Chapter 4
Interrupts
ECE 3120

Dr. Mohamed Mahmoud

http://iweb.tntech.edu/mmahmoud/
mmahmoud@tntech.edu
Outline

4.1 What is interrupt?

4.2 Interrupt programming

4.3 IRQ

4.4 Real-time interrupt (RTI)
- An interrupt is an event that requests the CPU to suspend the program execution briefly and run a subroutine called Interrupt Service Routine (ISR).

- After executing the ISR, the CPU resumes the program execution from the point it left the program.

```
org $2000
Entry:
movb #10, $1000
movb #15, $1001
movb #20, $1002
ldaa $1000
adda $1001
adda $1002
staa $1003
```

The ISR code should take some necessary actions to respond to the event.

No command to call the ISR subroutine. The event calls it.

An interrupt means an event has occurred and a relevant action should be taken.
- An event can be a signal coming from a device such as sensor, circuit, timer, etc.

- Example: A sensor outputs a pulse when a train approaches a crossing

- This signal requests interrupt and the MCU runs the relevant ISR that lowers the crossing gate

- Serving an interrupt means executing its subroutine (ISR)
Why are interrupts used?

Better utilization of CPU  →  Programming big applications

To make a delay

1- **Without interrupts**

CPU executes some commands so that the execution time = the delay → remember Delay_yms

→ Wasting the CPU capability

2- **With Interrupt**

CPU does not waste time

It is interrupted every y ms
Example
Write a program to repeatedly turn on a LED for a second and turn it off for a second. Also, read from the keypad and display on the LCD.

Without Interrupt
Begin:
Turn a LED on
Wait 1 second
Turn a LED off
Wait 1 second
Go to begin

Cannot add the keypad and LCD code because CPU is busy all the time to drive the LED

With Interrupt
Main program
Code for keypad and LCD
ISR
If a LED is on
Turn it off
Else
Turn it on
Called every 1 second

Because the CPU is so fast, the LEDs, LCD and keypad run smoothly
To detect events (interrupt request signals)

1- **Without interrupts (Polling)**
   - To avoid missing an interrupt signal, the CPU must continuously read the sensor pin. **The CPU can’t do anything else** (e.g., sensing and controlling other devices)

2- **Interrupt**
   - The interrupt circuit will notify the CPU when there is an event

---

Polling wastes processor resources checking for events that rarely happen
Without Interrupt

Begin:

If the sensor pin is 1, lower the gate

Go to Begin

CPU continuously checks the sensor pin because interrupt can happen at any time

CPU **should not** do anything else to avoid missing the event

With Interrupt

You can write code for other operations

ISR

Called when the sensor gives a pulse

lower the gate
Outline

4.1 What is interrupt?

4.2 Interrupt programming

4.3 IRQ

4.4 Real-time interrupt (RTI)
1- Maskable interrupts:
   - The program can ask the CPU to respond or ignore them, i.e., can be enabled and disabled.
   - **CPU does not execute the ISR if the interrupt is disabled**
   - Most of the interrupts are maskable

2- Non-maskable interrupts:
   - Can’t be disabled or ignored by the CPU.
   - Used for critical applications, e.g., in case of loss of power, the CPU can save the contents of the registers in a nonvolatile memory when the voltage drops below a threshold.
Enable/Disable interrupts

- Two levels of interrupt enabling capability

1- Global masking capability

- When none of the maskable interrupts are desirable, the processor can disable them by **setting** the global interrupt mask flag (I in the CCR)

- To **set** flag I (**disable** all maskable interrupts): sei or orcc #%00010000

To **clear** flag I (**enable** all maskable interrupts): cli or andcc #%11101111

- By default, the I bit is set to 1 during reset. An instruction should be written to clear the I bit to enable maskable interrupts.

2- A local interrupt masking capability

Each interrupt source has an enable bit to enable/disable this interrupt source
- If the global interrupt mask flag (I) is disabled, **all the interrupts are disabled**.

- An interrupt is enabled if **both the global interrupt mask flag (I)** and **the local interrupt mask flags** are enabled.
How to request interrupt from the CPU?

The I-bit is the “gatekeeper” for all the maskable interrupts.

The I-bit is enabled when:
1. Save registers on stack
2. Fetch address of ISR
3. Disable interrupts

When enabled, takes control of CPU, and does these things:

There are many devices that can cause interrupts:
- Each sets a flag
- Each has its own interrupt enable switch

In addition to the internal subsystems, /IRQ is an external pin that can generate an interrupt.

/IRQ

The “/” means that the signal is asserted when low
Interrupt Flag Bit

- When an interrupt source wants to request interrupt, it sets a hardware flag (i.e., a flip-flop).

- Each interrupt source has its interrupt flag.

- The interrupt flag bit **should be cleared** when the interrupt is served.

- As long as this bit is set, there is an interrupt that is not served yet.
- To serve an interrupt, the CPU needs to know the starting address of the interrupt service routine (ISR). This address is called **Interrupt Vector**.

- The interrupt vector of each interrupt source should be stored at a **predefined fixed memory location** called Vector address.

When IRQ requests interrupt, the CPU fetches the ISR starting address from $FFF2:$FFF3

Vector address: $FFF2:$FFF3

Interrupt vector: $4103

We cannot change $FFF2:$FFF3 but we can change $4103

- To set up IRQ interrupt, the programmer should store the ISR starting address at the predefined location.

**Org $FFF2**
**dc.w IRQ_ISR**
### Table 6.1: Interrupt vector map

<table>
<thead>
<tr>
<th>Vector address</th>
<th>Interrupt source</th>
<th>CCR mask</th>
<th>Local Enable</th>
<th>HPRIO value to elevate to highest I bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>$FFFE</td>
<td>Reset</td>
<td>none</td>
<td>none</td>
<td>-</td>
</tr>
<tr>
<td>$FFFC</td>
<td>Clock monitor reset</td>
<td>none</td>
<td>COPCTL(CME, FCME)</td>
<td>-</td>
</tr>
<tr>
<td>$FFFA</td>
<td>COP failure reset</td>
<td>none</td>
<td>COP rate selected</td>
<td>-</td>
</tr>
<tr>
<td>$FF8</td>
<td>Unimplemented instruction trap</td>
<td>none</td>
<td>none</td>
<td>-</td>
</tr>
<tr>
<td>$FF6</td>
<td>SWI</td>
<td>none</td>
<td>none</td>
<td>-</td>
</tr>
<tr>
<td>$FF4</td>
<td>XIRQ</td>
<td>X bit</td>
<td>none</td>
<td>-</td>
</tr>
<tr>
<td>$FF2</td>
<td>IRQ</td>
<td>I bit</td>
<td>INTCR(IRQEN)</td>
<td>$F2</td>
</tr>
<tr>
<td>$FF0</td>
<td>Real time interrupt</td>
<td>I bit</td>
<td>RTICR(RTIE)</td>
<td>$F0</td>
</tr>
<tr>
<td>$FEF</td>
<td>Timer channel 0</td>
<td>I bit</td>
<td>TMSK1(C0I)</td>
<td>$EE</td>
</tr>
<tr>
<td>$FEE</td>
<td>Timer channel 1</td>
<td>I bit</td>
<td>TMSK1(C11)</td>
<td>$EC</td>
</tr>
<tr>
<td>$FEA</td>
<td>Timer channel 2</td>
<td>I bit</td>
<td>TMSK1(C21)</td>
<td>$EA</td>
</tr>
<tr>
<td>$FE8</td>
<td>Timer channel 3</td>
<td>I bit</td>
<td>TMSK1(C31)</td>
<td>$E8</td>
</tr>
<tr>
<td>$FE6</td>
<td>Timer channel 4</td>
<td>I bit</td>
<td>TMSK1(C41)</td>
<td>$E6</td>
</tr>
<tr>
<td>$FE4</td>
<td>Timer channel 5</td>
<td>I bit</td>
<td>TMSK1(C51)</td>
<td>$E4</td>
</tr>
<tr>
<td>$FE2</td>
<td>Timer channel 6</td>
<td>I bit</td>
<td>TMSK1(C61)</td>
<td>$E2</td>
</tr>
<tr>
<td>$FE0</td>
<td>Timer channel 7</td>
<td>I bit</td>
<td>TMSK1(C71)</td>
<td>$E0</td>
</tr>
<tr>
<td>$FDE</td>
<td>Timer overflow</td>
<td>I bit</td>
<td>TMSK2(TOI)</td>
<td>$DE</td>
</tr>
<tr>
<td>$FDC</td>
<td>Pulse accumulator overflow</td>
<td>I bit</td>
<td>PACTL(PAOVI)</td>
<td>$DC</td>
</tr>
<tr>
<td>$FDA</td>
<td>Pulse accumulator input edge</td>
<td>I bit</td>
<td>PACTL(PAI)</td>
<td>$DA</td>
</tr>
<tr>
<td>$F8</td>
<td>SPI serial transfer complete</td>
<td>I bit</td>
<td>SP0CR1(SPIE)</td>
<td>$D8</td>
</tr>
<tr>
<td>$F6</td>
<td>SCI0</td>
<td>I bit</td>
<td>SC0CR2(TIE, TCIE, RIE, ILIE)</td>
<td>$D6</td>
</tr>
<tr>
<td>$F4</td>
<td>SCI1</td>
<td>I bit</td>
<td>SC1CR2(TIE, TCIE, RIE, ILIE)</td>
<td>$D4 (1, 3, 4)</td>
</tr>
<tr>
<td>$F2</td>
<td>ATD0 or ATD1</td>
<td>I bit</td>
<td>ATDxCTL2(ASClE)</td>
<td>$D2</td>
</tr>
<tr>
<td>$F0</td>
<td>MSCAN 0 wakeup</td>
<td>I bit</td>
<td>C0RIER(WUPIE)</td>
<td>$D0 (1*, 2, 2*)</td>
</tr>
<tr>
<td>$FCE</td>
<td>Key wakeup J or H</td>
<td>I bit</td>
<td>KWIEJ[7:0] and KWIEH[7:0]</td>
<td>$C (1, 3, 4)</td>
</tr>
</tbody>
</table>

ATD: Analog to digital
<table>
<thead>
<tr>
<th>Vector address</th>
<th>Interrupt source</th>
<th>CCR mask</th>
<th>Local Enable</th>
<th>HPRIO value to elevate to highest I bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>$FFCC</td>
<td>Modulus down counter underflow</td>
<td>I bit</td>
<td>MCCTL(MCZI)</td>
<td>$CC</td>
</tr>
<tr>
<td>$FFCA</td>
<td>Pulse accumulator B overflow</td>
<td>I bit</td>
<td>PBCTL(PBOVI)</td>
<td>$CA</td>
</tr>
<tr>
<td>$FFC8</td>
<td>MSCAN 0 errors</td>
<td>I bit</td>
<td>C0RIER(RWRNIE, TWRNIE, RERRIE, TERRIE, BOFFIE, OVRIE)</td>
<td>$C8 (2*,3,4)</td>
</tr>
<tr>
<td>$FFC6</td>
<td>MSCAN 0 receive</td>
<td>I bit</td>
<td>C0RIER(RXFIE)</td>
<td>$C6 (2*,3,4)</td>
</tr>
<tr>
<td>$FFC4</td>
<td>MSCAN 0 transmit</td>
<td>I bit</td>
<td>C0TCR(TXEIE[2:0])</td>
<td>$C4 (2*,3,4)</td>
</tr>
<tr>
<td>$FFC2</td>
<td>CGK lock and limp home</td>
<td>I bit</td>
<td>PLLCR(LOCKIE, LHIE)</td>
<td>$C2 (3,4)</td>
</tr>
<tr>
<td>$FFC0</td>
<td>IIC Bus</td>
<td>I bit</td>
<td>IBCR(IBIE)</td>
<td>$C0 (3,4)</td>
</tr>
<tr>
<td>$FFBE</td>
<td>MSCAN 1 wakeup</td>
<td>I bit</td>
<td>C1RIER(WUPIE)</td>
<td>$BE (3,4)</td>
</tr>
<tr>
<td>$FFBC</td>
<td>MSCAN 1 errors</td>
<td>I bit</td>
<td>C1RIER(RWRNIE, TWRNIE, RERRIE, TERRIE, BOFFIE, OVRIE)</td>
<td>$BC (3,4)</td>
</tr>
<tr>
<td>$FFBA</td>
<td>MSCAN 1 receive</td>
<td>I bit</td>
<td>C1RIER(RXFIE)</td>
<td>$BA (3,4)</td>
</tr>
<tr>
<td>$FFB8</td>
<td>MSCAN 1 transmit</td>
<td>I bit</td>
<td>C1TCR(TXEIE[2:0])</td>
<td>$B8 (3,4)</td>
</tr>
<tr>
<td>$FFB6</td>
<td>Reserved</td>
<td>I bit</td>
<td></td>
<td>$B6</td>
</tr>
<tr>
<td>$FF80-$FB5</td>
<td>Reserved</td>
<td>I bit</td>
<td></td>
<td>$80-$B4</td>
</tr>
</tbody>
</table>

Note: 1. Available in 812 A4  
2. Used as BDLC interrupt vector for 912B32, 912BE32  
3. Available in D60  
4. Available in DG128 (DT128)
- It is possible that several interrupts would be pending at the same time.
- The CPU cannot serve more than one interrupt at the same time.
- CPU has to decide which interrupt to serve first → the interrupts should be prioritized.
- The ISR of the highest priority interrupt is executed first.
- In table 6.1, **the interrupt that has higher vector address has higher priority**, e.g., /IRQ has the higher priority than timer channel 0.
- However, we can **raise one** of the maskable interrupts to the highest level so that it can get **quicker service**.
- To do that, write the low byte of the vector address to the highest priority interrupt register (HPRIO).

```
movb #$E0, HPRIO
```

**Raise the timer channel 7 interrupt to the highest priority.** The relative priorities of the other interrupts remain the same.

The address of HPRIO is defined in “mc9s12dg256.inc” file.
1. When an event occurs, a **flag** bit should be set to interrupt the CPU.

2. This interrupt request is served if:-
   
   (1) **I** bit and **local interrupt enable bit** are **enabled** in case of maskable interrupts
   
   (2) It is the only interrupt or the highest priority interrupt if there are several interrupts pending for service

3. To serve the interrupt, the CPU automatically pushes all the registers (except SP) on the stack (9 bytes total). This includes the return address stored in PC and CCR register

   **do not mess up stack!**

---

**Stack on entry of an ISR**

<table>
<thead>
<tr>
<th>SP</th>
<th>CCR</th>
</tr>
</thead>
<tbody>
<tr>
<td>SP+1</td>
<td>A</td>
</tr>
<tr>
<td>SP+2</td>
<td>B</td>
</tr>
<tr>
<td>SP+3</td>
<td>X high byte</td>
</tr>
<tr>
<td></td>
<td>X low byte</td>
</tr>
<tr>
<td>SP+5</td>
<td>Y high byte</td>
</tr>
<tr>
<td></td>
<td>Y low byte</td>
</tr>
<tr>
<td>SP+7</td>
<td>PC high byte</td>
</tr>
<tr>
<td></td>
<td>PC low byte</td>
</tr>
</tbody>
</table>
5. The CPU prevents further interrupts from occurring till the ISR is done by setting the I bit. Nested interrupts are not allowed.

6. Resolve the interrupt vector and transfer control to the interrupt service routine (ISR). PC = the ISR starting address

7. **Cancel the interrupt request** by clearing the interrupt flag.
   
   Without this step the interrupt will be executed over and over again and the main program would never execute again

8. Execute the ISR instructions.
   
   **Use all the CPU registers without fear of interfering with the main program**, but for memory locations, it is the programmer’s responsibility to ensure that the ISR does not change memory locations used by the main program

9. The last instruction in an ISR is always “RTI” (return from interrupt)
   
   - RTI retrieves the registers’ original values before executing the interrupt from the stack. Enable I bit and return back to the main program
Interrupt programming

1- Main program

1.1 Initializing the interrupt vector

1.2 Configuration, e.g., respond to level or falling edge, set time out, etc.

1.3 Enable interrupt: Global and local

- It is important not to enable the interrupts before the initialization is done

2- The interrupt subroutine

2.1 Clear the interrupt flag bit

2.2 Must end with RTI instruction
4.1 What is interrupt?

4.2 Interrupt programming

4.3 IRQ

4.4 Real-time interrupt (RTI)
- Port E pin 1 (PE1)
- The only external maskable interrupt for the HCS12.
1- Main program

1.1 Initializing the interrupt vector

```
Org $FFF2
dc.w IRQ_ISR
```

IRQ_ISR is the name of the routine

1.2 Configuration

When does the interrupt occur? Low level (/IRQ = 0) or falling edge (/IRQ transfers from 1 to 0).

1.3 Enable local interrupt

1.2 and 1.3 are done by setting the interrupt control register (INTCR) that has the address $001E.
- **Local enable bit**: IRQEN bit 6 of the IRQCR register. IRQEN = 1 enabled and IRQEN = 0 disabled

- **The triggering method**: IRQE = 0 respond to low level, and IRQE = 1 responds to falling edge

```
    movb #$40,INTCR  ;enable IRQ interrupt and respond to low level edge
```

### INTCR register

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>IRQE</td>
<td>IRQEN</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

reset: 0 1 0 0 0 0 0 0

**IRQE -- /IRQ edge sensitive only bit**

- 1 = /IRQ pin responds only to falling edge
- 0 = /IRQ pin responds to low level.

**IRQEN -- /IRQ enable bit**

- 1 = /IRQ pin interrupt enabled
- 0 = /IRQ pin interrupt disabled
Cli ; to enable global interrupts

2- The interrupt subroutine

2.1 Clear the interrupt flag bit

  Automatically cleared by the microcontroller

2.2 Must end with RTI instruction
This program is similar to the one at 3-41 but by using /IRQ

The /IRQ pin of HCS12 is connected to push button 3 to make interrupt each time the button is pressed. Write a program to turn on LED number 0 and turn off the other LEDs initially. Each time /IRQ is interrupted, the LED is off an the next one on left is on. Always only one LED is on and each time the /IRQ interrupt happens, the on LED shifts to left.

In main program:
1. Write ISR starting address into the /IRQ vector address
2. Configure Port B as output port
3. Configure /IRQ to respond to low level
4. Enable interrupts including IRQ local interrupt and global maskable interrupts
5. Run endless loop while awaiting interrupt

In the ISR:
1. Shift port B
2. If Port B = 0, Port B =1
absentry entry ;entry point of application
include 'mc9s12dg256.inc'

Org $FFF2
dc.w IRQ_ISR       ;load IRQ ISR vector

Org $1000
entry: 1ds #$2500

movb #$FF,DDRB       ; port b is output
movb #%00000000,DDRH ; port H is input - push buttons

bset DDRJ,#$02 ;configure PJ1 pin for output
bclr PTJ,#$02 ;enable LEDs to light

movb #$FF,DDRP ; disable 7 segments that are connected
movb #$0F,PTP ;'

movb #%00000001,PORTB ; first LED is on

movb #$40,INTCR ;enable IRQ interrupt and respond to low level
cli ; enable interrupt systems

LOOP: bra LOOP ; wait for interrupt - you can replace this
command with a code if you want to do more things
IRQ_ISR:

;;;;; Debouncing --------------------------
    ldy #20
    jsr Delay_yms

here: brclr PORTE,#%00000010,here

    ldy #20
    jsr Delay_yms

;------------------------------------------------

      lsl PORTB
      bne _end

      movb #%00000001,PORTB ; reinitialize to 01h

_end:
    rti
4.1 What is interrupt?

4.2 Interrupt programming

4.3 IRQ

4.4 RTI
Real-time interrupt (RTI)

- The RTI can be used to generate periodic interrupts at a fixed rate, e.g., an interrupt every 100 ms.

### IRQ Interrupt programming

1. Main program

1.1 Initializing the interrupt vector

```
Org $FFF0
dc.w RTI_ISR
```

RTI_ISR is the name of the routine

1.2 Configuration: setting the timeout

\[
\text{Timeout (µs)} = \frac{(N+1) \times 2^{(M+9)}}{8}
\]

- N = RTR[0:3]
- M = RTR[4:6]

M = 0 means RTI off

- The address of RTICTL is $003B

Figure 6.16 CRG RTI control register (RTICTL)
The numbers in the table are \((N+1) \times 2^{(M+9)}\). Divide them by 8 to get the delay in \(\mu s\).

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 ((\div 1))</td>
<td>2(^{10})</td>
<td>10.24 ms</td>
</tr>
<tr>
<td>0001 ((\div 2))</td>
<td>2(\times)2(^{10})</td>
<td>10.24 ms</td>
</tr>
<tr>
<td>0010 ((\div 3))</td>
<td>3(\times)2(^{10})</td>
<td>10.24 ms</td>
</tr>
<tr>
<td>0011 ((\div 4))</td>
<td>4(\times)2(^{10})</td>
<td>10.24 ms</td>
</tr>
<tr>
<td>0100 ((\div 5))</td>
<td>5(\times)2(^{10})</td>
<td>10.24 ms</td>
</tr>
<tr>
<td>0101 ((\div 6))</td>
<td>6(\times)2(^{10})</td>
<td>10.24 ms</td>
</tr>
<tr>
<td>0110 ((\div 7))</td>
<td>7(\times)2(^{10})</td>
<td>10.24 ms</td>
</tr>
<tr>
<td>0111 ((\div 8))</td>
<td>8(\times)2(^{10})</td>
<td>10.24 ms</td>
</tr>
<tr>
<td>1000 ((\div 9))</td>
<td>9(\times)2(^{10})</td>
<td>10.24 ms</td>
</tr>
<tr>
<td>1001 ((\div 10))</td>
<td>10(\times)2(^{10})</td>
<td>10.24 ms</td>
</tr>
<tr>
<td>1010 ((\div 11))</td>
<td>11(\times)2(^{10})</td>
<td>10.24 ms</td>
</tr>
<tr>
<td>1011 ((\div 12))</td>
<td>12(\times)2(^{10})</td>
<td>10.24 ms</td>
</tr>
<tr>
<td>1100 ((\div 13))</td>
<td>13(\times)2(^{10})</td>
<td>10.24 ms</td>
</tr>
<tr>
<td>1101 ((\div 14))</td>
<td>14(\times)2(^{10})</td>
<td>10.24 ms</td>
</tr>
<tr>
<td>1110 ((\div 15))</td>
<td>15(\times)2(^{10})</td>
<td>10.24 ms</td>
</tr>
<tr>
<td>1111 ((\div 16))</td>
<td>16(\times)2(^{10})</td>
<td>10.24 ms</td>
</tr>
</tbody>
</table>

**Max. delay = 131.072 ms**
To set the delay to 131.072ms, \( M \) (RTR[6:4]) = 111 and \( N \) (RTR[3:0]) = 1111

- Code: `movb #$7F,RTICTL`

**1.3 Enable local interrupt**

**2.1 Clear the interrupt flag bit**

- RTI interrupt enable bit (RTIE) is bit 7 in CRGINT
- Set this bit to enable RTI interrupt

```
bset CRGINT,#$80 ; RTIE = 1 (enables RTI interrupts) crgint address is $0038
```
- Interrupt flag (RTIF): RTI requests interrupt service by setting the flag RTIF in CRGFLG register. This flag will be set at a fixed rate.

- It is the programmer responsibility to clear this flag when the interrupt is served.

  \[
  \text{movb}\ #$80,\text{CRGFLG} \; \text{;clear RTIF by writing} \; 1.\]

Example: Program the RTI to flash a LED at a rate of about 1 per second, i.e., repeatedly turn on for one second, then turn off for 1 sec.

- The maximum delay using RTI is 131.072ms as shown in slide 4-27

- RTI will be interrupted 8 times in one second. \(8 \times 131.072 = \) around 1 sec

- Change the LED status every 8 interrupts to turn the LED on for one second and off for other second.

  RTI_ISR
  - Clear the interrupt flag (write 1 to the RTIF bit in CRGFLG)
  - Increment count
  - If count is equal to 8, toggle PB0 and clear count
  - Execute the RTI instruction to return to the main program
Main program
- Load the RTI ISR vector into locations $FFF0:FFF1
- Set up PB0 for output
- Set up RTI rate to 131.072ms (bits RTR6:RTR0 in register RTICTL)
- Enable RTI interrupts (set RTIE bit in register CRGINT)
- Initialize count to zero (count is a variable in fixed global memory)
- Turn on interrupt system (e.g., the CLI instruction)
- Go into an infinite loop waiting for interrupt

```plaintext
absentry entry ;entry point of application
include 'mc9s12dg256.inc'

org $1000
rti_count dc.b 0

Org $FFF0
dc.w RTI_ISR ;load RTI ISR vector

Org $2000

entry:  lds  #$2500
```
bset DDRJ,#$02 ;configure PJ1 pin for output
bclr PTJ,#$02 ;enable LEDs to light

movb #$FF,DDRB ; port b is output
movb #$01,PORTB ; Turn the LED on

movb #$FF,DDRP ; disable 7 segments that are connected
movb #$0F,PTP ; ‘’

movb #$7F,RTICTL ; set up slowest RTI rate

bset CRGINT,#$80 ; RTIE=1 (enables RTI interrupts)
cli ; enable interrupt systems

LOOP: bra LOOP ; wait for interrupt - you can replace this command with a code if you want to do more things
We use memory location “rti_count” instead of a register because

**Interrupt routines do not memorize the register’s value resulted from last routine call.**

This is because interrupts push all the registers and pull the original values back when they return to the main program.
absentry entry ;entry point of application
include 'mc9s12dg256.inc'

org $1000
rti_count dc.b 0

Org $FFF0
dc.w RTI_ISR ;load RTI ISR vector

Org $FFF2
dc.w IRQ_ISR ;load IRQ ISR vector

Org $2000

entry: lds #$2500

movb #$FF,DDRB ; port b is output
movb #$01,PORTB ; port b is output
bset DDRJ,#$02 ;configure PJ1 pin for output
bclr PTJ,#$02 ;enable LEDs to light
movb %00000000,DDRH ; port H is input - push buttons

movb #$FF,DDRP ; disable 7 segments that are connected
movb #$0F,PTP ;

movb #$7F,RTICTL ; set up slowest RTI rate
bset CRGINT,#$80 ; RTIE=1 (enables RTI interrupts)

movb #$40,INTCR ; enable IRQ interrupt and respond to low level

cli ; enable interrupt systems

LOOP: bra LOOP ; wait for interrupt - you can replace this command with a code if you want to do more things

RTI_ISR:

movb #$80,CRGFLG ; clear RTIF by writing a 1 to it.

inc rti_count
ldaa rti_count
cmpa #8 ; check if count = 8
bne RTI_done ; if no, we are done
clr rti_count ; if yes clear count
lda PORTB ; --- togle PB0
eora #1
staa PORTB

RTI_done: rti

IRQ_ISR:

;;;;; Debouncing -----------------------

ldy #20 jsr Delay_yms
here: brclr PORTE,#%00000010,here
ldy #20 jsr Delay_yms

;------------------------------------------------

ldaa CRGINT
eora #$80 ; change the status of bit 7 to enables/disable RTI interrupts
staa CRGINT

rti
Thank You!

Questions

Mohamed Mahmoud