EE 319K
Introduction to Embedded Systems

Lecture 9: Interrupts, Output Compare Periodic Interrupts

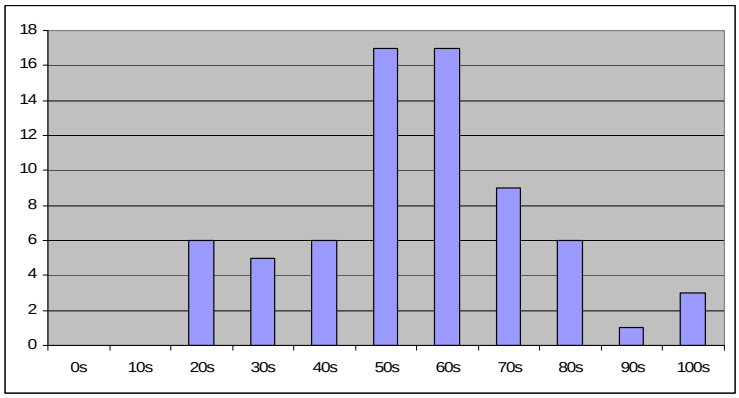
Read Book Sections 9.1, 9.2, 9.4, 9.6.1, 9.6.2, 9.10

Announcements

- No homework next week
  - Familiarize yourself with Metrowerks
    - Next homework will be in C using Metrowerks
- No lab next week
  - Go to your lab session to get Metrowerks introduction by the TAs

Exam 2

- Statistics



  - Average: 60, Median: 60, Std. dev.: 20, Range: 22-100
    - Exam 2G: 60/59/17/22-93 (raw avg/high: 52/100)
    - Exam 2F: 60/62/22/25-100 (raw avg/high: 48/76

Agenda

- Recap
  - Fixed-point numbers
  - I/O synchronization
  - LCD interfacing
- Outline
  - Interrupts
  - Output compare interrupts on the 9S12
  - Metrowerks C programming

Interrupts and Threads

Interrupt

- An **interrupt** is the automatic transfer of software execution in response to a hardware event (trigger) that is asynchronous with current software execution.
- H/w event:
  - An external I/O device (like a keyboard or printer) requesting service,
  - An Internal event (like an op code fault, or a periodic timer) occurs
- When the hardware needs service it performs a *busy to done* state transition (sets a flag)

Thread

- A **thread** is defined as the path of action of software as it executes.
- ISR is a **background thread**
  - A new background thread is created for each interrupt request.
  - Local variables and registers used in the interrupt service routine are unique
  - Threads share globals

Interrupt Solution Scenarios

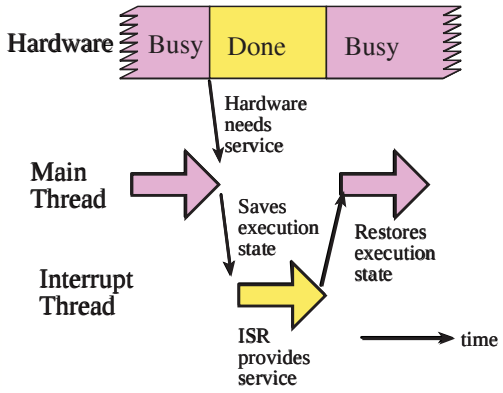
Busy-wait	Interrupts	DMA
Predictable	Variable arrival times	Low Latency
Simple I/O	Complex I/O, different speeds	High Bandwidth
Fixed load	Variable load	
Dedicated, single thread	Other functions to do	
Single Process	Multithread or multiprocess	
Nothing else to do	Infrequent but important alarms; periodic background tasks	
	Program errors	
	Overflow, invalid op code	
	Illegal stack or memory access	
	Machine errors	
	Power failure, memory fault	
	Breakpoints for debugging	
	Real time clocks	
	Data acquisition and control	

Interrupt Processing

What happens when an interrupt is processed?

1. The execution of the main program is suspended
   - The current instruction is finished,
   - pushes registers on the stack
   - sets the I bit (disallow interrupts while processing interrupt)
   - gets the vector address from high memory (The ISR previously was registered at this vector address)
2. the interrupt service routine (ISR), or background thread is executed,
   - clears the flag that requested the interrupt
   - performs necessary operations
   - communicates using global variables
3. the main program is resumed when ISR executes `rti`.
   - pulls the registers from the stack

Interrupt Processing



Interrupts

- Each potential interrupt source has a separate **arm** bit. E.g., **C0I**
  - Set arm bits for those devices from which it wishes to accept interrupts,
  - Deactivate arm bits in those devices from which interrupts are not allowed
- Each potential interrupt source has a separate **flag** bit. E.g., **C0F**
  - hardware sets the flag when it wishes to request an interrupt
  - software clears the flag in ISR to signify it is processing the request
- Interrupt **enable** bit, I, which is in the condition code register.
  - enable all armed interrupts by setting I=0, or **cli**
  - disable all interrupts by setting I=1, or **sei**
    I=1 does not dismiss the interrupt requests, rather it postpones

Trigger-Arm-Enable

Three conditions must be true simultaneously for an interrupt to occur:

1. Initialization software will set the arm bit. e.g., **C0I** individual control bit for each possible flag that can interrupt
2. When it is convenient, the software will enable, **I=0** allow all interrupts now
3. Hardware action (trigger) sets a flag to indicate service request e.g., **C0F**
   - new input data ready,
   - output device idle,
   - periodic,
   - alarm

9S12 Interrupts (Partial)

0xFFD4	interrupt 21 SCI1
0xFFD6	interrupt 20 SCI0
0xFFDE	interrupt 16 timer overflow
0xFFE0	interrupt 15 timer channel 7
0xFFE2	interrupt 14 timer channel 6
0xFFE4	interrupt 13 timer channel 5
0xFFE6	interrupt 12 timer channel 4
0xFFE8	interrupt 11 timer channel 3
0xFFEA	interrupt 10 timer channel 2
0xFFEC	interrupt 9 timer channel 1
0xFFEE	**interrupt 8 timer channel 0**
0xFFF0	interrupt 7 RTI real time interrupt
0xFFF6	interrupt 4 SWI software interrupt
0xFFF8	interrupt 3 trap software interrupt
0xFFFE	interrupt 0 reset

Interrupt Number is used by Metrowerks C Compiler

Periodic Interrupts using Output Compare

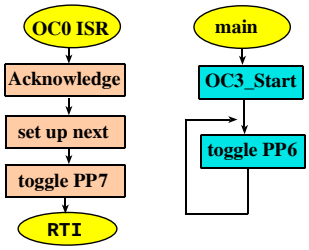
Address	msb															lsb	Name
$0044	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**	**TCNT**
$0050	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**	**TC0**
$0052	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	TC1
$0054	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	TC2
$0056	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	TC3
$0058	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	TC4
$005A	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	TC5
$005C	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	TC6
$005E	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	TC7

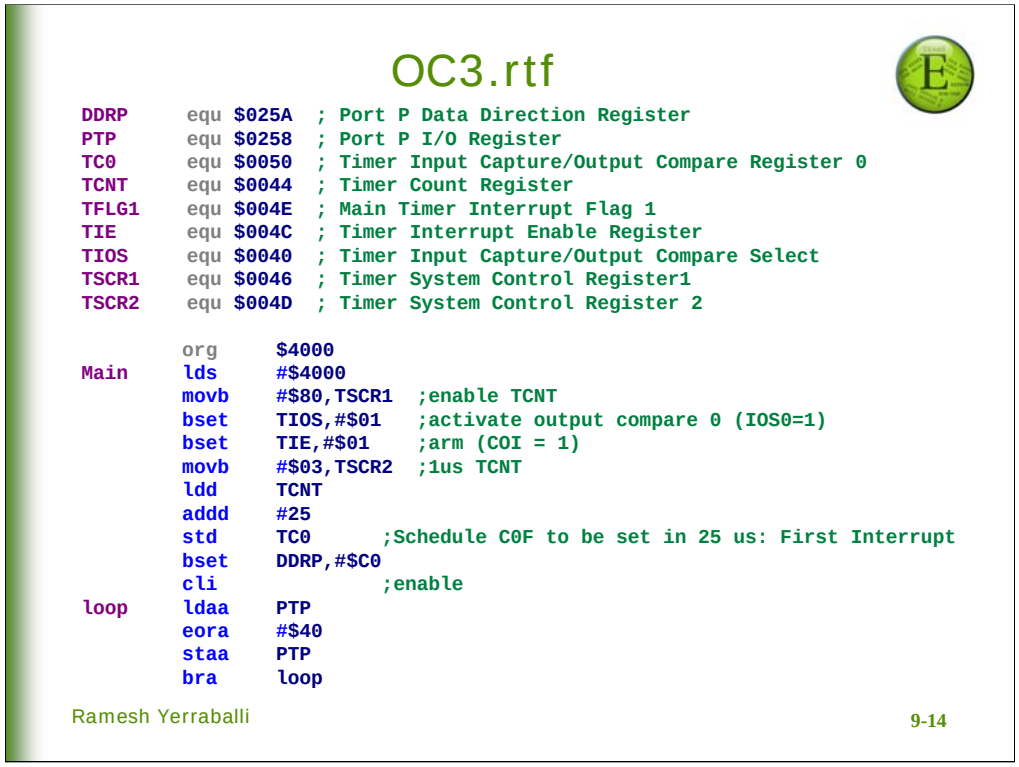
Address	Bit 7	6	5	4	3	2	1	Bit 0	Name
$0046	**TEN**	TSWAI	TSBCK	TFFCA	0	0	0	0	TSCR1
$004D	TOI	0	0	0	TCRE	**PR2**	**PR1**	**PR0**	TSCR2
$0040	IOS7	IOS6	IOS5	IOS4	IOS3	IOS2	IOS1	**IOS0**	TIOS
$004C	C7I	C6I	C5I	C4I	C3I	C2I	C1I	**C0I**	TIE
$004E	C7F	C6F	C5F	C4F	C3F	C2F	C1F	**C0F**	TFLG1
$004F	TOF	0	0	0	0	0	0	0	TFLG2

The rate is dependent on PLL, TSCR2 and 1000 in ISR

Example: OC3.rtf

Note: OC3.rtf is based on, but modified from OC.rtf that comes with TExaS installation





Access to the shared port PTP will create a race condition in this example. What can happen is that the interrupt occurs in the middle of toggling PP6 in the main loop, where toggling means reading PTP, toggling bit 6 and writing the result back to PTP. If the interrupt occurs right after the 'ldaa PTP', for example, the interrupt handler will toggle PP7, but once control returns back to main, it will have the old value of PTP in RegA, which means PP7 will be toggled back immediately. Two ways to solve this issue:

1.Use different, non-shared ports for the two LEDs

2.Turn the code to toggle the LED in main into a critical section that can not be interrupted, i.e. executes in a so-called *atomic* (= non-interruptible) fashion.

The latter can be achieved, for example, by disabling all interrupts while toggling:

```
loop   sei                  ; disable interrupts, enter critical section
       ldaa   PTP
       eora   #$40
       staa   PTP
       cli                  ; re-enable interrupts, leave critical section
       bra    loop
```

...OC3.rtf

```
;interrupts every 1000 TCNT cycles
;every 1ms, assuming TCNT at 1us
TC0handler
        movb #$01,TFLG1  ;acknowledge, clear C0F
        ldd  TC0
        addd #1000
        std  TC0       ;setup the time for next interrupt
         ; this stuff (here toggle PP7)gets executed every 1 ms
        ldaa PTP
        eora #$80
        staa PTP
        rti

        org  $FFEE      ; output compare 0 interrupt vector
        fdb  TC0handler ; Register ISR (Handler to run when OC0
                        ; interrupt occurs)
        org  $FFFE
        fdb  Main        reset vector
```


Running an FSM in the background:

•Turn FSM engine into a subroutine (“execute” one state each time it is called)

•Call FSM engine routine from OC Timer interrupt (at regular intervals)

ISR Dos

Things you must do in every interrupt service routine

- **Acknowledge (clear flag that requested the interrupt)**

Things you must do in an OC interrupt service routine

- **Acknowledge (make C0F become one)**
- **Set the timer to specify when to interrupt next**

Ritual

Things you must do in every ritual

- ❑ Arm (specify a flag may interrupt)
- ❑ Enable (allow all interrupts on the 9S12)

Things you must do in an OC ritual

- ❑ Turn on TCNT (TEN=1)
- ❑ Set channel to output compare (TIOS)
- ❑ Specify TCNT rate (TSCR2, PACTL, PLL)
- ❑ Arm (C0I=1)
- ❑ When to generate first interrupt
- ❑ Enable (I=0)