Experiment 7: Interrupts

Instructional Objectives:

- To understand how to implement interrupts to simultaneous perform several time and event sensitive tasks.
- To understand how to alter the linker parameter file to handle multiple entries in the vector table.

Introduction:

The final lab for the ECE362 course explains how to implement interrupts. An interrupt is a signal to the microcontroller that is generated by a hardware or software event. Whenever an interrupt occurs, the program completes the current instruction and then jumps to a special subroutine called an interrupt service routine (ISR). In previous labs, polling was used to control just one or two peripherals on the I/O board. However, polling is not an ideal way to control several peripherals in one program.

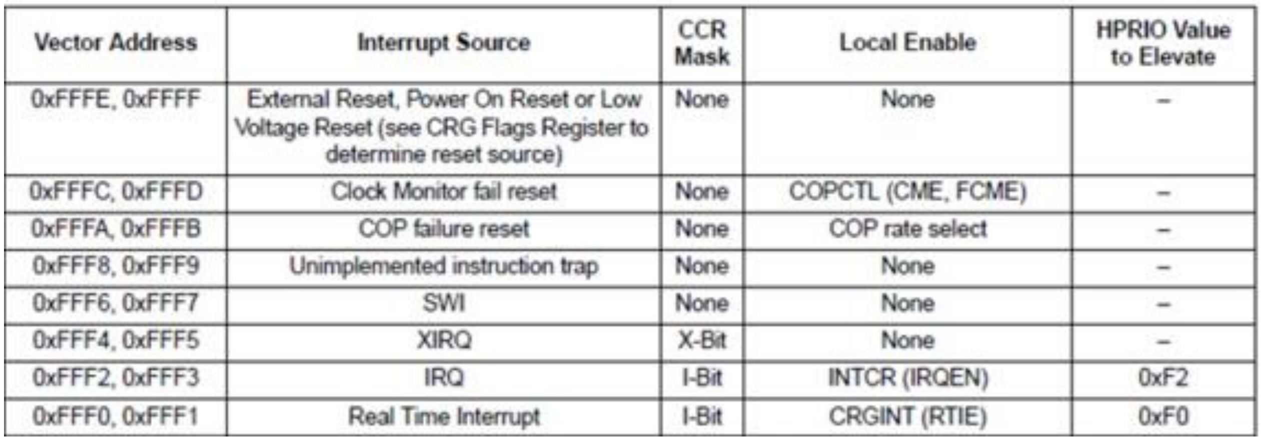
A good analogy is the ECE362 lecture. Suppose that at the end of every lecture, the professor asks every student one at a time if they understood the lecture or if they have any questions. This is effective if the classroom only has one or two students. However, in a classroom with 40-50 students, this becomes ineffective. In this scenario, the student raises their hand to interrupt the lecture and ask a question. The professor completes their sentence and then stops the lecture to answer the question. Once the question has been answered, the professor resumes the lecture from where they left off.

This experiment will introduce the concept of interrupts and give two examples of how to implement interrupt sources on the HC(S)12. In addition, this experiment will provide instructions for how to alter the linker parameter file to handle interrupts.

Interrupts:

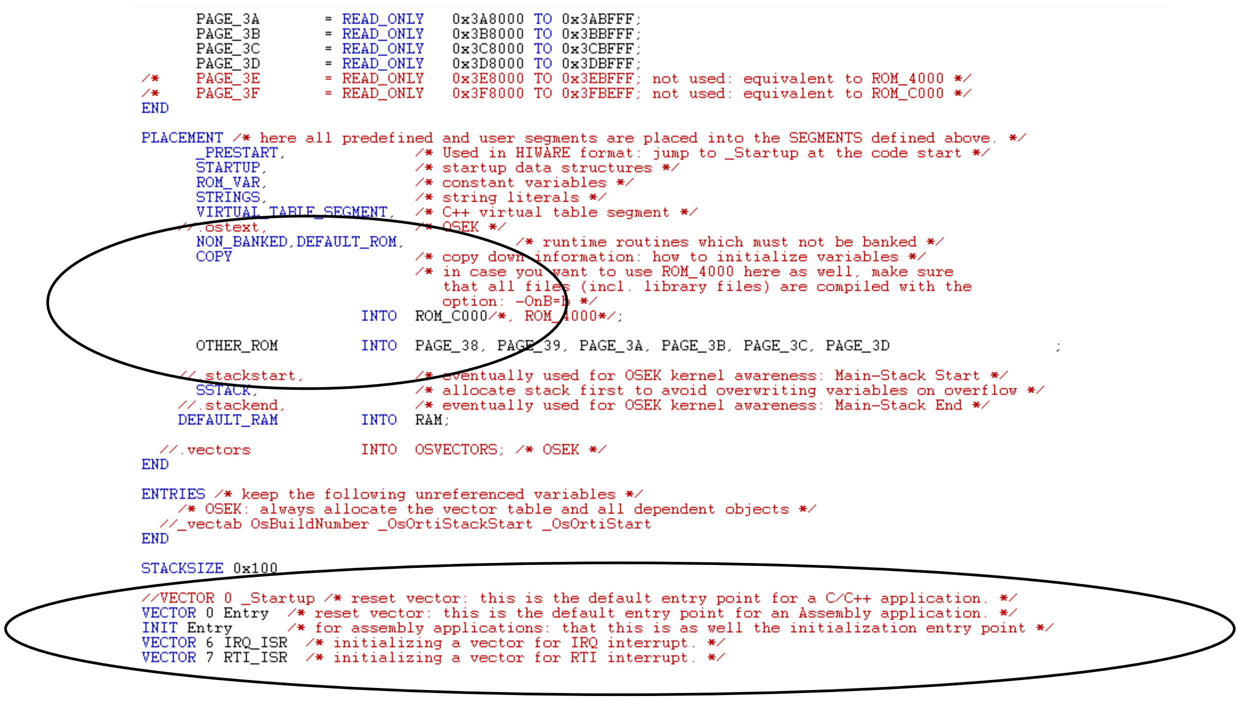
As stated above, an interrupt is a signal to the microcontroller that is generated by a hardware or software event. There are several sources for interrupt signals, including external hardware signals, timers, software interrupts, and others. Each interrupt has its own interrupt service routine (ISR). Upon the occurrence of an interrupt event, the following steps occur:

- The CPU finishes the currently executed instruction.
- The return address is pushed to the stack.
- The contents of all the registers are pushed onto the stack.
- The program jumps to the ISR.
- The program executes the ISR.
- The contents of all the registers are pulled from the stack.
- The CPU returns from the ISR and continues the main program.



Vector Address	Interrupt Source	CCR Mask	Local Enable	HPRIO Value to Elevate
0xFFFE, 0xFFFF	External Reset, Power On Reset or Low Voltage Reset (see CRG Flags Register to determine reset source)	None	None	–
0xFFFC, 0xFFFD	Clock Monitor fail reset	None	COPCTL (CME, FCME)	–
0xFFFA, 0xFFFB	COP failure reset	None	COP rate select	–
0xFFF8, 0xFFF9	Unimplemented instruction trap	None	None	–
0xFFF6, 0xFFF7	SWI	None	None	–
0xFFF4, 0xFFF5	XIRQ	X-Bit	None	–
0xFFF2, 0xFFF3	IRQ	I-Bit	INTCR (IRQEN)	0xF2
0xFFF0, 0xFFF1	Real Time Interrupt	I-Bit	CRGINT (RTIE)	0xF0

Above is an excerpt from the vector table for the HC(S)12. The interrupt vector table is a predefined memory space for storing interrupt vectors. Interrupt vectors are pointers to the beginning of an ISR. Every interrupt source has an associated interrupt vector. On the HC(S)12, interrupt vectors begin at the address $FF80 and ends at the address $FFFF. The higher the vector is in the table, the higher priority that interrupt has. For example, external reset has a higher priority than real time interrupts. The vector table is initialized at the bottom of the linker parameter file, as seen below.



```
      PAGE_3A        = READ_ONLY   0x3A8000 TO 0x3ABFFF;
      PAGE_3B        = READ_ONLY   0x3B8000 TO 0x3BBFFF;
      PAGE_3C        = READ_ONLY   0x3C8000 TO 0x3CBFFF;
      PAGE_3D        = READ_ONLY   0x3D8000 TO 0x3DBFFF;
/*    PAGE_3E        = READ_ONLY   0x3E8000 TO 0x3EBFFF; not used: equivalent to ROM_4000 */
/*    PAGE_3F        = READ_ONLY   0x3F8000 TO 0x3FBEFF; not used: equivalent to ROM_C000 */
END

PLACEMENT /* here all predefined and user segments are placed into the SEGMENTS defined above. */
      _PRESTART,                /* Used in HIWARE format: jump to _Startup at the code start */
      STARTUP,                  /* startup data structures */
      ROM_VAR,                  /* constant variables */
      STRINGS,                  /* string literals */
      VIRTUAL_TABLE_SEGMENT,    /* C++ virtual table segment */
    //.ostext,                  /* OSEK */
      NON_BANKED,DEFAULT_ROM,             /* runtime routines which must not be banked */
      COPY                      /* copy down information: how to initialize variables */
                                /* in case you want to use ROM_4000 here as well, make sure
                                   that all files (incl. library files) are compiled with the
                                   option: -OnB=b */
                          INTO  ROM_C000/*, ROM_4000*/;

      OTHER_ROM           INTO  PAGE_38, PAGE_39, PAGE_3A, PAGE_3B, PAGE_3C, PAGE_3D          ;

    //.stackstart,              /* eventually used for OSEK kernel awareness: Main-Stack Start */
      SSTACK,                   /* allocate stack first to avoid overwriting variables on overflow */
    //.stackend,                /* eventually used for OSEK kernel awareness: Main-Stack End */
    DEFAULT_RAM           INTO  RAM;

  //.vectors              INTO  OSVECTORS; /* OSEK */
END

ENTRIES /* keep the following unreferenced variables */
    /* OSEK: always allocate the vector table and all dependent objects */
  //_vectab OsBuildNumber _OsOrtiStackStart _OsOrtiStart
END

STACKSIZE 0x100

//VECTOR 0 _Startup /* reset vector: this is the default entry point for a C/C++ application. */
VECTOR 0 Entry  /* reset vector: this is the default entry point for an Assembly application. */
INIT Entry      /* for assembly applications: that this is as well the initialization entry point */
VECTOR 6 IRQ_ISR  /* initializing a vector for IRQ interrupt. */
VECTOR 7 RTI_ISR  /* initializing a vector for RTI interrupt. */
```

Note: The keyword DEFAULT_ROM has been moved next to the label NON_BANKED. This forces the entire program to be allocated to the page of non-banked memory starting at the memory location 0xC000. Failing to do this will result in an error upon assembling the program.

Real Time Interrupts:

Real-time interrupts are an interrupt signal that occurs whenever the internal clock of the microcontroller reaches a certain value. This generates a hardware interrupt that occurs at a programmed periodic rate. Real-time interrupts are useful for periodic tasks, such as motor controls, keypad reads, timed patterns on LEDs, and playing different tones through a speaker.

Real-time interrupts must be enabled both at the CPU and at the interrupt enable register. To enable maskable interrupts, the I flag of the CCR must be cleared. This can be done using the `CLI` instruction. In addition, the interrupt enable register (CRGINT) must also be configured to enable real time interrupts. This register is located at memory location `$0038`. Bit 7 of this register is the bit that enables/disables real time interrupts. To enable Real-time interrupts, set bit 7 of this register. Please refer to the excerpt from the MC9S12E128 below:

4.3.2.5 CRG Interrupt Enable Register (CRGINT)

This register enables CRG interrupt requests.

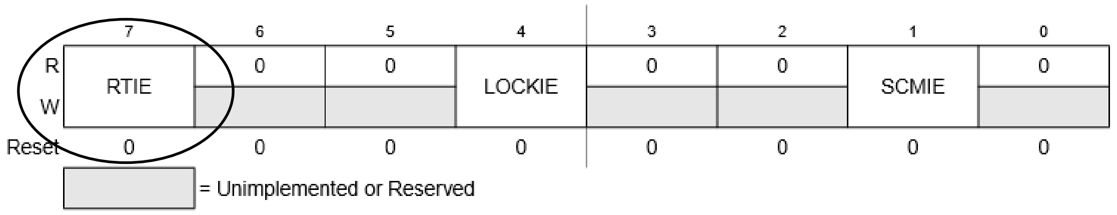


Figure 4-8. CRG Interrupt Enable Register (CRGINT)

Read: anytime

Write: anytime

Table 4-3. CRGINT Field Descriptions

Field	Description
7 RTIE	**Real-Time Interrupt Enable Bit** 0 Interrupt requests from RTI are disabled. 1 Interrupt will be requested whenever RTIF is set.
4 LOCKIE	**Lock Interrupt Enable Bit** 0 LOCK interrupt requests are disabled. 1 Interrupt will be requested whenever LOCKIF is set.
1 SCMIE	**Self-Clock Mode Interrupt Enable Bit** 0 SCM interrupt requests are disabled. 1 Interrupt will be requested whenever SCMIF is set.

To control the Real-time Interrupt interval, the control register (RTICTL) at address `$003B` must be set to a value corresponding to the desired rate. The values are used to enable or disable t-gates and set the modulus counter that scales the OSCCLK from the crystal. The circuit that handles this scaling operation is presented in the lecture slides. Please refer to the next page for the data sheet information on how to configure the control register to reach a desired RTI interval.

4.3.2.8 CRG RTI Control Register (RTICTL)

This register selects the timeout period for the real-time interrupt.

	7	6	5	4	3	2	1	0
R	0	RTR6	RTR5	RTR4	RTR3	RTR2	RTR1	RTR0
W	▨							
Reset	0	0	0	0	0	0	0	0

▨ = Unimplemented or Reserved

Figure 4-11. CRG RTI Control Register (RTICTL)

Read: anytime

Write: anytime

NOTE

A write to this register initializes the RTI counter.

Table 4-6. RTICTL Field Descriptions

Field	Description
6:4 RTR[6:4]	**Real-Time Interrupt Prescale Rate Select Bits** — These bits select the prescale rate for the RTI. See Table 4-7.
3:0 RTR[3:0]	**Real-Time Interrupt Modulus Counter Select Bits** — These bits select the modulus counter target value to provide additional granularity. Table 4-7 shows all possible divide values selectable by the RTICTL register. The source clock for the RTI is OSCCLK.

Table 4-7. RTI Frequency Divide Rates

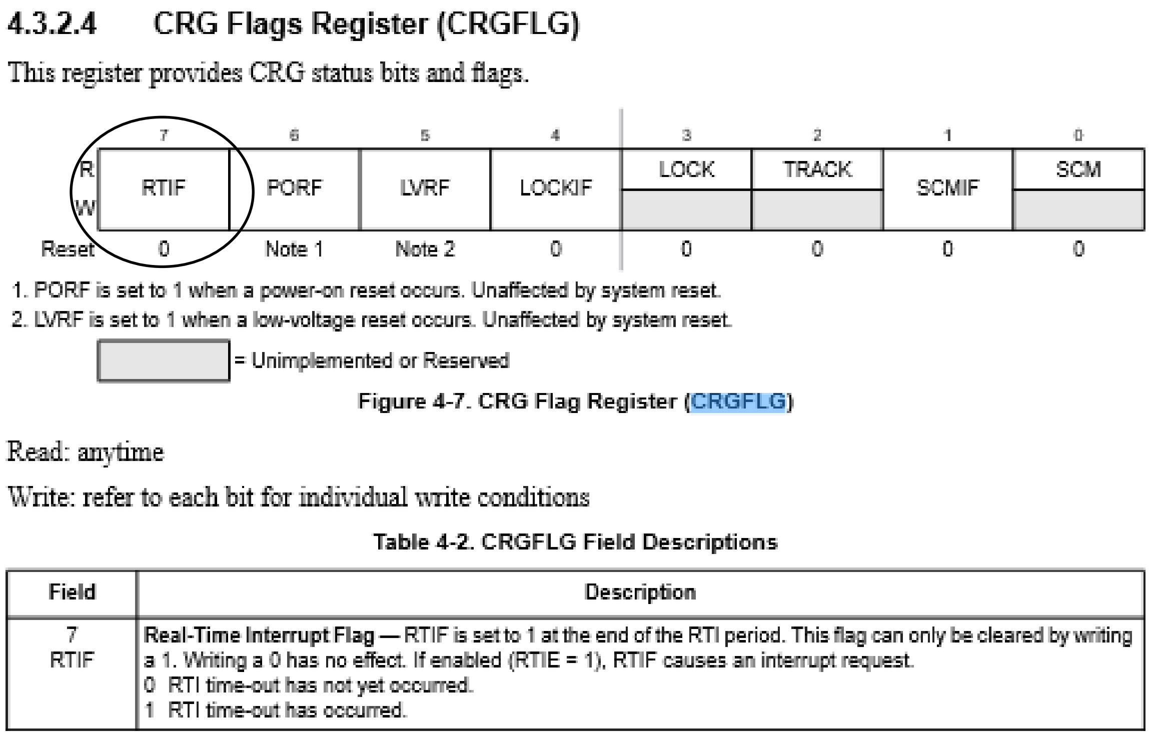
RTR[3:0]	**RTR[6:4] =**							
	000 (OFF)	**001 (2^{10})**	**010 (2^{11})**	**011 (2^{12})**	**100 (2^{13})**	**101 (2^{14})**	**110 (2^{15})**	**111 (2^{16})**
0000 (÷1)	OFF*	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}
0001 (÷2)	OFF*	$2x2^{10}$	$2x2^{11}$	$2x2^{12}$	$2x2^{13}$	$2x2^{14}$	$2x2^{15}$	$2x2^{16}$
0010 (÷3)	OFF*	$3x2^{10}$	$3x2^{11}$	$3x2^{12}$	$3x2^{13}$	$3x2^{14}$	$3x2^{15}$	$3x2^{16}$
0011 (÷4)	OFF*	$4x2^{10}$	$4x2^{11}$	$4x2^{12}$	$4x2^{13}$	$4x2^{14}$	$4x2^{15}$	$4x2^{16}$
0100 (÷5)	OFF*	$5x2^{10}$	$5x2^{11}$	$5x2^{12}$	$5x2^{13}$	$5x2^{14}$	$5x2^{15}$	$5x2^{16}$
0101 (÷6)	OFF*	$6x2^{10}$	$6x2^{11}$	$6x2^{12}$	$6x2^{13}$	$6x2^{14}$	$6x2^{15}$	$6x2^{16}$
0110 (÷7)	OFF*	$7x2^{10}$	$7x2^{11}$	$7x2^{12}$	$7x2^{13}$	$7x2^{14}$	$7x2^{15}$	$7x2^{16}$
0111 (÷8)	OFF*	$8x2^{10}$	$8x2^{11}$	$8x2^{12}$	$8x2^{13}$	$8x2^{14}$	$8x2^{15}$	$8x2^{16}$
1000 (÷9)	OFF*	$9x2^{10}$	$9x2^{11}$	$9x2^{12}$	$9x2^{13}$	$9x2^{14}$	$9x2^{15}$	$9x2^{16}$
1001 (÷10)	OFF*	$10x2^{10}$	$10x2^{11}$	$10x2^{12}$	$10x2^{13}$	$10x2^{14}$	$10x2^{15}$	$10x2^{16}$
1010 (÷11)	OFF*	$11x2^{10}$	$11x2^{11}$	$11x2^{12}$	$11x2^{13}$	$11x2^{14}$	$11x2^{15}$	$11x2^{16}$
1011 (÷12)	OFF*	$12x2^{10}$	$12x2^{11}$	$12x2^{12}$	$12x2^{13}$	$12x2^{14}$	$12x2^{15}$	$12x2^{16}$
1100 (÷ 13)	OFF*	$13x2^{10}$	$13x2^{11}$	$13x2^{12}$	$13x2^{13}$	$13x2^{14}$	$13x2^{15}$	$13x2^{16}$
1101 (÷14)	OFF*	$14x2^{10}$	$14x2^{11}$	$14x2^{12}$	$14x2^{13}$	$14x2^{14}$	$14x2^{15}$	$14x2^{16}$
1110 (÷15)	OFF*	$15x2^{10}$	$15x2^{11}$	$15x2^{12}$	$15x2^{13}$	$15x2^{14}$	$15x2^{15}$	$15x2^{16}$
1111 (÷ 16)	OFF*	$16x2^{10}$	$16x2^{11}$	$16x2^{12}$	$16x2^{13}$	$16x2^{14}$	$16x2^{15}$	$16x2^{16}$

* Denotes the default value out of reset.This value should be used to disable the RTI to ensure future backwards compatibility.

The above table can be used to choose a period for the RTI interval. For example, `$40` corresponds with the entry in the table above that is circled. If the value `$40` is written to the register RTICTL, then the RTI interval is given as:

$$T = \frac{Frequency\ Divide\ Rate}{OSCCLK} = \frac{2^{13}}{8MHz} = 1.024ms$$

Whenever an RTI event has occurred, the RTIF bit of the flag register (CRGFLG) at address `$37` is set. If this bit is not cleared at the end of the interrupt service routine, the program will not properly return to the main program. Please refer to the information from the datasheet below.



4.3.2.4 CRG Flags Register (CRGFLG)

This register provides CRG status bits and flags.

	7	6	5	4	3	2	1	0
R	RTIF	PORF	LVRF	LOCKIF	LOCK	TRACK	SCMIF	SCM
W					▨	▨		▨
Reset	0	Note 1	Note 2	0	0	0	0	0

1. PORF is set to 1 when a power-on reset occurs. Unaffected by system reset.
2. LVRF is set to 1 when a low-voltage reset occurs. Unaffected by system reset.

▨ = Unimplemented or Reserved

Figure 4-7. CRG Flag Register (CRGFLG)

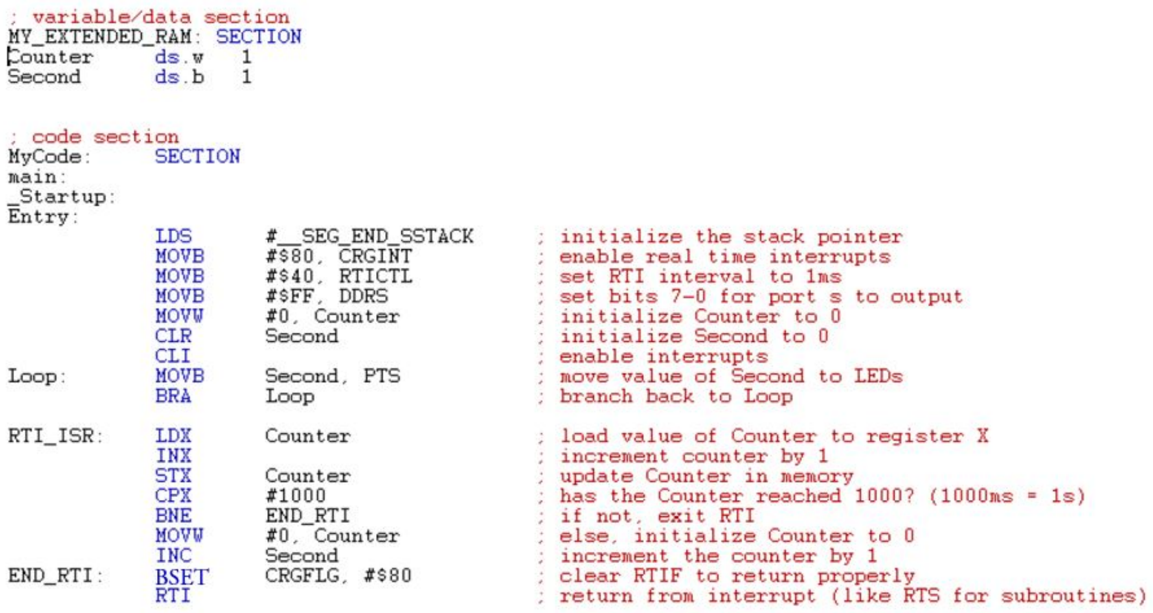
Read: anytime

Write: refer to each bit for individual write conditions

Table 4-2. CRGFLG Field Descriptions

Field	Description
7 RTIF	**Real-Time Interrupt Flag** — RTIF is set to 1 at the end of the RTI period. This flag can only be cleared by writing a 1. Writing a 0 has no effect. If enabled (RTIE = 1), RTIF causes an interrupt request. 0 RTI time-out has not yet occurred. 1 RTI time-out has occurred.

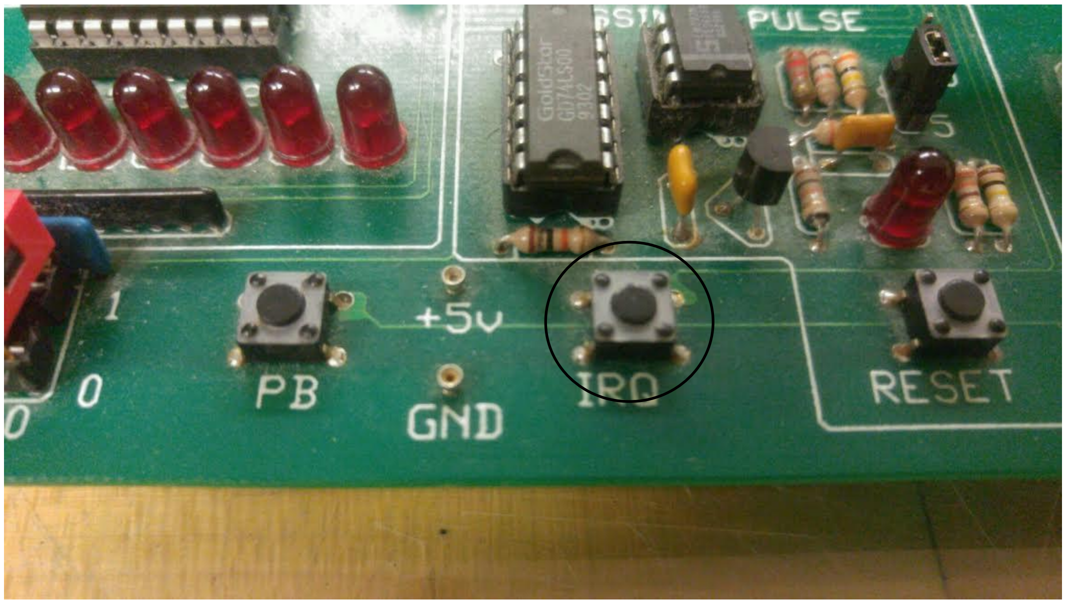
On the next page is an example of how to use real-time interrupts to count the seconds from 0-255 in binary on the LEDs.



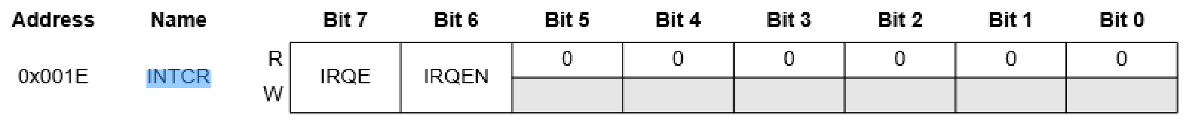
```
; variable/data section
MY_EXTENDED_RAM: SECTION
Counter      ds.w   1
Second       ds.b   1

; code section
MyCode:      SECTION
main:
_Startup:
Entry:
             LDS      #__SEG_END_SSTACK     ; initialize the stack pointer
             MOVB     #$80, CRGINT          ; enable real time interrupts
             MOVB     #$40, RTICTL          ; set RTI interval to 1ms
             MOVB     #$FF, DDRS            ; set bits 7-0 for port s to output
             MOVW     #0, Counter           ; initialize Counter to 0
             CLR      Second                ; initialize Second to 0
             CLI                            ; enable interrupts
Loop:        MOVB     Second, PTS           ; move value of Second to LEDs
             BRA      Loop                  ; branch back to Loop

RTI_ISR:     LDX      Counter               ; load value of Counter to register X
             INX                            ; increment counter by 1
             STX      Counter               ; update Counter in memory
             CPX      #1000                 ; has the Counter reached 1000? (1000ms = 1s)
             BNE      END_RTI               ; if not, exit RTI
             MOVW     #0, Counter           ; else, initialize Counter to 0
             INC      Second                ; increment the counter by 1
END_RTI:     BSET     CRGFLG, #$80          ; clear RTIF to return properly
             RTI                            ; return from interrupt (like RTS for subroutines)
```

Interrupt request line (IRQ):

In a microcontroller, an interrupt request is a hardware signal sent to the processor that temporarily stops a running program and allows an interrupt handler to run. Instead of the signal being generated internally by the OSCCLK, as with Real-time Interrupts, the signal is generated by an external source. For the equipment used in the lab, the IRQ pin is connected to a pushbutton on the I/O board. Please see the figure below.



0x001E – 0x001E MEBI Map 2 of 3 (HCS12 Multiplexed External Bus Interface)



Address	Name		Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0x001E	INTCR	R	IRQE	IRQEN	0	0	0	0	0	0
		W								

The register to enable the IRQ (INTCR) is located at memory location $1E. Bit 6 of this register enables the IRQ line, while bit 7 of this register sets the IRQ to be an edge triggered event instead of a level triggered event. To use the IRQ, write the value $C0 to this register. Although the IRQ will not be implemented in this lab, it will be implemented in the final project, so keep this information handy.

Experimental Procedure:

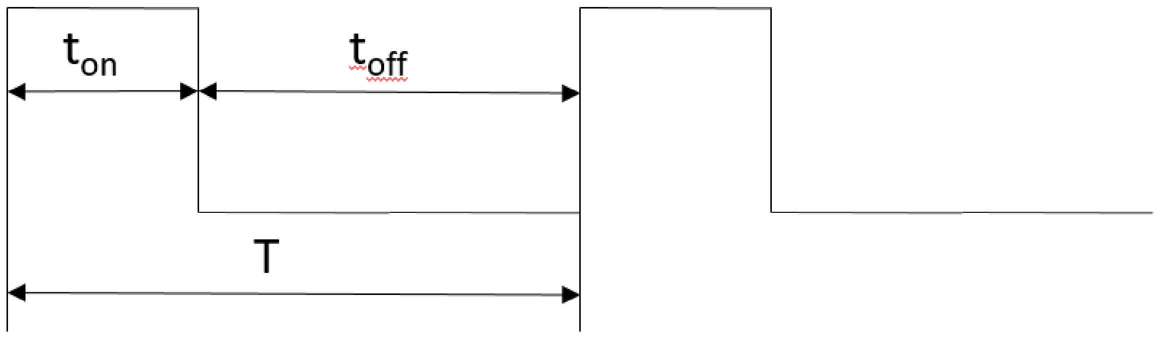
Laboratory 7.1: PWM, DC Motor Control using polling.

The objective of this experiment is to write an assembly program that implements Pulse Width Modulation (PWM) to control the speed of the DC motor on the I/O board in the lab. The DC motor is connected to bit 3 of Port T. Before beginning this experiment, make sure that jumper J2 is on the left two pins and that jumper J5 is on the upper two pins. Also, be sure to set bit 3 of Port T as an output by setting the appropriate bit of the Data Direction Register located at the address $242. Switch 4 must be up for the DC Motor to spin. To start the motor, set bit 3 of Port T at the address $240. To halt the motor, clear bit 3 of Port T.

In order to generate the PWM signal to control the motor, the keypad routine from lab 5 will be used. For lab 7.1, make the following changes to the keypad routine:

- If a match isn't returned in any of the four rows of the keypad, instead of branching back to search the rows again, branch down to the routine that handles the PWM signal to the DC motor. If this isn't done correctly the DC motor will only spin (briefly) when a key is pressed and released.
- Remove the section of code that debounces and waits for a key to be released. If this isn't done correctly, the DC motor will stop when the key is pressed, and then resume spinning when the key is released.

Also, in addition to the 1ms delay subroutine used for debouncing the keypad upon a keypress, also write a 4ms delay subroutine in order to implement the PWM strategy.



Above is an example of a Pulse Width Modulated waveform. Since the DC motor is only controlled by one bit, the only way to control the speed of the motor is by implementing a PWM strategy. This allows the average value of the power sent to the DC motor to change, based on the size of t_{on}. The average value of the above waveform is given by:

$$< v > = \frac{1}{T}\int_0^T v(t)\partial t = v_{dd}\frac{t_{on}}{T}$$

The period T should be sufficiently short. Choose a T = 60ms for the period. Divide this into 15 intervals to correspond with keys \$0-\$F on the keypad. The value read from the keypad will determine the duty cycle. The value t_{on} should be the value returned from the keypad routine. The value t_{off} should be 15 minus the value returned from the keypad routine. For example, if the keypad returns the value of \$C = 12, then $t_{on} = 12$ and $t_{off} = 15 – 12 = 3$.

Use the following instructions to successfully implement the PWM strategy described above.

- Call the altered keypad routine. If a key has not been pressed, use the previous value for t_{on} and t_{off}.
- Check if t_{on} is equal to zero, if it is, branch down to check t_{off}.
- Else, set bit 3 of Port T, delay for 4ms, decrement t_{on}, and check again.
- Check if t_{off} is equal to zero, if it is, branch back to the keypad routine.
- Else, clear bit 3 of Port T, delay for 4ms, decrement t_{off}, and check again.

Laboratory 7.2: PWM, DC Motor Control using Real Time Interrupts.

The objective of this experiment is to write an assembly program that implements Pulse Width Modulation (PWM) to control the speed of the DC motor on the I/O board in the lab. Instead of using delay loops and polling to implement the PWM strategy, Real Time Interrupts will be used. In the main program and the linker parameter file, initialize interrupts by using the information in the background section above. Select an RTI interval of 4ms.

The unaltered keypad subroutine from lab 5.2 should be called in the main program. It is okay to calculate t_{on} and t_{off} in the main program, but **delay loops are not to be used to control the PWM signal.**

The simplest way to complete this lab is to define a byte variable to use as a counter and initialize it to zero. In the interrupt routine, do the following:

- Increment the count. If the count is less than or equal to t_{on}, set bit 3 of Port T and exit the interrupt service routine.
- Else, if the count is greater than t_{on}, but less than or equal to 15, clear bit 3 of Port T and exit the interrupt service routine.
- Else, if the count is greater than 15, clear the count and exit the interrupt service routine.

Laboratory 7.3: Wall Clock

The objective of this experiment is to implement the wall clock code found in the lecture and in the background section above to display the seconds on the LEDs. Count from 0-60 instead of from 0-255. Also, instead of counting the seconds in Binary, count the seconds in BCD. Refer to the table below.

Decimal	Binary	BCD
0	0000 0000	0000 0000
1	0000 0001	0000 0001
2	0000 0010	0000 0010
3	0000 0011	0000 0011
4	0000 0100	0000 0100
5	0000 0101	0000 0101
6	0000 0110	0000 0110
7	0000 0111	0000 0111
8	0000 1000	0000 1000
9	0000 1001	0000 1001
10	0000 1010	0001 0000
11	0000 1011	0001 0001
12	0000 1100	0001 0010
13	0000 1101	0001 0011
14	0000 1110	0001 0100
15	0000 1111	0001 0101

Extra Credit: Display the time on the LCD instead of on the LEDs. You'll have to include the necessary files from lab 6.3 in order to accomplish this.