CHAPTER 1

ARCHITECTURE

1.1. Microprocessors and Microcontrollers

A digital computer typically consists of three major components: the Central Processing Unit (CPU), program and data memory, and an Input/Output (I/O) system. The CPU controls the flow of information among the components of the computer. It also processes the data by performing digital operations. Most of the processing is done in the Arithmetic-Logic Unit (ALU) within the CPU. When the CPU of a computer is built on a single printed circuit board, the computer is called a minicomputer. A microprocessor is a CPU that is compacted into a single-chip semiconductor device. Microprocessors are general-purpose devices, suitable for many applications. A computer built around a microprocessor is called a microcomputer. The choice of I/O and memory devices of a microcomputer depends on the specific application. For example, most personal computers contain a keyboard and monitor as standard input and output devices.

A microcontroller is an entire computer manufactured on a single chip. Microcontrollers are usually dedicated devices embedded within an application. For example, microcontrollers are used as engine controllers in automobiles and as exposure and focus controllers in cameras. In order to serve these applications, they have a high concentration of on-chip facilities such as serial ports, parallel input-output ports, timers, counters, interrupt control, analog-to-digital converters, random access memory, read only memory, etc. The I/O, memory, and on-chip peripherals of a microcontroller are selected depending on the specifics of the target application. Since microcontrollers are powerful digital processors, the degree of control and programmability they provide significantly enhances the effectiveness of the application.

Embedded control applications also distinguish the microcontroller from its relative, the general-purpose microprocessor. Embedded systems often require real-time operation and multitasking capabilities. Real-time operation refers to the fact that the embedded controller must be able to receive and process the signals from its environment as they are received. That is, the environment must not wait for the controller to become available. Similarly, the controller must perform fast enough to output control signals to its environment when they are needed. Again, the environment must not wait for the controller. In other words, the embedded controller should not be a bottleneck in the operation of the system. Multitasking is the capability to perform many functions in a simultaneous or quasi-simultaneous manner.

The embedded controller is often responsible of monitoring several aspects of a system and responding accordingly when the need arises.

The 8051 is the first microcontroller of the MCS-51 family introduced by Intel Corporation at the end of the 1970s. The 8051 family with its many enhanced members enjoys the largest market share, estimated to be about 40%, among the various microcontroller architectures. The architecture of the 8051 family of the microcontrollers is presented in this chapter. First, the original 8051 microcontroller is discussed, followed by the enhanced features of the 8032, and the 80C515.

1.2. The 8051 Microcontroller Family Architecture

The architecture of the 8051 family of microcontrollers is referred to as the MCS-51 architecture, or sometimes simply as MCS-51. The microcontrollers have an 8-bit data bus. They are capable of addressing 64K of program memory and a separate 64K of data memory. The 8051 has 4K of code memory implemented as on-chip *Read Only Memory* (ROM). The 8051 has 128 bytes of internal *Random Access Memory* (RAM). The 8051 has two timer/counters, a serial port, 4 general purpose parallel input/output ports, and interrupt control logic with five sources of interrupts. Besides internal RAM, the 8051 has various *Special Function Registers* (SFR), which ·re the control and data registers for on-chip facilities. The SFRs also include the :cumulator, the B register, and the *Program Status Word* (PSW), which contains the ?U flags. Programming the various internal hardware facilities of the 8051 is achieved by placing the appropriate control words into the corresponding SFRs. The 8031 is similar to the 8051, except it lacks the on-chip ROM.

As stated, the 8051 can address 64K of external data memory and 64K of external program memory. These may be separate blocks of memory, so that up to 128K of memory can be attached to the microcontroller. Separate blocks of code and data memory are referred to as the Harvard architecture. The 8051 has two separate read signals, RD# (P3.7) and PSEN#. The first is activated when a byte is to be read from external data memory, the other, from external program memory. Both of these signals are so-called active low signals. That is, they are cleared to logic level 0 when activated. All external code is fetched from external program memory. In addition, bytes from external program memory may be read by special read instructions such as the MOVC instruction. There are separate instructions to read from external data memory, such as the MOVX instruction. That is, the instructions determine which block of memory is addressed, and the corresponding control signal, either RD# or PSEN# is activated during the memory read cycle. A single block of memory may be mapped to act as both data and program memory. This is referred to as the Von Neumann[1] architecture. In order to read from the same block using either the RD#

[1] Named after John Von Neumann, a prolific mathematician and computer scientist of the 20th century.

signal or the PSEN# signal, the two signals are combined with a logic AND operation. This way, the output of the AND gate is low when either input is low. The advantage of the Harvard architecture is not simply doubling the memory capacity of the microcontroller. Separating program and data increases the reliability of the microcontroller, since there are no instructions to write to the program memory. A ROM device is ideally suited to serve as program memory. The Harvard architecture is somewhat awkward in evaluation systems, where code needs to be loaded into program memory. By adopting the Von Neumann architecture, code may be written to memory as data bytes, and then executed as program instructions.

The 8052 has 256 bytes of internal RAM and 8K of internal code ROM. The 8051 and 8052 internal ROM cannot be programmed by the user. The user must supply the program to the manufacturer, and the manufacturer programs the microcontrollers during production. Due to the setup costs, the factory masked ROM option is not economical for small quantity productions. The 8751 and 8752 are the *Erasable Programmable Read Only Memory* (EPROM) versions of the 8051 and 8052. Many manufacturers offer the EPROM versions in windowed ceramic and non-windowed plastic packages. These are user programmable. However, the non-windowed versions cannot be erased. These are usually referred to as One-Time-Programmable (OTP) microcontrollers, which are more suitable for experimental work or for small production runs. The 8951 and 8952 contain FLASH EEPROMs (Electrically Erasable Programmable Read Only Memory). These chips can be programmed as the EPROM versions, using a chip programmer. Moreover, the memory may be erased. Similar to EPROMs, Erasing FLASH memory sets all data bits (data bytes become FFh). A bit may be cleared (made 0) by programming. However, a zero bit may not be programmed to a one. This requires erasing the chip. Some larger FLASH memories are organized in banks or sectors. Rather than erasing the entire chip, you may erase a given sector and keep the remaining sectors unchanged.

During the past decade, many manufacturers introduced enhanced members of the 8051 microcontroller. The enhancements include more memory, more ports, analog-to-digital converters, more timers with compare, reload and capture facilities, more interrupt sources, higher precision multiply and divide units, idle and power down mode support, watchdog timers, and network communication subsystems. All microcontroller of the family use the same set of machine instructions, the MCS-51. The enhanced features are programmed and controlled by additional SFRs. In the remainder of this chapter, the hardware architecture of the 8051 is presented. The enhancements brought by the 8052 and 80C515 follow. Some of the more popular enhanced members of the family are reviewed at the end of Chapter 2. The reader is referred to the manufacturers' data books for the specifics of other enhanced members.

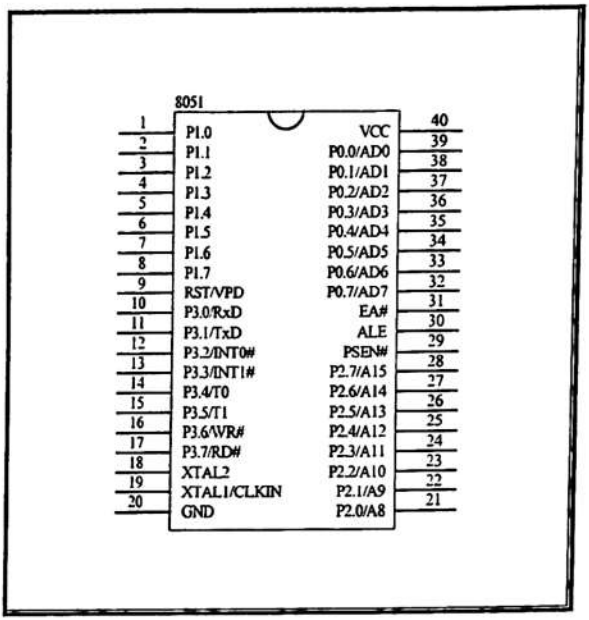


Figure 1.1. Pinout of the 8051 Microcontroller (NC stands for No Connection).

If the program fits into the on-chip ROM and if the internal RAM is sufficient, the MCS-51 family of microcontrollers requires no additional logic to implement a complete controller system. The following discusses the 8051 in detail.

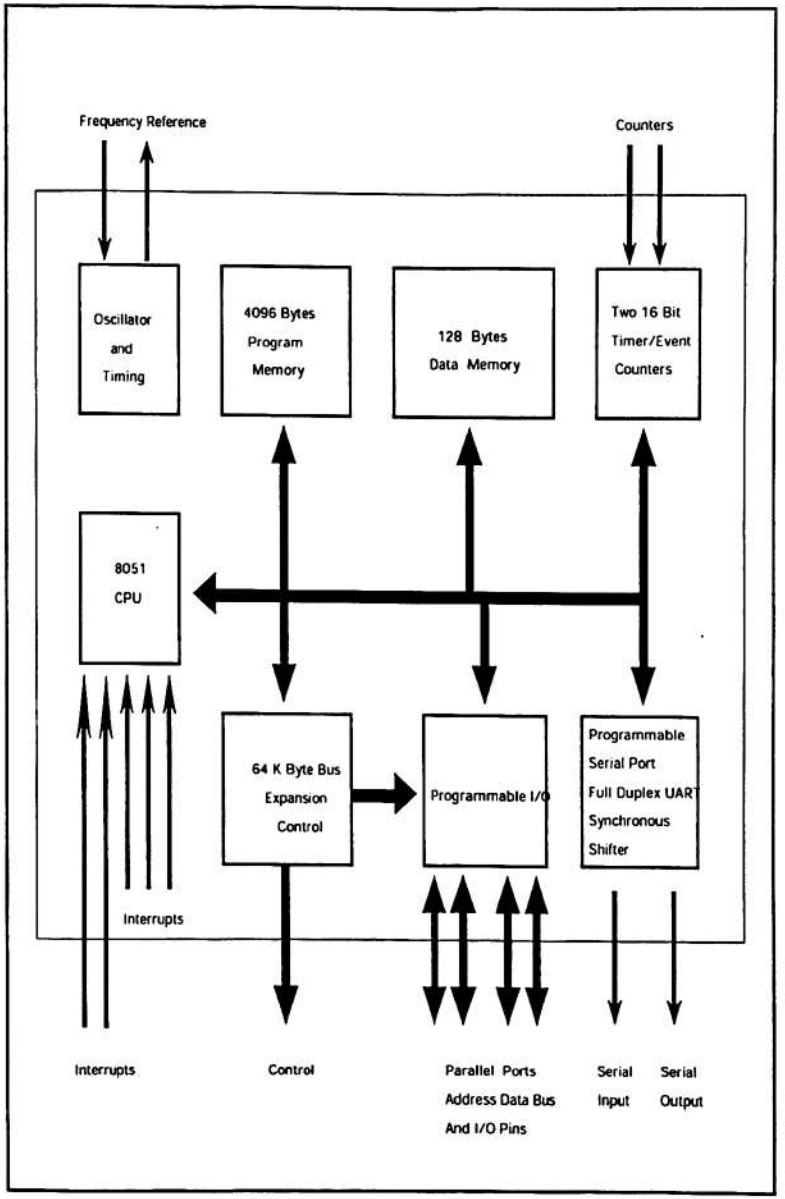


Figure 1.2. The Block Diagram of the 8051 Microcontroller.

1.2.1. On-Chip Memory

1.2.1.1. Internal Random Access Memory

The internal RAM of the 8051 contains the registers and the bit addressable registers, as well as general purpose RAM, as illustrated below.

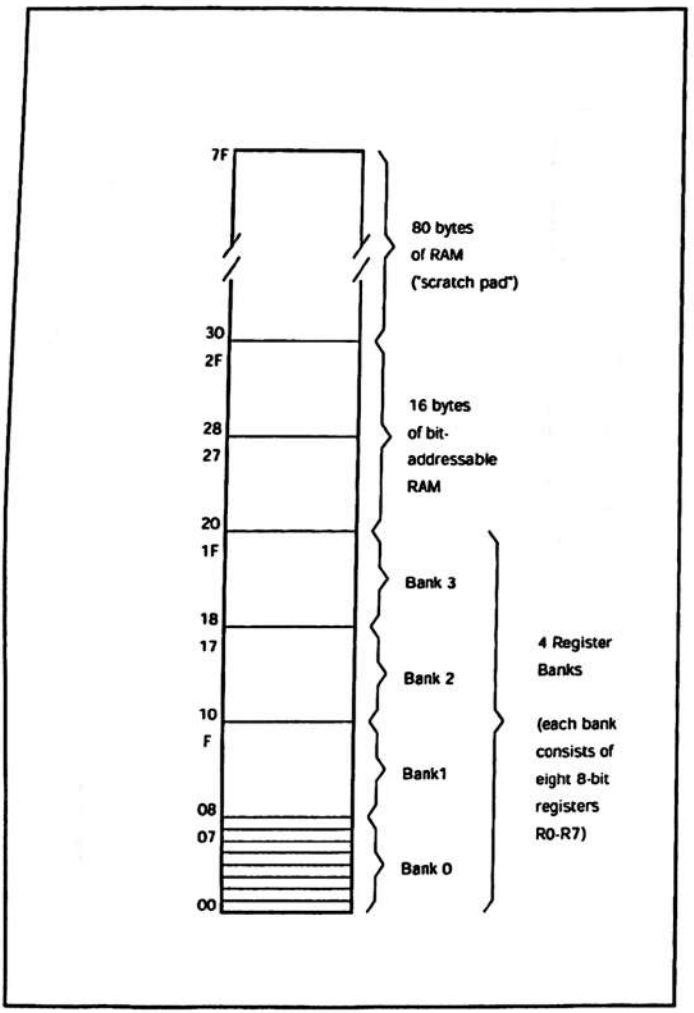


Figure 1.3. The Memory Map of the 8051 Internal RAM.

Random access memory is used as a general-purpose scratch pad. Memory location addresses range from 0 to 7Fh. The MOV instruction transfers data to or from the internal RAM. Direct or indirect addressing is allowed. For example, the instruction

```
mov a, 35h
```

will move the contents of the internal RAM byte 35h to the accumulator. As will be discussed later, the 8051 assembly language reserves the pound sign (#) to specify constants. The instruction

```
mov a, #35h
```

is thus different from the previous instruction in that it will load the accumulator with the constant 35h. As an example of indirect addressing,

```
mov a, @R0
```

moves the content of the internal RAM byte whose address is stored in REGISTER 0 to the accumulator. Rn stands for REGISTER n, where n is a number from 0 to 7. In the above example, R0 should contain a valid address. The 8051 has 128 bytes of memory, so R0 should contain a number between 0 and 7FH. Starting with the 8052, the enhanced members of the family have 256 bytes of internal memory, so R0 may be in the range of 0 to FFh. This range is compactly denoted by the notation [0..FFh]. Only registers R0 and R1 of the currently selected register bank can be used for indirect addressing.

1.2.1.2. Registers

Although all internal RAM locations are accessible as described above, two blocks of internal RAM can be accessed in different modes. The block of internal RAM [0..1FH] is used for the registers. Specifically, there are four banks, with eight registers in each bank. The registers of a bank are denoted referred to as R0 to R7. Register bank 0, 1, 2, and 3 contain internal RAM locations [0..7], [8..FH], [10H..17H], and [18H..1FH], respectively. In order to address a register, first the register bank it belongs to must be selected as the active register bank. Such selection is done by flags in the PSW. Once the associated register bank is selected, Register n is addressed simply as Rn. Thus, the instruction

```
mov a, R5
```

for example, moves into the accumulator the contents of REGISTER 5, which, depending on the register bank currently selected, may be internal RAM byte 5, 0Dh, 15h, or 1Dh. It is often a good idea to dedicate register banks to key subroutines or interrupt service routines.

Internal Data RAM								Bank	Register Addressing							
1F	1E	1D	1C	1B	1A	19	18	3	R7	R6	R5	R4	R3	R2	R1	R0
17	16	15	14	13	12	11	10	2	R7	R6	R5	R4	R3	R2	R1	R0
0F	0E	0D	0C	0B	0A	09	08	1	R7	R6	R5	R4	R3	R2	R1	R0
07	06	05	04	03	02	01	00	0	R7	R6	R5	R4	R3	R2	R1	R0

Figure 1.4. Register Banks.

The active register bank is selected by a two-bit field in the Program Status Word (PSW).

1.2.1.3. Bit-Mapped Memory

The block of memory, in the range [20h..2Fh] is bit-addressable. Notice that there are 128 bits in this block. Each bit has a bit address from 0 to 7Fh. Bit 0 of byte 20h has bit address 0 and bit 7 of byte 2Fh has bit address 7Fh. For example, the instruction

```
mov C, 13h
```

moves the Boolean content of bit 13H into the *Carry Flag*. Bit 13h is the third digit of the internal RAM byte at location 34, or 22h. The 8051 assembly language allows the alternative notation 34.3 (or 22h.3) to refer to the third bit of byte 22h. The instruction

```
mov C, 22h.3
```

has the same effect as the above instruction. There are other bit-addressable SFRs. The bit addresses of the SFR bits are in the range [80h..FFh].

1.2.1.4. Special Function Registers

SFRs contain data and control registers. Each on-chip facility such as the timers, the serial port, and the interrupt system, has one or more dedicated SFRs. The serial port, for example, is controlled by the SFR SCON, while its associated data is read from or written to the SFR SBUF. Individual bits of SCON set the different modes of the serial port. SCON is referred to as a control register, while SBUF is called a data register.

SFRs have byte addresses in the range [80h..FFh]. The 8051 has 128 bytes of internal RAM which occupy the addresses [0..7Fh]. The SFRs are mapped into the next block of 128 bytes, in the range of [80h..FFh]. The 8051 may access internal RAM and SFRs by the direct addressing mode described in Section 1.2.1.1. Note that the internal RAM is also accessible through register indirect addressing. The enhanced members of the 8051 family, starting with the 8052, have 256 bytes of internal RAM. Although the RAM addresses range from 0 to FFh, it should be considered as two blocks of memory, each block of the size 128 bytes. The lower block is the same as the 8051 internal RAM, accessible either through direct

addressing or through register indirect addressing. The higher block of 128 bytes is accessible only through register indirect addressing. Thus, the instruction

```
mov a, 80h
```

moves the value stored in the SFR 80h into the accumulator, while the instruction

```
mov a, @R0
```

where R0 is previously set to 80h moves the contents of the internal RAM location 80h into the accumulator.

The accumulator, the B register, and the PSW are contained in the set of SFRs. PSW contains the system flags, such as the carry flag, as well as a two-bit field to select the active register bank. Following is a list of the 8051 SFRs.

Register	Name (mnemonic)	Internal Address	Reset Value	Notes
Port 0 Latch	P0	80h	FFh	
Stack Pointer	SP	81h	07h	
Data Pointer (as a word)	DPTR	82h-83h	00h	
Data Pointer Low Byte	DPL	82h	00h	
Data Pointer High Byte	DPH	83h	00h	
Power Control	PCON	87h	00h	1, 2
Timer/Counter Control	TCON	88h	00h	
Timer/Counter Mode Control	TMOD	89h	00h	
Timer/Counter 0 Low Byte	TL0	8Ah	00h	
Timer/Counter 1 Low Byte	TL1	8Bh	00h	
Timer/Counter 0 High Byte	TH0	8Ch	00h	
Timer/Counter 1 High Byte	TH1	8Dh	00h	
Port 1 Latch	P1	90h	FFh	
Serial Port Control	SCON	98h	00h	
Serial Data Port	SBUF	99h	Undefined	
Port 2 Latch	P2	A0h	FFh	
Interrupt Enable	IE	A8h	00h	1
Port 3 Latch	P3	B0h	FFh	
Interrupt Priority Control	IP	B8h	00h	1
Timer/Counter 2 Control	T2CON	C8h	00h	2
Reload/Capture 2 Low Byte	RCAP2L	CAh	00h	2
Reload/Capture 2 High Byte	RCAP2H	CBh	00h	2
Timer/Counter 2 Low Byte	TL2	CCh	00h	2
Timer/Counter 2 High Byte	TH2	CDh	00h	2
Program Status Word	PSW	D0h	00h	
Accumulator	ACC or A	E0h	00h	
B Register	B	F0h	00h	

Figure 1.5. SFRs of the 8051/8052

Notes:

1. The SFRs contain unused bits. When read immediately after reset, these bits are read as 1's. For example, IE register of the 8052 would read as 40h after reset.
2. These SFRs are implemented in the 8052 but not in the 8051.

Of the SFRs, SP, DPTR, DPL, DPH, PCON, SBUF, TMOD, TL0, TL1, TH0, and TH1 are only byte-addressable. That is, these SFRs can only be read, written to, operated on, and compared as full bytes. The remaining SFRs, ACC, B, PSW, P0, P1, P2, P3, IE, IP, SCON, and TCON are also bit-addressable, meaning that their individual bits can also be read, written to, operated on, and compared. Notice that the (byte) addresses of the bit-addressable SFR are multiples of eight. The bit addresses of their individual bits are computed as the byte address of the SFR plus the digit of the

bit within the SFR. For example, the bit address of the 5-th bit of the accumulator is E5H, and the 0-th bit of the serial port control register is 98H.

1.2.1.5. Read Only Memory

Some members of the 8051 family have on-chip ROM. The 8051 and 8052 have 4K (kilobytes) and 8K of factory masked ROM. The 80C515A-5 and the 80C517A-5 have 32K of on-chip ROM. The 8751 and 8752 are EPROM versions of the 8051 and 8052. Similarly, the 8951 and 8952 are FLASH EEPROM versions of the 8051 and 8052. Internal ROM occupies the lowest block of program memory. The 8031, 8032, 80C535A, and the 80C537A are the ROM-less versions of the 8051, 8052, 80C515A, and the 80C517A. The EA# (External Enable) pin of the microcontroller is tied to +5 volts to allow the program to be fetched directly from internal ROM. Of course, any program that resides above the 4K or 8K block is fetched from external program memory. If EA# is connected to ground (0 volts), then all of the program is fetched from external memory. The pound sign (#), when used as a suffix signifies that the signal is active when low. For example, external memory is enabled when the EA# line is held at logic low (close to ground voltage).

1.2.1.6. Program Counter

The *Program Counter* (PC) is internally used to point to the next instruction byte to be fetched. It is not directly accessible. However, it is modified by the branching instructions such as jump (JMP) and call (CALL). It can also be used as a base address for indexed addressing when reading from the program memory.

1.2.1.7. Program Status Word

As mentioned, the *Program Status Word* (PSW) holds the CPU flags. PSW is at address 0D0h in the SFR space. It is noteworthy that almost all microprocessors and microcontrollers have a flags register. In this respect, the PSW is regarded as a standard aspect of the CPU rather than a peripheral.

The PSW contains bit-wide status information about the CPU. For example, an external carry generated by an addition is placed in one of the PSW flags.

Bit	7	6	5	4	3	2	1	0
Flag	CY	AC	F0	RS1	RS0	OV	F1	P
Name	Carry Flag	Auxiliary Carry Flag	User Flag 0	Register Bank Select 1	Register Bank Select 0	Overflow Flag	User Flag 1	Parity Flag

Figure 1.6. The Fields of the Program Status Word (PSW).

The PSW contains two user flags F0 and F1. Lastly, the PSW contains a two-bit control field that selects the active register bank.

1.2.2. Using the On-Chip Facilities

1.2.2.1. Parallel Input/Output Ports

The 8051 has four parallel input/output ports. When a port is to be used as an output port, the data is put in the corresponding SFR. More specifically, the value written to the corresponding SFR is forwarded to a latch, which continues to emit the signal after the write operation is completed. The value of an output port is changed when a new value is latched. When a port is to be used as an input port, the value FFH must first be written to the port. Then, any input that pulls the pin voltage low will be recognized as a zero. The port can then be read from the corresponding SFR. More specifically, reading the corresponding SFR reads the value of the port pin. The latched output drives the port pin to logic level 1 if there is no external circuit sinking the current on the pin.

Notice that although the same SFR is used, internally there are two distinct operations carried out when a port is written to, and when it is read. These operations are carried out entirely in hardware, shielding the user from keeping track of the direction in which data is moved.

It should be mentioned that although all read operations read the value on the port pins, some implicit read operations read the value of the port latch. These are called the *read-modify-write* operations. For example the instruction

```
inc P1
```

reads the contents of the port latch, increments the value, and writes the result back to the port. Again, these operations are carried out entirely in hardware. Several operations on port SFRs are read-modify-write instructions. These instructions are ANL, ORL, XRL, INC, DEC, and DJNZ, as well as the bit-oriented operations JBC, CPL, MOV, CLR, and SETB. The bitwise move instruction,

```
mov P1.0, C
```

for example, moves the value of the carry flag (C) into bit 0 of Port 1. This operation is carried out in hardware in two parts: Port 1 is read from its latch, and after its zero-th bit is modified, it is written back to the latch.

Ports 0, 2, and 3 of the 8051 have alternative functions. That is, the individual pins of these ports may be used as general digital input/output lines, or alternatively, be used for their secondary functions. The secondary function of Ports 0 and 2 is to interface with external memory. When external program or data memory is accessed, Port 2 outputs the high byte of the 16-bit address. Port 0 first outputs the low byte of the 16-bit address, and then sends or receives the data byte. The address low byte must be

latched externally. The microcontroller signals a valid address by activating the ALE pin, which is used to latch the address low byte.

Alternative functions of Port 3 pins include interrupt and timer inputs, serial port input and output, and control signals for interfacing with external memory. The following list summarizes the alternative functions of Port 3.

Bit	Alternate Function	Mnemonic/Designation
0	Serial Input Port	RXD
1	Serial Output Port	TXD
2	External Interrupt 0	INT0#
3	External Interrupt 1	INT1#
4	Timer/Counter 0 External Input	T0
5	Timer/Counter 1 External Input	T1
6	External Memory Write Strobe	WR#
7	External Memory Read Strobe	RD#

In order to implement the alternative function, the corresponding SFR bit must be set (made equal to 1).

1.2.2.2. System Clock Generator

The 8051 contains hardware to generate a system clock (oscillator) using an external crystal and two external capacitors. The oscillator frequency is the same as the crystal frequency. The oscillator frequency divided by 12 (prescaled by 12) is used as an input by the on-chip timers. Using a 12MHz crystal, the timers increment at 1MHz.

The standard 8051 uses 12 oscillator cycles per machine cycle. The 8051 has 255 operation codes grouped as 111 instructions. For example, there are 15 different operation codes for byte-oriented move instructions depending on the source and destination bytes and the addressing mode. In addition, there are two bit-oriented move instructions and one word oriented move instruction, namely the instruction that moves a 16-bit constant into the data pointer. Of these 255 operation codes, 159 take one machine cycle to complete, 51 take two machine cycles, 43 take three machine cycles, and 2 take four machine cycles. Since many of the frequently used instructions take only one machine cycle, the 8051 is considered to be capable of roughly 1 Million Instructions Per Second (MIPS). Many 8051-based systems use a crystal frequency of 11.0592MHz. This choice is because many high Baud rates can be generated by this clock frequency. With a 12MHz clock, a maximum of 4800 Baud is the practical limit. Baud rates up to 57600 can be generated using an 11.0592MHz crystal.

1.2.2.3. Serial Port

The serial port is controlled by the SFR SCON. Data to and from the serial port is channeled through the SFR SBUF. Once the serial port is configured, simply writing a

byte to SBUF initiates the serial transmission. Similarly, a received byte is read from SBUF. Note that although SBUF appears as a single SFR, its hardware implementation contains two separate buffers, one for transmission and one for receiving the bytes. A serial transmission and a serial receive may take place simultaneously, which in serial communications is referred to as a full duplex operation. There are four modes of operation, which determine the source of the transmission rate and the number of bits per data word transmitted.

The serial transmission rate, also referred to as the BAUD rate, may be derived from the system clock or from Timer 1, in case of the 8051. If Timer 1 is put in the 8-Bit Auto-Reload Mode (Mode 2), then the Baud rate is given by the following formula.

$$\text{Baud Rate} = \frac{2^{\text{SMOD}} \cdot (\text{Oscillator Frequency})}{384 \cdot (256\text{-TH1})}$$

SMOD is bit 7 of the Power Control Register PCON and TH1 is a SFR. The quantity 2^{SMOD} is two to the power SMOD; if SMOD is cleared then the quantity is 1, if set, it is 2. The bit-addressable control register SCON contains the following bits.

Bit	7	6	5	4	3	2	1	0
Field	SM0	SM1	SM2	REN	TB8	RB8	TI	RI
Name	Serial Mode			Receive Enable	8-th Transmit Bit	8-th Receive Bit	Transmit Interrupt Flag	Receive Interrupt Flag

Figure 1.7. The Fields of the SCON Register.

RI (bit 0): Receive Interrupt Flag. Set by hardware to signal serial receive completion; must be cleared by software.

TI (bit 1): Transmit Interrupt Flag. Set by hardware to signal serial transmit completion; must be cleared by software.

RB8 (bit 2): Receive Bit 8. Its use depends on the mode of operation. In modes 2 and 3 where 9 bits of data are received, the last bit is copied into RB8. In mode 1, where eight bits of data is transmitted, provided that SM2 (bit 5 of SCON) is cleared, the stop bit is copied into RB8.

TB8 (bit 3): Transmit Bit 8. The 9-th data bit to be transmitted in modes 2 and 3. Set or cleared by software as desired.

REN (bit 4): Receive Enable. Set by software to enable serial reception. Serial reception will be blocked if REN is cleared by software.

SM2 (bit 5): Serial Mode (Bit 2). Used in modes 2 or 3 to facilitate multiprocessor communications. Refer to Siemens data books for further discussion.

SM1 and SM0: Serial Mode (bits 6 and 7 of SCON, respectively). The two-bit number (SM1, SM0) gives the operating mode. Mode 0 runs the serial port as a simple shift register with a shift-out rate equal to one

machine cycle or 1/12 of the crystal frequency. Mode 1 uses internal Timer 1 to generate the BAUD rates to transmit and receive eight bits of data. Mode 2 implements a 9-bit serial port with a Baud rate of 1/32-th or 1/64-th of the crystal frequency. Mode 3 implements a 9-bit serial port with a Baud rate determined by Timer 1.

1.2.2.4. Timers

The 8051 has two internal timers, referred to as Timer 0 and Timer 1. Each timer has two SFRs dedicated to it. Associated with Timer 0 are TH0 and TL0, and with Timer 1, TH1 and TL1. These registers make the timer count 16 bits wide. The notation TH0:TL0 is used to denote the register pair holding thetwo0byte value. In operation, at each input pulse, the 16-bit count stored by THn:TLn is incremented (n=0,1). A timer uses the system clock as the source of its input pulses. Thus timer increments are periodical while a counter uses external pulses to increment its count. The external pulses are received through two of the bits of Port 3, which, as alternative functions, are assigned to the two timers. The current count of a timer can be read from THn and TLn (n=0,1). The timers also generate interrupts, provided that the associated interrupts are not masked by the Interrupt Enable (IE) SFR. When the count overflows (and restarts, i.e., rolls over to zero), an interrupt is generated. The status of the associated interrupts is observable through the SFR TCON. The following is a simplified block diagram of Timer 0.

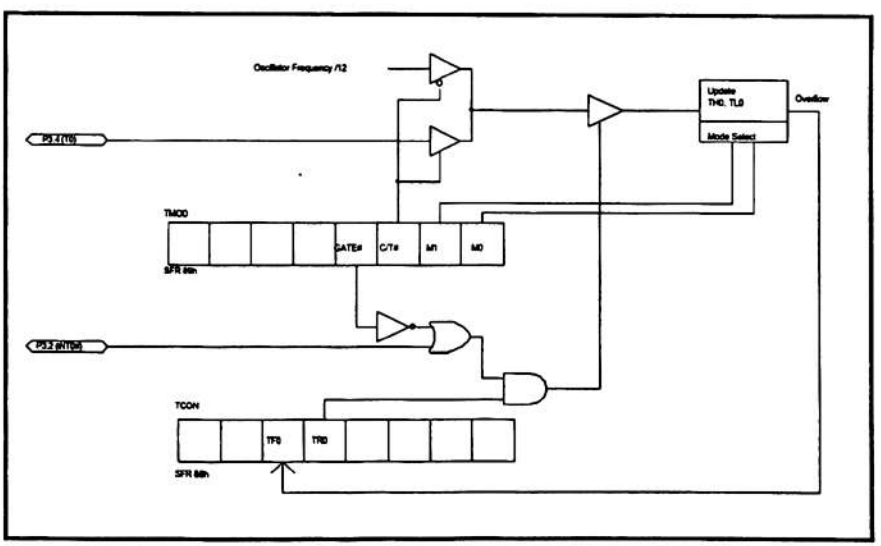


Figure 1.8. Simplified Block Diagram of Timer 0.

An 8051 timer/counter is perhaps best explained by partitioning its operation into three stages:

1. input source
2. operation control
3. update mode.

The input may be either 1/12-th the oscillator frequency or pulses received at one of the Port3 pins T0 (P3.4) for Timer 0 and T1 (P3.5) for Timer 1. The input source is selected by the control bit C/T# of the TMOD register. There are actually two such control bits, one for each timer.

The operation of the timer/counter is controlled by blocking or passing the pulses from the source to the count data registers. The source is forwarded to the Timer 0 data registers if the condition

(GATE# or INT0#) and TR0

is true. GATE# is the Timer 0 control bit of TMOD. The 8051 refers to the control bit simply as GATE. However, since the timer counter operates when GATE is at logic level low, to be consistent with our notation, we denote this control bit as GATE#. TR0 is also a Timer 0 control bit, found in the TMOD register. INT0# is pin P3.2 of the 8051. This condition specifies that Timer 0 data registers are updated according to the source as long as the control bit TR0 is set, and either the control bit GATE# is cleared, or Port P3.2 is at logic level 1. The notation INT0# is a little confusing, since the condition requires INT0# to be at logic level 1 to function. This is because P3.2 actually has two alternate functions, external interrupt 0 (INT0#) and external control signal for Timer 0. Its first alternate function is activated by a logic level 0 or a transition from 1 to 0. The pin is labeled for its first alternate function INT0#. When the port bit is used to control Timer 1, it enables the timer operation when at logic level 1.

TR0 can be interpreted as the master software switch that may turn off Timer 0 irrespective of any other condition. Once TR0 is set, the timer may be run either by software, by clearing the control bit GATE# or by hardware, by bringing P3.2 to the logic level 1.

There are four operating modes of the timer/counters. The operating mode of both timers is determined by the SFR TMOD. The low nibble of TMOD is associated with Timer 0, and the high, with Timer 1. The low nibble bits of TMOD are described below.

	Timer 1				Timer 0			
Bit	7	6	5	4	3	2	1	0
Name	GATE#	C/T#	M1	M0	GATE#	C/T#	M1	M0

Figure 1.9. The Fields of the TMOD Register.

M0 and M1: (bits 0 and 1 of TMOD, respectively): Mode Select. The two-bit field (M1, M0) selects one of four modes. Mode 0 is a 13-bit counter. An interrupt is generated when the counter overflows. Thus, it takes 2^{13} or 8192 input pulses to generate the next interrupt. Mode 1, similar to Mode 0, implements a 16-bit counter. It takes 2^{16} or 65526 input pulses to generate the next interrupt. Mode 2 operates in an 8-bit reload fashion. TLn serves as an 8-bit timer/counter. When the count overflows, the number stored in THn is copied into TLn, and the count continues. An interrupt is generated each time counter overflows and a reload is performed. In Mode 3, Timer 1 is inactive and simply holds its count. Timer 0 operates as two separate 8-bit timers. TL0 is controlled by Timer 0 control bits. TH0 also generates a Timer 0 interrupt at overflow. TH0 operates as a timer driven by the system clock, prescaled by 12, and causes a Timer 1 interrupt at overflow.

C/T# (bit 2): Counter/Timer Select. When set the timer/counter operates as a counter. The count increment is caused by external pulses through the T0 pin - the alternate function of P3.4. When cleared the timer/counter operates as a timer. The count increment is caused by every 12-th system clock pulse. That is, the input to the counter is the system clock prescaled by 12.

GATE# (bit 3): Gate. Provided that TR0 of TCON (see below) is set, clearing GATE enables (starts) counter/timer operation. If GATE is set, again provided that TR0 is set, the timer/counter operation is enabled if INT0# is high.

Bits of the TCON SFR are described next. The low nibble of TCON is related to the control of external interrupts.

Bit	7	6	5	4	3	2	1	0
Name	TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0

Figure 1.10. The Fields of the TCON Register.

IT0 (bit 0): Interrupt 0 Type. External interrupt 0 is received through bit 2 of Port 3 as an alternative function assignment. Setting IT0 causes the INT0# to be recognized at the falling edge of the input signal.

Clearing IT0 causes an interrupt to be generated when the external signal is low. If the external signal stays low, INT0 will be generated repeatedly if IT0 is cleared, but will not be generated if IT0 is set. IT0 is completely under software control.

IE0 (bit 1): Interrupt 0 Edge Flag. Set by hardware when an external interrupt edge is detected. Cleared when a Return from Interrupt (RETI) instruction is executed.

IT1 (bit 2): Interrupt 1 Type. IT1 is associated with External Interrupt 1. Its function is similar to IT0 as described above.

IE1 (bit 3): Interrupt 1 Edge Flag. IE1 is associated with External Interrupt 1. Its function is similar to IE0 as described above.

TR0 (bit 4): Timer 0 Run Control Bit. Timer/Counter 0 is disabled when TR0 is cleared. Setting TR0 is necessary but not sufficient to enable Timer/Counter 0 (see GATE# and INT0#). TR0 is completely under software control.

TF0 (bit 5): Timer 0 Overflow Flag. TF0 is set by hardware when Timer/Counter 0 overflows. TF0 is cleared by hardware when the processor branches to the associated interrupt service routine. TF0 along with IE0 may be inspected by software to determine the state of the timer.

TR1 (bit 6): Timer 1 Run Control Bit. TR1 is associated with Timer/Counter 1. Its function is similar to TR0 described above.

TF1 (bit 7): Timer 0 Overflow Flag. TF1 is associated with Timer/Counter 1. Its function is similar to TF0 described above.

1.2.2.5. Interrupt Control

The 8051 has five sources of interrupts: TF0, TF1, INT0#, INT1#, and the serial port events. TF0 and TF1 of the TCON register constitute the two timer/counter interrupts. TF0 and TF1 are generated when the associated counter overflows. INT0# and INT1# constitute the two external interrupts. These interrupts are caused by external signals received through bits 2 and 3 of Port 3 as alternative functions. The interrupt control hardware may be programmed to respond either to the falling edge of the external signals or to the low level of the external signals. This selection is determined by the control bits IT0 and IT1 of TCON SFR. The final source of interrupts is from the serial port. The interrupt flags RI and TI of SCON SFR are combined by an OR gate so that either flag may generate an interrupt. It is the responsibility of software to check both the TI and RI flag to determine which caused the interrupt.

There are two SFRs associated with interrupt control. The Interrupt Enable Register (IE) is used to mask individual interrupts. Interrupt Priority Register (IP) assigns either a high priority or a low priority to each of the five interrupt sources. When two interrupt sources are programmed to the same priority level, priorities are determined

by the following order. IE0 has the highest priority, followed by, TF0, IE1, TF1, and finally by RI-or-TI.

Provided that it is not masked (disabled), the processor acknowledges an interrupt and branches to pre-specified locations in program memory by a LCALL instruction.

Many general-purpose CPUs automatically push the flags register on to the stack before branching to an Interrupt Service Routine (ISR). The 8051 does not push PSW or any other SFR. It is the responsibility of the programmer to push and pop the registers critical to the application within the ISR. Below are the fixed memory locations where interrupt service routines must start. The last two interrupts are actually the OR-ed signal from two separate sources. Provided that it is enabled, interrupt RI+TI is invoked if either the receive interrupt flag, or the transmit interrupt flag (or both) are set. The last interrupt exists in the 8052 but not in the 8051. It is related to the extra timer, Timer 2, of the 8052. TF2+EXF2 is invoked if at least one of the sources, Timer 2 overflow, or External interrupt 2 is invoked.

Interrupt	Service Routine Address	Default Priority
IE0	3H	1 (Highest)
TF0	BH	2
IE1	13H	3
TF1	1BH	4
RI+TI	23H	5
TF2+EXF2	2Bh	6 (8052 only)

Descriptions of the components of the bit-addressable IE and IP SFRs follow.

Bit	7	6	5	4	3	2	1	0
Name	EA	-	ET2	ES	ET1	EX1	ET0	EX0

Figure 1.11. The Fields of the IE Register.
(ET2 is implemented in the 8052 but not in the 8051.)

EX0 (bit 0): Enable External Interrupt 0. Set by software to enable external interrupt 0, cleared to disable.

ET0 (bit 1): Enable Timer/Counter Interrupt 0. Set by software to enable timer/counter overflow interrupt 0, cleared to disable.

EX1 (bit 2): Enable External Interrupt 1. Set by software to enable external interrupt 1, cleared to disable.

ET1 (bit 3): Enable Timer/Counter Interrupt 1. Set by software to enable timer/counter overflow interrupt 1, cleared to disable.

ES (bit 4): Enable Serial Port Interrupt. Set by software to enable serial port interrupt 1, cleared to disable.

ET2 (bit 5): Enable Timer/Counter Interrupt 2. Set by software to enable timer/counter overflow interrupt 2, cleared to disable.

EA (bit 7): Enable All. When cleared, all interrupts are masked (disabled). When set, each interrupt is enabled or disabled depending on its individual interrupt enable control bit.

Interrupts have default priorities, as shown in Section 1.2.2.5. These priorities may be changed by the Interrupt Priority (IP) register. More specifically, setting the corresponding bit in IP places the interrupt in the set of high interrupts. The priority among the high priority interrupts is resolved again by the default ordering given in Section 1.2.2.5.

Bit	7	6	5	4	3	2	1	0
Name	-	-	PT2	PS	PT1	PX1	PT0	PX0

Figure 1.12. The Fields of the IP Register.
(PT2 is implemented in the 8052 but not in the 8051.)

PX0 (bit 0): Priority of External Interrupt 0. External interrupt 0 receives a high priority level when PX0 is set.

PT0 (bit 1): Priority of Timer/Counter Interrupt 0. Timer/counter interrupt 0 receives a high priority level when PT0 is set.

PX1 (bit 2): Priority of External Interrupt 1. External interrupt 1 receives a high priority level when PX1 is set.

PT1 (bit 3): Priority of Timer/Counter Interrupt 1. Timer/counter interrupt 1 receives a high priority level when PT1 is set.

PS (bit 4): Priority of Serial Port Interrupt. Serial port interrupts receive a high priority level when PS is set.

PT2 (bit 4): Priority of Timer/Counter Interrupt 2. Timer/counter interrupt 2 receives a high priority level when PT2 is set.

1.2.2.6. The 8052 Microcontroller

The first enhanced members of the 8051 family were the 8032 and the 8052. The difference between the 8051 and the 8052 is minimal. The 8052 has 256 bytes of internal RAM as compared to the 128 bytes of the 8051. In addition, the 8052 implements an additional timer, called Timer 2. Timer 2 also provides a sixth interrupt source. The 8052 has additional on-chip ROM, similar to the 8032.

The upper half of the internal RAM space [128 to 255] can only be addressed by the register indirect addressing mode. For example,

```
mov   R0, #90h
mov   a, @R0
```

first places the constant 90h in the R0 register and then uses this as the address of the register to be read into the accumulator. The direct addressing mode,

```
mov    a, 90h
```

would transfer the contents of the SFR 90h into the accumulator. The SFR 90h is Port 1 (see Figure 1.5.). Thus, the direct addressing mode would read the value of P1, not the internal RAM location 90h.

Timer 2 differs from Timers 0 and 1. It is a timer with 16-bit capture and reload capabilities. Timer 2 has a separate control register. In addition, Timer 2 may issue an interrupt upon overflow. Thus, the 8052 has one more interrupt source compared to the 8051. This interrupt is active in both the capture and compare modes.

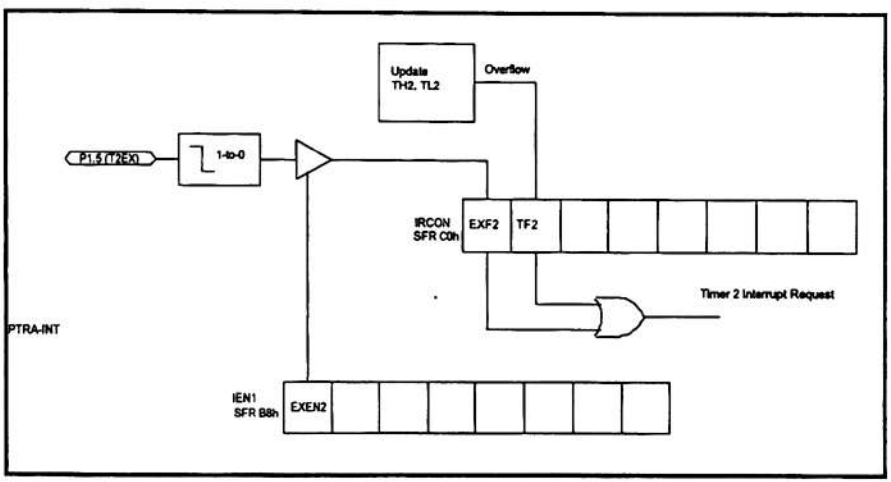


Figure 1.13. The Timer 2 Interrupt.

Two of the Port 1 bits, P1.0 (T2) and P1.1 (T2EX), are implemented as the counter input for Timer 2 and the external interrupt line. The sixth interrupt is actually the Timer 2 overflow bit (TF2) and the external input T2EX logically OR-ed. When both interrupt sources are present, it is left to software to investigate whether an overflow or an external signal caused the interrupt. The 8052 can use Timer 2 to generate the Baud rates for the serial port. The additional timer also enables the 8052 to generate different Baud rates for the serial-transmit and serial-receive operations, when used with another timer. Associated with Timer 2 are four data registers and one control register. The data registers are the low byte (TL2) and high byte (TH2) that jointly constitute the 16-bit register TH2:TL2, and the two auto-reload/capture registers RCAP2L and RCAP2H. These registers are used either to reload Timer 2 upon

overflow or external signal, or to save the current count in TL2 and TH2 upon an external signal. The external signal in either case is a high-to-low transition on pin T2EX.

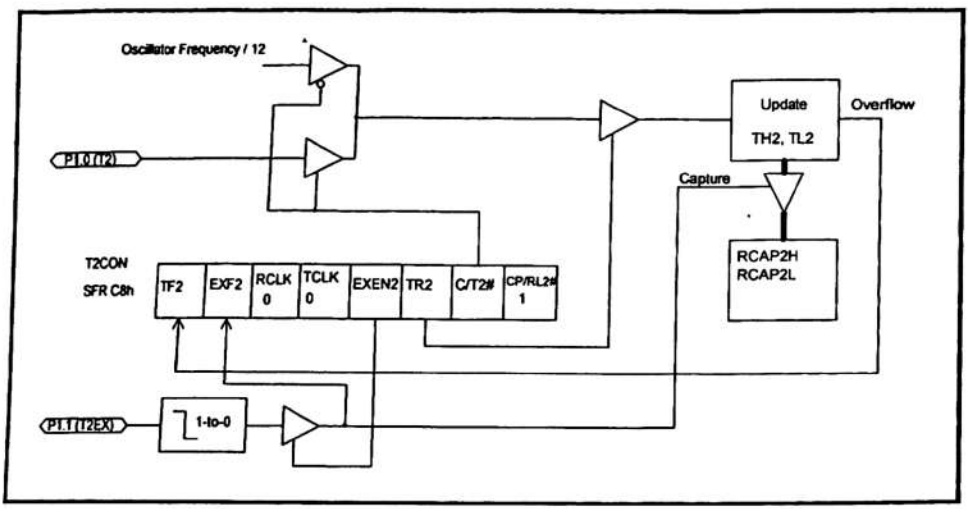


Figure 1.14. Timer 2 Capture Operation.

The capture mode is programmed by setting the control bit CP/RL2#. Note that the control bits RCLK and TCLK must be zero so that Timer 2 is not used for Baud rate generation.

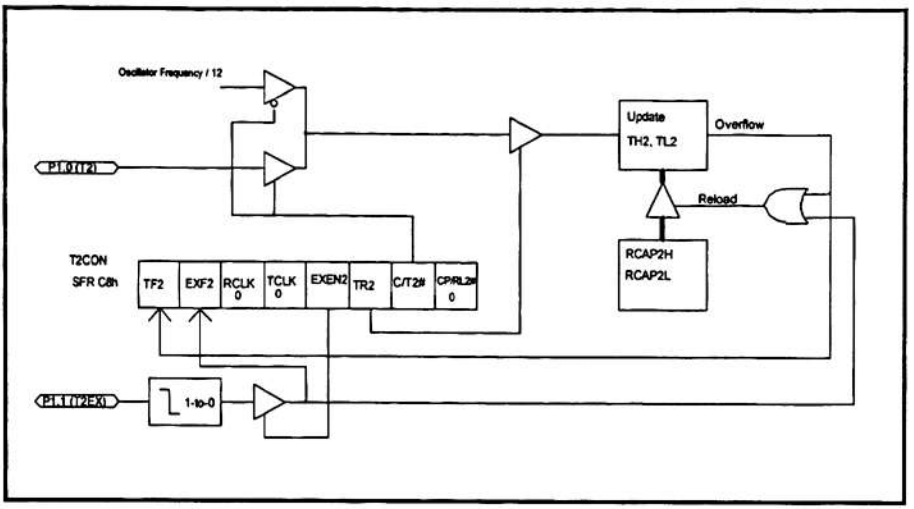


Figure 1.15. Timer 2 Reload Operation.

Timer 2 is controlled by the SFR T2CON.

Bit	7	6	5	4	3	2	1	0
Name	TF2	EXF2	RCLK	TCLK	EXEN2	TR2	C/T2#	CP/RL2#

Figure 1.16. The Fields of the T2CON Register.

CP/RL2# (bit 0):
T2CON2.0 is a control flag. When set, provided that EXEN2=1, the count in TL2 and TH2 are copied into RCAP2L and RCAP2H upon a 1-to-0 transition on pin T2EX. When CP / RL2# is cleared, the contents of RCAP2L and RCAP2H are copied into TL2 and TH2 upon timer overflow, or, provided that EXEN2=1, upon a 1-to-0 transition on pin T2EX. (When Timer 2 is used as a Baud rate generator, reload occurs after timer overflow and not upon a 1-to-0 transition on pin T2EX, even if EXEN2=1.)

C/T2# (bit 1): When T2CON.1 is set, Timer 2 operates as a counter, incremented by the falling edge of the signal at pin T2. When cleared, Timer 2 operates as a timer, incremented every 12 oscillator cycles.

TR2 (bit 2): Enables the timer/counter when set.

EXEN2 (bit 3): When EXEN2 is set, Timer 2 responds (either a reload or a capture operation) to 1-to-0 transitions on pin T2EX. This bit is ignored if Timer 2 is used as a Baud rate generator.

TCLK (bit 4): When TCLK is set, the serial port uses Timer 2 overflow pulses to shift the bits out. The serial port must be operating in mode 1 or 3. When TCLK is cleared, the serial port uses Timer 1 overflow pulses.

RCLK (bit 5): When RCLK is set, the serial port uses Timer 2 overflow pulses to time the sampling of the serial input port. The serial port must be operating in mode 1 or 3. When RCLK is cleared, the serial port uses Timer 1 overflow pulses.

EXF2 (bit 6): T2CON.6 is a status flag. When set, it indicates that a 1-to-0 transition has occurred on pin T2EX. If not masked, this will cause an interrupt. EXF2 must be cleared by software.

TF2 (bit 7): T2CON.7 is a status flag. It is set when Timer 2 overflows. If not masked, this will cause an interrupt (except when Timer 2 is used as a Baud rate generator). TF2 must be cleared by software.

Note that the status bits EXF2 and TF2 are OR-ed and used as an interrupt source. When such an interrupt occurs, it is not immediately known whether an overflow or an external signal, or both are responsible for the interrupt. Thus, it is expected that software will inspect EXF2 and TF2 to determine the cause of the interrupt. These status bits are then expected to be cleared by software.

The enhancements of the 8052 over the 8051 are implemented by establishing new SFRs. Specifically, the 8052 has five new SFRs, T2CON, TL2, TH2, RCAP2L, and RCAP2H. The new on-chip facility, Timer 2, is implemented by using these control and status/data registers. Note that since all 8051 SFRs are kept, the 8052 is backward compatible with the 8051. Establishing new SFRs to implement high-performance features is the way all of the enhanced members of the 8051 family are realized.

1.2.2.7. Power Control

The CMOS version of the 8052, called the 80C52, can be put into either an idle or power down mode by programming the power control register (PCON) bits. In the idle mode, the system clock is removed from the CPU, but not from the interrupt control logic, the serial port, or the timer/counters. The idle mode is terminated either when an interrupt is received, or upon a system reset through the RESET pin. The power down mode stops the system clock completely. The only way to reactivate the microcontroller is by a system reset. In both modes, the internal RAM and the SFRs are preserved. A system Reset, however, initializes SFRs. Thus, if any of the SFRs must be saved, they should be copied into internal or external RAM before initiating the power down mode.

Bit	7	6	5	4	3	2	1	0
Name	SMOD	-	-	-	GF1	GF0	PD	IDL

Figure 1.17. The Fields of the PCON Register.

IDL (bit 0): Idle Mode. Initiates the idle mode when set by software.
PD (bit 1): Power Down Mode. Initiates the power down mode when set by software.
GF0 (bit 2): General Purpose Flag 0. General-purpose flag set or cleared by software.
GF1 (bit 3): General Purpose Flag 1. General-purpose flag set or cleared by software.
SMOD (bit 7): Serial Port Baud Rate Control Bit. Doubles the Baud rate if set when Timer 1 is used to generate the Baud rates in modes 1, 2, or 3 (see Section 1.2.2.3.).

CHAPTER 2

ASSEMBLY LANGUAGE PROGRAMMING

This chapter reviews the assembly language of the 8051 family of microcontrollers. The examples in this chapter were assembled with the Reads51 *Integrated Development Environment* (IDE). The latest version of the IDE and the users' manual may be downloaded from www.rigelcorp.com. Once the example programs are assembled, there are three options available to the reader who wishes to experiment with microcontroller programming:

1. Use the chip simulator (RChipSim51), which is an integral module of the IDE.
2. Build an 8051-based microcontroller system as described in Chapter 7 and run the programs on the system,
3. Use a commercial evaluation system or an in-circuit emulator.

Using the RChipSim51 chip simulator is perhaps the most convenient way to start experimenting with the examples given in this section. RChipSim51 includes virtual ports and a virtual TTY window. You may not only single step through your code, but also run them without breakpoints and interact with the ports by clicking on the virtual ports. Chapter 7 contains information on how to build a minimal 8051-based microcontroller system. Also described is an 80C515-based minimal system. The programs discussed in this chapter may be downloaded and run on the minimal system. Since the minimal system does not have user-friendly firmware to allow single stepping, the registers and memory that need to be inspected must be displayed by calling the print subroutine. Refer to Chapter 7 for more details on using the minimal system to run the examples in this chapter. There a number of commercially available evaluation boards for the 8051 family. The experiments given in this textbook were tested on Rigel's R31JP, RMB-S, R535J, RIC320 and R515JC boards.

The examples focus on a single class of instructions at a time, for example, move instructions. Almost all applications software require a synergistic mix of instructions, such as using move instructions along with branching instructions and arithmetic or logical instructions. Concepts pertaining to the effective integration of various classes of instructions are discussed in the next chapter.

2.1. Background

Conceptually, a typical microcontroller system reads signals or data from its ports, executes appropriate data manipulation instructions depending on the (run-time) values of these signals, and finally emits the results (processed data) as output signals. The instruction set of a microcontroller may be partitioned into data transfer

instructions, data processing or manipulations instructions, and program flow control instructions.

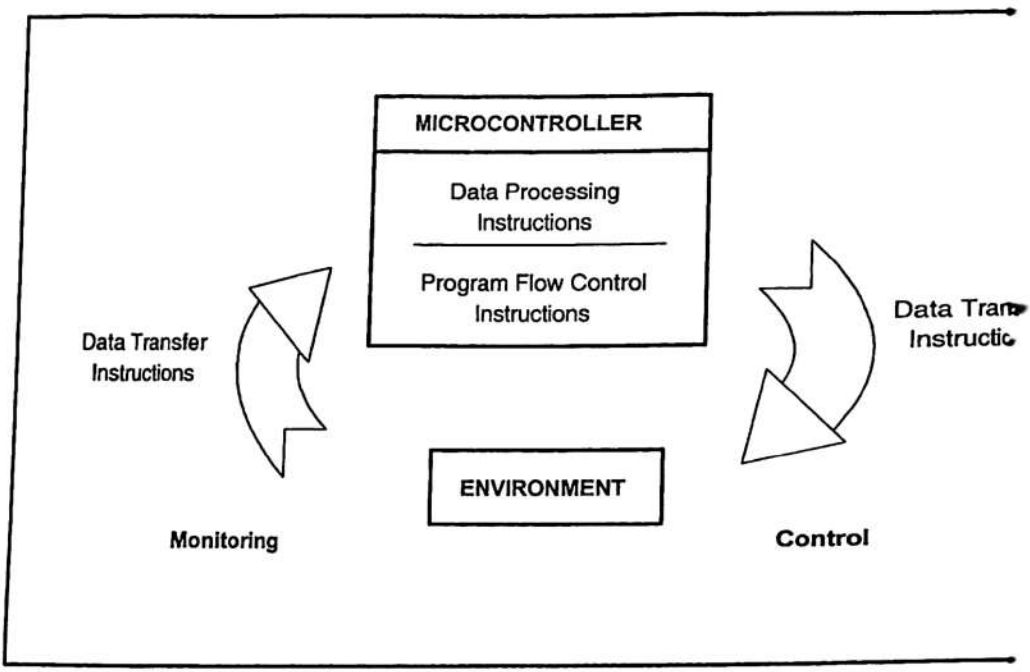


Figure 2.1. A Microcontroller and its Environment.

Physically, the operation of a microcontroller may be described by an instruction-fetch cycle. The processor places an address on the program memory address bus and performs a memory read operation. The data read is interpreted as an instruction. The address of the instruction is stored by the microcontroller in a register, often called the Program Counter (PC). The instruction may be, for example, an arithmetic operation involving two registers. Instructions may be longer than a single byte, thus the processor may need to read a few instruction bytes before actually executing the instruction. Once the instruction is executed, the PC is updated to the next instruction address and a new instruction-fetch cycle begins. The instruction set defines the set of operations the microcontroller is capable of executing. Each instruction is represented numerically by a value. For example, the instruction

```
inc a
```

is represented by the value 0x04 (04h). This is a simple, single-byte instruction. The value 0x04 is unique to the instruction INC A. That is to say, if 0x04 is the value of the

byte that is fetched during an instruction fetch cycle, the value of the accumulator is incremented. As part of the instruction fetch cycle, the PC is also incremented to point to the next instruction. The value 0x04 is referred to as the operation code or simply as opcode.

This instruction increments the accumulator. The increment operation may also be applied to other registers. Of course, the instructions that increment other registers have an opcode different from 0x04. In this case, the target register is viewed as an operand to the instruction. Information about the operands of an instruction is appended into the instruction. For example, the instruction

```
inc 35h
```

needs to increment the value of the internal RAM at address 0x35 (or 35h). The address 0x35 must also be packaged as part of the instruction. The machine code for this instruction takes two bytes: 0x05 followed by 0x35. During the instruction fetch cycle, the first byte (0x05) is read. The microcontroller interprets this as the command "increment the value of internal RAM at address nn where the byte nn is the next byte in program memory." The PC is incremented and the byte nn is read, in this case 0x35. The processor then proceeds to increment the value in internal RAM at address 0x35. Again, the PC is incremented so that it now points to the next instruction to be fetched. Note that the PC is incremented twice during this instruction. Again, the value 0x05 uniquely describes the instruction INC nn. The machine codes 0x04 and 0x05 are both increment opcodes but support different addressing modes. The concept of addressing modes is central to the architecture of any architecture and instruction set. Addressing modes are discussed below.

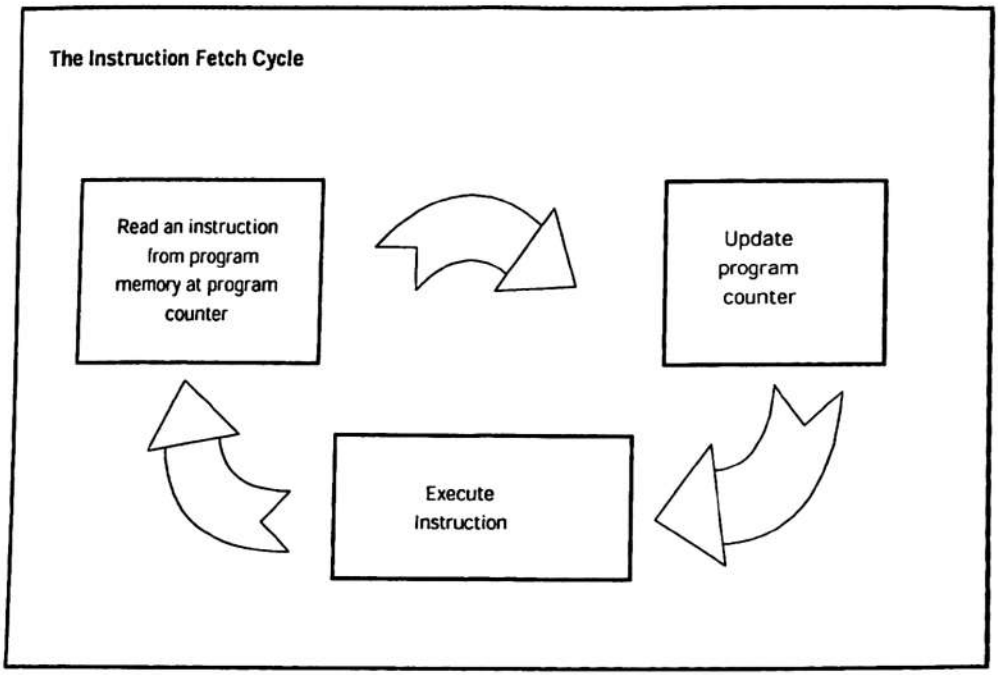


Figure 2.2. The Instruction Fetch Cycle.

The MCS-51 instruction set contains powerful data transfer instructions. These instructions transfer bytes or individual bits. Data sources and destinations may be internal data RAM or SFRs, as well as program or external data memory. Since parallel and serial ports are implemented as SFRs, data transferred from such SFRs constitute reading ports, or inputting signals from the environment. Similarly, writing to these SFRs allows the microcontroller to affect its environment by emitting signals.

An important concept in data transfer instructions is that of addressing modes. Addressing modes refer to the ways of specifying the source or destination bytes or bits of an operation. For example, in a data transfer operation, the source byte may be in a register, in internal RAM, in program memory, or in external data memory. Moreover, the address of the source byte may be given as a part of the instruction or may be placed in a register. The 8051 instructions utilize five different addressing modes: immediate, direct, register, register indirect, and indexed byte. Direct or flag (carry flag) addressing modes are used for bits. Operands used in the instructions imply the addressing mode(s) used.

The following list gives the addressing mode notation. Perhaps the best way to gain an understanding of the addressing modes is to refer to the extensive collection of MOV instruction explained in detail in Section 2.2.

Rn:	One of eight registers in the active register bank.
direct:	An 8-bit address of an internal RAM location, including bit-addressable memory and the SFRs.
@Ri:	Indicates the register-indirect addressing mode. Only R0 or R1 of the active register bank can be used in this mode. Ri holds the 8-bit address of an internal RAM location, including bit-addressable memory and the SFRs.
#data:	An 8-bit constant used in immediate addressing, i.e., data is embedded within the code.
#data16:	A 16-bit constant used in immediate addressing, i.e., data is embedded within the code.
addrs16:	A two-byte vector, pointing anywhere within the 64K of Program Memory.
addrs11:	An 11-bit vector, pointing anywhere within the current 2K block of Program Memory.
rel:	A two's compliment offset byte used in relative branching. The program may branch up to -128 locations preceding or 127 locations following the (updated) Program Counter.
bit:	An 8-bit address of bit addressable RAM or SFRs.

Data processing instructions are divide into arithmetic and logic instructions. The basic arithmetic instructions are *Addition*, *Subtraction*, *Multiplication*, and *Division*. *Increment* and *Decrement* operations as well as a decimal adjust instruction to process Binary-Coded Decimal (BCD) numbers are also considered as arithmetic operations. The basic logic operations are the byte-wise *AND*, *OR*, *EXCLUSIVE-OR*, and *NOT (Complement)* operations. *Clear*, *Set*, *Complement*, *Rotate*, and *Swap* nibbles are also considered to be logic operations. The MCS-51 also has the capability to manipulate single-bit data. These instructions are referred to as Boolean operations. Logical AND, OR, and Complement operations may be performed on individual bits. Arithmetic and logic operations affect the flags. The 8051 family of microcontrollers has carry, overflow, and zero flags. For example, if the addition of two bytes results in a value greater than 255, the external carry bit is copied into the carry flag. These flags are interrogated by conditional jump instructions.

The program flow control instructions, also called branching instructions, manipulate the program counter. There are two types of branching instructions: jumps and call-return instructions. Jump instructions modify the PC, changing the address of the next instruction to be fetched. Conditional jump instructions modify the PC only if certain flags are set or cleared. Thus, the program may be directed to different

routines depending on the results of data processing instructions. For example, a port may be read and compared to a reference constant. A routine may be executed if the port is the same as the reference, and bypassed if different. Call and return instructions also modify the PC. A call is similar to an unconditional jump instruction, except that the current value of the PC is pushed on the stack before the call. The complementary return instruction pops the PC previously pushed on the stack, thereby returning the PC to the instruction immediately following the previous call instruction.

This section illustrates the MCS-51 assembly language instructions by examples. The examples are intended to be run on RChipSim51 or on an evaluation board. The debugging features of these environments will help illustrate the various aspects of the examples.

The following examples assume that the reader is familiar with the common number systems, namely decimal, binary, and hexadecimal systems. Please refer to Appendix C for a short description of number systems. Most often, bytes are expressed as hexadecimal numbers. The Reads51 assembler accepts both the traditional assembly language and the C language syntax. For example, 254 decimal may be expressed either as 0FEh or as 0xFE.

The Reads51 IDE V4 toolchain uses the MCS-51 assembly language syntax. Both absolute assembly and relative assembly formats are supported. The simple examples given in this chapter use the absolute assembler. Each absolute assembly segment starts with the directive "cseg at XXXX," where XXXX is the origin of the program. Most examples set the code segment to start at 0x8000. This is the default start address of RAM on the minimal system discussed in Chapter 7 as well as on most Rigel boards. If you use the chip simulator, you should specify this code start address in the options dialogs. Refer to the latest Reads51 help files and tutorials for more information about debugging with the chip simulator. Each absolute segment ends with the keyword "end."

The 8051 instructions are grouped into data transfer instructions, arithmetic instruction, logic instructions, and branching instructions. Each line of assembly code contains an operation code (opcode) or a pseudo operation (pseudo-op). An opcode is the unique assembly language instruction written as a mnemonic, such as MOV or INC. A pseudo-op is similar to an opcode; however, it does not correspond to any machine instructions. Instead, it is an instruction to the assembler to take some action. Pseudo-ops are similar to compiler directives. For example

```
org 0x8000
```

appearing in an assembly language program does not generate any machine instructions for the processor to fetch and execute. Rather it is a command to the

assembler to place the instructions that follow starting at program memory address 8000h. The mnemonic ORG is an abbreviation for the word "origin."

Opcodes and pseudo-ops usually have one or more operands. Optional are labels and comments. A typical line of assembly code is given below.

```
start: mov b, #0x35    ; initialize the b register
```

Here, `start` is a label. The assembler assigns numerical values to labels. Specifically, the value of the label is its address in code memory. This address is the address of the first byte of the instruction immediately following the label. Here, the value of the label is the address of the move instruction. The opcode MOV has two operands: the destination B and the source #0x35, as will be discussed later. The semicolon indicates the beginning of the comment. The semicolon and any other character right of it are ignored by the assembler.

2.2. Data Transfer Instructions

The MCS-51 contains a rich set of data transfer instructions. The byte- and bit-oriented MOV instructions are grouped according to the addressing modes in the table below. The addressing modes describe how the operand "byte" is selected. The register-specific addressing mode implies that operand "byte" is the accumulator.

Table 2.1. Summary of Data Transfer Instructions

Mnemonic	Description	Addressing Modes				
		Direct	Register Indirect	Register	Immediate Constant	Register-Specific
`MOV A, byte`	Move byte to Accumulator	✓	✓	✓	✓	
`MOV byte, A`	Move Accumulator to byte	✓	✓	✓	✓	
`MOV Rn, byte`	Move byte to a Register of the Current Bank	✓			✓	✓
`MOV direct, byte`	Move byte to an Internal Data Register	✓	✓	✓	✓	✓
`MOV @Ri, byte`	Move byte to Internal Register whose Address is in Ri	✓			✓	✓
`MOV DPTR, data16`	Move into Data Pointer				✓	
`PUSH byte`	Increment stack pointer, move byte to stack.	✓				
`POP byte`	Move from stack to byte, decrement stack pointer.	✓				
`XCH A, byte`	Exchange accumulator and byte.	✓	✓	✓		
`XCHD A, byte`	Exchange low nibbles of accumulator and byte.		✓			

In general, the source and the destination specification may use different addressing modes. For example,

```
mov    r3, 0x32
```

copies the contents of internal register 0x32 into register 3 of the currently selected register bank. In this example, the source is specified by direct addressing (of the

internal register) and by register addressing of the destination. Note that if register bank 0 is currently selected, the above instruction is effectively the same as

```
mov    3, 0x32
```

where both the source and the destination are specified in the direct addressing mode.

2.2.1. The Immediate Addressing Mode

Immediate addressing, or perhaps more explicitly, immediate constant addressing, refers to the source being a constant embedded in the code. For example, the instruction

```
mov    a, #1
```

is a two-byte instruction: 0x74 0x01 (74h 01h). The first byte 0x74 represents the operation code, or opcode, for the instruction to move the following data byte, in this case the 0x01, into the accumulator. Data is directly placed in the code. That is, the data byte (1) immediately follows the opcode in code memory. This addressing mode is often used in the initialization segment of the programs.

Assemble the following short routine.

```
cseg   at 0x8000     ; set the origin
mov    a, #0         ; put 0 into the accumulator
mov    a, #0x11      ; put 0x11 into the accumulator
mov    a, #27        ; put 27 (0x1B) into the accumulator
ljmp   0             ; return to the monitor
end
```

The program starts at 8000h of external RAM. Place a break point at 8000h. Run the program and inspect the value of the accumulator. The initial value of the accumulator depends on what previous instructions were executed. It may be assumed to be arbitrary. Single-step through the program and verify that the accumulator contains 0, 11h, and 1Bh. Note that the last instruction of the program is a long jump to address 0. The processor starts executing from address 0 upon a reset. The program assumes that a monitor routine, such as the one explained in Section 7.3., is running and that it may be reached by emulating a reset. This last instruction is not needed if you are running the program with the chip simulator.

The following program uses the immediate constant addressing mode instruction to load constants into the 8051 registers. The program puts zero into the PSW. Recall that bits 4 and 3 of the PSW determine the active register bank. Register bank 0 is selected first. The second half of the program selects register bank 1 and repeats

putting constant values into the registers. Note that the register-addressing mode is used to specify the destination.

Assemble the following program. Single step through the program and verify the register contents. One can see the internal RAM locations 0 through Fh get written with 0, 1, 2, 3, 4, 5, 6, 7, 0, 1, 2, 3 4, 5, 6, and 7. It is also observed that by selecting a different register bank, the values of the registers of the previously selected bank are not overwritten. Writing 7, 6, 5, 4, 3, 2, 1, 0 into registers R0 through R7 of bank 1 has no effect on the registers R0 to R7 of bank 0. The registers of bank 0 still contain the values 0, 1, 2, 3, 4, 5, 6, and 7.

```
cseg at 0x8000      ; set the origin

mov   psw, #0       ; select register bank 0
mov   r0, #0        ; put 0 into register 0
mov   r1, #1        ; put 1 into register 1
mov   r2, #2        ; put 2 into register 2
mov   r3, #3        ; put 3 into register 3
mov   r4, #4        ; put 4 into register 4
mov   r5, #5        ; put 5 into register 5
mov   r6, #6        ; put 6 into register 6
mov   r7, #7        ; put 7 into register 7

mov   psw, #8       ; select register bank 1
mov   r0, #7        ; put 0 into register 7
mov   r1, #6        ; put 1 into register 6
mov   r2, #5        ; put 2 into register 5
mov   r3, #4        ; put 3 into register 4
mov   r4, #3        ; put 4 into register 3
mov   r5, #2        ; put 5 into register 2
mov   r6, #1        ; put 6 into register 1
mov   r7, #0        ; put 7 into register 0

ljmp  0             ; return to the monitor
end
```

The following program uses immediate addressing to put values into the internal registers 0x70 through 0x73. Assemble the following program. Single step through the program and verify the register contents.

```
cseg  at  0x8000    ; set the origin
mov   0x70, #0      ; put 0 into internal register 70h
mov   0x71, #1      ; put 1 into internal register 71h
mov   0x72, #2      ; put 2 into internal register 72h
mov   0x73, #3      ; put 3 into internal register 73h
ljmp  0             ; return to the monitor
end
```

There is a data transfer instruction that moves a 16-bit value to a pair of SFRs. Specifically, the instruction

```
mov    dptr, #CONSTANT
```

moves the 16-bit constant into the data pointer. For example,

```
mov    dptr, #0x1234
```

places 0x12 into DPH and 0x34 into DPL.

2.2.2. The Direct Addressing Mode

The direct addressing mode refers to specifying an internal data register or a SFR by its address. Addresses in the range [0x00-0x7F] refer to internal data RAM, and those in the range [0x80-0xFF] refer to the SFRs (see Table 1.1). For example, 0x90 is the address of Port 1. Writing to the SFR 0x90 effectively sends out a signal from Port 1. Similarly, reading the SFR 0x90 effectively inputs the signals on Port 1 pins.

```
cseg   at  0x8000    ; set the origin
mov    a, 0x70       ; copy contents of internal register
                     ; 70h to a
clr    a             ; clear the accumulator
mov    0x90, a       ; copy the accumulator contents to SFR
                     ; 90h (Port 1)
ljmp   0             ; return to the monitor
end
```

Type the above program and verify that the contents of register 70h are copied into the accumulator by single stepping through the program. If you are using an evaluation board, connect Port 1 bits to eight LEDs (see Chapter 7). Placing a 0 on the port bit will turn on the connected LED. Try outputting different values to SFR 90h (Port 1) by modifying the instruction

```
clr    a
```

to

```
mov    a, #0xNN
```

in the above program.

Referring to the SFRs by their value reduces the readability of the programs. The preferred way to handle references to the SFRs is by defining symbols using pseudo operations. The EQU pseudo-op is a general assembly language construct. The MCS-51 language introduces the keyword "data" to define internal registers and SFRs. Consider the same program written with the help of EQUs and "data" definitions.

```
InByte equ  0x70    ; source byte
Port1  data 0x90    ; Port 1 SFR

cseg   at 0x8000    ; set the origin
mov    a, InByte    ; copy contents of inByte to a
mov    a, #0        ; clear the accumulator
mov    Port1, a     ; copy the accumulator contents Port 1
ljmp   0            ; return to the monitor
end
```

This version is much easier to read since the operands Port1 and InByte are more descriptive than 70h and 90h (or 0x70 and 0x90). Note that P1 is a predefined symbol of the MCS-51 assembler. The Reads51 assembler defines all the 8051 SFRs in the include file "sfr51.inc". The header file is placed in the ".\include" directory. You may simply include this file in your source code and then refer to the 8051 SFRs by name.

```
#include <sfr51.inc>
   InByte equ  0x70    ; source byte

   cseg   at 0x8000    ; set the origin
   mov    a, InByte    ; copy contents of inByte to a
   mov    a, #0        ; clear the accumulator
   mov    P1, a        ; copy the accumulator contents Port 1
   ljmp   0            ; return to the monitor
   end
```

Note that the include directive is similar to the C syntax. The file name is placed between pointed brackets rather than double quotation marks. The pointed brackets imply that the file is a "system" file to be found in the ".\include" directory. Otherwise, the file is sought in the current directory (or system path). However, if you wish to refer to the port as Port1 rather than P1, the "data" definition or EQU pseudo-op is necessary. When the assembler comes across a symbol, such as `InByte`, it refers to the list of definitions to find its value. This is similar to the "search and replace" operation of a word processor where the value is substituted for the symbol. The "data" directive is better than the EQU pseudo op, because it also contains type information.

Next, connect Port 3 bits P3.2 to P3.5 to four push buttons. Type, assemble, and download the following program.

```
Port3  data 0xB0    ; port 4
Port1  data 0x90    ; port 1

cseg   at 0x8000    ; set the origin
```

```
mov    a, Port3      ; copy the contents of Port 3 to acc
mov    Port1, a      ; copy the accumulator contents Port 1
ljmp   0             ; return to the monitor
end
```

Single step through the program and verify that the status of the push buttons is first read into the accumulator, and then transferred to Port 1. Note that pressing a push button clears the corresponding bit. Try experimenting while pressing different button combinations. Also, try the following program.

```
#include <sfr51.inc>

    cseg   at 0x8000     ; set the origin
    mov    P1, P3        ; copy P3 to P1
    ljmp   0x8000        ; repeat
    end
```

This program reads the push buttons connected to Port 3 bits and copies their states to the LEDs connected to Port 1 bits. Press the RESET button to abort the program. Note that although you may copy Port 3 to Port 1, it is not a good idea to copy P1 to P3 when the 8051 uses external data or the serial port. Two of the P3 bits are used for external memory access control, and two others for the serial communications. Writing to the control signals would most probably disrupt external memory access.

2.2.3. The Register Addressing Mode

The register-addressing mode refers to either the source or the destination being one of the eight registers of the currently selected register bank. The following code selects register bank 2 and places the value of the accumulator and the b register into registers 0 and 7. Register addressing is used to specify the destinations of the move instructions.

```
#include <sfr51.inc>

    cseg   at 0x8000     ; set the origin
    mov    psw, #0x10    ; select register bank 2
    mov    r0, a         ; copy the contents of the accumulator
                         ; into register 0
    mov    r7, b         ; copy the contents of the B register
                         ; into register 7
    ljmp   0             ; return to the monitor
    end
```

PSW is defined in the header file "sfr51.inc." Assemble the program. Signal step through the program and verify that the contents of the accumulator and the B register are copied to registers 0 and 7 of the register bank 2. These registers are the same

as internal registers 0x10 (16) and 0x17 (23). Register addressing may be used to specify the destination as well. Review the following program for an example.

```
#include <sfr51.inc>

cseg   at 0x8000      ; set the origin
mov    psw, #0x18     ; select register bank 3
mov    a, 0x18        ; copy register 0 contents to the
                      ; accumulator
mov    0x1F, a        ; copy the contents of the accumulator
                      ; into register 7
ljmp   0              ; return to the monitor
end
```

Since register bank 3 is selected, r0 and r7 are physically the same as internal data memory 0x18 and 0x1F, respectively.

2.2.4. The Register-Specific Addressing Mode

Some instructions are specific to the registers used. No further addressing is required. Recalling that the SFR 0xE0 is the accumulator, consider the following two move instructions.

```
cseg   at 0x8000      ; set the origin
mov    a, #1          ; move the constant 1 into the accumulator
mov    0xE0, #1       ; move the constant 1 into SFR E0h
ljmp   0              ; return to the monitor
end
```

Type and assemble the program. Then view the list file (with the .LST extension). Observe that the first move instruction is a two-byte instruction, consisting of the bytes 0x74 and 0x01. Note that the first byte 0x74 is interpreted as "take the following byte and place in the accumulator." That is the use of the accumulator is implicitly known.

The second instruction moves the constant 1 into the SFR 0xE0. This SFR is the accumulator. Therefore, effectively the two move instructions accomplish the same task. This instruction consists of three bytes: 75 E0 01, in hexadecimal. The first byte 75 is interpreted as "of the following two bytes, the first is the address of a register, into which, put the second byte." That is, the instruction 75 does not implicitly specify a target; rather, it indicates that the address of the target is to be read from the next byte in code.

The abbreviation 'ACC' is defined as a symbol of the constant 0xE0, the address of the accumulator. Provided that you include "sfr51.inc", of the two instructions

```
mov    a, #1          ; constant 1 to the accumulator
mov    acc, #1        ; constant 1 to SFR ACC=E0h
```

the first is register specific, while the second uses direct addressing to specify the destination. That is "a" is assembled as a direct reference whereas "acc" is a symbol to be replaced by its value of 0xE0. These two instructions generate different machine code. The first instruction is assembled as two bytes: 0x74, 0x01. The second instruction is assembled as three bytes: 0x75, 0xE0, and 0x01. The second instruction not only requires another byte of code memory to specify the address 0xE0, it also takes longer; two machine cycles instead of the one cycle required by the first instruction. The 8051 microcontroller is optimized in favor of the key registers such as the accumulator, the b register, and the registers of the current bank. The number of bytes and the execution times of each instruction are given in the manufacturers' data books. This information becomes essential when efficient tight code is needed, such as for interrupt service routines that need fast attention.

2.2.5. The Register Indirect Addressing Mode

This is one of the most powerful addressing modes. The address of the source or destination is not given explicitly. Instead, the content of a register is used as the target address. Two registers, r0 and r1, of the currently selected register bank are used to specify the addresses. Consider the following program.

```
#include <sfr51.inc>

cseg   at 0x8000     ; set the origin
mov    psw, #0       ; select register bank 0
mov    r0, #0x78     ; move 78h into register 0
mov    @r0, #1       ; set the register whose address is
                     ; specified in the r0 register to the
                     ; constant 1.
ljmp   0             ; return to the monitor
end
```

Assemble the program. Single step through the program. Observe that r0 is set to 0x78. Then verify that register 0x78 is set to 1.

In all of the addressing modes, one may deduce the source or destination of the data transfer by inspecting the code. In this mode, the source or destination address is simply the contents of yet another register. Since registers may be considered to hold data, the source or destination address in register indirect addressing may be viewed as data. The exact source or destination is not necessarily known when the program is assembled. This allows the data flow to be manipulated in run time. Such indirect addressing allows powerful programs to be constructed with relatively fewer lines of code. This power does not come, however without its responsibilities. The programmer must be more careful not only in writing the initial program but also during debugging. Debugging a program in which no indirect addressing is relatively straightforward. It requires checking the operation of the program with different data

values in the source bytes and verifying that the correct destination byte values are generated. When indirect addressing is used, however, care must be taken to ensure that also the source and destination byte addresses are valid. For example, if an incorrect destination byte address is placed into R0 and an indirect addressing mode transfer is used, the program may overwrite important information in internal RAM. If this happens to be part of the stack that holds a return address, it would most probably crash the program. Note however, that this occurs only when a bad destination address is generated. That is to say, the program may occasionally crash. Debugging intermittent problems such as these is perhaps among the most difficult tasks in code development. An In-Circuit Emulator (ICE) is an indispensable debugging tool in such cases allowing the state of the processor to be tracked and recorded so that the cause of the crash can later be identified.

Consider the following twist to the program.

```
#include <sfr51.inc>

   cseg   at 0x8000      ; set the origin
   mov    psw, #0        ; select register bank 0
   mov    r0, P1         ; copy Port 1 into register 0
   mov    @r0, #1        ; set the register whose address is
                         ; specified in the R0 register to the
                         ; constant 1.
   ljmp   0              ; return to the monitor
   end
```

Data is placed into the register whose address is the value read from Port 1. This address cannot be deduced by inspecting the assembled program. Once the program is executed, depending on the signals present on Port 1, the data is placed into one of the internal registers. Of course, such a program may potentially produce disastrous results if Port 1 specifies a sensitive register, such as the program status word, or Port 3, while fetching instructions from external program memory. (In the latter case, Port 3 bits are used as the control signal read and write. Blocking the read signal, for example, will prevent fetching the next instruction.)

The register indirect addressing mode is useful in constructing data transfer loops. Refer to Section 3.2 for an example.

There also are register indirect addressing mode instructions that transfer data between the accumulator and external data memory. Consider the following examples.

```
#include <sfr51.inc>

    cseg   at 0x8000      ; set the origin
    mov    a, #1          ; put the constant 1 into the
                          ; accumulator

    mov    dptr, #0x9000; set the data pointer to 9000h
    movx   @dptr, a       ; copy the accumulator into
                          ; external data at the address
                          ; specified by the data pointer.
                          ; In this case, the 1 is copied
                          ; into external memory location
                          ; 9000h

    mov    dptr, #9001h ; set the data pointer to 9001h
    movx   a, @dptr       ; copy the contents of external
                          ; data memory at location 9001h
                          ; into the accumulator.

    mov    a, #1          ; put the constant 1 into the
                          ; accumulator
    mov    r0, #0         ; put 0 into register 0 of the
                          ; current bank
    mov    P2, #90h       ; put 90h on Port 2 (SFR A0h)
    movx   @r0, a         ; write the accumulator contents
                          ; (1) into external data memory
                          ; address 9000h

    mov    r0, #1         ; put 1 into register 0 of the
                          ; current bank
    movx   a, @r0         ; read external data at 9001h
                          ; into the accumulator

    ljmp   0              ; return to the monitor.
    end
```

Assemble the program. Before running the program, set external data memory locations 0x9000 and 0x9001 to 0xFF. Single step through the program. While single stepping, you may dump external data memory and inspect a block of 256 bytes. Observe that the constant 1 is put into the accumulator, that the data pointer is modified to 0x9000, and that the accumulator is copied into external data memory at location 0x9000. Similarly, observe that the value 0xFF in external data memory at 0x9001 is moved into the accumulator.

In the second portion of the program, register r0 holds the lower address byte. The higher address byte, 0x90, is explicitly moved to Port 2, which emits the address bits A8 to A15 when fetching external memory.

2.2.6. The Register Indexed Addressing Mode

A useful application of this powerful addressing mode is implementing look-up tables. In this mode, the source or destination address is obtained by adding the value held in the accumulator to the base address. The base address may be either the data pointer DPTR, or the program counter PC.

The following program evaluates the pattern of a seven-segment display given a single hexadecimal digit. It is assumed that the accumulator holds a value between 0 and 0x0F (15). The pattern is intended to be output to a port that is connected to a seven-segment display. The least-significant bit is connected to segment a, and bit 6 is connected to segment g.

```
#include <sfr51.inc>

   cseg  at 0x8000     ; set the origin

; put the table base address into the data pointer
   mov   dptr, #0x8100

   mov   a, #0         ; get the pattern for 0
   movc  a, @a+dptr    ; a contains the pattern for 0
   .
   .                   ; ...use the pattern...
   .
   ljmp  0             ; return to the monitor

   org   0x8100        ; the table starts at 8100h
   db    0xC0          ; 0
   db    0xF9          ; 1
   db    0xA4          ; 2
   db    0xB0          ; 3
   db    0x99          ; 4
   db    0x92          ; 5
   db    0x82          ; 6
   db    0xF8          ; 7
   db    0x80          ; 8
   db    0x90          ; 9
   db    0x88          ; A
   db    0x83          ; B
   db    0xC6          ; C
   db    0xA1          ; D
   db    0x86          ; E
   db    0x8E          ; F
   end
```

The "org" (origin) pseudo-op sets the assembler's location counter. The "db" (define byte) pseudo-ops simply place constants into code data. Refer to the MCS-51 assembly language syntax in the Reads51 help files.

Assemble the program. Single step through the program and verify that the instruction

```
movc   a, @a+dptr
```

moves the pattern 0xC0 into the accumulator. 0xC0 has all segments except segment g lit. A segment is lit when the corresponding bit is 0. Bit 7 is not used by the seven-segment display and is set to 1 for all patterns. Run the program again. This time while single stepping, modify the accumulator before the register indexed addressing mode instruction. Verify that all patterns are correctly read by the `mov` instruction.

Register indirect addressing mode instruction may also be used to transfer data from external program memory to the accumulator. The two available instructions are

```
movc       a, @a+dptr
```

and

```
movc       a, @a+pc.
```

An example of implementing a look-up table to be read by a subroutine is given in Section 3.3 after branching instructions are discussed.

2.2.7. Stack-Oriented Data Transfer

Another form of register indirect addressing is implemented with the 'push' and 'pop' instructions. These instructions use the SFR Stack Pointer (SP). The content of the SP is the address of the destination register in push operations, and is the address of the source register in pop operations. The SP is incremented before the data transfer in push instructions and decremented after pop instructions.

Typically, the SP is set to a value in the interval 0x2F to, say 0x6F during program initialization. A push instruction increments the SP, thus with an initial SP value of 0x2F, the first byte to be pushed is transferred to register 0x30. The initial SP value of 0x2F is often desirable since it is above the register banks and the block of bit addressable registers. An initial SP value of 0x6F leaves only 16 bytes to be used as stack. Although these values are plausible, it is the needs of the application program that determines the exact size and initial value (bottom) of the stack.

Push and pop instructions use the direct addressing mode. In order to push the contents of the accumulator onto the stack, the instruction

```
push   acc
```

must be used. Note that the operand acc is the symbol defined to be 0xE0. The instruction

```
push   a
```

is not recognized, since it uses the register-specific addressing mode.

The stack is a convenient means to temporarily store and retrieve intermediate results. The advantage of using the stack is that the specific address where the byte is stored need not be explicitly known. The stack is a first-in-last-out buffer. Bytes are stored and retrieved in that sequence. This property may be a disadvantage if data bytes need to be retrieve in an order different from that in which they were pushed.

Consider the following program. The stack pointer is initialized to 0x4F. Thus, the first byte pushed is placed in register 0x50.

```
#include <sfr51.inc>

   cseg   at 0x8000      ; set the origin

   mov    sp, #0x4F      ; initialize the stack pointer
   mov    a, #0x45       ; put 45h in the accumulator
   push   acc            ; push the accumulator
   mov    b, #0          ; clear the b register
   pop    b              ; pop top of stack into the B register

   ljmp   0              ; return to the monitor
   end
```

The program first initializes the stack pointer to 0x4F. The constant 0x45 is placed in the accumulator. The following push instruction increments the SP to 0x50 and puts the constant 0x45 into register 0x50. Next, the B register is cleared. The following pop instruction retrieves the byte (0x45) on the top of stack and copies it into the b register. Assemble the program. Single step through the program while observing the values of the accumulator, the B register, SP and register 0x50.

2.2.8. Exchange Instructions

Exchange instructions perform powerful two-way data transfers. The contents of the accumulator and a register, for example, may be exchanged without the need for a temporary storage byte. The 8051 family of microcontrollers offers two exchange operations: the byte-wise XCH and the nibble-wise XCHD (exchange digit). The first operand in the XCH instruction is the accumulator. The second operand may be an internal data register (direct addressing mode), a register of the currently selected register bank (register addressing mode), or the internal data register whose address is in r0 or r1 (register indirect addressing mode). The XCHD instruction exchanges

the lower nibbles of the accumulator with an internal data register whose address is in R0 or R1 (register indirect addressing mode). The XCHD instruction is helpful in working with Binary-Coded Decimal (BCD) numbers. Refer to Appendix C for more information about BCD numbers.

The following program simply places some values into the accumulator and the b register. Register r0 is initialized to zero and r1 to 0x7F. Moreover, internal RAM location 0x7F is initialized to one. The program simply employs exchange instructions to swap data among the registers and internal RAM. Assemble the program. Single step through the program and verify the operation of the exchange instructions.

```
#include <sfr51.inc>

    cseg   at 0x8000     ; set the origin

    mov    a, #0xAA      ; initialize the accumulator
    mov    b, #0xBB      ; initialize the B register
    mov    psw, #0       ; select register bank 0
    mov    r0, #0        ; initialize register 0
    mov    r1, #0x7F     ; initialize r1
    mov    0x7F, #1      ; initialize register 0x7F

    xch    a, b          ; exchange the contents of a and b
    xch    a, b          ; restore a and b

    xch    a, r0         ; exchange the contents of a and r0
    xch    a, r0         ; restore a and r0

    xch    a, @r1        ; exchange the contents of a and
                         ; register 0x7F
    xch    a, @r1        ; restore a and register 0x7F

    xchd   a, @r1        ; exchange low nibbles of a and
                         ; register 0x7F

    ljmp   0             ; return to the monitor
    end
```

2.2.9. Bit-Oriented Data Transfer

An embedded controller often manipulates single-bit data signals, such as the status of a push button, or a motor driver that turns the motor on and off. The 8051 family of microcontrollers offers extensive bit-oriented instructions for these applications.

Most importantly, perhaps, is the feature that all bits of internal registers in the range 0x20 to 0x2F and all the bits of SFR whose addresses are multiples of eight are individually addressable. For SFRs, any internal register address with a low nibble of

0 or 8 has addressable bits. For example, Port 1 is mapped to the SFR 0x90. Since the low nibble of 0x90 is 0, bits of Port 1 are addressable. The address of bit 0 0x90 is also 0x90. It is helpful to refer to the latter 0x90 as a bit address, as opposed to the former, a register address. Similarly, bits 1, 2, ..., 7 of Port 1 have bit addresses 0x91, 0x92, ..., 0x97. With this convention, the bit whose address is 0x98 is bit 0 of the register 0x98.

The internal registers in the interval 0x20 to 0x2F are also bit addressable. Bit j of internal register i has the address

$$8\times(i-0x20) + j, \quad \text{for } i=0x20, 0x21, ..., 0x2F, \text{ and } j=0, 1, ...,7.$$

Equivalently, the bit whose address is n is the j-th bit of register i, where

$$i = \text{int}\left(\frac{n}{8}\right) + 0x20,$$

$$j = n - 8\times(i-0x20), \qquad \text{for } n=0, 1, ..., 0x7F.$$

Bit 0 of register 0x20 has address 0, and bit 7 of register 0x2F has address 0x7F. Notice that register 0x20 immediately follows r7 of register bank 3.

Table 2.2. Bit Addresses of Internal Registers 20h to 2Fh and the SFRs.

	Register	bit 7	bit 6	bit 5	Bit 4	bit 3	bit 2	bit 1	bit 0
SFRs	0xF8	0xFF	0xFE	0xFD	0xFC	0xFB	0xFA	0xF9	0xF8
	0xF0	0xF7	0xF6	0xF5	0xF4	0xF3	0xF2	0xF1	0xF0
	0xE8	0xEF	0xEE	0xED	0xEC	0xEB	0xEA	0xE9	0xE8
	0xE0	0xE7	0xE6	0xE5	0xE4	0xE3	0xE2	0xE1	0xE0
	0xD8	0xDF	0xDE	0xDD	0xDC	0xDB	0xDA	0xD9	0xD8
	0xD0	0xD7	0xD6	0xD5	0xD4	0xD3	0xD2	0xD1	0xD0
	0xC8	0xCF	0xCE	0xCD	0xCC	0xCB	0xCA	0xC9	0xC8
	0xC0	0xC7	0xC6	0xC5	0xC4	0xC3	0xC2	0xC1	0xC0
	0xB8	0xBF	0xBE	0xBD	0xBC	0xBB	0xBA	0xB9	0xB8
	0xB0	0xB7	0xB6	0xB5	0xB4	0xB3	0xB2	0xB1	0xB0
	0xA8	0xAF	0XAE	0xAD	0xAC	0xAB	0xAA	0xA9	0xA8
	0xA0	0Xa7	0XA6	0xA5	0xA4	0xA3	0xA2	0xA1	0xA0
	0x98	0x9F	0x9E	0x9D	0x9C	0x9B	0x9A	0x99	0x98
	0x90	0x97	0x96	0x95	0x94	0x93	0x92	0x91	0x90
	0x88	0x8F	0x8E	0x8D	0x8C	0x8B	0x8A	0x89	0x88
	0x80	0x87	0x86	0x85	0x84	0x83	0x82	0x81	0x80
Bit Registers	0x2F	0x7F	0x7E	0x7D	0x7C	0x7B	0x7A	0x79	0x78
	0x2E	0x77	0x76	0x75	0x74	0x73	0x72	0x71	0x70
	0x2D	0x6F	0x6E	0x6D	0x6C	0x6B	0x6A	0x69	0x68
	0x2C	0x67	0x66	0x65	0x64	0x63	0x62	0x61	0x60
	0x2B	0x5F	0x5E	0x5D	0x5C	0x5B	0x5A	0x59	0x58
	0x2A	0x57	0x56	0x55	0x54	0x53	0x52	0x51	0x50
	0x29	0x4F	0x4E	0x4D	0x4C	0x4B	0x4A	0x49	0x48
	0x28	0x47	0x46	0x45	0x44	0x43	0x42	0x41	0x40
	0x27	0x3F	0x3E	0x3D	0x3C	0x3B	0x3A	0x39	0x38
	0x26	0x37	0x36	0x35	0x34	0x33	0x32	0x31	0x30
	0x25	0x2F	0x2E	0x2D	0x2C	0x2B	0x2A	0x29	0x28
	0x24	0x27	0x26	0x25	0x24	0x23	0x22	0x21	0x20
	0x23	0x1F	0x1E	0x1D	0x1C	0x1B	0x1A	0x19	0x18
	0x22	0x17	0x16	0x15	0x14	0x13	0x12	0x11	0x10
	0x21	0x0F	0x0E	0x0D	0x0C	0x0B	0x0A	0x09	0x08
	0x20	0x07	0x06	0x05	0x04	0x03	0x02	0x01	0x00

Single bits may be moved between any addressable microcontroller bit and the carry flag. The carry flag (C) is used by the 8051 family of microcontrollers as a single-bit accumulator of bit-oriented operations. The carry flag is bit 7 of the PSW. All bit-wise logical operations involve C. Consider the following example. Connect bit 0 of Port 1 to a push button and bit 1 of Port 1 to an LED.

```
#include <sfr51.inc>

   cseg   at 0x8000      ; set the origin

   mov    C, P1.0        ; move the button state into the carry
                         ; flag
   mov    P1.1, C        ; move the carry flag to the LED

   ljmp   0x8000         ; repeat
   end
```

The program reads the state of the push button connected to bit 0 of Port 1 and stores it in the carry flag. The carry flag is then transferred to the LED connected to bit 1 of Port 1. With this arrangement, the LED lights when the push button is pressed.

Strictly speaking, a pushbutton will bounce while being pressed. Bouncing refers to the phenomenon that the circuit is closed and opened several times as the button is being pressed due to the minute irregularities of the switch mechanics. Although bouncing usually happens at a high rate, the microcontroller is fast enough to observe the state transitions on pin P1.0. This means that the LED will flicker a few times before stabilizing in an on or off state. Since the human eye is not fast enough to observe the effects of bouncing, the above program ignores this effect.

Note that, P1.0 and P1.1 are symbols representing the constants 90h and 91h, defined by equ pseudo-ops. The above instructions may equivalently be written as

```
mov    C, 0x90        ; move the button state into the
                      ; carry flag
mov    0x91, C        ; move the carry flag to the LED
```

however, the former instructions are clearly more readable.

2.3. Data Processing Instructions

Data processing instructions are usually divided into two categories: arithmetic and logical instructions. Data processing instructions may also be classified according to the number of operands: unary or single-operand instructions and binary or two-operand instructions. These instructions affect the flags. For example, the zero flag is set when the arithmetic or logical operation results in zero.

2.3.1. Arithmetic Instructions

The following table summarizes the arithmetic instructions of the 8051 family of microprocessors. Note that many of the arithmetic instructions use the register-specific addressing mode.

Table 2.3. Summary of Arithmetic Instructions

Mnemonic	Description	Addressing Modes				
		Direct	Register-Indirect	Register	Immediate Constant	Register-Specific
ADD A, byte	add	✓	✓	✓	✓	
ADDC A, byte	add with carry	✓	✓	✓	✓	
SUBB A, byte	subtract with borrow	✓	✓	✓	✓	
INC A	increment accumulator					✓
INC byte	increment byte	✓	✓	✓		
INC DPTR	increment data pointer					✓
DEC A	decrement accumulator					✓
DEC byte	decrement byte	✓	✓	✓		
MUL AB	multiply accumulator by b register					✓
DIV AB	divide accumulator by b register					✓
DA A	decimal adjust the accumulator					✓

2.3.1.1. Addition and Subtraction Instructions

There are two addition instructions: add (ADD) and add with carry (ADDC). Both instructions compute the sum of two bytes. Add with carry (ADDC) increments the result if the carry bit was previously set. Three of the flags, the carry flag (C), the auxiliary carry flag (AC), and the overflow flag (OV) are set or cleared depending on the result. The C flag is set if the sum exceeds FFh, and cleared otherwise. The AC flag is set if there is a carry from the low nibble to the high nibble, that is, from bit 3 to bit 4, and cleared otherwise. The auxiliary carry is useful in adding Binary Coded Decimal (BCD) numbers. The OV flag is set if there is a carry from bit 7 but not from bit 6, or, a carry from bit 6 but not from bit 7. The OV flag is useful in working with signed integers represented in the two's complement form. Two conditions cause setting the OV flag. If the sum of two positive numbers exceeds 7Fh but is less than FFh, the result, in two's complement notation, appears as a negative number. The OV flag is set to indicate that the result is not to be interpreted as a negative number.

Adding two negative numbers will always result in an external carry bit. If the result of adding two negative numbers is between 0 and 7Fh (100h and 17Fh considering the external carry) then the OV flag is set to indicate that the result is not to be interpreted as a positive number. The first operand is always in the accumulator. The result is kept in the accumulator. The second operand is specified by the direct, register, register indirect, or immediate constant addressing mode.

Type, assemble, and download the following program. Single step through the program to verify that the addition operations perform correctly.

```
#include <sfr51.inc>

   cseg   at 0x8000      ; set the origin

   mov    a, #1          ; put 1 into the accumulator
   add    a, #2          ; add the constant 2 to the
                         ; accumulator (1+2=3)

   mov    0x78, #3       ; put 3 into internal register 78h
   add    a, 0x78        ; add accumulator and register 78h
                         ; (3+3=6)

   mov    0x79, #4       ; put 4 into internal register 79h
   mov    r0, #0x79      ; put 79h into R0
   add    a, @r0         ; add accumulator and 79h (6+4=10)

   mov    r5, #5         ; put 5 into register 5 of current bank
   add    a, r5          ; add accumulator and r5 (10+5=15)

   ljmp   0              ; return to monitor
   end
```

The "add with carry" instruction ADDC is the same as the add instruction, except that the carry flag is added to the result. The C, AC, and OV flags are affected in the same manner as with the ADD instruction. ADDC is especially useful in adding long integers. Consider adding two 16-bit integers X and Y. Let XH and XL be the high and low bytes of X. Similarly, let YH and YL be the high and low bytes of Y. Let these bytes be stored in internal registers, as given below.

Byte	Register
XL	78h
XH	79h
YL	7Ah
YH	7Bh

Now consider the following program. The registers are defined in internal data RAM using the MCS-51 directive "data."

```
#include <sfr51.inc>
XL data 0x78
XH data 0x79
YL data 0x7A
YH data 0x7B

   cseg at 0x8000      ; set the origin

; --- first set up the long integers X and Y
   mov    XL, #0x34    ; X=1234h, XL=34h
   mov    XH, #0x12    ;          XH=12h
   mov    YL, #0xEF    ; Y=12EFh, YL=EFh
   mov    YH, #0x12    ;          YH=12h

; --- add X and Y.  Place result in X

   mov    a, XL        ; first the low byte
   add    a, YL        ; add YL to XL
   mov    XL, a        ; put result in XL
; --- note that at this point CL holds 0x23 and ...
; --- ... that the carry flag is set

   mov    a, XH        ; next the high byte
   addc   a, YH        ; include the carry bit from previous
                       ; operation (XL+YL)
   mov    XH, a        ; store result in XH (XH holds 0x25)

   ljmp   0            ; return to monitor
   end
```

Assemble the program. Single step through the program and verify that the result 0x2523 is correctly stored in internal registers XH:XL (78h and 79h). Note that multi-byte values stored in several registers are denoted by the byte register names separated by colons. In this case, XH:XL is a two-byte value held in XH and XL.

The subtraction operation SUBB also uses the accumulator as the first operand. The second operand is subtracted from the accumulator and the result is placed in the accumulator. The carry flag (C) is used to indicate a borrow in multiple precision subtraction. The carry flag is set if the subtraction operation requires an external borrow. If the C flag was set before the subtraction instruction, then the result is decremented. This way, a multiple precision subtraction can be performed by starting with the least significant bytes and proceeding towards the most significant bytes. At each subtraction, if an external borrow is required, the C flag is set, and during the subsequent subtraction, the result is decremented to account for the external borrow

of the previous operation. Since the result is decremented if the carry flag was set before the operation, it is necessary to assure that the carry flag is cleared before the first subtraction. The carry flag may be cleared by the bit-oriented clear instruction

```
clr C
```

The Auxiliary Carry (AC) flag and the Overflow flag (OV) are similarly set. The AC is set if there is a borrow needed for the low nibble (bit 3). The OV flag is set if there is borrow needed for bit 7 but not for bit 6, or, if there is a borrow needed for bit 7 but not for bit 6. The overflow flag is useful in working with signed integers represented in two's complement notation. The OV flag indicates that a positive number is obtained when a positive number is subtracted from a negative number, or when a negative number is obtained when a negative number is subtracted from a positive number. The addressing modes are the same as for the addition operation.

Again, consider the 16-bit integers X and Y as discussed above. The following program finds the 16-bit difference X-Y and stores the result in X.

```
#include <sfr51.inc>
XL data 0x78
XH data 0x79
YL data 0x7A
YH data 0x7B

    cseg   at 0x8000     ; set the origin

; --- first set up the long integers X and Y
    mov    XL, #0x34     ; X=1234h, XL=34h
    mov    XH, #0x12     ;          XH=12h
    mov    YL, #0x35     ; Y=1135h, YL=35h
    mov    YH, #0x11     ;          YH=11h

    clr    C             ; clear the carry flag
; --- subtract Y from X.  Place result in X

    mov    a, XL         ; first subtract YL from XL
    subb   a, YL         ; XL-YL
    mov    XL, a         ; put result in XL

; --- note that at this point XL holds 0xFF and
; --- that the carry flag, indicating an external
; --- barrow, is set

    mov    a, XH         ; next subtract YH from XH
    subb   a, YH         ; include the carry bit from
                         ; previous operation
    mov    XH, a         ; store result in XH (XH holds 0)
```

```
        ljmp   0              ; return to monitor
        end
```

Assemble the program. Single step through the program and verify that the result 0x00FF is correctly stored in internal registers XH:XL (78h:79h).

Review the simple program that illustrates the ADD instruction with different addressing modes given above. Replace the ADD instructions with SUBB instructions. Make sure that the carry flag is cleared before the operation by inserting

```
    clr C
```

or

```
    mov psw, #0
```

instructions. Single step through the program to verify that SUBB instructions produce the correct results.

2.3.1.2. Increment and Decrement Instructions

The 8051 increment and decrement instructions use register-specific, register, direct, and register indirect addressing modes. The direct addressing mode allows incrementing or decrementing any internal RAM byte or SFR without the need to move its contents first into the accumulator, modifying the value, and transferring the result back into internal RAM or SFR.

Increment and decrement operations are useful in implementing loop counters or pointers to data. For example, a string to be transmitted from the serial port may reside in consecutive registers. A period may be used to denote the end of the string. A simple loop can initialize a pointer to be the address of the first character of the string. The loop then may read the character from the pointer address, send it to the serial port, increment the pointer, and repeat until the period is transmitted.

The following program illustrates adding multiple-precision integers. Let X and Y be 4-byte long integers. Let X be stored in register X3:X2:X1:X0 (73h:72h:71h:70h). Similarly, let Y be stored in registers Y3:Y2:Y1:Y0 (7Bh:7Ah:79h:78h). Let r3 hold the number of bytes in X and Y, i.e., 4, which is the precision of the integers.

Now consider the following program. Although branching instructions are explained in Section 2.4, the program necessarily uses a branching instruction at the end of each loop iteration. The loop counter r3 is decremented at the end of each iteration. If R3 is not zero, the loop continues. Otherwise, the loop terminates.

```
    #include <sfr51.inc>

    X0 data 0x70
```

```
X1 data 0x71
X2 data 0x72
X3 data 0x73

Y0 data 0x78
Y1 data 0x79
Y2 data 0x7A
Y3 data 0x7B

   cseg  at 0x8000 ; set the origin

; --- first set up the 4-byte long integers X and Y
   mov   X0, #0x67    ; X=01234567h
   mov   X1, #0x45
   mov   X2, #0x23
   mov   X3, #0x01

   mov   Y0, #0xEF    ; Y=89ABCDEFh
   mov   Y1, #0xCD
   mov   Y2, #0xAB
   mov   Y3, #0x89

; --- specify the precision
   mov   r3, #4

; --- set up the pointers r0 and r1
   mov   r0, #0x70    ; r0 points to X0
   mov   r1, #0x78    ; r1 points to Y0

; --- add X and Y.  Place result in X
   clr C                ; clear carry flag
loop:
   mov   a, @r0         ; get byte of X
   addc  a, @r1         ; add byte of Y
   mov   @r0, a         ; put result in byte of X
   inc   r0             ; increment pointer to next byte
                        ; of X
   inc   r1             ; increment pointer to next byte
                        ; of Y
   djnz  r3, loop       ; decrement loop counter and jump
                        ; to label loop: if not zero

   ljmp  0              ; return to monitor
   end
```

Assemble and single step through the program and verify that the loop is executed four times and that the correct result is placed in the bytes X3:X2:X1:X0 (73h to 70h). Try the experiment with different values of X and Y. Also, try the experiment with

different number of bytes. Use registers 0x70 to 0x77 for up to 8-byte X values. Similarly, use 0x78 to 0x7F for up to 8-byte Y values. Initialize r3 to the precision of the numbers. You may define the higher bytes of X and Y as shown below to improve the readability of your program.

```
X0 data 0x70
X1 data 0x71
X2 data 0x72
X3 data 0x73
X4 data 0x74
X5 data 0x75
X6 data 0x76
X7 data 0x77

Y0 data 0x78
Y1 data 0x79
Y2 data 0x7A
Y3 data 0x7B
Y4 data 0x7C
Y5 data 0x7D
Y6 data 0x7E
Y7 data 0x7F
```

2.3.1.3. Multiplication and Division

The 8051 family of microcontrollers has hardware multiplication and division units. These instructions take the longest time (four machine cycles). The multiplication and division instructions are register specific. The multiplication and division instructions always use the accumulator and B register. The instruction

```
mul ab
```

multiplies the two 8-bit unsigned integers in the accumulator and in the B register. The low byte of the 16-bit result is kept in the accumulator, and the high byte, in the B register. Note that the result cannot be greater than 0xFFFF. Thus, the carry flag is never set. The overflow flag is set if the result is greater than 0xFF. The overflow flag being cleared implies that the B register is 0. The instruction

```
div ab
```

divides the 8-bit unsigned integer in the accumulator by the 8-bit unsigned integer in the B register. The integer part of the result is kept in the accumulator. The remainder is placed in the B register. The carry flag is always cleared. The overflow flag is used to indicate a divide-by-zero condition. If the B register contained 0 before the instruction, both the integer and remainder of the result are undetermined, and the overflow flag is set.

Consider the following program.

```
#include <sfr51.inc>

    cseg   at 0x8000      ; set the origin

    mov    a, #9          ; set up accumulator
    mov    b, #5          ; set up B register
    mul    ab             ; multiply (9X5=45 or 2Dh)

; --- observe that B=0 and A=0x2D ------

    mov    a, #99         ; set up accumulator
    mov    b, #5          ; set up B register
    mul    ab             ; multiply (99*5=495 or 0x1EF)

; --- observe that B=1 and A=0xEF ------

    mov    a, #10         ; set up accumulator
    mov    b, #5          ; set up B register
    div    ab             ; divide (10/5=2, remainder=0)

; --- observe that B=0 and A=2 ------

    mov    a, #99         ; set up accumulator
    mov    b, #5          ; set up B register
    div    ab             ; divide(99/5=19, or 13h, and
                          ; remainder=4)

; --- observe that B=4 and A=13h ------

    mov    a, #99         ; set up accumulator
    mov    b, #0          ; set up B register
    div    ab             ; divide(99/0.  result is undefined
; --- observe that both B and A are arbitrary and that
; --- the overflow flag is set.

    ljmp   0              ; return to monitor
    end
```

Assemble and single step through the program to verify the results. The overflow flag is bit 2 of the PSW.

2.3.1.4. The Decimal Adjust Instruction

This instruction is used after adding Binary Coded Decimal (BCD) numbers. Please refer to Appendix C for a brief overview of BCD numbers. Each nibble of a BCD number represents a decimal digit. Thus, the value of each nibble may be in the

interval [0..9]. Consider adding the two BCD numbers 12 and 29. These numbers are represented by 12h and 19h. Their sum obtained by an ADD or ADDC instruction without a previous carry produces 0x2B. The value 0x2B is no longer a BCD number. Since the first digit (Bh) exceeds 9, an adjustment is required. Specifically, 10 should be subtracted from the first nibble, and a carry should be added to the high nibble. This will make the low nibble (Bh - 10 = 1) and the high nibble (2+1=3). These two adjustments can be carried out by simply adding 6 to the low nibble. The same adjustment is required if the sum of the two low nibbles exceeds 16. This condition is detected from the fact that the Auxiliary Carry (AC) flag is set by the ADD or ADDC instruction when a carry from bit 3 to bit 4 is generated. Similarly, if the high nibble exceeds 9 after an ADD or ADDC instruction, the decimal adjust instruction adds 6 to the high nibble, or 0x60 to the byte.

The decimal adjust instruction may be viewed as adding 0, 6, 0x60, or 0x66 to the accumulator, depending on the contents of the accumulator and of the flags in PSW. Note that the decimal adjust instruction by itself cannot convert a hexadecimal number to a decimal number.

Consider the following program.

```
#include <sfr51.inc>

    cseg   at 0x8000     ; set the origin

    mov    a, #0x12      ; set up accumulator
    mov    b, #0x29      ; set up B register
    add    a, b          ; add (12h+29h=2Bh, not a BCD)
    da     a             ; decimal adjust
; --- observe that 6 is added to the result
; --- 2Bh+6=31h, the correct BCD of 12(BCD) + 29(BCD)

    mov    a, #0x55      ; set up accumulator
    mov    b, #0x66      ; set up B register
    add    a, b          ; add (55h+66h=BBh, not a BCD)
    da     a             ; decimal adjust
; --- observe that 0x66 is added to the result
; --- BBh+66h=121h, the correct BCD of 55(BCD) + 66(BCD)
; --- the accumulator holds 21 (BCD) and the carry flag
; --- is set.

    ljmp 0               ; return to monitor
    end
```

Assemble and single step through the program to verify the results. The overflow flag (OV) is bit 2 of the PSW.

2.3.2. Logic Instructions

The byte-wise logical operations are summarized below.

Table 2.4. Summary of Byte-Oriented Logical Instructions

	Addressing Modes					
Mnemonic	Description	Direct	Register Indirect	Register	Immediate Constant	Register-Specific
ANL A, byte	And	✓	✓	✓	✓	
ANL byte, A	And	✓				
ANL byte, #constant	Add	✓				
ORL A, byte	Or	✓	✓	✓	✓	
ORL byte, A	Or	✓				
ORL byte, #constant	Or	✓				
XRL A, byte	Exclusive or	✓	✓	✓	✓	
XRL byte, A	Exclusive or	✓				
XRL byte, #constant	Exclusive or	✓				
CLR A	Clear accumulator	✓	✓	✓		
CPL A	Complement accumulator					✓
RL A	Rotate accumulator left					✓
RLC A	Rotate accumulator and carry flag left					✓
RR A	Rotate accumulator left					✓
RRC A	Rotate accumulator and carry flag left					✓
SWAP A	Swap accumulator nibbles					✓

2.3.2.1. AND, OR, and EXCLUSIVE-OR Instructions

The *AND*, *OR*, and *EXCLUSIVE-OR* instructions have the same addressing modes. The byte-wise logical operations may be viewed as eight bit-wise operations performed on corresponding bits of the operand bytes. Note that one of the operands

is always the accumulator. The first operand is also the destination, i.e., where the result is stored. No flags are affected.

The AND and OR instructions are useful in masking specific bits of a control register. Consider selecting register bank 3 without affecting the other flags in the PSW. Of course, since the individual bits of the PSW can be addressed, setting or clearing individual PSW bits to select the register bank is the preferred procedure in most applications.

The following instructions force bits 3 and 4 of the PSW to 1. Note that the operand PSW is a constant defined to be the value D0h, the address of the special function register PSW.

```
orl    psw, #0x18    ; bits 3 and 4 high
```

Recall that the result of an OR operation is 1 except if both operands are 0. Now consider selecting register bank 0. In this case, bits 3 and 4 must be forced to 0.

```
anl    psw, #0xE7    ; bits 3 and 4 low
```

Note that 0xE7=0xFF-0x18, i.e., 0xE7 is the complement of 0x18. The mask byte 0xE7 sets all bits except bits 3 and 4 of the mask byte. The mask byte 0x18 clears all bits except bits 3 and 4. Now consider selecting register bank 1 using mask bytes and logical operations.

```
anl    psw, #0xEF    ; bit 4 low
orl    psw, #0x08    ; bit 3 high
```

The Reads51 assembler supports the C-like complement operator applied to constant expressions. The constant 0xEF may be written as ~0x10. Then, as mentioned above, the AND instruction may be written as below.

```
anl    psw, #~0x10   ; bit 4 low
```

Use the complemented version if you find it to be more readable. In fact, the Reads51 expression evaluator supports almost all C-type expressions. The instruction may also be written using the shift operator as follows.

```
anl    psw, #~(1<<4); bit 4 low
```

Note that the expression evaluator reduces the expression ~(1<<4) to 0xEF during assembly. That is, the expression ~(1<<4) is simply replaced by the result (0xEF) before the code is assembled. Thus, using such expressions does not add any addition instructions to be executed in run time. C programmers may find the C-like expressions more readable. Write a short program that implements the above

operations. Assemble the program. Single step through the program and verify its operation.

As another experiment, connect eight LEDs to Port 1 (see Chapter 7). Assemble the following program.

```
#include <sfr51.inc>

   cseg   at 0x8000     ; set the origin

   mov    P1, #0xAA     ; AAh=10101010b
   anl    P1, #0x0F     ; force high nibble low

   mov    P1, #0xAA     ; AAh=10101010b
   orl    P1, #0x0F     ; force low nibble high

   ljmp   0             ; return to the monitor
   end
```

Placing 0xAA on Port 1 lights LEDs 0, 2, 4, and 6, that is, the LEDs corresponding to a bit with value 0. Similarly, the set bits 1, 3, 5, and 7 turn the connected LEDs off. When P1 is AND-ed with 0Fh, the high nibble is forced low, thus lighting LEDs 5 and 7 as well. In the second portion, the low nibble is forced high, thus extinguishing LEDs 0 and 2.

2.3.2.2. Complement and Clear Instructions

The clear and complement instructions are register specific, operating on the accumulator. The clear instruction

```
clr    a
```

clears all bits of the accumulator. This instruction may be viewed as moving the constant 0 into the accumulator. The instruction

```
cpl    a
```

complements each bit of the accumulator. Consider the following program.

```
cseg   at 0x8000     ; set the origin

mov    a, #0xAA      ; place AAh in the accumulator
cpl    a             ; complement a.  complement of
                     ; AAh is 55h
clr    a             ; clear the accumulator

ljmp   0             ; return to the monitor
end
```

Assemble the program. Single step through the program and verify that first the accumulator is set to AAh, then it is complemented, leaving 55h in the accumulator, and finally, is cleared.

2.3.2.3. Rotate Instructions

Rotate instructions are register specific, operating only on the accumulator. The bits of the accumulator are shifted one digit, similar to the operation of a shift register. However, in a rotate instruction, the bit that is shifted out is used as the new bit shifted in. There are four rotate instructions, two for rotating to the right and two for left. Rotates may involve only the bits of the accumulator, or they may include the carry flag as the ninth bit.

The rotate instructions are useful in generating mask bytes as well as in multiplying or dividing by powers of 2. Shifting the accumulator to the left one digit, provided that bit 0 is first cleared, is equivalent to multiplying the accumulator by 2. Similarly, two successive rotates constitute multiplying the accumulator by 4. Rotating to the right is equivalent to division by 2. Consider the following program.

```
        cseg    at 0x8000       ; set the origin

        mov     a, #0x80        ; place 80h in the accumulator
        rr      a               ; rotate right, resulting in 40h
                                ; (80h/2)

        mov     a, #0x12        ; place 12h in the accumulator
        rl      a               ; rotate left, resulting in 24h
                                ;(12h*2)

        mov     a, #0x21        ; place 21h in the accumulator
        setb    C               ; set the carry flag
        rrc     a               ; rotate accumulator and C to the
                                ; right
;  --- C becomes the most significant bit in the
;  --- accumulator, and the least significant bit of the
;  --- accumulator is moved to the carry flag.  Note that
;  --- after instruction, the carry flag is set, and the
;  --- accumulator holds 90h.

        ljmp    0               ; return to the monitor
        end
```

Assemble the program. Single step through the program and verify the operation of the rotate instructions. If the carry bit was cleared, instead of set, just before the rotate right with carry instruction, the value of the accumulator after the instruction would be 10h, with the carry flag set. Verify this prediction.

Replace the rotate-right and rotate-right-with-carry instructions with rotate-left and rotate-left-with-carry instructions, and repeat the experiment.

2.3.2.4. The Swap Instruction

The instruction

```
swap      a
```

interchanges the two nibbles of the accumulator. It can be viewed as rotating the accumulator 4 times, or 4 bits (in either direction). This instruction is useful in converting two ASCII characters to a byte. Refer to the subroutine BINASC discussed in Section 3.4 for an example.

2.3.2.5. Bit-Oriented Logical Instructions

The 8051 family of microcontrollers has extensive bit-oriented features. The addressable bits of the 8051 family of microcontrollers are discussed in Section 2.2.9.

The carry flag (C) is used by the 8051 family of microcontrollers as the one-bit accumulator of bit-oriented operations. For example, all bit-wise logical operations involve C. The bit-oriented logical operations may use the source bit or its complement. That is, the source bit need not be complemented explicitly. The value of the source bit remains unchanged, although its complement may be used in bit-wise logical operations.

Table 2.5. Summary of Bit-Oriented Logical Instructions

Mnemonic	Description
ANL C, bit	And carry flag with addressable bit
ANL C, /bit	And carry flag with complement of addressable bit
ORL C, bit	Or carry flag with addressable bit
ORL C, /bit	Or carry flag with complement of addressable bit
CLR C	Clear carry flag
CLR bit	Clear addressable bit
CPL C	Complement carry flag
CPL bit	Complement addressable bit
SETB C	Set Carry flag
SETB bit	Set addressable bit

Consider the following program.

```
#include <sfr51.inc>

    cseg   at 0x8000     ; set the origin
```

```
clr   a              ; clear the accumulator
setb  acc.0          ; set bit 0 of the accumulator

setb  C              ; set the carry flag
orl   C, acc.0       ; OR C with bit 0 of the
                     ; accumulator
setb  C              ; set the carry flag
orl   C, /acc.0      ; OR C with complement of bit 0
                     ; of the accumulator
ljmp  0              ; return to the monitor
end
```

First, the least significant bit of the accumulator and the carry flag are set. Note that acc.0 has the bit address A0h. Using the symbol acc.0 rather than A0h improves code readability. Since both bits are set, the EXOR operation results in a 0, which is placed in the carry flag.

Next C is set again, and it is EXOR-ed with the complement of acc.0. Note that the result of this operation is 1. Also, note that although the complement of acc.0 is used in the operation, the value of acc.0 is not modified. The accumulator still holds 1. Assemble the program. Single step through the program to verify its operation.

2.4. Program Flow Control Instructions

Conceptually, branching instructions allow the microcontroller to take different actions at run time, depending on the data. For example, depending on the state of a push button, the microcontroller may be programmed to stop or continue running a motor. There are two different actions, one to stop the motor, and the other, to continue running it. Each of these actions is ultimately implemented as a set of microcontroller instructions. Typically, the push button is connected to an input port. Its state thus becomes the state of a port bit. The state of the port constitutes a single bit of data. Depending on this data, one of the two possible sets of instructions is executed.

In hardware, a branching instruction is one that modifies (or conditionally modifies) the program counter defining the address of the next instruction to be fetched. The instruction fetch cycle was discussed in Section 2.1. The microcontroller updates the program counter to point to the next instruction. Normally, the next instruction to be executed is thus the one immediately following the current instruction. If an instruction places a different value in the program counter, the next instruction to be executed may be anywhere in the microcontrollers program memory.

Branching instructions include jump, call and return instructions. A jump instruction simply modifies the program counter. Jump instructions may further be partitioned into unconditional and conditional jumps. The 8051 family of microcontrollers has three unconditional jump instructions: sjmp, ajmp, and ljmp. These instructions have a single operand that describes how the program counter is to be modified.

2.4.1. Unconditional Jump Instructions

Below is a summary of the unconditional jump instructions. The program flow in each of these instructions is altered by modifying the contents of the program counter. The indexed jump instruction is a powerful instruction that adds the contents of the accumulator to the data pointer to compute the address of the next instruction to be fetched and executed.

Table 2.6. Unconditional Jump Instructions

Mnemonic	Description
SJMP <rel add>	Short jump. The operand is a single-byte two's complement value which is added to the program counter. When the microcontroller executes the instruction, the program counter points to the immediately following instruction. The next instruction may be up to 127 bytes ahead or up to 128 bytes behind this following instruction.
AJMP <Address 11>	Absolute jump. The operand is an 11-bit address within the current 2-kilobyte block of program memory.
LJMP <Address 16>	Long jump. The operand is a 16-bit address anywhere in the 64-kilobyte program memory.
JMP @A+DPTR	Long jump. The address of the next instruction to be executed is the sum of the data pointer and the accumulator.

The short jump instructions are used with conditional jumps. Short jumps are limited to the range of 127 bytes ahead or 128 bytes behind the instruction following the jump instruction. Note that since the jump address is described as a relative offset from the next instruction, the resultant code is relocatable. A program or block of code is called relocatable if it executes correctly, no matter where it is placed within the program memory. Collections of simple operations such as move or arithmetic instructions are clearly always relocatable since the instruction fetch cycle will assure the execution of the instructions in sequence no matter where they are placed. Relocatability becomes important when branching instructions are used. If a branching instruction directs the flow of the program to a specific address in the program memory, it is the responsibility of the programmer to assure that valid code exists at the jump address. If a block of code uses short jumps, however, since the jump address is computed as a relative offset from the next instruction, the program will execute properly no matter where it is placed. Consider, for example, the following program segment discussed in Section 2.2.9.

```
#include <sfr51.inc>

    cseg   at 0x8000
    mov    C, P1.0        ; move the button state into the carry
                          ; flag
    mov    P1.1, C        ; move the carry flag to the LED
    ljmp   0x8000         ; repeat
    end
```

This program must be placed starting at 0x8000 since its last instruction makes a jump to 0x8000. If the program were to be placed, say, at 0x9000, it would execute, copying the state of the pushbutton to the LED. But then, as the last instruction is executed, the program would jump to 0x8000 and attempt to continue to execute. If no meaningful program were present at 0x8000, the program would most probably crash. Now consider the following modification.

```
    cseg   at 0x8000
start:
    mov    C, P1.0
    mov    P1.1, C
    sjmp   start
    end
```

If the origin were to be changed to, say, 0x9000, the program will still function properly. Each of the three instructions takes two bytes. That is the PC has the value 0x8000 as it executes the first instruction. The first instruction is 2 bytes long, so the PC is incremented twice, and now has the value 0x8002. Similarly, the PC is incremented to 0x8004 before the last instruction is executed. The last instruction is also 2 bytes long, incrementing the PC to 0x8006. However, since the last instruction is a branching instruction, the relative jump of 6 bytes behind results in the PC value of 0x8000. The processor thus is in an endless loop. If origin is changed to 0x9000, the program still functions properly. The instructions start at addresses 0x9000, 0x 9002, and 0x 9004. When the short jump instruction is executed, the PC is restored to 0x 9000, repeating the loop. The second version of the program with the short jump is relocatable anywhere within the program memory, where as the first version is memory specific.

The above discussion can be generalized to state that programs that use long jumps are memory specific, while programs using short jumps are relocatable. It is easy to see that attaining relocatability by disallowing the use of long jumps is counter-productive. This gives rise to the need to separate the compilation and linking phases of code generation. In compilation, most of the assembly code is converted into machine language except for the specific addresses, such as those that follow long jumps. Code segments and subroutines so compiled are later linked together to generate the final executable program. The latter linking phase is more than just a

simple concatenation of the individual modules. External references to jump addresses as well as variables must be resolved at this stage.

Absolute jumps of the 8051 provide an alternative to separating the compile and link processes while still allowing for some relocatability of the code. Absolute jumps define the least significant 11 bits of the jump address. Thus, the jump location is within the same 2K block of program memory. Any program segment written using only absolute jumps and short jumps may be placed at any 2K block. That is at any of the origins 0, 0x 800, 0x 1000, 0x1800, ... , 0xF800. Since many embedded control software fits within 2K of memory, absolute jumps provide a practical way of writing a small chunk of code which can be located at 32 different addresses with minimal effort.

Many of the example programs given in this book terminate with a long jump to address 0. This essentially causes a reset, and any monitor program running on the board takes over. The indexed jump instruction provides a convenient means to construct a jump table. Consider the following program.

```
#include <sfr51.inc>

    cseg   at 0x8000

    mov    a, #0          ; initialize the accumulator
                          ; (try other values)
    anl    a, #3          ; make sure a=0, 1, 2, or 3.
    rl     a              ; multiply by 2
    rl     a              ; multiply by 2 again
    mov    dptr, #alternatives
    jmp    @a+dptr        ; branch to an entry of the
                          ; alternatives table

alternatives:
    mov    b, P0          ; action for routine 0
    sjmp   continue

    mov    b, P1          ; action for routine 1
    sjmp   continue

    mov    b, P2          ; action for routine 2
    sjmp   continue

    mov    b, P3          ; action for routine 3

continue:
    mov    a, P1          ; get Port 1 and place in a
    mul    ab             ; multiply Port 1 and Port n,
                          ; where n is the contents of the
```

```
                        ; accumulator at the beginning of
                        ; the program.
    ljmp   0
    end
```

This program multiplies the contents of P1 with the contents of P0, P1, P2, or P3, depending on the value of the accumulator. Notice that the "alternatives" table has four entries, consisting of a move instruction and a short jump instruction. Since these two instructions total 4 bytes, the accumulator is multiplied by 4. This allows up to 64 alternatives. The data pointer is initialized as the base address of the "alternatives" table. The indexed jump instruction branches the program flow to the desired move instruction. The program then proceeds to the continue label. Assemble the program. Single step through the program to verify its operation. Try starting with different values of the accumulator.

If the action associated with each alternative requires many bytes of code, the above program may be modified to implement a jump table. Consider the following modification. Each routine may consist of many instructions. The capacity of the jump table is still 64. Again, assemble the program and single step through the program to verify its operation.

```
#include <sfr51.inc>
    cseg   at0x8000

    mov    a, #0          ; initialize the accumulator
                          ; (also try 0, 1, 2, and 3)
    anl    a, #3          ; make sure a=0, 1, 2, or 3.
    rl     a              ; multiply by 2
    rl     a              ; multiply by 2 again
    mov    dptr, #jump_table
    jmp    @a+dptr        ; branch to an entry of the jump
                          ; table

    jump_table:
    ljmp   routine_0      ; each ljmp instruction is 3
                          ; bytes.
    nop                   ; with the NOP instruction, the
                          ; ljmp instructions are 4 bytes
                          ; apart
    ljmp   routine_1
    nop
    ljmp   routine_2
    nop
    ljmp   routine_3      ; up to 64 jumps may implemented

; --- the set of routines pointed to by entries of the
; --- jump table. each routine terminates with a jump to
```

```
; --- label 'continue'
routine_0:
   mov    b, P0          ; action for routine 0
   ljmp   continue
routine_1:
   mov    b, P1          ; action for routine 1
   ljmp   continue
routine_2:
   mov    b, P2          ; action for routine 2
   ljmp   continue
routine_3:
   mov    b, P3          ; action for routine 3

continue:
   mov    a, P1          ; get Port 1 and place in a
   mul    ab             ; multiply Port 1 and Port n,
                         ; where n is the contents of the
                         ; accumulator at the beginning of
                         ; the program.
   ljmp   0
   end
```

As seen from these experiments, the MCS-51 indexed jump is reminiscent of the "computed goto" statement of older languages such as Fortran.

2.4.2. Conditional Jump Instructions

The conditional jumps modify the program counter only if a certain condition holds. For example, the instruction

```
jnc    label
```

modifies the program counter so that the next instruction to be executed is at the location 'label' only if the carry flag is cleared. The mnemonic jnc is short for "jump if no carry."

The 8051 family of microcontrollers has powerful higher-level jump instructions that perform an operation and branch depending on the result of the operation. The instruction

```
cjne   <src-byte>, <dest-byte>, <address>
```

compares the two bytes, the first and second operands, referred to as the source byte and the destination byte. The program flow branches to the new instruction at <address> if the two bytes are different. The first two operands are typically a register and a constant. The mnemonic cjne is short for "compare and jump if not equal." The carry flag is set if the unsigned integer value of the <src-byte> is less than the

unsigned integer value of <dest-byte>. Otherwise, the carry flag is cleared. Neither operand is affected.

Similarly, the instruction

```
djnz   <byte>, <address>
```

decrements the first operand <byte>, which is typically a register, and branches to the instruction at <address> if the decrement operation did not result in zero. The mnemonic djnz stands for "decrement and jump if not zero." These instructions simplify constructing program loops. The conditional instructions are summarized below.

Table 2.7. Conditional Jump Instructions

Mnemonic	Description
`JZ  <rel add>`	The operand is a single-byte two's complement value which is added to the program counter if accumulator holds zero.
`JNZ <rel add>`	The operand is a single-byte two's complement value which is added to the program counter if accumulator is not zero.
`JC  <rel add>`	The operand is a single-byte two's complement value which is added to the program counter if the carry flag is set.
`JNC <rel add>`	The operand is a single-byte two's complement value which is added to the program counter if the carry flag is cleared (not set).
`JB  <bit>, <rel add>`	The second operand is a single-byte two's complement value which is added to the program counter if the first operand, the addressable bit, is set.
`JNB <bit>, <rel add>`	The second operand is a single-byte two's complement value which is added to the program counter if the first operand, the addressable bit, is cleared (not set).
`JBC <bit>, <rel add>`	The second operand is a single-byte two's complement value which is added to the program counter if the first operand, the addressable bit is set. The bit is cleared after the instruction.
`CJNE A, direct, <rel add>`	The first operand is the accumulator, the second, an internal register, and the third, a relative address. The relative address is added to the program counter if the content of the accumulator is different from that of the register.

CJNE A, #data, <rel add>	The first operand is the accumulator, the second, a constant, and the third, a relative address. The relative address is added to the program counter if the contents of the accumulator is different from that the constant.
CJNE Rn, #data, <rel add>	The first operand is a register of the current register bank, the second, a constant, and the third, a relative address. The relative address is added to the program counter if the content of the register is different from the constant.
CJNE @Ri, #data, <rel add>	The first operand is the register whose address is in register Ri, the second, a constant, and the third, a relative address. The relative address is added to the program counter if the content of the register is different from the constant.
DJNZ Rn, <rel add>	The first operand is a register of the current register bank, and the second, a relative address. The register is decremented. The relative address is added to the program counter if the (decremented) content of the register is non-zero.
DJNZ Direct, <rel add>	The first operand is an internal register, and the second, a relative address. The register is decremented. The relative address is added to the program counter if the (decremented) content of the register is non-zero.

The following simple example flashes an LED connected to bit 0 of Port 1. The LED is turned on for a period of time, and then off again. The on and off periods are determined by the loops which serve no purpose other than to waste the processors time. We use an internal bit variable, which we name LED_flag. The bit variable is defined using the MCS-51 "bit" directive. This is similar to the "EQU" pseudo-op, but is more preferable since it also conveys type information.

```
#include <sfr51.inc>

LED_flag bit 0x0F       ; software flag, set when
                        ; LED is on (you may
                        ; replace "bit" by EQU
   cseg at 0x8000
start:
; if LED is currently on branch to LED_OFF
   jb    LED_flag, LED_OFF

; else set LED_flag and turn LED on
   setb  LED_flag
   clr   P1.0
```

```
; implement a nested loop to waste time
    mov    r6, #0
    mov    r7, #0
wait_1:
    djnz   r6, wait_1
    djnz   r7, wait_1
    sjmp   start          ; repeat

LED_OFF:                  ; branch here if LED is
                          ; currently on
    clr    LED_flag       ; first clear flag
    setb   P1.0           ; turn LED off

    mov    r6, #0         ; implement a nested loop
                          ; to waste time
    mov    r7, #0
wait_2:
    djnz   r6, wait_2
    djnz   r7, wait_2

    ljmp   start          ; repeat
    end
```

This program illustrates conditional jumps, "decrement and jump if not zero" instructions, and unconditional jumps. The nested loops are intended to waste a little of the microcontrollers time. Notice that the inner loop is executed 256 times, until r6 is decremented to 0. The outer loops are also executed 256 times, until r7 is decremented to 0. Each inner loop takes 2 machine cycles, or 2 microseconds with a 12 MHz crystal. Thus, each inner loop takes 512+2 microseconds. The additional 2 microseconds come from executing a single iteration of the outer loop. The outer loop is executed 256 times, giving a total delay period of 256×514=131584 microseconds, or about 0.13 seconds.

Connect an LED to P1.0 through a 220 to 470 Ohm resistor. Assemble the program. Single step through the program. You may set new break points immediately after the wait loops to prevent single stepping through the entire nested loop.

2.4.3. Call and Return Instructions

The call and return instructions differ from jumps in that a call pushes the current program counter on the stack before branching. The return instruction is the counterpart of the call instruction. It pops the program counter off the stack. Thus, the instruction that is executed after a return is the one whose address was at the top of the stack. Strictly speaking, a return instruction need not follow a call. The program may explicitly push an address on the stack and branch to that address by a return instruction. This scheme is useful in implementing a "computed goto."

Table 2.8. Call and Return Instructions

ACALL <Address 11>	The 11-bit address <Address 11> is placed in the program counter as the current program counter is pushed on stack. The 11-bit address allows the program to branch to any instruction within the current 2 kilobytes block of program memory
LCALL <Address 16>	Branch anywhere within program memory and store the current program counter on stack. <Address 16> is the 16-bit address of the 64-kilobyte program memory.
RET	Pop program counter off the stack.
RETI	Pop program counter off the stack and reset interrupt hardware.

Again, the absolute call differs from the long call in that only the least significant 11 bits of the target address are specified. The branching is limited to the current 2K block of program memory. Similar to the case of absolute and long jumps described in Section 2.4.1, absolute calls allow relocatability of 2K blocks of code.

Consider the program given in the previous section. There are two identical wait loops, one to wait for the LED to turn on, and the other, to turn off. Since these nested loops are identical, a subroutine may be written to implement the nested loops, and the subroutine may be called twice, one to wait for the LED to be turned on, and the other, to be turned off.

```
#include <sfr51.inc>

LED_flag bit 0x0F       ; software flag, set when
                        ; LED is on
    cseg   at 0x8000
start:
; if LED is currently on branch to LED_OFF
    jb     LED_flag, LED_OFF

; else set LED_flag and turn LED on
    setb   LED_flag
    clr    P1.0

    lcall wait          ; delay loop
    sjmp   start        ; repeat

LED_OFF:                ; branch here if LED is currently on
    clr    LED_flag     ; first clear flag
    setb   P1.0         ; turn LED off
    lcall  wait
    ljmp   start        ; repeat
```

```
; --- subroutine wait ---------------------------------
wait:
    mov    r6, #0        ; implement a nested loop
    mov    r7, #0        ; to waste time
wait_1:
    djnz   r6, wait_1
    djnz   r7, wait_1
    ret
    end
```

Call and return instructions are necessary to implement subroutines. The skillful use of subroutines is essential to write structured programs. A structured program is one that consists of many layers of subroutines. Each logical task is accomplished by a subroutine. These subroutines are debugged and saved. Higher-level subroutines call the lower level subroutines, thus improving code readability, and simplifying code maintenance and modifications.

As mentioned, a return instruction need not follow a corresponding call instruction. Recall that RET branches to the code address that is currently at the top of the stack. If we deliberately push an address on stack and invoke the RET address, we essentially branch to the address pushed on stack. Since the address to be pushed on stack, is now completely under software control, we can branch to addresses taken from tables, or even computed in some manner. The following portion of a program is given to illustrate how a "computed goto" may be accomplished in assembly language.

```
#include <sfr51.inc>
    cseg   at 0x8000
    mov    a, #0x81      ; set the accumulator to 81h
    push   acc           ; push 81h on stack
    clr    a             ; move 0 in to the accumulator
    push   acc           ; push 0 on to stack
    ret                  ; a return instruction
                         ; effectively performs a long
                         ; jump to 8100h
    end

    cseg   at 0x8100
    mov    b, #0xBB      ; at 8100h, the instruction is to
                         ; move BBh into the B register
    ljmp   0             ; return to the monitor.
    end
```

Assemble and single step through the program to verify that the two bytes, 81h and 0 are pushed on stack. Next, verify that the return instruction branches the program flow to address 8100h. Note, again, that pushing known constants to stack and

issuing a return instruction has the same effect as writing a long jump instruction. The former methods is flexible, however, since the values pushed on stack may be computed as a result of an operation. Then the program flow may be branched to a point that is computed depending on the data during program run time. Note that there are two absolute code segments defined. The first starts at 0x8000, and the other 0x8100. Also, note that each segment terminates with an "end" directive.

Some more sophisticated processors allow register indirect jumps. For example, the MCS-51 instruction

```
jmp   @a+dptr
```

was mentioned in Section 2.4.1. However, the MCS-51 instruction set does not include an indirect jump based on the registers r0 or r1. Below are two ways to implement a jump to an address that is specified by r1:r0.

```
mov   dph, r1
mov   dpl, r0
clr   a
jmp   @a+dptr
```

Alternatively,

```
mov   a, r1
push  acc
mov   a, r0
push  acc
ret
```

would accomplish the same branch. If the current register bank is known, the code may even be simplified. For instance, if register bank 0 is active, r0 and r1 are the same as internal data RAM 0 and 1.

```
push  1          ; r1 when register bank 0 is active
push  0          ; r0 when register bank 0 is active
ret
```

Note that the return instructions above do not have matching call instructions. Perhaps this is a good place to implement a macro with arguments. The Reads51 assembler supports C-style macro definitions. Consider the following macro.

```
#define REG_INDIRECT_JMP(HI,LO)  push HI \
                                 push LO \
                                 ret
```

The backslash characters at the end of the first two lines imply that the macro continues on the next line. The arguments HI and LO are simply formal variables, to be replaced with the specific arguments when the macro is invoked. We may now use the macro in a program.

```
cseg      at 0x8000    ; set the origin

    REG_INDIRECT_JMP(1,0)
    .
    .
    .
    end
```

The macro expands to

```
    push 1
    push 0
    ret
```

implementing the register-indirect jump.

2.4.4. Subroutines and Interrupt Service Routines

A subroutine is a portion of code terminated by a RET instruction. The subroutine is branched to by a CALL instruction. The CALL instruction pushes the current value of the program counter on stack, i.e., the address of the instruction immediately following the CALL instruction. The program counter is now loaded with the value given by the CALL instruction. For example, the `LCALL 0x8F2C` places `0x8F2C` into the program counter while the current value of the program counter is pushed onto stack. The next instruction to be executed is thus fetched from address `8F2C`, which should be the entry point of a subroutine. The last instruction of the subroutine, i.e., the RET instruction, pops two bytes, the value of the program counter pushed by the CALL instruction. This effectively is a branching instruction back to the instruction immediately following the CALL instruction. Notice that during the execution of the subroutine, care must be taken to use the stack in a balanced manner. Anything that is pushed on stack must be popped, or else, the RET instruction will place the wrong return address into the program counter and thus, the program will probably crash.

An Interrupt Service Routine (ISR) is a special type of subroutine. It is almost never called by an explicit CALL statement. The processor takes steps similar to those when executing a call statement. However, in this case, the call is triggered by a hardware signal. Such a trigger is called an interrupt and the routine invoked in this manner is called an ISR. The term interrupt implies that the processor preempts its regular task, which is sometimes called the foreground task. Note that the foreground task may be executing any one of its instructions when the ISR is invoked. In other words, since there is no explicit call to the ISR, it is not known exactly which

foreground instruction is to be interrupted. The 8051 has two external inputs P3.2 and P3.3, which, when so configured, act as interrupt (trigger) inputs. In addition, several of the 8051 internal peripherals are capable of generating interrupts. For example, an interrupt may be configured to fire when a timer overflows (increments from, say 0xFFFF, and rolls over to 0). Internal and external interrupt sources provide a rich mechanism to implement event-driven software architectures. In its purest form, the processor executes no foreground task. It simply configures the interrupts and runs an infinite loop doing nothing. Events simply trigger interrupts and the corresponding ISR performs the necessary tasks to handle the event. If the timers are used to generate periodic interrupts, then the software architecture takes on a so-called "real time" aspect. The term "real time" means that the processor will keep up with real time events. It will not miss any important changes in its environment. Real-time computing is possible when the timer interrupts occur more frequently than the fastest changing environmental variable. In this sense, real time is a relative concept. A real-time controller for slow changing physical phenomena, such as Heating, Ventilation, and Air Conditioning (HVAC) systems, could easily be several orders of magnitude slower than those for fast changing environments, say for fighter aircraft fly-by-wire control.

Observe the following example. The main program executes four CALL instructions, invoking two subroutines, SUB1 and SUB2.

```
; --- main program ----------
    .
    .
    .
    mov   a, #1
    lcall SUB1
    mov   r0, a
    lcall SUB2
    mov   a, #2
    lcall SUB1
    mov   r1, a
    lcall SUB2
    .
    .
    .
; --- end of main program ---

; --- subroutines -----------
SUB1:
    .
    (body of subroutine 1)
    .
    .
    ret
```

```
SUB2:
    .
    (body of subroutine 2)
    .
    .
    ret
; --- end of subroutines ----
```

Note that the use of subroutines will save program memory. Standard tasks that are repeated throughout the program are candidates for being implemented as subroutines. Structured programming is a paradigm that advocates the program to be broken down to logical units or sub-programs, each unit responsible for a well-defined task. Each subtask is then written as a subroutine. Structured programming improves code readability, facilitates teamwork in generating software, and simplifies future code revisions and modifications. Of course, a given program may be partitioned into subtasks in many ways. An initial thorough study and planning of the problem is essential to structured programming.

Parameter passing is a common issue in working with subroutines. More precisely, how should information be passed on to a subroutine, and how should the subroutine return information to the calling program. Many subroutines (e.g. the square root function) take numerical values and return numerical values. In the example above, the constant 1 is written to the accumulator before SUB1 is called. This is a good way to pass parameters to the subroutine, especially if the number of bytes to be passed on is low. A subroutine may similarly return values in the accumulator, b register, or other registers. Another approach is to push the parameters on the stack before calling the subroutine. The subroutine cannot simply pop the data off the stack, since the top of the stack is the return address. Nevertheless, there are obvious ways around this problem. Consider, for example, the case where we pass a single byte to the subroutine and the subroutine returns a single byte back.

```
    mov   a, #0x12      ; 0x12 sent to SUB3
    push  acc
    lcall SUB3
    pop   acc           ; result in acc
    .
    .

SUB3:
    pop   dpl           ; top two bytes on stack
    pop   dph           ; are the return address
                        ; dptr has the return address

    pop   acc           ; this is the value sent to SUB3
    .
```

```
    .
    .
; compute and place return value in acc
    .
    push   acc          ; return value
    push   dph          ; push return address on stack
    push   dpl
    ret
```

Parameters may be passed on to subroutines on internal stack. Most high-level languages construct and use a software stack implemented in external data memory. This eases the size restrictions on the internal stack. We will illustrate software stacks in the next chapter. Finally, values may be passed on to a subroutine in known memory addresses. This corresponds to setting the values of the global variables before the call, and reading them after the call. Here, the global variables correspond to the known (reserved) memory locations. This is also a common practice, especially used by older languages such as BASIC. The extensive use of global variables hinders modular programming practices. In addition, it becomes difficult to implement re-entrant subroutines when parameters are passed through global variables. Re-entrant routines are also presented in the next chapter.

Almost all of the applications of microcontrollers require that the controller interface with some physical device. Information, in terms of analog or digital signals, is received from the device. In turn, the microcontroller generates control signals that are sent back to the device to direct its operation. Real-time control refers to the requirement that the microcontroller must provide the control signals upon request, without any delays. The difference of a user-friendly menu driver, for example, and a DC motor speed regulator, exemplifies real-time requirements. Menu processing need not be real-time, since the user can wait a fraction of time without any adverse effects to the application. In motor control, however, the signals must be generated as quickly as possible to maintain the required speed regulation. Of course, in any system, there will be a delay due to the time it takes the microcontroller to process data. Real-time control should be interpreted as processing data without any avoidable delays. Ultimately, the capabilities of the microcontroller - primarily its speed - determine if it can fulfill the requirements of real-time control.

Interrupts are the most common way of handling real-time information processing. An interrupt is a signal given to the processor, which causes the processor to branch to a special subroutine, called the interrupt service routine (ISR). Typically, each interrupt source has its own ISR. Interrupt service routines differ from common subroutines in that they should terminate with RETI instructions rather than simple RET instructions. The RETI instruction resets the interrupt logic so that further interrupts are acknowledged by the processor.

The processor executes instructions of the main program and branches to the interrupt service routines as it receives the interrupt signals. Note that there are no explicit CALL instructions in MAIN. Moreover, the exact time the interrupt service routines are called cannot be deduced from inspecting MAIN. The exact time the program invokes the interrupt service routines can only observed at run time.

The interrupt service routines must preserve the registers and flags it modifies, since it may be invoked while the program keeps important information in the registers. Consider, for example, the following section of code from a program that regulates the speed of a motor. The accumulator holds the actual speed and the b register holds the reference value (desired speed). If actual speed is equal to the desired speed, then we jump to label DONE. Otherwise, we would like to call a subroutine, named FIX, to compute a correction signal for the motor speed controller.

```
subb   a, b          ; find error (difference)
jz     DONE          ; if no error jump to DONE
lcall FIX            ; else call subroutine FIX
```

Now suppose an interrupt is received just after the processor executes the SUBB operation. If the interrupt service routine uses the accumulator and modifies its value, upon return, the conditional jump instruction will inspect the modified value in the accumulator. The remedy is to push the accumulator in the interrupt service routine, and pop it before the RETI instruction. Similarly, most ISRs push PSW to preserve the flags of the foreground task.

The 8051 family of processors has five sources of interrupts, as listed in Section 1.2.2.4. Both external and internal interrupt sources are supported. For example, IE0 is the 0-th interrupt source activated when a signal is received, while TF0 is activated when the internal Timer 0 overflows. Interrupts may be prioritized or masked (disabled). The 8051 does not have a nonmaskable interrupt. Although activating the RESET pin always causes a jump to address 0, it is not a true interrupt source since the contents of PC are not pushed on stack.

Microprocessors are usually given many concurrent tasks. Consider the above example of regulating motor speed versus supporting a menu system. Typically, it would not be unreasonable to expect the processor to perform both tasks. The menu system can be implemented as the main task. An ISR can be written to service the motor speed control. The speed may be regulated at regular intervals, for example. One of the internal timers would be appropriate to periodically interrupt the processor and perform operations to regulate the speed of the motor. Alternatively, the motor may generate a signal if it detects that its speed has deviated from the required speed. The ISR may then be tied to an external interrupt receiving this error signal. In either case, as soon as the ISR completes the regulation task, the processor returns to serve the menu system. That is, speed regulation is done in real-time,

whereas menu servicing may be interrupted. Of course, in realistic applications, this interruption will be very short, most probably undetectable by the user. Many serious control applications consist entirely of ISRs. Such architectures are called *event-driven* since nothing happens unless an event (interrupt) is received.

2.5. Enhanced Members of the 8051 Family

Although the 8051 architecture is two decades old, it has maintained a large market share due to the introduction of many new enhanced members. The enhancements include faster processors, low power microcontrollers, on-board FLASH memory, and more on-chip peripherals. Many members of the family now have more RAM and ROM. One-Time-Programmable (OTP) members and those with EEPROM or FLASH memory are available. Other peripherals include extended interrupt processing, analog-to-digital converters, additional ports and timer units, inter-processor networking subsystems, additional data pointers, CAN units, I^2C bus, USB drivers, and arithmetic subsystems such as double precision multiply and divide units. The SFR of the 8051 architecture provide a unified way to access the additional on-chip peripherals of the enhanced members. This chapter illustrates some examples of such enhancements.

Three major bottlenecks of the 8051 microcontroller, namely, the single 16-bit data pointer, the absence of 32-bit divide and 16-bit multiply capabilities are remedied in the 80C517 manufactured by Infineon. The 80C517 has eight data pointers and an on-chip hardware multiply and divide unit. In addition, 21 high-speed outputs are implemented in the 80C517. These outputs are generated by the powerful 16-bit compare/capture/reload unit. These high-speed channels are convenient to generate pulse-width modulation (PWM) signals.

Both Infineon and Philips offer a full line of 8051 family of microcontrollers. Other manufacturers including Intel, Dallas Semiconductor, OKI, Standard Microsystems Corporation, and Atmel/Temic/Matra offer enhanced members of the 8051 family. At the time of the publication of this book, well over 100 variants of the 8051 family of microcontrollers were in production.

2.5.1. Fast Microcontrollers

Most manufacturers are now producing faster versions of the 8051. Some companies have increased speed by increasing clock frequency, some have increased speed by reducing the number of instruction cycles per clock, and some companies have done both. The two fastest 8051's advertised on the market today are by WiNEDGE Electronics and Dallas Semiconductor. Both companies have reduced instruction cycles to one per clock (a one-cycle core), and increased clock speed.

The ULTRA51™IC by WiNEDGE Electronics is a general-purpose RISC 805x-based microcontroller. On average, the execution cycle time is 8 times the performance of a

standard 805x for the same clock frequency. Each individual instruction executes up to 12 times faster than the original 805x. Additional features of the chip include full-featured program code, a large DATA ROM, and a large RAM storage. The CPU uses an efficient RISC (Reduced Instruction Set Computer) 805x compatible core. The core can potentially access 4Mbytes of program and data memory. This chip will be available for sampling at the end of 2000.

Dallas Semiconductor has been producing fast 8051's for many years. With the introduction of the DS89C420, which has the one-cycle core and top clock rate of 50 MHz, Dallas Semiconductor has hit a peak performance of 50 MIPS (million instructions per second). This speed currently exceeds any other 8051, and even many of the 16-bit processors on the market. The DS89C420 has maintained 100% instruction set and pin compatibility making it a drop in replacement for any standard 8051. The DS89C420's features also include 2 serial ports, and 16 Kbytes of in-system programmable FLASH. The FLASH supports the one-cycle processor operation and multiple modes of programming. This chip is available for sampling starting in October 2000 with full production scheduled for February 2001.

2.5.2. Enhanced Memory Options

There are many enhanced members of the 8051 family with RAM and ROM. All major manufacturers have ROM, EPROM, and FLASH versions of their controllers in various memory sizes. The EPROM versions that come in windowless plastic packages are also referred to as the One-Time-Programmable (OTP) devices. Unlike an EPROM, the microcontroller must be active with its clock running while the on-chip EPROM is being programmed. Several EPROM burner manufacturers offer adapters for the EPROM-ed versions of the 8051(see Appendix D). 8051's with FLASH are available in two versions. Most manufacturers carry the standard 8051 with FLASH ROM. Some manufacturers have an in-system programmable version, which allows greater flexibility in updating the code. The In-System Programmable (ISP) chips may be programmed after the microcontroller is connected to external circuitry. For example, you may build a product with an unprogrammed ISP 8051. Later, the product may be customized by programming the FLASH ROM with user-specific code and data. The product may later be reprogrammed without the need to remove the microcontroller from the circuitry. A good example is a cellular telephone, which is programmed with the assigned telephone number after its manufacture.

The DS5000 microcontroller manufactured by Dallas Semiconductor uses a lithium battery to supply its on-chip RAM. The battery-backed nonvolatile memory can be partitioned into ROM and RAM by programming the controller during its bootstrap cycle. The initialization of the DS5000 is administered by an on-chip system ROM that remains inaccessible to the user. The system ROM makes the DS5000 an intelligent microcontroller. The application program placed in the nonvolatile memory may be encrypted similar to the encryption of the EPROM devices. Once the security bit is set, the embedded program memory of the DS5000 cannot be accessed

externally. The DS5000 has several features that make it a robust microcontroller, especially suitable for harsh environments. It has a watchdog timer, which resets the microcontroller if not serviced periodically. Moreover, the DS5000 continuously measures the supply voltage (VCC). If VCC drops below acceptable limits, all memory contents are preserved and further memory access is disabled. These features make the DS5000 virtually crash proof. The DS5001 is a newer version with up to 128K bytes of battery-backed memory and with a real-time clock calendar.

The T89C51RD2 manufactured by Atmel Wireless & Microcontrollers (formerly Temic) has 64 Kbytes of in-circuit programmable FLASH, 2 Kbytes EEPROM, and 1280 bytes RAM. The processor can also operate at speeds of 30 MHz in the standard mode, or 60 MHz using a reduced number of instruction cycles (six) per clock.

The AT89S4D12 manufactured by Atmel is an 80C51 microcontroller with 128K bytes of in-system reprogrammable FLASH data memory and 4K bytes of downloadable FLASH program memory. The ROM is programmable while the chip is in the application circuit and electrically erasable. This makes the AT89C51 a versatile field-programmable controller.

2.5.3. Additional Data Pointers

The 80C517 microcontroller manufactured by Infineon has eight data pointers which all occupy the same SFR image 82h (DPL) and 83h (DPH). The specific data pointer among the eight to be active is selected by the SFR named DPSEL (Data Pointer SELect), which occupies address 92h. The instructions

```
movc   a, @a+dptr
movx   a, @dptr
movx   @dptr, a
```

use the active data pointer. Having many data pointers is convenient in defining data structures, such as arrays. Moreover, in multi-tasking applications, or where the application has many interrupt service routines, each task may be assigned a block of external memory. A data pointer may be dedicated to each task.

The program segment given below copies a block of memory to a new address in external memory. The memory blocks are assumed not to overlap. The source block is defined by its start and end addresses. More specifically, the bytes in the range `StartAddress` to `EndAddress-1`, inclusive are copied to the new location starting at address `TargetAddress`. The constants `StartAddress`, `EndAddress`, and `TargetAddress` should be defined by macros.

```
.
.
.
```

```
    mov   DPSEL, #0                  ; select dptr0
    mov   DPTR, #StartAddress        ; dptr0 points to the source
    mov   DPSEL, #1                  ; select dptr1
    mov   DPTR, #TargetAddress       ; dptr1 points to destination

XferLoop:
    mov   DPSEL, #0
    movx  a, @dptr
    inc   dptr
    mov   DPSEL, #1
    movx  @dptr, a
    inc   dptr

    mov   DPSEL, #0                  ; next check if done
    mov   a, dpl                     ; first check address low byte
    cjne  a, #low(EndAddress), XferLoop
    mov   a, dph                     ; then check address high byte
    cjne  a, #high(EndAddress), XferLoop
    .
    .
    .
```

Several other MCS-51 family members have two data pointers, e.g., Dallas Semiconductor and Infineon. The WinBond 8051's have an additional instruction that decrements the data pointer. Recall that the instruction set has an instruction to increment DPTR but lacks one to decrement it. The additional instruction uses the otherwise unused opcode 0xA5. This additional instruction may be supported simply by a macro definition, such as below.

```
#define DEC_DPTR db 0xA5
```

Note that the opcode 0xA5 is used by the Intel MCS-251 architecture as an "escape" instruction. In the so-called "binary mode," all advanced 8-bit and all 16-bit instructions of the 80C251 starts with the opcode 0xA5. In the so-called "source mode," the MCS-51 instructions begin with the opcode 0xA5. This scheme makes the MCS-251 binary code as well as instruction set compatible with the MCS-51.

2.5.4. The Multiply and Divide Unit of the 80C517

The 80C517 microcontroller has a fast hardware Multiply and Divide Unit (MDU) which performs a 32-bit by 16-bit division in 6 machine cycles and a 16-bit by 16-bit multiply in 4 machine cycles. In addition, the MDU performs shift and normalize operations. A normalize operation consists of an unknown number of shifts until the most significant bit is 1. Upon operation completion, the number of shifts performed is available in the SFR ARCON (ARithmetic CONtrol). Normalization operations are convenient in implementing floating-point arithmetic routines where the exponent is adjusted to make the most significant bit of the mantissa is always 1. The following

operations are performed by the MDU. The MDU is a good example of the flexibility and extensibility of the MCS-51 architecture.

Table 2.9. 80C517 MDU Operations

Operation	Result Precision	Remainder Precision	Execution Time in Machine Cycles *
division (32-bit ÷ 16-bit)	32 bits	16 bits	6
division (16-bit ÷ 16-bit)	16 bits	16 bits	4
multiplication (16-bit × 16-bit)	32 bits		4
32-bit normalize			6 or less
32-bit left or right shift			6 or less

* A machine cycle consists of 12 of the oscillator periods. With a 12MHz crystal, a machine cycle is 1 microsecond.

The operation of the MDU is initiated by writing the operand bytes into the data registers, the SFRs MD0 through MD5. The MDU then executes the operation in parallel with the CPU. The results are then read from the data registers, or in the case of a normalization operation, from ARCON. The specific operation to be performed is determined by the data registers involved and by the sequence in which the operands are written and read. The following table summarizes the multiplication and division operations.

Table 2.10. 80C517 MDU Multiply and Divide Operations

Operation	32-bit / 16-bit	16-bit / 16-bit	16-bit * 16-bit
first write	MD0 dividend low byte	MD0 dividend low byte	MD0 multiplicand low byte
	MD1 dividend	MD1 dividend high byte	MD4 multiplier low byte
	MD2 dividend	MD4 divisor low byte	MD1 multiplicand high byte
	MD3 dividend high byte	MD5 divisor high byte	MD5 multiplier high byte
	MD4 divisor low byte		
last write	MD5 divisor high byte		
first read	MD0 quotient low byte	MD0 quotient low byte	MD0 product low byte
	MD1 quotient	MD1 quotient high byte	MD4 product
	MD2 quotient	MD4 remainder low byte	MD1 product
	MD3 quotient high byte	MD5 remainder high byte	MD5 product high byte
	MD4 remainder low byte		
last read	MD5 remainder high byte		

For example, a 16-bit by 16-bit division is initiated by writing the four operand bytes to MD0, MD1, MD4, and MD5, in that order. The bytes are the dividend and divisor low and high bytes, respectively. After 4 machine cycles, the 4 result bytes are read. The quotient low and high bytes are read from MD0 and MD1, respectively. This is immediately followed by reading the remainder low and high bytes from MD4 and MD5.

The shift and normalize operations involve the SFR ARCON. The shift and normalize operations are performed on the 32-bit number stored in MD0, MD1, MD2, and MD3. MD0 holds the low byte, and MD3, the high byte. ARCON also includes two status bits: the error flag MDEF and the overflow flag MDOV. These are bits 7 and 6 of ARCON, respectively. The error flag is set if a wrong sequence of bytes is written to the data registers MD0 through MD5. Note that MDEF is always cleared by hardware after ARCON is read. MDOV is set if a division by zero is attempted or if a 16-bit by 16-bit multiplication exceeds 16 bits, even though the 32-bit result is computed and stored in the data registers.

Bit 5 of ARCON is the direction flag used with the 32-bit shift operation. The shift is performed to the right when SLR is set. The remaining 5 bits determine the number of shifts to be performed. The shift operation is initiated upon a write to ARCON, provided that the number of shifts selected is not zero. If the number of shifts is set to zero, the normalization operation is performed. The 32-bit number is shifted to the left until the most significant bit is 1. The number of shifts performed may then be read from ARCON. Note that although ARCON may be read at any time, a write to ARCON initiates a shift or normalize operation.

2.5.5. 10-Bit Analog-to-Digital Conversion

The 80C515A and the C515C are enhanced members of the 8051 family of microcontrollers manufactured by Infineon. They are upwardly compatible with the 80C515. They have 10-bit analog-to-digital converters. The two additional bits are evaluated during the conversion cycle. At the end of the conversion cycle, the higher 8 bits are available in the register ADDAT (at address D9h), just as in the 80C515. The lowest 2 bits are available in a new SFR ADDATL at address DAh. ADDATL is an abbreviation of Analog-to-Digital converter DATa Low byte. In the 80C515A, the SFR at D9h that holds the high byte is more named ADDATH.

The 80C515A and the C515C incorporate two other features beyond the 80C515. Firstly, a new control register is used with the analog-to-digital converter. ADCON at address D8h is renamed as ADCON0, and a new SFR, ADCON1 is implemented at address DCh. The most significant bit, named ADCL, of ADCON1 controls a prescaler. The 80C515 analog-to-digital converter clock input is the oscillator frequency divided by 8. The converter clock frequency needs to be 2 MHz or less. Since the 80C515A may be used at frequencies higher than 16 MHz, a divide-by-two prescaler is placed before the analog-to-digital converter clock input. When bit ADCL is set the prescaler is enabled. The other feature allows the analog-to-digital conversion to be initiated upon an external signal. The previously unused bit 5 of ADCON (ADCON0) determines the source of the conversion start signal. If this bit, named ADEX, is cleared, the conversion starts upon a write to DAPR, as in the 80C515. When ADEX is set, a high-to-low transition on Port 4.0 initiates the conversion. Initiating conversions upon external signals allows reading analog values in a synchronized manner in real-time control applications.

2.5.6. The I^2C Bus Support

Several microcontrollers including the 83C751, manufactured by Philips have all the hardware necessary to support the Inter-Integrated Circuit (I^2C) bus. The I^2C bus is a two-line multi-master / multi-slave network interface with collision detection. Each node on the bus may initiate a message, and then transmit or receive data. The two lines of the network consist of the serial data line, SDA, and the serial clock line (SCL). Each node on the network has a unique address, which accompanies any message passed between nodes. Note that besides microcontrollers, there are several peripheral devices such as intelligent display drivers, memory devices, and

analog-to-digital converters that support the I^2C bus. Interfacing a microcontroller to such peripherals is simplified when only two wires are required to interconnect the devices.

Two bits of Port P0, P0.0 and P0.1 of the 87C751 alternatively function as the SCL and SDA lines in I^2C support. The microcontroller has three additional SFRs to support the I^2C bus. The I^2C subsystem is configured by writing to the I^2C configuration register, I2CFG at location 0D8h. The I2CON register at location 98h is a control and status register. Writing to I2CON programs the I^2C bus operation, while reading from I^2C returns the status of the I^2C subsystem. Finally, the transmitted or received data is passed through the I2DAT register. Since serial communications are performed one bit at a time, only the most significant bit of I2DAT is used. The 87C751 uses an additional timer, Timer 1, to support the serial bus communications. Timer 1 is programmed and controlled by the control bits of the I2CFG register. Many other Philips microcontrollers support both bit and byte transfers over the I^2C bus.

2.5.7. ARCNET Token Bus Network Support

ARCNET is a powerful variable-message-length multi-node token ring network protocol. Arcnet is recognized by the American National Standards Institute (ANSI) as the ANSI 878.1 standard. The COM20051 manufactured by Standard Microsystems Corporation is an 8032 microcontroller with a complete on-chip ARCNET subsystem. The COM20051 comes in a 44-pin Plastic Leaded Chip Carrier (PLCC) package. The additional four pins are used for network support. The physical connections among the nodes may be twisted pair, coaxial, fiber optic, or RS-485 connections. Data transfer rates of up to 5 million bits per second are possible. Messages may be 1 to 508 bytes long.

Compared to the I^2C protocol, Arcnet is a relatively sophisticated system for microcontroller-based systems. The COM20051 Arcnet support subsystem is referred to as the Arcnet core. The Arcnet core contains a separate 1K byte RAM for transmitting and receiving message. Arcnet requires so many control and status registers that they cannot be fitted within the SFR space of the 8032. The COM20051 uses external data memory to accommodate the Arcnet registers. A page of 256 bytes of external data memory is put aside for the Arcnet core. The 256-byte page may be anywhere in the 64K memory space. The page address (the high byte of the base address of the page) is placed in external data memory location FFFFh. For example, if FFFFh contains 20h, the memory block from 2000h to 20FFh is reserved for the Arcnet core. Offsets into this block are the various control and status registers.

The detailed operation of the Arcnet driver is beyond the scope of this book. However, it is noteworthy that many of the transmit and receive operations, and bus arbitration when a new node is added or an existing one removed, are handled by the Arcnet core. The highly capable network subsystem considerably simplifies message-passing operations. The 1K buffer is partitioned into two pages of 512 bytes each.

This way, the microcontroller may write to or read from a page while transmitting or receiving the other. Before the COM20051 sends a message package, it is placed into one of the pages of the 1K Arcnet buffer in a byte-by-byte fashion through one of the Arcnet registers. Then, a command issued by setting the control bits of the Arcnet registers initiates the transfer of data.

The Arcnet core is not only a good example of how significantly sophisticated subsystems may be integrated with the 8051 family of microcontrollers, even when there are not enough registers in the SFR space, but also a good indication of how the two-decade-old architecture is kept popular by its enhanced members.

2.5.8. Controller Area Network (CAN) Bus Support

The Controller Area Network (CAN bus) is a serial communications bus for real-time control applications. CAN is an international standard and is documented in ISO 11898 (for high-speed applications) and ISO 11519 (for lower-speed applications). CAN was originally developed by Bosch GmbH originally to be used in the automotive industry. CAN operates at data rates of up to 1 Megabits per second and has excellent error detection and confinement capabilities. CAN allows multiple devices to communicate along a simple bus media while maintaining robust data integrity in high noise environments. The car industry continues to use CAN, but because of its proven reliability and robustness. In addition, CAN is now also being used in many other industrial control applications. In fact, higher protocol layers have been implemented over CAN. Most notably, DeviceNet and Can In Automation (CIA) are two popular such protocols used in industrial automation and control.

Data messages transmitted from any node on a CAN bus do not contain addresses of either the transmitting node or any intended receiving node. Instead, the content of the message is labeled by an identifier that is unique throughout the network. All other nodes on the network receive the message and each performs an acceptance test on the identifier to determine if the message is intended for that particular node. If the message is relevant, it will be processed; otherwise, it is ignored. The unique identifier also determines the priority of the message. The lower the numerical value of the identifier, the higher the priority. In situations where two or more nodes attempt to transmit at the same time, a non-destructive arbitration technique guarantees that messages are sent in order of priority and messages are not lost.

Several 8051 manufacturers have 8051 microcontrollers available with on-chip CAN. Infineon's C515C includes a full CAN module, 10-bit ADC, 64 KByte on-chip ROM/OTP memory, PWM capabilities, and a synchronous serial interface. The P8XC592 by Philips includes a full CAN module, 10-bit ADC, I^2C-bus, and ROM/OTP options. The DS80C390 manufactured by Dallas Semiconductor includes two full-function CAN bus controllers, a high-speed math coprocessor, enhanced memory architecture (addresses up to 4 MB external program memory), two serial ports, and three times the processing capability per clock cycle. The two CAN controllers can

service more devices and by communicating transparently with each other, provide back-up redundancy. The microcontroller can also serve as a bridge between two independent networks.

2.5.9. Universal Serial Bus (USB) Support

The Universal Serial Bus (USB) specification is a standardized peripheral connection developed by the PC and telecom industry leaders, Compaq, DEC, IBM, Intel, Microsoft, NEC and Northern Telecom. USB is designed to make adding new peripherals to a PC easy with plug-and-play, is nearly 100 times faster than the original PC serial port, and supports multiple device connectivity.

USB allows expandability of the PC's capabilities via an external port, eliminating the need for users to open the system chassis. Since USB supports multiple peripheral devices simultaneously, it allows users to run numerous devices (up to 127 devices) such as printers, scanners, digital cameras and speakers from a single PC. USB also allows for automatic device detection and installation, making connectivity a true plug-and-play experience for end users.

In keeping with the expansion of the 8051 into new markets, several manufacturers have chips that support USB. Cypress Semiconductor has two different series of chips, the AN2100 series, and CY7C64600 series with USB support. The Infineon C541U, and Standard Microsystems SMC97C100 and SMC97C102 also support USB.

2.5.10. System-On-Chip 8051's

Being perhaps the best-known 8-bit architecture, the MCS-51 is an ideal candidate to power System-On-Chip (SoC) products. SoC takes the idea of the microcontroller a step further: it includes not only the CPU, memory, and I/O but also application-specific hardware on the same chip.

The P-51 manufactured by Cybernetic Micro Systems contains all the hardware to directly interface to the PC ISA bus. This is the Industry Standard Architecture bus that is used by the standard adaptor cards of the IBM PC. Although most newer adaptor cards use higher speed busses, e.g. PCI, almost all PCs still have a few ISA slots. Moreover, industrial control PCs such as the PC-104 standard use the ISA bus. This makes the P-51 an ideal controller to power a PC-104 peripheral.

The E5 family from Trisend Corporation is based on the MCS-51 architecture. The E5 contains FPGAs (Field-Programmable Gate Array) on the same chip, which allows reprogrammability to the SoC processors. The E5 family is referred to a CSoC (Configurable SoC) processor. The user has the capability to assign different memory, I/O, or application-specific logic functions to the FPGA. This makes the E5 a highly flexible platform.

2.5.11. Migrating to 16-Bit Architectures

Intel and Philips both offer a 16-bit upgrade to the standard 8051 microcontroller. The upgrades include features especially tuned to support high-level languages. The Intel and Philips upgrades take different approaches.

The 80C251 is the first microcontroller of the MCS-51 architecture. It is pin-to-pin compatible to the standard 8051. Moreover, the MCS-251 instruction set is a superset of the MCS-51 instruction set. This means that the 80C251 executes the MCS-51 instructions without the need to recompile the source code. These features make the 80C251 a drop-in-replacement for the 8051. Besides the MCS-51 instructions, the MCS-251 instruction set includes many new 8, 16, and 32 bit instructions. The register based CPU supports a 40-byte register file. The MCS-51 architecture supports a 16Mbyte memory space. The 80C251 microcontroller uses a 256-Kbyte expanded external code/data memory space and 64-Kbyte-stack space. The new controller is also designed to execute C code efficiently.

The Philips XA also supports a 16Mbyte address space. It supports the bit-oriented operations of the 8051, but does not run existing 8051 code. Existing source code must be recompiled for the XA. The XA instruction set has support for high-level languages, especially C, and for multitasking operating systems. The XA is not pin-to-pin compatible with the 8051.

2.5.12. Information Sources

Due to the continued popularity of the 8051 family, many publications and magazines have been increasing their coverage of the related hardware, software, support tools, and services. Perhaps the best way to follow the developments in this area is to get on the web and visit the manufacturers web sites. Much information on products and services are available from the manufacturers. In addition, there are thousands of independent web sites and user forums, which support the 8051 with free application programs, circuit diagrams, and example source code. A list of related publications and the addresses of the manufacturers appear in Appendix D.

CHAPTER 3

ASSEMBLY LANGUAGE TECHNIQUES

3.1. Introduction

This chapter discusses software concepts and shows how instructions of the 8051 family of microcontrollers can be combined to write powerful subroutines. Many of the examples are from the monitor programs used on Rigel's evaluation boards. Table look up procedures, n-way branching, floating-point and signed, arithmetic operations, character and string manipulation techniques, and interrupt service routines are presented. The exposure is software oriented. The next chapter discusses how to use the on-chip facilities of the 8051 family of microcontrollers.

3.2. Data Block Transfer Routines

Often a block of external memory needs to be transferred to a different location. A memory block is uniquely described by its base address and its size. The base address refers to the address of the first byte, i.e., the byte with the lowest address. The block transfer may easily be accomplished by a loop of move instructions.

In many cases, the source and destination blocks will be known. The data pointer and r0 (or r1) may be used as pointers to the source and destination bytes. If the source and destination blocks do not overlap, it is more convenient to start with the base and proceed towards the top of each block, since the data pointer may be incremented but not decremented.

Unlike the data pointer, r0 (or r1) as a pointer cannot span the entire 64K data memory space. In this case, the high byte of the pointer may be placed into port P2 explicitly. When external data memory is accessed, port P2 emits the high byte of the address. Other times, P2 emits the value in its corresponding SFR. The instructions

```
movx   a, @ri
movx,  @ri, a
```

use the contents of r0 or r1 as the low byte of external data. These instructions may be used to access any external data memory location provided that the address high byte is placed in port P2. The following code transfers 2Fh bytes from location 280h to location FF00h. A similar loop is used in the monitor programs to transfer the interrupt vectors to high memory upon reset.

```
    mov    dptr, #0FF00h; set up destination pointer
    mov    P2, #2        ; set up source pointer high byte
    mov    r0, #80       ; set up source pointer low byte
    mov    r1, #2Fh      ; set up loop counter

transfer:
    movx   a, @r0        ; read source byte
    movx   @dptr, a      ; move to destination
    inc    r0            ; increment source pointer
    inc    dptr          ; increment destination pointer
    djnz   r1, transfer ; repeat 2Fh times
```

A general purpose data block transfer routine needs to determine whether to start from the first byte or the last byte of the memory block. If the source and destination blocks do not overlap, memory bytes may be moved in either direction, i.e., starting from the first or starting from the last byte. If the source and destination blocks overlap, then the block move must start with the last byte (the byte with the highest address) if the destination address is higher than the source address. Similarly, if the destination address is lower, the block move must start with the first byte (the byte with the lowest address).

3.3. Table Look Up Procedures

In a general sense, the microcontroller implements a function or procedure, which, given the input, produces an output. A table of constants is a convenient way to implement such a function. Tables are desirable especially if the analytical form of the function is unavailable. Even known analytical functions that require much CPU time to evaluate are often better implemented as tables (e.g. highly nonlinear functions).

The first example returns the first 16 prime numbers. Since there is no known formula to generate the n-th prime number, a list of numbers is a convenient way to program a microcontroller, especially if the first few primes are of interest.

```
; ================================================
; subroutine prime
; input    : n is the nibble in accumulator
; output   : the n-th prime number
; destroys: a
; ------------------------------------------------
prime:
    inc    a
    movc   a, @a+pc
    ret
    db     2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53
```

The entire subroutine is built around the accumulator-indexed data transfer instruction

```
movc   a, @a+pc.
```

As this move instruction is executed, the program counter is updated to point to the next instruction, which is the return instruction. Following the return instruction are 16 bytes of patterns, written as a block within the program memory using define byte pseudo operations.

The return instruction is a single-byte instruction, 22h. Thus, if the accumulator holds 0, the accumulator-indexed move instruction would place 22h in the accumulator. If the accumulator holds 1, the move instruction would place C0h, the pattern corresponding to digit 0, in the accumulator. That is to say, in order to evaluate the pattern for digit j, the accumulator should hold j+1 just before the indexed move instruction. This offset of 1 is due to the single-byte return instruction. In general, if there are n bytes of instructions between the indexed move instruction and the first entry of the table, then n must be added to the accumulator.

Adding an offset to the accumulator to compensate for the instructions between the indexed move instruction and the first entry of the table limits the size of the table. Notice that if there were n bytes of instructions, and n were added to the accumulator, the size of the table must be less than or equal to 100h-n. The following example uses a table to convert hexadecimal fractions 0 to 0.FFh to decimal fractions 0 to 0.99. At entry, the accumulator holds the 2-digit hexadecimal fraction. The 2-digit decimal fraction is returned in the accumulator in Binary-Coded Decimal (BCD) form. That is, each nibble of the accumulator is a decimal digit between 0 and 9.

Notice that the table contains 255 elements, rather than 256 elements. The case where the accumulator held FFh is treated separately. A test is performed to see if the accumulator is FFh. If so, the decimal fraction 99h (99BCD) is returned. Otherwise, the program branches to `anotff` (accumulator not FF), and the decimal fraction is obtained by table look up.

Although converting hexadecimal fractions to BCD fractions can be expressed by a simple analytical function, the table look up procedure requires no division operation. It is also faster, requiring 5 machine cycles for fractions 0 to 0.FEh, and 3 machine cycles for 0.FFh, excluding the call and return instructions to branch to the subroutine and return from it. The time savings is at the cost of 264 bytes of program memory, 255 bytes of which is data, rather than code.

```
; ================================================
; subroutine percent - converts [0-ff] to [0-99] bcd
; input    : byte in accumulator
; output   : [0-99] bcd in accumulator
; destroys: a, r0, dptr, carry flag
; ------------------------------------------------
```

```
percent:
   cjne   a, #0ffh, anotff
   mov    a, #99h
   ret
anotff:
   inc    a
   movc   a, @a+pc
   ret
; table of bcd corresponding to the byte
   db 00h, 00h, 01h, 01h, 02h, 02h, 02h, 03h   ;   0 -   7
   db 03h, 04h, 04h, 04h, 05h, 05h, 05h, 06h   ;   8 -   F
   db 06h, 07h, 07h, 07h, 08h, 08h, 09h, 09h   ; 10 - 17
   db 09h, 10h, 10h, 11h, 11h, 11h, 12h, 12h   ; 18 - 1F
   db 12h, 13h, 13h, 14h, 14h, 14h, 15h, 15h   ; 20 - 27
   db 16h, 16h, 16h, 17h, 17h, 18h, 18h, 18h   ; 28 - 2F
   db 19h, 19h, 20h, 20h, 20h, 21h, 21h, 21h   ; 30 - 37
   db 22h, 22h, 23h, 23h, 23h, 24h, 24h, 25h   ; 38 - 3F

   db 25h, 25h, 26h, 26h, 27h, 27h, 27h, 28h   ; 40 - 47
   db 28h, 29h, 29h, 29h, 30h, 30h, 30h, 31h   ; 48 - 4F
   db 31h, 32h, 32h, 32h, 33h, 33h, 34h, 34h   ; 50 - 57
   db 34h, 35h, 35h, 36h, 36h, 36h, 37h, 37h   ; 58 - 5F
   db 37h, 38h, 38h, 39h, 39h, 39h, 40h, 40h   ; 60 - 67
   db 41h, 41h, 41h, 42h, 42h, 43h, 43h, 43h   ; 68 - 6F
   db 44h, 44h, 45h, 45h, 45h, 46h, 46h, 46h   ; 70 - 77
   db 47h, 47h, 48h, 48h, 48h, 49h, 49h, 50h   ; 78 - 7F

   db 50h, 50h, 51h, 51h, 52h, 52h, 52h, 53h   ; 80 - 87
   db 53h, 54h, 54h, 54h, 55h, 55h, 55h, 56h   ; 88 - 8F
   db 56h, 57h, 57h, 57h, 58h, 58h, 59h, 59h   ; 90 - 97
   db 59h, 60h, 60h, 61h, 61h, 61h, 62h, 62h   ; 98 - 9F
   db 62h, 63h, 63h, 64h, 64h, 64h, 65h, 65h   ; A0 - A7
   db 66h, 66h, 66h, 67h, 67h, 68h, 68h, 68h   ; A8 - AF
   db 69h, 69h, 70h, 70h, 70h, 71h, 71h, 71h   ; B0 - B7
   db 72h, 72h, 73h, 73h, 73h, 74h, 74h, 75h   ; B8 - BF

   db 75h, 75h, 76h, 76h, 77h, 77h, 77h, 78h   ; C0 - C7
   db 78h, 79h, 79h, 79h, 80h, 80h, 80h, 81h   ; C8 - CF
   db 81h, 82h, 82h, 82h, 83h, 83h, 84h, 84h   ; D0 - D7
   db 84h, 85h, 85h, 86h, 86h, 86h, 87h, 87h   ; D8 - DF
   db 87h, 88h, 88h, 89h, 89h, 89h, 90h, 90h   ; E0 - E7
   db 91h, 91h, 91h, 92h, 92h, 93h, 93h, 93h   ; E8 - EF
   db 94h, 94h, 95h, 95h, 95h, 96h, 96h, 96h   ; F0 - F7
   db 97h, 97h, 98h, 98h, 98h, 99h, 99h        ; F8 - FE
```

3.4. ASCII Conversion Routines

Many applications require converting data expressed in one format into another. Most notably, input from the user may be received as text strings, from which numerical

values need to be extracted. The examples given below are adapted from Rigel's monitor program utilities.

The first example produces two ASCII characters, returned in the accumulator and in register R2, which respectively are the high and low digits of the (binary) value in the accumulator. The subroutine also illustrates the use of exchange and swap instructions.

```
;=========================================================
; subroutine binasc
; binasc takes the contents of the accumulator and
; converts it into two ascii characters (hex numbers).
; The result is returned in the accumulator (the high
; digit) and in register R2 (the low digit).
;
; input    : binary value in the accumulator
; output   : ASCII codes of the digits in acc and r2
; destroys: flags
;=========================================================
binasc:
    mov    r2, a          ; save number in r2
    anl    a, #0x0F       ; convert least significant digit.
    add    a, #0xF6       ; adjust it
    jnc    noadj1         ; if a-f then re-adjust
    add    a, #0x07
noadj1:
    add    a, #0x3A       ; make ascii
    xch    a, r2          ; put result in reg 2

    swap   a              ; convert most significant digit
    anl    a, #0x0F       ; look at least sig half of acc
    add    a, #0xF6       ; adjust it
    jnc    noadj2         ; if a-f then re-adjust
    add    a, #0x07
noadj2:
    add    a, #0x3A       ; make ascii
    ret
```

The program converts the low nibble and then repeats the procedure after swapping the nibbles. A mask byte 0Fh is used in masking off the high nibble with a logical AND operation. Next, a test is performed to see if the number is between 0 and 9, inclusively. If the number is between 0 and 9, subtracting 9 from the number would not cause an external borrow. Equivalently, adding F6h (which is 100h-Ah) to the number would not cause an external carry. The latter test is applied. Let the nibble be denoted by n. If n is between 0 and 9, and thus no carry is generated when F6h is added, in order to find the ASCII code, first n must be restored, and then the ASCII code of character '0' must be added. First adding F6h and then adding Ah to n is

equivalent to adding 100h to n, which leaves n in the accumulator and sets the carry flag. The carry flag is ignored when adding Ah. The ASCII codes for characters 0 to 9 are 30h to 39h, respectively. The steps for n between 0 and 9 are listed below.

step	operation	description
1	n	nibble value in the range 0 to 0Fh
2	n+F6h	check carry flag to determine the range of n if no carry, n was less than 0Ah, so continue
3	n+F6h+Ah=n+100h	adding 0Ah leaves n in the accumulator (with external carry)
4	n+F6h+Ah+30h=n+30h	adding the ASCII code for '0' gives the ASCII code of n

Of course, the two additions, Ah and 30h are combined, and 3Ah (0Ah+30h) is added to the result of n+0F6h, provided that the carry flag is not set during the addition n+0F6h.

If n is between 0Ah and 0Fh, adding 0F6h sets the carry flag. In this case, the ASCII code for the nibble is a character from 'A' to 'F'. The ASCII code for '9' is 39h, but unfortunately, the ASCII code for 'A' is not 3Ah. The ASCII code for 'A' is 41h. The difference between 41h and 3Ah is 7, which must also be added to the sum if n is in the interval 0Ah to 0Fh. The comments "re-adjust" refer to the addition of the offset 7.

Although the routine `binasc` gives the correct result, as well as provides us with an opportunity to hone our programming skills, it is probably more efficient to use a table look up procedure. The routine below returns the ASCII code of the low nibble of the byte given in the accumulator.

```
binascLookup:
    anl     a, #0x0F
    inc     a
    movc    a, @a+pc
    ret
    db      '0','1','2','3','4','5','6','7'
    db      '8','9','A','B','C','D','E','F'
```

The characters placed in single quotation marks are interpreted by the assembler as the ASCII code of the character. For instance, '5' is interpreted as 0x35, the ASCII code for the character (5).

The next example takes the ASCII code of a character either in the range '0' to '9', or in the range 'A' to 'F', or in the range 'a' to 'f', and returns the corresponding (binary) value in the low nibble of the accumulator. If the accumulator holds the ASCII code of a character that is not a valid hexadecimal digit, an error flag is set. The error flag is assumed to be one of the addressable bits defined by a macro or by the MCS-51 "bit"

directive. The code below may seem a bit cryptic. This is in part due to the way subtractions are carried out. The subroutine makes extensive use of replacing subtractions

```
accumulator - constant
```

operations by additions

```
accumulator + (100h-constant).
```

The program is now presented. A detailed explanation follows.

```
;=======================================================
; subroutine ascbin
; this routine takes the ASCII character passed to it in
; the accumulator and converts it to a 4-bit binary
; number which is returned in the accumulator.
;
; input    : ASCII code in the accumulator
; output   : binary value in the accumulator
; destroys: flags
;=======================================================
ascbin:
    add    a,  #0D0h     ; if chr < 30 then error
    jnc    notnum
    add    a,  #0F6h     ; check if chr is 0-9 adjust it
    jc     hextry        ; jmp if chr not 0-9
    add    a,  #0Ah      ; if it is then adjust it
    ret

hextry:
    clr    acc.5         ; convert to upper
    add    a,  #0F9h     ; check if chr is a-f adjust it
    jnc    notnum        ; if not a-f then error
    add    a,  #0FAh     ; see if char is 46h or less
    jc     notnum        ; if carry then not hex
    anl    a,  #0Fh      ; clear unused bits
    ret

notnum:
    setb   errorf        ; if not a valid digit
    ret
```

First, the ASCII code is tested to see if it is in the range 0 to 2Fh. If so, the character is not a valid hexadecimal digit. Typically, one would clear the carry flag and subtract 30h from the ASCII code, and inspect the carry flag. If the number is less than 30h, the carry flag is set to indicate an external borrow. Note that this test requires two instructions, one to clear the carry flag and the other to subtract 30h from the

accumulator. A slightly different approach is taken in the subroutine. Instead of subtracting 30h from the accumulator, 0D0h is added. Note that 0D0h=100h-30h. If the accumulator holds a value less than 30h, this addition will not produce an external carry. Thus, the range check is accomplished with only one instruction. In the subroutine, if the accumulator holds a number less than 30h, the program is directed to label `notnum` (not a number). Subsequently, an error flag is set and the subroutine returns to the calling program.

Next, the ASCII code is tested to see if it is less than 3Ah. Since the constant 0D0h is already added to the ASCII code, the test should check if the accumulator holds a value less than 0Ah. Again, equivalently, adding 0F6h (which is 100h-0Ah) to the accumulator will produce an external carry only if the accumulator holds a number Ah or greater. If an external carry is generated, the program branches to label `hextry` to see if the ASCII code is that of a character in the interval 'A' to 'F' or 'a' to 'f'. Else, if the carry flag is not set, the ASCII code is that of a character in the interval '0' to '9'. The constant 0Ah must be added to restore the correct value before the subroutine returns.

The portion of code starting with label `hextry` first clears the fifth bit of the accumulator to convert any lower case ASCII character to upper case. Note that the ASCII codes of capital letters start with 41h, and of lower case letters start with 61h. Since the ASCII code 41h for 'A' is 7 more than the ASCII code 39h for '9', the offset of 7 must be subtracted from the accumulator. Equivalently, one may add 0F9h (which is 100h-7) to the accumulator. At this point, only if the original ASCII code, with its fifth bit cleared, was not between '9' and 'A', exclusively, an external carry is generated. If no external carry is generated, the program branches to `notnum` to set the error flag and return.

If the program still has not returned, the original ASCII code, with its fifth bit cleared, corresponds to the character 'A' or above. A last test determines if the character is 'F' or below. This test subtracts 6 or equivalently adds 0FAh to the accumulator. If the carry flag is not set, then the original ASCII code was in the interval 'A' to 'F' (inclusive). The following show how the character 'B' (ASCII code 42h) is processed by the subroutine.

step	operation	accumulator	carry flag	branch
1		42h		
2	add D0h	12h	set	no - next instruction
3	add F6h	8	set	yes - to hextry
4	clear bit 5	8		
5	add F9h	1	set	no - next instruction
6	add FAh	FBh	cleared	no - next instruction
7	and 0Fh	Bh		

3.5. Jump Tables

A jump table is a list of jump instructions. The interrupt vectors of the 8051 is an example of a jump table. A jump table is a good way to fix the entry point of routines. It is especially useful when you have multiple programs which were not linked at compilation, but nevertheless need to call functions of each other.

Suppose you want to place a number of utility routines in ROM. Once the routines are programmed in ROM, their addresses cannot be changed. Suppose we had the two routines `_delay` and `_decDptr` in ROM. The first routine uses two no-operation instructions to implement a very short delay. The second decrements the data pointer by adding the constant 0xFFFF to it. Suppose that the routine `_delay` occupies ROM address 0x20. Since it is six bytes long, `_decDptr` starts at address 0x26.

```
; --- starts at ROM address 0x20
_nops:
    nop
    nop
    nop
    nop
    nop
    ret

_decDptr:
    push   acc
    mov    a, dpl
    add    a, #0xFF
    mov    dpl, a
    mov    a, dph
    addc   a, #0xFF
    mov    dph, a
    pop    acc
    ret
```

Now, suppose several another routines, including `_compute` are separately assembled and placed in the same ROM. If these routines need to call `_decDptr`, a simple solution would be to directly use its address.

```
_compute:
    .
    lcall 0x26
    .
    lcall 0x26
    .
    .
    ret
```

This arrangement works, but it becomes exceedingly unwieldy when code needs to be modified. Suppose, for example, that the delay implemented by the first routine needs to be increased by adding another no-operation instruction. The routine `_decDptr` now starts at 0x27. This requires that all calls to 0x26 be replaced by calls to 0x27. Of course, you may define a symbol, rather than to use the numerical value of the routine address. That is,

```
#define DECDPTR 0x26
```

and make the change only in one place. But still, modifying one program affects the other so much that it also has to be reassembled and programmed into ROM. The two separate programs are thus not so independent. In practice, the need for code independence is so important that it often is the main criterion in code development. A jump table is a low-tech way to improve the independence. First we make a list of jumps as follows.

```
; --- starts at ROM address 0x20

    ljmp  _nops
    ljmp  _decDptr
    .
    .              ; jumps to other routines
    .

_nops:
    nop
    nop
    ret

_decDptr:
    push  acc
    mov   a, dpl
    add   a, #0xFF
    mov   dpl, a
    mov   a, dph
    addc  a, #0xFF
    mov   dph, a
    pop   acc
```

```
    ret
```

Instead of a direct call to _decDptr, we may now call the jump instruction in the table. The jump instruction to _decDptr is the second long jmp instruction. Each long jump instruction takes three bytes. So, the jump table entry for _decDptr is at 0x23.

```
#define DECDPTR 0x23

_compute:
    .
    lcall DECDPTR
    .
    lcall DECDPTR
    .
    .
    ret
```

The long call to the table entry, i.e. 0x23, is directed to the routine _decDptr. When routines preceding _decDptr are modified, the ROM address of _decDptr will change. But this is now reflected in the long jump instruction in the jump table. So other programs may continue to call the routine _decDptr by

```
    lcall 0x23
```

or

```
    lcall DECDPTR
```

This provides a higher level of independence. Jump tables were more common in the early days of computing. Contemporary approaches rely on relative assembly and linking, rather than the "hand linking" of the modules. After all, the main task of the linker is to resolve the addresses among the various relative assembly modules.

The RROS EPROM used on Rigel boards maintain a jump table for the utility routines. You may access these routines, provided that you keep the EPROM in low memory. The code in RROS is as below.

```
    cseg    at 100h
;           routine     ljmp instruction address
;                       (entry point)
;           -------     ------------------------
    ljmp    ascbin      ; 0100h
    ljmp    autoexec    ; 0103h
    ljmp    beep        ; 0106h
    ljmp    binasc      ; 0109h
    ljmp    break       ; 010Ch
    ljmp    chkbrk      ; 010Fh
```

```
        ljmp    cret            ; 0112h
        ljmp    crlf            ; 0115h
        ljmp    delay           ; 0118h
        ljmp    display         ; 011Bh
        ljmp    getbyt          ; 011Eh
        ljmp    getchr          ; 0121h
        ljmp    getchrx         ; 0124h
        ljmp    init            ; 0127h
        ljmp    inkey           ; 012Ah
        ljmp    mdelay          ; 012Dh
        ljmp    mon_return      ; 0130h
        ljmp    percent         ; 0133h
        ljmp    print           ; 0136h
        ljmp    prsphx          ; 0139h
        ljmp    prtstr          ; 013Ch
        ljmp    prthex          ; 013Fh
        ljmp    sdelay          ; 0142h
        ljmp    setintvec       ; 0145h
        ljmp    sndchr          ; 0148h

; --- body of the routines -----------------
ascbin:
    .
    {instructions...}
    .
    ret
autoexec:
    .
    {instructions...}
    .
    ret
    .
    .
    .
    end
```

Suppose a call to, say, address 136h is made. The instruction found at address 136h is a long jump to the label `print`. The program then continues executing the utility routine `print`. The utility routines terminate with `ret` instructions, which pass the control back to the calling program. This arrangement has the advantage that the utility routines have standard entry points at 100h, 103h, 106h, ..., and so on, irrespective of the size and location of the actual routines. If new versions of the routines are compiled at a later date, or new routines are added to the list, the entry points do not change. This means that earlier application programs written to access the utility routines need not be changed whenever the routines are revised.

Provided that the EPROM is in low memory, the following header may be included in the application program to give access to the most commonly used system calls.

```
; ---------------------------------------------
; system calls              registers used
; ---------------------------------------------
ascbin      equ  0100h  ; a, r2, error flag
autoexec    equ  0103h
beep        equ  0106h  ;  none
binasc      equ  0109h  ; a
break       equ  010ch  ; a, (reads accumulator)
chkbrk      equ  010fh  ; a, (reads serial port)
cret        equ  0112h  ; a
crlf        equ  0115h  ; a
delay       equ  0118h  ; a
display     equ  011bh  ; a
getbyt      equ  011eh  ; a, b
getchr      equ  0121h  ; a
getchrx     equ  0124h  ; a
init        equ  0127h
inkey       equ  012ah  ; a
mdelay      equ  012dh  ; a
mon_return  equ  0130h
percent     equ  0133h  ; a
print       equ  0136h  ; a, dptr
prsphx      equ  0139h  ; a, r2.
prtstr      equ  013ch  ; a
prthex      equ  013fh  ; a, r2
sdelay      equ  0142h  ; a
setintvec   equ  0145h  ; a, dptr
sndchr      equ  0148h  ; a
```

You may also use the more modern "#define" macros rather than the "equ" pseudo-ops.

3.6. N-Way Branching

A powerful use of the return instruction is the implementation of an n-way branch. This operation may be considered as an accumulator-indexed jump, or as a computed goto operation. The return instruction simply pops two bytes off the stack and places them into the program counter. This value in the program counter is the address of the next instruction to be fetched. If an address is pushed onto the stack by any means and a return instruction is invoked, the program branches to the address pushed on stack.

The following subroutine is adopted from Rigel's monitor program. The accumulator is assumed to hold a number from 0 to 26h. This number is the ASCII code of a single-letter monitor command minus 40h. Note that the ASCII code for 'A' is 41h,

and for 'Z' is 5Ah. The at symbol '@' has the ASCII code '40h'. The subroutine uses the accumulator value to branch to one of 27 possible monitor subroutines, each terminating with a return instruction. Subroutine nway is called from the command line parser. So when nway starts executing, the top of the stack holds the return address to the parser. Since nway branches to one of the 27 possible monitor subroutines without a call (without pushing a return address back to nway), when the monitor subroutine terminates, the program flow returns directly to the parser.

```
;=======================================================
; subroutine nway
; this routine branches (jumps) to the appropriate
; monitor routine. routine number is in r2
;
; input   : monitor routine number 0 to 1Ah in the accumulator
; output  : none
; destroys: dptr, r2
;=======================================================
nway:
   mov   r2, a          ; save monitor subroutine number in r2
   mov   dptr,#jumtab   ; point data pointer at beginning of
                        ; jump table.
   rl    a              ; multiply accumulator by two
   inc   a              ; put the offset of the table entry
                        ; holding the high byte of the
                        ; monitor routine address in the
                        ; accumulator
   movc  a, @a+dptr     ; read monitor routine address high byte
   push  acc            ; push monitor routine address high byte
   mov   a, r2          ; load accumulator with monitor subroutine
                        ; number
   rl    a              ; the accumulator holds the offset of the
                        ; table entry holding the low byte of
                        ; the monitor routine address
   movc  a, @a+dptr     ; read monitor routine address low byte
   push  acc            ; push monitor routine address high byte
   ret                  ; jump to the monitor routine
;=============================================================
; monitor jump table
;
; Each entry is a two-byte address.  The constants hstcmd,
; badcmd, etc. are labels or subroutine entry points.
;=============================================================
jumtab:
   dw hstcmd
   dw badcmd            ; command '@' 00    used
   dw pbreak            ; command 'a' 01
   dw codmem            ; command 'b' 02    used
                        ; command 'c' 03    used
```

```
dw datmem          ; command 'd' 04    used
dw badcmd          ; command 'e' 05
dw badcmd          ; command 'f' 06
dw goaddr          ; command 'g' 07    used
dw help            ; command 'h' 08    used
dw badcmd          ; command 'i' 09
dw badcmd          ; command 'j' 0a
dw breset          ; command 'k' 0b    used
dw dlloop          ; command 'l' 0c    used
dw badcmd          ; command 'm' 0d
dw badcmd          ; command 'n' 0e
dw badcmd          ; command 'o' 0f
dw setprt          ; command 'p' 10    used
dw badcmd          ; command 'q' 11
dw shwreg          ; command 'r' 12    used
dw setsfr          ; command 's' 13    used
dw badcmd          ; command 't' 14
dw badcmd          ; command 'u' 15
dw badcmd          ; command 'v' 16
dw badcmd          ; command 'w' 17
dw extmem          ; command 'x' 18    used
dw badcmd          ; command 'y' 19
dw badcmd          ; command 'z' 1a
```

Each table entry is a two-byte address. These addresses are the entry points of the monitor routines. The monitor routines are not given above. Note that with this arrangement up to 128 labels may be placed in the table.

3.7. Signed Arithmetic Routines

Two's complement representation is a convenient form to store signed integers. Bytes in two's complement notation have the range -128 to 127, and two-byte words have the range -32768 to 32767. Sometimes it is preferred to store signed integers by their absolute value and a separate signed bit. Specifically, signed multiplication and division operations are simpler with the latter format. The addressable bits of the 8051 family of microcontrollers facilitate the working with sign bits.

The routines use directly addressable internal registers (in the range 0 to 7Fh). The registers are used in pairs to hold 16-bit values. Four general-purpose 32-bit operands, referred to as X, Y, Z, and PQ, are used. Each operand may be up to 32-bits in size. The registers _X3, _X2, _X1, and _X0 are associated with X. For example, if X is a 16-bit value, the register pair _X1:_X0 is used. Here, _X0 is the low byte, and _X1, the high byte. The registers for the other operands are similarly defined.

```
MathRegs segment data

        rseg MathRegs
_X0:    ds  1
_X1:    ds  1
_X2:    ds  1
_X3:    ds  1
_Y0:    ds  1
_Y1:    ds  1
_Y2:    ds  1
_Y3:    ds  1
_Z0:    ds  1
_Z1:    ds  1
_Z2:    ds  1
_Z3:    ds  1
_ZEXP:  ds  1
_PQ0:   ds  1
_PQ1:   ds  1
_PQ2:   ds  1
_PQ3:   ds  1
        end
```

Similarly, we define some bit variables. Associated with each of the general-purpose operands X, Y, and Z, we define a sign bit and an overflow bit.

```
MathBits segment bit

        rseg MathBits
_XS:    dbit 1
_XOV:   dbit 1
_YS:    dbit 1
_YOV:   dbit 1
_ZS:    dbit 1
_ZOV:   dbit 1
_ZADJ:  dbit 1
        end
```

The register RQ holds the partial quotient during a 32-bit by 16-bit division. Several signed and unsigned arithmetic routines are given below. Note that these routines are not optimized for speed or size. Rather, the routines are for easier understandability.

3.7.1. Two's Complement Conversion Routines

First consider single-byte signed integers. The conversion between two's complement to absolute-value/sign-bit formats is illustrated by the following subroutines.

The routine TC2MS8 converts the two's complement signed byte in _X0 to the magnitude/sign bit (M/S) firmat and places it in _Y0.

The bit XS shows the sign of _Y0, XS=1 if _X0<0.

```
TC2MS8:
    mov    a, _X0        ; read_X into accumulator
    jb     acc.7, Xneg   ; _X0 is negative if bit 7 is 1
    mov    _Y0, _X0      ; else,_X0 is the absolute value
    clr    _YS           ; clear sign bit for 'positive'
    ret                  ; done
Xneg:
    setb   _YS           ; set sign bit for 'negative'
    cjne   a, #80h, Not80h
    mov    _Y0, a
Not80h:
    cpl    a             ; _X is negative so find absolute value
    inc    a             ; _Y0 = complement(X0)+1
    mov    _Y0, a        ; save absolute value
    ret
```

If X is positive, it is equal to its magnitude. It is simply copied into Y, and the sign bit cleared. If X is negative, its magnitude is computed by finding the two's complement of X. The sign bit is set. Note that X= -128 is a special case. Here, the magnitude of X exceeds the range of two's complement bytes. That is, +128 is not in the range of two's complement bytes. Hence, unlike other negative numbers, in this case, the magnitude is simply set to 128 and the sign bit set.

The inverse operation is performed by routine MS2TC8. The byte in _X0 is assumed to be in the magnitude/sign bit format. Note that the 8-bit magnitude and the single bit sign acts as a 9-bit number, whose range is –255 to +255. Thus, if the number exceeds 127 or is less than –128, it cannot be written in the two's complement format. In this case, the overflow flag _YOV is set.

```
MS2TC8:
    mov    a, _X0              ; get absolute value
    jnb    acc.7, range_OK     ; see if MSB is 0
    cjne   a, #80h, range_error
    jnb    _XS, range_error
    clr    _YOV
    mov    _Y0, a
    ret
range_error:
    setb   _YOV          ; set overflow bit
    ret
range_OK:
    clr    _YOV          ; no overflow error
    jb     _XS, Xneg1    ; check sign
    mov    _Y0, _X0      ; if positive,_Y0 = _X0
    ret                  ; done
```

```
Xneg1:
   mov   a, _X0        ; if negative, get absolute value
   cpl   a             ; complement absolute value of_X
   inc   a             ;_XT = complement(XA)+1
   mov   _Y0, a        ; two's complement in _Y0
   ret
```

Again, the case –128 is handled separately.

The 16-bit counterparts of these conversion routines are given next. In routine TC2MS16 the internal registers _X1:_T0 hold the high and low byte of the 16-bit number in two's complement representation. The 16-bit number is converted to the magnitude/sign bit format and stored in _Y1:_Y0.

```
TC2MS16:
   mov   a, _X1        ; read X high byte into accumulator
   jb    acc.7, Xneg2  ; X is negative if bit 7 is 1
   mov   _Y1, _X1      ; else, X is the absolute value
   mov   _Y0, _X0
   clr   _YS           ; clear sign bit for 'positive'
   ret                 ; done
Xneg2:
   setb  _YS
   mov   a, _X0        ; special case (0x8000)
   jnz   TC16
   mov   a, _X1
   cjne  a, #0x80, TC16
   mov   _Y1, a
   mov   _Y0, #0
   ret
TC16:                  ; complement _X1:_X0, place in _Y1:_Y0
   mov   a, _X0        ; X is negative
   cpl   a             ; find its absolute value
   inc   a             ; Y0 = complement(X0)+1
   mov   _Y0, a        ; Y0 is found
   jnz   Y0_OK         ; if not 0, high byte is not incremented
   mov   a, _X1        ; else get high byte
   cpl   a             ; complement high byte...
   inc   a             ; ... and increment
   mov   _Y1, a        ; store in _Y1
   ret                 ; done
Y0_OK:
   mov   a, _X1        ; get high byte
   cpl   a             ; complement high byte - don't increment
   mov   _Y1, a        ; store in _Y1
   ret                 ; done
```

The routine MS2TC16 performs the inverse operation.

```
MS2TC16:
    mov     a, _X1              ; get absolute value high byte
    jnb     acc.7, range_OK1    ; see if MSB is 0
    clr     acc.7               ; see if _X=-8000h
    orl     a, _X0              ; acc=0 only if_X=8000
    jnz     range_error1        ; else error
    jb      _XS, range_OK1      ; also error if_X is positive
range_error1:
    setb    _XOV                ; set overflow bit
    ret
range_OK1:
    clr     _YOV          ; no overflow error
    jb      _XS, TC16     ; if negative, do two's complement
    mov     _Y1,_X1       ; if positive,_Y1:_Y0 = _X1:_X0
    mov     _Y0,_X0
    ret                   ; done
```

Once again, that 0x8000 (–32678) is a special case that is handled separately in both of the above routines. The routines may be further streamlined, since several instructions are duplicated in both branches after testing whether the incrementing the low byte causes a carry into the high byte. These routines are used in the addition and multiplication routines discussed below.

3.7.2. 8-Bit Signed Addition and Subtraction

First consider two 8-bit signed numbers X and Y, and their sum Z=X+Y. If X and Y are in two's complement notation, their sum may be found by the ADD instruction. The overflow flag OV is set if the sum of two positive numbers result in a negative number or if the sum of two negative numbers produce a positive number. If either X, or Y, or both are in the absolute-value / sign-bit format, the numbers may first be converted to two's complement notation using the routines given above.

The difference of two 8-bit signed numbers Z=X-Y is also accomplished by the SUBB instruction if X and Y are in two's complement format. Again the overflow flag OV is set if the result is out of range. For example, the operation (10) - (-120) gives 130, which is greater than the upper limit 127 of single-byte signed integers, and thus, OV is set.

3.7.3. 8-Bit Signed Multiplication and Division

It is easier to multiply and divide signed bytes if they are in the absolute-value / sign-bit format. The sign of the result is determined from the sign of the operands. The magnitude of the result is computed by multiplying or dividing the magnitudes of the operands. If bytes are in two's complement notation, they are first converted into the absolute-value / sign-bit format using the routines discussed above.

Let _X0, and _Y0 be the absolute values of the bytes X and Y. Let addressable bits XS and YS hold the signs. The byte is negative if the corresponding sign bit is set.

First consider the multiplication Z=(X)×(Y). Let _Z1 and _Z0 be the high and low bytes of the absolute value of the result Z. Similarly, let the addressable bit ZS be the sign bit of the result, set if the result is negative.

```
MSMUL8:
    mov    a, _X0          ; read_X and ...
    mov    b, _Y0          ; ..._Y
    mul    ab              ; multiply_X and_Y
    mov    _Z1, b          ; save result high ...
    mov    _Z0, a          ; ... and low byte
    lcall  getZS           ; get sign of_Z
    ret

getZS:
    jb     _XS, gzs_00     ; _ZS = _XS EXOR _YS
    jb     _YS, gzs_set
gzs_clr:
    clr    _ZS
    ret
gzs_set:
    setb   _ZS
    ret
gzs_00:
    jb     _YS, gzs_clr
    sjmp   gzs_set
```

The routine simply multiplies the operands as unsigned bytes. The sign bit of the result is the exclusive or of the sign bits of the operands. Since the MCS-51 instruction set does not include a bit-oriented EXOR operation, the sign bit is evaluated by checking the sign bits of the operands and branching to different parts of the routine `getZS`.

The signed division of bytes in the magnitude/sign bit format is also simplified by the use of the MCS-51 division instruction. The sign of the result is obtained as in the multiplication case.

```
MSDIV8:
    mov    a,_X0           ; read_X and ...
    mov    b,_Y0           ; ..._Y
    div    ab              ; divide_X and_Y
    mov    C, OV           ; get overflow flag
    mov    _ZOV, C         ; save in_ZOV.  (set if_Y=0)
    mov    _Z1, a          ; save result quotient
    mov    _Z0, b          ; save remainder
    lcall  getZS           ; get sign of_Z
    ret
```

3.8. 16-Bit Unsigned Arithmetic Routines

3.8.1. 16-Bit Addition and Subtraction

The 16-bit integer addition and subtraction operations are straight forward extensions of their 8-bit counterparts. The 16-bit operations are initiated with the low bytes. Any carry or borrow from the low-byte operation is carried to the high-byte operation. The carry flag CY is set during the high-byte operation if the result is out of range.

The following signed addition and subtraction routines assume that the internal registers _X1:_X0, _Y1:_Y0, and _Z1:_Z0 hold the high and low bytes of 16-bit integers X, Y, and the result Z. The addressable bit ZOV is set if there is an overflow, i.e., if the result Z is out of range.

```
UADD16:
    mov     a, _X0          ; load X low byte into accumulator
    add     a, _Y0          ; add Y low byte
    mov     _Z0, a          ; put result in Z low byte
    mov     a, _X1          ; load X high byte into accumulator
    addc    a, _Y1          ; add Y high byte and the carry from
                            ; low byte operation
    mov     _Z1, a          ; save result in Z high byte
    mov     _ZOV, C         ; set _ZOV if external carry
    ret

USUB16:
    clr     C               ; clear carry flag
    mov     a, _X0          ; load _X low byte into accumulator
    subb    a, _Y0          ; subtract _Y low byte
    mov     _Z0, a          ; put result in _Z low byte
    mov     a, _X1          ; load _X high byte into accumulator
    subb    a, _Y1          ; subtract _Y high byte with the borrow
                            ; from low byte operation
    mov     _Z1, a          ; save result in _Z high byte
    mov     _ZOV, C         ; set _ZOV if an external borrow is
                            ; produced
    ret
```

These routines may easily be extended to 3- or 4-byte addition and subtraction operations. The following subroutine illustrates the subtraction of 32-bit unsigned integers. It is a straight forward extension of subroutine USUB16.

```
USUB32:
    clr     C               ; clear carry flag
    mov     a, _X0          ; load X low byte into accumulator
    subb    a, _Y0          ; subract Y low byte
    mov     _Z0, a          ; put result in Z low byte
    mov     a, _X1          ; repeat with other bytes...
```

```
    subb   a,_Y1
    mov    _Z1, a
    mov    a, _X2
    subb   a, _Y2
    mov    _Z2, a
    mov    a, _X3
    subb   a, _Y3
    mov    _Z3, a
    mov    _ZOV, C        ; set _ZOV if an external borrow
                          ; is produced
    ret
```

3.8.2. 16-Bit Unsigned Multiplication and Division

The multiplication of two 16-bit integers is accomplished by two multiplication operations, each of an 8-bit integer with a 16-bit integer, followed by the addition of the two results. First consider the multiplication of an 8-bit integer with a 16-bit integer. The product of an 8-bit integer and a 16-bit integer is a 24-bit integer, which is stored in three bytes. Note that there is no overflow. The result of an unsigned 8-by 16-bit multiplication always fits in three bytes.

```
UMUL8_16:
    mov    a, _X0         ; load X into accumulator
    mov    b, _Y0         ; load Y low byte into b register
    mul    ab             ; multiply
    mov    _Z0, a         ; save result low byte
    mov    r0, b          ; save result high byte in r0
                          ; this byte needs to be added to the
                          ; low byte of the product _X0*_Y1
    mov    a, _X0         ; load X into accumulator
    mov    b, _Y1         ; load Y high byte into b register
    mul    ab             ; multiply
    add    a, r0          ; _X0*_Y0 high byte + _X0*_Y1 low byte
    mov    _Z1, a         ; save result
    clr    a              ; clear accumulator
    addc   a, b           ; a = b + carry flag
    mov    _Z2, a         ; save result
    ret                   ; done
```

The subroutine UMUL16 multiplies two 16-bit words. It uses the subroutine UMUL8_16 to multiply _X0, the low byte of X, by the 16-bit number _Y1:_Y0. The 3-byte result is saved in _Z2:_Z1:_Z0. These three bytes are pushed on stack. Then, _X1 is moved into _X0 and the routine UMUL8_16 is called again. The result of this multiplication needs to be multiplied by 256 and added to the result of the previous multiplication. Multiplying by 256 is the same as shifting to the left 8 bitsSimply moving the result from _Z2:_Z1:_Z0 to _Z3:_Z2:_Z1 has the same effect. The result of the previous multiplication may now be added.

```
UMUL16:
    lcall  UMUL8_16      ; multiply _X0 with (_Y1:_Y0)
    push   _Z2
    push   _Z1
    push   _Z0
    mov    _X0, _X1      ; put _X1 in _X0
    lcall  UMUL8_16      ; multiply _X1 with (_Y1:_Y0)
    mov    _Z3, _Z2
    mov    _Z2, _Z1
    mov    _Z1, _Z0

    pop    _Z0           ; result from previous 8 by 16 mul

    pop    acc           ; previous _Z1
    add    a, _Z1
    mov    _Z1, a

    pop    acc           ; previous _Z2
    addc   a, _Z2
    mov    _Z2, a

    clr    a
    addc   a, _Z3
    mov    _Z3, a

    ret
```

The external carry from the first addition (a+_Z1) is carried to the second addition (a+_Z2). Similarly, the external carry from the second addition is carried to Z3. UMUL16 uses a straightforward approach. It can easily be extended to multiplications involving longer words by following the same procedures.

The division of a 16-bit unsigned integer by an 16-bit unsigned integer is accomplished by a series of 16 subtractions. The process takes as many steps as the length of the dividend. In this case, it takes 16 steps. At each step, the divisor is multiplied by 2^n for n=15, 14, ...,0. If the divided is greater than this value, then the divisor is subtracted from the dividend, and the result is used as the updated dividend. At the same time, bit n of the quotient is set. An example may illustrate the procedure.

0x5678 / 0x0123 = 0x004C0, remainder 0x0014.

n	dividend	(2^n) * divisor	subtract	updated dividend	result bit
15	0x5678	0x00918000	no	0x5678	0
14	0x5678	0x0048C000	no	0x5678	0
13	0x5678	0x00246000	no	0x5678	0
12	0x5678	0x00123000	no	0x5678	0
11	0x5678	0x00091800	no	0x5678	0
10	0x5678	0x00048C00	no	0x5678	0
09	0x5678	0x00024600	no	0x5678	0
08	0x5678	0x00012300	no	0x5678	0
07	0x5678	0x00009180	no	0x5678	0
06	0x5678	0x000048C0	yes	0x0DB8	1
05	0x0DB8	0x00002460	no	0x0DB8	0
04	0x0DB8	0x00001230	no	0x0DB8	0
03	0x0DB8	0x00000918	yes	0x04A0	1
02	0x04A0	0x0000048C	yes	0x0014	1
01	0x0014	0x00000246	no	0x0014	0
00	0x0014	0x00000123	no	0x0014	0

The 16 result bits gives 0x004C, as read from the last column. The remainder is the value left after the series of subtractions. Namely, the final updated dividend, 0x0014.

First we check if the divisor is zero. If so, the overflow flag is set, and the procedure aborted. Clearly, if bit 16 of (2^n)*(divisor) is set, the result bit is 0. Thus, the procedure may be started by first finding the smallest n for which the most significant bit (MSB), or bit 15, of (2^n)*(divisor) is 1. The routine `u16flush` keeps shifting the divisor to the left until its MSB is 1. Note that since the divisor is now known not to be zero, eventually, its MSB will be 1. The partial result (partial quotient) is kept in r3:r2. The register pair r1:r0 holds 2^n during the process. As n is decremented during the process, r1:r0 is updated by shifting it to the right. The routine `udiv16rr` shifts both r1:r0 and (2^n)*(divisor) kept in Y to the right. The procedure terminates when r1:r0 reaches 1. The quotient is then copied from r3:r2 to _Z1:_Z0. The remainder is left in _X1:_X0.

```
UDIV16:
    mov    a, _Y1        ; get divisor high byte
    orl    a, _Y0        ; OR with low byte
    jnz    div16_OK      ; divisor OK if not 0
    setb   _ZOV          ; else, overflow
    ret
div16_OK:
    mov    r3, #0        ; r3:r2 partial quotient
```

```
    mov     r2, #0
    mov     r1, #0          ; r1:r0 holds the "1<<n"
    mov     r0, #1
    lcall u16flush          ; rotate until MSB of Y is 1

udiv16Loop:
    lcall  USUB16           ; X-(1<<n)
    jb      _ZOV, udiv16zero
    mov     a, r3           ; update partial quotient
    orl     a, r1
    mov     r3, a
    mov     a, r2
    orl     a, r0
    mov     r2, a
    mov     _X1, _Z1
    mov     _X0, _Z0
udiv16zero:
    mov     a, r0
    jb      acc.0, udiv16done
    lcall  udiv16rr
    sjmp   udiv16Loop
udiv16done:
    mov     _Z1, r3         ; quotient
    mov     _Z0, r2
    ret

u16flush:                   ; rotate "(1<<n)" and Y left until
                            ; MSB of Y is 1
    mov     a, _Y1
    jnb     acc.7, u16f00
    ret
u16f00:
    clr     C
    mov     a, _Y0
    rlc     a
    mov     _Y0, a
    mov     a, _Y1
    rlc     a
    mov     _Y1, a
    clr     C
    mov     a, r0
    rlc     a
    mov     r0, a
    mov     a, r1
    rlc     a
    mov     r1, a
    sjmp   u16flush

udiv16rr:                   ; rotate "(1<<n)" and Y right
```

```
    clr   C
    mov   a, _Y1
    rrc   a
    mov   _Y1, a
    mov   a, _Y0
    rrc   a
    mov   _Y0, a

    clr   C
    mov   a, r1
    rrc   a
    mov   r1, a
    mov   a, r0
    rrc   a
    mov   r0, a
    ret
```

Sometimes it is desirable to divide a 32-bit unsigned integer by a 16-bit unsigned integer. The following subroutine takes the same approach as `UDIV16`. The rotates now involve four bytes. The partial quotient is kept in _PQ3: _PQ2: _PQ1: _PQ0, and the value (2^n) in r3:r2:r1:r0.

```
UDIV32:
    mov   a, _Y1          ; is divisor 0?
    orl   a, _Y0          ; OR with low byte
    jnz   div32_OK        ; divisor OK if not 0
    setb  _ZOV            ; else, overflow
    ret
div32_OK:
    mov   _Y3, _Y1
    mov   _Y2, _Y0
    mov   _Y1, #0
    mov   _Y0, #0
    mov   _PQ3, #0        ; PQ3..PQ0 partial quotient
    mov   _PQ2, #0
    mov   _PQ1, #0
    mov   _PQ0, #0
    mov   r3, #0          ; r3:r2:r1:r0 holds the "1<<n"
    mov   r2, #1
    mov   r1, #0
    mov   r0, #0
    lcall u32flush        ; rotate until MSB of Y is 1

udiv32Loop:
    lcall USUB32          ; X-(1<<n)
    jb    _ZOV, udiv32zero
    mov   a, _PQ3         ; update partial quotient
    orl   a, r3
```

```
    mov     _PQ3, a
    mov     a, _PQ2
    orl     a, r2
    mov     _PQ2, a
    mov     a, _PQ1
    orl     a, r1
    mov     _PQ1, a
    mov     a, _PQ0
    orl     a, r0
    mov     _PQ0, a
    mov     _X3, _Z3
    mov     _X2, _Z2
    mov     _X1, _Z1
    mov     _X0, _Z0
udiv32zero:
    mov     a, r0
    jb      acc.0, udiv32done
    lcall   udiv32rr
    sjmp    udiv32Loop
udiv32done:
    mov     _Z3, _PQ3     ; quotient
    mov     _Z2, _PQ2
    mov     _Z1, _PQ1
    mov     _Z0, _PQ0
    ret

u32flush:                 ; rotate "(1<<n)" and Y left until
                          ; MSB of Y is 1
    mov     a, _Y3
    jnb     acc.7, u32f00
    ret
u32f00:
    clr     C
    mov     a, _Y0
    rlc     a
    mov     _Y0, a
    mov     a, _Y1
    rlc     a
    mov     _Y1, a
    mov     a, _Y2
    rlc     a
    mov     _Y2, a
    mov     a, _Y3
    rlc     a
    mov     _Y3, a
    clr     C
    mov     a, r0
    rlc     a
    mov     r0, a
```

```
    mov   a, r1
    rlc   a
    mov   r1, a
    mov   a, r2
    rlc   a
    mov   r2, a
    mov   a, r3
    rlc   a
    mov   r3, a
    sjmp  u32flush

udiv32rr:                ; rotate "(1<<n)" and Y right
    clr   C
    mov   a, _Y3
    rrc   a
    mov   _Y3, a
    mov   a, _Y2
    rrc   a
    mov   _Y2, a
    mov   a, _Y1
    rrc   a
    mov   _Y1, a
    mov   a, _Y0
    rrc   a
    mov   _Y0, a

    clr   C
    mov   a, r3
    rrc   a
    mov   r3, a
    mov   a, r2
    rrc   a
    mov   r2, a
    mov   a, r1
    rrc   a
    mov   r1, a
    mov   a, r0
    rrc   a
    mov   r0, a
    ret
```

3.9. 16-Bit Signed Arithmetic Routines

Signed 16-bit integers in two's complement format are added and subtracted in the same manner as unsigned integers. The only difference is that the overflow flag OV is set during the addition or subtraction of the most significant bytes when the result is out of range. Signed multiplication and division is easier if the integers are in the absolute-value/sign-bit format. Then the absolute value of the result is computed in

the same manner as multiplying or dividing unsigned integers. The sign of the result if computed from the signs of the operands.

3.9.1. 16-Bit Signed Addition and Subtraction

The 16-bit signed integer addition and subtraction operations are straight forward extensions of their 8-bit counterparts. The 16-bit operations are initiated with the low bytes. Any carry or borrow from the low-byte operation is carried to the high-byte operation. Again, the overflow flag OV is set during the high-byte operation if the result is out of range.

The following signed addition and subtraction routines assume that the internal registers _X1:_X0, _Y1:_Y0, and _Z1:_Z0 hold the high and low bytes of 16-bit signed integers X, Y, and Z, expressed in two's complement format. The addressable bit ZOV is set if there is an overflow, i.e., if the result Z is out of range.

Addition and subtraction routines require no further work beyond calling the MCS-51 instructions `add`, `addc`, and `subb`. In fact, it is the applicability of these instructions to unsigned as well as signed numbers in two's complement form that makes the two's complement format practical.

```
TCADD16:
    mov    a, _X0        ; load _X low byte into accumulator
    add    a, _Y0        ; add _Y low byte
    mov    _Z0, a        ; put result in _Z low byte
    mov    a, _X1        ; load _X high byte into accumulator
    addc   a, _Y1        ; add _Y high byte with the carry from
                         ; low byte operation
    mov    _Z1, a        ; save result in_Z high byte
    mov    C, OV         ; get the overflow flag and ...
    mov    _ZOV, C       ; ... transfer to bit_ZOV
    ret                  ; done

TCSUB16:
    mov    a, _X0        ; load X low byte into accumulator
    clr    C             ; clear carry flag
    subb   a, _Y0        ; subract Y low byte
    mov    _Z0, a        ; put result in Z low byte
    mov    a, _X1        ; load X high byte into accumulator
    subb   a, _Y1        ; subtract Y high byte with the
                         ; borrow from low byte operation
    mov    _Z1, a        ; save result in Z high byte
    mov    C, OV         ; get the overflow flag and ...
    mov    _ZOV, C       ; ... transfer to bit ZOV
    ret                  ; done
```

These routines may easily be extended to 3- or 4-byte signed addition and subtraction operations.

3.9.2. 16-Bit Signed Multiplication and Division

The signed 16-bit multiplication and division operations are performed simply by multiplying the operands as with unsigned 16-bit operations and then computing the sign bits, provided that the signed number are in the absolute-value/sign-bit format. If the signed numbers are represented in two's complement notation, first they are converted to the absolute-value/sign-bit format. The result may then be converted back to two's complement format. The following subroutines illustrate multiplying and dividing 16-bit signed integers in the absolute-value/sign-bit format. The routine for division is given below.

```
MSDIV16:
   lcall  UDIV16        ; divide absolute values
   jb     _XS, SD16_2   ; get sign of_X
   jb     _YS, SD16_1
   clr    _ZS
   ret
SD16_1:
   setb   _ZS
   ret
SD16_2:
   jb     _YS, SD16_3
   setb   _ZS
   ret
SD16_3:
   clr    _ZS
   ret
```

3.10. Floating-Point Arithmetic Routines

There are several formats to represent floating point numbers. For example, the double precision floating point defined by IEEE (Institute of Electrical and Electronic Engineers) uses 64 bits, or 8 bytes. There is a sign bit, an 11-bit exponent, and a 52-bit mantissa. The range is approximately $\pm 1.7 \times 10^{\pm 308}$, which gives 15 to 16 decimal digits of precision. In general, the number of mantissa bits determine the precision, and the number of exponent bits determine the range. Formats other than the IEEE double precision format are possible. In fact, it may be desirable to construct and adopt the format most suitable for the application. The subroutines given in this section use a 3-byte, 24-bit format. There are 8 bits of exponent, a sign bit, and 15 bits of mantissa. The most significant bit of the mantissa is an implied '1', as shown below.

$$f = (\text{eeee eeee smmm mmmm mmmm mmmm})_2$$

$$f = (-1)^S \times 1.mmm\ mmmm\ mmmm\ mmmm \times 2^{(eeee\ eeee) - 127}$$

Notice that the mantissa is 16 bits, with its most significant bit assumed to be 1. If the number is 0, then the exponent eeee eeee is 0. Again, note that the exponent is not stored as in the two's complement format, but simply written with a negative offset of 127. Storing the exponent in a single byte simplifies the effort in manipulating the numbers.

3.10.1. Floating-Point Multiplication and Division

Floating point multiplication and division requires multiplying or dividing the mantissas and adding or subtracting the exponents. A check should be performed for the special case, to see if either number is zero, as well as to see if the exponent is out of range. Notice that if neither number is zero, the most significant bit of the product of mantissas will be 1. The exponents of the two multiplicands are simply added. Next consider the mantissas. Since the mantissa, with the implied 1, is the fraction expressed as

$$1.mmm\ mmmm\ mmmm\ mmmm$$

the product of two mantissas will be in the range [1, 4). The interval includes 1 but excludes 4, indicated by the square left paranthesis and the regular right paranthesis. That is, greater than or equal to 1 and strictly less than 4. If the product of the mantissas exceeds 1, the result must be adjusted by shifting to the right once, and the exponent incremented.

The exponents of the numbers are added. Since an offset of 127 is used, this value must be subtracted from the sum. In the program given below the value 128 is subtracted from the sum of exponents. If the product of the mantissas is strictly less than 2, the exponent is incremented, which is equivalent to subtracting 127 from the sum of the exponents.

Similarly, in dividing two floating point numbers, it may be necessary to adjust the exponent so that the most significant bit of the mantissa of the result is 1. The division of the mantissas is strictly between 0.5 and 2, exclusively. If the result is less than 1, then the mantissa is shifted to the left, and the exponent is decremented. A simple comparison subtracts the divisor from the dividend. If the result does not produce a carry, then the result of the division is at least 1, and no adjustment is necessary. Else, the mantissa is shifted and the exponent decremented.

First consider the following subroutine for floating point multiplication.

```
FMUL:
    mov     a, _X1          ; get the sign bit plus 7 mantissa
                            ; bits of X
```

```
    mov   C, acc.7      ; read sign bit
    mov   _XS, C        ; save sign bit
    mov   a, _Y1        ; get the sign bit plus 7 mantissa
                        ; bits of Y
    mov   C, acc.7      ; read sign bit
    mov   _YS, C        ; save sign bit
    jb    _XS, FMUL_2   ; get sign of X
    jb    _YS, FMUL_1
    clr   _ZS
    sjmp  FMUL_4
FMUL_1:
    setb  _ZS
    sjmp  FMUL_4
FMUL_2:
    jb    _YS, FMUL_3
    setb  _ZS
    sjmp  FMUL_4
FMUL_3:
    clr   _ZS
FMUL_4:
    mov   r2, _X2       ; save exponent of X in r2
    mov   r3, _Y2       ; save exponent of Y in r3
    cjne  r2, #0, XNZ   ; see if X is zero
    mov   _Z2, #0       ; if X=0, then Z=0
    mov   _Z1, #0
    mov   _Z0, #0
    ret                 ; done
XNZ:
    cjne  r3, #0, YNZ   ; see if Y is zero
    mov   _Z2, #0       ; if Y=0, then Z=0
    mov   _Z1, #0
    mov   _Z0, #0
    ret                 ; done
YNZ:
    mov   a, r2         ; get exponent of X
    add   a, r3         ; add exponent of Y
    mov   b, #0         ; clear b register
    mov   b.0, C        ; move carry bit into b register
                        ; b register is high byte of sum of
                        ; exponents
    clr   C
    subb  a, #128       ; subtract 128 from sum of exponents
                        ; low byte
    mov   r2, a         ; save exponent in r2
    mov   a, b          ; get sum of exponents high byte
    subb  a, #0         ; subtract high bytes
    jnc   NoUNF         ; external borrow means underflow
    mov   _Z2, #0       ; underflow...set Z=0
    mov   _Z1, #0       ; it may be more appropriate in some
```

```
    mov     _Z0, #0         ; applications to set ZOV and signal
                            ; an error
    ret                     ; done
NoUNF:
    jz      NoOVF           ; if high byte is 1 then overflow
    setb    _ZOV            ; set overflow error flag
    ret                     ; done

; ---      if the program reaches NoOVF, the exponent is in r2 ---
; --- and the sign is in ZS.  next multiply mantissas      ---
NoOVF:
    mov     a, _X1          ; get X1
    setb    acc.7           ; set the implied MSB=1
    mov     _X1, a          ; save completed mantissa of X
    mov     a, _Y1          ; get Y1
    setb    acc.7           ; set the implied MSB=1
    mov     _Y1, a          ; save completed mantissa of Y
    lcall   UMUL16          ; multiply the mantissas
    mov     a, _Z3          ; get mantissa high byte
    jb      acc.7, XYOK     ; mantissa is OK if MSB is 1

; --- if MSB of XY is 0, then the mantissa is shifted left ---
; --- and the exponent is incremented ----------------------
    inc     r2              ; this may result in an overflow
    cjne    r2, #0, EXP_OK
    setb    _ZOV            ; overflow
    ret
EXP_OK:
    clr     C
    mov     a, _Z1
    rlc     a
    mov     a, _Z2
    rlc     a
    mov     _Z0, a
    mov     a, _Z3
    rlc     a
    mov     C, _ZS
    mov     acc.7, C        ; sign bit in place of the implied '1'
    mov     _Z1, a
    mov     _Z2, r2
    ret
XYOK:
    mov     _Z0, _Z2        ; mantissa low byte
    mov     a, _Z3          ; mantissa high byte
    mov     C, _ZS
    mov     acc.7, C        ; sign bit in place of the implied '1'
    mov     _Z1, a
    mov     _Z2, r2         ; get exponent
    ret                     ; done
```

As a numerical example, consider X = 4.29 and Y = 89.5. The product is 383.955. First we write X in our three-byte floating-point format. This requires X to be written as a floating-point number in the range of [1, 2) multiplied by a power of 2.

$$4.29 = 1.0725 * 4 = 1.0725 * (2^2)$$

Hence, the exponent is 2. With the 127 offset, _X3=129=0x81. Next we write 1.0725 in binary. The most-significant-bit of the mantissa is the whole number, and the remaining 15 bits are the fraction. That is,

1.0725 = 1.abc defg hijk lmno

where a, b, .., o, are the 15 binary digits. If we were to multiply both sides by 2, we woul have,

2*1.0725 = 2.145 = 1a.bc defg hijk lmno

Now the integral part of the binary representation comprises two digits (1a) and the fraction part, 14 digits. Continuing this argument,

$1.0725 * 2^{15} = 35143.68$ = 1abc defg hijk lmno.

or

35143 = 0x8947 = 1000 1001 0100 0111 = 1abc defg hijk lmno.

So the mantissa is 0x8947. Notice that the decimal value 35143.68 still contains a fraction. Since the binary representation is truncated after 16 digits, this fraction, 0.68 is not accounted for. The conversion of a decimal fraction to a binary fraction may require an infinite number of digits, just as 10/3 requires an infinite number of digits on a decimal calculator. The error caused by the left over fraction is called a truncation error. In our three-byte floating-point format, we use the most significant bit of the mantissa as the sign bit. The most significant bit is implied to be 1. Since X is positive, the sign bit is cleared. The three bytes of the floating-point representation are as follows.

_X2 = 0x81
_X1 = 0x09
_X0 = 0x47

Similarly, we find the exponent and mantissa of 89.5. Note that 89.5 ends in 0.5, a simple binary fraction, namely 2^{-1}, and thus, will not introduce to a truncation error.

$$Y = 89.5 = 1.3984375 * 64 = 1.3984375 * 2^6$$

Again,

$$1.3984375 * 2^{15} = 45824 = 0xB300.$$

Note that there is no truncation error in this case.

_Y2= 0x85
_Y1= 0x33
_Y0= 0x00

The routine FMUL returns the result

_Z2= 0x87
_Z1= 0x3F
_Z0= 0xF9

which corresponds to an exponent of 8 and a mantissa of 1.499786376953125.

$$1.499786376953125 * 2^{8} = 383.9453125.$$

Comparing the result to the expected (383.955), we find a small difference, due to the truncation error in X.

The division of two floating numbers requires the division of the mantissas and the subtraction of the exponent of the divisor from that of the dividend. Since the most significant bit of the mantissas are 1, the division of the mantissas will result in a number strictly greater than half and strictly less than 2. A simple comparison of the dividend and the divisor mantissas will further clarify the range of the result. If the mantissa of the dividend is greater than that of the divisor, then the division of the mantissas will result in a value between 1 and 2. Similarly, if the mantissa of the dividend is less than that of the divisor, then the division of the mantissas will result in a value between 0.5 and 1. The subroutine FDIV divides the mantissa of the dividend by the mantissa of the divisor. The exponent of the divisor is then subtracted from that of the dividend. The exponent must be adjusted by adding 127. However, if the division of the mantissas results in a value greater than 1, then the exponent is decremented, or effectively, 128 is added to the difference of the exponents.

```
FDIV:
    mov     a, _Y2          ; get divisor exponent
    jnz     fd_1            ; continue if Y is not 0
    setb    _ZOV            ; set overflow flag if Y=0
    ret
fd_1:
```

```
      mov   a, _X1        ; sign bit plus 7 mantissa bits of X
      mov   C, acc.7      ; read sign bit
      mov   _XS, C        ; save sign bit
      setb  acc.7         ; put the implied 1
      mov   _X1, a        ; store mantissa with implied 1 restored
      mov   a, _Y1        ; sign bit plus 7 mantissa bits of Y
      mov   C, acc.7      ; read sign bit
      mov   _YS, C        ; save sign bit
      setb  acc.7         ; put the implied 1
      mov   _Y1, a        ; store mantissa with implied 1 restored
      lcall getZS         ; _ZS = _XS EXOR _YS

      mov   r7, _X2       ; save the numbers..
      mov   r6, _X1       ; ..note the implied 1's are inserted
      mov   r5, _X0
      mov   r4, _Y2
      mov   r3, _Y1
      mov   r2, _Y0
      lcall USUB16        ; is X >= Y ?
      mov   C, _ZOV       ; get the external borrow
                          ; no adjustment if X >= Y
      mov   _ZADJ, C      ; no adjustment if _ZADJ=0
      mov   a, r7         ; get exponent of X
      mov   b, #0         ; use b register as subb high byte
      jb    _ZADJ, fd_6   ; depending on ZADJ...
      add   a, #127       ; ... add the offset
      sjmp  fd_7
fd_6:
      add   a, #126       ; offset with adjustment for
                          ; (mantissa Y) > (mantissa X)
fd_7:
      mov   b.0, C        ; external carry from sum low byte
      clr   C             ; get ready for subb
      subb  a, r4         ; next subtract exponent of Y
      mov   _ZEXP, a      ; save exponent in ZEXP
      mov   a, b          ; get sum high byte
      subb  a, #0         ; complete the subtraction
      jnc   fd_8          ; underflow if carry is set
      mov   _Z0, #0       ; make result 0
      ret                 ; done

; --- if the program reaches fd_8, the exponent is in ZEXP ---
; --- and the sign is in ZS.  next divide mantissas ----------
fd_8:
      mov   _X3, r6       ; make X a 32-bit number...
      mov   _X2, r5       ; ... multiply X by 2^16
      mov   _X1, #0
      mov   _X0, #0
      mov   _Y1, r3
```

```
    mov     _Y0, r2
    lcall   UDIV32          ; compute the mantissa of Z=X/Y

    jnb     _ZADJ, fd_9     ; if result is 1 or greater, rotate

    mov     a, _Z1
    mov     C, _ZS
    mov     acc.7, C
    mov     _Z1, a
    mov     _Z2, _ZEXP      ; get exponent
    ret                     ; done
fd_9:
    mov     C, _ZS          ; rotate right
    mov     a, _Z1
    rrc     a
    mov     _Z1, a
    mov     a, _Z0
    rrc     a
    mov     _Z0, a
    mov     _Z2, _ZEXP      ; get exponent
    ret                     ; done
```

3.11. Pseudo-Random Numbers

Many applications, such as search algorithms or games, require random numbers. Random numbers that are generated by a finite state machine, such as a microcontroller or a mainframe computer, by an algorithmic approach are called pseudo random. Sequences of pseudo-random numbers repeat. However, the period of these sequences are typically very long and the random numbers in each period are uniformly distributed.

A simple algorithm to generate random numbers is given by the iterative relationship

$$x_{i+1} = (a\ x_i) \bmod m$$

where x_i is the i-th pseudo random number in the sequence and a and m are constants parameters. At each iteration, the current pseudo-random number is multiplied by the parameter a and the result is divided by m. The remainder is the next pseudo-random number. The pseudo-random numbers have a distribution which very closely approximates the uniform distribution with an interval [0, m-1]. If the random numbers are viewed as fractions, i.e. x_i/m, then their distribution is considered to be the unit uniform distribution.

In order to reduce the computational effort, it is convenient to let the parameter m be 2^8, 2^{16}, or 2^{32}. These choices eliminate the need for the division operation. The

next pseudo-random number is simply the lower 8, 16, or 32 bits of the multiplication result.

The entire sequence of pseudo random numbers is uniquely determined by the constant parameters a and m, and by the initial value x_0. The initial value x_0 is referred to as the seed of the sequence. The length or period of the sequence is limited by the word length m. More specifically, the pseudo-random number sequence repeats after at most 2^{m-2} iterations. The following parameters are examples that achieve the maximum length sequences.

Table 3.9.

	x_0	a	m	Sequence Length
16-bit random numbers	731	517	16	16384
32-bit random numbers	1759	941	32	1073741824

A few of the 16-bit pseudo-random numbers generated by the seed 731 and the parameter a=517 are computed as

731, 50247, 25443, 46831, 28843, 35159, 23731, 13695, 2427, 9575, 35075....

Since the same seed will always produce the same sequence of pseudo-random numbers, the seed should be chosen as randomly as possible to generate different sequences of numbers. Depending on the application, the seed may be taken as the current value of a variable. For example, the seed may be the time of the hour represented in seconds, or perhaps the product of the ASCII codes of the users initials.

3.12. String Manipulation Routines

The 8051 family microcontrollers have a serial port which may be used to talk to a host PC or peripheral devices such as a printer. The SFR SCON and SBUF are the control and data registers of the serial port. Before serial communications may begin, the serial port must be initialized. Refer to the next chapter for examples on how the serial port is set up an operated in either a polled fashion or an interrupt-driven fashion. The subroutines GETC and PUTC transmit and receive a single character, respectively. The accumulator holds the byte received or transmitted. These subroutines are given in Chapter 4. This section discusses techniques for transmitting strings out of the serial port. Many of the subroutines given below are used as utility routines in other programs in this book.

The first utility subroutine transmits a sequence of two characters, the carriage return and the line feed character. If a ASCII terminal or a host computer emulating an ASCII terminal is attached to the microcontroller through the serial port, this routine will cause a newline. The subroutine is self explanatory. Although it may look redundantly simple, it illustrates the use of structured programming principles. Many programs in this book make a call to CRLF. Also note that LFEED is an entry point to the routine which simply sends a line-feed character. Althugh DOS requires a carrige return and a line-feed character to initiate a new line, in UNIX, the line-feed character suffices.

```
crlf:
    mov    a, #LF        ; print lf
    lcall  putc
lfeed:
    mov    a, #CR        ; print cr
    lcall putc
    ret
```

The macros CR and LF are defined in <util51.inc> as below.

```
#define LF 0x0A
#define CR 0x0D
```

Programs often need to display numerical results on a terminal or host computer. The numerical value of the accumulator, for example, is first converted to two characters and then transmitted. The following two subroutines accomplish this task. The only difference in the two subroutines is that PRSPHX (PRint SPace HeX) includes a blank prefix, which is useful if many numbers are to be displayed in a row. The routines call the binary-to-ASCII conversion routine BINASC to obtain the ASCII values of the two nibbles.

```
; ========================================================
; subroutine prthex
; this routine takes the contents of the accumulator and
; prints it out as a 2 digit ASCII hex number.
;
; input    : number in accumulator
; output   : characters out the serial port
; calls    : binasc, putc
; destroys: a, r2
; ========================================================
prthex:
    lcall  binasc        ; convert acc to ascii
    lcall  putc          ; print first ascii hex digit
    mov    a,  r2        ; get second ascii hex digit
    lcall  putc          ; print it
    ret
```

```
; ========================================================
; subroutine prsphx
; this routine first prints a space
; then takes the contents of the accumulator and prints
; it out as a 2 digit ASCII hex number.
;
; input   : number in accumulator
; output  : characters out the serial port
; calls   : binasc, putc
; destroys: a, r2
; ========================================================
prsphx:
   mov   r2, a          ; save acc in r2
   mov   a, #20h        ; print space
   lcall putc
   mov   a, r2          ; recall acc
   lcall prthex         ; print it
   ret
```

The subroutine PRINT is used by placing a null-terminated string immediately after the call instruction. Note that the program counter is incremented to the beginning of the string after the call instruction. The program counter, now a pointer to the string is pushed on stack. The subroutine pops the string pointer from stack and places the pointer in DPTR. Each character is transmitted as the data pointer is incremented, until a 0 is encountered. The subroutine then makes a jump to the calling program.

```
; ========================================================
; subroutine print
; print takes the string immediately following the call and
; sends it out the serial port.  the string must be terminated
; with a null. this routine will ret to the instruction
; immediately following the string.
;
; input   : nothing
; output  : characters out the serial port
; calls   : putc
; destroys: a, dptr
; ========================================================
print:
   pop   dph                ; put return address in dptr
   pop   dpl
prtstr:
   clr   a                  ; set offset = 0
   movc  a, @a+dptr         ; get chr from code memory
   cjne  a, #0h, mchrok     ; if chr = ff then return
   sjmp  done
mchrok:
```

```
    lcall  putc              ; send character
    inc    dptr              ; point at next character
    ljmp   prtstr            ; loop till end of string
done:
    mov    a,  #1h           ; to instruction after string
    jmp    @a+dptr           ; return
```

Subroutine GETBYT shown below reads two characters from the serial port, converts each character to a hexadecimal character, combines the low and high nibbles into a byte and returns the result in the accumulator. The previously discussed conversion routine ASCBIN is employed in obtaining the numeric values of the nibbles. The error flag may be checked to detect if any of the characters is not a valid hexadecimal digit.

```
; ========================================================
; subroutine getbyt
; this routine reads in an 2 digit ascii hex number from the
; serial port. the result is returned in the accumulator.
;
; input    : characters in the serial port
; output   : byte in accumulator
; calls    : getc, ascbin
; destroys: b, r2
; ========================================================
getbyt:
    lcall  getc          ; get msb ASCII chr
    lcall  ascbin        ; conv it to binary
    swap   a             ; move to most sig half of acc
    mov    b,  a         ; save in b
    lcall  getc          ; get lsb ascii chr
    lcall  ascbin        ; conv it to binary
    orl    a,  b         ; combine two halves
    ret
```

The subroutine GETBYT has no editing features. For example, if an incorrect key is pressed at the host and received by GETBYT, a backspace key or a delete key cannot remove the incorrect character. An attempt is made to convert the two characters to a valid hexadecimal number. Moreover, if a single digit hexadecimal value is to be entered, say 3, it must be presented to GETBYT in two characters as 03. Simply typing 3 and a carriage return (<CR>) does not work. Similarly, there is no room for leading or trailing blanks.

Adding simple edit features complicates the routine. It becomes necessary to implement a small line buffer. The characters received are first accumulated in the line buffer. Special keys, such as backspace, delete, and <CR> may modify the contents of the buffer. Moreover, the user needs to be informed the status of the buffer, and the position of the cursor.

The following routine GETWORD implements a 16-byte buffer in internal registers, starting at address LBuffer (Line Buffer). Characters are received and echoed to the terminal or host. Backspace and delete characters are also processed. If a backspace character is received, it is echoed to the host, which moves the host cursor back one space. Then a space character is transmitted to effectively erase the last character on the host display. Then another backspace character is sent to move the host cursor back one step. This way the user, when backspace is pressed, sees the cursor backed up one space and the last character removed from the display. The <CR> character has the special meaning that the line editing is completed. The routine may then start inspecting the characters and attempt to convert them into a hexadecimal word of 2 bytes. The word is stored in two consecutive internal registers starting at address PBuffer (Parameter Buffer). Line editing is also terminated if the line buffer becomes full.

```
; ========================================================
; subroutine getword
; this subroutine reads in an ASCII string from the serial
; port.  the line must be terminated with a <cr>.  the line
; is stored in internal registers starting at address
; LBuffer. the maximum line length is 16 bytes including
; the <cr>.  the word is returned in internal registers at
; location PBuffr.
;
; input   : characters in the serial port
; output  : up to 16 bytes in LBuffer
; calls   : getc, putc
; destroys: nothing (restores used registers)
; ========================================================
getword:
    push  psw
    mov   psw, #0
    push  3
    push  4
    push  5
    push  6

    clr   errorf
    clr   ri                  ; flush serial port
    mov   r0, #LBuffr         ; init line buffer ram to 0's
gwinit:
    mov   @r0, #0FFh
    inc   r0
    cjne  r0, #LBuffr+11h, gwinit

    mov   r4, #0h             ; init line buffer count (pointer)
gw0:
    mov   a, r4               ; if line length > 15 then error
```

```
    jnb     acc.4, gwok0
    ljmp    gwerr_ret
gwok0:
    lcall   getc                ; read in character
    mov     r1, a               ; save in r1
    cjne    r1, #7fh, gwnodel   ; if del then delete
    sjmp    gwdel
gwnodel:
    cjne    r1, #08h, gwnorub   ; if back space then del
gwdel:
    mov     a, r4               ; do not back up over prompt
    jz      gw0
    mov     a, #08h             ; backspace
    lcall   putc
    mov     a, #20h             ; send a space
    lcall   putc
    mov     a, #08h             ; backspace
    lcall   putc
    dec     r4                  ; dec line buffer pointer.
    sjmp    gw0
gwnorub:
    mov     a, r1               ; recall char
    clr     c
    add     a, #0c0h
    jnc     gw_num              ; if alphabetic...
    mov     a, r1               ; then make upper case
    clr     acc.5
    mov     r1, a
gw_num:
    mov     a, r4               ; save char in line buffer.
    add     a, #lbuffr          ; compute address
    mov     r0, a
    mov     a,  r1              ; get char
    mov     @r0, a
    lcall   putc                ; echo character

    inc     r4                  ; point to next location in buffer
    cjne    r1, #0dh, gw0       ; if not cr then return
    mov     a,  #0ah            ; if chr=cr then line feed and do
                                ; a normal ret
    lcall   putc
    mov     a, r4               ; if cr only then error
    cjne    a, #1h, gwok
    ljmp    gwerr_ret
gwok:
    mov     r0, #lbuffr-1       ; point before first character
                                ; in the line buffer
```

```
; ----------  read in parameter  ---------------------
gw1:
   inc    r0                  ; point to next location in line
                              ; buffer
   mov    a,  @r0             ; get chr from line buffer
   cjne   a,  #0dh, gw2       ; if cr then end
   ljmp   gwerr_ret           ; return on error
gw2:
   cjne   a,  #20h, gw3       ; if chr=space then loop
   sjmp   gw1
gw3:
   mov    r1, #pbuffr         ; parameter 0
   lcall  getparx             ; get parameter from line buffer
   jnb    errorf, gwret
gwerr_ret:
   setb   errorf
; --- an error message may be generated if appropriate ---
gwret:
   pop    6
   pop    5
   pop    4
   pop    3
   pop    psw
   ret
```

3.13. Software Timing Routines

It is almost always preferable to use one of the microcontrollers timers rather than software loops to insert delays. A timer has the advantage of issuing an interrupt upon overflow, thus signaling the end of a delay period. Moreover, while the timer keeps time, the microcontroller is free to execute other less critical tasks. Nonetheless, simple software loops to waste just the right amount of time always find applications. The number of times the delay loop is executed in the subroutine given below depends on the value of the accumulator. Specifying the delay time makes the delay routine flexible enough to be used in non-critical applications where CPU loads are light.

A 12 MHz crystal frequency is assumed. The accumulator holds the number of milliseconds to delay. The subroutine execution time is 2 microseconds less than the specified delay, since 2 microseconds are required for the call to the routine. For example, in a typical implementation,

```
instruction1
lcall  delay
instruction2
```

instruction 2 is executed exactly nn milliseconds after instruction1, where nn is the contents of the accumulator.

```
; ================================================
; subroutine delay - millisecond delay
; accumulator holds microseconds to delay
; - 2 microseconds are reserved for the call
; to this routine.  A 12MHz crystal is assumed.
;
; input    : milliseconds to delay in accumulator
; output   : none
; destroys: nothing - uses a
; ------------------------------------------------
; 100h-A6h=5Ah=(90)decimal
; 90 * 11 = 990
; plus 10 gives 1 millisecond
;
;                          microseconds (cycles)
;                         ----------------------
delay:
        dec   a           ;    1
d_olp:
        push acc          ;    2            \
        mov   a, #0a6h    ;    1            |
d_ilp:                    ;                 |
        inc   a           ;    1 .\         |
        nop               ;    1  |         |
        nop               ;    1  |         |
        nop               ;    1  |         |
        nop               ;    1  |         |
        nop               ;    1  |- 11     | (acc-1)
        nop               ;    1  | cycles  |- msec
        nop               ;    1  |         |
        nop               ;    1  |         |
        jnz   d_ilp       ;    2  /         |
                          ;                 |
        nop               ;    1            |
        nop               ;    1            |
        nop               ;    1            |
        pop   acc         ;    2            |
                          ;                 |
        djnz  acc,d_olp   ;    2            /

; need to wait 998 microseconds more

        mov   a, #0A6h    ;    1

d_lp2:  inc   a           ;    1  \
        nop               ;    1  |
```

```
        nop             ;     1   |
        nop             ;     1   |
        nop             ;     1   |
        nop             ;     1   |- 11
        nop             ;     1   |  cycles
        nop             ;     1   |
        nop             ;     1   |
        jnz   d_lp2     ;     2   /
        nop             ;     1
        nop             ;     1
        nop             ;     1
        nop             ;     1
        nop             ;     1

        ret             ;     2
```

3.14. Re-entrant Routines

A routine that calls itself is said to be re-entrant. Re-entrant routines are natural mechanisms to implement recursive algorithms. Consider computing the sum of integers 1+2+...N. The simple program below illustrates a straight-forward approach. Here, the sum is actually computed as N + (N-1) + ... + 2 + 1. The upper bound N is defined by a macro.

```
#define N 7             ; sum 1 to N

    cseg at 0x8000
    mov    r0, #N       ; r0=N
    lcall  _sum         ; find 1+2+3+...+N, (N=r0)
    nop                 ; r1 holds the sum (observe r1)
    sjmp   $            ; absorbing loop

; --- subroutine _SUM ---
_SUM:
    mov    r1, #0
    inc    r0           ; prepare for the loop (since
                        ; djnz preincrements)
_LOOP:
    djnz   r0, _MORE
    ret
_MORE:
    mov    a, r1
    add    a, r0
    mov    r1, a
    sjmp   _LOOP

    end                 ; ends cseg
```

The routine _SUM may be viewed as the implementation of the function, say S(N) = 1+2+3+ ... +N. The program given above relies on the subroutine _SUM to add successively decremented numbers to the running sum kept in r1. When _SUM returns, the result (in this case, 28) is in r1.

Alternatively, the function S(.) may be defined by a recursive relationship. Here, the relationship2

$$S(n) = n + S(n-1). \qquad n>0$$

along with the initial condition

$$S(1)=1$$

uniquely specify the function S(.). The routine below implements this recursion. The upper bound N is placed in the accumulator before the routine is called. When the routine returns, the result is in the accumulator.

```
_SUM:
    cjne   a, #1, _more
    ret                    ; if N=1, then sum=1
_MORE:
    push   acc             ; save N
    dec    a               ; acc=N-1
    lcall  _sum            ; find 1+2+...(N-1)
    pop    b               ; this is N on stack
    add    a, b            ; compute sum(N-1) + N
    ret
```

The routine simply returns if the accumulator holds 1. This corresponds to the initial condition. Otherwise, the routine saves the accumulator, decrements the accumulator and calls itself. Where the accumulator (n) is to be saved before S(n-1) is computed is the key insight needed to write re-entrant routines. Clearly, (n) cannot be saved in variable with a fixed address. Otherwise, the next time the routine is called, the previous value would be overwritten. Here, the value is saved on stack. When the function returns, it is popped and added to the return value.

Single step the program and observe that during the recursion, the successive values (N), (N-1), ..., (2) are pushed on stack.

Here, the depth of the recursion (N-1) is limited by the size of the stack. Deeper recursions are possible if a software stack is implemented in external data memory,

2 More formally, S(.) is a sequence in n, satisfying the first-order difference equation S(n) – S(n-1) = n. A single boundary condition is sufficient to uniquely determine S(.).

instead of using the internal hardware stack. Software stacks are presented in the next section.

3.15. Software Stacks

The stack is a good mechanism to temporarily store intermediate information. The stack is the preferred structure in implementing recursive algorithms, evaluating expressions with nested parantheses, and passing arguments to subroutines. A variable number of arguments may be passed, and a variable number of return bytes may be received through the stack. Almost all compilers use a software stack in external data memory rather than the internal stack. The internal stack is also called the CPU stack. The external software stack removes the size constraint of the CPU stack. Since the 8051 supports up to 64K of external memory, relatively large stacks could be built in software.

In the following example, the two registrs SPH:SPL are used as a pointer to the software stack kept in external data memory. A byte placed in the accumulator is pushed on the stack by the subroutine _SSPUSH. Similarly, _SSPOP pops a byte off the stack and places it in the accumulator. In this example, the software stack grows upward (pushes increment SPH:SPL). More precisely, pushes use a post-increment, and pops use a pre-decrement. These are relatively arbitrary conventions. You may easily modify the code to use pre-increment pushes and post-decrement pops. You may also change the code to make the stack grow downwards, i.e., let the pushes decrement and pops increment the stack pointer.

```
SPL     data        0x10
SPH     data        0x11

__SSPUSH:
    push    dpl         ; save dptr on internal stack
    push    dph
    mov     dph, SPH    ; dptr = SPH:SPL
    mov     dpl, SPL
    movx    @dptr, a    ; read [SPH:SPL]
    inc     dptr        ; update dptr
    mov     SPH, dph    ; save updated dptr in SPH:SPL
    mov     SPL, dpl
    pop     dph         ; restore dptr
    pop     dpl
    ret

__SSPOP:
    push    dpl         ; save dptr on internal stack
    push    dph
    mov     dph, SPH    ; dptr = SPH:SPL
    mov     dpl, SPL
    mov     a, dpl      ; dec dptr by adding 0xFFFF
```

```
add     a, #0xFF
mov     dpl, a
mov     SPL, a
mov     a, dph
addc    a, #0xFF
mov     dph, a
mov     SPH, dph
movx    a, @dptr        ; get top of software stack
pop     dph             ; restore dptr
pop     dpl
ret
```

The pushes and pops on and off the software stack act as the regular push and pop operations, even though there is a some code overhead associated with the operations. Note that the two subroutines given above first save the data pointer on the CPU stack. The data pointer is restored (by pop instructions) immediately before the subroutines return. Thus, if the calling program had set the data pointer, its value will be unchanged when the subroutines execute their code and return.

3.16. Macro Definitions, Preprocessor Directives, and Conditional Assembly

The Reads51 assembler calls the internal preprocesor before assembling the source file. The preprocessor reads the source file, inserts the include files, and expands the macros. In addition, the preprocessor processes the conditional directives `#ifdef`, `#ifndef`, `#else`, `#endif`.

The macro definitions may be simple parameter or string substitutions. For example,

```
#define CONSTANT   42h

    .
    mov     a, #CONSTANT
    .
    .
    mov     b, #CONSTANT
    .
```

substitutes `CONSTANT` for 42h anywhere in the code. Such macro definitions are useful if the constant appears at several places in the code. Then, with one change in the macro, the constant is modified for the entire code. Macro definitions may also use formal arguments. Consider, for example, the preprocessor expands the macro

```
#define INC_A_BY(X)    add a, #X

        .
        INC_A_BY(1)
        .
        INC_A_BY(7)
        .
```

as

```
        .
        add    a,#1
        .
        add    a,#7
        .
```

Here, the source code is more readable using the INC_A_BY() macro. Macro definitions may have more than one formal argument. Moreover, the backslash character ('\') may be used to define multi-line macros. The MCS-51 instruction set does not have an instruction to multiply two direct data registers. The following macro may be considered.

```
#define MULREGS(X,Y)    push    acc    \
                        push    b      \
                        mov     a, X   \
                        mov     b, Y   \
                        mul     ab     \
                        mov     X, a   \
                        mov     Y, b   \
                        pop     b      \
                        pop     acc
```

Then, in the source, the macro is simply invoked.

```
        .
        MULREGS(3,4)
        .
        MULREGS(0x35,0x45)
        .
```

It should be clear that the above macros are not single instructions. Each invocation is actually expanded to the nine-line definition. Thus, if the code requires a lot of register multiplications, then perhaps a routine should be used instead of a macro. In this case, a mechanism is needed to transfer arguments to the function. Moreover, the function introduces some run-time overhead, as the call and return instructions need to be executed. On the other hand, the macro implementations add to the size of the source. The trade off is thus between speed and size. Otherwise, both approaches produce well-organized code relatively easy to read.

The preprocessor directives define blocks of source lines that are included or excluded depending on the condition. Consider, for example, the following code segment.

```
#define DEBUG

    .
    .
    .
    lcall putc
#ifdef DEBUG
    lcall print
    db    "value is : ", 0
    mov   a, r0
    lcall prthex
    lcall crlf
#endif
    mov   r0, #1
    .
    .
    .
```

Since the macro DEBUG is defiined, the preprocessor output includes the print code for debugging.

```
    .
    .
    .
    lcall  putc
    lcall  print
    db "value is : ", 0
    mov     a, r0
    lcall  prthex
    lcall  crlf
    mov    r0, #1
    .
    .
    .
```

Now, if you remove the DEBUG macro definition, or comment it out, the preprocessor output excludes the print instructions.

```
    .
    .
    .
    lcall  putc
    mov    r0, #1
    .
```

```
        .
        .
```

This is referred to as conditional assembly or conditional compilation, when used in conjunction with source code for a high-level language. Conditional assembly allows the programmer to place several versions of code in the same source file. The versions actually to be included in the output, and hence, assembled, are then controlled by the macro definitions. Note that preprocessor directives may be nested. The `#else` directive selects an alternative block of code if the macro is not defined. A single `#endif` if used at the end of the block.

```
        .
#ifdef MODE_1
        .
        .
#else
        .
        .
#endif
        .
```

The directive `#ifndef` is similar to `#ifdef`. It selects the following block if the macro is not defined.

CHAPTER 4

INTRODUCTORY ASSEMBLY LANGUAGE EXPERIMENTS

The following experiments introduce the capabilities of the MCS-51 assembly language. These experiments use the on-chip peripherals. Simple input and output devices need to be interfaced to the microcontroller. Some of the simpler programs can be assembled and inspected with the chip simulator. RChipSim51 includes a simulated view of the ports. The program may be run without breakpoints, and the user may interact with the ports by clicking on them. Refer to Appendix A for information about the Reads51 IDE. An evaluation board is needed to carry out the experiments to their full extent. The linker parameters may need to be adjusted depending on the type of evaluation board used. You may start your code in low memory (say 0x100 to leave room for the interrupt vectors) if you use RChipSim51. If you use hardware, such as R31JP, RIC320, R515JC, or RMB-S, with ROM in low memory, specify the CODE parameter to be 0x8000. This way, your code is loaded into RAM and executed under the monitor supervision. Note that the examples are implemented as Reads51 assembly projects. The projects may be downloaded from the web along with the Reads51 IDE. The discussions below often only give relevant segments of the code for brevity.

Most of the experiments can be run on a simple 8051-based system. The User Input Devices (UID) and the User Output Devices (UOD) described in Chapter 7 is a convenient platform to study these experiments. The UIDs and UODs are referred to by their name and number. For example, LED0 is the zero-th LED, and DIP-SW1 is the first DIP (toggle) switch of the UODs. Refer to circuit diagrams in Chapter 7 for more information. There are a few examples that require the enhanced peripherals of the 80C515.

The experiments illustrate digital input/output, timer, counter, and analog-to-digital conversion operations with the 80C515. Fundamental software concepts in using subroutines and interrupt service routines are demonstrated. The experiments use many of the general-purpose subroutines discussed in the previous chapter. For example, the routines `putc`, `getc`, `prthex`, and `puts` provide convenient means to communicate with the host PC. These subroutines are collected into a file UTIL51.ASM. This file may simply be included as a module of the project. However, it is better to compile this file into a library, UTIL51.LIB, and include the library in the projects. This way, the source need not be assembled for each project.

Alternatively, you may use the editor's cut and paste features to add the source code of the required subroutines in your program. Finally, the general-purpose routines may be included on EPROM and called via a table as explained in Chapter 7. In this sense, the routines are considered to be system calls. They serve the same purpose as the BIOS routines do on an IBM compatible PC.

The sample programs given in this chapter may be used as tutorials or as templates for basic software building blocks in more elaborate applications. The experiments discussed in Chapter 5 refer to and use the software building blocks explained and illustrated in this chapter.

4.1 Input/Output Operations

It was mentioned in Section 3.1 that LED 0 can be turned on or off by connecting L0 to PB0 and pressing or releasing push button 0. Now consider the following experiment. Connect L0 to P1.0 and PB0 to P1.1. Assemble the following very short program.

```
        setb    P1.1            ; prepare P1.1 for input
loop:
        mov     C, P1.1         ; read the state of PB0 and
                                ; store in the Carry flag
        mov     P1.0, C         ; transfer PB0 status to LED0
        sjmp    loop            ; loop forever...
```

This program illustrates the Boolean (bit-oriented) capabilities of the MCS-51 family of processors. Notice that all bit transfers use the carry flag. The program also illustrates that digital inputs are sensed only when the port bit is set and the input signal sinks or "pulls down" the port bit (see the microcontroller manufacturer's data book for details on internal port hardware). This program will loop until the RESET button on the board is pressed.

The above program is superfluous since the same task can be accomplished by directly connecting L0 and PB0. Now consider a simple addition to the program.

```
        setb    P1.1            ; prepare P1.1 for input
loop:
        mov     C, P1.1         ; read the state of PB0 and
                                ; store in the Carry flag
        cpl     C               ; complement the carry flag
        mov     P1.0, C         ; transfer complemented PB0 status to L0
        sjmp    loop            ; loop forever...
```

This program will light LED 0 when push button 0 is released. It will turn LED 0 off when push button 0 is pressed. Notice how trivial an addition this is to the software, while no changes are made to the hardware (the beauty of programmable control!).

Moreover, several inputs can be monitored to make a decision whether to light the LED. Let DIP switch 0 (DIP-SW0) be connected to Port 1 bit 2. The following program lights L0 when PB0 is pressed, provided that DIP-SW0 is off. If DIP-SW0 is on, then P1.2 is pulled to ground, and consequently, L0 is lit when PB0 is released.

```
    setb   P1.1          ; prepare P1.1 for input
    setb   P1.2          ; prepare P1.2 for input

loop:
    mov    C, P1.1       ; read the state of PB0, store in the
                         ; Carry flag
; -- to invert or not to invert -- ask P1.2
    jb     P1.2, show
    cpl    C             ; complement the carry flag
show:
    mov    P1.0, C       ; transfer complemented
                         ; PB0 status to L0
    sjmp loop            ; loop forever...
```

Suppose that, several input bits need to be considered to make a decision whether to light L0. For example, let PB1 be connected to Port 1 bit 2 (P1.2). The following program will light L0 if both PB0 and PB1 are pressed.

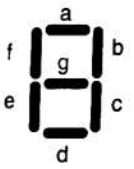
```
    setb   P1.1          ; prepare P1.1 for input
    setb   P1.2          ; prepare P1.2 for input
loop:
    mov    C, P1.1       ; read the state of PB0, store in the
                         ; carry flag
    orl    C, P1.2       ; the logic OR operator
    mov    P1.0, C       ; transfer complemented PB0 or PB1 to L0
    sjmp loop            ; loop forever...
```

Notice that in the above program both PB0 and PB1 need to be pressed so that both P1.1 and P1.2 are at logic level 0. Only in this case will PB0 and PB1 be 0, lighting the LED.

4.2. A Subroutine for Driving Seven-Segment Displays

We next focus our attention to subroutines. Subroutines are useful in implementing repetitive procedures in a program. When the program calls a subroutine the program counter (pointing to the instruction immediately following the call instruction) is stored on stack. The subroutine terminates with a return instruction, which pops an instruction address into the program counter. The program flow thus returns to the calling program.

Standard tasks that are repeated several times during the course of program execution are candidates for subroutines. Consider the task of obtaining a pattern for a seven-segment display. The segments of the display are enumerated by the first seven letters of the alphabet as shown below. The numeral three, for instance, is obtained by lighting segments a, b, c, d, and g.



A hexadecimal value from 0h to 0Fh is to be converted to a display pattern. Such a task is commonly required by many application programs. The following subroutine accomplishes this task.

```
; ================================================
;  subroutine display - display string
;  input     : nibble in accumulator
;  output    : 7-segment pattern in accumulator
;              (acc.0 is segment a, acc.6 is segment g)
;  destroys : a
; ------------------------------------------------
display:
   inc    a
   movc   a, @a+pc
   ret
   db     0C0h          ; 0
   db     0F9h          ; 1
   db     0A4h          ; 2
   db     0B0h          ; 3
   db      99h          ; 4
   db      92h          ; 5
   db      82h          ; 6
   db     0F8h          ; 7
   db      80h          ; 8
   db      90h          ; 9
   db      88h          ; a
   db      83h          ; b
   db     0C6h          ; c
   db     0A1h          ; d
   db      86h          ; e
   db      8Eh          ; f
```

This subroutine adds the value of the accumulator to the program counter and reads the value of this program memory location into the accumulator. The seven-segment-display bit patterns, from the pattern of 0h to that of 0Fh follow the return instruction. The accumulator thus returns the bit pattern. The value in the accumulator is incremented to compensate for the single-byte return instruction.

It is a good idea to have a header for each subroutine announcing the purpose of the routine, the registers used, the information that needs to be passed on to the routine, and the return information, if any. Note that information may be passed to and from a subroutine through registers, stack, or external data memory. The MCS-51 language has two call instructions `LCALL` and `ACALL`, as explained in Chapter 3.

Now let us write a program that calls subroutine `DISPLAY`. Connect the segments of the seven-segment display to Port 1. That is, connect segment a to P1.0, segment b to P1.1, ..., and segment g to P1.6. Either one of the displays, LOW DIGIT or HIGH DIGIT may be used. In addition, connect push button PB0 to bit 0 of Port 1, P1.0. The following program increments the value displayed on the seven-segment display each time PB0 is pressed.

```
; UIOD connections
#define LOW_DIGIT P1    ; output (segment a to P1.0, ...
                        ; segment g to P1.6)
#define PB0 P3.5        ; input
; ------------------------------------------------
#include <sfr51.inc>
    cseg   at 0         ; run using the monitor G8000 command
    setb   PB0          ; prepare bit for input
    mov    b, #0        ; current count
    mov    a, b         ; load a with current count
    lcall  Display      ; get display pattern
    mov    LOW_DIGIT, a ; display the digit
loop:
    jb     PB0, $       ; stay here until PB pressed
    jnb    PB0, $       ; stay here until PB released
    inc    b            ; update count
    anl    b, #0Fh      ; modulus 10h (16)
    mov    a, b         ; put current count in a
    lcall  Display      ; get display pattern
    mov    LOW_DIGIT, a ; display the digit
    sjmp   loop         ; repeat forever...
    end
```

4.3. Serial Communications

Before any serial communications take place, the serial port must be initialized. The following uses Timer 1 to run the serial port at 9600 Baud using an 11.0592MHz crystal.

```
; ===========================================================
; subroutine initSIO
; Initializes the serial port for 9600 Baud using
; Timer 1.  Assumes a crystal frequency of 11.0592MHz.
; -----------------------------------------------------------
initSIO:
   anl   TMOD, #0x0F  ; set counter 1 for auto reload - mode 2
   orl   TMOD, #0x20
   mov   TCON, #0x41  ; run counter 1 and set edge trig ints
   mov   TH1,  #0xFD  ; set counter 1 for 9600 Baud
                      ; xtal=11.0592mhz
   mov   SCON, #0x50  ; set serial control register for 8-bit
                      ; data and mode 1
   ret
```

Note that the first two instructions set the high nibble of TMOD to 2 without altering the low nibble. The first instruction clears the high nibble. This is referred to as masking. The constant 0x0F is said to mask the lower nibble while changing (in this case, clearing) the high nibble, just as masking tape is used to protect areas while painting adjacent ones. The reload value 0xFD would need to change for different Baud rates or the crystal frequencies.

The ASCII routines given in Chapter 3 use the two low-level routines `PUTC` and `GETC` to transmit and receive single characters, respectively.

```
; ===========================================================
; subroutine getc
; Waits for and returns the character received by the
; serial port in the accumulator
;
; input   : nothing
; output  : char in acc
; destroys: nothing
; -----------------------------------------------------------
getc:
   jnb   ri, $        ; wait for char
   clr   ri
   mov   a, sbuf
   ret
```

```
; ================================================================
; subroutine putc
; Transmits the char given in the accumulator and wait
; for transmission completion.
;
; input    : char in acc
; output   : nothing
; destroys: nothing
; ----------------------------------------------------------------
putc:
    mov    sbuf, a
    jnb    ti, $          ; wait for transmission
    clr    ti
    ret
```

The routine PUTC may repeatedly be called to transmit a string. Constant strings are usually placed in code memory. The following routine, puts, assumes that the address of the first byte of the string is in the data pointer. The end of the string is marked by a null character (0).

```
; ================================================================
; subroutine puts
; Transmits a string out the serial port.
; The null-terminated string is expected to be in code memory,
; whose address is in DPTR.
;
; input    : string base in DPTR
; output   : serial port
; destroys: accumulator
; ----------------------------------------------------------------
puts:
    clr    a
    movc   a, @a+dptr
    jz     putsDone
    lcall  putc
    inc    dptr
    sjmp   puts
putsDone:
    ret
```

If the string appears only in one statement in the code, the routine puts becomes a little more involved than necessary. It needs three components: set dptr, call putc, and include the string somewhere in code memory. An alternative is shown below.

```
print:
    pop    dph
    pop    dpl
next:
```

```
    clr   a
    movc  a, @a+dptr
    inc   dptr
    jz    prtDone
    lcall putc
    sjmp  next
prtDone:
    push  dpl
    push  dph
    ret
```

Here the null-terminated string is expected to immediately follow the call to `print`. The routine pops the top of stack into the data pointer. The return address on top of the stack is thus the base of the string. The routine prints the characters of the string until the null character is encountered. The data pointer now holds the address of the byte immediately following the null character. This address is pushed from the data pointer to the top of stack. A return instruction continues code execution from immediately after the string.

These routines, along with the ASCII conversion routines presented in Chapter 3 are placed in the library UTIL51.LIB. Application programs may simply use these routines by linking with the library. The program below sends a string to a terminal. You may run this program and observe the string in the Reads51 TTY window.

```
#include <util51.inc>

    cseg  at 0
    lcall InitSIO       ; initialize serial port (9600 Baud)
loop:
    mov   dptr, #Msg
    lcall puts          ; print msg
    lcall getc          ; wait for a char
    lcall print         ; print dots. . .
    db    ". . . . ", 0
    lcall crlf          ; new line
    sjmp  loop          ; repeat forever...

Msg:
    db "Hello world...", 0
    end
```

The message is defined by a "db" pseudo-op. Note that the string terminates with the null character. The string resides in the code segment, initiated by the "`cseg`" directive. The project must include the library UTIL51 so that the routines `InitSIO`, `puts`, `getc`, and `crlf` may be linked. The include file `<util51.inc>` placed in the ".\include" directory contains the "extern" definitions for the routines in the library.

The routines peekc, putc, getc, puts, prthex, prsphx are especially useful in writing application programs that communicate with the host through the RS-232 port. Let us consider the following program. Here, the count that is displayed on the seven-segment display is also echoed to the host screen. Observe how the utility routines initSIO, puts, prthex, and crlf are used.

```
#define LOW_DIGIT P1    ; output (segment a to P1.0, ...
                        ; segment g to P1.6)
#define PB0 P3.5        ; input

#include <sfr51.inc>
#include <util51.inc>

    cseg   at 0
    lcall  InitSIO      ; initialize serial port (9600 Baud)
    setb   PB0          ; prepare P3.5 for input
    mov    b, #0        ; current count
    mov    a, b         ; load a with current count
    lcall  display      ; get display pattern
    mov    LOW_DIGIT, a ; display the digit
    mov    dptr, #Msg
    lcall  puts         ; send message to host
    mov    a, b         ; get current count
    lcall  prthex       ; echo count to host
    lcall  crlf         ; start a new line
loop:
    jb     PB0, $       ; stay here until PB pressed
    jnb    PB0, $       ; stay here until PB released
    inc    b            ; update count
    anl    b, #0Fh      ; modulus 10h (16)
    mov    a, b         ; put current count in a
    lcall  display      ; get display pattern
    mov    LOW_DIGIT, a ; display the digit
    mov    dptr, #Msg
    lcall  puts         ; send message to host
    mov    a, b         ; get current count
    lcall  prthex       ; echo count to host
    lcall  crlf         ; start a new line
    sjmp   loop         ; repeat forever

Msg:
    db     "current count is ",0
    end                 ; end of program
```

You may peek at the serial port to see if a termination character, say CTRL-C, is received. CTRL-C is universally used as a break command. The routine peekc given in Chapter 3 is used below for this purpose. If a termination character is found, a jump to the reset vector (0) effectively terminates the program by returning to the

monitor... The break routine should be called in the loop that waits for the push button.

```
loop:
   lcall  peekc          ; termination char ?
   cjne   a, #CTRL_C, continue
   ljmp   0              ; return to monitor
continue:
   jb     PB0, $         ; stay here until PB pressed
   jb     PB0, $         ; stay here until PB pressed
   .
   .
```

Note that the loop is the portion of code that will consume almost all of the CPU time. Once a pushbutton is pressed and released, it takes very little CPU time to finish the tasks and once again start polling the push button.

4.4. Interrupt Service Routines

We discussed and demonstrated the usefulness of subroutines in the previous examples. When a subroutine is called, the return address is stored on stack. Upon a return instruction, the program flow continues from where the subroutine was called. Note that the exact time of such a call is determined by the calling program. In many control applications, the controller needs to service an external request as soon as possible. If the controller is busy with a lengthy but relatively low priority task, then an external request may needlessly be delayed.

As mentioned in the previous chapters, an *Interrupt Service Routine* (ISR) is a special type of subroutine. When invoked, the program branches to the ISR, and upon termination of the ISR, the program resumes its flow from where it was interrupted. The difference is that the program does not explicitly call the ISR. Thus, the program does not control when exactly the ISR is invoked. Rather, the ISR is invoked as a response to an external signal. Then, an implicit LCALL is performed by the CPU on recognition of the interrupt. This phenomenon partitions segments of the programs into two classes: the foreground tasks and the background tasks.

The foreground tasks are those under program control, i.e., the main program and all the subroutines explicitly called from the main program. The background tasks are segments of the program that are not called by the main program, but are invoked by signals. This arrangement gives rise to many desirable modes of operation.

Firstly, since a task can be coded as an ISR and tied to an external signal, it simplifies real-time control. Real-time control loosely refers to servicing external requests without delay. This arrangement also allows program segments to be coded and run independently. Foreground tasks and ISRs can be developed, tested, and then

combined. This enforces structured programming, and simplifies the development of large-scale complex programs.

The 8051 family of processors has a fairly sophisticated interrupt control logic. Both external signals and internal signals (conditions) may invoke interrupts. An example of the latter is a timer overflow. When the timer overflows, it may be programmed to issue an interrupt.

The starting address of an ISR is called an interrupt vector. The 8051 family of processors has fixed interrupt vectors as shown below. Refer to Chapter 1 for more information.

Interrupt Name	ISR Vector	Default Priority
IE0	3H	1 (Highest)
TF0	BH	2
IE1	13H	3
TF1	1BH	4
RI-or-TI	23H	5 (Lowest)

We start our demonstrations with a very simple example. The main program, i.e., the foreground task, implements a delay loop to waste some time and then updates the digit displayed on the seven-segment display. The foreground task is fashioned after the examples in the previous section. There is one ISR, tied to the 1-st external interrupt, INT1. When the INT1 input experiences a 1-to-0 transition, the ISR is invoked. The INT1 input (the alternate function of port bit P3.3) is connected to push button PB0. The ISR toggles port bit P3.5, which is connected to LED0. Thus, as the foreground task periodically updates the digit displayed, pressing PB0 toggles LED0 (if the LED is off, pressing the button turns it on, and if it is on, pressing the button turns the LED off). The remarkable feature is that the LED toggling routine is never called by the main program. Things seem to happen in parallel. Of course, in reality, the main program is interrupted by the ISR.

```
; External interrupt 1 is enabled and set up to respond
; to 1-to-0 transitions.  Each time an interrupt is
; received, LED0 is toggled. The foreground task displays
; digits 0 to F on LOW DIGIT.  Two nested loops are
; used to delay between updates of LOW DIGIT.  Note
; that there is no explicit calls to the interrupt
; service routine ISR.  ISR is invoked upon  external
; signal.
; ================================================
; UIOD connections
#define LOW_DIGIT  P1  ; output (segment a to P1.0,...
                       ; segment g to P1.6)
#define L0  P3.5       ; output bit
```

```
; ------------------------------------------------
#include <sfr51.inc>
#include <util51.inc>

   cseg  at 0          ; run by swapping RAM into
   ljmp  Start         ; low memory after reset
   end

   cseg  at 0x13       ; external interrupt 1 vector
   ljmp  ISR
   end

   cseg  at 100h       ; program starts here
Start:
   setb  IT1           ; set external interrupt
                       ; to be transition sensitive
   setb  EX1           ; enable external interrupt 0
   setb  EA            ; master interrupt enable flag

   mov   b, #0         ; current count
   setb  L0
   mov   r6, #0        ; initialize delay loop...
   mov   r7, #0        ; ... parameters
loop:
   djnz  r6, loop      ; waste a little time
   djnz  r7, loop

   mov   a, b          ; get current count
   lcall display       ; get display pattern
   mov   LOW_DIGIT, a  ; display on LOW DIGIT
   inc   b             ; update current count
   anl   b, #7         ; modulus 8
   lcall break         ; terminate if CTL-C pending
   sjmp  loop          ; repeat forever...

; ================================================
;  subroutine ISR
;  interrupt service routine for external interrupt 1
;  input    : external interrupt 1
;  output   : toggles the LED0.
;  destroys : nothing
; ------------------------------------------------
ISR:
    cpl   L0                ; blink LED0
    reti                    ; return
; ------------------------------------------------
    end                     ; end of program
```

You may run this program in the RChipSim51 and observe its behavior. If you are using one an evaluation board, such as the one given in Chapter 7, or one of the Rigel boards, the code must be placed in low memory, or rather, switched into low memory after download. More specifically, the vector for external interrupt 1 is set to be 0x13. Normally, low memory is occupied by ROM (or EPROM). The interrupt vectors cannot be altered (redirected) unless a new ROM is programmed. For experimentation purposes, it is highly desirable to have access to the interrupt vectors. That is, it is desirable to direct interrupts to their respective service routines. This requires placing a long jump instruction at the low address vectors, at address 0x13, in this case. In the circuit diagrams given in Chapter 7, either the RAM or the EPROM may occupy the low block of memory, determined by the state of the double-pole-double-throw switch. Note that the program above has an origin of 0. At location 0x13, a jump instruction redirects program flow to the interrupt service routine. In order to run the program, the hardware is set up with the monitor program in low memory. The program given above is assembled and downloaded into RAM. At this time, the RAM occupies the high block of memory. The download routine always sets the most significant bit of the address of the downloaded program. Once the program is downloaded into RAM, before it is run, the memory map must be changed so that the program starts from address 0. Simply press and hold the RESET button while toggling the double pole double throw switch. Note that during this switch, the microcontroller does not execute any instructions. Once RAM is selected to occupy the low block, releasing the RESET button starts execution from address 0, in this case, executing the application program Int1.asm. The monitor program is invoked by toggling the switch and pressing RESET.

4.4.1. Using an Interrupt Vector Table to Redirect the Interrupts

The scheme described above is useful in testing the application program at low addresses. Once the application program is fully debugged, it may be programmed into EPROM. Then, the application program is invoked upon the power-on-reset cycle. An alternative is to redirect all the interrupts to RAM to begin with. With this scheme, the monitor program always occupies low memory. This is the approach taken with RROS, the monitor program used on Rigel boards. The term redirect refers to a jump instruction placed at the fixed ISR start addresses. The interrupt causes a branching operation to the low-memory address, where the instruction `ljmp 0FFxxh` is found. The low address byte of the jump instruction depends on the interrupt source. At location 0FFxxh the application program may place another jump instruction, this time to the start of the ISR. The jump instructions at high memory locations 0FFxxh are referred to as the interrupt vector table. The motivation to redirect the interrupt vectors to high memory is that high memory is occupied by RAM, and that jump instructions may be placed in high memory (0FFxxh) in run time. This arrangement allows installing and removing ISRs by software.

The system call SETINTVEC modifies the interrupt vector table so that interrupt source, from 0 to 11, indicated by the value of the accumulator (A), is directed to the interrupt service routine whose starting address is held in DPTR. The routine is given as part of MINMON in Chapter 7. Except for the two registers A and DPTR, and the PSW, `setintvec` does not affect any registers. Note that the 80C515 processor is an enhanced version of the 8051 family of processors. It contains 12 interrupt sources as listed below. The list includes the 80C515 interrupts.

source number	designation	description
0	int0	external interrupt 0
1	tint0	timer 0 overflow interrupt
2	int1	external interrupt 1
3	tint1	timer 1 overflow interrupt
4	sint	serial port interrupt
5	exint	timer 2 overflow interrupt
6	adcint	analog-to-digital converter
7	int2	external interrupt 2
8	int3	external interrupt 3
9	int4	external interrupt 4
10	int5	external interrupt 5
11	int6	external interrupt 6

Notice that the source numbers follow the default priorities of the interrupts. Refer to Chapter 7 for more information on the interrupt vector table and redirecting the interrupts.

Modify the above example so that the beginning of the code is as follows.

```
    cseg   at 8000h      ; use the G8000 command to run
Start:
    mov    a, #2         ; INT1 is the 2-nd interrupt source
    mov    dptr, #ISR    ; set up dptr for a call to setintvec
    lcall  setintvec     ; install the ISR for INT0

    setb   IT1           ; set external interrupt
                         ; to be transition sensitive
    setb   EX1           ; enable external interrupt 0
    setb   EA            ; master interrupt enable flag
```

Other than calling the system call `setintvec` and starting the program from address 8000h, the code is the same as before. In the remainder of this book, the examples are written to start at low memory. This is convenient for RChipSim51 as well as for the boards. Simply use the switch to swap RAM into low memory to run the

programs. The redirection of interrupts with `setintvec` is necessary only if you use a Rigel board and would like to single step on the board rather than RChipSim51. Moreover, placing the application code in low memory is more natural, especially if the code is ultimately to be placed in ROM and run in an embedded control application.

4.4.2. Using Multiple Interrupts

The next example is a straightforward extension of the one given in the preceding section. In addition to the interrupt source INT1, INT0 is also implemented as an interrupt source. Both interrupts are triggered by 1-to-0 transitions. INT0 and INT1 are tied to pushbuttons PB0 and PB1. The routines ISR0 and ISR1 toggle the LEDs connected to P3.4 and P3.5. Thus, as the digits are updated on the display by the foreground task, pressing PB0 or PB1 toggles LED0 and LED1, respectively. Notice that in this example, there are three (3) independent program segments: main (foreground), ISR0, and ISR1. None of these segments explicitly calls the others.

```
; UIOD connections
#define LOW_DIGIT P1    ; output (segment a to P1.0,...
                        ; segment g to P1.6)
#define L0        P3.4  ; output bit
#define L1        P3.5  ; output bit
; ------------------------------------------------
#include <sfr51.inc>
#include <util51.inc>

    cseg   at 0          ; run by swapping RAM into
    ljmp   Start         ; low memory after reset
    end

    cseg   at 0x03       ; external interrupt 0 vector
    ljmp   ISR_0
    end

    cseg   at 0x13       ; external interrupt 1 vector
    ljmp   ISR_1
    end

    cseg   at 100h       ; program starts here
Start:
    setb   IT0           ; make interrupt transition sensitive
    setb   EX0           ; enable external interrupt 0
    setb   IT1           ; make interrupt transition sensitive
    setb   EX1           ; enable external interrupt 1
    setb   EA            ; master interrupt enable flag

    mov    b, #0         ; current count
    setb   L0
    mov    r6, #0        ; initialize delay loop...
```

```
    mov   r7, #0          ; ... parameters
loop:
    djnz  r6, loop        ; waste a little time
    djnz  r7, loop

    mov   a, b            ; get current count
    lcall display         ; get display pattern
    mov   LOW_DIGIT, a    ; display on LOW DIGIT
    inc   b               ; update current count
    anl   b, #7           ; modulus 8
    lcall break           ; terminate if CTL-C pending
    sjmp  loop            ; repeat forever...

; ================================================
;  subroutines ISR_0 and ISR_1
;  interrupt service routine for external interrupts
;  0 and 1
;  input    : external interrupt 0 or 1
;  output   : toggles outputs each time an external
;             interrupt is received.
;  destroys : nothing
; ------------------------------------------------
ISR_0:
    cpl   L0              ; blink LED0
    reti                  ; return
ISR_1:
    cpl   L1              ; blink LED1
    reti                  ; return
; ------------------------------------------------
    end
```

Interrupt service routines are perhaps the most important building blocks of embedded control software. A good understanding of interrupts and interrupt service routines is essential to write real-time applications software. Several examples of such applications are discussed in the next chapter. An example in this chapter illustrates how the 80C515 analog-to-digital converter may be operated in an interrupt-driven mode.

4.5. Counter Operations

The 8051 family of processors has 2 timers Timer 0 and Timer 1. These two timers are identical in their operation. The 8052 as well as the 80C515 has a third timer, Timer 2. These can be used as timers, incremented by the system clock, or as counters, incremented by an external signal. Review Chapter 1, especially Section 1.2.2.3. for more information on the timer and counter operations. This section gives short programs to illustrate how Timer 0 works.

First consider the counter example given below. Timer 0 is used as a counter in mode 0 or in mode 1. The input is thus the signal applied to T0. Note that input T0 is the alternate function of port bit P3.4. The counter is incremented upon a 1 to 0 transition. First set T0 to be an input port.

```
setb   T0              ; prepare T0 for input
```

Then program Timer 0 by placing the corresponding control word into the SFR TMOD. Its bits determine the operating mode of both Timers 0 and 1. Setting GATE# to 0 and TR0 to 1 activates the timer. TR0 is a control bit in the TCON. The timer operates as a timer if C/T# is 0 and as a counter if C/T# is 1. The two bits M1 and M0 determine the mode. M1 is the more significant bit.

Timer 1 is identical to Timer 0. It is programmed by the corresponding bits in the high nibble of the control register TMOD, and the control bits TR1 and TF1 of TCON. The bits of TMOD and TCON are shown below.

TMOD

Timer/Counter 1				Timer/Counter 0			
GATE#	C/T#	M1	M0	GATE#	C/T#	M1	M0

TCON

TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0

The instruction below disables the GATE# of counter 1 while initiating counter 0 in mode 0. Timer t will run in mode 0 if TR1 is set by software and P3.3 is at logic level 1. Timer 0 is ready to be activated by TR0 since its GATE# is 0. The C/T# flag is 1 indicating that the device will operate as a counter. Moreover, M1=M0=0, selecting mode 0.

```
mov    TMOD, #84h    ; configure timer/counter 0
                     ; counter in mode 0
```

The final initiation step is to activate the timer by turning on TR0.

```
setb   TR0           ; turn timer/counter 0 on
```

The program displays the current count on the seven-segment display LOW DIGIT of the UIODs. The seven segments of LOW DIGIT are driven by Port 1 bits P1.0 to P1.6. P1.0 is connected to segment a, P1.1 to segment b, ..., and P1.6 to segment g. The utility routine `display` is used to find the seven-segment display patterns. The current count is stored in the b register. The count value stored in TL0 is compared to that in b. If not equal, b is incremented and the new count is displayed. Note that b is reset to 0 after it reaches 0Fh and there is a new count. The program is given below.

```
#define LOW_DIGIT    P1    ; output (segment a to P1.0, ...
;                                    segment g to P1.6)
#define PB0          P3.4 ; (T0) counter input
; ------------------------------------------------
#include <sfr51.inc>
#include <util51.inc>

   cseg  at 0

   setb  T0            ; prepare T0 for input
; --- use only one of the following instructions. --------
; --- i.e., counter in  either mode 0 or mode 1 -------
   anl   TMOD, #0xF0  ; timer 0 in counter mode 1
   orl   TMOD, #0x05
; ------------------------------------------------
   setb  TR0           ; turn timer 0 on
   mov   b, #0         ; current count

   mov   a, b          ; load a with current count
   lcall display       ; get display pattern
   mov   LOW_DIGIT, a ; display the digit
   mov   a, b          ; get current count
   lcall prthex        ; echo count to host
   lcall crlf          ; new line on host

loop:
   lcall break         ; terminate if CTL-C pending
   mov   a, TL0        ; get counter value
   anl   a, #0Fh       ; modulus 10h
   xrl   a, b          ; compare with current count
   jz    loop          ; repeat if equal

   inc   b             ; else increment current count
   anl   b, #0Fh       ; modulus 10h
   mov   a, b          ; load a with current count
   lcall display       ; get display pattern
   mov   LOW_DIGIT, a ; display the digit
   mov   a, b          ; get current count
   lcall prthex        ; echo count to host
   lcall crlf          ; new line on host
   sjmp  loop          ; repeat forever...

   end                 ; end of program
```

The next program is similar to the previous one, except that the counter is operated in mode 2. In this mode, TL0 is an 8-bit counter that is reloaded with the value in TH0

when it rolls over. The contents of the two count registers TL0 and TH0 are displayed after each increment.

```
#define LOW_DIGIT    P1    ; output (segment a to P1.0, ...
;                                    segment g to P1.6)
#define PB0          P3.4 ; (T0) counter input

#include <sfr51.inc>
#include <util51.inc>

    cseg   at 0
    setb   T0            ; prepare T0 for input
    anl    TMOD, #0xF0   ; configure timer 0 counter in mode 2
    orl    TMOD, #0x06
    mov    TL0, #0xFF
    mov    TH0, #0xF5    ; reload value (try other values)
    setb   TR0           ; turn timer/counter 0 on
    mov    r7, TL0       ; old timer value in r7

    lcall  ShowRegs
loop:
    lcall  break         ; terminate if CTL-C pending
    mov    a, TL0        ; get counter value
    xrl    a, r7         ; compare with current count
    jz     loop          ; repeat if equal
    mov    r7, TL0       ; update register
    lcall  ShowRegs
    sjmp   loop          ; repeat forever...

ShowRegs:
    mov    a, TL0        ; load acc with current count
    anl    a, #0x0F      ; modulus 16
    mov    b, a
    lcall  display       ; get display pattern
    mov    LOW_DIGIT, a  ; display the digit
    mov    a, b          ; recall current count
    lcall  prthex        ; echo count to host
    mov    a, TH0        ; load a with counter high byte
    lcall  prsphx        ; display on host
    mov    a, TL0        ; load a with counter low byte
    lcall  prsphx        ; display on host
    lcall  crlf          ; new line on host
    ret

    end                  ; end of program
```

The local routine `ShowRegs` displays the contents of the timer registers.

4.6. Timer Operations

Timers 0 and 1 are identical in their operation. Each timer is configured by placing the appropriate control word in the register TMOD. The C/T# flag must be cleared (0) to select the timer function. The timer internal count registers are incremented every machine cycle (1/12 of the crystal frequency). The timers can be turned off by clearing the corresponding control bit TRi, i=0 or 1, of TCON. Provided that TRi is set, the timers are enabled either by clearing the corresponding GATE#, or by making the corresponding port pin (P3.2 for Timer 0 and P3.3 for Timer 1) high. The four operation modes are still valid.

Consider the following program, which is fashioned after the counter examples. The seven-segment display is updated periodically following the timer output. This program initialized timer 0 to be a timer operating in mode 0, i.e., a 13-bit timer. When the timer overflows, the hardware sets the flag TF0. The program spends most of its time checking if the timer overflow flag TF0 is set. Notice that with a 12Mhz crystal frequency, the machine cycles are one microsecond. Thus, the timer overflows every 8192 microseconds. This is much too rapid to increment the display. A software counter is implemented. Register r7 is used to keep the software count. Each time Timer 0 overflows, the software counter is decremented. When the software counter, r7, reaches zero (0), the displayed digit is updated. The current display count is stored in the b register, as in the previous examples. With a software count value of 122 (7Ah), the digit is updated every 8192*122=999040 microseconds, or about once every second. This computation, of course, does not take into account the overhead, that is, the time spent by the controller executing instructions. Note that the timer overflow flag TF0 is cleared by software after each timer overflow.

```
; UIOD connections
;
#define LOW_DIGIT     P1    ; output (segment a to P1.0, ...
;                                     segment g to P1.6)
#define PB0           P3.4 ; (T0) counter input
; ------------------------------------------------

#define SCOUNT 0x7A

#include <sfr51.inc>
#include <util51.inc>

   cseg   at 0
   anl    TMOD, #0xF0   ; run timer 0 in mode 0
   setb   TR0           ; turn timer 0 on
   mov    b, #0         ; current display count
   mov    r7, #SCOUNT   ; software counter

   lcall ShowRegs
```

```
loop:
    lcall  break          ; terminate if CTL-C pending
    jnb    TF0, loop      ; repeat until Timer 0 times out
    clr    TF0            ; clear timer overflow flag
    djnz   r7, loop       ; decrement r7, repeat until zero
    mov    r7, #SCOUNT    ; reset software counter
    inc    b              ; when software counter overflows
                          ; increment current count
    anl    b, #0Fh        ; modulus 16
    lcall  ShowRegs
    sjmp loop             ; repeat forever...

ShowRegs:
    mov    a, b           ; load a with current count
    lcall  display        ; get display pattern
    mov    LOW_DIGIT, a   ; display the digit
    mov    a, b           ; get current count
    lcall  prthex         ; echo count to host
    lcall  crlf           ; new line on host
    ret

    end
```

The next example implements Timer 0 in mode 1, a 16-bit timer. Note that in mode 1, with a 12Mhz crystal frequency, the timer overflows every 65536 microseconds. Register r7 is implemented as a software counter as in the previous example. Register r7 is decremented every time Timer 0 overflows. If r7 starts with a value of 15, it reaches zero (0) in about 15*65536=983040 microseconds, or about once per second. The code is the same as before except for the definition of SCOUNT and the initialization of timer 0. Change the first few lines of code as follows.

```
#define SCOUNT 0x7A

    cseg   at 0
    anl    TMOD, #0xF0   ; run timer 0 in mode 1
    orl    TMOD, #0x01
```

The final example initializes Timer 0 in mode 2, an 8-bit timer with auto-reload at overflow. The count stored in TL0 is incremented every machine cycle. When TL0 overflows, its value is set to that stored in TH0. The counter register TL0 is then continued to be incremented. The longest possible time between timer overflows is 256 microseconds with a 12Mhz crystal frequency. Even with a single-byte software counter, this period is far too short. Thus a two-byte software counter is implemented. Two registers, r6 and r7, are used to store the software counter low byte and high byte respectively. The low byte is decremented upon Timer 0 overflow. When the low byte reaches zero (r6=0), the software counter high byte is decremented. The software counters are reloaded with constant values SCOUNT_L=0 and

SCOUNT_H=20h=32. With these values the software counter reaches 0 after 8192 timer overflows. Timer 0 reload value is set to 86h=100h-7Ah. It takes the timer 7A=122 microseconds to overflow. Thus, the software timer reaches zero in 8192*122=999040 microseconds, or about once per second.

Note that several possible values for TH0, SCOUNT, and SCOUNT would give the same period between display updates. Again, register b stores the current display value.

```
#define LOW_DIGIT     P1    ; output (segment a to P1.0,
;                                     segment g to P1.6)
#define PB0           P3.4 ; (T0) counter input
; ------------------------------------------------
#define SCOUNT 0x0E

#include <sfr51.inc>
#include <util51.inc>

#define SCOUNT_L 0x00          ; software counter reload low byte
#define SCOUNT_H 0x20          ; software counter reload high byte
#define RELOAD   0x86          ; TL0 auto reload value in TH0

   cseg  at 0
   anl   TMOD, #0xF0           ; configure timer 0 timer in mode 1
   orl   TMOD, #0x02
   mov   TH0, #RELOAD          ; timer reload value
   setb  TR0                   ; turn timer on
   mov   b, #0                 ; current count
   mov   r6, #SCOUNT_L         ; software counter low byte
   mov   r7, #SCOUNT_H         ; software counter high byte

   lcall ShowRegs

loop:
   lcall break                 ; terminate if CTL-C pending
   jnb   TF0, loop             ; repeat until Timer 0 times out
   clr   TF0                   ; clear timer overflow flag
   djnz  r6, loop              ; decrement software counter
                               ; low byte, repeat until zero
   mov   r0, #SCOUNT_L         ; reset software counter
   djnz  r7, loop              ; decrement software counter high
                               ; byte, repeat until zero
   mov   r7, #SCOUNT_H         ; reset software counter

   inc   b                     ; when software counter overflows
                               ; increment current count
   anl   b, #0Fh               ; modulus 16
```

```
    lcall ShowRegs
    sjmp  loop                  ; repeat forever...

ShowRegs:
    mov   a, b                  ; load a with current count
    lcall display               ; get display pattern
    mov   LOW_DIGIT, a          ; display the digit
    mov   a, b                  ; get current count
    lcall prthex                ; echo count to host
    lcall crlf                  ; new line on host
    ret

    end
```

The most appropriate timer mode depends on the application. Several applications in the next chapter illustrate the use of the timer modes. The reload mode, for example, is convenient to produce Pulse Width Modulation (PWM) signals.

4.7. Using the PTRA Unit Compare and Capture Functions

The 80C515 Programmable Timer/Counter Register Array (PTRA) was discussed in Chapter 1. The PTRA unit is built around Timer 2, but it is not compatible with Timer 2 of the 8052. The PTRA unit is used in all versions of the 515 core, including the 80C515A and the C515C. This section gives four examples using the PTRA unit. The first example illustrates how to generate signals with a given period and duty cycle using the PTRA compare unit. The following examples inspect an external TTL-level signal. The attributes of the signal that are measured are the measuring pulse width, the period, the duty cycle, and the frequency. A detailed technical description of the PTRA unit may be found in the manufacturer's data sheet.

4.7.1. Pulse Train Generation with the PTRA Unit

The C515/C517 Timer 2 compare registers are used to generate pulse trains. More specifically, the frequency and duty cycle of a square wave output is selected by programming the Programmable Timer/Counter Register Array (PTRA). The CPU is involved in setting up and controlling the PTRA unit. Otherwise, the signal is generated without any overhead to the CPU. This makes the PTRA unit a highly efficient and precise pulse train generator. The pulse train signal may be regulated by a low-pass filter to generate a voltage proportional to the duty cycle. This scheme is often used with microcontrollers as a digital-to-analog conversion technique.

One can also generate pulse trains using the earlier members of the 8051 family, such as the standard 8051 or the 8052. A simple setting and clearing of a port bit with time delays in between to control high and low times of the signal generates a pulse train. In the simplest form, a timer is run and the CPU periodically checks the current

count to detect time outs. Such time outs may be recorded as flags implemented in software. However, this technique uses CPU time as the CPU is continuously involved in the setting and clearing of the port bit and polling the time overflow flags. As an improvement, the timer may be run in an interrupt driven fashion. The CPU is then relieved from polling the timer. The timer interrupt signals the end of the high or low period, and the CPU simply toggles the port pin and repeats the process. Due to the interrupt latency, which is dependent on the instruction being executed at the time the interrupt occurs, the high and low times are subject to variation. As the high and low times are generated repeatedly in a software loop, the pulse train displays jittery edges. Such variability may be tolerable in some applications, especially if the high and low times are relatively long relative to the instruction execution times (1/12th of the crystal frequency for single cycle instructions). The PTRA unit handles signal high and low times in hardware, and thus the resulting pulse train is jitter-free and of precisely the prescribed frequency and duty cycle. In this respect, the PTRA unit is clearly the superior method of generating pulse trains.

The approach relies on running Timer 2 in the reload mode and comparing its count to CCH1:CCL1. In its compare mode 0, port pin P1.1 is set when a match occurs, and cleared when Timer 2 overflows. Timer 2 operation is controlled by the CFR T2CON. The reload mode is selected by T2R1=1 and T2R0=0. The reload value is placed in registers CRCH:CRCL. Compare mode 0 is selected by T2CM=0. There are four capture/compare units, each controlled by a two-bit field of CCEN. Bits 3 and 2 are set to 1 and 0, respectively, to configure CC1 to run in the compare mode.

The reload value determines the period of the signal. The larger the reload value, the less time to timer overflow, and hence the shorter the period, or the higher the frequency. If the reload value is 0, with a 12MHz crystal, Timer 2 overflows every 65536 microseconds, and if FFFFh, it overflows every microsecond. The example given below sets the reload value to 0xF000, so the timer overflows every 0x1000 (4096) microseconds, or approximately 250 times a second. The Compare/Capture registers CCH1:CCL1 hold the compare value. With a reload value of 0xF000, the compare value should be between 0xF000 and 0xFFFF. A compare value of 0xF000 corresponds to a duty cycle of 100%, and 0xFFFF to 0% (actually 1/4096, since it takes one cycle to overflow). In the example below, the PWM output is used to drive an LED by sinking current. Thus, the duty cycle is inverted, i.e. 0% for a compare value of 0xF000, and 100% for 0xFFFF.

The demonstration program asks the user to input a single hexadecimal number that is OR-ed with 0xF0 and placed as the compare register high byte. The compare register low byte is kept 0. Note that, if a large number is input, the PWM output will stay low for a longer period. If the output P1.1 is connected to an LED, the LED will be brighter for larger reload values.

```
;  Pulse Width Modulation using Timer 2 Compare
;
; UIOD connections
;  P1.1 is the PWM signal output.
;  Connect P1.1 to an LED and observe its brightness
;
; bytes:
#define PeriodL     0     ; period low  byte (reload value)
#define PeriodH     F0h   ; period high byte (reload value)
#define OnPeriodL   0     ; on period low byte
#define OnPeriodH   F8h   ; on period high byte
; ------------------------------------------------
; --- port definitions ---------------------------
; output ports
#define PWM P1.0
; ------------------------------------------------

#include <sfr515.inc>
#include <util51.inc>

; --- reset vector ---
    cseg   at 0
    ljmp   main
    end

; --- foreground routine ---
    cseg at 0x100
main:

; start 535 serial port
    setb   BD                  ; enable internal Baud rate generator
    mov    SCON, #50h          ; 8 bit data, mode 1
    mov    PCON, #80h          ; double the Baud rate

    mov    CRCL, #PeriodL      ; initialize reload values...
    mov    CRCH, #PeriodH      ; ... (determines the period)

    mov    CCL1, #OnPeriodL    ; initialize compare values...
    mov    CCH1, #OnPeriodH    ; ... (determines the duty cycle)

    mov    CCEN, #00001000b    ; enable compare 1 (CC1)
    mov    T2CON, #00010000b   ; initialize Timer 2
           ;  T2I1=0 and T2I0=0 : timer off
           ;  T2CM=0            : compare mode 0
           ;  T2R1=1 and T2R0=0 : auto reloads
           ;  I3FR=0 and I2FR=0 : no interrupts
           ;  T2PS=0            : no prescalar

    setb   T2I0                ; start Timer 2
```

```
loop:                          ; this loop is the foreground task
   lcall print
   db CR, LF, "input duty cycle [0-F] or 'q' to quit : ",0
getNum:
   lcall getc                  ; press any key to continue...
   cjne  a, #'q', noQuit
   lcall print                 ; print message
   db    CR, LF, "Program terminates...", CR, LF, 0
   sjmp  $                     ; absorbing state

noQuit:
   lcall ascbin                ; convert to a (binary) number
   cjne  a,#0xFF, validNum     ; abort if char out of range
   sjmp  getNum
validNum:
   mov   b, a                  ; store character in b register
   lcall prthex                ; echo character
   mov   a, b                  ; recall character
   orl   a, #0xF0
   mov   CCH1, a
   sjmp  loop

   end
```

The PWM period may be changed by changing the reload value. The optimal PWM frequency depends on the specific load. The lower the frequency, the choppier the response. On the other hand, higher PWM frequencies cause excessive heating of inductive loads. Typical DC motor control applications use PWM frequencies in the 100Hz to 1000Hz range.

4.7.2. Pulse Width Modulation

Pulse train generation with a fixed frequency and duty cycle, as described above, can be extended to vary the duty cycle from period to period, or once every n periods. Again, we use Timer 2 in mode 0, as discussed in the previous section. Two events may be used to generate interrupts.

i) When Timer 2 overflows, an interrupt may be generated.
ii) When the compare value in one of the CCx or CRC registers matched the current Timer 2 count, an interrupt can be generated. This event was not used to generate interrupts in the previous example.

Both of these events can be used to generate interrupts to reprogram the compare value in one of the CCx or CRC registers. Once the compare value is changed, the high time generated in the following timer period will differ from the previous timer period. One can generate a pulse train with varying high times, and thus varying duty

cycle, by reprogramming the compare values. This corresponds to modulating the pulse width.

As mentioned above, this pulse train may be regulated through a low-pass filter to generate an analog signal. The varying pulse width will then generate varying voltage levels of the analog signal at the output of the low-pass filter. For example, it is quite straightforward to synthesize a sine-wave signal with this pulse-width modulation scheme. The signal, when filtered, should be sufficient to be applied to the winding of a DC motor. Sometimes, the winding itself can act as part of the low-pass filter. Note that the inductance of the winding as well as the inertia of the rotor contributes to the low pass characteristics of the filter.

The following figures illustrate the output voltage as a function of the duty cycle, P/T. As the duty cycle increases, so does the effective (average) DC output.

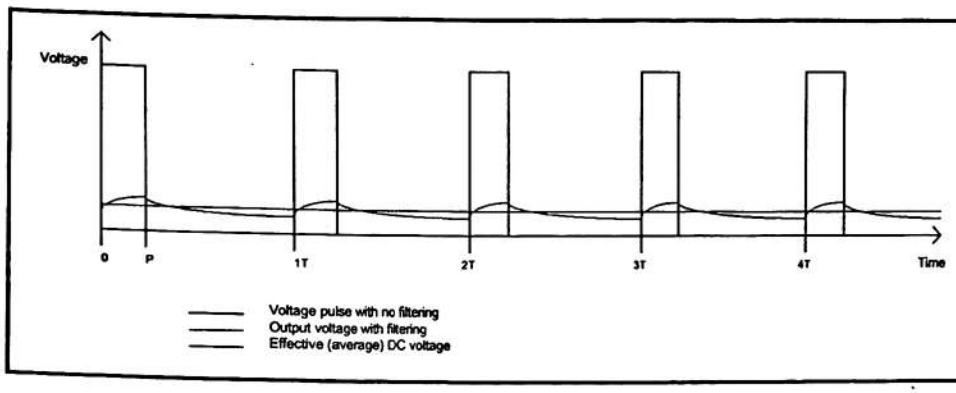


Figure 4.1. PWM output with a low duty cycle.

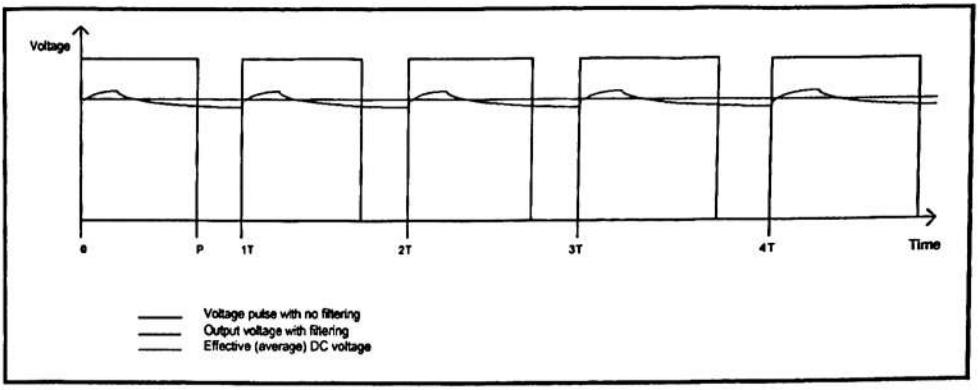


Figure 4.2. PWM output with a high duty cycle.

4.7.3. Pulse Width and Period Measurement

Consider an external TTL-level square wave signal. The signal is connected to port P1.0. The program below uses the 80C515 Timer 2 capture registers to measure the duration of a positive going pulse or its period. The timer registers TH2 and TL2 are copied to the Compare/Capture Registers CRCH and CRCL upon an external signal. The period is assumed to be shorter than 2^16 machine cycles.

The approach relies on the multiple functions of the port pin P1.0. It acts both as an external interrupt, EX3, and a trigger to capture the current Timer 2 count into the registers CRCH:CRCL. The control bit I3FR of SFR T2CON determines whether EX3 is triggered by the falling edge (1-to-0) or raising edge (0-to-1) of the input signal. The interrupt also captures the Timer 2 count. The ISR for EX3 implements most of the low-level functionality. As explained below in more detail, it initializes the unit to be triggered by the first falling edge of the input signal. This generates an interrupt, which clears the contents of Timer 2. The input is now configured to be triggered upon a rising edge. Thus, the next interrupt marks the end of the negative pulse. The captured value is copied into internal data memory as the width of the pulse. The input is once again configured to be triggered by a falling edge. So the third interrupt marks the end of the period, and the beginning of the next period. The captured Timer 2 value is saved as the width of the period. Having measured both the width of the pulse and the period, further interrupts are disabled.

The program is written in a modular fashion. The subroutine `Measure` prepares the microcontroller to make a single pulse and period measurement. It clears the flags `Busy`, `Overflow` and `BeginPeriod`. It sets the flag `SignalLow` to indicate that it is the negative pulse to be measured first. Similarly, the SFR flag `I3FR` is cleared so

that the next EX3 interrupt is triggered by a 1-to-0 transition. This is the beginning of the negative pulse. `Measure` then enables the T2 and EX3 interrupts and starts Timer 2 by setting the SFR control bit `T2I0`. The routine waits until the `Busy` flag is cleared by the lower level functions. When the `Busy` flag is cleared, `Measure` turns the timer off and disables the interrupts.

```
; ================================================
;  subroutine Measure
;
;  Performs a single pulse (or period) measurement
;
;  input     : none
;  output    : Pulse width in PulseH:PulseL
;              Period width in PeriodH:PeriodL
;  destroys : a, r2, dptr
; ------------------------------------------------
Measure:
    lcall  print
    db     CR,LF, "Armed! "
    db     "Waiting for negative pulse on P1.0...",CR,LF,0
    clr    Overflow      ; clear measurement error flag
    clr    BeginPeriod
    setb   Busy
    setb   SignalLow     ; initialize polarity flag
    clr    I3FR          ; next interrupt at 1-to-0 transition
    setb   ET2           ; allow T2 interrupts
    setb   EX3           ; allow EX3 interrupts
    setb   T2I0          ; start Timer 2
here:
    jb     Busy, $       ; wait until pulse/period measured
    clr    T2I0          ; stop Timer 2
    clr    TF2           ; clear timer 2 overflow flag...
    clr    EX3           ; allow EX3 interrupts
    ret
```

The lower-level functionality is coded in the Interrupt Service Routines (ISRs). Timer 2 overflows generate interrupts that are serviced by the routine isrT2. If Timer 2 overflows before the measurements are completed, then the pulse or the period width exceeds the maximum count (0xFFFF) corresponding to about 65 milliseconds. Timer 2 ISR simply sets the `Overflow` flag.

```
; ================================================
;  subroutine isrT2
;
;  Timer 2 interrupt
;
;  input     : Timer 2 overflow
```

```
; output   : sets the error flag "Overflow"
; destroys : nothing
; ------------------------------------------------
isrT2:
   setb  Overflow     ; pulse/period exceeds FFFF
   clr   TF2
   reti
```

The pulses are received on port P1.0, which is also the input to external interrupt 3 (EX3). The SFR control bit `I3FR` determines whether a 1-to-0 or a 0-to-1 transition triggers an interrupt. This bit is initialized to 0 by the routine `Measure`, selecting the 1-to-0 transition activated interrupt. The interrupts are serviced by the routine `isrEx3`. The ISR uses the software flags `BeginPeriod` and `SignalLow`. More specifically, the ISR responds to the three consecutive events as listed below.

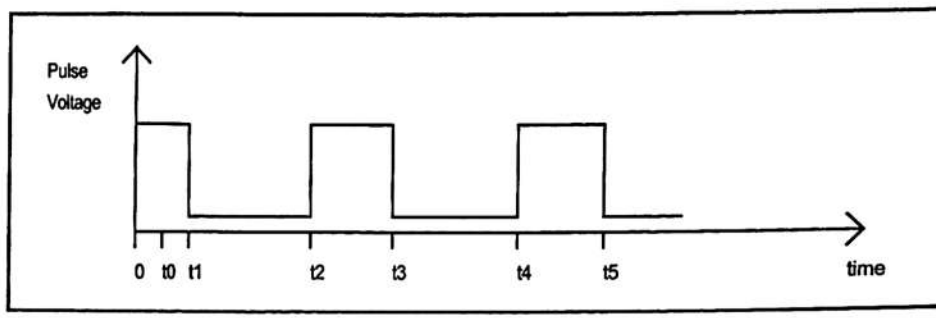


Figure 4.3. Pulse and Period Measurement.

Event (time)	Description	I3FR	BeginPeriod	SignalLow	Action
initialize (t0)		0	0	1	
1-to-0 transition (t1)	begin negative pulse	0	1	1	T2=0, I3FR=1
0-to-1 transition (t2)	end negative pulse	1	1	0	pulse width = CRC1, I3FR=0
1-to-0 transition (t3)	end period	0	1	0	period width = CRC1

The first interrupt is triggered by the first 1-to-0 transition. This marks the beginning of the negative pulse and the period. The timer count T2 is initialized to 0. The control bit I3FR is set so that the next interrupt is triggered by the 0-to-1 interrupt. The next interrupt is thus invoked when the negative pulse ends. The capture registers CRC1 (CCH1:CCL1) are saved as PulseH:PulseL. Note that the timer continues to run. The control bit I3FR is once again cleared so that the next interrupt is activated by a 1-to-0 transition. The CRC1 value at the next interrupt marks the end of the period. The flags are used to direct the ISR into one of the three sections, each corresponding to one of the three actions. The third action also disables the timer and external interrupts. The cycle is repeated when Measure is called and the flags and the interrupts are reinitialized.

```
; ================================================
;  subroutine isrEx3
;
;  External Interrupt 3
;
;  input     : External Interrupt 3
;  output    : Pulse width in PulseH:PulseL
;              Period width in PeriodH:PeriodL
;  destroys : nothing
; ------------------------------------------------
isrEx3:
    push   acc
    jb     BeginPeriod, isr01
; period starts
    mov    TL2, #0        ; clear T2 count
    mov    TH2, #0
    setb   BeginPeriod
    setb   I3FR           ; next capture upon 0-to-1
    pop    acc
    reti
isr01:
    jnb    SignalLow, isr02
; negative pulse ends
    mov    PulseL, CRCL
    mov    PulseH, CRCH
    clr    SignalLow
    clr    I3FR           ; next capture upon 1-to-0
    pop    acc
    reti
isr02:
; period complete
    mov    PeriodL, CRCL
    mov    PeriodH, CRCH
    clr    Busy
    clr    EX3            ; disable EX3 interrupts
```

```
    clr   ET2          ; disable T2 interrupts
    pop   acc
    reti
```

The subroutine `ShowResults` prints the measured values.

```
; ================================================
;  subroutine ShowResults
;
;  Displays pulse measurement results
;
;  input    : Pulse width in PulseH:PulseL
;             Period width in PeriodH:PeriodL
;  output   : writes messages to host screen
;  destroys : a, r2, dptr
; ------------------------------------------------
ShowResults:
   jnb   Overflow, ResultsOK
   lcall print        ; error
   db "Error : measurements exceed Timer 2 max count.",CR,LF,0
ret

ResultsOK:
   lcall print        ; print pulse width
   db "pulse width : ", 0
   mov   a, PulseH
   lcall prsphx
   mov   a, PulseL
   lcall prsphx
   lcall crlf

   lcall print        ; print pulse width
   db "period width : ", 0
   mov   a, PeriodH
   lcall prsphx
   mov   a, PeriodL
   lcall prsphx
   lcall crlf
   end
```

The main program defines the internal byte and bit registers. Timer 2 and the capture/compare registers are initialized. Lastly, jump instructions at the interrupt vectors. The routine `Measure` is called in a loop. At each iteration, and the results displayed.

```
; Pulse and Period Measurement
; Uses Timer 2 Capture Function
;
```

```
; UIOD connections
; Connect P1.0 to the input TTL-level pulse.

#include <sfr515.inc>
#include <util51.inc>

   dseg   at 0x70
PulseL:                  ; beginning count low byte
   ds     1
PulseH:                  ; beginning count high byte
   ds     1
PeriodL:                 ; end count low byte
   ds     1
PeriodH:                 ; end count high byte
   ds     1
   end

   bseg   at 0x10
SignalLow:               ; set while pulse is low
   dbit 1
BeginPeriod:             ; set at the beginning of the period
   dbit 1
Busy:                    ; set while measurement in progress
   dbit 1
Overflow:                ; set if pulse or period exceeds max count
   dbit 1
   end

; --- reset vector ---
   cseg   at 0
   ljmp   main
   end

; --- timer 2 interrupt vector ---
   cseg   at 0x2B
   ljmp   isrT2
   end

; --- external interrupt 3 vector ---
   cseg   at 0x53
   ljmp   isrEx3
   end

; --- foreground routine ---
   cseg at 0x100
main:
; start 535 serial port
   setb   BD            ; enable internal Baud rate generator
   mov    SCON, #50h    ; 8 bit data, mode 1
```

```
    orl    PCON, #80h    ; double the Baud rate

    mov    T2CON, #0     ; initialize Timer 2
           ;  T2I1=0 and T2I0=0
           ;  T2CM=0
           ;  T2R1=0 and T2R0=0 (no reload)
           ;  I3FR=0 (capture upon 1-to-0)
           ;  I2FR=0
           ;  T2PS=0
    mov    CCEN, #01     ; CRC capture upon external signal
loop:                    ; each iteration makes a pulse width
                         ; measurement
    lcall Measure        ; measure negative pulse and period
    lcall ShowResults    ; display measurement results
; --- ask host whether to continue or quit ---
    lcall print
    db "press any key to continue", CR, LF
    db "or 'q' to quit...", CR, LF, 0
    lcall getc
    cjne  a, #'q', loop
    lcall print          ; print message
    db CR, LF, "Program terminates...", CR, LF, 0
    clr    EAL           ; no more interrupts
    clr    ET2           ; disable T2 interrupts
    clr    EX3           ; disable EX3 interrupts
    clr    T2I0          ; stop Timer 2
    sjmp   $             ; absorbing loop
Again:
    ljmp   Loop
```

4.7.3.1. Frequency an Duty Cycle Measurement

There are a few ways to measure the frequency of the signal. One method is to count the number of pulses received during a known period of time. Timer 2 may be set up as a one-shot 16-bit counter. The signal applied to P1.0 may be used to invoke interrupts (external interrupt 3) upon a 0-to-1 transition. The interrupt service routine for external interrupt 3 may keep a software counter that is incremented upon every transition.

Alternatively, the approach given in the preceding section may be used to first find the period. Then, using the 32-bit unsigned division routine given in Chapter 3, the frequency may be computed as the reciprocal of the period. More specifically, the frequency is computed as

$$\frac{\text{Number of Machine Cycles per Second}}{\text{Number of Machine Cycles per Period}}$$

$$\frac{1000000}{\text{Number of Machine Cycles per Period}}$$

Note that 1000000 is 0xF4240. The code segment given below uses the routines given in the previous section to measure the period. The number of machine cycles per second is given as a 32-bit constant, in 4 bytes, NMCPS0 (the least significant byte) through NMCPS3 (the most significant byte). The result is displayed as a hexadecimal number.

```
; constants (NMCPS : Number of Machine Cycles Per Second)
#define NMCPS0   0x40     ; 12MHz crystal give 1000000
#define NMCPS1   0x42     ; machine cycles per second
#define NMCPS2   0x0F     ; 1000000=F4240h
#define NMCPS3   0x00

    .
    .
    lcall  Measure              ; get PeriodH:PeriodL
    lcall  ShowResults          ; display measurement results

    mov    _Y0, PeriodL         ; save period as the
    mov    _Y1, PeriodH         ; 16-bit divisor
    mov    _X3, #NMCPS3         ; dividend is the...
    mov    _X2, #NMCPS2         ; number of machine cycles...
    mov    _X1, #NMCPS1         ; per second...
    mov    _X0, #NMCPS0
    lcall  UDIV32
    jnb    _ZOV, NoError
    lcall  print
    db "Error! Division by Zero ",CR,LF,0
    sjmp   Loop                 ; repeat
NoError:
    lcall  print                ; print division result
    db "Frequency : ",0
    mov    a, _Z3
    lcall  prthex
    mov    a, _Z2
    lcall  prthex
    mov    a, #'.'              ; radix
    lcall  putc
    mov    a, _Z1
    lcall  prthex
    mov    a, _Z0
    lcall  prthex
    lcall  print
    db "   (hexadecimal)",CR,LF,0
```

Duty cycle measurement requires the measurement of the period and of the pulse width. The duty cycle is then computed as the ratio of the positive-going pulse width to the period. The program below uses the pulse width and period measurement techniques discussed above. The results, each a 16-bit number, are divided. The Pulse width is used as the higher 2 bytes of a 32-bit dividend. This is equivalent to multiplying the pulse width by 2^16. The most significant byte of the quotient is the fraction, in hexadecimal, of the time the pulse is high. This fraction is converted to a decimal fraction using the table look up routine `Percent`. The duty cycle is then displayed as a decimal percentage.

```
    .
    .
    lcall  Measure              ; measure period
    lcall  ShowResults          ; display measurement results
    mov    _Y0, PeriodL         ; save period as the
    mov    _Y1, PeriodH         ; 16-bit divisor
    mov    _X2, PulseL          ; save the pulse width as the
                                ; dividend. note that the 16-bit
    mov    _X3, PulseH          ; pulse width is stored as the
    mov    _X1, #0              ; high bytes of the 32-bit
    mov    _X0, #0              ; dividend
    lcall  UDIV32
    jnb    _ZOV, NoError
    lcall  print
    db "Error! Division by Zero ",CR,LF,0
    sjmp   Loop                 ; repeat
NoError:
    lcall  print
    db "Division Result : ",0
    mov    a, _Z3
    lcall  prsphx
    mov    a, _Z2
    lcall  prsphx
    lcall  print
    db CR,LF, "Duty Cycle : ",0
    mov    a, _Z3
    lcall  percent
    lcall  prsphx
    lcall  print
    db "% ",CR,LF,0
```

The routine `Percent` is part of the Util51 library, as explained in Chapter 3.

4.7.4. Software Timers Using the PTRA Unit

The PTRA unit provides a facility to generate 4 time-out periods, independent of each other, but based on the time base of Timer 2. By programming the timer as a free-running timer where it counts from 0 to FFFFh and not using the auto-reload function,

one has the four compare registers at his disposal. The four compare registers are the CRC, CC1, CC2, and CC3. By assigning different compare values to these registers, four different time-out periods may be obtained. These four time-out events will be indicated by the interrupts generated upon the matches. Essentially, four independent software times are available through this scheme.

The compare/capture feature is further enhanced in the 80C517 and in the 80C517A family members. For example, in the 80C517 capture/compare unit (CAPCOM) another time base running at half the oscillator frequency (six times faster than the 80C515 Timer 2) is provided. Up to 21 lines can be controlled by the two time bases provided in the CAPCOM unit. Up to five channels can be used for capturing (pulse width measurement). A new concept of shadow compare registers is introduced in the 80C517. When used, the compare events are generated when the shadow compare registers match with the timer values. Moreover, the shadow registers are automatically loaded from their corresponding compare registers when the compare timer overflows. This implementation prevents loss of compare events that may occur when the loading of compare values is not synchronized with the timer count. In the 80C517A, the CAPCOM is further enhanced by providing more interrupt capabilities for timer events.

4.8. Analog-to-Digital Converter

The 80C515 contains an on-chip Analog-to-Digital Converter. The ADC has 8 multiplexed inputs. The ADC is programmed by placing control words into the control registers ADCON and DAPR. The bits of the ADCON register are given below. MX2, MX1, and MX0 select the input channel (0..7). ADM sets the mode, ADM=1 the ADC continuously converts the selected channel to its digital value. If ADM=0, the conversion is a one-shot process. BSY is a status flag. BSY=1 while a analog-to-digital conversion is in progress. BD and CLK are not used by the ADC system. Bit 5 of ADCON must be 0.

BD	CLK	-	BSY	ADM	MX2	MX1	MX0

Once the digital value is obtained, it is stored in the ADDAT register. ADDAT as well as ADCON and DAPR are special function registers. Writing to the DAPR register starts the analog-to-digital conversion. This also has an effect on the internal voltage reference sources (refer to Section 1.3.1.2.). Upon conversion completion, the result is placed into the ADDAT register. The analog-to-digital conversion takes 13 machine cycles in the 80C515, that is, 13 microseconds with a 12 MHz crystal. Since the period is the reciprocal of frequency, 1/13 microseconds gives an upper bound of 76923076 conversions per second, or roughly 77KHz. This of course is not attainable, since some overhead is necessary to read and process the analog-to-digital conversion values. Signals with periods greater than 77KHz are clearly not observable. In fact, the highest frequency input signal is limited to half of the sampling

frequency, in this case 77KHz/2 = 38.5KHz, due to the phenomenon known as aliasing. This limit, half of the sampling frequency, is known as the Nyquist frequency.

This example implements a simple voltmeter using the 80C515 analog-to-digital converter. The output is displayed on the 7-segment displays. Connect DIGIT LOW to Port 4 (segment a to P4.0, b to P4.1, ..., segment g to P4.6) and DIGIT HIGH to Port 5 (in the same manner as DIGIT LOW). Connect the AN0 input to the processor to the AN0 potentiometer post. Note that JP2 and JP3 must be present so the ADC reference voltages are 0 and 5 volts, respectively.

The program below reads analog channel 0 using the subroutine `getAN0`. The value is displayed as two digits on the seven-segment displays. The subroutine `getAN0` waits if there is a conversion in progress. This may seem to be an unnecessary step, but it is good programming practice to be prepared for the unexpected. Next, the lower nibble of ADCON as well as bit 5 is forced to 0. A conversion is initiated by writing a 0 value to the DAPR register. This programs the IVAREF=VAREF=5V and IVAGND=VAGND=0V. The routine then waits until the busy flag is cleared by the ADC. Finally, the digital value is read and returned in the accumulator. Note that the main program inserts a delay of 1 second between display refreshes. Without this delay, the display, especially the low digit, flickers.

```
; Simple Analog-to-Digital Converter Example
;
; UIOD Connections:
; seven-segment diplay outputs
#define LOW_DIGIT  P4
#define HIGH_DIGIT P5

; analog input
#define AN0 P6.0
; ------------------------------------------------
#include <sfr515.inc>
#include <util51.inc>

   cseg at 0
begin:
; start 535 serial port
   setb  BD          ; enable internal baud rate generator
   mov   SCON, #50h  ; 8 bit data, mode 1
   orl   PCON, #80h  ; double Baud rate

; iniitialize ADC
   lcall getAN0      ; convert and get AN0 value
   push  acc         ; save Analog0 on stack
   anl   a, #0x0F    ; isolate the low nibble
   lcall display     ; convert to display pattern
```

```
    mov     LOW_DIGIT, a ; display low nibble on LOW DIGIT
    pop     acc          ; recall Analog0
    push    acc
    swap    a            ; swap nibbles
    anl     a, #0x0F     ; isolate low nibble
    lcall   display      ; get display pattern
    mov     HIGH_DIGIT, a; display on HIGH DIGIT
    lcall   print
    db      CR, LF, "ADC = ", 0
    pop     acc
    lcall   prthex

delay:
    djnz    r6, $
    mov     r6, #0xC0
    djnz    r7, delay    ; a little delay...

    lcall   peekc        ; see if there is a CTL-C pending
    cjne    a, #CTRL_C, begin
    sjmp    $            ; absorbing loop

; ================================================
;  subroutine getAN0 - get analog 0
;  input     : none
;  output    : returns the digital value of analog
;              channel 0 in the accumulator
;  destroys  : a, psw
; ------------------------------------------------
getAN0:
    jb      BSY, $       ; wait here if ADC in progress
    mov     a, adcon     ; read adc control register
    anl     a, #0xD0     ; select channel 0, single shot
    mov     adcon, a
    mov     dapr, #0     ; start conversion
    jb      BSY, $       ; wait here if ADC in progress
    mov     a, ADDAT     ; read the digital value
    ret

    end
```

4.8.1. Interrupt-Driven Timer and Analog-to-Digital Conversion Operation

You may have noticed that the program discussed in the previous section spends much of its time in the delay function or waiting for the ADC to complete the conversion. This may be a terrible waste of the microcontroller's time depending on the application. An alternative is to install the ADC as an interrupt source. Since the ADC progresses independently of the processor, all that is required is the ADC to

issue an interrupt when the current conversion is completed. The 80C515 recognizes the ADC completion as the 6-th interrupt source.

The program below illustrates an interrupt-driven ADC operation. The most recent analog value is stored in internal register named Analog0. When the ADC completes its conversion, it issues an interrupt. The ISR transfers the value in the ADDAT register to internal register Analog0 and initiates a new conversion. The main routine continuously reads the most recent value of analog channel 0 and displays it on the seven-segment displays as before. Note that the processor never waits for the ADC Busy Flag.

The ISR is initialized by the subroutine `INITADC`. Although the initialization code can be placed at the beginning of the main program, placing the initialization code in the subroutine is in the spirit of structured programming. It improves the organization and readability of the program and simplifies future modifications. Note that with this example, the processor still spends a good deal of time waiting between display refreshes.

```
; Simple Interrupt-Driven Analog-to-Digital
; Converter Example
;
; ------------------------------------------------
; UIOD Connections:
; seven-segment diplay outputs
#define LOW_DIGIT  P4
#define HIGH_DIGIT P5

; analog input
#define AN0 P6.0

; register use
#define Analog0 r5      ; save ADC result here
; ------------------------------------------------
#include <sfr515.inc>
#include <util51.inc>

; reset vector
   cseg  at 0
   ljmp  main
   end

; ADC interrupt vector
   cseg  at 0x43
   ljmp  isrADC
   end

   cseg  at 0x100
```

```
main:
; start 535 serial port
    setb   BD             ; enable internal baud rate generator
    mov    SCON, #50h     ; 8 bit data, mode 1
    orl    PCON, #80h     ; double Baud rate

; iniitialize ADC
    lcall  initADC        ; initialize ADC
    setb   EAL            ; master interrupt enable flag
begin:
    mov    a, Analog0     ; get Analog0 value
    push   acc            ; save Analog0 on stack
    anl    a, #0Fh        ; isolate the low nibble
    lcall  display        ; convert to display pattern
    mov    LOW_DIGIT, a   ; display low nibble on LOW DIGIT
    pop    acc            ; recall Analog0
    push   acc
    swap   a              ; swap nibbles
    anl    a, #0Fh        ; isolate low nibble
    lcall  display        ; get display pattern
    mov    HIGH_DIGIT,a   ; display on HIGH DIGIT
    lcall  print
    db CR, LF, "ADC = ", 0
    pop    acc
    lcall  prthex

delay:
    djnz   r6, $
    mov    r6, #0x80
    djnz   r7, delay      ; a little delay...

    lcall peekc           ; see if there is a CTL-C pending
    cjne   a, #CTRL_C, begin
    sjmp   $              ; absorbing loop

; ================================================
;  subroutine initADC - initialize adc
;  input     : none
;  output    : initializes adc interrupt
;              to continuously read analog 0
;  destroys : a, dptr
; ------------------------------------------------
initADC:
    clr    eadc           ; disable adc interrupts
    clr    iadc           ; reset adc interrupt flag
    setb   eadc           ; adc interrupt mask bit

    mov    a, adcon       ; read adc control register
    anl    a, #0d0h       ; select channel 0, single shot
```

```
    mov    adcon, a
    mov    dapr, #0      ; start conversion
    ret

; ================================================
; subroutine isrADC
; input    : adc interrupt
; output   : continuously obtains and stores analog 0
;            value in internal ram.
; destroys : nothing - uses a, r0, and r1
; ------------------------------------------------
isrADC:
   push   psw
   mov    psw, #0       ; select register bank 0
   push   acc
   push   1             ; push register 1

   mov    a, ADDAT      ; read analog value
   mov    Analog0, a    ; store analog value
   mov    a, ADCON      ; ADC control register
   anl    a, #0xD0
   mov    ADCON, a      ; select channel 0
   clr    IADC          ; reset adc interrupt flag
   mov    DAPR, #0      ; start next conversion

   pop    1
   pop    acc
   pop    psw
   reti

   end
```

The final program installs Timer 0 in mode 1 (16-bit timer), invoking interrupts upon overflow. The corresponding interrupt service routine, `Tim0isr`, refreshes the display. Again, the timer initiation code is placed in a subroutine for better program structure. With the two interrupt-driven tasks, the foreground program only checks if a Control-C is pending.

Notice that the timer interrupt is given higher priority by setting the corresponding bit in the Interrupt Priority 0 (IP0) register.

```
; UI0D Connections:
; seven-segment diplay outputs
#define LOW_DIGIT  P4
#define HIGH_DIGIT P5

; analog input
#define AN0 P6.0
```

```
; register use
#define Analog0 r5       ; save ADC result here
; ------------------------------------------------
#include <sfr515.inc>
#include <util51.inc>

; reset vector
    cseg   at 0
    ljmp   main
    end

; Timer 0 interrupt vector
    cseg   at 0x0B
    ljmp   isrT0
    end

; ADC interrupt vector
    cseg   at 0x43
    ljmp   isrADC
    end

    cseg   at 0x100
main:
; start 535 serial port
    setb   BD              ; enable internal baud rate generator
    mov    SCON, #50h      ; 8 bit data, mode 1
    orl    PCON, #80h      ; double Baud rate

; iniitialize ADC and Timer 0
    lcall  initAdc         ; initialize ADC
    lcall  initT0          ; initialize Timer 0
    mov    IP0, #2         ; timer interrupt higher priority
    setb   EAL             ; master interrupt enable flag

begin:
    lcall  peekc           ; see if there is a CTL-C pending
    cjne   a, #CTRL_C, begin
    clr    eal             ; if CTL-C, clear master interrupt
                           ; enable flag
    clr    TR0             ; Turn timer 0 off
    sjmp   $               ; absorbing loop

; ================================================
;  subroutine initadc - initialize adc
;  input     : none
;  output    : initializes adc interrupt
;              to continuously read analog 0
;  destroys  : a, dptr
```

```
; ------------------------------------------------
initAdc:
   clr   EADC          ; disable adc interrupts
   clr   IADC          ; reset adc interrupt flag
   setb  EADC          ; ADC interrupt mask bit

   mov   a, ADCON      ; read adc control register
   anl   a, #0xD0      ; select channel 0, single shot
   mov   ADCON, a
   mov   DAPR, #0      ; start conversion
   ret

; ================================================
; subroutine IsrAdc
; input    : adc interrupt
; output   : continuously obtains and stores analog 0
;            value in internal ram.
; destroys : nothing - uses a, r0, and r1
; ------------------------------------------------
isrADC:
   push  psw
   push  acc

   mov   a, addat      ; read analog value
   mov   Analog0, a    ; store analog value
   mov   a, adcon      ; ADC control register
   anl   a, #0d0h
   mov   adcon, a      ; select channel 0
   clr   iadc          ; reset adc interrupt flag
   mov   dapr, #0      ; start next conversion

   pop   acc
   pop   psw
   reti

; ================================================
; subroutine initTim0 - initialize Timer 0
; installs the interrupt service routine
; Tim0ISR to be invoked upon timer overflow
;
; input    : none
; output   : initializes Timer 0 in mode 1
; destroys : a, dptr
; ------------------------------------------------
initT0:
   mov   TMOD, #81h    ; configure timer/counter 0
                       ;  timer in mode 1
   mov   TL0, #0       ; start timer from 0
   mov   TH0, #0       ; start timer from 0
```

```
    setb   ET0          ; Timer 0 interrupt mask bit
    setb   TR0          ; turn timer/counter on
    ret

; ================================================
;  subroutine isrT0
;  input     : Timer 0 overflow interrupt (TF0)
;  output    : The value Analog0 is displayed on
;              the seven-segment displays.
;
;  destroys : nothing - uses a
; ------------------------------------------------
isrT0:
    push   psw
    push   acc

    mov    a, Analog0   ; get Analog0 value
    push   acc          ; save Analog0 on stack
    anl    a, #0x0F     ; isolate the low nibble
    lcall  display      ; convert to display pattern
    mov    LOW_DIGIT, a ; display low nibble on LOW DIGIT
    pop    acc          ; recall Analog0
    push   acc
    swap   a            ; swap nibbles
    anl    a, #0x0F     ; isolate low nibble
    lcall  display      ; get display pattern
    mov    HIGH_DIGIT,a ; display on HIGH DIGIT

    lcall  print
    db CR, LF, "ADC = ", 0
    pop    acc
    lcall  prthex

    pop    acc
    pop    psw
    reti

    end
```

4.8.2. 10-Bit Resolution with the Analog-to-Digital Conversion

The 80C515 Digital-to-Analog converter Program Register (DAPR) contains two nibbles, determining internal reference voltages IVAREF and IVAGND. With external reference voltages at 5 Volts and 0 Volts, 4 bits of resolution allows setting the internal reference voltages in 16 steps. In order to get 10-bits of resolution, the internal reference voltages is selected so that the interval [IVAREF-IVAGND] is 5/4=1.25 Volts. The analog-to-digital converter has a resolution of 8 bits within the internal

reference voltages. Thus, if the interval of the internal reference voltages contains the input voltage, 8 bits of resolution in 1.25 Volts gives a resolution of

$$\frac{1.25}{2^8} = \frac{5}{2^{10}} \text{ volts}$$

or about 5 millivolts.

The technique to get 10 bits of resolution with the 80C515 analog-to-digital converter consists of two steps. The first step is to determine the interval that contains the input voltage. The reference voltages are set to 5 Volts and 0 Volts, and an analog-to-digital conversion is performed. The result of this conversion is used to determine the appropriate reference voltages for the second step. The second step sets the internal reference voltages and performs another analog-to-digital conversion. This technique amounts to a tradeoff between resolution and sampling rate. 10 bits of resolution is obtained at the expense of reducing the sampling frequency by a factor of at least 2. That is, if two conversions are needed for 10 bits of resolution, two 13-microsoecond analog-to-digital conversions are needed at the 12 MHz oscillator frequency. This amounts to an effective conversion time of at 26 microseconds. The actual conversion time is longer since some programming overhead is needed to manage the multi-stage conversion process.

At first, it may seem sufficient to perform an analog-to-digital conversion on the full scale (0-5 Volt) and use the most significant two bits to determine the interval for the second conversion. That is, one of the four intervals [0,1.25], [1.25, 2.5], [2.5, 3.75], or [3.75, 5.0]. The result of the second conversion is then appended to the two most significant bits of the first conversion. This scheme works except for cases where the input voltage is close to one of the interval boundaries, 1.25, 2.5, or 3.75 Volts. Since the analog-to-digital converter has an error of 1/2 least significant bits, an input voltage on one of the boundaries may result in selecting the wrong interval for the second conversion. The remedy is to consider overlapping intervals: consider four primary intervals as given above, and three secondary intervals with the same width of 1.25 Volts that overlap the primary intervals.

Table 4.10. Primary and Secondary Intervals for 10-Bit Analog-to-Digital Conversion

	Primary Intervals		Secondary Intervals	
Interval	Lower Limit	Upper Limit	Lower Limit	Upper Limit
0	0	1.25		
1			0.625	1.875
2	1.25	2.5		
3			1.875	3.125
4	2.5	3.75		
5			3.125	4.375
6	3.75	5.0		

Its number (from 0 to 6) uniquely identifies each interval. The Primary Intervals are the even-numbered ones. With this scheme, there is always an interval that contains an input voltage strictly in its interior. The first conversion result is used to determine which interval to use for the second interval. If a primary interval is chosen, the most significant two bits of the first conversion are used as the first two bits of the 10-bit conversion result. If a secondary interval is used, an offset of 80h must be added to the 10-bit result. Except for the possible carry caused by the addition of the offset, the most significant 2 bits of the 10-bit result is obtained by the first conversion. The interval chosen is the one with its mid-point is closest to the input value. Dividing the full range into 16 equal segments simplifies the selection of the interval, as shown below.

Table 4.11. Selection of Intervals, High Byte, and Low Byte Offset

First Conversion Result	Lower Limit	Upper Limit	Second Conversion Interval	Second Conversion DAPR value	High Bits	Low Byte Offset
0000 xxxx	0	0.3125	0	40h	00	0
0001 xxxx	0.3125	0.625	0	40h	00	0
0010 xxxx	0.625	0.9375	0	40h	00	0
0011 xxxx	0.9375	1.25	1	62h	00	80h
0100 xxxx	1.25	1.5625	1	62h	00	80h
0101 xxxx	1.5625	1.875	2	84h	01	0
0110 xxxx	1.875	2.1875	2	84h	01	0
0111 xxxx	2.1875	2.5	3	A6h	01	80h
1000 xxxx	2.5	2.8175	3	A6h	01	80h
1001 xxxx	2.8175	3.125	4	C8h	10	0
1010 xxxx	3.125	3.4375	4	C8h	10	0
1011 xxxx	3.4375	3.75	5	EAh	10	80h
1100 xxxx	3.75	4.0625	5	EAh	10	80h
1101 xxxx	4.0625	4.375	6	0Ch	11	0
1110 xxxx	4.375	4.6875	6	0Ch	11	0
1111 xxxx	4.6875	5.0	6	0Ch	11	0

The conversion results given in the first column ignore the low nibble. The 'x's are interpreted as "don't care" bits. The result "0101 xxxx", for example, denotes the range of results "0101 0000" to "0101 1111". Here, The second conversion gives the lower 8 bits of the 10-bit analog to digital conversion. The result of the second conversion is appended to the most significant two bits given by the table. If a primary interval is used, the 10-bit result is complete. If a secondary interval is used, an offset of 80h must be added to obtain the result.

The program below illustrates an interrupt-driven analog-to-digital conversion routine that implements the technique discussed.

```
; Analog to digital converter with 10-bit resolution
;
; UIOD Connections:
; analog input
#define AN0 P6.0

#include <sfr515.inc>
```

```
#include <util51.inc>

; internal direct data memory use
   dseg at 0x10
Analog0L:
   ds    1              ; save ADC result here
Analog0H:
   ds    1              ; save ADC result here
DAPRvalue:
   ds    1              ; 2nd conversion DAPR value
AngOffset:
   ds    1              ; Offset if secondary interval
   end

; bit-addressable memory use
   bseg at 0
SecondConv:
   dbit  1              ; set after first conversion
   end

; reset vector
   cseg  at 0
   ljmp  main
   end

; ADC interrupt vector
   cseg  at 0x43
   ljmp  isrADC10
   end

   cseg  at 0x100
main:
; start 535 serial port
   setb  BD             ; bit BD enables internal
                        ; baud rate generator
   mov   SCON, #50h     ; 8 bit data, mode 1
   mov   PCON, #80h     ; double Baud rate

   lcall print
   db "Press 'q' to quit...",CR,LF,0

   mov   SP, #0x2F      ; stack starts after
                        ; bit addressable memory
   lcall initAdc10      ; initialize adc
   setb  EAL
loop:
   lcall ShowAdc10
delay:
   djnz  r6, $          ; waste a little time
```

```
    djnz   r7, delay

    mov    DAPR, #0      ; start next conversion
    lcall  peekc
    cjne   a, #'q', loop
    clr    eal
    lcall  print
    db "Program terminates...",CR,LF,0
    sjmp   $             ; absorbing loop

; ================================================
; subroutine ShowAdc10
;  input    : internal registers Analog0L and Analog0H
;  output   : print message to host
;  destroys : a, r1, r2
; ------------------------------------------------
ShowAdc10:
   mov    r1, #Analog0H; analog 0 high byte
   mov    a, @r1
   lcall  prthex
   mov    r1, #Analog0L; analog 0 low byte
   mov    a, @r1
   lcall  prthex
   lcall  crlf
   ret

; ================================================
;  subroutine GetInterval
;  determines the correct interval for the second
;  conversion.  the DAPR value and the offset are
;  computed.
;
;  input    : first conversion result in accumulator
;  output   : registers DAPRvalue and Anl0ffset are
;             updated with the DAPR value and the
;             offset to be used with the second
;             conversion.  Analog0H is determined
;             except for a possible carry after the
;             offset is added to Analog0L.
;  destroys : a, b, dptr
; ------------------------------------------------
GetInterval:
   anl    a, #0F0h
   swap   a             ; isolate high nibble
   mov    b, a          ; save for next look up
   mov    dptr, #DAPRtable
   movc   a, @a+dptr    ; table look up
   mov    r0, #DAPRvalue
   mov    @r0, a        ; store DAPR value
```

```
    mov     a, b            ; recall nibble
    mov     dptr, #OffsetTable
    movc    a, @a+dptr      ; table look up
    mov     r0, #AngOffset
    mov     @r0, a          ; store Offset

    mov     a, b            ; recall nibble
    mov     dptr, #HighByteTable
    movc    a, @a+dptr      ; table look up
    mov     r0, #Analog0H
    mov     @r0, a          ; store High Byte

    ret
DAPRtable:
    db 40h, 40h, 40h, 62h, 62h, 84h, 84h
    db 0A6h, 0A6h, 0C8h, 0C8h, 0EAh, 0EAh
    db 0Ch, 0Ch, 0Ch
OffsetTable:
    db 0, 0, 0, 80h, 80h, 0, 0, 80h, 80h
    db 0, 0, 80h, 80h, 0, 0, 0
HighByteTable:
    db 0,0,0,0,0, 1,1,1,1, 2,2,2,2, 3,3,3

; ================================================
;  subroutine initAdc10 - initialize 10-bit adc
;  input     : none
;  output    : initializes adc interrupt
;              to continuously read analog 0
;  destroys : a, dptr
; ------------------------------------------------
initAdc10:
    clr     SecondConv      ; clear 2nd Conversion Flag
    clr     eadc            ; disable adc interrupts
    clr     iadc            ; reset adc interrupt flag
    setb    eadc            ; adc interrupt mask bit
    mov     a, adcon        ; read adc control register
    anl     a, #0D0h        ; select channel 0, single shot
    mov     adcon, a
    mov     dapr, #0        ; start conversion
    ret

; ================================================
;  subroutine isrAdc10
;  input     : adc interrupt
;  output    : continuously obtains and stores analog 0
;              values in internal registers AnalogH and
;              AnalogL.  Two conversions are required.
;              The first gives the range, which is used
```

```
;            as the most significant 2 bits.  The
;            second conversion gives 8 bits.  The flag
;            ScndConv is set after the first conversion.
;            The digital-to-analog interval is stored in
;            the b register
; destroys : nothing - uses a, b, and r0
; ------------------------------------------------
isrAdc10:
   push  psw
   mov   psw, 0              ; select register bank 0
   push  acc
   push  0                   ; effecively pushes r0
   jb    SecondConv, SC      ; second step
   mov   a,   ADDAT          ; read analog value
   lcall GetInterval         ; get Analog0H, DAPR value and Offset
   mov   a, ADCON            ; read adc control register
   anl   a, #0D0h            ; select channel 0, single shot
   mov   ADCON, a
   setb  SecondConv          ; set conversion step 2
   clr   IADC                ; reset adc interrupt flag
   mov   r0, #DAPRvalue
   mov   a, @r0              ; get next DAPR value
   mov   DAPR, a             ; start next conversion
   sjmp  isrDone             ; ISR done
SC:
   mov   r0, #AngOffset
   mov   a, @r0              ; get offset
   add   a, ADDAT            ; add analog value
   mov   r0,  #Analog0L      ; address of Analog0L
   mov   @r0, a              ; store analog value low byte
   jnc   NoCarry             ; 10-bit result OK if no carry
   mov   r0,  #Analog0H      ; else increment high byte
   inc   @r0
NoCarry:
   mov   a, ADCON            ; read adc control register
   anl   a, #0D0h            ; select channel 0, single shot
   mov   ADCON, a
   clr   SecondConv          ; reset conversion step
   clr   IADC                ; reset adc interrupt flag
isrDone:
   pop   0
   pop   acc
   pop   psw
   reti
; ================================================
   end
```

4.9. Serial Communications

The serial port of the 8051 family of microcontrollers is a versatile subsystem that supports 8 or 9 data bits of transmission in polled or an interrupt-driven fashion. Programming the serial port requires the specification of the Baud rate and the source of the clock signal at that Baud rate. Internal timers may be used to generate the Baud rate. The 80C515 has an additional internal Baud rate generator which can be used to generate 4800 and 9600 Baud clock signals with a 12 MHz crystal. The internal Baud rate generator is preferable in cases where all 3 timers are needed for the application. In the 80C517 an additional serial channel is added. The internal Baud rate generator for this new channel is programmable. In case of the 80C517A both serial channels have programmable Baud rate generators.

In the UART (Universal Asynchronous Receiver/Transmitter) modes serial transmission starts as soon as the data byte is written to the serial data buffer SFR, SBUF. The asynchronous mode is the most commonly used mode in serial communications, e.g. in RS-232C. The serial control SFR, SCON at address 98h, includes the flag bit TI, which is set when the transmission is completed. Serial data is received and accumulated until the data byte is complete. The flag bit RI of SCON is set upon receive completion. The two flags TI and RI are OR-ed and used as an interrupt source.

Serial communications may be implemented in a polled or an interrupt-driven manner. The polled operation requires the CPU to inspect the flags TI and RI to determine if the previous transmission is completed or if a new data byte has been received. If the flag RI is not polled frequently enough, a new byte may be received, overwriting the old byte. On the other hand, if RI is polled frequently, valuable CPU time may be wasted on unnecessary instructions. In an interrupt-driven approach, the interrupt service routine needs to inspect the RI and TI flags to determine the source of the interrupt. If both flags are set, depending on the program implementation, either the receive event or the transmit event will be serviced. Servicing the receive event is usually preferable, since the transmit operation is under user control. Not servicing the received byte, however, may result in loss of data as discussed above. The choice of the operating mode depends on the specific application.

The program below illustrates the polled operation of the UART. The program initiates the serial port and polls RI. Whenever a character is received, it is echoed back. The flag TI is inspected after the transmission is initiated. The subroutine returns upon transmission completion, that is, when TI is set. The received character is inspected. The program enters an endless absorbing loop if the character received is 'q'.

```
; 8-bit UART operated in a polled manner.
; Echoes the characters received.  Quits upon 'q'.

#include <sfr515.inc>
#include <util51.inc>

; reset vector
   cseg  at 0
   ljmp  main
   end

   cseg  at 0x100
main:
; --- use either one of the initialization routines --------
   lcall initSio              ; start serial port

loop:
   lcall getchr               ; receive character in accumulator
   lcall sndchr               ; echo character back
   cjne  a, #'q', loop        ; quit if 'q'
   sjmp  $                    ; absorbing loop

; ========================================================
; subroutine getchr
; this routine reads in a character from the serial port and
; returns it in the accumulator.
; ========================================================
getchr:
   jnb   RI, $                ; wait until character received
   mov   a, SBUF              ; get character
   clr   acc.7                ; mask off 8th bit for 7-bit ASCII
   clr   RI                   ; clear receive flag
ret

; ========================================================
; subroutine sndchr
; this routine takes the character in the accumulator and
; sends it out the serial port.
; ========================================================
sndchr:
   clr   TI                   ; clear flag
   mov   SBUF, a              ; initiate transmission
   jnb   TI, $                ; wait until character is sent
   ret
; ========================================================
   end
```

Next, consider the interrupt-driven operation of the UART. The characters received are placed in a circular buffer by an interrupt service routine. If the CPU is busy with

lengthy tasks, the characters accumulate in the circular buffer. When the CPU is free, it inspects the buffer and echoes the characters. The circular buffer is 16 bytes long. It is placed in internal registers. Alternatively, external memory may be used, especially if a large buffer is needed. There are two pointers to the buffer. The head pointer (HP) gives the offset of the byte next to be read from the buffer. The tail pointer (TP) gives the offset of the byte last added to the buffer. Thus when a character is received, the tail pointer must be updated before the byte is stored. An empty buffer is indicated by the convention that TP=HP. Each time a new character is received, the pointers are inspected to see if the buffer is full. If updating TP causes the condition TP=HP, then the buffer is full, and TP should not be incremented. If the buffer is full, an overflow flag SPOV is set. The main routine may inspect the overflow flag and take appropriate action. The subroutine SNDCHR uses an auxiliary transmit flag. When a transmit interrupt is received, the auxiliary transmit flag is set. This signals the SNDCHR routine that the transmission is completed. The routine then clears the auxiliary flag. The transmit flag TI is cleared by the interrupt service routine so that no further transmit interrupts are generated.

```
; 8-bit UART operated in an interrupt-driven manner.
; Echoes the characters received.  Quits upon 'q'.

UartCode segment code

#include <sfr51.inc>
#include <util51.inc>

; internal direct data memory
    dseg   at 0x6E
CBHP:
    ds     1            ; circular buffer head pointer
CBTP:
    ds     1            ; circular buffer tail pointer
CBuffer:
    ds     16           ; 16-byte circular input buffer
    end

; bit-addressable memory
    bseg   at 0
SPOV:
    dbit   1            ; overflow flag
AUX_TI:
    dbit   1            ; auxiliary transmit flag
    end

; reset vector
    cseg   at 0
    ljmp   main
    end
```

```
; serial port interrupt vector
    cseg  at 0x23
    ljmp  isrSP
    end

    rseg  UartCode
main:
    lcall initSio              ; start serial port
    mov   CBTP, #0
    mov   CBHP, #0
    setb  ES                   ; enable serial port interrupts
    setb  EA                   ; enable interrupts
loop:
;   . put some time consuming operations here
;   . serial data received is stored in the circular buffer
;   .
time_consuming:
    djnz  r6, $
    djnz  r7, time_consuming

    mov   a, CBHP              ; get head pointer
    cjne  a, CBTP, echo        ; see if buffer empty
    sjmp  loop
echo:
    add   a, #Cbuffer          ; compute address
    mov   r0, a
    mov   a, @r0               ; get character from buffer
    cjne  a, #'q', noQuit ; 'q' (quit) ?
    sjmp  $                    ; if so, return to monitor
noQuit:
    lcall sndchr               ; echo character back
    inc   CBHP                 ; update head pointer
    anl   CBHP, #0Fh           ; modulus 16
    sjmp  loop                 ; repeat

; ========================================================
; subroutine sndchr
; this routine takes the character in the accumulator and
; sends it out the serial port.
; ========================================================
sndchr:
    mov   SBUF, a              ; initiate transmission
    jnb   AUX_TI, $            ; wait until character is sent
    clr   AUX_TI               ; clear flag
    ret
```

```
; ========================================================
; subroutine isrSP
; this routine reads in a character from the serial port and
; saves it in the circular buffer.
; ignores transmit interrupts
; ========================================================
isrSP:
   push   acc
   push   b
   jnb    TI, isr01          ; transmit interrupt ?
   setb   AUX_TI             ; set auxiliary flag
   clr    TI                 ; clear TI -- no more interrupts
isr01:
   jnb    RI, isr03          ; done if no receive interrupt
   mov    b,  SBUF           ; get character
   clr    b.7                ; mask off 8th bit for 7-bit ASCII
   clr    RI                 ; clear serial status bit

   mov    a, CBTP            ; get tail pointer
   inc    a                  ; update tail pointer
   anl    a, #0Fh            ; modulus 16

   cjne   a, CBHP, isr02     ; see if TP=HP
   setb   SPOV               ; if HP=TP buffer full (set flag)
   sjmp   isr03              ; done
   reti
isr02:
   mov    CBTP, a            ; if HP not equal to TP write new TP
   add    a, #Cbuffer        ; compute address
   mov    r0, a              ; register-indirect addressing
   mov    a, b               ; recall character received
   mov    @r0, a             ; store character in circular buffer
isr03:
   pop    b
   pop    acc
   reti

   end
```