## SECTION 11.1: 8051 INTERRUPTS

In this section, first we examine the difference between polling and interrupts and then describe the various interrupts of the 8051.

### Interrupts vs. polling

A single microcontroller can serve several devices. There are two ways to do that: interrupts or polling. In the *interrupt* method, whenever any device needs its service, the device notifies the microcontroller by sending it an interrupt signal. Upon receiving an interrupt signal, the microcontroller interrupts whatever it is doing and serves the device. The program associated with the interrupt is called the *interrupt service routine* (ISR) or *interrupt handler*. In *polling*, the microcontroller continuously monitors the status of a given device; when the status condition is met, it performs the service. After that, it moves on to monitor the next device until each one is serviced. Although polling can monitor the status of several devices and serve each of them as certain conditions are met, it is not an efficient use of the microcontroller. The advantage of interrupts is that the microcontroller can serve many devices (not all at the same time, of course); each device can get the attention of the microcontroller based on the priority assigned to it. The polling method cannot assign priority since it checks all devices in a round-robin fashion. More importantly, in the interrupt method the microcontroller can also ignore (mask) a device request for service. This is again not possible with the polling method. The most important reason that the interrupt method is preferable is that the polling method wastes much of the microcontroller's time by polling devices that do not need service. So in order to avoid tying down the microcontroller, interrupts are used. For example, in discussing timers in Chapter 9 we used the instruction "`JNB TF,target`", and waited until the timer rolled over, and while we were waiting we could not do anything else. That is a waste of the microcontroller's time that could have been used to perform some useful tasks. In the case of the timer, if we use the interrupt method, the microcontroller can go about doing other tasks, and when the TF flag is raised the timer will interrupt the microcontroller in whatever it is doing.

### Interrupt service routine

For every interrupt, there must be an interrupt service routine (ISR), or interrupt handler. When an interrupt is invoked, the microcontroller runs the interrupt service routine. For every interrupt, there is a fixed location in memory that holds the address of its ISR. The group of memory locations set aside to hold the addresses of ISRs is called the interrupt vector table, shown in Table 11-1.

## Steps in executing an interrupt

Upon activation of an interrupt, the microcontroller goes through the following steps.

1. It finishes the instruction it is executing and saves the address of the next instruction (PC) on the stack.
2. It also saves the current status of all the interrupts internally (i.e., not on the stack).
3. It jumps to a fixed location in memory called the interrupt vector table that holds the address of the interrupt service routine.
4. The microcontroller gets the address of the ISR from the interrupt vector table and jumps to it. It starts to execute the interrupt service subroutine until it reaches the last instruction of the subroutine, which is RETI (return from interrupt).
5. Upon executing the RETI instruction, the microcontroller returns to the place where it was interrupted. First, it gets the program counter (PC) address from the stack by popping the top two bytes of the stack into the PC. Then it starts to execute from that address.

Notice from Step 5 the critical role of the stack. For this reason, we must be careful in manipulating the stack contents in the ISR. Specifically, in the ISR, just as in any CALL subroutine, the number of pushes and pops must be equal.

## Six interrupts in the 8051

In reality, only five interrupts are available to the user in the 8051, but many manufacturers' data sheets state that there are six interrupts since they include reset. The six interrupts in the 8051 are allocated as follows.

1. Reset. When the reset pin is activated, the 8051 jumps to address location 0000. This is the power-up reset discussed in Chapter 4.
2. Two interrupts are set aside for the timers: one for Timer 0 and one for Timer 1. Memory locations 000BH and 001BH in the interrupt vector table belong to Timer 0 and Timer 1, respectively.
3. Two interrupts are set aside for hardware external hardware interrupts. Pin numbers 12 (P3.2) and 13 (P3.3) in port 3 are for the external hardware interrupts INT0 and INT1, respectively. These external interrupts are also referred to as EX1 and EX2. Memory locations 0003H and 0013H in the interrupt vector table are assigned to INT0 and INT1, respectively.
4. Serial communication has a single interrupt that belongs to both receive and transmit. The interrupt vector table location 0023H belongs to this interrupt.

Notice in Table 11-1 that a limited number of bytes is set aside for each interrupt. For example, a total of 8 bytes from location 0003 to 0000A is set aside for INT0, external hardware interrupt 0. Similarly, a total of 8 bytes from location 000BH to 0012H is reserved for TF0, Timer 0 interrupt. If the service routine for a given interrupt is short enough to fit in the memory space allocated to it, it is placed in the vector table; otherwise, an LJMP instruction is placed in the vector table to point to the address of the ISR. In that case, the rest of the bytes allocated to that interrupt are unused. In the next three sections we will see many examples of interrupt programming that clarify these concepts.

From Table 11-1, also notice that only three bytes of ROM space are assigned to the reset pin. They are ROM address locations 0, 1, and 2. Address location 3 belongs to external hardware interrupt 0. For this reason, in our program we put the LJMP as the first instruction and redirect the processor away from the interrupt vector table, as shown in Figure 11-1. In the next section we will see how this works in the context of some examples.

**Table 11-1: Interrupt Vector Table for the 8051**

| Interrupt | ROM Location (Hex) | Pin | Flag Clearing |
|---|---|---|---|
| Reset | 0000 | 9 | Auto |
| External hardware interrupt 0 (INT0) | 0003 | P3.2 (12) | Auto |
| Timer 0 interrupt (TF0) | 000B | | Auto |
| External hardware interrupt 1 (INT1) | 0013 | P3.3 (13) | Auto |
| Timer 1 interrupt (TF1) | 001B | | Auto |
| Serial COM interrupt (RI and TI) | 0023 | | Programmer clears it. |

```
          ORG  0     ;wake-up ROM reset location
          LJMP MAIN  ;bypass interrupt vector table

;---- the wake-up program
          ORG  30H
MAIN:
          ....
          END
```

**Figure 11-1. Redirecting the 8051 from the Interrupt Vector Table at Power-up**

## Enabling and disabling an interrupt

Upon reset, all interrupts are disabled (masked), meaning that none will be responded to by the microcontroller if they are activated. The interrupts must be enabled by software in order for the microcontroller to respond to them. There is a register called IE (interrupt enable) that is responsible for enabling (unmasking) and disabling (masking) the interrupts. Figure 11-2 shows the IE register. Note that IE is a bit-addressable register.

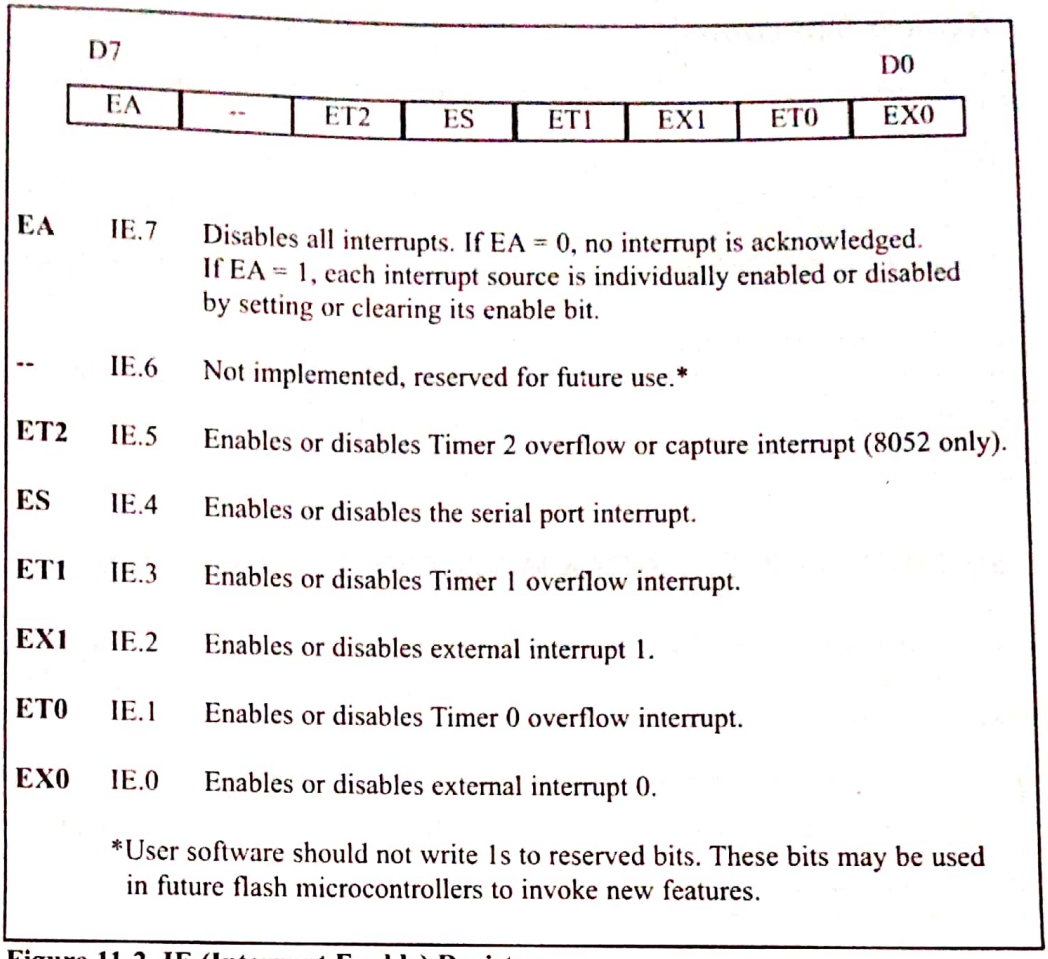
From Figure 11-2 notice that bit D7 in the IE register is called EA (enable all). This must be set to 1 in order for the rest of the register to take effect. D6 is unused. D5 is used by the 8052. The D4 bit is for the serial interrupt, and so on.

### Steps in enabling an interrupt

To enable an interrupt, we take the following steps:

1. Bit D7 of the IE register (EA) must be set to high to allow the rest of register to take effect.
2. If EA = 1, interrupts are enabled and will be responded to if their corresponding bits in IE are high. If EA = 0, no interrupt will be responded to, even if the associated bit in the IE register is high.

To understand this important point look at Example 11-1.




| | | |
|---|---|---|
| **EA** | IE.7 | Disables all interrupts. If EA = 0, no interrupt is acknowledged. If EA = 1, each interrupt source is individually enabled or disabled by setting or clearing its enable bit. |
| -- | IE.6 | Not implemented, reserved for future use.* |
| **ET2** | IE.5 | Enables or disables Timer 2 overflow or capture interrupt (8052 only). |
| **ES** | IE.4 | Enables or disables the serial port interrupt. |
| **ET1** | IE.3 | Enables or disables Timer 1 overflow interrupt. |
| **EX1** | IE.2 | Enables or disables external interrupt 1. |
| **ET0** | IE.1 | Enables or disables Timer 0 overflow interrupt. |
| **EX0** | IE.0 | Enables or disables external interrupt 0. |

*User software should not write 1s to reserved bits. These bits may be used in future flash microcontrollers to invoke new features.

**Figure 11-2. IE (Interrupt Enable) Register**

**Example 11-1**

Show the instructions to (a) enable the serial interrupt, Timer 0 interrupt, and external hardware interrupt 1 (EX1), and (b) disable (mask) the Timer 0 interrupt, then (c) show how to disable all the interrupts with a single instruction.

**Solution:**

```
(a)   MOV IE,#10010110B   ;enable serial, Timer 0, EX1
```

Since IE is a bit-addressable register, we can use the following instructions to access individual bits of the register.

```
(b)   CLR IE.1        ;mask(disable) Timer 0 interrupt only
(c)   CLR IE.7        ;disable all interrupts
```

Another way to perform the "MOV IE,#10010110B" instruction is by using single-bit instructions as shown below.

```
SETB IE.7       ;EA=1, Global enable
SETB IE.4       ;enable serial interrupt
SETB IE.1       ;enable Timer 0 interrupt
SETB IE.2       ;enable EX1
```