2 Basic Registers

This chapter deals with the addressing modes, interrupts and internal peripherals (timers, serial and parallel input/output ports) of the basic 8051 and goes into more detail on the actual internal registers and how they are use in order to program and control the peripherals.

2.1 The Accumulator, Address E0H, Bit-addressable

Hex Byte Address	Bit-addressable								Symbol
E0	E7	E6	E5	E4	E3	E2	E1	E0	ACC
	ACC.7	ACC.6	ACC.5	ACC.4	ACC.3	ACC.2	ACC.1	ACC.0	Bit – ASM
	ACC^7	ACC^6	ACC^5	ACC^4	ACC^3	ACC^2	ACC^1	ACC^0	Bit – KEIL C

Table 2-1 ACC

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

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

2.2 The R registers

There are 4 banks of registers, with 8 registers, named R0, R1, R2, R3, R4, R5, R6 and R7 per bank. The default bank is Bank 0, with R0 having address 00H and R7 having address 07H.

These registers are used as auxiliary registers in many operations. To continue with the above example, suppose we are adding the numbers 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 we would use the command:

```
    ADD A, R4
```

After executing this instruction the Accumulator will contain the value 30. We may think of the R registers as some very important auxiliary or helper registers. The Accumulator alone would not be very useful if it were not for these R registers.

These registers are also used to store values temporarily. For example, let us say we 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 to the accumulator
        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 is not the most efficient way to calculate

$$(R1+R2) - (R3 +R4)$$

but it does illustrate the use of the R registers as a way of storing values temporarily. Note that we are assuming that the resultant sum of (R1+R2) and (R3+R4) fits in an 8-bit register.

2.3 The B Register, address F0H, Bit-addressable

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

Hex Byte Address	**Bit-addressable**								**Symbol**
F0	F7	F6	F5	F4	F3	F2	F1	F0	B
	B.7	B.6	B.5	B.4	B.3	B.2	B.1	B.0	Bit - ASM
	B^7	B^6	B^5	B^4	B^3	B^2	B^1	B^0	Bit - KEIL C

Table 2-2 B

The B register is only used directly by two 8051 instructions: MUL AB and DIV AB. Thus, if we want to quickly and easily multiply or divide A by another number, we 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.

2.4 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 registers.

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 the external memory, it will access it at the address indicated by DPTR.

While DPTR is most often used to point to data in external memory, many programmers often take advantage of the fact that it is the only true 16-bit register available. It is often used to store 2-byte values which have nothing to do with memory locations. Moreover, it can be used as 2 separate and independent 8-bit registers, the high byte register DPH and the low byte register DPL.

2.5 The Program Counter (PC)

The Program Counter (PC) register is not part of the SFRs. It contains a 2-byte address which tells the 8051 where the next instruction to execute is found in memory. When the 8051 is initialized, the PC is set to 0000h and is incremented each time an instruction is executed. It is important to note that PC is not always incremented by one after each instruction. This is because of the fact that some instructions require 2 or 3 bytes and the PC will therefore be incremented by 2 or 3 in these cases.

The Program Counter is special in that there is no way to directly modify its value. That is to say, we cannot code something like PC = 2430h. On the other hand, if we execute LJMP 2430h (meaning junp to location 2430 hex), we would have effectively accomplished the same thing, since the micro-controller would need to load the program counter with the address of the location where it needs to jump to and continue the execution of the code from there.

It is also interesting to note that while we may change the value of the PC (by executing a jump instruction, etc.) there is no specific direct instruction to read the value of the PC. That is to say, there is no way to ask the 8051 "What is the address of the instruction you are 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. When for example a CALL is executed, the address of the instruction after the CALL is pushed on stack (first the low byte followed by the high byte). Once it is on the stack, this address can be popped or modified at will! This trick is used extensively in the PaulOS Real-Time Operating System (RTOS) and other RTOSs in order to swap tasks.

2.6 The Stack Pointer (SP), address 81H

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

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

This order of operation is important. When the 8051 is initialized (reset), the SP will be set to 07h. If we 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 the SP (from 07h to 08h) and then it will store the pushed value at that memory address (08h).

The SP is modified directly by the 8051 by the following 6 instructions: PUSH, POP, ACALL, LCALL, RET, and RETI. It is also used intrinsically whenever an interrupt is triggered (more on interrupts in section 2.13). The SP always points to an indirectly addressable internal memory area and these instructions act as in the Indirect Addressing mode. (see section 2.7.3). They make use of or modify the contents of the indirectly addressable memory pointed to by the SP. Since the stack resides in the indirectly addressable internal memory, there is a limit to the size of stack which can be used, which is also affected by the number and type of the variables being stored in this same area.

2.7 Addressing Modes

An addressing mode refers to how we 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 Indirect	MOVX A,@DPTR
Code Indirect	MOVC A,@A+DPTR

Each of these addressing modes provides important flexibility. Moreover, the type of addressing mode also determines the memory area that is being accessed by the instruction. Reference to Table 13 would be helpful at this stage.

2.7.1 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
```

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

2.7.2 Direct Addressing – Data in Directly Addressable Internal RAM

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

```
MOV A, 30h
```

will read the data out of the Directly Addressable Internal RAM address 30 (hexadecimal) and store it in the Accumulator. Direct addressing is generally fast since, although the value to be loaded is not included in the instruction, it is quickly accessible since it is stored in the 8051's Internal Directly Addressable 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, referring to the 8051 Internal memory map, in Table 1.2.3. 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 micro-controller itself.

Certain versions of the 8051 such as the 8032 have 256 bytes (0 to FF hex) of Internal ram. The obvious question that may arise is, "If indirect addressing, an address from 80h through FFh refers to SFRs, how can we access the upper 128 bytes of Internal RAM that are available on the 8032?" The answer is: We cannot access them using direct addressing. As stated earlier, if we directly refer to an address in the range of 80h through FFh, we will be referring to an SFR. However, we may access the 8032's upper 128 bytes of RAM by using the next addressing mode, which is indirect addressing.

2.7.3 Indirect Addressing – Data in Indirectly Addressable Internal RAM

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 Indirectly Addressable Internal RAM found on an 8032 or other improved 8051 versions.

A typical instruction using Indirect addressing is the following:

```
MOV A, @R0
```

This instruction causes the 8051 to examine 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. R0 is simply acting as a pointer to an Indirectly Addressable Internal memory location.

For example, let us 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 the Indirectly Addressable Internal RAM only; it never refers to an SFR. 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 into R0
MOV @R0, #01h       ;Send 01 to the serial port – Wrong!!
```

This is not the correct way. Since indirect addressing always refers to Indirect Internal RAM these two instructions would write the value 01h to Internal RAM address 99h on an 8032. On an 8051 these two instructions would produce an undefined result since the 8051 only has 128 bytes of Internal RAM. Indirect addressing cannot therefore be used to access the SFRs, which can only be accessed using direct addressing. The correct way would therefore be:

```
MOV 99h, #01h          ;Load location 99h (serial port SBUF register location) with 01
```

or since the assembler would know that SBUF resides at address 99h

```
MOV SBUF, #01h                  ;Send 01 to the serial port SBUF register
```

2.7.4 External Indirect – 16-bit address

External Memory is accessed using a very limited number of commands. In the case of a 16-bit external data memory address, there are only two commands that can be used for External Indirect addressing mode:

```
MOVX A, @DPTR

MOVX @DPTR, A
```

The X in MOVX signifies that an External address is being used. As we can see, both commands utilize DPTR. In these instructions, DPTR must first be loaded with the address of external memory (or memory mapped device such as an 8255 input/output port chip) that we wish to read from or write to. 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 us to write the value of the Accumulator to the external memory address pointed to by DPTR.

If the address to be accessed is the Program (or Code) area, then the following commands must be used:

```
MOVC A, @A + PC
```

or

```
MOVC A, @A + DPTR
```

Here the address of the byte fetched is the sum of the original unsigned 8-bit Accumulator contents and the contents of either the 16-bit Program Counter (PC) or the 16-bit Data Pointer (DPTR). In some instances therefore, the accumulator has to be zeroed in order to use these commands.

In other cases, the value in the accumulator comes in handy when using translation or conversion tables. As a simple example assume that we have a table with the heights of a number of students (as an 8 bit integer 0–255 cms), and we want to get the height of a particular student.

The accumulator would be loaded with that student number (also in the range from 0 to 255) and DPTR would be loaded with the address of the start of the table.

Using MOVC A,@A + DPTR we can immediately get the height of that particular student loaded in the accumulator.

In conjunction with the MOVX and MOVC instructions, the micro-controller internal hardware would also set up the special control signals RD (Read), ALE (Address Latch Enable) and PSEN (Program Store Enable) which should be used by the external logic to enable the correct ROM or RAM for program and/or data access.

2.7.5 External Indirect – 8-bit

This form of addressing is usually only used in relatively small projects that have a very small (256 bytes max) amount of external data RAM. An example of this addressing mode is:

```
    MOVX @R0, A
```

Once again, the value of R0 (containg the external RAM address) 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 8-bit addressing; however, it is usually easier to use the DPTR version of addressing if the project in hand has more than 256 bytes of External RAM.

It should be noted here that if we are using C, the compiler when converting the C source program to machine code, is intelligent enough to choose the correct addressing mode to address the variables. When declaring the variable types in our C progam, the location of their storage space would also be given or implied. Thus the compiler would know in which part of memory they are being stored so that it would be able to refer to them in the correct addressing mode.

2.8 Program Flow

When an 8051 is first initialised, it resets the PC to 0000h. The 8051 then begins to execute the 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, calls to subroutines, and returns from subroutines. Additionally, interrupts, when enabled, can cause the program flow to deviate from its otherwise sequential flow.

2.8.1 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 satisfied (true).

Let us take for example, the JB instruction. This instruction means Jump if Bit Set. An example of the JB instruction might be:

```
            JB 45h, HELLO
            MOV A, #10
             …….
             …….
HELLO:       ....
```

In this case, the 8051 will analyse the contents of bit 45h. If the bit is set (1) then the program execution will jump immediately to the label HELLO, skipping the MOV A, #10 instruction and those following it. If the bit is not set (0) the conditional branch fails and the program execution continues, as usual, with the MOV A, #10 instruction which follows.

Conditional branching is really the fundamental building block of program logic since all decisions are accomplished by using conditional branching. These 8051 assembly language conditional branching instructions can be thought of as the equivalent "IF…THEN" structure found in other higher level programming languages.

An important note worth mentioning about conditional branching is that the program may only branch to instructions located within 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 +127 /-128 bytes of the memory address which contains the conditional branching instruction.

If it so happens that in our program we cannot avoid having the label HELLO occurring very far from the conditional branch address, then we can use what is referred to as a Stepping Stone. This is easily understood by following this example, where the target jump for the conditional JB instruction is actually HELLO. We instead make use of the stepping stone label HELLO2:

```
             JB 45h, HELLO2      ; use the stepping stone, to a near location
             SJMP CONTINUE       ; skip over the stepping stone
HELLO2:
             LJMP HELLO          ; now jump to the far location HELLO
CONTINUE:
             MOV A, #10
             …….
             …….
HELLO:       ....      ; this label is very far away from the JB 45h, HELLO2 location
```

2.8.2 Direct Jumps

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

This is accomplished in the 8051 using the 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 (Long Jump) instruction when executed, the PC is loaded with the address of NEW_ADDRESS location and program execution continues sequentially from there.

The obvious difference between the Direct Jump / 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 (Short Jump) and AJMP (Absolute Jump) commands. Functionally, these two commands perform exactly the 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 +127/-128 bytes of the SJMP command (hence the Short in the name).

The AJMP command can only jump to an absolute address that is in the same 2KB block of memory where the AJMP command is residing. 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).

We may ask, "Why would we want to use the SJMP or AJMP command which have restrictions as to how far they can jump, if we can just use the LJMP command which can jump to any location in memory?" The answer is simply a matter of code usage.

The LJMP command requires three bytes of code memory whereas both the SJMP and AJMP commands require only two. Thus, if we are developing an application that has memory restrictions we can often save quite a bit of memory using the 2-byte AJMP/SJMP instructions instead of the 3-byte instruction. Speed is not affected since all the three instruction types require 2 machine cycles to execute.

Suppose we are writing a program that requires 2100 bytes of memory but we have a memory restriction of 2KB (2048 bytes). If we do a simple search/replace operation to change if possible the LJMPs to SJMPs or AJMPs, the program might shrink down to an allowable size. Thus, without changing any logic whatsoever in our program, we might save enough bytes to meet our 2048 byte code memory restriction.

Some quality assemblers will actually do the above conversion for us automatically. That is, they will automatically change our LJMPs to SJMPs whenever possible. This is a very powerful capability that we may want to look for in an assembler if we plan to develop many projects that have code memory restrictions.

If we are using C, most compilers, when converting the C source program to machine code, are intelligent enough to choose the correct JMP type in order to save code space.

2.8.3 Direct Calls

Another operation that will be familiar to seasoned programmers is the LCALL or ACALL instruction. This is similar to a function call in C.

When the 8051 executes an LCALL instruction, the PC is incremented twice to obtain the address of the following instruction. It then pushes the updated Program Counter onto the stack and then continues executing code at the 16-bit address indicated by the LCALL instruction.

When the 8051 executes an ACALL instruction, the PC is incremented twice to obtain the address of the following instruction. It then pushes the updated Program Counter onto the stack and then continues executing code at the 16-bit address formed by successively concatinating the 5 high-order bits of the updated PC with the 11-bit address supplied with the ACALL instruction. The subroutine called must therefore start within the same 2KB block, since its address must have the same higher 5-bits as the updated PC.

2.8.4 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 unconditional in the sense that it always changes program flow without basing it on any condition. It is also variable in the sense that program flow can be different each time the RET instruction is executed, since this depends on the address of the CALL instruction (and the address popped on stack when the CALL was made).

2.8.5 Interrupts and RETI

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 substituted 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 (saving on stack the return address and updating the stack pointer register) and executes a special section of code referred to as an interrupt handler or an interrupt service routine (ISR) by changing the program counter contents. The interrupt handler performs whatever special functions are required to handle the event and then returns control to the 8051 (using the RETI instruction) at which point program execution continues as if it had never been interrupted (naturally some time would have been lost while executing the interrupt routine).

The topic of interrupts is somewhat tricky but very important. For that reason, an entire section (2.13) will be dedicated to the topic, but for now suffice it to say that Interrupts can cause program flow to change.

2.9 Low-Level Information

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

The 8051 operations are based on an external crystal clock. This is an electrical device which, when supplied with energy, 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 MHz or 11.059 MHz – with the latter being much more common. Why would anyone pick such an odd frequency? There is a good reason for it – it has to do with generating baud rates for the serial port and we will talk more about it in the Serial Communications section 2.12. For the remainder of this discussion, unless stated otherwise, we will assume that we are using an 11.0592 MHz crystal.

Micro-controllers (and many other electrical systems) use crystals oscillators in order to synchronise operations and the 8051 is no exception. Effectively, the 8051 operates using what are called "machine cycles". A single machine cycle is the minimum amount of time (or clock cycles) in which a single 8051 instruction can be executed, although many instructions take multiple cycles.

A machine cycle in the basic 8051 is in reality 12 pulses of the crystal clock. 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 11,059,200 times per second and that one machine cycle is 12 pulses, we can calculate how many instruction cycles the 8051 can execute in one second:

11,059,200 / 12 = 921,600

This means that the 8051 can execute 921,600 single-cycle instructions per second. Since a large number of 8051 instructions are single-cycle instructions it is often stated 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 average estimate of about 600,000 instructions per second is more realistic.

For example, if we are using exclusively 2-cycle instructions we would find that the 8051 would execute 460,800 instructions per second. The 8051 also has two really slow instructions (MUL AB and DIV AB) that require a full 4 cycles to execute – if we were to execute nothing but those instructions we would find the performance reduced to about 230,400 instructions per second.

Many 8051 derivative chips change the instruction timing. For example, many optimised versions of the 8051 execute instructions in 1 oscillator cycle instead of 12; such a chip would be effectively 12 times faster than the 8051 when used with the same 11.0592 MHz crystal. Moreover, these modern 8051 derivative micro-controllers use crystals of 22.1184 MHz or even higher, making them, overall at least 24 times faster than the standard 8051.

Since all the instructions require different amounts of time to execute a very obvious question comes to mind: How can we 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. This will be the topic of the next section.

2.10 Timers

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

- Keeping time and/or calculating the time elapsed between events.
- Counting the events themselves.
- 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 section 2.11.2.

2.10.1 How does a timer count?

The answer to this question is very simple: A timer always counts up. It does not matter whether the timer is being used as a timer, as a counter, or as a baud rate generator: a timer is always **incremented** by the micro-controller. Moreover, when the timer register reaches the upper limit, a timer flag is set (TF0 or TF1) which can be checked by the program or it can even be made to generate an interrupt. The timer then resumes counting from zero unless instructed otherwise by having it setup in the auto-reload mode.

2.10.2 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 in the following sections and then in section 2.10.15 we 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.

2.10.3 How long does a timer take to count?

First, it is 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 at every machine cycle. As you will recall from section 2.9, a single machine cycle consists of 12 crystal pulses. Thus a running timer will be incremented at the rate of

11,059,200 / 12 = 921,600 times per second

or

1/921600 seconds per count (1.0851 micro-seconds)

Unlike instructions which require 1, 2 or 4 machine cycles, the timers are consistent; they will always be incremented once per machine cycle. Even new variants of the 8051 which run very fast and use only one clock cycle per machine cycle, they all have the option to run the timers slower (dividing the clock frequency by twelve or more) so that the timings remain the same thus maintaining compatibility between different versions of the micro-controller. Thus if a timer has counted from 0 to 65535 (the maximum count of 65536 times) we may calculate the elapsed time to be:

65,536 / 921,600 = 0.0711 seconds or approximately 71 milliseconds

This would represent the maximum time we can use on a 16-bit timer. Normally we would need to execute a certain section of code say once every second, or we would need to have a delay of say 50 milliseconds. Since the timer registers can only hold integer values ranging from 0 to 65535 we should find suitable integer values which can give us some suitable delay, which we can then use to get our actual required delay (of 1 second) in our program.

So we come to another important calculation. Let us say we want to know how many times will the timer be incremented in 0.05 seconds (50 milliseconds). We can do simple multiplication:

0.05 * 921,600 = 46,080

which also happens to be an exact integer and thus we can use it in our timer to get accurate timings.

This tells us that it will take 0.05 seconds to count from 0 to 46,080. Now this is a little more useful. If we know that it takes 1/20th of a second to count from 0 to 46,080 and we want to execute some event every second we simply wait for the timer to count from 0 to 46,080 twenty times (also an exact integer); then we execute our event. We would need to reset the timer every time it reaches 46080, unless we are using the auto-reload mode as will be explained later. We would need to wait for the timer to count up another 20 times. In this manner we will effectively be executing our event once every second, accurate to within thousandths of a second.

If we are using the timer as a 16-bit (0 to 65535) timer and since as we have already stated, the timer actually counts up and moreover noting that it will set the overflow flag or even generate an interrupt when it overflows, then we would actually load the timer registers with 19456, (which is 65536–46080) so that it would take another 46080 counts in order to overflow. Thus we can have an interrupt generated every 1/20 of a second and then counting 20 interrupts before executing the required code. Otherwise, we can start the timer, wait for the overflow flag to be set by the timer, resetting the overflow flag and reloading the timer registers with 19456 and repeating the process 20 times and then proceed once the 1 second has passed.

Thus, we now have a system with which to measure time. All we need to review is how to control the timers and initialise them to provide us with the interrupt delay that we need. Figure 2-1 shows the pins, bits and SFRs which control Timer 1, and similarly for Timer 0. These will be used to configure the timers as explained further down.

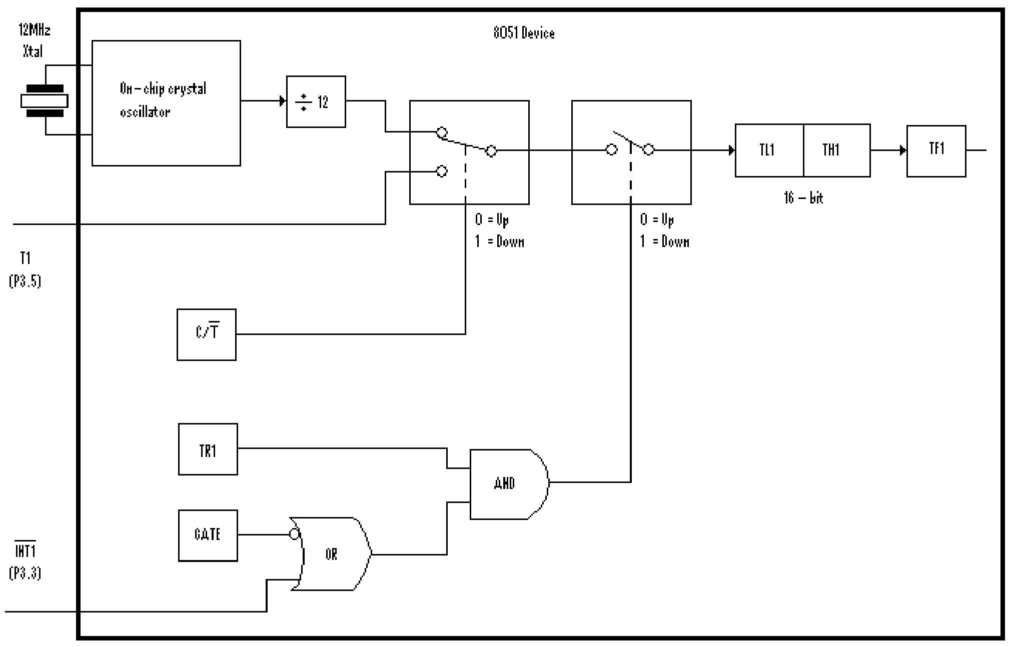

Figure 2-1 Timer/Counter 1 Mode 0 and Mode 1 operation

2.10.4 Timer SFRs

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

The SFRs have been assigned names (which all assemblers and compilers know) in order to make it easier to refer to, 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 shown in Table 2-3.

SFR Name	Description	SFR Hex Address
TH0	Timer 0 High Byte	8C
TL0	Timer 0 Low Byte	8A
TH1	Timer 1 High Byte	8D
TL1	Timer 1 Low Byte	8B
TCON	Timer Control Register	88
TMOD	Timer Mode Register	89

Table 2-3 Timer-related SFRs

When we enter the name of an SFR into an assembler, it internally converts it to its correct address. 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. First, let us talk about Timer 0 which has two SFRs dedicated exclusively to TH0 and TL0. Without making things too complicated to start off with, we may just think of these as the high and low bytes of the timer counter. 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 decimal, 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 we must multiply the high byte by 256 and add the low byte to get the final 16-bit decimal value. That is to say:

$$(TH0 * 256) + TL0 = 1000$$

$$(3 * 256) + 232 = 1000$$

Or else, knowing the final decimal value (1000), we can calculate what values we need to load into TH0 and TL0 using the following simple C instructions:

```
TH0 = 1000/256;      // (integer division, just taking the integer part of the answer)
TL0 = 1000%256;      // (modular division, just taking the remainder after dividing)
```

Certain assembler/compilers can work out these simple equations for us or else we can write our own macros. We might also use the following alternative instructions, obviously giving the same result:

```
          TH0 = 1000>>8;      //(shift the number 8 bits to the right, to get the high byte)
          TL0 = 1000 & 255;   //(bitwise AND, in order to get the lower 8 bits)
```

Timer 1 works in exactly the same way, but its SFRs are designated as TH1 and TL1.

Since there are only two bytes devoted to the value of each timer it is obvious 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. It is this overflow action which triggers the interrupt if enabled.

2.10.5 The TMOD SFR

Let us 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 this SFR gives the micro-controller specific information concerning how to run a timer. The higher four bits (bits 4 through 7) relate to Timer 1 whereas the lower four bits (bits 0 through 3) perform the exact same functions, but for Timer 0. TMOD is not bit-addressable.

The functions of the individual bits of TMOD are shown in Table 2-4.

Not Bit-addressable			
Bit	**Name**	**Timer**	**Explanation of the Timer Functions**
7	GATE	1	When this bit is set (1), Timer 1 will only run when INT1 (P3.3, EXT1) pin is high, provided that TR1 is set to 1. When this bit is cleared (0), timer 1 will run as dictated by the state of TR1, regardless of the state of INT1 pin. In each case, TR1 (in TCON) must be set to 1 for the timer to run.
6	$C/\overline{T}$	1	When this bit is set (1), Timer 1 will count events (pulses) on T1 (P3.5) pin. When this bit is cleared (0), the timer will increment every machine cycle (XTAL/12)
5	M1	1	Timer 1 mode bit (see Table 25)
4	M0	1	Timer 1 mode bit (see Table 25)
3	GATE	0	When this bit is set (1), Timer 0 will only run when INT0 (P3.2, EXT0) pin is high, provided that TR0 is set to 1. When this bit is cleared (0), timer 0 will run as dictated by the state of TR0, regardless of the state of INT0 pin. In each case, TR0 (in TCON) must be set to 1 for the timer to run.
2	$C/\overline{T}$	0	When this bit is set (1), Timer 0 will count events (pulses) on T0 (P3.4) pin. When this bit is cleared (0), the timer will increment every machine cycle (XTAL/12)
1	M1	0	Timer 0 mode bit (see Table 25)
0	M0	0	Timer 0 mode bit (see Table 25)

Table 2-4 TMOD (89H) SFR

As we can see in the Table 2-5 below, four bits (two for each timer, TM0 and TM1) are used to specify a mode of operation for the particular timer.

M1	M0	Timer Mode	Description
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

Table 2-5 Timer Mode Control bits

2.10.6 13-bit Timer Mode (mode 0)

Timer mode 0 is a 13-bit timer mode. This is a relic that was kept in the 8051 to maintain compatibility with its predecessor, the 8048. Generally the 13-bit timer mode is not used in new development projects.

When the timer is in 13-bit mode, TLx (meaning TL0 or TL1) 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 hardly any reason to use this mode and it is only mentioned so we would not be surprised if we ever end up analysing archaic code which has been passed down through generations of programmers.

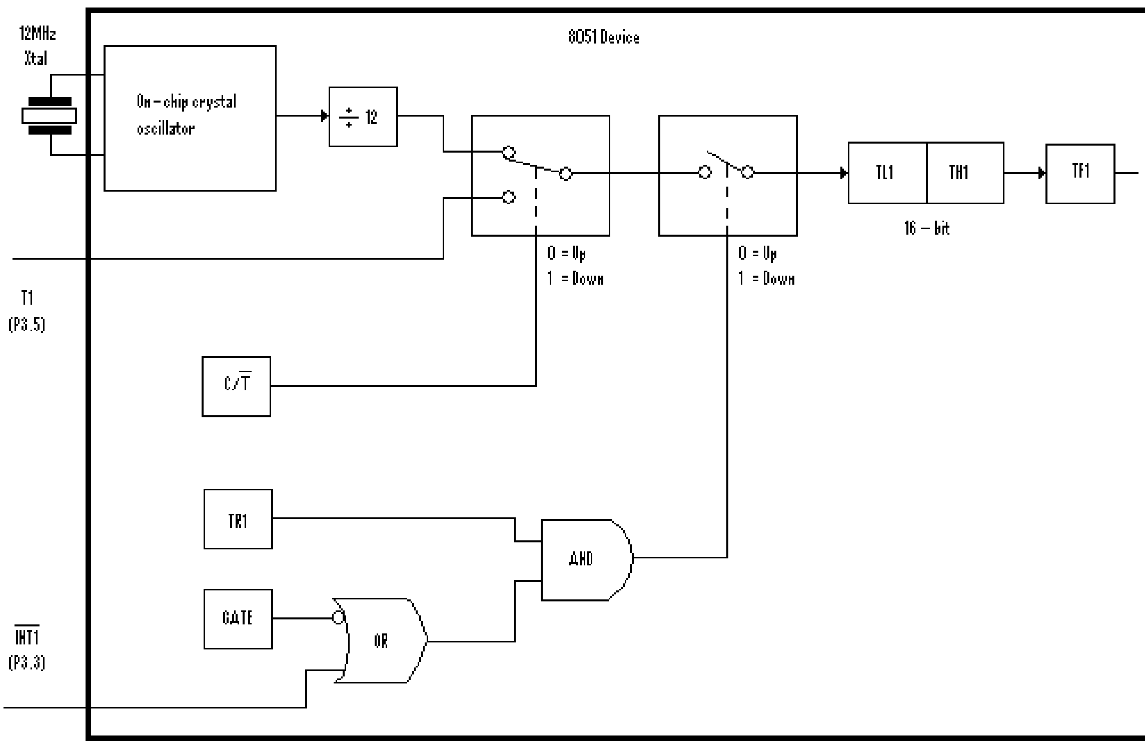

Figure 2-2 Timer 1 Mode 1

2.10.7 16-bit Timer Mode (mode 1)

Timer mode 1 is a 16-bit timer as shown in Figure 2-2 for the case of Timer 1. This is a very commonly used mode and it functions just like 13-bit mode except that all 16 bits are used.

TLx (TL0 or TL1) 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, resulting in the longest delay possible.

2.10.8 8-bit Timer Mode (mode 2)

Timer mode 2 is an 8-bit auto-reload mode, as shown in Figure 2-3 for Timer 1. When a timer is in mode 2, THx holds the reload value and TLx is the 8-bit timer register itself. Thus, TLx starts counting up and 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 reload value stored in THx.

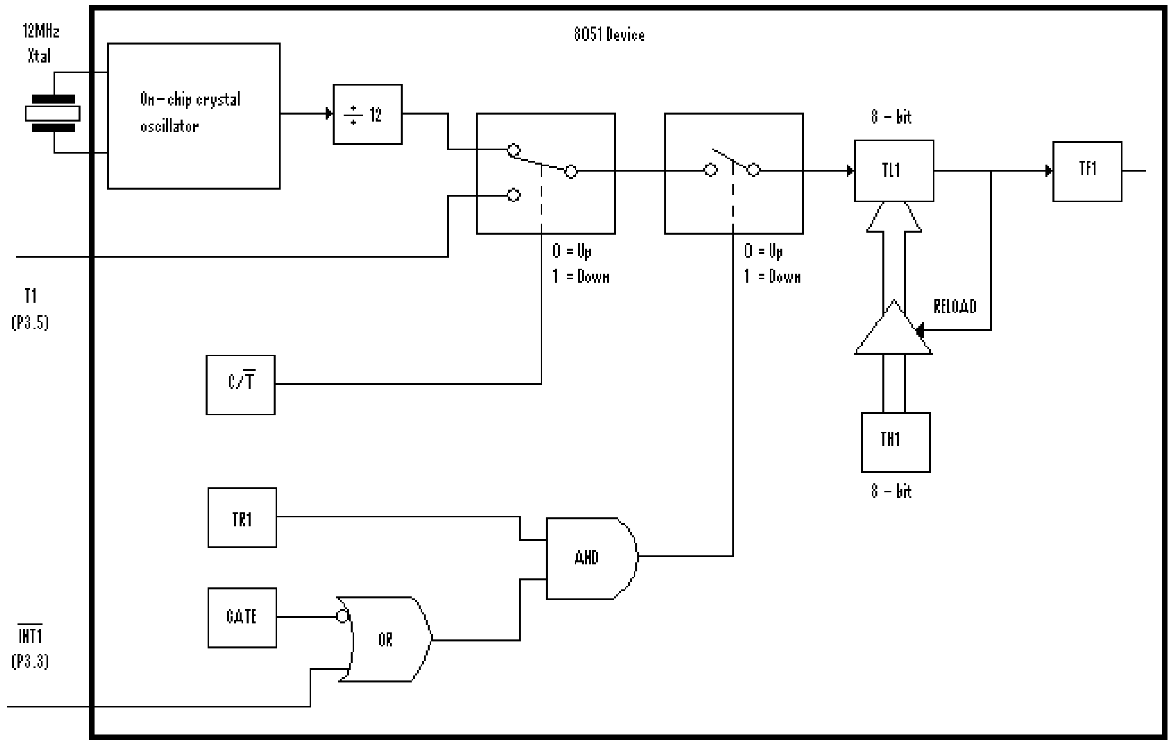

Figure 2-3 Timer 1 Mode 2

For example, let us say TH0 holds the value FDh and TL0 holds the value FDh. After 1 machine cycle, TL0 would be incremented to FEH and if we were to watch the values of TH0 and TL0 for a few machine cycles this is what we would see:

Machine Cycle	TH0 Hex Value	TL0 Hex Value
1	FD	FE
2	FD	FF
3	FD	FD
4	FD	FE
5	FD	FF
6	FD	FD
7	FD	FE

Table 2-6 Timer counters registers

As we can see, the value of TH0 never changed. In fact, when we use mode 2 we almost always set THx to a known value and TLx is the SFR that is constantly incremented. Whenever TLx overflows, the overflow flag TFx will be set, and an interrupt will be generated if so desired.

What is the benefit of auto-reload mode? Perhaps we want the timer to always have a value from 200 to 255 (i.e. we always need the timer to overflow after 56 counts) . If we use mode 0 or 1, we would have to check in code to see if the timer had overflowed and, if so, reset the timer to 200. This wastes time in checking the value and/or to reload it. When we use mode 2 the micro-controller takes care of this for us. Once we have configured a timer in mode 2 we do not have to worry about checking to see if the timer has overflowed nor do we have to worry about resetting the value; the micro-controller hardware will do it all for us.

The auto-reload mode is very commonly used for establishing a baud rate which we will talk more about in the Serial Communications chapter. It is also frequently used whenever we need to have interrupt signals at regular intervals, thus avoiding the need to reset the timer counter registers in the Interrupt Service Routine. We will expand on this later on, in the Interrupts section.

It should be remembered that for Timers 0 and 1, auto-reload is only available in 8-bit mode. Enhanced versions of the 8051, such as the 8031 have other timers which have 16-bit auto-reload capabilities.

2.10.9 Split Timer Mode (mode 3)

Timer mode 3 is a split-timer mode and can be used only with Timer 0. When Timer 0 is placed in mode 3, it essentially becomes two separate 8-bit timers. That is to say, Timer 0 runs using TL0 and a new Timer 00 running using TH0. Both timers count from 0 to 255 and overflow back to 0. The bits TR1 and TF1 that are related to the real Timer 1 will now be tied to Timer 00 and thus TH0 now controls the original Timer 1 interrupt.

While Timer 0 is in split mode, the real Timer 1 (i.e. TH1 and TL1) can be put into mode 0, 1 or 2 in the normal way but without any interrupt (since TF1 is being used by Timer 0 in mode 3), and may be started or stopped by switching it out of and into its own mode 3. An example of the timers operating in this mode is given in Appendix F.

The only real necessity of using split timer mode is if we need to have two separate timers and, additionally, a baud rate generator and we are using the original 8051 with only two timers available. In such a case we can use the real Timer 1 as a baud rate generator (usually in mode 2) and use TH0 and TL0 as two separate 8-bit timers, by setting Timer 0 in mode 3. Most modern upgrades of the 8051 family have 4 timers or more, making this mode not so really useful.

2.10.10 The TCON SFR

Finally, there is one more SFR that controls the two timers and provides valuable information about them. As we may notice, we have only defined the higher 4 (nibble) of the 8 bits. That is because the other lower 4 bits of the SFR do not have anything to do with timers – they have to do with external interrupts and they will be discussed in section 2.10.16.

The TCON SFR has the following structure, as shown in Table 2-7. A new piece of information in this table is the column bit address. This is because this SFR is bit-addressable (note that the address of the SFR is divisible by 8, hence it is bit-addressable as mentioned earlier on).

Bit-addressable						
Bit	**Name**	**Alternate Name (ASM)**	**Alternate Name (Keil C)**	**Bit Hex Address**	**Explanation Of Function**	**Timer Number**
7	TF1	TCON.7	TCON^7	8F	Timer 1 overflow flag. This bit is set by the micro-controller when timer register overflows.	1
6	TR1	TCON.6	TCON^6	8E	Timer 1 start/stop. Timer runs if this bit is set to 1.	1
5	TF0	TCON.5	TCON^5	8D	Timer 0 overflow flag. This bit is set by the micro-controller when timer register overflows.	0
4	TR0	TCON.4	TCON^4	8C	Timer 0 start/stop. Timer runs if this bit is set to 1.	0
The lower 4 bits have nothing to do with the timers as such and are not being listed here. They are used to detect and initiate external interrupts and they are discussed in a later section, under external interrupts (section 2.10.16).						

Table 2-7 TCON (88H) SFR

This bit-addressing capability means that if we want to set the bit TF1, which is the highest bit of TCON, instead of executing:

```
MOV TCON, #80h        ;(sets bit 7, and clears the other bits)
```

or

```
ORL TCON, #80h        ;(sets bit 7 only, without modifying the other bits)
```

we could just execute the command:

```
SETB TF1              ;(sets bit 7 only)
```

which is much more easy and user friendly.

This has the benefit of setting the high bit of TCON without changing the value of any of the other bits of the SFR and also it is more easily understood by anybody seeing the code. Usually when we start or stop a timer we do not want to modify the other values in TCON, so we take advantage of the fact that this SFR is bit-addressable.

2.10.11 Initialising a Timer

Now that we have discussed the timer-related SFRs we are ready to write a piece of code that will initialise the timer and start it running.

As we will recall, we must first decide what mode we want the timer to be in. In this case let us suppose that we want a 16-bit timer that runs continuously; that is to say it is not dependent on any external pin condition.

We must first initialise the TMOD SFR. Assume we are working with timer 0. We will therefore be using the lowest 4 bits of TMOD. The first two bits, GATE0 and C/$\overline{T}$ 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 initialise the timer we execute the instruction:

```
   MOV TMOD,#01h        ; sets bit 0 and clears the other bits, hence affecting Timer 1 too.
```

or

```
ANL TMOD, #0F0h     ; clears the lower T0 mode control bits, leaving T1 bits unchanged
                    ; momentarily placing Timer 0 in mode 0.
ORL TMOD, #01h      ; sets bit 0 only (mode 1), leaving the other bits unchanged.
```

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 and we can do that by executing the instruction:

```
SETB TR0;
```

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

2.10.12 Reading the Timer registers

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

2.10.13 Reading the value of a Timer register

If our timer is in an 8-bit mode, that is either 8-bit auto-reload mode or in split timer mode, then reading the value of the timer is simple. We simply read the 1-byte value of the timer register (TLx or THx depending on the mode we are in) and we are done.

However, if we are dealing with a 13-bit or 16-bit timer this gets a little more complicated. Let us suppose that the timer registers are presently loaded with the values 14 and 255 (high byte 14, low byte 255). Consider what would happen if we read the low byte first then go on to read the high byte of the timer. It could well happen that we read the low byte of the timer as 255, then read the high byte of the timer as 15. Why? We correctly read the low byte as 255, but when we executed the next instruction a small amount of time would have passed, small but long enough for the timer to increment again at which time the values of the register pairs THx,TLx would have rolled over from 14, 255 to 15, 0. But in the process we would have wrongly read the timer registers as being 15,255 and this is a problem which may well lead to complete failure of our program.

The solution is not too tricky, really. We read the high byte of the timer, then read the low byte, then read the high byte again. If the high byte which we read the second time is not the same as the high byte read the first time we repeat the cycle, because we would conclude there was a roll-over. In assembly code, this would appear as:

```
REPEAT:     MOV A, TH0                ; read THO and store it in the Accumulator
            MOV R0, TL0               ; read TL0 and store it register R0
            CJNE A, TH0, REPEAT       ; compare the new TH0 with the previous
                                      ; value and jump to REPEAT if not the same
                    ……
                    ……
```

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 that we read out of TH0 the first time, which is now stored in the Accumulator is the same as the current TH0 high byte, now read by the CJNE A, TH0, REPEAT instruction. If it is not the same, it means that we have just rolled over and must read again the timer values – which we do by going back to REPEAT. When the loop exits we will correctly have the low byte of the timer register (TL0) in R0 and the high byte (TH0) in the Accumulator.

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

2.10.14 Detecting Timer Overflow

Often it is necessary to determine when the timer has reset to 0. That is to say, we are not particularly interested in the value of the timer but rather we are interested in knowing when the timer has overflowed and starts back to 0.

Whenever a timer overflows from its highest value back to 0, the micro-controller automatically sets the TFx bit or flag in the TCON register. This is useful since rather than checking the exact value of the timer we 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. We calculated earlier that it takes the 8051 1/20th of a second to count from 0 to 46,080. However, the TFx flag is set when the timer overflows back to 0. 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 46080, or 19456. If we therefore set the timer to 19456 then 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:

```
; 50 millisecond delay

MOV TH0, #76          ; High byte of 19456 (19456/256 = 76 exactly)

                      ; You can use MOV TH0, #HIGH(19456)

MOV TL0, #00          ; Low byte of 19456 (19456%256 = 0 i.e. no remainder after dividing)

                      ; You can use MOV TLO, #LOW(19456)

ANL TMOD, #0F0h       ; clears the lower T0 mode bits, leaving T1 bits unchanged

ORL TMOD, #01         ; Put Timer 0 in 16-bit mode

CLR TF0               ; Clear the overflow flag

SETB TR0              ; Start Timer 0 in order to begin counting

JNB TF0,$             ; If TF0 is not set, jump back

                      ; to this same instruction, that is

                      ; wait here until timer overflows, and 0.05s have passed

CLR TR0       ; switch off timer 0

PROCEED: ..
```

In the above code the first two lines initialise the Timer 0 registers starting value to 19456. The next four instructions configure Timer 0, clear the overflow flag and turn the timer on. Finally, the last instruction JNB TF0,$ translates to "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, setting the TF0 bit, and program execution will then break out of this one-line loop and continues at label PROCEED.

The program can easily be modified as shown below, to get the one second delay mentioned earlier.

```
; ONE SECOND DELAY
MOV R0, #20            ; We need to count the 50ms delay twenty times
ANL TMOD, #0F0h        ; clears the lower T0 mode bits, leaving T1 bits unchanged
ORL TMOD, #01          ; Put Timer 0 in 16-bit mode

DELAY_50MS:
MOV TH0, #76           ; High byte of 19456 (19456/256 = 76 exactly)
                       ; You can use MOV TH0, #HIGH(19456)
MOV TL0, #00           ; Low byte of 19456 (19456%256 = 0 i.e. no remainder after dividing)
                       ; You can use MOV TLO, #LOW(19456)
CLR TF0                ; Clear the overflow flag
SETB TR0               ; Start Timer 0 in order to begin counting
JNB TF0,$              ; If TF0 is not set, jump back
                       ; to this same instruction, that is
                       ; wait here until timer overflows, and 0.05s have passed
CLR TR0                ; switch off timer 0
DJNZ R0, DELAY_50MS    ; repeat 0.05s delay 20 times to get the one second delay
PROCEED: ..
```

2.10.15 Timing the length of events

The 8051 provides another important option that can be used to time the length of events.

For example, let us say that we are trying to save electricity in the office and we are interested in how long a light is turned on each day. When the light is turned on, we want to measure the time that it is on. When the light is turned off we do not want to time it. One option would be to connect the light switch (voltage level suitably converted to the 0-5V DC range) to one of the pins, constantly read the pin, and turn the timer on or off using TR0 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 GATE. So far we have 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 (see Figure 2-4). All we need to do is connect the light switch (having the voltage level suitably scaled down and rectified, since obviously we cannot apply 230V AC directly to the 8051 pin) to pin INT0 (P3.2) on the 8051 and set the bits GATE and TR0 to 1. When both the GATE and TR0 are set, Timer 0 will only run if P3.2 is high. When P3.2 is low (i.e., the light switch is off) the timer will automatically be stopped.

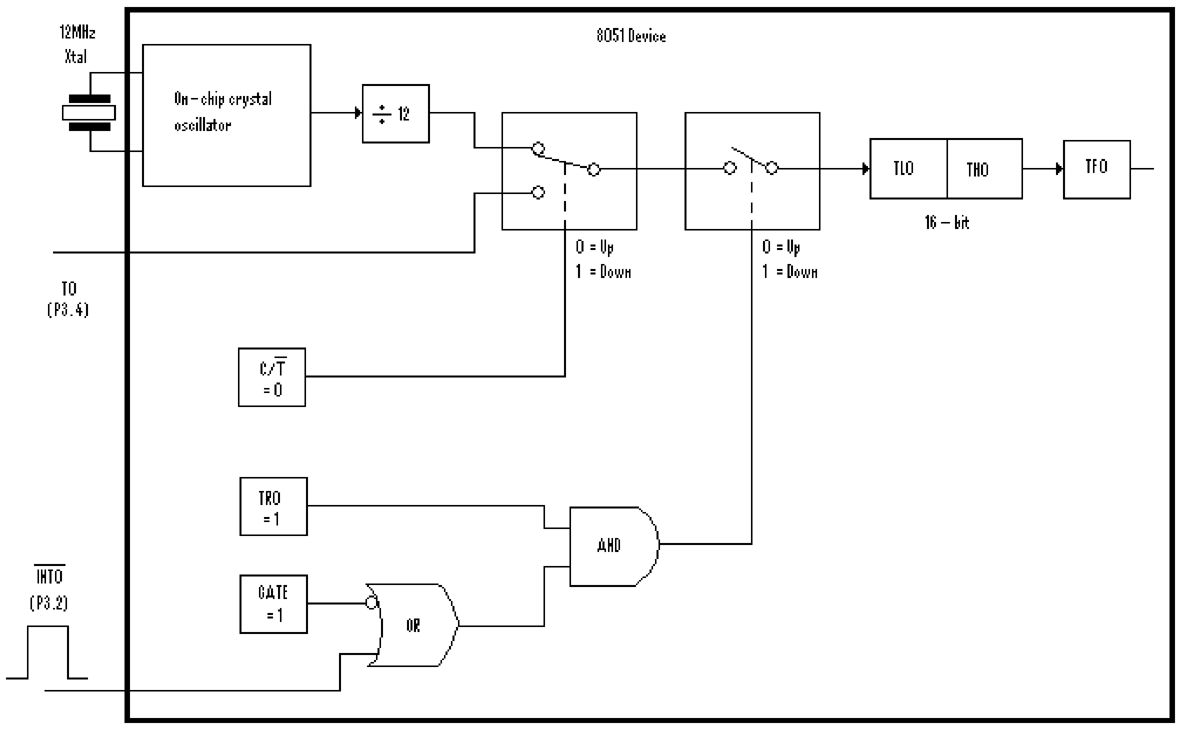

Figure 2-4 Timer 0 16-bit pulse-duration mode

Thus, with no control code whatsoever, the external pin P3.2 can control whether or not our timer is running. Naturally, our code would have to be adjusted so that we can then count also the number of overflows that have occurred so that at the end of the day we can add up the total time. This is explained in the next example.

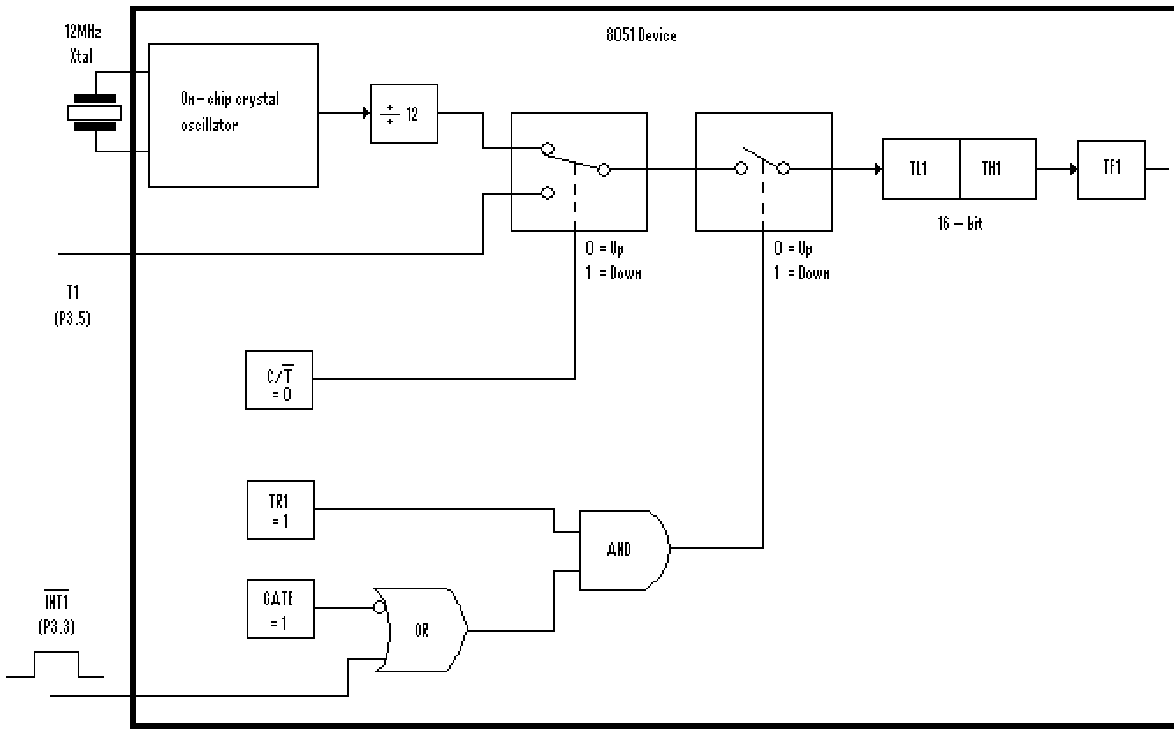

Figure 2-5 Timer 1 16-bit pulse duration mode

2.10.16 Using Timers to calculate a pulse or signal duration

The circuits shown in Figure 2-4 and in Figure 2-5 can also be used to calculate the duration of a positive pulse. Let us suppose we are using Timer 0 as a 16-bit timer (since Timer 1 would most probably be used for the serial port baud-rate generation). The positive pulse would be fed to pin P3.2 and the Gate and TR0 would be set to one. Thus due to the AND,OR logic gates the timer would effectively be operational only when there is a positive pulse on pin P3.2, and it would shut itself off as soon as the signal goes back down to zero. The counter registers TH0 and TL0 would have therefore counted the duration of the pulse. Assuming that we are using a crystal of 11.0592 MHz, and a timer counting rate of 1/12 this frequency, it would work out that every count is equivalent to 12/11.0592 microseconds (1.085 microseconds). To read the values of TH0,TL0 we would need to be monitoring the pulse so that we would know that it has finished. Thus a 16-bit timer starting from 0, would take 65536*1.085 microseconds or 71.11ms till it overflows.

It should be noted, that if the pulse lasts longer than approximately 71 ms, then the TH0,TL0 registers would overflow (setting TF0 =1) and the counter would continue counting from zero. Hence for longer pulses, the TF0 interrupt should be used so as to keep track of the number of overflows. Jumping to the interrupt service routine would automatically clear the TF0 flag for the next overflow interrupt.

However, we can even use another facility which is provided by the 8051, so that we can automate this process even further, without the need to monitor or poll the signal to know when it has finished. Since the signal is being fed directly to pin P3.2, and since this 'happens' to be the external 0 (interrupt 0) pin, we can activate the EXT0 interrupt capability of the 8051, make it falling-edge triggered (setting IT0=1, see Table **2-8**), and thus the signal itself would generate an EXT0 interrupt when it falls back to zero. Thus we would have a timer counting whilst the signal is present, TF0 interrupt being used to trigger a routine so that we can count the number of overflows (if any occur during the duration of the pulse), and EXT0 interrupt to signal the end of pulse (using an appropriate Interrupt Service Routine (ISR) to read the registers and calculate the pulse duration).

Bit-addressable						
Bit	**Name**	**Alternate Name (ASM)**	**Alternate Name (Keil C)**	**Bit Hex Address**	**Explanation Of Function**	**Timer Number**
7	TF1	TCON.7	TCON^7	8F	Timer overflow. This bit is set by the micro-controller	1
6	TR1	TCON.6	TCON^6	8E	Start(1), Stop (0) timer	1
5	TF0	TCON.5	TCON^5	8D	Timer overflow. This bit is set by the micro-controller	0
4	TR0	TCON.4	TCON^4	8C	Start(1), Stop (0) timer	0
3	IE1	TCON.3	TCON^3	8B	Ext Interrupt flag. Set/ cleared by hardware	1
2	IT1	TCON.2	TCON^2	8A	Falling edege (1), low level (0) triggering selection	1
1	IE0	TCON.1	TCON^1	89	Ext Interrupt flag. Set/ cleared by hardware	0
0	IT0	TCON.0	TCON^0	88	Falling edege (1), low level (0) triggering selection	0
The lower 4 bits are used to detect and initiate external interrupts.						

Table 2-8 TCON (88H) SFR

Here is an example of the important routines written in assembly language. The KEIL IDE provides all the necessary explanations for the keywords used, such as SEGMENT, RSEG etc and the reader is urged to consult the KEIL IDE package for further details.

```
; PulseT0.a51
; Test

MyBitData SEGMENT BIT
RSEG MyBitData
PULSE_OK: DBIT 1

MyData SEGMENT DATA
RSEG MyData

TIMER_OVF: DS 1

TIMER_OV: DS 1          ; values for use in other section of the program
TIMER_HI: DS 1          ; values for use in other section of the program
TIMER_LO: DS 1          ; values for use in other section of the program

MyCode SEGMENT CODE
RSEG MyCode
            LCALL Main

ORG 0003H     ; external 0 ivt
            LJMP EXT0_ISR

ORG 000BH     ; timer 0 ivt
            LJMP TIM0_ISR

ORG 30H
```

```
Main:
; First clear the 8032 Internal RAM (from 0 to FFH)
          CLR A
          MOV R0, #0FFH
CLR_RAM:
          MOV @R0, A
          DJNZ R0, CLR_RAM

; next set up the stack pointer
          MOV SP, #2FH

; Program starts here
; set up timer 0 for pulse width counting
          ANL TMOD, #0F0H
          ORL TMOD, #09H ; set 16-bit timer mode, gate = 1
          MOV TH0, #0
          MOV TL0, #0 ; reset 16-bit counter
          CLR TF0 ; clear the timer overflow flag
          SETB IT0 ; falling-edge triggered ext0 interrupt
          SETB P3.2 ; set p3.2 for input
          SETB TR0 ; prepare timer to count whenever P3.2 is high
          SETB EX0 ; enable external zero interrupts
          SETB ET0 ; enable timer 0 interrupts
          SETB EA ; enable global interrupts

; display pulse width values
; the number of overflows, the value of TH0 and the value of TL0
; at the end of the pulse.
; the actual duration of the pulse would be
; (65536*timer_ov + 256*TH0 + TL0) * 1.085 microseconds
;
; assuming that we have a routine called disp_reg which
; displays on screen the 8-bit value of register r0, the
; program stays looping in this loop_again routine
; prints values only if bit pulse_ok is set by the end of pulse isr
```

```
LOOP_AGAIN:
        JNB PULSE_OK, LOOP_AGAIN
        MOV R0,TIMER_OV
        LCALL DISP_REG
        MOV R0,TIMER_HI
        LCALL DISP_REG
        MOV R0,TIMER_LO
        LCALL DISP_REG
        CLR PULSE_OK
        SJMP LOOP_AGAIN

; this routine executes only every ext0 interrupt,
; that is when the pulse has just ended
; the main program would be halted and would continue only
; after this routine has finished
EXT0_ISR:
        MOV TIMER_OV, TIMER_OVF         ; store values
        MOV TIMER_HI, TH0
        MOV TIMER_LO, TL0

        MOV TIMER_OVF, #0               ; reset counters
        MOV TH0, #0
        MOV TL0, #0
        SETB PULSE_OK ; set flag indicating pulse time ready for printing
        RETI
; this interrupt service routine executes when timer0 overflows
TIM0_ISR:
        INC TIMER_OVF ; increment the overflow counter
        RETI

        end
```

And here is the same example this time written in C:

```
/* Test program PULSE */

#include <reg52.h>
#include <absacc.h>
#include <stdio.h>
#include "SerP3Pkg.h" // used for setting the UART for serial input/output

// Variables
unsigned int Timer_OVF, Timer_Overflows;
unsigned long Timer_CNT;
unsigned char Timer_HI, Timer_LO;
bit PulseOK;

void init_timer0(void)
{
   TMOD &= 0xF0;                 // clear timer 0 bits, leaving timer 1 as it was set
   TMOD |= 0x09;                 // timer 0 as a 16-bit timer, GATE = 1
   P3 |= 0x04; // set P3.2 for input
   TF0 = 0;                      // clear the overflow flag
   TR0 = 1;                      // timer 0 ready to count, whenever pin P3.2 is a 1 (pulse present)
   IT0 = 1;                      // external 0 interrupt falling-edge triggered (pulse just off)
   EX0 = 1;                      // enable external 0 interrupts
   ET0 = 1;                      // enable timer 0 interrupts
   EA = 1;                       // enable global interrupts
}

// This ISR executes when the pulse has just ended
void ext0_isr (void) interrupt 0 using 1
{
   Timer_OVF = Timer_Overflows;             /* save all timer readings */
   Timer_HI = TH0;
   Timer_LO = TL0;
   Timer_CNT = Timer_OVF*65536 + Timer_HI*256 + TL0;
   TH0 = TL0 = Timer_Overflows = 0;         /* reset all timer readings */
   PulseOK = 1; /* indicates that a NEW pulse reading has been taken */
}
```

```
// This ISR executes when timer 0 overflows (TF0 =1)
// TF0 is cleared (reset to 0) automatically when using interrupts
void tf0_isr (void) interrupt 1 using 2
{
   Timer_Overflows++;
}

void main(void)
{
         init_serial(57600);
         init_timer0();
         printf("\n\n Pulse Duration Timing\n\n\r");
         printf("\n\nToggle P3.2 to simulate pulse.\n\n\r");
         printf(" Timer Overflows TH0 TL0 Total Counts Microseconds\n\r");
   while (1)
         {
         if (PulseOK==1) {
                  printf(" %05u %03bu %03bu %010lu %12.1f\r",
                           Timer_OVF,
                           Timer_HI,
                           Timer_LO,
                           Timer_CNT,
                           1.0851*Timer_CNT); /* 12/11.0592 = 1.0851 */
                  PulseOK = 0;
         }
   }
}
```

And finally here is yet again the same example written in C, but this time making use of the UNION to make calculations even simpler.

```
/* Test program PULSE2 */

#include <reg52.h>
#include <absacc.h>
#include <stdio.h>
#include "SerP3Pkg.h"

/* types */
typedef union UTYPELONG {
            unsigned long Long;
            unsigned int Int[2];
            unsigned char Char[4];
}UTYPELONG;

// Variables

unsigned int Timer_Overflows;
UTYPELONG Timer_CNT;
bit PulseOK;

void init_timer0(void)
{
    TMOD &= 0xF0;       // clear timer 0 bits, leaving timer 1 as it was set
    TMOD |= 0x09;       // timer 0 as a 16-bit timer, GATE = 1
    P3 |= 0x04;         // set P3.2 for input
    TF0 = 0;            // clear timer overflow flag
    TR0 = 1;            // timer 0 ready to count, whenever pin P3.2 is a 1 (pulse present)
    IT0 = 1;            // external 0 interrupt falling-edge triggered (pulse just off)
    EX0 = 1;            // enable external 0 interrupts
    ET0 = 1;            // enable timer 0 interrupts
    EA = 1;             // enable global interrupts
}
```

```
void ext0_isr (void) interrupt 0 using 1
{
   Timer_CNT.Int[0] = Timer_Overflows;        /* save all timer readings */
   Timer_CNT.Char[2] = TH0;
   Timer_CNT.Char[3] = TL0;                   /* NOTE the Big Endian storage style */
   TH0 = TL0 = Timer_Overflows = 0;           /* reset all timer readings */
   PulseOK = 1;          /* indicates that a NEW pulse reading has been taken */
}

// This ISR executes when Timer 0 overflows (TF0=1)
// TF0 is cleared (reset to 0) automatically when interrupts are used
void tf0_isr (void) interrupt 1 using 2
{
   Timer_Overflows++;
}

void main(void)
{
         init_serial(57600);                  /* set up UART */
         init_timer0();                       /* set up timer 0 */

         printf("\n\n                 Pulse Duration Timing\n\n\r");
         printf("\n\nToggle P3.2 to simulate pulse.\n\n\r");
         printf("Timer Overflows TH0 TL0 Total Counts Microseconds\n\r");

   while (1)
         {
         if (PulseOK == 1) {
                  printf(" %05u %03bu %03bu %010lu %12.1f\r",
                           Timer_CNT.Int[0],
                           Timer_CNT.Char[2],
                           Timer_CNT.Char[3],
                           Timer_CNT.Long,
                           1.0851*Timer_CNT.Long); /* 12/(11.0592) = 1.0851 */
                  PulseOK = 0;
         }
   }
}
```

2.10.17 Using Timers as event counters

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

How can this be useful? Let us say we had a sensor placed across a road that would send a pulse every time a car wheel 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 us say we hooked the sensor to P1.0; the code to count cars passing would look something like this:

```
CAR:
JNB P1.0,$          ; If a car has not raised the signal, keep waiting
JB P1.0,$           ; The line is high which means the car is on the sensor right now
                    ; hence wait here until the car passes
INC COUNTER         ; The wheel has passed completely, so we count it
SJMP CAR            ; go back to wait for another car
```

As we can see, it is only four lines of code. But what if we need to be doing other processing at the same time? We do not want to be stuck in the JNB P1.0,$ loop waiting for a car to pass if we need to be doing other things. Of course, there are ways to get around 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 do not have to bother with it. It is actually easy since we only have to configure one additional bit.

Let us say we want to use Timer 0 to count the number of cars that pass. If we look back to the bit table for the TMOD SFR (see Table 2-4 and Figure 2-4) we will see that there is a bit called "$C/\overline{T0}$" – it is bit 2 (TMOD.2). Reviewing the explanation of the bit we see that if the bit is cleared then timer 0 will be incremented at every machine cycle, using the crystal oscillator. This is what we have already used in order to measure time. However, if we set $C/\overline{T0}$ to 1, then timer 0 will monitor the P3.4 line. Instead of being incremented every machine cycle, timer 0 will count events (pulses) 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 registers TL0 and TH0. This value of timer 0 will be the number of wheels that have passed. If we expect more than 65535 pulses, then we would also need to take care of how many overflows have taken place, but this too is easy since the overflows are indicated by TF0 bit being set. This can also be programmed to cause an interrupt and hence the TF0 interrupt routine simply counts the number of overflows automatically. Each overflow would indicate that 65536 wheels have passed. For this setup, TR0 is set to 1 and the GATE is set to 0. Thus TMOD = xxxx0101, setting Timer 0 in 16-bit mode.

So what exactly is an event? What does timer 0 actually count? Speaking at the electrical level, the 8051 counts 1-0 (high to low) transitions on the P3.4 line. This means that when a wheel 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 wheel has passed the sensor, the input 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. If we are really counting cars, the final value would obviously have to be divided by two since each car has got front and rear wheels, with both pairs triggering the timer count.

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 say 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 detect a maximum of 500,000 events per second (12.000 MHz * 1/24 = 500,000), even though the timer itself works at twice that frequency. If the event being counted occurs more frequently than 500,000 times per second it will not be able to be accurately detected and counted by the 8051.

2.11 Serial Port Operation

One of the 8051's many powerful features is its integrated UART, otherwise known as a serial port. The fact that the 8051 has an integrated serial port means that we 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 requiring 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. The clocking out of the bits has to be done at the pre-defined speed or baud rate.

However, we do not have to do this. Instead, we simply need to configure the serial port's operating mode and baud rate. Once configured, all we have to do is write to an SFR (SBUF) to transmit a value from the serial port (through the TXD pin P3.1) or read the same SFR to get the received value from the serial port (using the RXD pin P3.0). 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.

2.11.1 Setting the Serial Port Mode

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

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

Bit-addressable					
Bit	**Name**	**Alternate Name (ASM)**	**Alternate Name (Keil C)**	**Bit Hex Address**	**Explanation of Function**
7	SM0	SCON.7	SCON^7	9F	Serial port mode bit 0
6	SM1	SCON.6	SCON^6	9E	Serial port mode bit 1
5	SM2	SCON.5	SCON^5	9D	Multiprocessor comms enable
4	REN	SCON.4	SCON^4	9C	Receiver enable. This bit must be set in order to receive characters.
3	TB8	SCON.3	SCON^3	9B	Transmit bit 8. The 9th bit which is transmitted when operating in mode 2 or mode 3
2	RB8	SCON.2	SCON^2	9A	Receive bit 8. The 9th bit which is received when operating in mode 2 or mode 3
1	TI	SCON.1	SCON^1	99	Transmit flag. Set when a byte has been completely transmitted. Will cause a serial interrupt if the interrupts are enabled. Must be cleared by software. Can also be set by software to signal that the transmitter is ready.
0	RI	SCON.0	SCON^0	98	Receive flag. Set when a byte has been completely received. Will cause a serial interrupt if the interrupts are enabled. Must be cleared by software.

Table 2-9 SCON (99H) SFR

It is necessary to define the function of SM0 and SM1 as in Table 2-10:

SM0	SM1	Serial Mode	Explanation	Baud Rate Clock
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.

Table 2-10 Serial Mode selection bits

The SCON SFR allows us to configure the Serial Port. Thus, we will go through each bit and review its function. The higher four bits (bits 4 through 7) are the configuration bits.

Bits SM0 and SM1 set the serial mode to a value between 0 and 3 inclusive. The four modes are defined in Table 2-10. As we 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 will 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" (Receiver 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 cleared, the RI flag will never be set. This can be useful in certain advanced serial applications, where we need to communicate with one out of many microcontrollers. (see Master-Slave section 2.12.4 below). For now it is safe to say that we will almost always want to clear this bit so that the RI 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. We will almost always want to set this bit.

The lower 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 these mode a total of nine data bits are transmitted. The first 8 bits are the 8 bits of the actual data to be transmitted (taken from SBUF), and the ninth bit is taken from TB8. If TB8 is set (1) and a value is written to the serial port, the data bits will be written to the serial line followed by a "set (1)" ninth bit. If TB8 is cleared the ninth bit will be "cleared (0)". (see Master-Slave section 2.12.4 below).

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 (from the RXD pin P3.0). In this case, the first eight bits received are the data of the serial byte received (stored in SBUF) and the value of the ninth bit received will be placed in RB8. (see 2.12.4).

TI means "Transmitter 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" or transmitted out of the serial port (TXD pin P3.1). If the program were to write another byte to the serial port before the first byte was completely sent, the data being sent would be garbled. Thus, the 8051 lets the program know that it has "clocked out" the last bit 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, RI means "Receiver Interrupt." It functions 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, which would overwrite the one just received.

2.11.2 Setting the Serial Port Baud Rate

Once the Serial Port Mode has been properly 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 your crystal is 11.0592 MHz, mode 0 baud rate will always be 921,583 baud.

In mode 2 the baud rate is the oscillator frequency divided by 32 (if SMOD [PCON.7] = 1) or 64, (if SMOD [PCON.7] = 0), so an 11.0592 MHz crystal frequency will yield a baud rate of 345,600 or 172,800.

In modes 1 and 3, the baud rate is determined by how frequently timer 1 overflows. It is this timer 1 overflow frequency which is divided either by 16 or by 32 (again depending on SMOD) to give the required baud rate. 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 the 8-bit auto-reload mode (timer 1 mode 2 as already seen in Figure 2-3) and set a reload value (TH1) that causes Timer 1 to overflow at a frequency appropriate to generate one of the standard baud rates. That is, the timer must overflow at 32 (SMOD=0) or 16 (SMOD=1) times the required baud rate, or in other words, the time it takes the timer to overflow must be equal to 1/32nd (or 1/16th) the time of one serial bit.

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 cleared, that is we are dividing by 32).

Oscillator freq. = (Crystal freq. / 12)

Time for one timer count = 1/(Osc. Freq.)
= (12/Crystal freq.) seconds

Since PCON.7 is assumed to be zero, the pulses coming from the timer overflow are divided by 32 to give the correct baud rate. We must therefore ensure that we have (32*Baud Rate) overflows every second.

Time to overflow = 1/(32 * Baud Rate) seconds

Thus, we need to determine how many counts are needed to get this overflow time. By simple proportion we can work it out very easily. If we have one count in (12/Crystal freq.) seconds, how many counts do we have in 1/(32 * Baud Rate) seconds. The answer is obviously

$$\frac{1}{32{*}Baud\ Rate} * \frac{Crystsal\ frequency}{12}\ \text{counts}$$

Since the timers always count up, we must start off the timer with register TH1 set at this value below the top, or:

TH1 = 256 – ((Crystal freq.) / (32 * 12 * Baud Rate))

i.e.

TH1 = 256 – ((Crystal freq.) / (384 * Baud Rate)) ………. **Equation 2-1**

If PCON.7 is set, then the divisor is set to 16 and not to 32 and the baud rate is effectively doubled, thus the equation becomes:

TH1 = 256 – ((Crystal freq.) / (192 * Baud Rate)) …….. **Equation 2-2**

For example, suppose we have an 11.0592 MHz crystal and we want to configure the serial port to 19,200 baud. Using Equation 2.1:

TH1 = 256 – ((11059200 / 384) / 19200)

TH1 = 256 – (28800 / 19200)

TH1 = 256 – 1.5 = 254.5

This is not an integer and therefore not possible to set correctly.

If we set TH1 to 254 we will get 14,400 baud and if we set it to 255 we will get 28,800 baud. It looks like we are stuck but there is a solution.

To achieve 19,200 baud we simply need to set PCON.7 (SMOD) to 1. When we do this, we double the baud rate and use equation 2-2. Thus we get:

TH1 = 256 – ((11059200 / 192) / 19200)

TH1 = 256 – ((57600) / 19200)

TH1 = 256 – 3 = 253

Here we get an exact integer TH1 value. Therefore, to obtain 19,200 baud with an 11.0592 MHz crystal we must:

- Configure Serial Port mode 1 (8-bit variable baudrate) or 3 (9-bit variable baudrate).
- Configure Timer 1 to timer mode 2 (8-bit auto-reload).
- Set TH1 and TL1 to 253
- Set PCON.7 (SMOD) to double the baud rate.

This is in fact the reason why the oddly numbered frequency of 11.0592 MHz is chosen. This will ensure that these calculations would always give an integer value for TH1 for the standard baud rates, as shown in Table 2-11 below. This table compares some values with another crystal frequency of 12 MHz (also used often since it results in timers incrementing every one microsecond).

Target Baud Rate	Crystal Frequency MHz	PCON.7 PCON^7 SMOD	TH1 Reload Value	Actual Baud Rate	Error (%)
9600	12	1	249 (F9H) (-7)	8923	7
2400	12	0	243 (F3H) (-13)	2404	0.16
1200	12	0	230 (E6H) (-26)	1202	0.16
57600	11.0592	1	255 (FFH) (-1)	57600	0
19200	11.0592	1	253 (FDH) (-3)	19200	0
9600	11.0592	0	253 (FDH) (-3)	9600	0
2400	11.0592	0	244 (F4H) (-12)	2400	0
1200	11.0592	0	232 (E8H) (-24)	1200	0

Table 2-11 Baud Rate calculation

With the standard 11.0592 MHz crystal, the equations for calculating TH1, can be simplified to:

TH1 = 256 – (28800 / Baud Rate) if SMOD = 0 ………… **Equation 23**

TH1 = 256 – (57600 / Baud Rate) if SMOD = 1 …………. **Equation 24**

or

Baud Rate = 28800 / (256 – TH1) if SMOD = 0 ………… **Equation 25**

Baud Rate = 57600 / (256 – TH1) if SMOD = 1 ………… **Equation 26**

Once the Serial Port has been properly configured as explained above, it is ready to be used to send and receive data.

To write a byte to the serial port we must simply write the value to the SBUF (99h) SFR. For example, if we want to send the letter "A" (the 8-bit ASCII code of the letter A is 65 decimal) to the serial port, it could be accomplished simply by loading the serial buffer register SBUF:

```
MOV SBUF, #"A" or MOV SBUF, #65
```

Upon execution of the above instruction the 8051 will begin transmitting the character via the serial port, starting with the low Start bit, bit 0 to bit 7 of the actual data, followed by a high Stop bit). Obviously transmission is not instantaneous – it takes a measurable amount of time to transmit. 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 by loading a new value into SBUF.

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 cleared
        MOV SBUF, #"A"          ; Start sending the letter "A" via the serial port
  HERE: JNB TI, HERE            ; Wait here until the transmission is done, namely TI bit is set.
```

The above three instructions will successfully transmit a character and wait for the TI bit to be set before continuing. Thus the 8051 will pause on the JNB instruction until the TI bit is set by the 8051 upon successful transmission of the character. We can then transmit another character.

2.11.3 Reading the Serial Port

Reading data received by the serial port is equally easy. To read a byte from the serial port we just need to read the value stored in the SBUF (99h) SFR after the 8051 has automatically set the RI flag in SCON to indicate that a character has just been received.

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

```
HERE: JNB RI, HERE      ; Wait here for the 8051 to set the RI flag
                        ; (wait for the reception of a character)
      MOV A, SBUF       ; Read the character from the serial port
      CLR RI            ; clear RI, ready for the next character to be received
```

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 "HERE: JNB RI, HERE" instruction continuously.

Once the RI bit is set upon character reception the above condition automatically fails and program flow falls through to the "MOV A, SBUF" instruction which reads the value and stores it in the accumulator. The RI flag is finally cleared so as to be ready for the next possible character to be received.

Section 3.2 describes a complete serial port example for transmitting and receiving text.

2.11.4 Master-Slave Operation

Mode 2 or mode 3, (9-bit mode), is generally used whenever inter micro-controller communication is desired. Generally speaking, mode 2 is used for high speed communications (up to 345600 baud with an 11.0592 MHz crystal, without using timers), and mode 3 is used when standard baud rates (using timer 1) are required.

For 9-bit mode 2, the baud rates are determined directly by the crystal frequency (assumed 11.0592 MHz) and the value of SMOD (PCON.7) as shown in the Table 2-12.

SMOD	Crystal divisor	Baud Rate (=Xtal/divisor)
0	64	172800
1	32	345600

Table 2-12 Crystal Divisor

When using the 9-bit mode, one micro-controller is generally configured to act as the master controller, with up to 256 other slave micro-controllers. They can be connected in a 3-wire setup, for two-way communication as shown in Figure 2-6, which shows the setup for a master board with two slave boards.

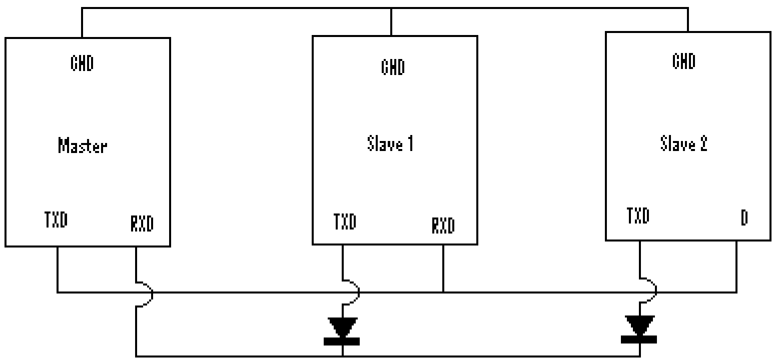

Figure 2-6 Master-slaves connection

The diodes are needed since all the slave TXD lines of the slaves are connected together and they are acting as output lines, feeding into the RXD line of the Master. Therefore one slave cannot send data ***into*** another TXD line of another slave and the diodes ensure that the TXD lines act only as output lines and do not sink any other signals. The switching time of these diodes will restrict the transmission speed of the slaves to the master.

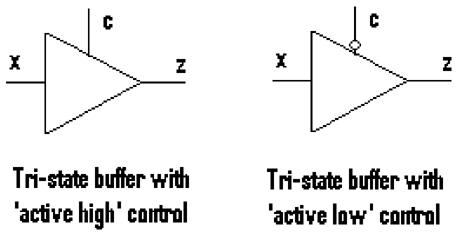
c
x
z
Tri-state buffer with 'active high' control
c
x
z
Tri-state buffer with 'active low' control

Figure 2-7 Tri-state buffers

Instead of these diodes, we can also use tri-state buffers (see Figure 2-7) which have the capability (when not in operation) to offer a high-impedance. Thus when a slave is not transmitting, it disables the tri-state buffer so that its TXD line is effectively isolated from the network. When a slave wants to transmit, it would enable the tri-state buffer so that the signal present on its TXD line is transferred on to the network. It would disable the tri-state buffer once it has finished with the transmission. In order to control the tri-state buffer, an additional pin from some port of the slave micro-controller has to be used in order to enable/disable the tri-state buffer. Depending on whether it is using active high or low control, a 1 or a 0 at the port pin would enable the buffer.

c	z
0	Z
1	x

Table 2-13 Tri-state truth table

As it can be seen in Table 2-13, when **c** = **1** the tri-state active-high device is active and **z** = **x**, that is the output z connected to the network would reflect the TXD signal at x. When **c** = **0** the tri-state device is not active, and **z** = **Z** (e.g., high impedance/no current).

If the slave units are not required to communicate back or acknowledge, then the diodes and their TXD lines can be left unconnected, and we just have a two-wire set up, with just the Ground and the TXD line from the master to the RXD pin of all the slave units.

Use is made of SM2, TB8 and RB8 as explained in the flow sequence below, which explains in detail how this master-slave communication can be achieved.

- All slave units have their UART serial device set to work under interrupt control, and with SM2 initially set 1. This means that their UART interrupt service routine (ISR) will only be called when the 9th received bit (bit number 8, since we normally count from bit 0) is a 1.
- All devices are set in 9-bit mode 2 or mode 3, depending on the required baud rates. Mode 3 is the most commonly used, especially if the diodes are being used. A slow baudrate would be required otherwise the diodes would not have enough time to pass through the data. TB8 is initially cleared, set to 0.
- All slave units are given (by means of a software #define statement) a particular unique 8-bit address (0–255).

- The master starts a transmission (not necessarily under interrupt control) by sending an address corresponding to the slave to which it wants to send data. When sending this address, the master sets its own TB8 bit to 1, so that the master is effectively adding on a '1' bit to the address it is sending. SM2 for the master is left cleared (=0).
- All slave units receive this address with the extra '1' bit and since they would at this stage all have their own SM2 bit set to 1, each one of the slave units would get a serial RI interrupt. The ISR would be activated and each slave unit would receive, read and check the address to see whether the master is intending to communicate with it.
- Only the slave unit whose address corresponds to the received address would be taking further action. The other slave units would simple return from their own ISR without doing or changing anything. They would simply wait for another address (with a 9th '1' bit) to be received.
- The slave unit with the correct address would now set its own SM2 to 0, so that from now on and until SM2 is changed again to 1, its own serial ISR would come into action for every data byte sent by the master (even if with a 9th bit of '0').
- The master, after sending the slave address, depending on the software algorithm could:
  - Either wait for an acknowledgement from the addressed slave.
  - Or just wait a while to give time for the slave to change its setup (mainly setting SM2 to 0).
- The master, after this waiting period, would set its own TB8 to 0 so that when sending the data over, it will affix a '0' at the end of each data byte, so as not to be interpreted as an address and trigger the serial interrupt of the other slave units. Some pre-arranged 'end of data' character would be sent at the end of all the data, as an indication to the addressed slave unit that no more data is going to be sent.
- Only the addressed slave unit would be interrupted to receive this data, since it would be the only slave controller with its SM2 reset to 0. The other slaves would not even notice that there is data passing, since their RI bit would not be set with bytes having a 9th bit of '0', and hence their own ISR would not be triggered.
- When the addressed slave unit receives the 'end of data' marker, it would once again revert back to the original mode, by setting SM2 to 1, and the transmission would pause. All the slave units would now once again be waiting for an address to be sent by the master board.
- Naturally, whilst waiting for the serial ISR to be activated, the slave units could be executing some other code for their particular application, rather than staying idle.

In general, address 255 is reserved for a 'general call' to be used whenever the master needs to send data to ALL slave units (such as an emergency switch off). Every slave unit would programmed to recognise this address, and all slave units would then switch their SM2 to 0 and react to this general transmission. No acknowledgement is sent by the slave units, otherwise there would simply be rubbish on the TxD line since every slave unit would be transmitting the acknowledgement at the same time.

We can also have special group addresses so that the master can send data simultaneously to a group of slave units, again without any acknowledgement coming from them.

This 3-wire multiprocessor communication is very effective over short distances and very easy to implement. A sample program is also given in the appendix.

2.12 Interrupts

As the name implies, an interrupt is some event which interrupts normal program execution.

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 by means of which we can "put on hold" the normal program flow, execute a subroutine, and then resume normal program flow as if we had never left it. This subroutine, called an interrupt handler or an interrupt service routine (ISR), is only executed when a certain event (interrupt) occurs. The event may be any of the following:

- one of the timers overflowing,
- receiving a character via the serial port,
- transmitting a character via the serial port,
- one of two external events, normally pulses on dedicated pins.

The 8051 may be configured so that when any of these events occur the main program is temporarily suspended and control is passed to a special section of code or interrupt service routine (ISR) which presumably would execute some function related to the event that has just occurred. Once the ISR is completed, control would be returned to the original program. The main program so to speak, would never even know that 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 events. If it were not for interrupts we would have to repeatedly check in our main program whether the timers had over-flowed, or whether we have received another character via the serial port, or if some external event has occurred. Besides making the main program ugly and hard to read, such a situation would make our program inefficient since we would be using precious instruction cycles, regularly and frequently checking for events even if they do not happen so frequently.

For example, let us say we have a program executing many subroutines performing many tasks. Let us also suppose that we want our program to automatically toggle the P3.0 port every time timer 0 overflows. The code to do this without using interrupts would look something like this:

```
TOP:    . . .
. . .
                JNB TF0, SKIP_TOGGLE        ; check for timer overflow here (2 cycles)
                CPL P3.0                    ; toggle the bit (1 machine cycle)
                CLR TF0                     ; clear the overflow flag to be ready for
                                            ; the next overflow (1 cycle)

 SKIP_TOGGLE:
                 . . .                      ; assume that 98 cycles are involved here
                . . .
                JMP TOP                     ; loop endlessly
```

"I studied English for 16 years but... ...I finally learned to speak it in just six lessons"
Jane, Chinese architect
ENGLISH OUT THERE
Click to hear me talking before and after my unique course download

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 machine 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 an additional 2 machine cycles to execute. To make the arithmetic easy, let us say that the rest of the code (until JMP TOP) in the program requires 98 machine cycles. Thus in total, our code consumes 100 machine cycles (98 instruction cycles plus the 2 that are executed at every iteration to determine whether or not timer 0 has overflowed). If we are in 16-bit timer mode, timer 0 will overflow every 65,536 machine cycles. In the time between overflows we would have performed 65536/100 or 655 JNB tests, consuming 1310 machine cycles plus another 2 machine cycles to perform the code when there is the overflow. So to achieve our goal, we have spent 1312 out of 65536 or 2% of the time just checking when to toggle P3.0. Moreover, we would not be reacting immediately to the overflow since we would only notice it when we come to the check instruction. And our code is not efficient because we have to make that check during every iteration of our main program loop.

Luckily with interrupts this is not necessary and we can forget about checking for the overflow condition. The micro-controller itself will check for the condition automatically *after every instruction* (thus the reaction is much quicker) and when the condition is met it will jump to a subroutine, execute the code, then return to where it was before the interrupt. In this case, our subroutine would be nothing more than:

```
CSEG AT 000BH        ; this ensures routine is written in
                     ; the correct vector table location for Timer 0 interrupt
CPL P3.0
RETI
```

First, it should be noted that the ISR has to be located at a specified code location, depending on the interrupt being used. (see section 2.13.1).

Secondly, it can be noticed that the CLR TF0 command has disappeared. That is because when the 8051 executes our "timer 0 interrupt routine," it automatically clears the TF0 flag which had originally generated the interrupt. Also instead of a normal RET instruction we have a RETI instruction. The RETI instruction does the same thing as a RET instruction (that is it pops the high- and low-order bytes of the program counter successively from the stack), but it also tells the 8051 that an interrupt routine has finished so that it would restore the interrupt logic to accept additional interrupts at the same priority level as the one just processed. *We must always end our interrupt service routines with RETI instruction.*

Thus, every 65536 instruction cycles (when timer 0 overflows), control is transferred to the ISR automatically at the end of the current instruction, and the CPL and the RETI instructions are executed only once. These two instructions together require 3 instruction cycles, and we have 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 the fact that it is much easier to read and understand because we do not have to remember to always check for the timer 0 flag in our main program. We just set up the interrupt and forget about it, secure in the knowledge that the 8051 will execute our code whenever it is 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 again 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 do we lose characters.

2.12.1 What Events Can Trigger Interrupts?

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 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.

By consulting Table 2-14 it can be seen 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 some code written at address 000BH that handles the situation of Timer 0 overflowing.

Interrupt Name	Interrupt Number	Flag	Interrupt Hex Vector Address
External 0	0	IE0	0003
Timer 0	1	TF0	000B
External 1	2	IE1	0013
Timer 1	3	TF1	001B
Serial	4	RI or TI	0023

Table 2-14 8051 Interrupt Vector Table location

The Interrupt Vector Addresses shown in this table indicate the location where the ISR code for that particular interrupt should be written. Only 8 bytes are allocated for every interrupt (provided that one is using them all) and so if the ISR requires more than 8 bytes, then one would simply write

```
LJMP MY_ISR
```

at the Interrupt Vector Address and then at MY_ISR (located elsewhere in the main code area) we can write our routine which can be of any length. This is also shown in the A51 template in the A51 examples in Chapter 3.

It should also be noted here, that both RI and TI interrupts cause the program to jump to the same address (0023H). Hence it is up to the ISR to check which event (RI or TI) caused the interrupt and subsequently clear the appropriate RI or TI flag. These are not cleared automatically by the controller.

2.12.2 Setting Up Interrupts

By default at power up, all interrupts are disabled. This means that even if, for example, the TF0 bit is set, the 8051 will not execute the interrupt service routine

Bit-addressable					
Bit	**Name**	**Alternate Name (ASM)**	**Alternate Name (Keil C)**	**Hex Bit Address**	**Explanation of Function**
7	EA	IE.7	IE^7	AF	Global Interrupt Enable/Disable
6	-	IE.6	IE^6	AE	Undefined on the 8051
5	-	IE.5	IE^5	AD	Undefined on the 8051
4	ES	IE.4	IE^4	AC	Enable/Disable Serial Interrupt
3	ET1	IE.3	IE^3	AB	Enable/Disable Timer 1 Interrupt
2	EX1	IE.2	IE^2	AA	Enable/Disable External 1 1nterrupt
1	ET0	IE.1	IE^1	A9	Enable/Disable Timer 0 Interrupt
0	EX0	IE.0	IE^0	A8	Enable/Disable External 0 Interrupt

Table 2-15 IE (A8H) SFR

Our program must specifically tell the 8051 that it wishes to enable interrupts and specifically which interrupts it wishes to enable. We can do this by modifying the IE SFR (A8h), setting the corresponding bits accordingly. As we can see in Table 2-15, each of the 8051's interrupts has its own bit in the IE SFR. We enable a given interrupt by setting the corresponding bit to 1. For example, if we wish to enable Timer 1 Interrupt only, we would execute either:

```
ORL IE, #08h
```

or

```
SETB ET1
```

Each 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. In C, this would simply be one ET1 = 1; line.

However, before Timer 1 Interrupt (or any other interrupt) is truly enabled, we must also set bit 7 of IE. (SETB EA). This bit, the Global Interrupt Enable/Disable bit, enables or disables all interrupts simultaneously. That is to say, if bit 7 is cleared then no interrupt servicing 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 we have time-critical code that needs to execute. In this case, we may need the code to execute from start to finish without any interrupt getting in the way. To accomplish this we can simply clear bit 7 of IE (CLR EA) just before starting the critical code and then set it again to 1 after our time-critical code has been executed.

We should also clear the interrupt flag initially, just to be sure that we start at the right condition. 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 instructions (assuming we have already written and properly stored the corresponding ISR):

```
    CLR TF1

    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). Moreover, the TF1 bit will automatically be cleared once the ISR is being executed.

2.12.3 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, as shown in Table 2-16:

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

Table 2-16 Polling Sequence Order

This means that if a Serial Interrupt occurs at the exact same instant that an External 0 Interrupt occurs, the External 0 ISR will be executed first and the Serial ISR will be executed only when the External 0 ISR has been completed. This order is only respected in the extreme case that interrupts happen exactly at the same time. It should be remembered, that interrupts having the same priority cannot interrupt each other, irrespective of the polling sequence order.

2.12.4 Interrupt Priorities

The 8051 offers two levels of interrupt priority: high and low. By using interrupt priorities we may assign a higher priority to certain important or critical interrupt conditions.

For example, we may have enabled Timer 0 Interrupt which is automatically called every time Timer 0 overflows. Additionally, we may have enabled the Serial Interrupt which is called every time a character is received via the serial port. However, we may consider that receiving a character is much more important than the timer interrupt. In this case, if Timer 0 Interrupt is already executing we may wish that the serial interrupt itself interrupts the Timer 0 ISR if it happens to be executing. When the serial interrupt is complete, control passes back to Timer 0 ISR to continue from where it had been stopped and finally back to the main program. We may accomplish this by assigning a high priority to the Serial Interrupt and a low priority to the Timer 0 Interrupt.

Interrupt priorities are controlled by the IP SFR (B8h), where setting the bit to one raises the priority of that particular interrupt. The IP SFR has the following format:

Bit-addressable					
Bit	**Name**	**Alternate Name (ASM)**	**Alternate Name (Keil C)**	**Bit Hex Address**	**Explanation of Function**
7	-	IP.7	IP^7	-	Undefined - future expansion
6	-	IP.6	IP^6	-	Undefined - future expansion
5	-	IP.5	IP^5	-	Undefined - future expansion
4	PS	IP.4	IP^4	BC	Serial interrupt priority
3	PT1	IP.3	IP^3	BB	Timer 1 interrupt priority
2	PX1	IP.2	IP^2	BA	External 1 interrupt priority
1	PT0	IP.1	IP^1	B9	Timer 0 interrupt priority
0	PX0	IP.0	IP^0	B8	External 0 interrupt priority

Table 2-17 IP (B8H) SFR

When considering interrupt priorities, the following rules apply:

- Nothing can interrupt a high priority interrupt; not even another high priority interrupt. Same priority interrupts cannot interrupt each other.
- A high priority interrupt may interrupt a low priority interrupt.
- A low priority interrupt may be dealt with only 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 these interrupts happen to have the same priority, the interrupt which is serviced first is determined by polling sequence order of Table 2-16.
- An new interrupt cannot pause an already running ISR which was triggered by an interrupt having the same priority as the new one, irrespective of the polling sequence order mentioned in Table 2-16.

2.12.5 What Happens When an Interrupt Occurs?

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

1. The current Program Counter (address of the next code/instruction to be executed) is saved (pushed) on the stack, low byte first.
2. Interrupts of the same and lower priority are blocked.

3. In the case of Timer and External interrupts, the corresponding interrupt flag is cleared automatically. Take special note of this third step: If the interrupt being handled is a Timer 0 or Timer 1 or External interrupt, the microcontroller automatically clears the interrupt flag before passing control to your interrupt handler routine. This is because there is no ambiguity as to what caused the interrupt, and it is not necessary to clear the bit in the ISR code. If the external interrupt was programmed as being level triggered, then the hardware has to ensure that the level is again restored so as no to cause repeated interrupts. *However we would have to clear the particular flag in the ISR software for the Serial port and Timer 2 (in the case of the 8032 controller).* This is due to the fact that for each of these devices, two different events can trigger the same interrupt number. For the serial port it can be a Received character interrupt (RI) or a Transmitted character interrupt (TI), or both, as explained in the section below. For the Timer 2, it could be the timer overflow flag (TF2) or EXT2 flag which caused the interrupt.
4. Program execution transfers to the corresponding interrupt handler vector address.
5. The Interrupt Service Routine executes.
6. At the end of the ISR, the Program Counter is popped back automatically from the stack once the RETI instruction is executed.

Additionally, apart from these automatic events, other precautions may need to be taken. It is good programming practice to save (push) the PSW register immediately at the beginning of the ISR so that we save the status of the various flags which might have been in use by the interrupted section of the code. The interrupt jump might have occurred just before our main program was about to execute a JC label (jump if carry bit is set). If the ISR routine modifies the carry bit, then when the ISR is finished and the main program resumes operation, it would not perform as expected. The PSW should then be popped back before executing the RETI instruction.

If we are absolutely certain that our ISR does not modify any flags, then there is no need to PUSH/POP the PSW register.

```
ISR_EXAMPLE_01:
            PUSH PSW            ;save flags
            ……………
            ……………
            POP PSW             ;restore flags
            RETI
```

If in our ISR we intend to overwrite and use some registers which are being used in our main program or in some other ISR, we would also need to be very careful.

We can push all these registers at the start of your ISR and pop them back at the end, before executing the RETI instruction.

```
ISR_EXAMPLE_02:
        PUSH PSW        ;save flags
        PUSH ACC        ;save registers being used in this routine
        PUSH B          ;with their original values
        PUSH 0          ;save register r0 bank 0 (address 0h)
        PUSH 1          ;save register r1 bank 0 (address 1h)
        …………….
        …………….
        POP 1           ;restore registers to original values
        POP 0
        POP B
        POP ACC
        POP PSW         ; restore flags
        RETI
```

Note that the registers should be popped out of the stack in the reverse order from the way they were pushed. The first register that was pushed on the stack, should be the last register that is popped from the stack.

Instead of pushing and popping registers R0-R7 in the ISR, we might consider using a separate dedicated bank for the ISR routine. The PSW (and ACC, B, DPL and DPH if used) should still be pushed/popped. This could be done by setting the corresponding bits in the PSW register as shown:

```
ISR_EXAMPLE_03:
        PUSH PSW    ;save flags and register bank in use flags
        SETB RS0
        CLR RS1     ; use register bank 1
        …………….
        …………….
        POP PSW     ; restore flags and original register bank
        RETI
```

The POP PSW instruction automatically restore the original register bank since RS0 and RS1 bits are actually part of the PSW SFR.

2.12.6 What Happens When an Interrupt Ends?

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

- The high and low order bytes of the program counter are popped back from the stack. These contain the address of the instruction to be executed next.
- The interrupt logic is restored so as to accept additional interrupts at the same priority level as the one just processed.

Using RET instead of RETI at the end of the ISR would ultimately cause the programme not to run as expected since the controller would not handle other interrupts correctly.

2.12.7 Serial Interrupts

Serial Interrupts are slightly different from 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 (or even both), a serial interrupt is triggered. As we 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 our serial interrupt is executed, it may have been triggered because the RI flag was set or because the TI flag was set or perhaps because both flags were set. It should be remembered that both transmission and reception can work simultaneously in the 8051. Thus, our routine must check the status of these flags to determine what action is appropriate, and hence the 8051 cannot and does not automatically clear the RI and TI flags. We must therefore make sure to clear these bits in the ISR.

A brief code example of such a serial ISR is in order. Note that the serial interrupt might have occurred because of a received character and/or a character has just been transmitted:

```
INT_SERIAL:
;First we check whether a character has just been transmitted
          JNB TI,CHECK_RX                    ; If the TI flag is not set, we
                                             ; jump to check RI

; transmitter section
          CLR TI                             ; Clear the TI flag
          JNB TX_BUFFER_FULL, CHK_RX         ; Check RI if nothing else to transmit
          MOV SBUF,TX_BUF                    ; Transmit character stored
                                             ; in location TX_BUF

          CLR TX_BUFFER_FULL                 ; Buffer now empty, ready for
                                             ; the next character
; We still need to check the receiver since BOTH TI and RI might have occurred.
; Hence once we are finished with the TI case, we fall through to the RI case.
;
; receiver section
CHK_RX:
          JNB RI,EXIT_ISR                    ; Ignore if RI is not set
          CLR RI                             ; Clear the RI flag
          JNB RX_BUFFER_FULL,RTR             ; Check if ok to store received character
          SJMP EXIT_ISR                      ; If not, then exit service routine, without saving
                                             ; it, thus losing the character.

RTR:
          MOV RX_BUF,SBUF                    ; Store character in buffer RX_BUF
          SETB RX_BUFFER_FULL                ; Indicate new character in buffer
EXIT_ISR:
          RETI
```

The main program would regularly check variable RX_BUFFER_FULL and gets the character from RX_BUF when available. It would then clear RX_BUFFER_FULL.

As we 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 we forget to clear the interrupt bits, the serial interrupt will be executed over and over again until we clear the bit. Thus it is very important that we always clear the interrupt flags in a serial interrupt.

The above code is an example of a simple serial routine. Other more complete routines can be found in the appendix.

2.12.8 Important Interrupt Considerations:

We now list some important considerations to be made when using interrupts and writing interrupt service routines.

2.12.8.1 Register Protection

One very important rule applies to all interrupt handlers:

Interrupts must leave the processor in the same state that it was in when the interrupt was initiated.

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

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

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

But what would happen if right after the MOV instruction an interrupt has occurred. Let us assume that during this interrupt service routine, the carry bit was set and the value of the accumulator was changed to 40h. When the interrupt finished and control is passed back to the main program, the ADDC would add 10h to 40h, and add an additional 01h because the carry bit is set. In this case, the accumulator will contain the value 51h at the end of execution.

The program has calculated the wrong answer. A programmer who is 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 that the interrupt did not protect the registers it was using.

What does this mean? It means that if our interrupt uses the accumulator, it must ensure 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 main code of the ISR 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 can 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 that it has protected or pushed on stack. 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 was executed.

In general, our interrupt routine must protect the following registers if somehow they are being modified in the ISR:

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

Remember that PSW consists of many individual bits that are set by various 8051 instructions. Unless we are absolutely sure of what we are doing and have a complete understanding of which instructions set which 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 our interrupts. The PSW register would preserve the status of the register bank we were using prior to the interrupt, the carry flag, the zero flag etc.

Note also that most assemblers (in fact, ALL assemblers that I know of) will not allow us 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. Hence R0 is not a valid memory address that the PUSH and POP instructions can use.

Thus, if we are using any "R" register in our interrupt routine, we will have to push that register's absolute address onto the stack instead of just saying PUSH R0. For example, when using bank 0 instead of PUSH R0 we would execute:

```
PUSH 00h
```

Of course, this only works if we have selected the default register bank 0. If we are using an alternate register set, we must PUSH the address which corresponds to the register we are using in that alternate bank. For example, if we are using register bank 1, then the register R0 for that bank would have an address of 08h, hence we would use:

```
PUSH 08h
```

Certain assemblers allow special keywords (such as PUSH AR2) to be used in order to calculate automatically the correct address for the register being pushed of popped. Such as:

```
; if we intend using register 2 of bank 1 in our ISR
PUSH    PSW     ; save PSW
SETB    RS0     ; select bank 1 (RS0=1, RS1=0)
CLR     RS1
USING   1       ; advise pre-processor to use register bank 1 (no code)
PUSH    AR2     ; push R2 in bank 1 (address 0Ah)

; if we intend using register 7 of bank 3 in our ISR
PUSH    PSW     ; save PSW
SETB    RS0     ; select bank 3 (RS0 = RS1 = 1)
SETB    RS1
USING   3       ; advise pre-processor to use register bank 3 (no code)
PUSH    AR7     ; push R7 in bank 3 (address 1Fh)
```

Note that the keyword USING does not generate any code. It is used by the pre-processor to calculate the correct address for the register being pushed or popped.

Alternatively, we might want to make use of a separate register bank for our ISR, a register bank which is used only in the ISR. In this case, provided that we do not use any other registers which are used elsewhere, there would not be the need to push any of the registers R0-R7. We push only the PSW and ACC (and perhaps B, DPH and DPL if we use them in the ISR, since they would be common with other sections of the code) and then set RS0, RS1 (two bits themselves residing in the PSW register) to select our reserved bank. Then, before leaving, we simply pop back the pushed registers in reverse order. There would not be the need to reset again RS0 and RS1 separately, since their original value would be re-instated anyway when we pop back the PSW register, where the original RS0 and RS1 bit settings were stored.

2.12.9 Common Problems with Interrupts

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

If we are using interrupts and our program is crashing or does not seem to be performing as we would expect, we should always review the interrupt-related issues. See section 11.7 for some hints on using interrupts.