ENGR 323L: Microprocessor Systems
Engineering Department, Trinity College
Instructor: Professor T. Ning

Interrupt Service Routines (ISRs)

Interrupt service routines are perhaps the most important of all building blocks. An *interrupt* is an event that interrupts normal program execution. A good understanding of interrupts and interrupt service routines is essential for writing real-time applications software.

When a subroutine is called, the return address is stored on stack. Upon a return instruction, the program flow continues from where the subroutine was called. The exact time of such a call is determined by the calling program.

ISR is a special type of subroutine. The calling program does not explicitly call the ISR (does not control when exactly). ISR is invoked as a response to an external signal.

Programs can be divided into two classes:
- foreground tasks are under program control, i.e., the main program and all the subroutines explicitly called from the main program
- background tasks are not called by the main program but are invoked by signals

The 8051 family of processors has interrupts that can be invoked by external signals or internal signals (conditions), e.g., the timer overflows may be programmed to issue an interrupt. The 8051 may be configured so that when any of these events occur the main program is temporarily suspended and control passed to ISR. Control will be returned after the target ISR is complete. The ability to interrupt normal program execution makes it easier and more efficient to handle certain conditions of interests.

Interrupt structure of the basic 8051 Family
- two external interrupts (IE0 & IE1)
- two timer overflow interrupts (TF0 & TF1)
- serial port interrupt

each interrupt has a unique starting address given below

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

Note: Most 8051 derivatives developed by different semiconductor manufacturers have more than 5 interrupts and 2 timers. The numbers of available timers and interrupts are proportional to the flexibilities of microcontrollers.

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:

- Two bytes are popped off the stack into the Program Counter to restore normal program execution.

Interrupt status is restored to its pre-interrupt status

SFRs of Interrupts: Each interrupt can be individually enabled or disabled through the interrupt enable register **(IE)** register. Each interrupt source can be individually programmed to one of two priorities through the interrupt priority register **(IP)** register.

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. Your program must specifically configure the 8051 to enable interrupts.

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

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

EX0 (IE.0) – external interrupt 0 enable bit
ET0 (IE.1) – internal timer_0 overflow enable bit
EX1 (IE.2) – external interrupt 1 enable bit
ET1 (IE.3) – internal timer_1 overflow enable bit
ES (IE.4) – serial transmit/receive enable bit
EA (IE.7) – master enable bit (set/clear to enable/disable all interrupts)

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

MOV IE, #08h
or
SETB ET1

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

IP - Interrupt Priority register:
PX0 (IP.0), PT0 (IP.1), PX1 (IP.2), PT1 (IP.3), PS (IP.4)

If the bit(s) set, the corresponding interrupts will be given high priorities. If two interrupts of the same priority-level occur at the same time, an internal polling sequence will determine which interrupt is served first (IE0 > TF0 > IE1 > TF1 > SERIAL)

In operation, all interrupt flags are latched into the interrupt control system and polled during the next machine cycle. If the flag of an enabled interrupt is set, LCALL will be automatically issued to the appropriate location (see the interrupt vector below), which causes the contents of the program counter **(PC)** to be pushed into the stack. No other registers will be pushed to the stack. It can improve the response time of interrupts.

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

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

SETB ET1
SETB EA

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

```
e.g.,   org     0
        ljmp    start
        org      03              ;external interrupt 0
        ljmp     ISR0

        org     100h
start:
        setb     IT0             ;transition sensitive
        setb     EX0             ;enable external interrupt 0
        setb     EAL             ;master interrupt enable

ISR0:   .....
        reti
```

Using Multiple Interrupts

The goal of a microcontroller's interrupt structure is to allow **real-time** events to control the flow of program execution. To do this, two capabilities of this structure are important:

1. CPU could, at any time, disable interrupts and then re-enable interrupts.
2. User should be able to enable and disable specific applications.

Major design considerations for multiple interrupts:

- Providing fast service of interrupts
- Minimizing any delay (latency) that an ISR needs to wait

<Example> To initiate and complete the task of reading a 20-byte message from another device using an on-chip UART at 9600 baud will require 20 slices of CPU time, i.e., extending over about 20 msec and at each millisecond, the UART will interrupt the CPU's execution of the mainline program.

Interrupt Density and Interval Constraints

There are few design guides to handle multiple interrupts and each case must be examined in an ad-hoc way with the chosen microprocessor and the underlying task. The following two constraints show a design case of multiple interrupts and the assumption that when an interrupt is served it will not be interrupted by others (even higher priority interrupts).

Let's consider N ***periodic interrupts*** must be served and the i^{th} interrupt will be repeated every T_{pi} (sec) and it needs T_i (sec) to complete the corresponding ISR. Lower numbered interrupts have higher priority (i.e., 1 > 2 > 3…> N). To simplify the design concern, we can assume that the ISR currently being served cannot be further interrupted by others, i.e., all interrupts demanding service must wait and go by the given priority until the current interrupt is complete.

CPU Utilization Constraint:

The ratio T_i/T_{pi} represents the percentage of CPU utilization for the i^{th} interrupt. The total CPU utilization must be less than 100% (a robust design will have much less CPU utilization),

$$\sum_{i=1}^{N} \frac{T_i}{T_{pi}} < 1.0 \qquad (1)$$

The inequality constraint in (1) only ensures that the CPU is capable of supporting these N interrupts but does not guarantee that all interrupts will be properly within the desired period (T_{pi}). Successful interrupt handling requires additional constraints.

Interrupt Latency Constraint:

To ensure that each interrupt service can be successfully served within the required timing (i.e., the CPU will grant T_i (sec) for the i^{th} interrupt every T_{pi} (sec) it needs, the following inequality condition must be satisfied for each interrupt i. To check for satisfaction, the designer must check from the highest priority interrupt, then the second highest, and down to until the lowest.

$$T_{i+} + \sum_{k=1}^{i-1} N_k T_k < (T_{pi} - T_i) \qquad (2)$$

$N_1 = \text{INT}((T_{pi} — T_i)/T_{p1}) + 1$
$N_2 = \text{INT}((T_{pi} - T_i)/T_{p2}) + 1$

- ***T_{pi} - T_i*:** the allowed latency that the i^{th} interrupt can wait and still completes the routine task before being invoked again.
- ***N_k***: the maximum number of the k^{th} interrupt can invoke during (T_{pi} - T_i). Note that the k^{th} interrupt service routine requires T_{pk} (sec) and the minimum number of N_k is 1.
- ***T_{i+}*** : The largest of (T_{i+1}, T_{i+2}, … T_N, ***CR***, ***INST***)

Critical Region *(CR)* – The CPU is executing some important task that cannot be interrupted until it is completed. Sometimes, programmers create CR to protect certain timing critical tasks. For example, in a precise timing control problem, a CR region is created by the user when Timer_0 is reloaded with the value for the next scheduled ISR. It will be a chaos if the ISRTFO being interrupted between mov THO … and mov TLO …

```
ISRTF0:
        ... protect the critical region and reinitialize Timer_0
        …reload-value = FFFF - target_count + offset
        clr     EA
        mov     TH0,    #3Ch
        mov     TL0,    #0AFh ; count from 15535 to 65535 for 50 msec
        seb     EA
        reti    ;can have more than one return
```

INST – the longest instruction of the target microprocessor. The CPU cannot do anything else until the instruction is done. MUL (multiply) and DIV (division) probably are the longest instructions in 8051.

Use the two inequality constraints to examine the following design examples.

<*Practice Example-1*> A microcontroller is targeted for an application with three periodic interrupt service routines (ISRs), where ISR-1has the highest priority while ISR-3 has the lowest priority. The period between interrupts for any of these sources is 100 us. That is,

$$T_{p1} = T_{p2} = T_{p3} = 100\ \mu s$$

Each ISR needs 30μs CPU time to complete its task, i.e.,

$$T_1 = T_2 = T_3 = 30\ \mu s$$

The critical region CR is 15 μs. The longest instruction used in the program codes requires 5 μs.

$$CR = 15 \text{ and } INST = 5\mu s$$

Finally, assume that each interrupt source must be completed before it being invoked again and other interrupts are left waiting (disabled) while the CPU executing the current ISR.

(a) Is the interrupt density inequality satisfied? **Check (1)**.

$$T_1/T_{p1} + T_2/T_{p2} + T_3/T_{p3} < 1.00\ ?$$

$$30/100 + 30/100 + 30/100 < 1.00\ ?$$

$$0.90 < 1.00 \quad \textbf{(yes)}$$

(b) Will ISR-1 be served properly? **Check (2) for i=1.**

$$T_{1+} < T_{p1} - T_1\ ?$$
$$\text{(largest of } T_2, T_3, CR, INST) < T_{p1} - T_1\ ?$$
$$T_2 < T_{p1} - T_1\ ?$$
$$30 < 100 - 30\ ?$$
$$30 < 70 \quad \textbf{(yes)}$$

(c) Will ISR-2 be served properly? **Check (2) for i=2.**

$$T_{2+} + N_1 T_1 < T_{P2} - T_2$$
$$\text{largest of } (T_3, CR, INST) + N_1 T_1 < T_{P2} - T_2$$
$$T_3 + 1^* T_1 < T_{P2} - T_2\ ?$$
$$30 + 30 < 70\ ?$$
$$60 < 70 \quad \text{(yes)}$$

(d) Will ISR-3 be served properly? **Check (2) for i=3.**

$$T_{3+} + N_1 T_1 + N_2 T_2 < T_{p3} - T_3$$
$$\text{(larger of CR, INST)} + N_1 T_1 + N_2 T_2 < T_{p3} - T_3$$
$$CR + T_1 + T_2 < T_{p3} - T_3$$
$$15 + 30 + 30 < 70?$$
$$75 < 70 \quad \text{(no)}$$

Only interrupts 1 & 2 can be satisfactorily served. ISR 3 will fail to be serviced in time.

<*Example -2* > A microcontroller is used in an application with just two interrupt sources.

T_{p1} = 60 μs T_{p2} = 100 μs
T_1 = 20 μs T2 = 35 μs
CR= 8 μs (longer than the execution time of the longest instruction)

(a) $T_1/T_{p1} + T_2/T_{p2} < 1.00$? **Check (1)**.
20/60 + 35/100 < 1.00 ?
0.33 + 0.35 < 1.00 ?
0.68 < 1.00 (yes)

(b) $T_{1+} < T_{P1} - T_1$? **Check (2) for i=1.**
(largest of T_2, CR, INST) < $T_{P1} - T_1$?
35 < 60 – 20 (yes)

(c) $T_{2+} + N_1 T_1 < T_{P2} - T_2$? **Check (2) for i=2.**

(larger of CR, INST) + 2 T_1 < $T_{P2} - T_2$?
8 + 40 < 100 -35 ?
48 < 65 (yes)

What if interrupt source 2 is given a higher priority instead?

$T_{2+} < T_{P2} - T_2$?
(larger of T_1, CR. INST) < $T_{P2} - T_2$?
20 < 100 -35 ?
20 < 65 (yes)

$T_{1+} + N_2 T_2 < T_{p1} - T_1$?
(larger of CR, INST) + T_2 < $T_{p1} - T_1$?
8 + 35 < 60 - 20 ?
43 > 40 (no)

Lesson Learned: Designer must be careful to assign each ISR with an appropriate priority. There are several suggested approaches to alleviate multiple interrupt design concerns

- Use a faster CPU (T_i reduces but T_{Pi} remains unchanged) to increase the allowed latency at the cost of more power consumption
- Rearrange the assigned priority order
- Shorten the task time (T_i) as much as possible (coding improvement or algorithm improvement)

What if the N ISRs are aperiodic events? Consider the worst case (assign a short period) and the design can handle the worst scenario.