trigger results in an error message, as shown in Figure 17–15). This is because pin #3 of the Port 0 is configured for output, and we are trying to drive it as an input.

Figure 17–15. Error Message



You have tried to modify a port pin which is configured as output!

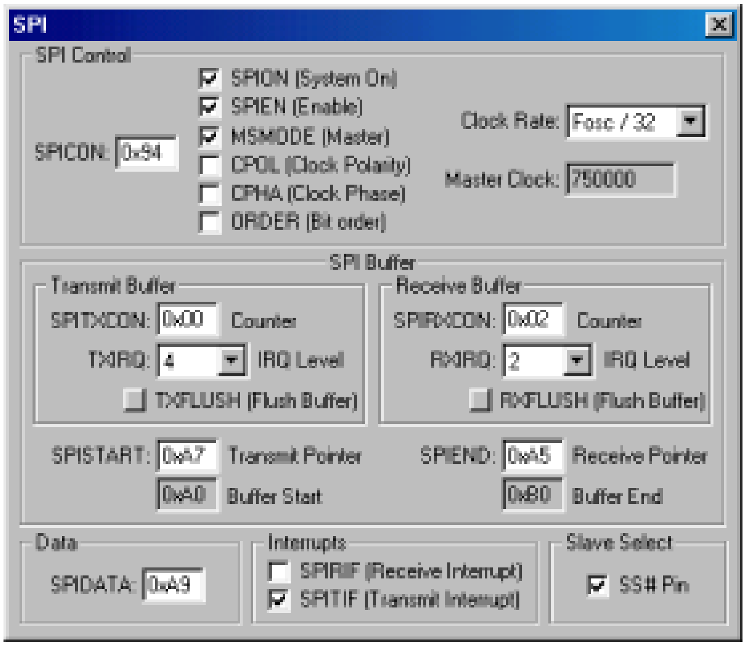
Sample routines of the I/O Port Peripheral module have been incorporated into sample programs for other peripheral modules that have been discussed earlier. Please study those programs for more information.

17.11 Serial Peripheral Interface (SPI)

The serial peripheral interface (SPI) implemented in this simulator package mimics the behavior and characteristics of a data memory access (DMA) SPI module, integrated into the MSC1210. The MSC1210 SPI module is an enhanced version of the popular SPI modules implemented by other manufacturers. Its enhancement involves substituting the single buffering on the transmit and receive ends with an adjustable depth First in first out (FIFO) circular buffer system which has all the signaling characteristics and observes all the data collection protocols of a typical DMA system. With this DMA enhancement, under normal operating conditions, if the SPI circular buffer is deep enough, the likelihood of a data overflow is virtually eliminated.

The snapshot in Figure 17–16 shows the freeze-framed picture of the SPI peripheral window in the middle of a typical data communication transmit/receive session.

Figure 17–16. SPI Peripheral Window



SPICON: 0x94
☑ SPION (System On)
☑ SPIEN (Enable)
☑ MSMODE (Master)
☐ CPOL (Clock Polarity)
☐ CPHA (Clock Phase)
☐ ORDER (Bit order)
Clock Rate: Fosc / 32
Master Clock: 750000

SPITXCON: 0x00 Counter
TXIRQ: 4 IRQ Level
☐ TXFLUSH (Flush Buffer)

SPIRXCON: 0x02 Counter
RXIRQ: 2 IRQ Level
☐ RXFLUSH (Flush Buffer)

SPISTART: 0xA7 Transmit Pointer
0xA0 Buffer Start
SPIEND: 0xA5 Receive Pointer
0xB0 Buffer End

SPIDATA: 0xA9
☐ SPIRIF (Receive Interrupt)
☑ SPITIF (Transmit Interrupt)
☑ SS# Pin

Like the other peripheral modules, pertinent SFRs could be programmed or updated by writing the appropriate data into the associated editable text window, or by placing or removing check marks from corresponding check boxes. SFR names and individual bit names are also preserved between the actual MSC1210 SPI module and the simulator SPI peripheral module.

Entries made into the editable SPICON text window will set or clear the check marks of the SPI enable (SPIEN), master (MSMODE), clock polarity (CPOL), clock phase (CPHA), bit order (ORDER) check boxes, and the corresponding divide by selection from the Clock Rate selection window. Conversely, clearing or setting the check mark on any of the check boxes, will change the value of data displayed in the SPICON window, on the basis of the bit position of the corresponding configuration bit within the SFR bit pattern. Likewise, changing the clock rate divide by value changes the SPICON entry accordingly. The result of the oscillator frequency divided by the selected divide by factor is displayed in the non-editable master clock window.

The trigger level for the transmit IRQ and receive IRQ are selectable through the selection windows marked IRQ Level within the transmit buffer and receive buffer blocks, respectively. The number of data bytes currently in the circular FIFO buffer (waiting to be transmitted or already received and waiting to be read) are displayed in the SPITCON and SPIRCON windows, respectively.

Clicking on the transmit flush buffer box, TXFLUSH, destroys the bytes of data within the circular buffer that are waiting to be transmitted. It changes the SPI transmit pointer so that it points to the same address as the FIFO OUT pointer, and clears the transmit counter value within the SPITCON SFR. The transmit counter indicates the number of bytes within the circular buffer that are waiting to be transmitted. Similarly, clicking on the receive flush buffer RXFLUSH box, destroys the bytes of data, within the circular buffer, that have already been received, but still waiting to be read by the processor. It forces the receive pointer to point to the same address location as the FIFO IN pointer, and it resets the receive counter value within the SPIRCON SFR. The receive counter indicates the number of bytes within the circular buffer that are waiting to be read by the processor.

It is worth stressing that all this window text editing and check box marking are just alternative methods of programming the various SFRs in software.

The circular buffer can be set or redefined, by writing the desired value into the editable transmit pointer window (SPISTART) and the editable receive buffer window (SPIEND). You will observe that whatever entry is made into the SPISTART text window also appears in the non-editable text window labeled Buffer Start, and the editable text window labeled SPIEND. The content of the non-editable text window labeled Buffer End remains unchanged. Writing the desired value for the other end of the circular buffer causes the entered value to appear in the Buffer End display window, but the data displayed in the SPIEND window automatically reverts to the value written into the SPISTART window. So, before data communication begins, SPISTART and SPIEND must contain the same value, implying that the buffer is empty, whereas the Buffer Start and Buffer End display the boundaries for the DMA circular buffer. Although the contents of the SPISTART and SPIEND windows will change as data is written into and transmitted out of the transmit buffer, and data is received into and read from the receive buffer, the contents of the non-editable text windows Buffer Start and Buffer End will not change. Notice that flushing the transmit buffer will not affect the contents of the SPISTART window; it seems as though the transmit data was never written into the buffer. Flushing the receive buffer will neither affect the contents of the SPISTART window nor those of the SPIEND window.

Data written into the editable text window SPIDATA will be handled as a piece of data to be transmitted. The SPITCON and SPISTART windows will be properly updated, the circular buffer entry will be made into the appropriate buffer address pointed to by the transmit pointer. The values in the SPITCON window and the TXIRQ window determine the check/clear status of the SPIT check box. As data is being received from the external device, the value of the received data will be momentarily displayed in the SPIDATA window, and the content of SPIRCON window is properly updated. In addition, as data is read from the circular buffer, the value displayed in the SPIEND windows is properly updated. The status of the SPIR bit is also updated on the basis of the values displayed in the SPIRCON and RXIRQ windows.

A simple code example for a typical SPI communication exercise is appended.

17.11.1 SPI Sample Code

The following program simulates the data communication interaction between two devices with SPI capabilities, where one operates in the master mode and the other in the slave mode. Like the other example covered so far, a C style program script was written using the µVision 2's debug functions protocol. This program runs in parallel to the main program. The main program is set up as the master, and the µVision 2's debug functions package is set up as a slave.

The various SFRs that are pertinent to the SPI module are enabled and initialized. The SPI peripheral is asserted as the master, and the communication speed is specified. The receive and transmit buffers are flushed, and IRQ levels of four and two are specified for the transmit and receive sections, respectively. The limits of the circular buffer are defined as 0x0A and 0x0B. Finally, the SPI transmit and receive interrupts are enabled, and the processor is globally interrupt enabled.

After having properly set up the I/O system for the Serial #1 window, and initializing the interrupt enables and the SPI Communication system, this program sends out a dummy data byte to start up the communication. The processor then enters an infinite loop that is interrupted anytime there is an SPI transmit or receive interrupt.

The SPIT flag is asserted whenever the transmitter IRQ level limit is not attained, and the SPIR flag is asserted when the receive IRQ level is exceeded. Either condition will generate a AI type interrupt. The transmit_receive ( ) ISR is called whenever this interrupt is acknowledged. As discussed earlier, the processor vectors to address, 0x33, from which a long jump instruction is executed. The processor branches to the appropriate section of the ISR routine on the basis of the value contained in the AISTAT SFR. If the interrupt was caused by triggering the transmit flag, SPIT, the processor branches into the transmit block of the ISR. If the interrupt was caused by the receive flag, SPIR, the receive block will be selected. Please refer to the chapter on SPI communication in this manual.

Within the receive block of the ISR, the processor reads the contents of the receive section of the circular buffer by reading the SPIDATA buffer continuously until the SPIRCON count expired. The ISR resets the SPIR interrupt flag.

Within the transmit block of the ISR, the processor increments the value of the static integer variable j and transmits its new value by writing it into the SPIDATA SFR. The ISR resets the SPIT interrupt flag.

Upon completion of the associated ISR routine block, the process returns to the infinite loop.

```
#include ”MSC1210.H”
//unsigned char data irqen_init _at_ 0x7f ; // image of PAI
#define FWVer 0x04
#define CONVERT 0

char   received_data[50];

void init_spi ();
void transmit_receive ();// interrupt 6;
void test_spi ();

void test_spi ()
{
   SPIDATA = 55;
   while (1);
}
void setport (void)
{
   P3DDRL &= 0xf0;
   P3DDRL |= 0x07;  //P30 input, P31 output
   TF2 = CLEAR; T2 = CLEAR;
   CKCON |= 0x20;      // Set timer 2 to clk/4
   RCAP2 = 0xffd9; //Set Timer 2 to Generate 57690 bps
   //Initialize TH2:TL2 so that next clock generates first Baud Rate pulse
   THL2=0xffff;
   T2CON = 0x34; // Set T2 for Serial0 Tx/Rx baud generation
   //SCON: Async mode 1, 8–bit UART, enable rcvr; TI=CLEAR, RI = CLEAR
   SCON  = 0x50;
   PCON |= 0x80; // Set SMOD0 for 16X baud rate clock
}
void init_spi ()
{
   /*enable SPI, specify Master SPI and specify clock rate, (Fosc / 32)
   and CPHA = 1*/
   SPICON = 0x96;
   /*Flush receive buffer, and set IRQ level to 2*/
   SPIRCON = 0x81;
   /*Flush transmit buffer, and set IRQ level to 4*/
   SPITCON = 0x82;
   /*buffer start address = 0x0ao, and end address = 0x0b0*/
   SPISTRT = 0x0a0;
   SPIEND = 0x0b0;
   /*Master mode: set MISO for input, and MOSI, SS & SCK for strong outputs*/
   P1DDRH = 0x75;
   P1 |= 0xF0;
   /*enable SPI interrupt*/
   PIREG |= 0x0c;
   IE |= 0x80;
   EPFI = 1;
}
void transmit_receive () interrupt 6 using 1
{
   /*This is a type 6 interrupt (AI). Processor vectors to 0x33,
   from which it is redirected to this ISR. Because both SPI transmit
   and SPI receive interrupts are enabled, additional steps must be
   taken to differentiate between transmit and receive interrupt requests.
   Hence, the ”if (AISTAT ...)” statements.*/
   int i, k;//, p;
   static int   j, l;
```

Each time the transmit block is selected; the value of the static integer j is incremented by two, and written into the SPIDATA SFR. This automatically advances the transmit pointer value, and places the value of j into the circular buffer. The count in the SPITCON SFR is incremented. Note that the SPITCON count will decrement automatically as soon as a byte has been successfully transmitted. Finally, the SPIT bit in the AISTAT SFR is cleared before exiting out of this block.

```
   if (AISTAT & 0x08)
   {/*Transmitter*/
      j += 2;   //set up value of j to be transmitted
      SPIDATA = j; //transmit j. This actually goes to the circular buffer
      AISTAT &= ~0x08; //clear SPI transmit interrupt flag
   }
```

The static integer variable i is checked to make sure that the array limits for the received_data array of characters is not exceeded. If the limit has been reached, the processor would print out the contents of the array to the Serial #1 window, and then reset the value for l to 0.

You must mask out the MSB of this SFR in order to extract the number of bytes because the SPIRCON SFR contains both the number of data bytes available in the circular buffer for retrieval and the RXFLUSH bit. When the processor reads a byte from the SPIDATA SFR, the oldest item in the current batch of received data bytes is read, and the SPIDATA will point to the next item in the batch. This implies that in order to read the batch of received data bytes that triggered the current interrupt, you could just read the SPIDATA SFR SPIRCON & 0x7F times with a FOR loop, and wait for the next time the interrupt is triggered.

```
   if (AISTAT & 0x04)
   {/*Receiver*/
      i = SPIRCON & 0x7F; //extract the count for the number of
      AISTAT &= ~0x04; //deactivate SPI receive flag
      if (l >= 50)
      {
         /*do not exceed the 50 received_data[] array limit*/
         for (l = 0; l < 5; l++)
         {
            printf (”\n”);
            for (k = 0; k < 10; k++)
            {
               printf (” %c”, received_data[k + l * 10] );
            }
         }
         printf (”\n”);
         l=0;   //reset the received_data[] array index.
      }
      for (k = 0; k < i; k++)
      {
         if (l >= 50)
            break;
         /*data received through the SPI channel is read at the SPIDATA SFR*/
         received_data[l++] = SPIDATA; //keep track of received data
      }
   }
}
void main(void)
{
   setport ();
   init_spi ();
   test_spi ();
}
```

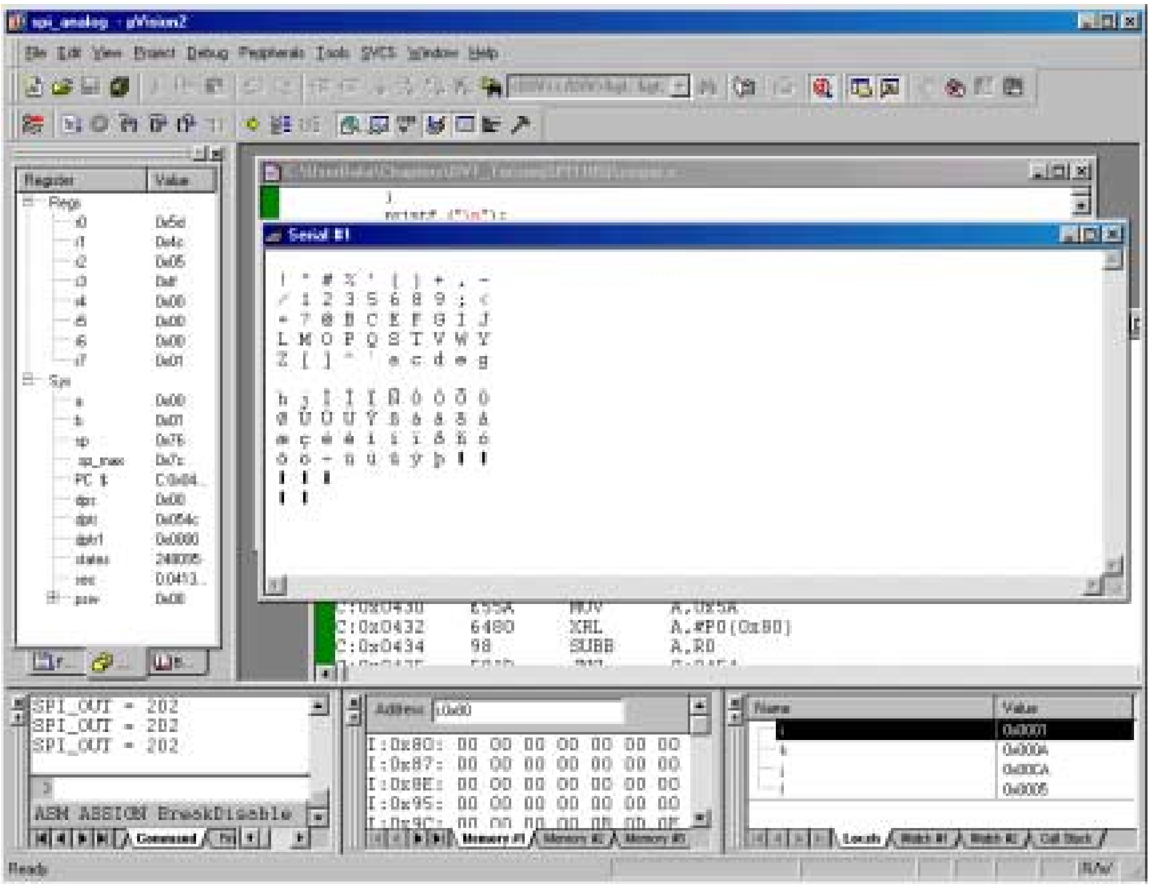
In addition to the main simulation program, a μVision 2 debugging program was also written to supply the received data to the test simulation program. This μVision program transmits and receives data at periodic intervals from the main program. The SPI communication protocol is used for this data transfer. The μVersion 2 debug program behaves as the slave, while the main program is the master. The debug program is presented and explained in the following section.

17.12 μVision 2 Debug Program Example

```
SIGNAL void spi_sim (void)
{
   /*This program runs in parallel with the main program.
   It sends out a character byte whose value is post incremented at the end of
   each associated time lapse.
   SPI_IN is the portal through which the byte data is sent to the main program.
   In addition, the data transmitted from the main program is received at the
   portal, SPI_OUT*/
   int j;
   j = 0x21; //initialize byte data value to be transmitted
   spi_in = j;  //send byte data
   twatch (100);//idle 100 clock cycles
   while (1) //start infinite loop
   {
      twatch (50);
      j++;  //increment value of byte data to be transmitted
      spi_in = j;  //transmit another byte of data
      twatch (97); //wait 97 clock cycles
      /*data transmitted from main program has been receive in portal SPI_OUT
      automatically. Its value is displayed in the Command Line display area*/
      printf (”\nSPI_OUT = %d”, spi_out);
      j++;
      /*send another incremented data byte, and receive and display a new data
byte transmitted from main.*/
      spi_in = j;
      twatch (116);
      printf (”\nSPI_OUT = %d”, spi_out);
   }
}
```

The data received from the μVersion 2 debug program, by the main SPI program is written to the Serial #1 window. A snapshot of this window is included in Figure 17–17.

Figure 17–17. Keil Debugger



```
! " # % ' ( ) + , -
/ 1 2 3 5 6 8 9 ; <
= ? @ B C E F G I J
L M O P Q S T V W Y
Z [ ] ^ ` a c d e g
```

```
C:0x0430    E55A    MOV     A,0x5A
C:0x0432    6480    XRL     A,#P0(0x80)
C:0x0434    98      SUBB    A,R0
```

```
SPI_OUT = 202
SPI_OUT = 202
SPI_OUT = 202
```

```
ASM ASSIGN BreakDisable
```

```
I:0x80: 00 00 00 00 00 00 00
I:0x87: 00 00 00 00 00 00 00
I:0x8E: 00 00 00 00 00 00 00
I:0x95: 00 00 00 00 00 00 00
```

The window labeled Serial #1 shows the printed ASCII character representation of the data bytes received by the main SPI program from the debugging program. Note that the first character printer is an ! mark which has a numerical value of 0x21. This was the first value of j to be transmitted from the debugger through the SPI_IN portal. The subsequent characters are supposed to correspond to ASCII characters whose numerical values are a value of one off from the previous character. However, once in a while, there is a skip in characters. This can be explained by the fact that once in a while, the two programs become unsynchronized. Better twatch delay timing would have resolved this issue, but that is not the essence of this example. The Serial #1 window shows the results of two 50-byte transfers from the µVersion 2 debug program.

Note:

Even though the µVersion 2 debug program and the main SPI program are separate and independent programs, they run in parallel, and they are synchronized. This way, the incoming data from the debugger is always ready when the main SPI program is ready to receive data. The delay timing computation is very delicate. If it is too short, data to be transmitted from the debugging program will be overwritten before it is transmitted. If it is too long, the data transmitted from the main SPI program will be overwritten before the debugging program reads it.

17.13 Serial Port I/O

In addition to the SPI communication protocol that was presented earlier in this manual, the more basic serial port I/O was also implemented in this simulator. Serial Ports 0 and 1 are simulated in this package. An example of this communication protocol has been used a couple of times in this section of the manual. The show_baud_gen ( ) subroutine is used to set up the output display of programming example results on the Serial #1 window. The show_baud_gen ( ) subroutine is described following.

Parallel port P3 is set so that P3.0 is an input pin, and P3.1 is an output pin. Referring to the chapter on Parallel ports will reveal that port pins 0 and 1 are also alternate pins for Serial Port 0 receive and Serial Port 0 transmit, respectively, for Serial Port 0 in either modes 1, 2, or 3. In Serial Port Mode 0, P3.0 is the bidirectional data transfer pin for Serial Port 0, and P3.1 emits the synchronizing clock for serial port 0 communication.

The Timer 2 timer overflow flag is cleared, in order to remove any preexisting Timer 2 interrupt request. The Timer 2 external input is also set to 0. Timer 2 in baud rate generation mode generates the data communication baud rate. On the basis of the f_{OSC} divide-by-4 selection made by setting the Timer 2 Clock Select bit (bit T2M of the CKCON SFR) and the oscillator clock frequency, the auto-reload value for the Timer 2 Capture pair, RCAP2H:RCAP2L (RCAP2), required to produce a baud rate of 37 500bps was computed to be 0xFFC4. Presetting the Timer 2 register pair TH2:TL2 (THL2) to 0xFFFF ensures that Timer 2 generates an overflow on the first f_{OSC} divide-by-4 clock. This automatically generates an overflow pulse, which is further divided by 16 to drive the Rx and/or Tx clocks. In addition, upon overflow, the contents of the RCAP2 register pair are automatically transferred into the THL2 register pair, which would have just rolled over to 0x0000. Note that in this mode, Timer 2 does not generate an overflow interrupt signal. Please refer to Section 8.5, *Timer 2*, for more information.

The timer overflow pulse for Timer 2 is used to generate baud rate clock for the transmit block, and Timer 1 overflow is used for the receive block. Setting the Timer 2 run control bit through T2CON also enables the Timer 2 clock. For this operation, the following options were selected for Timer 2: auto-reload, timer option, specify timer overflow pulse as baud rate clock for the transmit block and not for the receive block, and specify the option to ignore all external events on the T2EX (P1.1) pin.

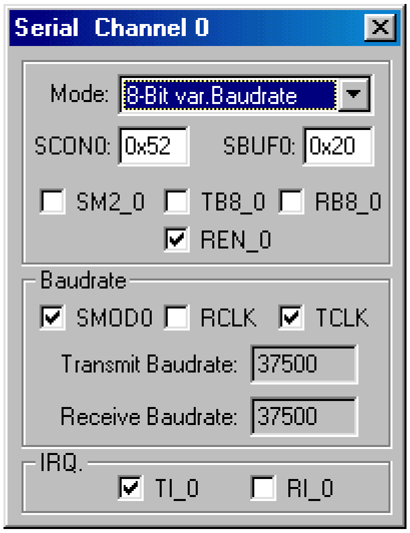
The bit pattern for the previous specifications requires that TCON2 be assigned a value of 0x14.

For the serial communication, Serial Port 0 was set for an asynchronous 10-bit (1 start bit, 8 data bits and one stop bit) mode 1, serial data communication operation. The serial port 0 is also receive enabled. These were accomplished by setting the SCON0 SFR to 0x50.

The baud rate doubling option 16X was selected by setting the SMOD0 bit of the PCON SFR.

A snapshot of the Serial Channel 0 communication peripheral after at typical show_baud_gen ( ) subroutines execution is shown in Figure 17–18.

Figure 17–18. Serial Channel 0 Communication Peripheral



Mode: 8-Bit var.Baudrate

SCON0: 0x52 SBUF0: 0x20

☐ SM2_0 ☐ TB8_0 ☐ RB8_0
☑ REN_0

☑ SMOD0 ☐ RCLK ☑ TCLK

Transmit Baudrate: 37500

Receive Baudrate: 37500

☑ TI_0 ☐ RI_0

The statuses of the transmit and/or receive flags are also reflected in the conditions of the TI_0 and RI_0 check boxes. The computed transmit and receive baudrates are displayed in the transmit baud rate and the receive baud rate non-editable windows respectively.

Note:

The transmit baudrate and the receive baudrate do not necessarily have to come from the same timer overflow source. You have a choice of two independent sources of the divide-by-16 transmit/receive counters. Setting or clearing the bit fields for RCLK and TCLK of the T2CON SFR, respectively, determine independently, whether the timer overflow source for the divide-by-16 transmit/receive counters is the Timer 1 overflow or the Timer 2 overflow.

The following section is a graft of the show_baud_gen ( ) subroutine used in earlier examples.

17.13.1 Serial Port 0 Operation Mode 1 Example

```
void show_baud_gen (void)
{
   P3DDRL &= 0xf0;
   P3DDRL |= 0x07;  //P30 input, P31 output
   TF2 = CLEAR; T2 = CLEAR;
   CKCON |= 0x30;      // Set timer 2 to clk/4
   RCAP2 = 0xFF16; //37500 bps
   THL2=0xFFFF;
   /* Set T2 for Serial0 Tx baudgen.
   Timer 2 is designated the clock source for the ”divide by 16” clock
   for the Transmit block, while Timer 1 is the implied source for the
   ”divide by 16” clock for the Receive block.
   TR2 is activated*/
   T2CON = 0x14;
   //SCON: Async mode 1, 8–bit UART, enable rcvr; TI=CLEAR, RI = CLEAR
   SCON = 0x50;
   PCON |= 0x80; // Set SMOD0 for 16X baud rate clock
   //set Timer 1 up for Rx Baud Rate Generation @ 37500 bps
   TH1 = 0xF6;
   /*Make the Timer 1 clocking Gated. This implies that for the Timer 1 to
   run, both TR1 and INT1# must be set.*/
   TMOD = 0xA0;
   TCON = 0x48;
// TI=SET;
}
```

17.13.2 Transmit Block Baud Rate Computation

In this example, two different baud rate sources have been used, one for receive, Timer 1 overflow, and the other for transmit, Timer 2 overflow. Of course, there is no good reason for this, except to show that it could be done, and to show how to use different timer modes for baud rate generation. The analyses for operational parameters for the individual timer overflow sources are described in the following paragraphs.

For the transmit baud rate generation, based on the SFR settings for the Timer 2 simulator peripheral, and the Timer 2 baud rate generator formulas outlined in the timer section of this manual, the communication baud rate computes as follows:

If the TM2 bit of CKCON is 0, then clock divide is $f_{OSC}/12$.

$$BaudRate = \frac{f_{OSC}}{2 \cdot 16 \cdot (0x10000 - RCAP2)}$$

RCAP2 is a concatenation of the SFR pair RCAP2H:RCAP2L. In this example, it carries a value of 0xFFC4. Hence, the generated baud rate works out to be 12 500bps.

If the faster clocking option of $f_{OSC}/4$ was selected, then this equation must be modified to accommodate this faster operation.

If the TM2 bit of CKCON is 0, then clock divide is $f_{OSC}/4$.

$$BaudRate = \frac{f_{OSC} \cdot 3}{2 \cdot 16 \cdot (0x10000 - RCAP2)}$$

In this case, the generated baud rate computes to be 37 500bps. The factor of three is a result of the fact that the divide-by-4 factor option was selected, as opposed to the divide-by-12 option.

Of course, these baud rate values scale proportionally with the selected value for f_{OSC}.

17.13.3 Receive Block Baud Rate Computation

Timer 1 is set for a mode 2 timer operation in an 8-bit auto-reload capacity. This is achieved by assigning a 0x20 value to the TMOD SFR. Recall that the SMOD0 bit of PCON has been set in an earlier section of this program. This effectively doubles the communication baud rate. Based on the baud rate computation formulas, it was determined that the desired 12 500 or 37 500 baud rate settings could be attained by assigning a value of 0xFB to the TH1 SFR.

If the TM2 bit of CKCON is 0, then clock divide is $f_{OSC}/12$.

$$BaudRate = \frac{2^{SMOD} \cdot f_{OSC}}{32 \cdot 12 \cdot (256 - TH1)}$$

And if the TM2 bit of CKCON is 1, then clock divide is $f_{OSC}/4$

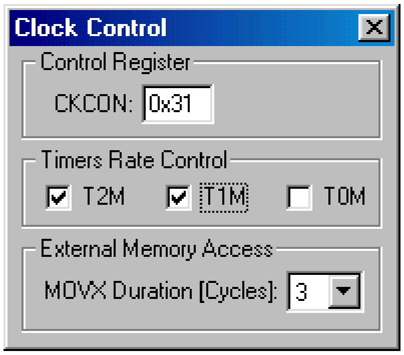
$$BaudRate = \frac{2^{SMOD} \cdot f_{OSC}}{32 \cdot 4 \cdot (256 - TH1)}$$

This forces a factor of two in the numerator in either case because SMOD carries a value of one. For the case in which the T1M bit of CKCON is cleared, the value of 12 in the denominator is a consequence of the $f_{OSC}/12$ option selected, whereas the factor of 4 in the denominator of the second expression is a consequence of choosing the $f_{OSC}/4$ option. With the value for TH1 set at 0xFB, the first expression results in a baud rate of 12 500bps, while the second expression results in a baud rate value of 37 500bps. Whatever the case may be, the correct value of the transmit and/or receive baud rates are properly reflected in the non-editable transmit baud rate and receive baud rate windows, respectively.

A value of 0x48 is assigned to the TCON SFR. This sets its TR1 (Timer 1 run) bit and its $\overline{\text{INT1}}$ bit. Actually, in this example, because we are not gating Timer 1, the status of $\overline{\text{INT1}}$ is irrelevant.

If the TR1 check box is cleared—setting the TR1 bit of TCON to 0—Timer 1 stops running, and the receive baud rate value becomes 0.

Figure 17–19. Clock Control Peripheral

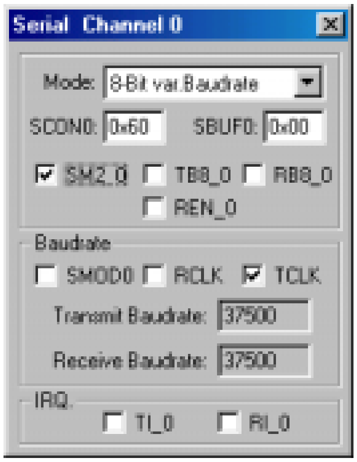


CKCON: 0x31

☑ T2M ☑ T1M ☐ T0M

MOVX Duration [Cycles]: 3

Figure 17–20. USART0 Preipheral



Mode: 8-Bit var.Baudrate

SCON0: 0x60 SBUF0: 0x00

☑ SM2_0 ☐ TB8_0 ☐ RB8_0
☐ REN_0

☐ SMOD0 ☐ RCLK ☑ TCLK

Transmit Baudrate: 37500

Receive Baudrate: 37500

☐ TI_0 ☐ RI_0

17.14 Additional Resource

It is highly recommended that you review the Keil Compiler tutorial integrated into this package for an animated demonstration of some useful IDE facilities.

Appendix A

Additional Features in the MSC1210 Compared to the 8052

Appendix A deals with additional features found in the MSC1210 as compared to the 8052.

Topic **Page**

A.1 Additional Features in the MSC1210 Compared to 8052

The MSC1210 includes the following features in addition to those that are included in a standard 8052 microcontroller.

- Flash memory, up to 32k partitionable as program and/or data memory.
- Low-voltage/brownout detection.
- High-speed core: 4 clocks per instruction cycle.
- Dual data pointers (DPTR).
- 1280 bytes on-chip SRAM (256 bytes internal RAM, 1024 bytes addressable as external RAM)
- 2k boot ROM
- 32-bit accumulator
- Watchdog timer
- Master/Slave SPI with DMA
- 16-bit PWM
- 24-bit ADC

Appendix B

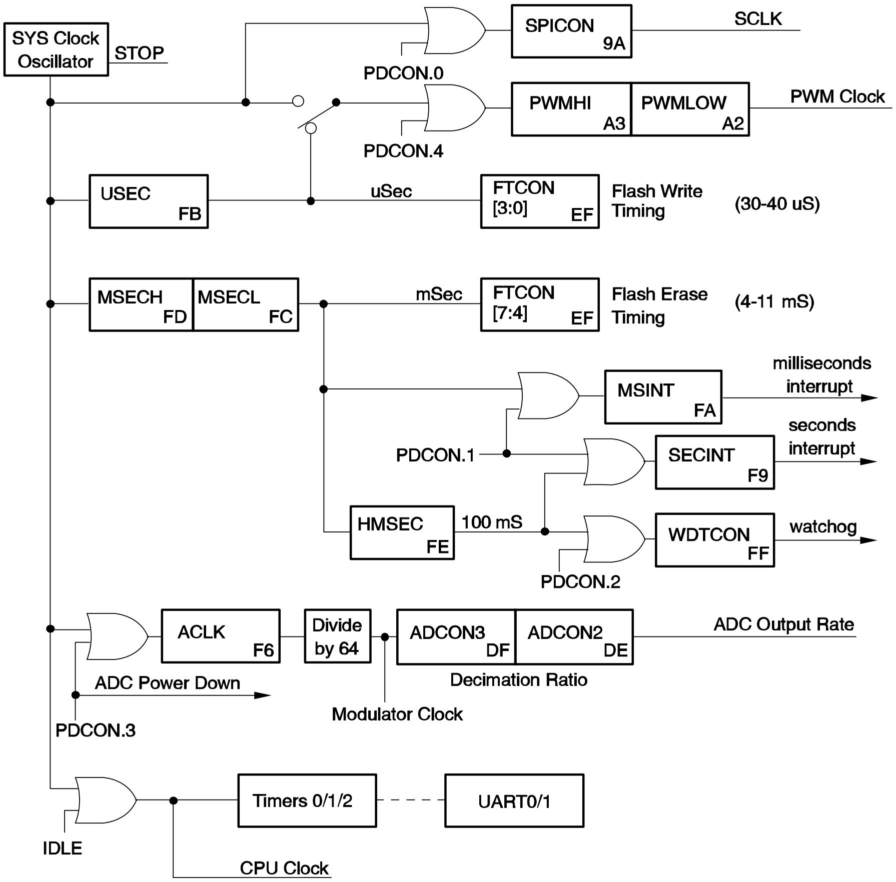
Clock Timing Diagram

Appendix B diagrams the MSC1210 ADC timing chain and clock control.

Topic **Page**

B.1 MSC1210 Timing Chain and Clock Control Diagram

Figure B–1. MSC1210 Timing Chain and Clock Control



Appendix C

Boot ROM Routines

Appendix C defines the MSC1210 ADC boot ROM routines.

Topic **Page**

C.1 Description

The MSC1210 has a 2K ROM. This code provides the interaction for serial and parallel programming. There are also several routines that are useful and necessary for use with user applications. For example, when writing to flash memory, the code cannot execute out of flash memory. By calling the flash write routine in ROM, this condition is satisfied. Convenient access to those routines is supplied through a jump table summarized in Table C–1.

Table C–1. Boot ROM Routines

Address	Routine	C Declarations	Description
FFD5	put_string	void put_string(char code *string);	Output string (see Section C.1.1)
FFD7	page_erase	char page_erase (int fadd, char fdat, char fdm);	Erase flash page
FFD9	write_flash	Assembly only; DPTR = address, R5 = data	Fast flash write
FFDB	write_flash_chk	char write_flash_chk (int fadd, char fdat, char fdm);	Write flash byte, verify
FFDD	write_flash_byte	char write_flash_byte (int fadd, char fdat, char fdm);	Write flash byte
FFDF	faddr_data_read	char faddr_data_read(char faddr);	Read HW config byte from address
FFE1	data_x_c_read	char data_x_c_read(int faddr, char fdm);	Read xdata or code byte
FFE3	tx_byte	void tx_byte(char);	Send byte to UART0
FFE5	tx_hex	void tx_hex(char);	Send hex value to UART0
FFE7	putok	void putok(void);	Send OK to UART0
FFE9	rx_byte	char rx_byte(void);	Read byte from UART0
FFEB	rx_byte_echo	char rx_byte_echo(void);	Read and echo byte on UART0
FFED	rx_hex_echo	char rx_hex_echo(void);	Read and echo hex on UART0
FFEF	rx_hex_int_echo	Int rx_hex_int_echo(void);	Read int as hex and echo: UART0
FFF1	rx_hex_rev_echo	Int rx_hex_rev_echo(void);	Read int reversed as hex and echo: UART0
FFF3	autobaud	void autobaud(void);	Set baud with received CR
FFF5	putspace4	void putspace4(void);	Output 4 spaces to UART0
FFF7	putspace3	void putspace3(void);	Output 3 spaces to UART0
FFF9	putspace2	void putspace2(void);	Output 2 spaces to UART0
FFFB	putspace1	void putspace1(void);	Output 1 space to UART0
FFFD	putcr	void putcr(void);	Output CR, LF to UART0
F97D(1)	cmd_parse	void cmd_parser(void)	See SBAA076B.pdf
FD3B(1)	monitor_isr	void monitor_isr() interrupt 6	Push registers and call cmd_parser

Note: 1) These addresses only relate to version 1.0 of the MSC1210 Boot ROM.

C parameters are passed to the subroutine code such that the first parameter is passed in R7, whereas additional parameters use lower R registers (R7 first, then R6, R5, etc.). In the case of multibyte parameters, the low byte uses the next available R register while the high byte uses the lower R register. Thus, the put_string routine uses R7 to receive the low byte of the address of the string while R6 is used to receive the high byte of the address of the string.

The result or error code is returned in R7 and/or R6, with the low byte in R7 and the high byte, if any, in R6.

C.1.1 Note Regarding the put_string Function

The put_string routine was designed to print strings that are referenced when the boot ROM is located at 0x0000 and also at 0xF800. This means that it forces the location of the string to match the same 2K segment the program is located in. This will lead to strange behavior if the string address is located in a different 2K segment. For this reason it is suggested that you use the following code instead:

```
void putstring(char code * data msg)

{

   while (*msg != 0)

         {

         tx_byte((unsigned char) *msg);

         if ( *msg==\n)

               tx_byte('\r');

         msg++

   }

}
```

Appendix D

8052 Instruction-Set Quick-Reference Guide

Appendix D gives a list of the 8052 instruction set.

Topic **Page**

D.1 8052 Instruction-Set Quick-Reference Guide

00	NOP	40	JC *relAddr*	80	SJMP *relAddr*	C0	PUSH *direct*
01	AJMP *pg0Addr*	41	AJMP *pg2Addr*	81	AJMP *pg4Addr*	C1	AJMP *pg6Addr*
02	LJMP *addr16*	42	ORL *direct*,A	82	ANL C,*bitAddr*	C2	CLR *bitAddr*
03	RR A	43	ORL *direct*,*#data8*	83	MOVC A,@A+PC	C3	CLR C
04	INC A	44	ORL A,*#data8*	84	DIV AB	C4	SWAP A
05	INC *direct*	45	ORL A,*direct*	85	MOV *direct*,*direct*	C5	XCH A,*direct*
06	INC @R0	46	ORL A,@R0	86	MOV *direct*,@R0	C6	XCH A,@R0
07	INC @R1	47	ORL A,@R1	87	MOV *direct*,@R1	C7	XCH A,@R1
08	INC R0	48	ORL A,R0	88	MOV *direct*,R0	C8	XCH A,R0
09	INC R1	49	ORL A,R1	89	MOV *direct*,R1	C9	XCH A,R1
0A	INC R2	4A	ORL A,R2	8A	MOV *direct*,R2	CA	XCH A,R2
0B	INC R3	4B	ORL A,R3	8B	MOV *direct*,R3	CB	XCH A,R3
0C	INC R4	4C	ORL A,R4	8C	MOV *direct*,R4	CC	XCH A,R4
0D	INC R5	4D	ORL A,R5	8D	MOV *direct*,R5	CD	XCH A,R5
0E	INC R6	4E	ORL A,R6	8E	MOV *direct*,R6	CE	XCH A,R6
0F	INC R7	4F	ORL A,R7	8F	MOV *direct*,R7	CF	XCH A,R7
10	JBC *bitAddr,relAddr*	50	JNC *relAddr*	90	MOV DPTR,*#data16*	D0	POP *direct*
11	ACALL *pg0Addr*	51	ACALL *pg2Addr*	91	ACALL *pg4Addr*	D1	ACALL *pg5Addr*
12	LCALL *address16*	52	ANL *direct*,A	92	MOV *bitAddr*,C	D2	SETB *bitAddr*
13	RRC A	53	ORL *direct*,*#data8*	93	MOVC A,@DPTR	D3	SETB C
14	DEC A	54	ANL A,*#data8*	94	SUBB A,*#data8*	D4	DA A
15	DEC *direct*	55	ANL A,*direct*	95	SUBB A,*direct*	D5	DJNZ *direct*,*relAddr*
16	DEC @R0	56	ANL A,@R0	96	SUBB A,@R0	D6	XCHD A,@R0
17	DEC @R1	57	ANL A,@R1	97	SUBB A,@R1	D7	XCHD A,@R1
18	DEC R0	58	ANL A,R0	98	SUBB A,R0	D8	XCHD A,R0
19	DEC R1	59	ANL A,R1	99	SUBB A,R1	D9	XCHD A,R1
1A	DEC R2	5A	ANL A,R2	9A	SUBB A,R2	DA	XCHD A,R2
1B	DEC R3	5B	ANL A,R3	9B	SUBB A,R3	DB	XCHD A,R3
1C	DEC R4	5C	ANL A,R4	9C	SUBB A,R4	DC	XCHD A,R4
1D	DEC R5	5D	ANL A,R5	9D	SUBB A,R5	DD	XCHD A,R5
1E	DEC R6	5E	ANL A,R6	9E	SUBB A,R6	DE	XCHD A,R6
1F	DEC R7	5F	ANL A,R7	9F	SUBB A,R7	DF	XCHD A,R7
20	JB *bitAddr,relAddr*	60	JZ *relAddr*	A0	ORL C,*/bitAddr*	E0	MOVX A,@DPTR
21	AJMP *pg1Addr*	61	AJMP *pg3Addr*	A1	AJMP *pg5Addr*	E1	AJMP *pg7Addr*
22	RET	62	XRL *direct*,A	A2	MOV C,*bitAddr*	E2	MOVX A,@R0
23	RL A	63	XRL *direct*,*#data8*	A3	INC DPTR	E3	MOVX A,@R1
24	ADD A,*#data8*	64	XRL A,*#data8*	A4	MUL AB	E4	CLR A
25	ADD A,*direct*	65	XRL A,*direct*	A5		E5	MOV A,*direct*
26	ADD A,@R0	66	XRL A,@R0	A6	MOV @R0,*direct*	E6	MOV A,@R0
27	ADD A,@R1	67	XRL A,@R1	A7	MOV @R1,*direct*	E7	MOV A,@R1
28	ADD A,R0	68	XRL A,R0	A8	MOV R0,*direct*	E8	MOV A,R0
29	ADD A,R1	69	XRL A,R1	A9	MOV R1,*direct*	E9	MOV A,R1
2A	ADD A,R2	6A	XRL A,R2	AA	MOV R2,*direct*	EA	MOV A,R2
2B	ADD A,R3	6B	XRL A,R3	AB	MOV R3,*direct*	EB	MOV A,R3
2C	ADD A,R4	6C	XRL A,R4	AC	MOV R4,*direct*	EC	MOV A,R4
2D	ADD A,R5	6D	XRL A,R5	AD	MOV R5,*direct*	ED	MOV A,R5
2E	ADD A,R6	6E	XRL A,R6	AE	MOV R6,*direct*	EE	MOV A,R6
2F	ADD A,R7	6F	XRL A,R7	AF	MOV R7,*direct*	EF	MOV A,R7
30	JNB *bitAddr,relAddr*	70	JNZ *relAddr*	B0	ANL C,*/bitAddr*	F0	MOVX @DPTR,A
31	ACALL *pg1Addr*	71	ACALL *pg3Addr*	B1	ACALL *pg5Addr*	F1	ACALL *pg7Addr*
32	RETI	72	ORL C,*bitAddr*	B2	CPL *bitAddr*	F2	MOVX @R0,A
33	RLC A	73	JMP @A+DPTR	B3	CPL C	F3	MOVX @R1,A
34	ADDC A,#data	74	MOV A,*#data8*	B4	CJNE A,*#data8*,*relAddr*	F4	CPL A
35	ADDC A,*direct*	75	MOV *direct*,*#data8*	B5	CJNE A,*direct*,*relAddr*	F5	MOV *direct*,A
36	ADDC A,@R0	76	MOV @R0,*#data8*	B6	CJNE @R0,*#data8*,*relAddr*	F6	MOV @R0,A
37	ADDC A,@R1	77	MOV @R1,*#data8*	B7	CJNE @R1,*#data8*,*relAddr*	F7	MOV @R1,A
38	ADDC A,R0	78	MOV R0,*#data8*	B8	CJNE R0,*#data8*,*relAddr*	F8	MOV R0,A
39	ADDC A,R1	79	MOV R1,*#data8*	B9	CJNE R1,*#data8*,*relAddr*	F9	MOV R1,A
3A	ADDC A,R2	7A	MOV R2,*#data8*	BA	CJNE R2,*#data8*,*relAddr*	FA	MOV R2,A
3B	ADDC A,R3	7B	MOV R3,*#data8*	BB	CJNE R3,*#data8*,*relAddr*	FB	MOV R3,A
3C	ADDC A,R4	7C	MOV R4,*#data8*	BC	CJNE R4,*#data8*,*relAddr*	FC	MOV R4,A
3D	ADDC A,R5	7D	MOV R5,*#data8*	BD	CJNE R5,*#data8*,*relAddr*	FD	MOV R5,A

Appendix E

8052 Instruction Set

Appendix E lists the 8052 instruction set.

Topic **Page**

E.1 Description

This appendix is a reference for all instructions in the 8052 instruction set. For each instruction, the following information is provided:

- **Instruction**—indicates the correct syntax for the given opcode.
- **OpCode**—the operation code, in the range of 0x00 through 0xFF, that represents the given instruction in machine code.
- **Bytes**—the total number of bytes (including the opcode byte) that make up the instruction.
- **Cycles**—the number of machine cycles required to execute the instruction.
- **Flags**—the flags that are modified by the instruction, if any.

When listing instruction syntax, the following terms will be used:

- *bitAddr*—Bit address value (00–FF)
- *pgXAddr*—Absolute 2k (13-bit) Address
- *data8*—Immediate 8-bit data value
- *data16*—Immedate 16-bit data value
- *address16*—16-bit code address
- *direct*—Direct address (IRAM 00–7F, SFR 80–FF)
- *relAddr*—Relative address (–127 to +128 bytes)

E.2 8052 Instruction Set

ACALL — Absolute Call within 2k Block

Syntax ACALL *codeAddress*

Instructions	OpCode	Bytes	Cycles	Flags
ACALL *pg0Addr*	0x11	2	2	None
ACALL *pg1Addr*	0x31	2	2	None
ACALL *pg2Addr*	0x51	2	2	None
ACALL *pg3Addr*	0x71	2	2	None
ACALL *pg4Addr*	0x91	2	2	None
ACALL *pg5Addr*	0xB1	2	2	None
ACALL *pg6Addr*	0xD1	2	2	None
ACALL *pg7Addr*	0xF1	2	2	None

ACALL unconditionally calls a subroutine at the indicated code address. ACALL pushes the address of the instruction that follows ACALL onto the stack, least significant byte first, and most significant byte second. The program counter is then updated so that program execution continues at the indicated address.

The new value for the program counter is calculated by replacing the least-significant-byte of the program counter with the second byte of the ACALL instruction, and replacing bits 0–2 of the most-significant-byte of the program counter with bits 5–7 of the opcode value. Bits 3–7 of the most-significant-byte of the program counter remain unchaged.

Calls must only be made to routines located within the same 2k block as the first byte that follows ACALL because only 11 bits of the program counter are affected by ACALL.

See also: LCALL, RET

ADD, ADDC Add Value, Add Value with Carry

Syntax

ADD A,*operand*
ADDC A,*operand*

Instructions	OpCode	Bytes	Cycles	Flags
ADD A,*#data8*	0x24	2	1	C, AC, OV
ADD A,*direct*	0x25	2	1	C, AC, OV
ADD A,@R0	0x26	1	1	C, AC, OV
ADD A,@R1	0x27	1	1	C, AC, OV
ADD A,R0	0x28	1	1	C, AC, OV
ADD A,R1	0x29	1	1	C, AC, OV
ADD A,R2	0x2A	1	1	C, AC, OV
ADD A,R3	0x2B	1	1	C, AC, OV
ADD A,R4	0x2C	1	1	C, AC, OV
ADD A,R5	0x2D	1	1	C, AC, OV
ADD A,R6	0x2E	1	1	C, AC, OV
ADD A,R7	0x2F	1	1	C, AC, OV
ADDC A,*#data8*	0x34	2	1	C, AC, OV
ADDC A,*direct*	0x35	2	1	C, AC, OV
ADDC A,@R0	0x36	1	1	C, AC, OV
ADDC A,@R1	0x37	1	1	C, AC, OV
ADDC A,R0	0x38	1	1	C, AC, OV
ADDC A,R1	0x39	1	1	C, AC, OV
ADDC A,R2	0x3A	1	1	C, AC, OV
ADDC A,R3	0x3B	1	1	C, AC, OV
ADDC A,R4	0x3C	1	1	C, AC, OV
ADDC A,R5	0x3D	1	1	C, AC, OV
ADDC A,R6	0x3E	1	1	C, AC, OV
ADDC A,R7	0x3F	1	1	C, AC, OV

ADD and ADDC both add the value operand to the value of the accumulator, leaving the resulting value in the accumulator. The value operand is not affected. ADD and ADDC function identically except that ADDC adds the value of operand as well as the value of the carry flag, whereas ADD does not add the carry flag to the result.

The carry (C) bit is set if there is a carry-out of bit 7. In other words, if the unsigned summed value of the accumulator, operand, and (in the case of ADDC) the carry flag exceeds 255, the carry bit is set. Otherwise, the carry bit is cleared.

The auxillary carry (AC) bit is set if there is a carry-out of bit 3. In other words, if the unsigned summed value of the low nibble of the accumulator, operand, and (in the case of ADDC) the carry flag exceeds 15, the auxillary carry flag is set. Otherwise, the auxillary carry flag is cleared.

The overflow (OV) bit is set if there is a carry-out of bit 6 or out of bit 7, but not both. In other words, if the addition of the accumulator, operand, and (in the case of ADDC) the carry flag treated as signed values results in a value that is out of the range of a signed byte (–128 through +127), the Overflow flag is set. Otherwise, the Overflow flag is cleared.

See also: SUBB, DA, INC, DEC

AJMP Absolute Jump within 2k Block

Syntax AJMP *codeAddress*

Instructions	OpCode	Bytes	Cycles	Flags
AJMP *pg0Addr*	0x01	2	2	None
AJMP *pg1Addr*	0x21	2	2	None
AJMP *pg2Addr*	0x41	2	2	None
AJMP *pg3Addr*	0x61	2	2	None
AJMP *pg4Addr*	0x81	2	2	None
AJMP *pg5Addr*	0xA1	2	2	None
AJMP *pg6Addr*	0xC1	2	2	None
AJMP *pg7Addr*	0xE1	2	2	None

AJMP unconditionally jumps to the indicated *codeAddress*. The new value for the program counter is calculated by replacing the least-significant-byte of the program counter with the second byte of the AJMP instruction, and replacing bits 0–2 of the most-significant-byte of the program counter with bits 5–7 of the opcode value. Bits 3–7 of the most-significant-byte of the program counter remain unchanged.

Jumps must only be made to code located within the same 2k block as the first byte that follows AJMP because only 11 bits of the program counter are affected by AJMP.

See also: LJMP, SJMP

ANL Bitwise AND

Syntax ANL *operand1*,*operand2*

Instructions	**OpCode**	**Bytes**	**Cycles**	**Flags**
ANL *direct*,A	0x52	2	1	None
ANL *direct*,*#data8*	0x53	3	2	None
ANL A,*#data8*	0x54	2	1	None
ANL A,*direct*	0x55	2	1	None
ANL A,@R0	0x56	1	1	None
ANL A,@R1	0x57	1	1	None
ANL A,R0	0x58	1	1	None
ANL A,R1	0x59	1	1	None
ANL A,R2	0x5A	1	1	None
ANL A,R3	0x5B	1	1	None
ANL A,R4	0x5C	1	1	None
ANL A,R5	0x5D	1	1	None
ANL A,R6	0x5E	1	1	None
ANL A,R7	0x5F	1	1	None
ANL C,*bitAddr*	0x82	2	1	C
ANL C,*/bitAddr*	0xB0	2	1	C

ANL does a bitwise AND operation between *operand1* and *operand2*, leaving the resulting value in *operand1*. The value of *operand2* is not affected. A logical AND compares the bits of each operand and sets the corresponding bit in the resulting byte only if the bit was set in both of the original operands. Otherwise, the resulting bit is cleared.

See also: ORL, XRL

CJNE Compare and Jump if Not Equal

Syntax CJNE *operand1*,*operand2*,*reladdr*

Instructions	OpCode	Bytes	Cycles	Flags
CJNE A,#data8,reladdr	0xB4	3	2	C
CJNE A,*direct*,*reladdr*	0xB5	3	2	C
CJNE @R0,*#data8*,*reladdr*	0xB6	3	2	C
CJNE @R1,*#data8*,*reladdr*	0xB7	3	2	C
CJNE R0,*#data8*,*reladdr*	0xB8	3	2	C
CJNE R1,*#data8*,*reladdr*	0xB9	3	2	C
CJNE R2,*#data8*,*reladdr*	0xBA	3	2	C
CJNE R3,*#data8*,*reladdr*	0xBB	3	2	C
CJNE R4,*#data8*,*reladdr*	0xBC	3	2	C
CJNE R5,*#data8*,*reladdr*	0xBD	3	2	C
CJNE R6,*#data8*,*reladdr*	0xBE	3	2	C
CJNE R7,*#data8*,*reladdr*	0xBF	3	2	C

CJNE compares the value of *operand1* and *operand2* and branches to the indicated relative address if the two operands are not equal. If the two operands are equal, program flow continues with the instruction following the CJNE instruction.

The carry (C) bit is set if *operand1* is less than *operand2*, otherwise it is cleared.

See also: DJNZ

CLR Clear Register

Syntax CLR *register*

Instructions	OpCode	Bytes	Cycles	Flags
CLR *bitAddr*	0xC2	2	1	None
CLR C	0xC3	1	1	C
CLR A	0xE4	1	1	None

CLR clears (sets to 0) the bit(s) of the indicated register. If the register is a bit (including the carry bit), only the specified bit is affected. Clearing the accumulator sets the accumulator value to 0.

See also: SETB

CPL Complement Register

Syntax CPL *operand*

Instructions	OpCode	Bytes	Cycles	Flags
CPL A	0xF4	1	1	None
CPL C	0xB3	1	1	C
CPL *bitAddr*	0xB2	2	1	None

CPL complements *operand*, leaving the result in *operand*. If *operand* is a single bit, the state of the bit is reversed. If *operand* is the accumulator, all the bits in the accumulator are reversed. This can be thought of as accumulator logical exclusive OR 255, or as 255-accumulator. If *operand* refers to a bit of an output port, the value complemented is based on the last value written to that bit, not the last value read from it.

See also: CLR, SETB

DA Decimal Adjust Accumulator

Syntax DA A

Instructions	OpCode	Bytes	Cycles	Flags
DA A	0xD4	1	1	C

DA adjusts the contents of the accumulator to correspond to a BCD (binary coded decimal) number after two BCD numbers have been added by the ADD or ADDC instruction.

If the carry bit is set or if the value of bits 0–3 exceed 9, 0x06 is added to the accumulator. If the carry bit was set when the instruction began, or if 0x06 was added to the accumulator in the first step, 0x60 is added to the accumulator.

The carry (C) bit is set if the resulting value is greater than 0x99. Otherwise, it is cleared.

See also: ADD, ADDC

DEC Decrement Register

Syntax DEC *register*

Instructions	OpCode	Bytes	Cycles	Flags
DEC A	0x14	1	1	None
DEC *direct*	0x15	2	1	None
DEC @R0	0x16	1	1	None
DEC @R1	0x17	1	1	None
DEC R0	0x18	1	1	None
DEC R1	0x19	1	1	None
DEC R2	0x1A	1	1	None
DEC R3	0x1B	1	1	None
DEC R4	0x1C	1	1	None
DEC R5	0x1D	1	1	None
DEC R6	0x1E	1	1	None
DEC R7	0x1F	1	1	None

DEC decrements the value of *register* by 1. If the initial value of *register* is 0, decrementing the value causes it to reset to 255 ($0xFF_H$).

Note:

The carry flag is not set when the value rolls over from 0 to 255.

See also: INC, SUBB

DIV Divide Accumulator by B

Syntax DIV AB

Instructions	OpCode	Bytes	Cycles	Flags
DIV AB	0x84	1	1	C, OV

Divides the unsigned value of the accumulator by the unsigned value of the B register. The resulting quotient is placed in the accumulator and the remainder is placed in the B register.

The carry (C) flag is always cleared.

The overflow (OV) flag is set if division by 0 was attempted. Otherwise, it is cleared.

See also: MUL AB

DJNZ Decrement and Jump if Not Zero

Syntax DJNZ *register,relAddr*

Instructions	OpCode	Bytes	Cycles	Flags
DJNZ *direct*,*relAddr*	0xD5	3	2	None
DJNZ R0,*relAddr*	0xD8	2	2	None
DJNZ R1,*relAddr*	0xD9	2	2	None
DJNZ R2,*relAddr*	0xDA	2	2	None
DJNZ R3,*relAddr*	0xDB	2	2	None
DJNZ R4,*relAddr*	0xDC	2	2	None
DJNZ R5,*relAddr*	0xDD	2	2	None
DJNZ R6,*relAddr*	0xDE	2	2	None
DJNZ R7,*relAddr*	0xDF	2	2	None

DJNZ decrements the value of *register* by 1. If the initial value of *register* is 0, decrementing the value causes it to reset to 255 ($0xFF_H$). If the new value of *register* is not 0, the program branchs to the address indicated by *relAddr*. If the new value of *register* is 0, program flow continues with the instruction following the DJNZ instruction.

See also: DEC, JZ, JNZ

INC Increment Reister

Syntax INC *register*

Instructions	OpCode	Bytes	Cycles	Flags
INC A	0x04	1	1	None
INC *direct*	0x05	2	1	None
INC @R0	0x06	1	1	None
INC @R1	0x07	1	1	None
INC R0	0x08	1	1	None
INC R1	0x09	1	1	None
INC R2	0x0A	1	1	None
INC R3	0x0B	1	1	None
INC R4	0x0C	1	1	None
INC R5	0x0D	1	1	None
INC R6	0x0E	1	1	None
INC R7	0x0F	1	1	None
INC DPTR	0xA3	1	2	None

INC increments the value of *register* by 1. If the initial value of *register* is 255 ($0xFF_H$), incrementing the value causes it to reset to 0.

Note:

The carry flag is not set when the value rolls over from 255 to 0.

In the case of INC DPTR, the two-byte value of DPTR is incremented as an unsigned integer. If the initial value of DPTR is 65 535 ($0xFFFF_H$), incrementing the value causes it to reset to 0. Again, the carry flag is not set when the value of DPTR rolls over from 65 535 to 0.

See also: ADD, ADDC, DEC

JB Jump if Bit Set

Syntax JB *bitAddr*,*relAddr*

Instructions	OpCode	Bytes	Cycles	Flags
JB *bitAddr*,*relAddr*	0x20	3	2	None

JB branches to the address indicated by *relAddr* if the bit indicated by *bitAddr* is set. If the bit is not set, program execution continues with the instruction following the JB instruction.

See also: JBC, JNB

JBC Jump if Bit Set and Clear Bit

Syntax JBC *bitAddr*,*relAddr*

Instructions	OpCode	Bytes	Cycles	Flags
JBC *bitAddr*,*reladdr*	0x10	3	2	None

JBC branches to the address indicated by *relAddr* if the bit indicated by *bitAddr* is set. Before branching to *relAddr*, the instruction clears the indicated bit. If the bit is not set, program execution continues with the instruction following the JBC instruction and the value of the bit is not changed.

See also: JB, JNB

JC Jump if Carry Set

Syntax JC *relAddr*

Instructions	OpCode	Bytes	Cycles	Flags
JC *relAddr*	0x40	2	2	None

JC branches to the address indicated by *relAddr* if the carry bit is set. If the carry bit is not set, program execution continues with the instruction following the JC instruction.

See also: JNC

JMP — Jump to Data Pointer + Accumulator

Syntax JMP @A+DPTR

Instructions	OpCode	Bytes	Cycles	Flags
JMP @A+DPTR	0x73	1	2	None

JMP jumps unconditionally to the address represented by the sum of the value of DPTR and the value of the accumulator.

See also: LJMP, AJMP, SJMP

JNB — Jump if Bit Not Set

Syntax JNB *bitAddr*,*reladdr*

Instructions	OpCode	Bytes	Cycles	Flags
JNB *bitAddr*,*relAddr*	0x30	3	2	None

JNB branches to the address indicated by *relAddr* if the indicated bit is not set. If the bit is set, program execution continues with the instruction following the JNB instruction.

See also: JB, JBC

JNC — Jump if Carry Not Set

Syntax JNC *reladdr*

Instructions	OpCode	Bytes	Cycles	Flags
JNC *relAddr*	0x50	2	2	None

JNC branches to the address indicated by *relAddr* if the carry bit is not set. If the carry bit is set, program execution continues with the instruction following the JNB instruction.

See also: JC

JNZ Jump if Accumulator Not Zero

Syntax JNZ *reladdr*

Instructions	OpCode	Bytes	Cycles	Flags
JNZ *relAddr*	0x70	2	2	None

JNZ branches to the address indicated by *relAddr* if the accumulator contains any value except 0. If the value of the accumulator is zero, program execution continues with the instruction following the JNZ instruction.

See also: JZ

JZ Jump if Accumulator Zero

Syntax JZ *reladdr*

Instructions	OpCode	Bytes	Cycles	Flags
JZ *relAddr*	0x60	2	2	None

JZ branches to the address indicated by *relAddr* if the accumulator contains the value 0. If the value of the accumulator is not zero, program execution continues with the instruction following the JNZ instruction.

See also: JNZ

LCALL Long Call

Syntax LCALL *address16*

Instructions	OpCode	Bytes	Cycles	Flags
LCALL *address16*	0x12	3	2	None

LCALL calls a program subroutine. LCALL increments the program counter by 3 (to point to the instruction following LCALL) and pushes that value onto the stack , low byte first, high byte second. The program counter is then set to the 16-bit value *address16*, causing program execution to continue at that address.

See also: ACALL, RET

LJMP Long Jump

Syntax LJMP *address16*

Instructions	OpCode	Bytes	Cycles	Flags
LJMP *address16*	0x02	3	2	None

LJMP jumps unconditionally to the specified *address16*.

See also: AJMP, SJMP, JMP

MOV Move Memory Into/Out of Accumulator

Syntax MOV *operand1*, *operand2*

Instructions	OpCode	Bytes	Cycles	Flags
MOV A,*#data8*	0x74	2	1	None
MOV A,@R0	0xE6	1	1	None
MOV A,@R1	0xE7	1	1	None
MOV @R0,A	0xF6	1	1	None
MOV @R1,A	0xF7	1	1	None
MOV A,R0	0xE8	1	1	None
MOV A,R1	0xE9	1	1	None
MOV A,R2	0xEA	1	1	None
MOV A,R3	0xEB	1	1	None
MOV A,R4	0xEC	1	1	None
MOV A,R5	0xED	1	1	None
MOV A,R6	0xEE	1	1	None
MOV A,R7	0xEF	1	1	None
MOV A,*direct*	0xE5	2	1	None
MOV R0,A	0xF8	1	1	None
MOV R1,A	0xF9	1	1	None
MOV R2,A	0xFA	1	1	None
MOV R3,A	0xFB	1	1	None
MOV R4,A	0xFC	1	1	None
MOV R5,A	0xFD	1	1	None
MOV R6,A	0xFE	1	1	None
MOV R7,A	0xFF	1	1	None
MOV *direct*,A	0xF5	2	1	None

MOV copies the value of *operand2* into *operand1*. The value of *operand2* is not affected.

See also: MOVC, MOVX, XCH, XCHD, PUSH, POP

MOV Move Into/Out of Carry Bit

Syntax MOV *bit1*,*bit2*

Instructions	OpCode	Bytes	Cycles	Flags
MOV C,*bitAddr*	0xA2	2	1	C
MOV *bitAddr*,C	0x92	2	2	None

MOV copies the value of *bit2* into *bit1*. The value of *bit2* is not affected. Either *bit1* or *bit2* must refer to the carry bit.

MOV Move into/out of Internal RAM

Syntax MOV *operand1*,*operand2*

Instructions	OpCode	Bytes	Cycles	Flags
MOV @R0,*#data8*	0x76	2	1	None
MOV @R1,*#data8*	0x77	2	1	None
MOV @R0,*direct*	0xA6	2	2	None
MOV @R1,*direct*	0xA7	2	2	None
MOV R0,*#data8*	0x78	2	1	None
MOV R1,*#data8*	0x79	2	1	None
MOV R2,*#data8*	0x7A	2	1	None
MOV R3,*#data8*	0x7B	2	1	None
MOV R4,*#data8*	0x7C	2	1	None
MOV R5,*#data8*	0x7D	2	1	None
MOV R6,*#data8*	0x7E	2	1	None
MOV R7,*#data8*	0x7F	2	1	None
MOV R0,*direct*	0xA8	2	2	None
MOV R1,*direct*	0xA9	2	2	None
MOV R2,*direct*	0xAA	2	2	None
MOV R3,*direct*	0xAB	2	2	None
MOV R4,*direct*	0xAC	2	2	None
MOV R5,*direct*	0xAD	2	2	None
MOV R6,*direct*	0xAE	2	2	None
MOV R7,*direct*	0xAF	2	2	None
MOV *direct*,*#data8*	0x75	3	2	None
MOV *direct*,@R0	0x86	2	2	None
MOV *direct*,@R1	0x87	2	2	None
MOV *direct*,R0	0x88	2	2	None
MOV *direct*,R1	0x89	2	2	None
MOV *direct*,R2	0x8A	2	2	None
MOV *direct*,R3	0x8B	2	2	None
MOV *direct*,R4	0x8C	2	2	None
MOV *direct*,R5	0x8D	2	2	None
MOV *direct*,R6	0x8E	2	2	None
MOV *direct*,R7	0x8F	2	2	None
MOV *direct1*,*direct2*	0x85	3	2	None

MOV copies the value of *operand2* into *operand1*. The value of *operand2* is not affected.

Note:

In the case of MOV direct1,direct2, the operand bytes of the instruction are stored in reverse order. That is, the instruction consisting of the bytes 85_H, 20_H, 50_H means move the contents of internal RAM location 0x20 to internal RAM location 0x50, although the opposite would be generally presumed.

See also: MOVC, MOVX, XCH, XCHD, PUSH, POP

MOV DPTR Move value into DPTR

Syntax MOV DPTR,*#data16*

Instructions	OpCode	Bytes	Cycles	Flags
MOV DPTR,*#data16*	0x90	3	2	None

Sets the value of the data pointer (DPTR) to the value *data16*.

See also**: MOVX, MOVC**

MOVC Move Code Byte to Accumulator

Syntax MOVC A,@A+register

Instructions	OpCode	Bytes	Cycles	Flags
MOVC A,@A+DPTR	0x93	1	2	None
MOVC A,@A+PC	0x83	1	1	None

MOVC moves a byte from code memory into the accumulator. The code memory address that the byte is moved from is calculated by summing the value of the accumulator with either DPTR or the PC. In the case of the program counter, PC is first incremented by 1 before being summed with the accumulator.

See also: MOV, MOVX

MOVX Move Data to/from External RAM

Syntax MOVX *operand1,operand2*

Instructions	OpCode	Bytes	Cycles	Flags
MOVX @DPTR,A	0xF0	1	2	None
MOVX @R0,A	0xF2	1	2	None
MOVX @R1,A	0xF3	1	2	None
MOVX A,@DPTR	0xE0	1	2	None
MOVX A,@R0	0xE2	1	2	None
MOVX A,@R1	0xE3	1	2	None

MOVX moves a byte to or from external memory into or from the accumulator.

If *operand1* is @DPTR, the accumulator is moved to the 16-bit external memory address indicated by DPTR. This instruction uses both P0 (port 0) and P2 (port 2) to output the 16-bit address and data. If *operand2* is DPTR then the byte is moved from external memory into the accumulator.

If *operand1* is @R0 or @R1, the accumulator is moved to the 8-bit external memory address indicated by the specified register. This instruction uses only P0 (port 0) to output the 8-bit address and data. P2 (port 2) is not affected. If *operand2* is @R0 or @R1, the byte is moved from external memory into the accumulator.

See also: MOV, MOVC

MUL Multiply Accumulator by B

Syntax MUL AB

Instructions	OpCode	Bytes	Cycles	Flags
MUL AB	0xA4	1	4	C, OV

MUL multiplies the unsigned value in the accumulator by the unsigned value in the B register. The least-significant byte of the result is placed in the accumulator and the most-significant byte is placed in the B register.

The carry (C) flag is always cleared.

The overflow (OV) flag is set if the result is greater than 255 (if the most-significant byte is not zero). Otherwise, it is cleared.

See also: DIV

NOP No Operation

Syntax NOP

Instructions	OpCode	Bytes	Cycles	Flags
NOP	0x00	1	1	None

NOP, as its name suggests, causes no operation to take place for one machine cycle. NOP is generally used only for timing purposes. Absolutely no flags or registers are affected.

ORL Bitwise OR

Syntax Syntax: ORL *operand1,operand2*

Instructions	OpCode	Bytes	Cycles	Flags
ORL *direct*,A	0x42	2	1	None
ORL *direct*,*#data8*	0x43	3	2	None
ORL A,*#data8*	0x44	2	1	None
ORL A,*direct*	0x45	2	1	None
ORL A,@R0	0x46	1	1	None
ORL A,@R1	0x47	1	1	None
ORL A,R0	0x48	1	1	None
ORL A,R1	0x49	1	1	None
ORL A,R2	0x4A	1	1	None
ORL A,R3	0x4B	1	1	None
ORL A,R4	0x4C	1	1	None
ORL A,R5	0x4D	1	1	None
ORL A,R6	0x4E	1	1	None
ORL A,R7	0x4F	1	1	None
ORL C,bitAddr	0x72	2	2	C
ORL C,*/bitAddr*	0xA0	2	1	C

ORL does a bitwise OR operation between *operand1* and *operand2*, leaving the resulting value in *operand1*. The value of *operand2* is not affected. A logical OR compares the bits of each operand and sets the corresponding bit in the resulting byte if the bit was set in either of the original operands. Otherwise, the resulting bit is cleared.

See also: ANL, XRL

POP Pop Value from Stack

Syntax POP *register*

Instructions	OpCode	Bytes	Cycles	Flags
POP *direct*	0xD0	2	2	None

POP pops the last value placed on the stack into the *direct* address specified. In other words, POP will load *direct* with the value of the internal RAM address pointed to by the current stack pointer. The stack pointer is then decremented by 1.

Note:

The address of direct must be an internal RAM or SFR address. You cannot POP directly into R registers such as R0, R1, etc.. For example, to POP a value off the stack into R0, POP the value into the accumulator and then move the value of the accumulator into R0.

Note:

When POPping a value off the stack into the accumulator, code the instruction as POP ACC, not POP A. The latter is invalid and will result in an error at assemble time.

See also: PUSH

PUSH Push Value onto Stack

Syntax PUSH *register*

Instructions	OpCode	Bytes	Cycles	Flags
PUSH *direct*	0xC0	2	2	None

PUSH pushes the value of the specified *direct* address onto the stack. PUSH first increments the value of the stack pointer by 1, then takes the value stored in *direct* and stores it in internal RAM at the location pointed to by the incremented stack pointer.

Note:

The address of *direct* must be an internal RAM or SFR address. You cannot PUSH directly from R registers such as R0, R1, etc. For example, to push a value onto the stack from R0, move R0 into the accumulator, and then PUSH the value of the accumulator onto the stack.

Note:

When PUSHing a value from the accumulator onto the stack into the, code the instruction as PUSH ACC, not PUSH A. The latter is invalid and will result in an error at assemble time.

See also: POP

RET Return from Subroutine

Syntax RET

Instructions	OpCode	Bytes	Cycles	Flags
RET	0x22	1	2	None

RET is used to return from a subroutine previously called by LCALL or ACALL. Program execution continues at the address that is calculated by POPping the top-most two bytes off the stack. The most-significant byte is POPped off the stack first, followed by the least-significant byte.

See also: LCALL, ACALL, RETI

RETI Return from Interrupt

Syntax RETI

Instructions	OpCode	Bytes	Cycles	Flags
RETI	0x32	1	2	None

RETI is used to return from an interrupt service routine. RETI first enables interrupts of equal and lower priorities to the interrupt that is terminating. Program execution continues at the address that is calculated by POPping the top–most 2 bytes off the stack. The most-significant byte is POPped off the stack first, followed by the least-significant byte.

RETI functions identically to RET if it is executed outside of an interrupt service routine.

See also: RET

RL Rotate Accumulator Left

Syntax RL A

Instructions	OpCode	Bytes	Cycles	Flags
RL A	0x23	1	1	C

RL shifts the bits of the accumulator to the left. The left-most bit (bit 7) of the accumulator is loaded into bit 0.

See also: RLC, RR, RRC

RLC – Rotate Accumulator Left Through Carry

Syntax RLC A

Instructions	OpCode	Bytes	Cycles	Flags
RLC A	0x33	1	1	C

RLC shifts the bits of the accumulator to the left. The left-most bit (bit 7) of the accumulator is loaded into the carry flag, and the original carry flag is loaded into bit 0 of the accumulator.

See also: RL, RR, RRC

RR Rotate Accumulator Right

Syntax RR A

Instructions	OpCode	Bytes	Cycles	Flags
RR A	0x03	1	1	None

RR shifts the bits of the accumulator to the right. The right-most bit (bit 0) of the accumulator is loaded into bit 7.

See also: RL, RLC, RRC

RRC Rotate Accumulator Right Through Carry

Syntax RRC A

Instructions	OpCode	Bytes	Cycles	Flags
RRC A	0x13	1	1	C

RRC shifts the bits of the accumulator to the right. The right-most bit (bit 0) of the accumulator is loaded into the carry flag, and the original carry flag is loaded into bit 7.

See also: RL, RLC, RR

SETB Set Bit

Syntax SETB *bitAddr*

Instructions	OpCode	Bytes	Cycles	Flags
SETB C	0xD3	1	1	C
SETB *bitAddr*	0xD2	2	1	None

SETB sets the specified bit.

If the instruction requires the carry bit to be set, the assembler will automatically use the 0xD3 opcode. If any other bit is set, the assembler will automatically use the 0xD2 opcode.

See also: CLR

SJMP Short Jump

Syntax SJMP *relAddr*

Instructions	OpCode	Bytes	Cycles	Flags
SJMP *relAddr*	0x80	2	2	None

SJMP jumps unconditionally to the address specified *relAddr*. *RelAddr* must be within –128 or +127 bytes of the instruction that follows the SJMP instruction.

See also: LJMP, AJMP

SUBB Subtract from Accumulator with Borrow

Syntax SUBB A,*operand*

Instructions	OpCode	Bytes	Cycles	Flags
SUBB A,#*data8*	0x94	2	1	C, AC, OV
SUBB A,*direct*	0x95	2	1	C, AC, OV
SUBB A,@R0	0x96	1	1	C, AC, OV
SUBB A,@R1	0x97	1	1	C, AC, OV
SUBB A,R0	0x98	1	1	C, AC, OV
SUBB A,R1	0x99	1	1	C, AC, OV
SUBB A,R2	0x9A	1	1	C, AC, OV
SUBB A,R3	0x9B	1	1	C, AC, OV
SUBB A,R4	0x9C	1	1	C, AC, OV
SUBB A,R5	0x9D	1	1	C, AC, OV
SUBB A,R6	0x9E	1	1	C, AC, OV
SUBB A,R7	0x9F	1	1	C, AC, OV

SUBB subtracts the value of *operand* from the value of the accumulator, leaving the resulting value in the accumulator. The value *operand* is not affected.

The carry (C) bit is set if a borrow was required for bit 7. Otherwise, it is cleared. In other words, if the unsigned value being subtracted is greater than the accumulator, the carry flag is set.

The auxillary carry (AC) bit is set if a borrow was required for bit 3. Otherwise, it is cleared. In other words, the bit is set if the low nibble of the value being subtracted was greater than the low nibble of the accumulator.

The overflow (OV) bit is set if a borrow was required for bit 6 or for bit 7, but not both. In other words, the subtraction of two signed bytes resulted in a value outside the range of a signed byte (–128 to 127). Otherwise, it is cleared.

See also: ADD, ADDC, DEC

SWAP Subtract Accumulator Nibbles

Syntax SWAP A

Instructions	OpCode	Bytes	Cycles	Flags
SWAP A	0xC4	1	1	None

SWAP swaps bits 0–3 of the accumulator with bits 4–7 of the accumulator. This instruction is identical to executing RR A or RL A four times.

See also: RL, RLC, RR, RRC

XCH Exchange Bytes

Syntax XCH A,*register*

Instructions	OpCode	Bytes	Cycles	Flags
XCH A,@R0	0xC6	1	1	None
XCH A,@R1	0xC7	1	1	None
XCH A,R0	0xC8	1	1	None
XCH A,R1	0xC9	1	1	None
XCH A,R2	0xCA	1	1	None
XCH A,R3	0xCB	1	1	None
XCH A,R4	0xCC	1	1	None
XCH A,R5	0xCD	1	1	None
XCH A,R6	0xCE	1	1	None
XCH A,R7	0xCF	1	1	None
XCH A,*direct*	0xC5	2	1	None

XCH exchanges the value of the accumulator with the value contained in *register*.

See also: MOV

XCHD Exchange Digit

Syntax XCHD A,*register*

Instructions	OpCode	Bytes	Cycles	Flags
XCHD A,@R0	0xD6	1	1	None
XCHD A,@R1	0xD7	1	1	None

XCHD exchanges bits 0–3 of the accumulator with bits 0–3 of the internal RAM address pointed to indirectly by R0 or R1. Bits 4–7 of each register are unaffected.

See also: DA

XRL Bitwise Exclusive OR

Syntax XRL operand1,operand2

Instructions	OpCode	Bytes	Cycles	Flags
XRL *direct*,A	0x62	2	1	None
XRL *direct*,*#data8*	0x63	3	2	None
XRL A,*#data8*	0x64	2	1	None
XRL A,*direct*	0x65	2	1	None
XRL A,@R0	0x66	1	1	None
XRL A,@R1	0x67	1	1	None
XRL A,R0	0x68	1	1	None
XRL A,R1	0x69	1	1	None
XRL A,R2	0x6A	1	1	None
XRL A,R3	0x6B	1	1	None
XRL A,R4	0x6C	1	1	None
XRL A,R5	0x6D	1	1	None
XRL A,R6	0x6E	1	1	None
XRL A,R7	0x6F	1	1	None

XRL does a bitwise exclusive OR operation between *operand1* and *operand2*, leaving the resulting value in *operand1*. The value of *operand2* is not affected. A logical exclusive OR compares the bits of each operand and sets the corresponding bit in the resulting byte if the bit was set in either (but not both) of the original operands. Otherwise, the bit is cleared.

See also: ANL, ORL

UNDEFINED

Undefined Instruction

Syntax

???

Instructions	OpCode	Bytes	Cycles	Flags
???	0xA5	1	1	C

The undefined instruction is, as the name suggests, not a documented instruction. The 8052 supports 255 instructions and OpCode 0xA5 is the single opcode that is not used by any documented function. It is not recommended that it be executed because it is not documented nor defined.

However, based on my research, executing this undefined instruction takes one machine cycle and appears to have no effect on the system except that the carry bit always seems to be set.

Note:

We received input from an 8052.com user that the undefined instruction really has a format of Undefined *bit1,bit2* and effectively copies the value of *bit2* to *bit1*. In this case, it would be a three-byte instruction. We have not had an opportunity to verify or disprove this report, so we present it to the world as additional information.

See also: NOP

Appendix F

Bit-Addressable SFRs (alphabetical)

Appendix F defines the MSC1210 bit-addressable special function registers (SFRs) in alphabetical order.

Topic **Page**

F.1 Bit Addressable SFRs (alphabetical)

Enable Interrupt Control (EICON)

SFR Name: EICON

SFR Address: $D8_H$

Bit-Addressable: Yes

Bit Definitions:

	bit 7	**bit 6**	**bit 5**	**bit 4**	**bit 3**	**bit 2**	**bit 1**	**bit 0**
Name	SMOD1	—	EAI	AI	WDTI	—	—	—
Bit Address	DF_H	DE_H	DD_H	DC_H	DB_H	DA_H	$D9_H$	$D8_H$

SMOD1—Serial Port 1 Mode. 0 = Normal baud rate for serial port 1, 1 = Serial port 1 baud rate doubled.

EAI—Enable Auxiliary Interrupt. 1 = Interrupt enabled.

AI—Auxiliary Interrupt Flag. 1 = Auxiliary interrupt pending, will trigger interrupt if EAI bit set.

WDTI—Watchdog Interrupt Flag. 1 = Watchdog interrupt pending, will trigger interrupt if Watchdog interrupt enabled.

Extended Interrupt Enable (EIE)

SFR Name: EIE

SFR Address: $E8_H$

Bit-Addressable: Yes

Bit Definitions:

	bit 7	**bit 6**	**bit 5**	**bit 4**	**bit 3**	**bit 2**	**bit 1**	**bit 0**
Name	—	—	—	EWDI	EX5	EX4	EX3	EX2
Bit Address	EF_H	EE_H	ED_H	EC_H	EB_H	EA_H	$E9_H$	$E8_H$

EWDI—Watchdog Interrupt Enable. 1 = Watchdog interrupt enabled.

EX5—External 5 Interrupt Enable. 1 = External 5 interrupt enabled.

EX4—External 4 Interrupt Enable. 1 = External 4 interrupt enabled.

EX3—External 3 Interrupt Enable. 1 = External 3 interrupt enabled.

EX2—External 2 Interrupt Enable. 1 = External 2 interrupt enabled.

Extended Interrupt Priority (EIP)

SFR Name: EIE

SFR Address: $F8_H$

Bit-Addressable: Yes

Bit Definitions:

	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
Name	—	—	—	PWDI	PX5	PX4	PX3	PX2
Bit Address	FF_H	FE_H	FD_H	FC_H	FB_H	FA_H	$F9_H$	$F8_H$

PWDI—Watchdog Interrupt Priority. 1 = Watchdog interrupt high-level priority, 0 = low-level priority.

PX5—External 5 Interrupt Priority. 1 = External 5 interrupt high-level priority, 0 = low-level priority.

PX4—External 4 Interrupt Priority. 1 = External 4 interrupt high-level priority, 0 = low-level priority.

PX3—External 3 Interrupt Priority. 1 = External 3 interrupt high-level priority, 0 = low-level priority.

PX2—External 2 Interrupt Priority. 1 = External 2 interrupt high-level priority, 0 = low-level priority.

Interrupt Enable (IE)

SFR Name: IE

SFR Address: $A8_H$

Bit-Addressable: Yes

Bit Definitions:

	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
Name	EA	—	ET2	ES	ET1	EX1	ET0	EX0
Bit Address	AF_H	AE_H	AD_H	AC_H	AB_H	AA_H	$A9_H$	$A8_H$

EA—Enable/Disable All Interrupts. 1 = interrupts enabled.

ET2—Enable Timer 2 Interupt. 1 = interrupt enabled.

ES—Enable Serial Interupt. 1 = interrupt enabled.

ET1—Enable Timer 1 Interupt. 1 = interrupt enabled.

EX1—Enable External 1 Interupt. 1 = interrupt enabled.

ET0—Enable Timer 0 Interupt. 1 = interrupt enabled.

EX0—Enable External 0 Interupt. 1 = interrupt enabled.

INTERRUPT PRIORITY (IP)

SFR Name: IP

SFR Address: $B8_H$

Bit–Addressable: Yes

Bit–Definitions:

	bit 7	**bit 6**	**bit 5**	**bit 4**	**bit 3**	**bit 2**	**bit 1**	**bit 0**
Name	—	—	PT2	PS	PT1	PX1	PT0	PX0
Bit Address	BF_H	BE_H	BD_H	BC_H	BB_H	BA_H	$B9_H$	$B8_H$

PT2—Priority Timer 2 Interupt. 1 = high-priority interrupt, 0 = low-priority interrupt.

PS—Priority Serial Interupt. 1 = high-priority interrupt, 0 = low-priority interrupt.

PT1—Priority Timer 1 Interupt. 1 = high priority interrupt, 0 = low-priority interrupt.

PX1—Priority External 1 Interupt. 1 = high priority interrupt, 0 = low-priority interrupt.

PT0—Priority Timer 0 Interupt. 1 = high priority interrupt, 0 = low-priority interrupt.

PX0—Priority External 0 Interupt. 1 = High priority interrupt, 0 = low-priority interrupt.

Port 0 (P0)

SFR Name: P0

SFR Address: 80_H

Bit-Addressable: Yes

Bit Definitions:

	bit 7	**bit 6**	**bit 5**	**bit 4**	**bit 3**	**bit 2**	**bit 1**	**bit 0**
Name	AD7	AD6	AD5	AD4	AD3	AD2	AD1	AD0
Bit Address	87_H	86_H	85_H	84_H	83_H	82_H	81_H	80_H

Note:

These bit names indicate the function of that I/O line on the P0 bus when used with external memory (code/RAM). A standard 8052 assembler will not recognize these bits by the given names; they will only be recognized as P0.7, P0.6, etc.

Note:

Port 0 is only available for general input/output if the project does not use external code memory or external RAM. When such external memory is used, Port 0 is used automatically by the microcontroller to address the memory and read/write data from/to said memory.

Port 1 (P1)

SFR Name: P1

SFR Address: 90_H

Bit–Addressable: Yes

Bit–Definitions:

	bit 7	**bit 6**	**bit 5**	**bit 4**	**bit 3**	**bit 2**	**bit 1**	**bit 0**
Name	P1.7	P1.6	P1.5	P1.4	P1.3	P1.2	T2EX	T2
Bit Address	97_H	96_H	95_H	94_H	93_H	92_H	91_H	90_H

T2EX—Timer 2 Capture/Reload. Optional external capturing or reloading of timer 2.

T2—Timer 2 External Input. Optionally used to control timer/counter 2 via external source.

Port 2 (P2)

SFR Name: P2

SFR Address: $A0_H$

Bit–Addressable: Yes

Bit–Definitions:

	bit 7	**bit 6**	**bit 5**	**bit 4**	**bit 3**	**bit 2**	**bit 1**	**bit 0**
Name	A15	A14	A13	A12	A11	A10	A9	A8
Bit Address	$A7_H$	$A6_H$	$A5_H$	$A4_H$	$A3_H$	$A2_H$	$A1_H$	$A0_H$

Note:

These bit names indicate the function of that I/O line on the P2 bus when used with external memory (code/RAM). A standard 8052 assembler will not recognize these bits by the given names; they will only be recognized as P2.7, P2.6, etc.

Note:

Port 2 is only available for general input/output if the project does not use external code memory or external RAM. When such external memory is used, Port 2 is used automatically by the microcontroller to address the memory and read/write data from/to said memory.

Port 3 (P3)

SFR Name: P3

SFR Address: B0_H

Bit–Addressable: Yes

Bit–Definitions:

	bit 7	**bit 6**	**bit 5**	**bit 4**	**bit 3**	**bit 2**	**bit 1**	**bit 0**
Name	$\overline{\text{RD}}$	$\overline{\text{WR}}$	T1	T0	$\overline{\text{INT1}}$	$\overline{\text{INT0}}$	TXD	RXD
Bit Address	B7_H	B6_H	B5_H	B4_H	B3_H	B2_H	B1_H	B0_H

RD—Read Strobe. 0 = external memory read strobe.

WR—Write Strobe. 0 = external memory write strobe.

T1—Timer/Counter 1 External Input. Optionally used to control timer/counter 1 via external source.

T0—Timer/Counter 0 External Input. Optionally used to control timer/counter 0 via external source.

INT1—External Interrupt 1. Used to trigger external interrupt 1.

INT0—External Interrupt 0. Used to trigger external interrupt 0.

TXD—Serial Transmit Data. 8052 serial transmit line (from 8052 to external device).

RXD—Serial Transmit Data. 8052 serial receive line (to 8052 from external device).

Note:

These bit names indicate the function of that I/O line on the P3 bus. A standard 8052 assembler will not recognize these bits by the given names; they will only be recognized as P3.7, P3.6, etc.

Program Status Word (PSW)

SFR Name: PSW

SFR Address: $D0_H$

Bit–Addressable: Yes

Bit–Definitions:

	bit 7	**bit 6**	**bit 5**	**bit 4**	**bit 3**	**bit 2**	**bit 1**	**bit 0**
Name	CY	AC	F0	RS1	RS0	OV	—	P
Bit Address	$D7_H$	$D6_H$	$D5_H$	$D4_H$	$D3_H$	$D2_H$	$D1_H$	$D0_H$

CY—Carry Flag. Set or cleared by instructions ADD, ADDC, SUBB, MUL, and DIV.

AC—Auxiliary Carry. Set or cleared by instructions ADD, ADDC.

F0—Flag 0. General flag available to developer for user-defined purposes.

RS1/RS0—Register Select Bits. These two bits, taken together, select the register bank used when using R registers R0 through R7, according to the following table:

RS1	**RS0**	**Register Bank**	**Register Bank Addresses**
0	0	0	00_H–07_H
0	1	1	08_H–$0F_H$
1	0	2	10_H–17_H
1	1	3	18_H–$1F_H$

OV—Overflow Flag. Set or cleared by instructions ADD, ADDC, SUBB, and DIV.

P—Parity Flag. Set or cleared automatically by core to establish even parity with the accumulator, so that the number of bits set in the accumulator plus the value of the parity bit will always equal an even number.

Serial Control (SCON)

SFR Name: SCON

SFR Address: 98_H

Bit–Addressable: Yes

Bit–Definitions:

	bit 7	**bit 6**	**bit 5**	**bit 4**	**bit 3**	**bit 2**	**bit 1**	**bit 0**
Name	SM0	SM1	SM2	REN	TB8	RB8	TI	RI
Bit Address	$9F_H$	$9E_H$	$9D_H$	$9C_H$	$9B_H$	$9A_H$	99_H	98_H

SM0/SM1– Serial Mode. These two bits, taken together, select the serial mode in which the serial port will operate.

SM0	SM1	Serial Mode	Description	Baud
0	0	0	Shift Register	Oscillator/12
0	1	1	8-Bit UART	Variable (T1 or T2)
1	0	2	9-Bit UART	Oscillator/64 or /32
1	1	3	9-Bit UART	Variable (T1 or T2)

SM2—Serial Mode 2 (Multiprocessor Communication). When this bit is set, multiprocessor communication is enabled in modes 2 and 3 causing the RI bit to only be set when the ninth bit of a byte received is set. In mode 1, RI is only set if a valid stop bit is received. SM2 must be cleared in mode 0.

REN—Received Enable. This bit must be set to enable data reception via the serial port. No data will be received by the serial port if this bit is clear.

TB8—Transmit Bit 8. When in modes 2 and 3, this is the ninth bit sent when a byte is written to SBUF.

RB8—Receive Bit 8. When in modes 2 and 3, this is the ninth bit that was received. In mode 1, and if SM2 is set, RB8 holds the value of the stop bit that was received. RB8 is not used in mode 0.

TI—Transmit Interrupt. Set by hardware when the byte previously written to SBUF has been completely clocked out the serial port.

RI—Receive Interrupt. Set by hardware when a byte has been received by the serial port and is available to be read in SBUF.

Timer Control (TCON)

SFR Name: TCON

SFR Address: 88_H

Bit–Addressable: Yes

Bit–Definitions:

	bit 7	**bit 6**	**bit 5**	**bit 4**	**bit 3**	**bit 2**	**bit 1**	**bit 0**
Name	TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0
Bit Address	$8F_H$	$8E_H$	$8D_H$	$8C_H$	$8B_H$	$8A_H$	89_H	88_H

TF1—Timer 1 Overflow Flag. This bit is set by the MCU when Timer 1 overflows from $FFFF_H$ back to 0000_H. Cleared by software, or cleared automatically by hardware if a Timer 1 interrupt is triggered.

TR1—Timer 1 Run Control. When this bit is set, Timer 1 counts depending on its configuration in TMOD. When this bit is clear, Timer 1 is stopped.

TF0—Timer 0 Overflow Flag. This bit is set by the MCU when Timer 0 overflows from $FFFF_H$ back to 0000_H. Cleared by software, or cleared automatically by hardware if a Timer 0 interrupt is triggered.

TR0—Timer 1 Run Control. When this bit is set, Timer 0 counts depending on its configuration in TMOD. When this bit is clear, Timer 0 is stopped.

IE1—External 1 Interrupt Flag. This bit is set by the MCU when an external 1 interrupt is detected on the $\overline{\text{INT1}}$ line. Cleared by software, or cleared automatically by hardware if an external 1 interrupt is triggered.

IT1—External 1 Interrupt Type Flag. This bit controls whether or not external 1 interrupt is edge-triggered or low-level-triggered. If this bit is set, external 1 interrupt is triggered when a 1-0 transition is detected on the $\overline{\text{INT1}}$ line. If this bit is clear, external 1 interrupt is triggered continuously when $\overline{\text{INT1}}$ is at a low state.

IE0—External 0 Interrupt Flag. This bit is set by the MCU when an external 0 interrupt is detected on the $\overline{\text{INT0}}$ line. Cleared by software, or cleared automatically by hardware if an external 1 interrupt is triggered.

IT0—External 0 Interrupt Type Flag. This bit controls whether or not external 0 interrupt is edge-triggered or low-level-triggered. If this bit is set, external 1 interrupt is triggered when a 1-0 transition is detected on the $\overline{\text{INT0}}$ line. If this bit is clear, external 0 interrupt is triggered continuously when $\overline{\text{INT0}}$ is at a low state.

Timer 2 Control (T2CON)

SFR Name: T2CON

SFR Address: $C8_H$

Bit–Addressable: Yes

Bit–Definitions:

	bit 7	**bit 6**	**bit 5**	**bit 4**	**bit 3**	**bit 2**	**bit 1**	**bit 0**
Name	TF2	EXF2	RCLK	TCLK	EXEN2	TR2	$C/\overline{T2}$	$CP/\overline{RL2C}$
Bit Address	CF_H	CE_H	CD_H	CC_H	CB_H	CA_H	$C9_H$	$C8_H$

TF2—Timer 2 Overflow Flag. This bit is set by the MCU when Timer 2 overflows from $FFFF_H$ back to 0000_H. When enabled, this bit causes a Timer 2 interrupt. Cleared by software. This bit is not set when TCLK or RCLK is set.

EXF—Timer 2 External Flag. This bit is set by the MCU when a timer capture or reload is triggered by a 1-0 transition on T2EX. When enabled, this bit causes a Timer 2 interrupt. Cleared by software.

RCLK—Timer 2 Receive Clock. When this bit is set, Timer 2 provides the serial port receive baud rate clock.

TCLK—Timer 2 Transmit Clock. When this bit is set, Timer 2 provides the serial port transmit baud rate clock.

EXEN2—Timer 2 External Enable. When this bit is set, a capture or reload is triggered on a 1-0 transition on the T2EX line.

TR2—Timer 2 Run Control. When this bit is set, Timer 2 is activated/run. When this bit is clear, Timer 2 is stopped.

$C/\overline{T2}$—Counter/Interval Timer. When this bit is set, Timer 2 acts as an event counter based on external stimulus on the T2EX line. When this bit is clear, Timer 2 acts as an interval timer.

$CP/\overline{RL2C}$—Capture/Reload. When set, a capture occurs on a 1-0 transition of T2EX. When clear, a reload occurs on timer overflow or on a 1-0 transition of T2EX. This bit is only relevant if EXEN2 is set, and does not apply if RCLK or TCLK are set.

Appendix G

SFRs/Address Cross-Reference Guide (alphabetical)

Appendix G lists an alphabetical cross-reference of the MSC1210 special function registers (SFRs) and their addresses.

G.1 SFR/Address Cross-Reference

SFR Name	Description	SFR Address (Hex)
ACLK	Analog Clock	$F6_H$
ADCON0	ADC Control 0	DC_H
ADCON1	ADC Control 1	DD_H
ADCON2	ADC Control 2	DE_H
ADCON3	ADC Control 3	DF_H
ADMUX	ADC Multiplexer	$D7_H$
ADRESH	ADC Result High	DB_H
ADRESL	ADC Result Low	$D9_H$
ADRESM	ADC Result Middle	DA_H
AIE	Auxiliary Interrupt Enable	$A6_H$
AISTAT	Auxiliary Interrupt Status	$A7_H$
BPCON	Breakpoint Control	$A9_H$
BPH	Breakpoint High	AB_H
BPL	Breakpoint Low	AA_H
CADDR	Configuration Address	93_H
CDATA	Configuration Data	94_H
CKCON	Clock Control	$8E_H$
DPL0	Data Pointer 0 Low	82_H
DPH0	Data Pointer 0 High	83_H
DPL1	Data Pointer 1 Low	84_H
DPH1	Data Pointer 1 High	85_H
DPS	Data Pointer Select	86_H
EIE	Extended Interrupt Enable	$E8_H$
EIP	Extended Interrupt Priority	$F8_H$
EWU	Enable Wake Up from Idle	$C6_H$
EXIF	External Interrupt Flag	91_H
FMCON	Flash Memory Control	EE_H
FTCON	Flash Memory Timing Control	EF_H
GCH	Gain Calibration High	$D6_H$
GCL	Gain Calibration Low	$D4_H$
GCM	Gain Calibration Middle	$D5_H$
HMSEC	Hundred Millisecond Counter	FE_H
HWPC0	Hardware Product Code 0	$E9_H$
HWPC1	Hardware Product Code 1	EA_H
LVDCON	Low Voltage Detection Control	$E7_H$
MCON	Memory Control	95_H

MPAGE	Memory Page	92_H
MSECH	Millisecond Counter High	FD_H
MSECL	Millisecond Counter Low	FC_H
MSINT	Microseconds Interrupt	Fa_H
MWS	Memory Write Select	$8F_H$
OCH	ADC Offset Calibration High	$D3_H$
OCL	ADC Offset Calibration Low	$D1_H$
OCM	ADC Offset Calibration Middle	$D2_H$
ODAC	Offset DAC	$E6_H$
P0	Port 0	80_H
P0DDRH	Port 0 Data Direction High	AD_H
P0DDRL	Port 0 Data Direction Low	AC_H
P1	Port 1	90_H
P1DDRH	Port 1 Data Direction High	AF_H
P1DDRL	Port 1 Data Direction Low	AE_H
P2	Port 2	$A0_H$
P2DDRH	Port 2 Data Direction High	$B2_H$
P2DDRL	Port 2 Data Direction Low	$B1_H$
P3	Port 3	$B0_H$
P3DDRH	Port 3 Data Direction High	$B4_H$
P3DDRL	Port 3 Data Direction Low	$B3_H$
PAI	Pending Auxiliary Interrupt	$A5_H$
PASEL	PSEN/ALE Select	$F2_H$
PCON	Power Control	87_H
PDCON	Power-Down Control	$F1_H$
PWMCON	PWM Control	$A1_H$
PWMHI	PWM High	$A3_H$
PWMLOW	PWM Low	$A2_H$
RCAP2H	Reload/Capture Timer 2 High	CB_H
RCAP2L	Reload/Capture Timer 2 Low	CA_H
SBUF0	Serial Buffer 0	99_H
SBUF1	Serial Buffer 1	$C1_H$
SCON0	Serial Control 0	98_H
SCON1	Serial Control 1	$C0_H$
SECINT	Seconds Interrupt	$F9_H$
SP	Stack Pointer	81_H
SPICON	SPI Control	$9A_H$
SPIDATA	SPI Data	$9B_H$
SPIEND	SPI Buffer End Address	$9F_H$

SPIRCON	SPI Receive Control	$9C_H$
SPISTART	SPI Buffer Start Address	$9E_H$
SPITCON	SPI Transmit Control	$9D_H$
SRST	System Reset	$F7_H$
SSCON	Summation/Shifter Control	$E1_H$
SSUMR0	Summation Register 0	$E2_H$
SSUMR1	Summation Register 1	$E3_H$
SSUMR2	Summation Register 2	$E4_H$
SSUMR3	Summation Register 3	$E5_H$
T2CON	Timer 2 Control	$C8_H$
TCON	Timer Control	88_H
TH0	Timer 0 High	$8C_H$
TH1	Timer 1 High	$8D_H$
TH2	Timer 2 High	CD_H
TL0	Timer 0 Low	$8A_H$
TL1	Timer 1 Low	$8B_H$
TL2	Timer 2 Low	$8C_H$
TMOD	Timer Mode	89_H
USEC	Microseconds	FB_H
WDTCON	Watchdog Timer Control	FF_H