8051 Interrupts

Dinesh Sharma

Department Of Electrical Engineering
Indian Institute Of Technology, Bombay

January 28, 2021

Interrupts

- When a program is running on a processor, events in the outside environment may require interrupting this program to take some urgent action.
- For example, we may need to read an A to D converter at precise intervals while another program is running.
- When it is time to take a reading, we need to stop the program, run a pre-written function which will read the ADC, and then resume the program which was running earlier.
- Requirements of this kind cannot be handled through software alone or through hardware alone.
- The architecture of the processor has to make it possible to handle this requirement.

Interrupt mechanism

The mechanism to handle interrupts involves the following:

- A facility for the outside world to signal that an event which needs to be handled has occurred.
- This signal can occur at any time – so recognizing its arrival cannot be done in software.
  (Where would you place the code to recognize it in your software?)
- A mechanism to determine if this event should be allowed to interrupt the running program or not.
- To handle the event, a function to take appropriate actions has to written in advance. (These functions are called event handlers).
  The question is, when and how is this function to be executed?
- The interrupt mechanism should be capable of handling multiple events which can cause an interrupt.
- The interrupt mechanism has to decide what to do if multiple events occur at the same time.

Hardware requirements

- The processor should designate the pins through which the outside world will signal that an interrupt causing event has occurred.
- The processor architecture decides what kind of electrical signal should be used when the event occurs: – a HIGH level, a LOW level, a positive edge or a negative edge.
- The outside world is not aware of the processor clock. This signal may have a transition which occurs at the same time as the processor clock.
  This would violate set up and hold time assumptions for the circuit used to latch the signal, leading to meta-stability or other malfunctions.
- Synchronizers need to be used to synchronize this external asynchronous signal to the system clock.

Hardware requirements

- Once the interrupt signal has been received and allowed to interrupt the running program, the instruction pointer should be loaded with the address where the handler function begins. How does the hardware know this address?
- What happens if another interrupt signal arrives while this function is executing?
- When the handler function ends, what actions need to be taken before we resume the originally running program?
- How do we determine where the main program was interrupted and needs to be re-started from?

Software requirements

- A program may contain segments where it may not perform correctly if it is interrupted.
- Such segments are called critical code.
- The program should disable interrupts while entering such segments and enable these again as soon as it exits the critical code segment.

Software requirements

- Since the call and execution of the handler program can take place at any time, the handler function should not alter any flag or register which is used by the main program.
- The start address of the handler function may have to be loaded at locations designated by the processor architecture, so that it can be looked up when the handler function is called. (These are called interrupt vectors).
- The time between signaling an event and starting of its handler is called interrupt latency. The hardware and software associated with the interrupt mechanism should make the interrupt latency as short as possible.

The task of saving registers can be simplified by bank switching.

Bank switching

- The interrupt handler (also known as interrupt service routine) must not alter any data or flag which is used by the main program.
- It is normal practice for all interrupt service routines to push all the registers it is going to use and flags on the stack. These are popped off the stack when the service routing returns.
- This task can be made easier and faster by bank switching.
- In 8051, the bank of registers R0-R7 can refer to different physical registers depending on two bank select bits in the program status word.
- For example, if these two bits are ‘00’, locations 00 through 07 represent these registers. However, if we change the bits to ‘01’, R0 through R7 will refer to locations ‘08’ through ‘0F’.
- We can permanently allot some bank(s) to the main program and some to the interrupt routine.

Bank switching

- If we permanently allocate some memory bank(s) to the main program and a different bank to the interrupt routine, we do not have to push the main program registers at the start of the interrupt service routinte.
- We can save PSW on stack and then change the bank select bits to point to the bank reserved for the service routine.
- References to R0-R7 will not change the registers being used by the main program.
- At the end, we just pop PSW. This will restore the bank of registers which was being used by the main program, in addition to restoring flags.

Interrupt Sources in 8051 architecture

The 8051 architecture can handle interrupts from 5 sources. These are:

- From two external interrupt lines,
- From two built in timers and
- From the serial interface.

8052 and all the recent version chips of the 8051 family add a third timer and many other peripheral functions. These may add to the interrupt sources defined for the basic 8051.

Each interrupt source is assigned an interrupt vector address.

External Interrupts

Port P3 of 8051 is a multi-function port. Different lines of this port carry out functions which are additional to data input-output on the port.

Additional functions of Port 3 lines

Port Line	P3.7	P3.6	P3.5	P3.4	P3.3	P3.2	P3.1	P3.0
Function	$\overline{RD}$	$\overline{WR}$	T1 in	T0 in	$\overline{INT1}$	$\overline{INT0}$	TxD	RxD

Lines P3.2 and P3.3 can be used as interrupt inputs.
Interrupts will be caused by a 'LOW' level, or a negative edge on these lines.

Special Function Registers for interrupts

Half of the special function register TCON is used for setting the conditions for causing interrupts from external sources.

This register is bit addressable.

SFR TCON at byte address 88H

Bit No.	7	6	5	4	3	2	1	0
Bit Name	TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0
Bit Addr	8F	8E	8D	8C	8B	8A	89	88

Bits 0 and 2 set the interrupt type (level or edge). Bits 1 and 3 indicate the current status of the external interrupts.

Special Function Registers for interrupts

SFR TCON at byte address 88H

Bit No.	7	6	5	4	3	2	1	0
Bit Name	TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0
Bit Addr	8F	8E	8D	8C	8B	8A	89	88

- IT1 and IT0 are the “Interrupt Type” flags for external sources 1 and 0 respectively. These decide whether a negative going edge or a ‘LOW’ level will cause an interrupt.
- If the bit is set, the corresponding interrupt is edge sensitive. If it is cleared, the interrupt is level sensitive.
- IE1 and IE0 are the status flags for the two external interrupt lines.
- If the flag is 1, the selected type of event (edge or level) has occurred on the corresponding interrupt line.

Internal Interrupts

- Internally generated interrupts can be either from the timers, or from the serial interface.
- The serial interface causes interrupts due to a receive event (RI) or due to a transmit event (TI).
- The receive event occurs when the input buffer of the serial line (sbuf in) is full and a byte needs to be read from it.
- The transmit event indicates that a byte has been sent a new byte can be written to output buffer of the serial line (sbuf out).

Timer Interrupts

SFR TCON at byte address 88H

Bit No.	7	6	5	4	3	2	1	0
Bit Name	TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0
Bit Addr	8F	8E	8D	8C	8B	8A	89	88

- 8051 timers always count up. When their count rolls over from the maximum count to 0000, they set the corresponding timer flag TF1 or TF0 in TCON.
- Counters run only while their run flag (TR1 or TR0) is set by the user program. When the run flag is cleared, the count stops incrementing.
- The 8051 can be setup so that an interrupt occurs whenever TF1 or TF0 is set.

Enabling Interrupts

At power-up, all interrupts are disabled. Suppose Timer 0 is started. When it times out, TF0 in the special function register TCON will be set. However, this will not cause an interrupt.
To enable interrupts, a number of steps need to be taken.

- There is an interrupt enable special function register IE at byte address A8H. This register is bit addressable. (The assembler gives special mnemonics to each bit address.)

SFR IE at byte address A8H

Bit No.	7	6	5	4	3	2	1	0
Function	IE	U	U	SI	TF1	Ex1	TF0	Ex0
Bit Addr	AF	AE	AD	AC	AB	AA	A9	A8
Bit Name	EA	-	-	ES	ET1	EX1	ET0	EX0

Enabling Interrupts

SFR IE at byte address A8H

Bit No.	7	6	5	4	3	2	1	0
Function	IE	U	U	SI	TF1	Ex1	TF0	Ex0
Bit Addr	AF	AE	AD	AC	AB	AA	A9	A8
Bit Name	EA	-	-	ES	ET1	EX1	ET0	EX0

- The most significant bit of the register is a global interrupt enable flag. This bit must be set in order to enable any interrupt.
- Bits 6 and 5 are undefined for 8051. (Bit 5 is used by 8052 for the third timer available in 8052).
- Bit 4, when set, enables interrupts from the serial port.

Enabling Interrupts

SFR IE at byte address A8H

Bit No.	7	6	5	4	3	2	1	0
Function	IE	U	U	SI	TF1	Ex1	TF0	Ex0
Bit Addr	AF	AE	AD	AC	AB	AA	A9	A8
Bit Name	EA	-	-	ES	ET1	EX1	ET0	EX0

- Bit 3 should be set to enable interrupts from Timer 1 overflow.
- Bit 2 is set to enable interrupts from external interrupt 1 (pin P3.3 on Port 3).
- Bit 1 enables interrupts from Timer 0 when it overflows.
- Bit 0, when set, will enable interrupts from external interrupt 0 (pin P3.2 on Port 3).

Interrupt Vectors

When an interrupt occurs, the updated PC is pushed on the stack and is loaded with the vector address corresponding to the interrupt.

The following table gives the vector addresses.
(The order of entries in the table is also the order in which the 8051 will poll these in case of multiple interrupts).

Interrupt Source	Vector address
External Interrupt 0	0003H
Timer 0 Overflow	000BH
External Interrupt 1	0013H
Timer 1 Overflow	001BH
Serial Interface	0023H

Interrupt Vectors

8051 starts executing from address 0000H at power-up or reset.

Interrupt Source	Vector
External Interrupt 0	0003H
Timer 0 Overflow	000BH
External Interrupt 1	0013H
Timer 1 Overflow	001BH
Serial Interface	0023H

The first 3 bytes of the program memory are typically used for placing a long jump instruction to start of the code area.
The interrupt vectors start from 0003 and are separated by 8 bytes from each other.

Actual handler functions cannot be accommodated within the 8 byte gap. So these functions are loaded at different addresses and Jump instructions to actual handler code are placed at these vector addresses.

Timer Interrupt Example

To enable interrupts from T0, we have to do the following:
SetB EA ; (or SetB IE.7) to enable interrupts
SetB ET0 ; (or SetB IE.1) to enable interrupts from T0
After this, whenever T0 overflows,

- TF0 will be set (in SFR TCON),
- The currently running program will be interrupted, its PC value will be put on the stack (PC-L first, PC-H after – because the stack grows upwards in 8051).
- PC will be loaded with 000B H (Timer 0 overflow vector).
- A jump instruction to the actual handler for T0 should be placed here.
- The handler should end with the instruction:
  RETI

Interrupt Priorities

- 8051 has two levels of interrupt priorities: high or low.
- By assigning priorities, we can control the order in which multiple interrupts will be serviced.
- Priorities are set by bits in a special function register called IP, which is at the byte address B8H. This register is also bit addressable. The assembler defines special names for bits of this register.

SFR IP at byte address B8H

Bit No.	7	6	5	4	3	2	1	0
Bit Addr	BF	BE	BD	BC	BB	BA	B9	B8
Bit Name	U	U	U	PS	PT1	PX1	PT0	PX0

Interrupt Priorities

SFR IP at byte address B8H

Bit No.	7	6	5	4	3	2	1	0
Bit Addr	BF	BE	BD	BC	BB	BA	B9	B8
Bit Name	U	U	U	PS	PT1	PX1	PT0	PX0

- 8051 has two levels of interrupt priorities: high or low.
- Bits are in the “polling order” of interrupts.
- A 1 in a bit position assigns a high priority to the corresponding source of interrupts – a 0 gives it a low priority.

Interrupt Priorities

In case of multiple interrupts, the following rules apply:

- While a low priority interrupt handler is running, if a high priority interrupt arrives, the handler will be interrupted and the high priority handler will run. When the high priority handler does 'RETI', the low priority handler will resume. When this handler does 'RETI', control is passed back to the main program.
- If a high priority interrupt is running, it cannot be interrupted by any other source – even if it is a high priority interrupt which is higher in polling order.

Interrupt Priorities

- A low-priority interrupt handler will be invoked only if no other interrupt is already executing. Again, the low priority interrupt cannot preempt another low priority interrupt, even if the later one is higher in polling order.
- If two interrupts occur at the same time, the interrupt with higher priority will execute first. If both interrupts are of the same priority, the interrupt which is higher in polling sequence will be executed first. This is the only context in which the polling sequence matters.

Serial Interrupts

Serial interrupts are handled somewhat differently from the timers.

- There are independent interrupt flags for reception and transmission of serial data, called RI and TI.
- RI indicates that a byte has been received and is available for reading in the input buffer.
- TI indicates that the previous byte has been sent serially and a new byte can be written to the serial port.
- A serial interrupt occurs if *either* of these flags is set. (Of course the serial interrupt must be enabled for this to occur).
- The interrupt service routine should check which of these events caused the interrupt. This can be done by examining the flags.

Serial Interrupts

The serial interrupt service routine needs to check which event (receive or transmit) caused the interrupt. This can be done by examining the flags.

- Either or both of the flags RI and TI might be set, requiring a read from or write to the serial buffer sbuf (or both).
- The input and output serial buffers are distinct but are located at the same address.
- A read from this address reads the input buffer while a write to the same address writes to the output buffer.
- The RI and TI flags are *not* automatically cleared when an interrupt is serviced. Therefore, the interrupt service routine must clear them before returning.

Serial Interrupts

Here is an example handler for serial interrupts:

```
ISRSerial:
            PUSH    PSW         ; Save flags and context
            PUSH    ACC         ; and accumulator
            JNB     RI,output   ; If RI not set, check for TI
            MOV     A, SBUF     ; Get the received character
            MOV     inchar, A   ; Save this character
            CLR     RI          ; clear receive interrupt flag
output:                         ; Check if output is required
            JNB     TI, done    ; If no transmit interrupt, leave
            MOV     A, outchar  ; Else get the char to send
            MOV     sbuf, A     ; Write to serial buffer
            CLR     TI          ; Clear Transmit interrupt flag
done:       POP     ACC         ; Restore Accumulator
            POP     PSW         ; and flags
            RETI                ; and return
```

Sequence of Events after an interrupt

When an enabled interrupt occurs,

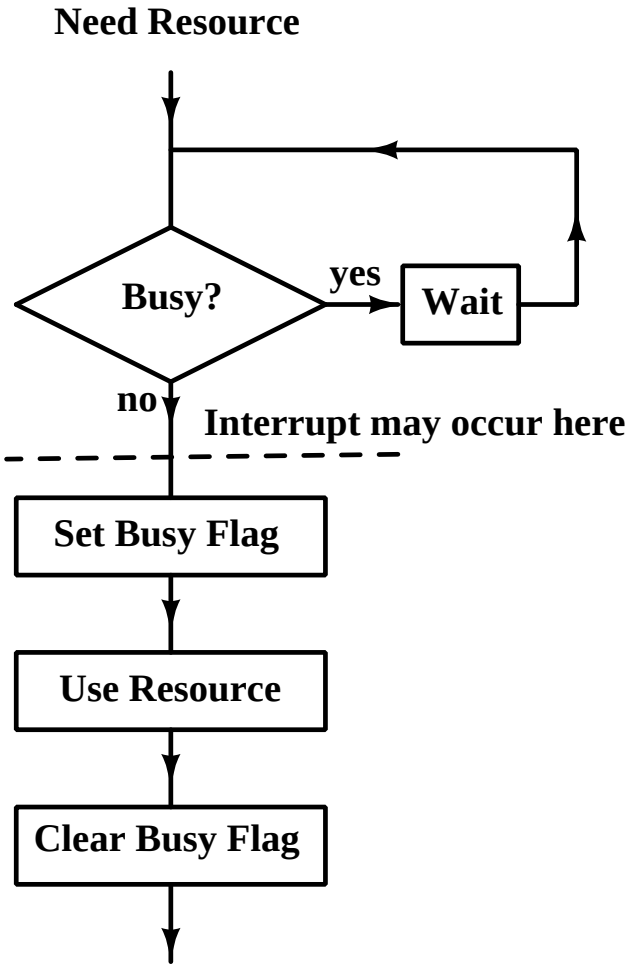
1. The PC is saved on the stack, low byte first. This is because the stack grows upwards in 8051.
2. Other interrupts of lower priority and same priority are disabled.
3. Except for the serial interrupt, the corresponding interrupt flag is cleared.
4. PC is loaded with the vector address corresponding to the interrupt.

When the handler executes ‘RETI"

1. PC is restored by popping the stack.
2. Interrupt status is restored to its original value. (Same and lower priority interrupts restored to original status).

Critical code for a shared resource

Suppose there is a resource, say a memory buffer which is used by the main program, as well as an interrupt handler.

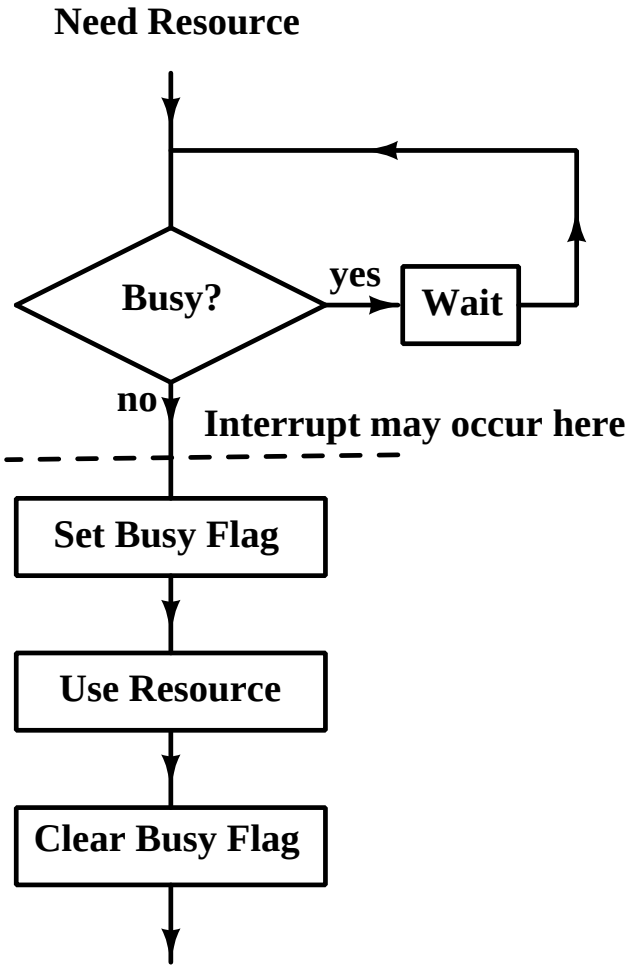


- The shared resource has a busy/available flag associated with it.
- Functions which need this resource use the procedure shown in the figure on the left.

Normally this procedure works fine, since a function uses a resource only when it is not busy – otherwise it waits for it to be available.

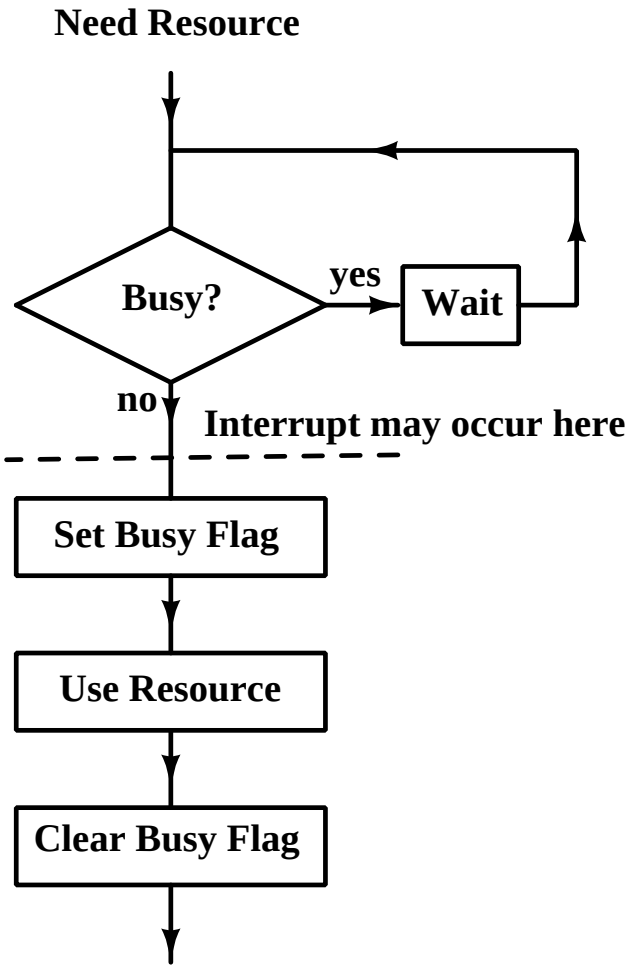
Critical code for a shared resource

Now consider the following sequence of events:



- Main program needs to use the resource, checks the flag and finds that it is not busy.
- After the check, main program would have set the busy flag and proceeded to use the resource.
- However, before setting the flag, an interrupt occurs.
- Flag has been checked but not set by the main program when the handler starts running.

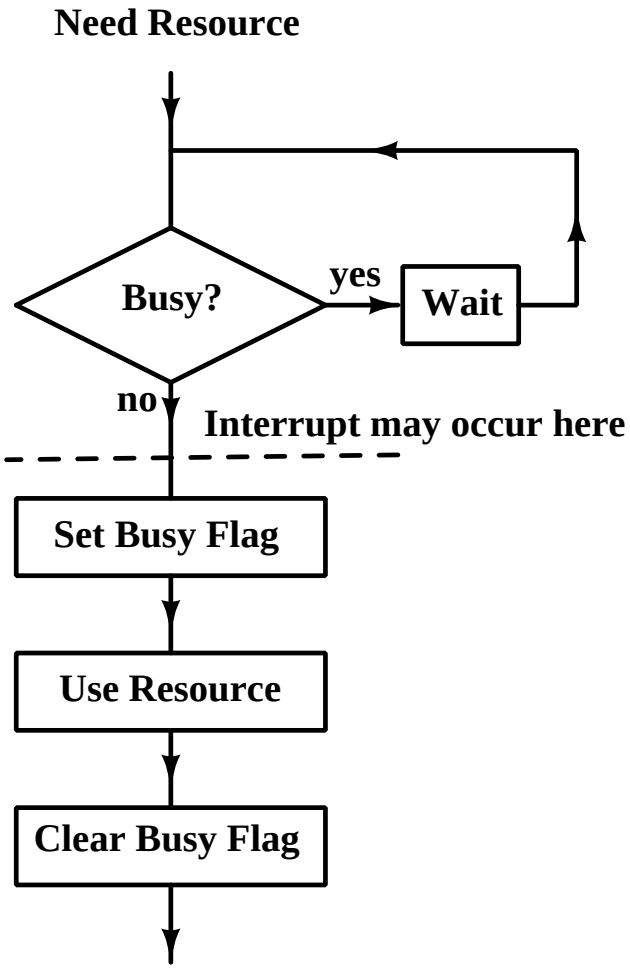
Critical code for a shared resource



The handler also needs the resource.

- It checks the flag and finds that it is not set to busy. It sets the flag as busy and starts using the resource.
- Now when the main program runs, it starts running from the point where it was interrupted (after checking, before setting):
- It now sets the flag to busy (unaware that it is already set!) and starts writing to the buffer, destroying the work done by the handler!

Critical code for a shared resource



- Clearly, for this code to run correctly, there should be no interruption between checking and setting the flag.
- Therefore the code sequence starting at flag checking and up to the point of setting it, is “critical”.
- Ideally the operation of checking the flag and setting it should be “atomic”.

Instructions for avoiding critical code

- The instruction JBC is useful for such a scenario.
- We use flag = 0 to indicate that the resource is busy and flag = 1 to show it is available.
- Now we use the instruction JBC with the flag as its argument and the function for using the resource as its destination.
- If the resource is available, the flag will be found to be set and the jump will occur. Simultaneously, the flag will be cleared (marked busy).
- Since checking the flag and clearing it occurs in a single instruction, an interrupt cannot intervene between the two actions.

Return to Interrupt Mechanism