8051 Tutorial

(C) Copyright 1997, 1998, 1999 by Vault Information Services. All Rights Reserved.
http:\\www.8052.com

Contents

8051 Tutorial: Introduction

Despite it's relatively old age, the 8051 is one of the most popular microcontrollers in use today. Many derivative microcontrollers have since been developed that are based on--and compatible with--the 8051. Thus, the ability to program an 8051 is an important skill for anyone who plans to develop products that will take advantage of microcontrollers.

Many web pages, books, and tools are available for the 8051 developer.

I hope the information contained in this document/web page will assist you in mastering 8051 programming. While it is not my intention that this document replace a hardcopy book purchased at your local book store, it is entirely possible that this may be the case. It is likely that this document contains everything you will need to learn 8051 assembly language programming. Of course, this document is free and you get what you pay for so if, after reading this document, you still are lost you may find it necessary to buy a book.

This document is both a tutorial and a reference tool. The various chapters of the document will explain the 8051 step by step. The chapters are targeted at people who are attempting to learn 8051 assembly language programming. The appendices are a useful reference tool that will assist both the novice programmer as well as the experienced professional developer.

This document assumes the following:

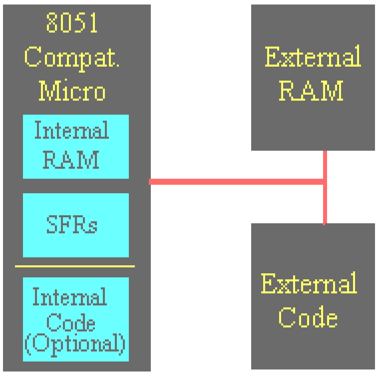
- A general knowledge of programming.
- An understanding of decimal, hexidecimal, and binary number systems.
- A general knowledge of hardware.

That is to say, no knowledge of the 8051 is assumed--however, it is assumed you've done some amount of programming before, have a basic understanding of hardware, and a firm grasp on the three numbering systems mentioned above. The concept of converting a number from deciminal to hexidecimal and/or to binary is not within the scope of this document--and if you can't do those types of conversions there are probably some concepts that will not be completely understandable.

This document attempts to address the need of the typical programmer. For example, there are certain features that are nifty and in some cases very useful--but 95% of the programmers will never use these features.

8051 Tutorial: Types of Memory

The 8051 has three very general types of memory. To effectively program the 8051 it is necessary to have a basic understanding of these memory types.



On-Chip Memory refers to any memory (Code, RAM, or other) that physically exists on the microcontroller itself. On-chip memory can be of several types, but we'll get into that shortly.

External Code Memory is code (or program) memory that resides off-chip. This is often in the form of an external EPROM.

External RAM is RAM memory that resides off-chip. This is often in the form of standard static RAM or flash RAM.

Code Memory

Code memory is the memory that holds the actual 8051 program that is to be run. This memory is limited to 64K and comes in many shapes and sizes: Code memory may be found *on-chip*, either burned into the microcontroller as ROM or EPROM. Code may also be stored completely *off-chip* in an external ROM or, more commonly, an external EPROM. Flash RAM is also another popular method of storing a program. Various combinations of these memory types may also be used--that is to say, it is possible to have 4K of code memory *on-chip* and 64k of code memory *off-chip* in an EPROM.

When the program is stored on-chip the 64K maximum is often reduced to 4k, 8k, or 16k. This varies depending on the version of the chip that is being used. Each version offers specific capabilities and one of the distinguishing factors from chip to chip is how much ROM/EPROM space the chip has.

However, code memory is most commonly implemented as off-chip EPROM. This is especially true in low-cost development systems and in systems developed by students.

Programming Tip: *Since code memory is restricted to 64K, 8051 programs are limited to 64K. Some assemblers and compilers offer ways to get around this limit when used with specially wired hardware. However, without such special compilers and hardware, programs are limited to 64K.*

External RAM

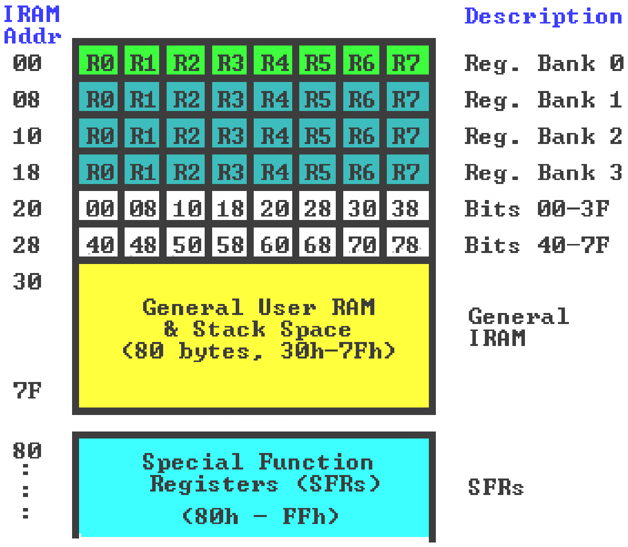
As an obvious opposite of *Internal RAM*, the 8051 also supports what is called *External RAM*. As the name suggests, External RAM is any random access memory which is found *off-chip*. Since the memory is off-chip it is not as flexible in terms of accessing, and is also slower. For example, to increment an Internal RAM location by 1 requires only 1 instruction and 1 instruction cycle. To increment a 1-byte value stored in External RAM requires 4 instructions and 7 instruction cycles. In this case, external memory is 7 times slower!

What External RAM loses in speed and flexibility it gains in quantity. While Internal RAM is limited to 128 bytes the 8051 supports External RAM up to 64K.

Programming Tip: *The 8051 may only address 64k of RAM. To expand RAM beyond this limit requires programming and hardware tricks. You may have to do this "by hand" since many compilers and assemblers, while providing support for programs in excess of 64k, do not support more than 64k of RAM. This is rather strange since it has been my experience that programs can usually fit in 64k but often RAM is what is lacking. Thus if you need more than 64k of RAM, check to see if your compiler supports it-- but if it doesn't, be prepared to do it by hand.*

On-Chip Memory.

As mentioned at the beginning of this chapter, the 8051 includes a certain amount of on-chip memory. On-chip memory is really one of two (SFR) memory. The layout of the 8051's internal memory is presented in the following memory map:



IRAM Addr									Description
00	R0	R1	R2	R3	R4	R5	R6	R7	Reg. Bank 0
08	R0	R1	R2	R3	R4	R5	R6	R7	Reg. Bank 1
10	R0	R1	R2	R3	R4	R5	R6	R7	Reg. Bank 2
18	R0	R1	R2	R3	R4	R5	R6	R7	Reg. Bank 3
20	00	08	10	18	20	28	30	38	Bits 00-3F
28	40	48	50	58	60	68	70	78	Bits 40-7F
30 7F	General User RAM & Stack Space (80 bytes, 30h-7Fh)								General IRAM
80 :	Special Function Registers (SFRs) (80h - FFh)								SFRs

As is illustrated in this map, the 8051 has a bank of 128 bytes of *Internal RAM*. This Internal RAM is found *on-chip* on the 8051 so it is the fastest RAM available, and it is also the most flexible in terms of reading, writing, and modifying it's contents. Internal RAM is volatile, so when the 8051 is reset this memory is cleared.

The 128 bytes of internal ram is subdivided as shown on the memory map. The first 8 bytes (00h - 07h) are "register bank 0". By manipulating certain SFRs, a program may choose to use register banks 1, 2, or 3. These alternative register banks are located in internal RAM in addresses 08h through 1Fh.

We'll discuss "register banks" more in a later chapter. For now it is sufficient to know that they "live" and are part of internal RAM.

Bit Memory also lives and is part of internal RAM. We'll talk more about bit memory very shortly, but for now just keep in mind that bit memory actually resides in internal RAM, from addresses 20h through 2Fh.

The 80 bytes remaining of Internal RAM, from addresses 30h through 7Fh, may be used by user variables that need to be accessed frequently or at high-speed. This area is also utilized by the microcontroller as a storage area for the operating *stack*. This fact severely limits the 8051's stack since, as illustrated in the memory map, the area reserved for the stack is only 80 bytes--and usually it is less since this 80 bytes has to be shared between the stack and user variables.

Register Banks

The 8051 uses 8 "R" registers which are used in many of its instructions. These "R" registers are numbered from 0 through 7 (R0, R1, R2, R3, R4, R5, R6, and R7). These registers are generally used to assist in manipulating values and moving data from one memory location to another. For example, to add the value of R4 to the Accumulator, we would execute the following instruction:

```
ADD A,R4
```

However, as the memory map shows, the "R" Register R4 is really part of Internal RAM. Specifically, R4 is address 04h. This can be see in the bright green section of the memory map. Thus the above instruction accomplishes the same thing as the following operation:

```
ADD A,04h
```

This instruction adds the value found in Internal RAM address 04h to the value of the Accumulator, leaving the result in the Accumulator. Since R4 is really Internal RAM 04h, the above instruction effectively accomplished the same thing.

But watch out! As the memory map shows, the 8051 has four distinct register banks. When the 8051 is first booted up, register bank 0 (addresses 00h through 07h) is used by default. However, your program may instruct the 8051 to use one of the alternate register banks; i.e., register banks 1, 2, or 3. In this case, R4 will no longer be the same as Internal RAM address 04h. For example, if your program instructs the 8051 to use register bank 3, "R" register R4 will now be synonomous with Internal RAM address 1Ch.

The concept of register banks adds a great level of flexibility to the 8051, especially when dealing with interrupts (we'll talk about interrupts later). However, always remember that the register banks really reside in the first 32 bytes of Internal RAM.

Programming Tip: *If you only use the first register bank (i.e. bank 0), you may use Internal RAM locations 08h through 1Fh for your own use. But if you plan to use register banks 1, 2, or 3, be very careful about using addresses below 20h as you may end up overwriting the value of your "R" registers!*

Bit Memory

The 8051, being a communications-oriented microcontroller, gives the user the ability to access a number of *bit variables*. These variables may be either 1 or 0.

There are 128 bit variables available to the user, numberd 00h through 7Fh. The user may make use of these variables with commands such as SETB and CLR.

It is important to note that Bit Memory is really a part of Internal RAM. In fact, the 128 bit variables occupy the 16 bytes of Internal RAM from 20h through 2Fh. Thus, if you write the value FFh to Internal RAM address 20h you've effectively set bits 00h through 07h.

But since the 8051 provides special instructions to access these 16 bytes of memory on a bit by bit basis it is useful to think of it as a separate type of memory. However, always keep in mind that it is just a subset of Internal RAM--and that operations performed on Internal RAM can change the values of the bit variables.

Programming Tip: *If your program does not use bit variables, you may use Internal RAM locations 20h through 2Fh for your own use. But if you plan to use bit variables, be very careful about using addresses from 20h through 2Fh as you may end up overwriting the value of your bits!*

Bit variables 00h through 7Fh are for user-defined functions in their programs. However, bit variables 80h and above are actually used to access certain SFRs on a bit-by-bit basis. For example, if output lines P0.0 through P0.7 are all clear (0) and you want to turn on the P0.0 output line you may either execute:

```
MOV P0,#01h  ||  SETB 80h
```

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

Programming Tip: *By default, the 8051 initializes the Stack Pointer (SP) to 08h when the microcontroller is booted. This means that the stack will start at address 08h and expand upwards. If you will be using the alternate register banks (banks 1, 2 or 3) you must initialize the stack pointer to an address above the highest register bank you will be using, otherwise the stack will overwrite your alternate register banks. Similarly, if you will be using bit variables it is usually a good idea to initialize the stack pointer to some value greater than 2Fh to guarantee that your bit variables are protected from the stack.*

Special Function Register (SFR) Memory

Special Function Registers (SFRs) are areas of memory that control specific functionality of the 8051 processor. For example, four SFRs permit access to the 8051's 32 input/output lines. Another SFR allows a program to read or write to the 8051's serial port. Other SFRs allow the user to set the serial baud rate, control and access timers, and configure the 8051's interrupt system.

When programming, SFRs have the illusion of being Internal Memory. For example, if you want to write the value "1" to Internal RAM location 50 hex you would execute the instruction:

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

Similarly, if you want to write the value "1" to the 8051's serial port you would write this value to the **SBUF** SFR, which has an SFR address of 99 Hex. Thus, to write the value "1" to the serial port you would execute the instruction:

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

As you can see, it appears that the SFR is part of Internal Memory. This is not the case. When using this method of memory access (it's called direct address), any instruction that has an address of 00h through 7Fh refers to an Internal RAM memory address; any instruction with an address of 80h through FFh refers to an SFR control register.

Programming Tip: *SFRs are used to control the way the 8051 functions. Each SFR has a specific purpose and format which will be discussed later. Not all addresses above 80h are assigned to SFRs. However, this area may NOT be used as additional RAM memory even if a given address has not been assigned to an SFR.*

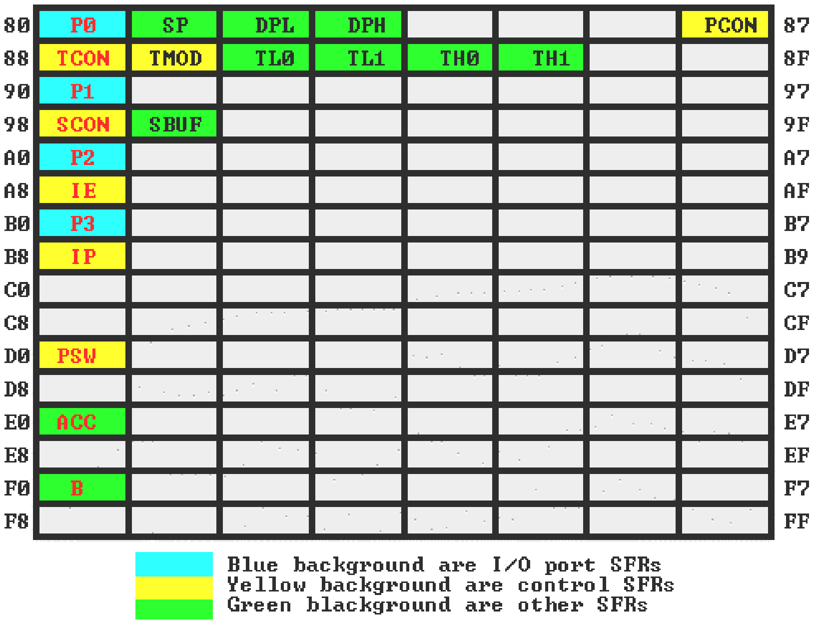
8051 Tutorial: SFRs

What Are SFRs?

The 8051 is a flexible microcontroller with a relatively large number of modes of operations. Your program may inspect and/or change the operating mode of the 8051 by manipulating the values of the 8051's Special Function Registers (SFRs).

SFRs are accessed as if they were normal Internal RAM. The only difference is that Internal RAM is from address 00h through 7Fh whereas SFR registers exist in the address range of 80h through FFh.

Each SFR has an address (80h through FFh) and a name. The following chart provides a graphical presentation of the 8051's SFRs, their names, and their address.



80	P0	SP	DPL	DPH				PCON	87
88	TCON	TMOD	TL0	TL1	TH0	TH1			8F
90	P1								97
98	SCON	SBUF							9F
A0	P2								A7
A8	IE								AF
B0	P3								B7
B8	IP								B9
C0									C7
C8									CF
D0	PSW								D7
D8									DF
E0	ACC								E7
E8									EF
F0	B								F7
F8									FF

🟦 Blue background are I/O port SFRs
🟨 Yellow background are control SFRs
🟩 Green blackground are other SFRs

As you can see, although the address range of 80h through FFh offer 128 possible addresses, there are only 21 SFRs in a standard 8051. All other addresses in the SFR range (80h through FFh) are considered invalid. Writing to or reading from these registers may produce undefined values or behavior.

Programming Tip: *It is recommended that you not read or write to SFR addresses that have not been assigned to an SFR. Doing so may provoke undefined behavior and may cause your program to be incompatible with other 8051-derivatives that use the given SFR for some other purpose.*

SFR Types

As mentioned in the chart itself, the SFRs that have a blue background are SFRs related to the I/O ports. The 8051 has four I/O ports of 8 bits, for a total of 32 I/O lines. Whether a given I/O line is high or low and the value read from the line are controlled by the SFRs in green.

The SFRs with yellow backgrouns are SFRs which in some way control the operation or the configuration of some aspect of the 8051. For example, **TCON** controls the timers, SCON controls the serial port.

The remaining SFRs, with green backgrounds, are "other SFRs." These SFRs can be thought of as auxillary SFRs in the sense that they don't directly configure the 8051 but obviously the 8051 cannot operate without them. For example, once the serial port has been configured using **SCON**, the program may read or write to the serial port using the **SBUF** register.

Programming Tip: *The SFRs whose names appear in red in the chart above are SFRs that may be accessed via bit operations (i.e., using the **SETB** and **CLR** instructions). The other SFRs cannot be accessed using bit operations. As you can see, all SFRs that whose addresses are divisible by 8 can be accessed with bit operations.*

SFR Descriptions

This section will endeavor to quickly overview each of the standard SFRs found in the above SFR chart map. It is not the intention of this section to fully explain the functionality of each SFR--this information will be covered in separate chapters of the tutorial. This section is to just give you a general idea of what each SFR does.

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

Programming Tip: *While the 8051 has four I/O port (P0, P1, P2, and P3), if your hardware uses external RAM or external code memory (i.e., your program is stored in an external ROM or EPROM chip or if you are using external RAM chips) you may not use P0 or P2. This is because the 8051 uses ports P0 and P2 to address the external memory. Thus if you are using external RAM or code memory you may only use ports P1 and P3 for your own use.*

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

Programming Tip: *The SP SFR, on startup, is initialized to 07h. This means the stack will start at 08h and start expanding upward in internal RAM. Since alternate register banks 1, 2, and 3 as well as the user bit variables occupy internal RAM from addresses 08h through 2Fh, it is necessary to initialize SP in your program to some other value if you will be using the alternate register banks and/or bit memory. It's not a bad idea to initialize SP to 2Fh as the first instruction of every one of your programs unless you are 100% sure you will not be using the register banks and bit variables.*

DPL/DPH (Data Pointer Low/High, Addresses 82h/83h): The SFRs DPL and DPH work together to represent a 16-bit value called the *Data Pointer*. The data pointer is used in operations regarding external RAM and some instructions involving code memory. Since it is an unsigned two-byte integer value, it can represent values from 0000h to FFFFh (0 through 65,535 decimal).

Programming Tip: *DPTR is really DPH and DPL taken together as a 16-bit value. In reality, you almost always have to deal with DPTR one byte at a time. For example, to push DPTR onto the stack you must first push DPL and then DPH. You can't simply plush DPTR onto the stack. Additionally, there is an instruction to "increment DPTR." When you execute this instruction, the two bytes are operated upon as a 16-bit value. However, there is no instruction that decrements DPTR. If you wish to decrement the value of DPTR, you must write your own code to do so.*

PCON (Power Control, Addresses 87h): The Power Control SFR is used to control the 8051's power control modes. Certain operation modes of the 8051 allow the 8051 to go into a type of "sleep" mode which requires much less power. These modes of operation are controlled through PCON. Additionally, one of the bits in PCON is used to double the effective baud rate of the 8051's serial port.

TCON (Timer Control, Addresses 88h, Bit-Addressable): The Timer Control SFR is used to configure and modify the way in which the 8051's two timers operate. This SFR controls whether each of the two timers is running or stopped and contains a flag to indicate that each timer has overflowed. Additionally, some non-timer related bits are located in the TCON SFR. These bits are used to configure the way in which the external interrupts are activated and also contain the external interrupt flags which are set when an external interrupt has occured.

TMOD (Timer Mode, Addresses 89h): The Timer Mode SFR is used to configure the mode of operation of each of the two timers. Using this SFR your program may configure each timer to be a 16-bit timer, an 8-bit autoreload timer, a 13-bit timer, or two separate timers. Additionally, you may configure the timers to only count when an external pin is activated or to count "events" that are indicated on an external pin.

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

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

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

SCON (Serial Control, Addresses 98h, Bit-Addressable): The Serial Control SFR is used to configure the behavior of the 8051's on-board serial port. This SFR controls the baud rate of the serial port, whether the serial port is activated to receive data, and also contains flags that are set when a byte is successfully sent or received.

Programming Tip: *To use the 8051's on-board serial port, it is generally necessary to initialize the following SFRs: SCON, TCON, and TMOD. This is because SCON controls the serial port. However, in most cases the program will wish to use one of the timers to establish the serial port's baud rate. In this case, it is necessary to configure timer 1 by initializing TCON and TMOD.*

SBUF (Serial Control, Addresses 99h): The Serial Buffer SFR is used to send and receive data via the on-board serial port. Any value written to SBUF will be sent out the serial port's TXD pin. Likewise, any value which the 8051 receives via the serial port's RXD pin will be delivered to the user program via SBUF. In other words, SBUF serves as the output port when written to and as an input port when read from.

P2 (Port 2, Address A0h, Bit-Addressable): This is input/output port 2. Each bit of this SFR corresponds to one of the pins on the microcontroller. For example, bit 0 of port 2 is pin P2.0, bit 7 is pin P2.7. Writing a value of 1 to a bit of this SFR will send a high level on the corresponding I/O pin whereas a value of 0 will bring it to a low level.

Programming Tip: *While the 8051 has four I/O port (P0, P1, P2, and P3), if your hardware uses external RAM or external code memory (i.e., your program is stored in an external ROM or EPROM chip or if you are using external RAM chips) you may not use P0 or P2. This is because the 8051 uses ports P0 and P2 to address the external memory. Thus if you are using external RAM or code memory you may only use ports P1 and P3 for your own use.*

IE (Interrupt Enable, Addresses A8h): The Interrupt Enable SFR is used to enable and disable specific interrupts. The low 7 bits of the SFR are used to enable/disable the specific interrupts, where as the highest bit is used to enable or disable ALL interrupts. Thus, if the high bit of IE is 0 all interrupts are disabled regardless of whether an individual interrupt is enabled by setting a lower bit.

P3 (Port 3, Address B0h, Bit-Addressable): This is input/output port 3. Each bit of this SFR corresponds to one of the pins on the microcontroller. For example, bit 0 of port 3 is pin P3.0, bit 7 is pin P3.7. Writing a value of 1 to a bit of this SFR will send a high level on the corresponding I/O pin whereas a value of 0 will bring it to a low level.

IP (Interrupt Priority, Addresses B8h, Bit-Addressable): The Interrupt Priority SFR is used to specify the relative priority of each interrupt. On the 8051, an interrupt may either be of low (0) priority or high (1) priority. An interrupt may only interrupt interrupts of lower priority. For example, if we configure the 8051 so that all interrupts are of low priority except the serial interrupt, the serial interrupt will always be able to interrupt the system, even if another interrupt is currently executing. However, if a serial interrupt is executing no other interrupt will be able to interrupt the serial interrupt routine since the serial interrupt routine has the highest priority.

PSW (Program Status Word, Addresses D0h, Bit-Addressable): The Program Status Word is used to store a number of important bits that are set and cleared by 8051 instructions. The PSW SFR contains the carry flag, the auxiliary carry flag, the overflow flag, and the parity flag. Additionally, the PSW register contains the register bank select flags which are used to select which of the "R" register banks are currently selected.

Programming Tip: *If you write an interrupt handler routine, it is a very good idea to always save the PSW SFR on the stack and restore it when your interrupt is complete. Many 8051 instructions modify the bits of PSW. If your interrupt routine does not guarantee that PSW is the same upon exit as it was upon entry, your program is bound to behave rather erradically and unpredictably--and it will be tricky to debug since the behavior will tend not to make any sense.*

ACC (Accumulator, Addresses E0h, Bit-Addressable): The Accumulator is one of the most-used SFRs on the 8051 since it is involved in so many instructions. The Accumulator resides as an SFR at E0h, which means the instruction **MOV A,#20h** is really the same as **MOV E0h,#20h**. However, it is a good idea to use the first method since it only requires two bytes whereas the second option requires three bytes.

B (B Register, Addresses F0h, Bit-Addressable): The "B" register is used in two instructions: the multiply and divide operations. The B register is also commonly used by programmers as an auxiliary register to temporarily store values.

Other SFRs

The chart above is a summary of all the SFRs that exist in a standard 8051. All derivative microcontrollers of the 8051 must support these basic SFRs in order to maintain compatability with the underlying MSCS51 standard.

A common practice when semiconductor firms wish to develop a new 8051 derivative is to add additional SFRs to support new functions that exist in the new chip.

For example, the Dallas Semiconductor DS80C320 is upwards compatible with the 8051. This means that any program that runs on a standard 8051 should run without modification on the DS80C320. This means that all the SFRs defined above also apply to the Dallas component.

However, since the DS80C320 provides many new features that the standard 8051 does not, there must be some way to control and configure these new features. This is accomplished by adding additional SFRs to those listed here. For example, since the DS80C320 supports two serial ports (as opposed to just one on the 8051), the SFRs SBUF2 and SCON2 have been added. In addition to all the SFRs listed above, the DS80C320 also recognizes these two new SFRs as valid and uses their values to determine the mode of operation of the secondary serial port. Obviously, these new SFRs have been assigned to SFR addresses that were unused in the original 8051. In this manner, new 8051 derivative chips may be developed which will run existing 8051 programs.

Programming Tip: *If you write a program that utilizes new SFRs that are specific to a given derivative chip and not included in the above SFR list, your program will not run properly on a standard 8051 where that SFR does not exist. Thus, only use non-standard SFRs if you are sure that your program wil only have to run on that specific microcontroller. Likewise, if you write code that uses non-standard SFRs and subsequently share it with a third-party, be sure to let that party know that your code is using non-standard SFRs to save them the headache of realizing that due to strange behavior at run-time.*

8051 Tutorial: Basic Registers

The Accumulator

If you've worked with any other assembly languages you will be familiar with the concept of an *Accumulator* register.

The Accumulator, as it's name suggests, is used as a general register to accumulate the results of a large number of instructions. It can hold an 8-bit (1-byte) value and is the most versatile register the 8051 has due to the shear number of instructions that make use of the accumulator. More than half of the 8051's 255 instructions manipulate or use the accumulator in some way.

For example, if you want to add the number 10 and 20, the resulting 30 will be stored in the Accumulator. Once you have a value in the Accumulator you may continue processing the value or you may store it in another register or in memory.

The "R" registers

The "R" registers are a set of eight registers that are named R0, R1, etc. up to and including R7.

These registers are used as auxillary registers in many operations. To continue with the above example, perhaps you are adding 10 and 20. The original number 10 may be stored in the Accumulator whereas the value 20 may be stored in, say, register R4. To process the addition you would execute the command:

```
ADD A,R4
```

After executing this instruction the Accumulator will contain the value 30.

You may think of the "R" registers as very important auxillary, or "helper", registers. The Accumulator alone would not be very useful if it were not for these "R" registers.

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

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

As you can see, we used R5 to temporarily hold the sum of R3 and R4. Of course, this isn't the most efficient way to calculate (R1+R2) - (R3 +R4) but it does illustrate the use of the "R" registers as a way to store values temporarily.

The "B" Register

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

The "B" register is only used by two 8051 instructions: MUL AB and DIV AB. Thus, if you want to quickly and easily multiply or divide A by another number, you may store the other number in "B" and make use of these two instructions.

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

The Data Pointer (DPTR)

The Data Pointer (DPTR) is the 8051's only user-accessable 16-bit (2-byte) register. The Accumulator, "R" registers, and "B" register are all 1-byte values.

DPTR, as the name suggests, is used to point to data. It is used by a number of commands which allow the 8051 to access external memory. When the 8051 accesses external memory it will access external memory at the address indicated by DPTR.

While DPTR is most often used to point to data in external memory, many programmers often take advantge of the fact that it's the only true 16-bit register available. It is often used to store 2-byte values which have nothing to do with memory locations.

The Program Counter (PC)

The Program Counter (PC) is a 2-byte address which tells the 8051 where the next instruction to execute is found in memory. When the 8051 is initialized PC always starts at 0000h and is incremented each time an instruction is executed. It is important to note that PC isn't always incremented by one. Since some instructions require 2 or 3 bytes the PC will be incremented by 2 or 3 in these cases.

The Program Counter is special in that there is no way to directly modify it's value. That is to say, you can't do something like PC=2430h. On the other hand, if you execute LJMP 2340h you've effectively accomplished the same thing.

It is also interesting to note that while you may change the value of PC (by executing a jump instruction, etc.) there is no way to read the value of PC. That is to say, there is no way to ask the 8051 "What address are you about to execute?" As it turns out, this is not completely true: There is one trick that may be used to determine the current value of PC. This trick will be covered in a later chapter.

The Stack Pointer (SP)

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

When you push a value onto the stack, the 8051 first increments the value of SP and then stores the value at the resulting memory location.

When you pop a value off the stack, the 8051 returns the value from the memory location indicated by SP, and then decrements the value of SP.

This order of operation is important. When the 8051 is initialized SP will be initialized to 07h. If you immediately push a value onto the stack, the value will be stored in Internal RAM address 08h. This makes sense taking into account what was mentioned two paragraphs above: First the 8051 will increment the value of SP (from 07h to 08h) and then will store the pushed value at that memory address (08h).

SP is modified directly by the 8051 by six instructions: PUSH, POP, ACALL, LCALL, RET, and RETI. It is also used intrinsically whenever an interrupt is triggered (more on interrupts later. Don't worry about them for now!).

8051 Tutorial: Addressing Modes

An "addressing mode" refers to how you are addressing a given memory location. In summary, the addressing modes are as follows, with an example of each:

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

Each of these addressing modes provides important flexibility.

Immediate Addressing

Immediate addressing is so-named because the value to be stored in memory immediately follows the operation code in memory. That is to say, the instruction itself dictates what value will be stored in memory.

For example, the instruction:

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

This instruction uses Immediate Addressing because the Accumulator will be loaded with the value that immediately follows; in this case 20 (hexidecimal).

Immediate addressing is very fast since the value to be loaded is included in the instruction. However, since the value to be loaded is fixed at compile-time it is not very flexible.

Direct Addressing

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

```
MOV A,30h
```

This instruction will read the data out of Internal RAM address 30 (hexidecimal) and store it in the Accumulator.

Direct addressing is generally fast since, although the value to be loaded isn't included in the instruction, it is quickly accessable since it is stored in the 8051's Internal RAM. It is also much more flexible than Immediate Addressing since the value to be loaded is whatever is found at the given address--which may be variable.

Also, it is important to note that when using direct addressing any instruction which refers to an address between 00h and 7Fh is referring to Internal Memory. Any instruction which refers to an address between 80h and FFh is referring to the SFR control registers that control the 8051 microcontroller itself.

The obvious question that may arise is, "If direct addressing an address from 80h through FFh refers to SFRs, how can I access the upper 128 bytes of Internal RAM that are available on the 8052?" The answer is: You can't access them using direct addressing. As stated, if you directly refer to an address of 80h through FFh you will be referring to an SFR. However, you may access the 8052's upper 128 bytes of RAM by using the next addressing mode, "indirect addressing."

Indirect Addressing

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

Indirect addressing appears as follows:

```
MOV A,@R0
```

This instruction causes the 8051 to analyze the value of the R0 register. The 8051 will then load the accumulator with the value from Internal RAM which is found at the address indicated by R0.

For example, let's say R0 holds the value 40h and Internal RAM address 40h holds the value 67h. When the above instruction is executed the 8051 will check the value of R0. Since R0 holds 40h the 8051 will get the value out of Internal RAM address 40h (which holds 67h) and store it in the Accumulator. Thus, the Accumulator ends up holding 67h.

Indirect addressing always refers to Internal RAM; it never refers to an SFR. Thus, in a prior example we mentioned that SFR 99h can be used to write a value to the serial port. Thus one may think that the following would be a valid solution to write the value ‘1’ to the serial port:

```
MOV R0,#99h      ;Load the address of the serial port
MOV @R0,#01h     ;Send 01 to the serial port -- WRONG!!
```

This is not valid. Since indirect addressing always refers to Internal RAM these two instructions would write the value 01h to Internal RAM address 99h on an 8052. On an 8051 these two instructions would produce an undefined result since the 8051 only has 128 bytes of Internal RAM.

External Direct

External Memory is accessed using a suite of instructions which use what I call "External Direct" addressing. I call it this because it appears to be direct addressing, but it is used to access external memory rather than internal memory.

There are only two commands that use External Direct addressing mode:

```
MOVX A,@DPTR
MOVX @DPTR,A
```

As you can see, both commands utilize DPTR. In these instructions, DPTR must first be loaded with the address of external memory that you wish to read or write. Once DPTR holds the correct external memory address, the first command will move the contents of that external memory address into the Accumulator. The second command will do the opposite: it will allow you to write the value of the Accumulator to the external memory address pointed to by DPTR.

External Indirect

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

```
MOVX @R0,A
```

Once again, the value of R0 is first read and the value of the Accumulator is written to that address in External RAM. Since the value of @R0 can only be 00h through FFh the project would effectively be limited to 256 bytes of External RAM. There are relatively simple hardware/software tricks that can be implemented to access more than 256 bytes of memory using External Indirect addressing; however, it is usually easier to use External Direct addressing if your project has more than 256 bytes of External RAM.

8051 Tutorial: Program Flow

When an 8051 is first initialized, it resets the PC to 0000h. The 8051 then begins to execute instructions sequentially in memory unless a program instruction causes the PC to be otherwise altered. There are various instructions that can modify the value of the PC; specifically, conditional branching instructions, direct jumps and calls, and "returns" from subroutines. Additionally, interrupts, when enabled, can cause the program flow to deviate from it's otherwise sequential scheme.

Conditional Branching

The 8051 contains a suite of instructions which, as a group, are referred to as "conditional branching" instructions. These instructions cause program execution to follow a non-sequential path if a certain condition is true.

Take, for example, the JB instruction. This instruction means "Jump if Bit Set." An example of the JB instruction might be:

```
        JB 45h,HELLO
        NOP
HELLO:
```

In this case, the 8051 will analyze the contents of bit 45h. If the bit is set program execution will jump immediately to the label HELLO, skipping the NOP instruction. If the bit is not set the conditional branch fails and program execution continues, as usual, with the NOP instruction which follows.

Conditional branching is really the fundamental building block of program logic since all "decisions" are accomplished by using conditional branching. Conditional branching can be thought of as the "IF...THEN" structure in 8051 assembly language.

An important note worth mentioning about conditional branching is that the program may only branch to instructions located withim 128 bytes prior to or 127 bytes following the address which follows the conditional branch instruction. This means that in the above example the label HELLO must be within +/- 128 bytes of the memory address which contains the conditional branching instruction.

Direct Jumps

While conditional branching is extremely important, it is often necessary to make a direct call to a given memory location without basing it on a given logical decision. This is equivalent to saying "Goto" in BASIC. In this case you want the program flow to continue at a given memory address without considering any conditions.

This is accomplished in the 8051 using "Direct Jump and Call" instructions. As illustrated in the last paragraph, this suite of instructions causes program flow to change unconditionally.

Consider the example:

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

The LJMP instruction in this example means "Long Jump." When the 8051 executes this instruction the PC is loaded with the address of NEW_ADDRESS and program execution continues sequentially from there.

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

It is worth mentioning that, aside from LJMP, there are two other instructions which cause a direct jump to occur: the SJMP and AJMP commands. Functionally, these two commands perform the exact same function as the LJMP command--that is to say, they always cause program flow to continue at the address indicated by the command. However, SJMP and AJMP differ in the following ways:

- The SJMP command, like the conditional branching instructions, can only jump to an address within +/- 128 bytes of the SJMP command.
- The AJMP command can only jump to an address that is in the same 2k block of memory as the AJMP command. That is to say, if the AJMP command is at code memory location 650h, it can only do a jump to addresses 0000h through 07FFh (0 through 2047, decimal).

You may be asking yourself, "Why would I want to use the SJMP or AJMP command which have restrictions as to how far they can jump if they do the same thing as the LJMP command which can jump anywhere in memory?" The answer is simple: The LJMP command requires three bytes of code memory whereas both the SJMP and AJMP commands require only two.

Thus, if you are developing an application that has memory restrictions you can often save quite a bit of memory using the 2-byte AJMP/SJMP instructions instead of the 3-byte instruction.

Recently, I wrote a program that required 2100 bytes of memory but I had a memory restriction of 2k (2048 bytes). I did a search/replace changing all LJMPs to AJMPs and the program shrunk downto 1950 bytes. Thus, without changing any logic whatsoever in my program I saved 150 bytes and was able to meet my 2048 byte memory restriction.

NOTE: Some quality assemblers will actually do the above conversion for you automatically. That is, they'll automatically change your LJMPs to SJMPs whenever possible. This is a nifty and very powerful capability that you may want to look for in an assembler if you plan to develop many projects that have relatively tight memory restrictions.

Direct Calls

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

When the 8051 executes an LCALL instruction it immediately pushes the current Program Counter onto the stack and then continues executing code at the address indicated by the LCALL instruction.

Returns from Routines

Another structure that can cause program flow to change is the "Return from Subroutine" instruction, known as RET in 8051 Assembly Language.

The RET instruction, when executed, returns to the address following the instruction that called the given subroutine. More accurately, it returns to the address that is stored on the stack.

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

Interrupts

An interrupt is a special feature which allows the 8051 to provide the illusion of "multi-tasking," although in reality the 8051 is only doing one thing at a time. The word "interrupt" can often be subsituted with the word "event."

An interrupt is triggered whenever a corresponding event occurs. When the event occurs, the 8051 temporarily puts "on hold" the normal execution of the program and executes a special section of code referred to as an interrupt handler. The interrupt handler performs whatever special functions are required to handle the event and then returns control to the 8051 at which point program execution continues as if it had never been interrupted.

The topic of interrupts is somewhat tricky and very important. For that reason, an entire chapter will be dedicated to the topic. For now, suffice it to say that Interrupts can cause program flow to change.

8051 Tutorial: Instruction Set, Timing, and Low-Level Info

In order to understand--and better make use of--the 8051, it is necessary to understand some underlying information concerning timing.

The 8051 operates based on an external crystal. This is an electrical device which, when energy is applied, emits pulses at a fixed frequency. One can find crystals of virtually any frequency depending on the application requirements. When using an 8051, the most common crystal frequencies are 12 megahertz and 11.059 megahertz--with 11.059 being much more common. Why would anyone pick such an odd-ball frequency? There's a real reason for it--it has to do with generating baud rates and we'll talk more about it in the Serial Communication chapter. For the remainder of this discussion we'll assume that we're using an 11.059Mhz crystal.

Microcontrollers (and many other electrical systems) use crystals to syncrhronize operations. The 8051 uses the crystal for precisely that: to synchronize it's operation. Effectively, the 8051 operates using what are called "machine cycles." A single machine cycle is the minimum amount of time in which a single 8051 instruction can be executed. although many instructions take multiple cycles.

A cycle is, in reality, 12 pulses of the crystal. That is to say, if an instruction takes one machine cycle to execute, it will take 12 pulses of the crystal to execute. Since we know the crystal is pulsing 11,059,000 times per second and that one machine cycle is 12 pulses, we can calculate how many instruction cycles the 8051 can execute per second:

11,059,000 / 12 = 921,583

This means that the 8051 can execute 921,583 single-cycle instructions per second. Since a large number of 8051 instructions are single-cycle instructions it is often considered that the 8051 can execute roughly 1 million instructions per second, although in reality it is less--and, depending on the instructions being used, an estimate of about 600,000 instructions per second is more realistic.

For example, if you are using exclusively 2-cycle instructions you would find that the 8051 would execute 460,791 instructions per second. The 8051 also has two really slow instructions that require a full 4 cycles to execute--if you were to execute nothing but those instructions you'd find performance to be about 230,395 instructions per second.

It is again important to emphasize that not all instructions execute in the same amount of time. The fastest instructions require one machine cycle (12 crystal pulses), many others require two machine cycles (24 crystal pulses), and the two very slow math operations require four machine cycles (48 crystal pulses).

NOTE: Many 8051 derivative chips change instruction timing. For example, many optimized versions of the 8051 execute instructions in 4 oscillator cycles instead of 12; such a chip would be effectively 3 times faster than the 8051 when used with the same 11.059 Mhz crystal.

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

Luckily, the 8051 includes timers which allow us to time events with high precision--which is the topic of the next chapter.

8051 Tutorial: Timers

The 8051 comes equipped with two timers, both of which may be controlled, set, read, and configured individually. The 8051 timers have three general functions:

1) Keeping time and/or calculating the amount of time between events,
2) Counting the events themselves, or
3) Generating baud rates for the serial port.

The three timer uses are distinct so we will talk about each of them separately. The first two uses will be discussed in this chapter while the use of timers for baud rate generation will be discussed in the chapter relating to serial ports.

How does a timer count?

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

Programming Tip: *Some derivative chips actually allow the program to configure whether the timers count up or down. However, since this option only exists on some derivatives it is beyond the scope of this tutorial which is aimed at the standard 8051. It is only mentioned here in the event that you absolutely need a timer to count backwards, you will know that you may be able to find an 8051-compatible microcontroller that does it.*

USING TIMERS TO MEASURE TIME

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

How long does a timer take to count?

First, it's worth mentioning that when a timer is in interval timer mode (as opposed to event counter mode) and correctly configured, it will increment by 1 every machine cycle. As you will recall from the previous chapter, a single machine cycle consists of 12 crystal pulses. Thus a running timer will be incremented:

11,059,000 / 12 = 921,583

921,583 times per second. Unlike instructions--some of which require 1 machine cycle, others 2, and others 4--the timers are consistent: They will always be incremented once per machine cycle. Thus if a timer has counted from 0 to 50,000 you may calculate:

50,000 / 921,583 = .0542

.0542 seconds have passed. In plain English, about half of a tenth of a second, or one-twentieth of a second.

Obviously it's not very useful to know .0542 seconds have passed. If you want to execute an event once per second you'd have to wait for the timer to count from 0 to 50,000 18.45 times. How can you wait "half of a time?" You can't. So we come to another important calculation.

Let's say we want to know how many times the timer will be incremented in .05 seconds. We can do simple multiplication: .05 * 921,583 = 46,079.15.

This tells us that it will take .05 seconds (1/20th of a second) to count from 0 to 46,079. Actually, it will take it .049999837 seconds--so we're off by .000000163 seconds--however, that's close enough for government work. Consider that if you were building a watch based on the 8051 and made the above assumption your watch would only gain about one second every 2 months. Again, I think that's accurate enough for most applications--I wish my watch only gained one second every two months!

Obviously, this is a little more useful. If you know it takes 1/20th of a second to count from 0 to 46,079 and you want to execute some event every second you simply wait for the timer to count from 0 to 46,079 twenty times; then you execute your event, reset the timers, and wait for the timer to count up another 20 times. In this manner you will effectively execute your event once per second, accurate to within thousandths of a second.

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

Timer SFRs

As mentioned before, the 8051 has two timers which each function essentially the same way. One timer is TIMER0 and the other is TIMER1. The two timers share two SFRs (TMOD and TCON) which control the timers, and each timer also has two SFRs dedicated solely to itself (TH0/TL0 and TH1/TL1).

We've given SFRs names to make it easier to refer to them, but in reality an SFR has a numeric address. It is often useful to know the numeric address that corresponds to an SFR name. The SFRs relating to timers are:

SFR Name	Description	SFR Address
TH0	Timer 0 High Byte	8Ch
TL0	Timer 0 Low Byte	8Ah
TH1	Timer 1 High Byte	8Dh
TL1	Timer 1 Low Byte	8Bh
TCON	Timer Control	88h
TMOD	Timer Mode	89h

When you enter the name of an SFR into an assembler, it internally converts it to a number. For example, the command:

```
MOV TH0,#25h
```

moves the value 25h into the TH0 SFR. However, since TH0 is the same as SFR address 8Ch this command is equivalent to:

```
MOV 8Ch,#25h
```

Now, back to the timers. Timer 0 has two SFRs dedicated exclusively to itself: TH0 and TL0. Without making things too complicated to start off with, you may just think of this as the high and low byte of the timer. That is to say, when Timer 0 has a value of 0, both TH0 and TL0 will contain 0. When Timer 0 has the value 1000, TH0 will hold the high byte of the value (3 decimal) and TL0 will contain the low byte of the value (232 decimal). Reviewing low/high byte notation, recall that you must multiply the high byte by 256 and add the low byte to calculate the final value. That is to say:

```
TH0 * 256 + TL0 = 1000
3 * 256 + 232 = 1000
```

Timer 1 works the exact same way, but it's SFRs are TH1 and TL1.

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

The TMOD SFR

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

The individual bits of TMOD have the following functions:

Bit	Name	Explanation of Function	Timer
7	GATE1	When this bit is set the timer will only run when INT1 (P3.3) is high. When this bit is clear the timer will run regardless of the state of INT1.	1
6	C/T1	When this bit is set the timer will count events on T1 (P3.5). When this bit is clear the timer will be incremented every machine cycle.	1
5	T1M1	Timer mode bit (see below)	1
4	T1M0	Timer mode bit (see below)	1
3	GATE0	When this bit is set the timer will only run when INT0 (P3.2) is high. When this bit is clear the timer will run regardless of the state of INT0.	0
2	C/T0	When this bit is set the timer will count events on T0 (P3.4). When this bit is clear the timer will be incremented every machine cycle.	0
1	T0M1	Timer mode bit (see below)	0
0	T0M0	Timer mode bit (see below)	0

As you can see in the above chart, four bits (two for each timer) are used to specify a mode of operation. The modes of operation are:

TxM1	TxM0	Timer Mode	Description of Mode
0	0	0	13-bit Timer.
0	1	1	16-bit Timer
1	0	2	8-bit auto-reload
1	1	3	Split timer mode

13-bit Time Mode (mode 0)

Timer mode "0" is a 13-bit timer. This is a relic that was kept around in the 8051 to maintain compatability with it's predecesor, the 8048. Generally the 13-bit timer mode is not used in new development.

When the timer is in 13-bit mode, TLx will count from 0 to 31. When TLx is incremented from 31, it will "reset" to 0 and increment THx. Thus, effectively, only 13 bits of the two timer bytes are being used: bits 0-4 of TLx and bits 0-7 of THx. This also means, in essence, the timer can only contain 8192 values. If you set a 13-bit timer to 0, it will overflow back to zero 8192 machine cycles later.

Again, there is very little reason to use this mode and it is only mentioned so you won't be surprised if you ever end up analyzing archaeic code which has been passed down through the generations (a generation in a programming shop is often on the order of about 3 or 4 months).

16-bit Time Mode (mode 1)

Timer mode "1" is a 16-bit timer. This is a very commonly used mode. It functions just like 13-bit mode except that all 16 bits are used. TLx is incremented from 0 to 255. When TLx is incremented from 255, it resets to 0 and causes THx to be incremented by 1. Since this is a full 16-bit timer, the timer may contain up to 65536 distinct values. If you set a 16-bit timer to 0, it will overflow back to 0 after 65,536 machine cycles.

8-bit Time Mode (mode 2)

Timer mode "2" is an 8-bit auto-reload mode. What is that, you may ask? Simple. When a timer is in mode 2, THx holds the "reload value" and TLx is the timer itself. Thus, TLx starts counting up. When TLx reaches 255 and is subsequently incremented, instead of resetting to 0 (as in the case of modes 0 and 1), it will be reset to the value stored in THx.

For example, let's say TH0 holds the value FDh and TL0 holds the value FEh. If we were to watch the values of TH0 and TL0 for a few machine cycles this is what we'd see:

Machine Cycle	TH0 Value	TL0 Value
1	FDh	FEh
2	FDh	FFh
3	FDh	FDh
4	FDh	FEh
5	FDh	FFh
6	FDh	FDh
7	FDh	FEh

As you can see, the value of TH0 never changed. In fact, when you use mode 2 you almost always set THx to a known value and TLx is the SFR that is constantly incremented.

What's the benefit of auto-reload mode? Perhaps you want the timer to always have a value from 200 to 255. If you use mode 0 or 1, you'd have to check in code to see if the timer had overflowed and, if so, reset the timer to 200. This takes precious instructions of execution time to check the value and/or to reload it. When you use mode 2 the microcontroller takes care of this for you. Once you've configured a timer in mode 2 you don't have to worry about checking to see if the timer has overflowed nor do you have to worry about resetting the value--the microcontroller hardware will do it all for you.

The auto-reload mode is very commonly used for establishing a baud rate which we will talk more about in the Serial Communications chapter.

Split Timer Mode (mode 3)

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

While Timer 0 is in split mode, the real Timer 1 (i.e. TH1 and TL1) can be put into modes 0, 1 or 2 normally--however, you may not start or stop the real timer 1 since the bits that do that are now linked to TH0. The real timer 1, in this case, will be incremented every machine cycle no matter what.

The only real use I can see of using split timer mode is if you need to have two separate timers and, additionally, a baud rate generator. In such case you can use the real Timer 1 as a baud rate generator and use TH0/TL0 as two separate timers.

The TCON SFR

Finally, there's one more SFR that controls the two timers and provides valuable information about them. The TCON SFR has the following structure:

Bit	Name	Bit Addres	Explanation of Function	Timer
7	TF1	8Fh	**Timer 1 Overflow**. This bit is set by the microcontroller when Timer 1 overflows.	1
6	TR1	8Eh	**Timer 1 Run**. When this bit is set Timer 1 is turned on. When this bit is clear Timer 1 is off.	1
5	TF0	8Dh	**Timer 0 Overflow**. This bit is set by the microcontroller when Timer 0 overflows.	0
4	TR0	8Ch	**Timer 0 Run**. When this bit is set Timer 0 is turned on. When this bit is clear Timer 0 is off.	0

As you may notice, we've only defined 4 of the 8 bits. That's because the other 4 bits of the SFR don't have anything to do with timers--they have to do with Interrupts and they will be discussed in the chapter that addresses interrupts.

A new piece of information in this chart is the column "bit address." This is because this SFR is "bit-addressable." What does this mean? It means if you want to set the bit TF1--which is the highest bit of TCON--you could execute the command:

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

or, since the SFR is bit-addressable, you could just execute the command:

```
SETB TF1
```

This has the benefit of setting the high bit of TCON without changing the value of any of the other bits of the SFR. Usually when you start or stop a timer you don't want to modify the other values in TCON, so you take advantage of the fact that the SFR is bit-addressable.

Initializing a Timer

Now that we've discussed the timer-related SFRs we are ready to write code that will initialize the timer and start it running.

As you'll recall, we first must decide what mode we want the timer to be in. In this case we want a 16-bit timer that runs continuously; that is to say, it is not dependent on any external pins.

We must first initialize the TMOD SFR. Since we are working with timer 0 we will be using the lowest 4 bits of TMOD. The first two bits, GATE0 and C/T0 are both 0 since we want the timer to be independent of the external pins. 16-bit mode is timer mode 1 so we must clear T0M1 and set T0M0. Effectively, the only bit we want to turn on is bit 0 of TMOD. Thus to initialize the timer we execute the instruction:

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

Timer 0 is now in 16-bit timer mode. However, the timer is not running. To start the timer running we must set the TR0 bit We can do that by executing the instruction:

```
SETB TR0
```

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

Reading the Timer

There are two common ways of reading the value of a 16-bit timer; which you use depends on your specific application. You may either read the actual value of the timer as a 16-bit number, or you may simply detect when the timer has overflowed.

Reading the value of a Timer

If your timer is in an 8-bit mode--that is, either 8-bit AutoReload mode or in split timer mode--then reading the value of the timer is simple. You simply read the 1-byte value of the timer and you're done.

However, if you're dealing with a 13-bit or 16-bit timer the chore is a little more complicated. Consider what would happen if you read the low byte of the timer as 255, then read the high byte of the timer as 15. In this case, what actually happened was that the timer value was 14/255 (high byte 14, low byte 255) but you read 15/255. Why? Because you read the low byte as 255. But when you executed the next instruction a small amount of time passed--but enough for the timer to increment again at which time the value rolled over from 14/255 to 15/0. But in the process you've read the timer as being 15/255. Obviously there's a problem there.

The solution? It's not too tricky, really. You read the high byte of the timer, then read the low byte, then read the high byte again. If the high byte read the second time is not the same as the high byte read the first time you repeat the cycle. In code, this would appear as:

```
REPEAT: MOV A,TH0
        MOV R0,TL0
        CJNE A,TH0,REPEAT
```

In this case, we load the accumulator with the high byte of Timer 0. We then load R0 with the low byte of Timer 0. Finally, we check to see if the high byte we read out of Timer 0--which is now stored in the Accumulator--is the same as the current Timer 0 high byte. If it isn't it means we've just "rolled over" and must reread the timer's value--which we do by going back to REPEAT. When the loop exits we will have the low byte of the timer in R0 and the high byte in the Accumulator.

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

Detecting Timer Overflow

Often it is necessary to just know that the timer has reset to 0. That is to say, you are not particularly interest in the value of the timer but rather you are interested in knowing when the timer has overflowed back to 0.

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

We can use this approach to cause the program to execute a fixed delay. As you'll recall, we calculated earlier that it takes the 8051 1/20th of a second to count from 0 to 46,079. However, the TFx flag is set when the timer overflows back to 0. Thus, if we want to use the TFx flag to indicate when 1/20th of a second has passed we must set the timer initially to 65536 less 46079, or 19,457. If we set the timer to 19,457, 1/20th of a second later the timer will overflow. Thus we come up with the following code to execute a pause of 1/20th of a second:

```
MOV TH0,#76       ;High byte of 19,457 (76 * 256 = 19,456)
MOV TL0,#01       ;Low byte of 19,457 (19,456 + 1 = 19,457)
MOV TMOD,#01      ;Put Timer 0 in 16-bit mode
SETB TR0          ;Make Timer 0 start counting
JNB TF0,$         ;If TF0 is not set, jump back to this same instruction
```

In the above code the first two lines initialize the Timer 0 starting value to 19,457. The next two instructions configure timer 0 and turn it on. Finally, the last instruction **JNB TF0,$**, reads "Jump, if TF0 is not set, back to this same instruction." The "$" operand means, in most assemblers, the address of the current instruction. Thus as long as the timer has not overflowed and the TF0 bit has not been set the program will keep executing this same instruction. After 1/20th of a second timer 0 will overflow, set the TF0 bit, and program execution will then break out of the loop.

Timing the length of events

The 8051 provides another cool toy that can be used to time the length of events.

For example, let's say we're trying to save electricity in the office and we're interested in how long a light is turned on each day. When the light is turned on, we want to measure time. When the light is turned off we don't. One option would be to connect the lightswitch to one of the pins, constantly read the pin, and turn the timer on or off based on the state of that pin. While this would work fine, the 8051 provides us with an easier method of accomplishing this.

Looking again at the TMOD SFR, there is a bit called GATE0. So far we've always cleared this bit because we wanted the timer to run regardless of the state of the external pins. However, now it would be nice if an external pin could control whether the timer was running or not. It can. All we need to do is connect the lightswitch to pin INT0 (P3.2) on the 8051 and set the bit GATE0. When GATE0 is set Timer 0 will only run if P3.2 is high. When P3.2 is low (i.e., the lightswitch is off) the timer will automatically be stopped.

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

USING TIMERS AS EVENT COUNTERS

We've discussed how a timer can be used for the obvious purpose of keeping track of time. However, the 8051 also allows us to use the timers to count events.

How can this be useful? Let's say you had a sensor placed across a road that would send a pulse every time a car passed over it. This could be used to determine the volume of traffic on the road. We could attach this sensor to one of the 8051's I/O lines and constantly monitor it, detecting when it pulsed high and then incrementing our counter when it went back to a low state. This is not terribly difficult, but requires some code. Let's say we hooked the sensor to P1.0; the code to count cars passing would look something like this:

```
JNB P1.0,$          ;If a car hasn't raised the signal, keep waiting
JB P1.0,$           ;The line is high which means the car is on the sensor right now
INC COUNTER         ;The car has passed completely, so we count it
```

As you can see, it's only three lines of code. But what if you need to be doing other processing at the same time? You can't be stuck in the JNB P1.0,$ loop waiting for a car to pass if you need to be doing other things. Of course, there are ways to get around even this limitation but the code quickly becomes big, complex, and ugly.

Luckily, since the 8051 provides us with a way to use the timers to count events we don't have to bother with it. It is actually painfully easy. We only have to configure one additional bit.

Let's say we want to use Timer 0 to count the number of cars that pass. If you look back to the bit table for the TCON SFR you will there is a bit called "C/T0"--it's bit 2 (TCON.2). Reviewing the explanation of the bit we see that if the bit is clear then timer 0 will be incremented every machine cycle. This is what we've already used to measure time. However, if we set C/T0 timer 0 will monitor the P3.4 line. Instead of being incremented every machine cycle, timer 0 will count events on the P3.4 line. So in our case we simply connect our sensor to P3.4 and let the 8051 do the work. Then, when we want to know how many cars have passed, we just read the value of timer 0--the value of timer 0 will be the number of cars that have passed.

So what exactly is an event? What does timer 0 actually "count?" Speaking at the electrical level, the 8051 counts 1-0 transitions on the P3.4 line. This means that when a car first runs over our sensor it will raise the input to a high ("1") condition. At that point the 8051 will not count anything since this is a 0-1 transition. However, when the car has passed the sensor will fall back to a low ("0") state. This is a 1-0 transition and at that instant the counter will be incremented by 1.

It is important to note that the 8051 checks the P3.4 line each instruction cycle (12 clock cycles). This means that if P3.4 is low, goes high, and goes back low in 6 clock cycles it will probably not be detected by the 8051. This also means the 8051 event counter is only capable of counting events that occur at a maximum of 1/24th the rate of the crystal frequency. That is to say, if the crystal frequency is 12.000 Mhz it can count a maximum of 500,000 events per second (12.000 Mhz * 1/24 = 500,000). If the event being counted occurs more than 500,000 times per second it will not be able to be accurately counted by the 8051.

8051 Tutorial: Serial Communication

One of the 8051's many powerful features is it's integrated *UART*, otherwise known as a serial port. The fact that the 8051 has an integrated serial port means that you may very easily read and write values to the serial port. If it were not for the integrated serial port, writing a byte to a serial line would be a rather tedious process requring turning on and off one of the I/O lines in rapid succession to properly "clock out" each individual bit, including start bits, stop bits, and parity bits.

However, we do not have to do this. Instead, we simply need to configure the serial port's operation mode and baud rate. Once configured, all we have to do is write to an SFR to write a value to the serial port or read the same SFR to read a value from the serial port. The 8051 will automatically let us know when it has finished sending the character we wrote and will also let us know whenever it has received a byte so that we can process it. We do not have to worry about transmission at the bit level--which saves us quite a bit of coding and processing time.

Setting the Serial Port Mode

The first thing we must do when using the 8051's integrated serial port is, obviously, configure it. This lets us tell the 8051 how many data bits we want, the baud rate we will be using, and how the baud rate will be determined.

First, let's present the "Serial Control" (SCON) SFR and define what each bit of the SFR represents:

Bit	Name	Bit Addres	Explanation of Function
7	SM0	9Fh	Serial port mode bit 0
6	SM1	9Eh	Serial port mode bit 1.
5	SM2	9Dh	Mutliprocessor Communications Enable (explained later)
4	REN	9Ch	Receiver Enable. This bit must be set in order to receive characters.
3	TB8	9Bh	Transmit bit 8. The 9th bit to transmit in mode 2 and 3.
2	RB8	9Ah	Receive bit 8. The 9th bit received in mode 2 and 3.
1	TI	99h	Transmit Flag. Set when a byte has been completely transmitted.
0	RI	98h	Receive Flag. Set when a byte has been completely received.

Additionally, it is necessary to define the function of SM0 and SM1 by an additional table:

SM0	SM1	Serial Mode	Explanation	Baud Rate
0	0	0	8-bit Shift Register	Oscillator / 12
0	1	1	8-bit UART	Set by Timer 1 (*)
1	0	2	9-bit UART	Oscillator / 32 (*)
1	1	3	9-bit UART	Set by Timer 1 (*)

Note: *The baud rate indicated in this table is doubled if PCON.7 (SMOD) is set.*

The SCON SFR allows us to configure the Serial Port. Thus, we'll go through each bit and review it's function. The first four bits (bits 4 through 7) are configuration bits.

Bits **SM0** and **SM1** let us set the *serial mode* to a value between 0 and 3, inclusive. The four modes are defined in the chart immediately above. As you can see, selecting the Serial Mode selects the mode of operation (8-bit/9-bit, UART or Shift Register) and also determines how the baud rate will be calculated. In modes 0 and 2 the baud rate is fixed based on the oscillator's frequency. In modes 1 and 3 the baud rate is variable based on how often Timer 1 overflows. We'll talk more about the various Serial Modes in a moment.

The next bit, **SM2**, is a flag for "Multiprocessor communication." Generally, whenever a byte has been received the 8051 will set the "RI" (Receive Interrupt) flag. This lets the program know that a byte has been received and that it needs to be processed. However, when SM2 is set the "RI" flag will only be triggered if the

9th bit received was a "1". That is to say, if SM2 is set and a byte is received whose 9th bit is clear, the RI flag will never be set. This can be useful in certain advanced serial applications. For now it is safe to say that you will almost always want to clear this bit so that the flag is set upon reception of *any* character.

The next bit, **REN**, is "Receiver Enable." This bit is very straightforward: If you want to receive data via the serial port, set this bit. You will almost always want to set this bit.

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

The **TB8** bit is used in modes 2 and 3. In modes 2 and 3, a total of nine data bits are transmitted. The first 8 data bits are the 8 bits of the main value, and the ninth bit is taken from TB8. If TB8 is set and a value is written to the serial port, the data’s bits will be written to the serial line followed by a "set" ninth bit. If TB8 is clear the ninth bit will be "clear."

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

TI means "Transmit Interrupt." When a program writes a value to the serial port, a certain amount of time will pass before the individual bits of the byte are "clocked out" the serial port. If the program were to write another byte to the serial port before the first byte was completely output, the data being sent would be garbled. Thus, the 8051 lets the program know that it has "clocked out" the last byte by setting the TI bit. When the TI bit is set, the program may assume that the serial port is "free" and ready to send the next byte.

Finally, the **RI** bit means "Receive Interrupt." It funcions similarly to the "TI" bit, but it indicates that a byte has been received. That is to say, whenever the 8051 has received a complete byte it will trigger the RI bit to let the program know that it needs to read the value quickly, before another byte is read.

Setting the Serial Port Baud Rate

Once the Serial Port Mode has been configured, as explained above, the program must configure the serial port’s baud rate. This only applies to Serial Port modes 1 and 3. The Baud Rate is determined based on the oscillator’s frequency when in mode 0 and 2. In mode 0, the baud rate is always the oscillator frequency divided by 12. This means if you’re crystal is 11.059Mhz, mode 0 baud rate will always be 921,583 baud. In mode 2 the baud rate is always the oscillator frequency divided by 64, so a 11.059Mhz crystal speed will yield a baud rate of 172,797.

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

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

$$\mathrm{TH1} = 256 - ((\mathrm{Crystal} / 384) / \mathrm{Baud})$$

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

$$\mathrm{TH1} = 256 - ((\mathrm{Crystal} / 192) / \mathrm{Baud})$$

For example, if we have an 11.059Mhz crystal and we want to configure the serial port to 19,200 baud we try plugging it in the first equation:

$$\begin{aligned}
&\mathrm{TH1} = 256 - ((\mathrm{Crystal} / 384) / \mathrm{Baud})\\
&\mathrm{TH1} = 256 - ((11059000 / 384) / 19200\ )\\
&\mathrm{TH1} = 256 - ((28{,}799) / 19200)\\
&\mathrm{TH1} = 256 - 1.5 = 254.5
\end{aligned}$$

As you can see, to obtain 19,200 baud on a 11.059Mhz crystal we’d have to set TH1 to 254.5. If we set it to 254 we will have achieved 14,400 baud and if we set it to 255 we will have achieved 28,800 baud. Thus we’re stuck...

But not quite... to achieve 19,200 baud we simply need to set PCON.7 (SMOD). When we do this we double the baud rate and utilize the second equation mentioned above. Thus we have:

$$\begin{aligned}
&\mathrm{TH1} = 256 - ((\mathrm{Crystal} / 192) / \mathrm{Baud})\\
&\mathrm{TH1} = 256 - ((11059000 / 192) / 19200)\\
&\mathrm{TH1} = 256 - ((57699) / 19200)\\
&\mathrm{TH1} = 256 - 3 = 253
\end{aligned}$$

Here we are able to calculate a nice, even TH1 value. Therefore, to obtain 19,200 baud with an 11.059MHz crystal we must:

1) Configure Serial Port mode 1 or 3.
2) Configure Timer 1 to timer mode 2 (8-bit auto-reload).
3) Set TH1 to 253 to reflect the correct frequency for 19,200 baud.
4) Set PCON.7 (SMOD) to double the baud rate.

Writing to the Serial Port

Once the Serial Port has been propertly configured as explained above, the serial port is ready to be used to send data and receive data. If you thought that configuring the serial port was simple, using the serial port will be a breeze.

To write a byte to the serial port one must simply write the value to the **SBUF** (99h) SFR. For example, if you wanted to send the letter "A" to the serial port, it could be accomplished as easily as:

```
MOV SBUF,#’A’
```

Upon execution of the above instruction the 8051 will begin transmitting the character via the serial port. Obviously transmission is not instantaneous--it takes a measureable amount of time to transmit. And since the 8051 does not have a serial output buffer we need to be sure that a character is completely transmitted before we try to transmit the next character.

The 8051 lets us know when it is done transmitting a character by setting the **TI** bit in SCON. When this bit is set we know that the last character has been transmitted and that we may send the next character, if any. Consider the following code segment:

```
CLR TI              ;Be sure the bit is initially clear
MOV SBUF,#’A’       ;Send the letter ‘A’ to the serial
port
JNB TI,$            ;Pause until the RI bit is set.
```

The above three instructions will successfully transmit a character and wait for the TI bit to be set before continuing. The last instruction says "Jump if the TI bit is not set to $"--$, in most assemblers, means "the same address of the current instruction." Thus the 8051 will pause on the JNB instruction until the TI bit is set by the 8051 upon successful transmission of the character.

Reading the Serial Port

Reading data received by the serial port is equally easy. To read a byte from the serial port one just needs to read the value stored in the **SBUF** (99h) SFR after the 8051 has automatically set the **RI** flag in SCON.

For example, if your program wants to wait for a character to be received and subsequently read it into the Accumulator, the following code segment may be used:

```
JNB RI,$            ;Wait for the 8051 to set the RI flag
MOV A,SBUF          ;Read the character from the serial port
```

The first line of the above code segment waits for the 8051 to set the RI flag; again, the 8051 sets the RI flag automatically when it receives a character via the serial port. So as long as the bit is not set the program repeats the "JNB" instruction continuously.

Once the RI bit is set upon character reception the above condition automatically fails and program flow falls through to the "MOV" instruction which reads the value.

8051 Tutorial: Interrupts

As stated earlier, program flow is always sequential, being altered only by those instructions which expressly cause program flow to deviate in some way. However, interrupts give us a mechanism to "put on hold" the normal program flow, execute a subroutine, and then resume normal program flow as if we had never left it. This subroutine, called an interrupt handler, is only executed when a certain event (interrupt) occurs. The event may be one of the timers "overflowing," receiving a character via the serial port, transmitting a character via the serial port, or one of two "external events." The 8051 may be configured so that when any of these events occur the main program is temporarily suspended and control passed to a special section of code which presumably would execute some function related to the event that occured. Once complete, control would be returned to the original program. The main program never even knows it was interrupted.

The ability to interrupt normal program execution when certain events occur makes it much easier and much more efficient to handle certain conditions. If it were not for interrupts we would have to manually check in our main program whether the timers had overflown, whether we had received another character via the serial port, or if some external event had occured. Besides making the main program ugly and hard to read, such a situation would make our program inefficient since we'd be burning precious "instruction cycles" checking for events that usually don't happen.

For example, let's say we have a large 16k program executing many subroutines performing many tasks. Let's also suppose that we want our program to automatically toggle the P3.0 port every time timer 0 overflows. The code to do this isn't too difficult:

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

Since the TF0 flag is set whenever timer 0 overflows, the above code will toggle P3.0 every time timer 0 overflows. This accomplishes what we want, but is inefficient. The **JNB** instruction consumes 2 instruction cycles to determine that the flag is not set and jump over the unnecessary code. In the event that timer 0 overflows, the CPL and CLR instruction require 2 instruction cycles to execute. To make the math easy, let's say the rest of the code in the program requires 98 instruction cycles. Thus, in total, our code consumes 100 instruction cycles (98 instruction cycles plus the 2 that are executed every iteration to determine whether or not timer 0 has overflowed). If we're in 16-bit timer mode, timer 0 will overflow every 65,536 machine cycles. In that time we would have performed 655 **JNB** tests for a total of 1310 instruction cycles, plus another 2 instruction cycles to perform the code. So to achieve our goal we've spent 1312 instruction cycles. So 2.002% of our time is being spent just checking when to toggle P3.0. And our code is ugly because we have to make that check every iteration of our main program loop.

Luckily, this isn't necessary. Interrupts let us forget about checking for the condition. The microcontroller itself will check for the condition automatically and when the condition is met will jump to a subroutine (called an interrupt handler), execute the code, then return. In this case, our subroutine would be nothing more than:

```
CPL P3.0
RETI
```

First, you'll notice the CLR TF0 command has disappeared. That's because when the 8051 executes our "timer 0 interrupt routine," it automatically clears the TF0 flag. You'll also notice that instead of a normal **RET** instruction we have a **RETI** instruction. The RETI instruction does the same thing as a RET instruction, but tells the 8051 that an interrupt routine has finished. You must always end your interrupt handlers with RETI.

Thus, every 65536 instruction cycles we execute the CPL instruction and the RETI instruction. Those two instructions together require 3 instruction cycles, and we've accomplished the same goal as the first example that required 1312 instruction cycles. As far as the toggling of P3.0 goes, our code is 437 times more efficient! Not to mention it's much easier to read and understand because we don't have to remember to always check for the timer 0 flag in our main program. We just setup the interrupt and forget about it, secure in the knowledge that the 8051 will execute our code whenever it's necessary.

The same idea applies to receiving data via the serial port. One way to do it is to continuously check the status of the RI flag in an endless loop. Or we could check the RI flag as part of a larger program loop. However, in the latter case we run the risk of *missing* characters--what happens if a character is received right after we do the check, the rest of our program executes, and before we even check **RI** a second character has come in. We will lose the first character. With interrupts, the 8051 will put the main program "on hold" and call our special routine to handle the reception of a character. Thus, we neither have to put an ugly check in our main code nor will we lose characters.

What Events Can Trigger Interrupts, and where do they go?

We can configure the 8051 so that any of the following events will cause an interrupt:

- Timer 0 Overflow.
- Timer 1 Overflow.
- Reception/Transmission of Serial Character.
- External Event 0.
- External Event 1.

In other words, we can configure the 8051 so that when Timer 0 Overflows or when a character is sent/received, the appropriate interrupt handler routines are called.

Obviously we need to be able to distinguish between various interrupts and executing different code depending on what interrupt was triggered. This is accomplished by jumping to a fixed address when a given interrupt occurs.

Interrupt	Flag	Interrupt Handler Address
External 0	IE0	0003h
Timer 0	TF0	000Bh
External 1	IE1	0013h
Timer 1	TF1	001Bh
Serial	RI/TI	0023h

By consulting the above chart we see that whenever Timer 0 overflows (i.e., the TF0 bit is set), the main program will be temporarily suspended and control will jump to 000BH. It is assumed that we have code at address 0003H that handles the situation of Timer 0 overflowing.

Setting Up Interrupts

By default at powerup, all interrupts are disabled. This means that even if, for example, the TF0 bit is set, the 8051 will not execute the interrupt. Your program must specifically tell the 8051 that it wishes to enable interrupts and specifically which interrupts it wishes to enable.

Your program may enable and disable interrupts by modifying the IE SFR (A8h):

Bit	Name	Bit Address	Explanation of Function
7	EA	AFh	Global Interrupt Enable/Disable
6	-	AEh	Undefined
5	-	ADh	Undefined
4	ES	ACh	Enable Serial Interrupt
3	ET1	ABh	Enable Timer 1 Interrupt
2	EX1	AAh	Enable External 1 Interrupt
1	ET0	A9h	Enable Timer 0 Interrupt
0	EX0	A8h	Enable External 0 Interrupt

As you can see, each of the 8051's interrupts has its own bit in the IE SFR. You enable a given interrupt by setting the corresponding bit. For example, if you wish to enable Timer 1 Interrupt, you would execute either:

```
MOV IE,#08h  ||  SETB ET1
```

Both of the above instructions set bit 3 of IE, thus enabling Timer 1 Interrupt. Once Timer 1 Interrupt is enabled, whenever the TF1 bit is set, the 8051 will automatically put "on hold" the main program and execute the Timer 1 Interrupt Handler at address 001Bh.

However, before Timer 1 Interrupt (or any other interrupt) is truly enabled, you must also set bit 7 of IE. Bit 7, the Global Interupt Enable/Disable, enables or disables all interrupts simultaneously. That is to say, if bit 7 is cleared then no interrupts will occur, even if all the other bits of IE are set. Setting bit 7 will enable all the interrupts that have been selected by setting other bits in IE. This is useful in program execution if you have time-critical code that needs to execute. In this case, you may need the code to execute from start to finish without any interrupt getting in the way. To accomplish this you can simply clear bit 7 of IE (CLR EA) and then set it after your time-criticial code is done.

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

```
SETB ET1
SETB EA
```

Thereafter, the Timer 1 Interrupt Handler at 01Bh will automatically be called whenever the TF1 bit is set (upon Timer 1 overflow).

Polling Sequence

The 8051 automatically evaluates whether an interrupt should occur after every instruction. When checking for interrupt conditions, it checks them in the following order:

1) External 0 Interrupt
2) Timer 0 Interrupt
3) External 1 Interrupt
4) Timer 1 Interrupt
5) Serial Interrupt

Interrupt Priorities

The 8051 offers two levels of interrupt priority: high and low. By using interrupt priorities you may assign higher priority to certain interrupt conditions.

For example, you may have enabled Timer 1 Interrupt which is automatically called every time Timer 1 overflows. Additionally, you may have enabled the Serial Interrupt which is called every time a character is received via the serial port. However, you may consider that receiving a character is much more important than the timer interrupt. In this case, if Timer 1 Interrupt is already executing you may wish that the serial interrupt itself interrupts the Timer 1 Interrupt. When the serial interrupt is complete, control passes back to Timer 1 Interrupt and finally back to the main program. You may accomplish this by assigning a high priority to the Serial Interrupt and a low priority to the Timer 1 Interrupt.

Interrupt priorities are controlled by the **IP** SFR (B8h). The IP SFR has the following format:

Bit	Name	Bit Address	Explanation of Function
7	-	-	Undefined
6	-	-	Undefined
5	-	-	Undefined
4	PS	BCh	Serial Interrupt Priority
3	PT1	BBh	Timer 1 Interrupt Priority
2	PX1	BAh	External 1 Interrupt Priority
1	PT0	B9h	Timer 0 Interrupt Priority
0	PX0	B8h	External 0 Interrupt Priority

When considering interrupt priorities, the following rules apply:

- Nothing can interrupt a high-priority interrupt--not even another high priority interrupt.
- A high-priority interrupt may interrupt a low-priority interrupt.
- A low-priority interrupt may only occur if no other interrupt is already executing.
- If two interrupts occur at the same time, the interrupt with higher priority will execute first. If both interrupts are of the same priority the interrupt which is serviced first by polling sequence will be executed first.

What Happens When an Interrupt Occurs?

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

- The current Program Counter is saved on the stack, low-byte first.
- Interrupts of the same and lower priority are blocked.
- In the case of Timer and External interrupts, the corresponding interrupt flag is set.
- Program execution transfers to the corresponding interrupt handler vector address.
- The Interrupt Handler Routine executes.

Take special note of the third step: If the interrupt being handled is a Timer or External interrupt, the microcontroller automatically clears the interrupt flag before passing control to your interrupt handler routine.

What Happens When an Interrupt Ends?

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

- Two bytes are popped off the stack into the Program Counter to restore normal program execution.
- Interrupt status is restored to its pre-interrupt status.

Serial Interrupts

Serial Interrupts are slightly different than the rest of the interrupts. This is due to the fact that there are two interrupt flags: RI and TI. If either flag is set, a serial interrupt is triggered. As you will recall from the section on the serial port, the RI bit is set when a byte is received by the serial port and the TI bit is set when a byte has been sent.

This means that when your serial interrupt is executed, it may have been triggered because the RI flag was set or because the TI flag was set--or because both flags were set. Thus, your routine must check the status of these flags to determine what action is appropriate. Also, since the 8051 does not automatically clear the RI and TI flags you must clear these bits in your interrupt handler.

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

As you can see, our code checks the status of both interrupts flags. If both flags were set, both sections of code will be executed. Also note that each section of code clears its corresponding interrupt flag. If you forget to clear the interrupt bits, the serial interrupt will be executed over and over until you clear the bit. Thus it is very important that you always clear the interrupt flags in a serial interrupt.

Important Interrupt Consideration: Register Protection

One very important rule applies to all interrupt handlers: Interrupts must leave the processor in the same state as it was in when the interrupt initiated.

Remember, the idea behind interrupts is that the main program isn't aware that they are executing in the "background." However, consider the following code:

```
CLR C          ;Clear carry
MOV A,#25h     ;Load the accumulator with 25h
ADDC A,#10h    ;Add 10h, with carry
```

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

But what would happen if right after the MOV instruction an interrupt occured. During this interrupt, the carry bit was set and the value of the accumulator was changed to 40h. When the interrupt finished and control was passed back to the main program, the ADDC would add 10h to 40h, and additionally add an additional 1h because the carry bit is set. In this case, the accumulator will contain the value 51h at the end of execution.

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

What has happened, in reality, is the interrupt did not protect the registers it used.

Restated: *An interrupt must leave the processor in the same state as it was in when the interrupt initiated.*

What does this mean? It means if your interrupt uses the accumulator, it must insure that the value of the accumulator is the same at the end of the interrupt as it was at the beginning. This is generally accomplished with a PUSH and POP sequence. For example:

```
PUSH ACC
PUSH PSW
MOV A,#0FFh
ADD A,#02h
POP PSW
POP ACC
```

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

In general, your interrupt routine must protect the following registers:

- PSW
- DPTR (DPH/DPL)
- PSW
- ACC
- B
- Registers R0-R7

Remember that PSW consists of many individual bits that are set by various 8051 instructions. Unless you are absolutely sure of what you are doing and have a complete understanding of what instructions set what bits, it is generally a good idea to *always* protect PSW by pushing and popping it off the stack at the beginning and end of your interrupts.

Note also that most assemblers (in fact, ALL assemblers that I know of) will not allow you to execute the instruction:

```
PUSH R0
```

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

Thus, if you are using any "R" register in your interrupt routine, you will have to push that register's absolute address onto the stack instead of just saying **PUSH R0**. For example, instead of PUSH R0 you would execute:

```
PUSH 00h
```

Of course, this only works if you've selected the default register set. If you are using an alternate register set, you must PUSH the address which corresponds to the register you are using.

Common Problems with Interrupts

Interrupts are a very powerful tool available to the 8051 developer, but when used incorrectly they can be a source of a huge number of debugging hours. Errors in interrupt routines are often very difficult to diagnose and correct.

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

- **Register Protection**: Make sure you are protecting all your registers, as explained above. If you forget to protect a register that your main program is using, very strange results may occur. In our example above we saw how failure to protect registers caused the main program to apparently calculate that 25h + 10h = 51h. If you witness problems with registers changing values unexpectedly or operations producing "incorrect" values, it is very likely that you've forgotten to protect registers. *ALWAYS PROTECT YOUR REGISTERS.*
- **Forgetting to restore protected values**: Another common error is to push registers onto the stack to protect them, and then forget to pop them off the stack before exiting the interrupt. For example, you may push ACC, B, and PSW onto the stack in order to protect them and subsequently pop only ACC and PSW off the stack before exiting. In this case, since you forgot to restore the value of "B", an extra value remains on the stack. When you execute the RETI instruction the 8051 will use that value as the return address instead of the correct value. In this case, your program will almost certainly crash. ALWAYS MAKE SURE YOU POP THE SAME NUMBER OF VALUES OFF THE STACK AS YOU PUSHED ONTO IT.

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

Simulators, such as Vault Information Services' 8052 Simulator for Windows, contain special features which will notify you if you fail to protect registers or commit other common interrupt-related errors.