LAB 4: INTERRUPT PROCESSING IN C

INTRODUCTION

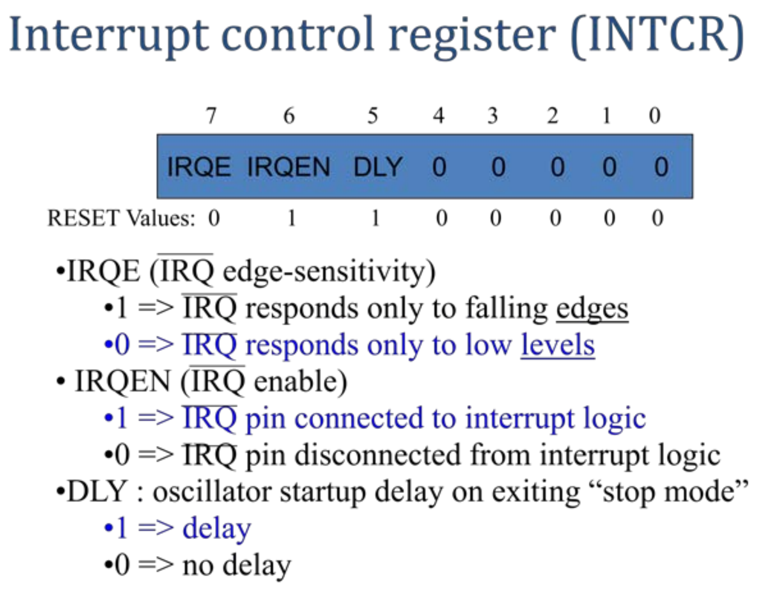
The previous labs worked with simple input/output (I/O) devices using ***program-controlled I/O***; the programs continuously monitored each "device" to determine when to take action. The primary disadvantage of using this approach to handle I/O devices is that the CPU spends considerable amounts of time scanning devices to see if they require attention, whether or not there is any device activity. The purpose of this lab is to learn how to design C programs for the MC9S12C32 (HCS12) microcontroller to handle devices in ***interrupt-driven I/O*** mode. This allows the CPU to perform tasks other than monitoring the devices, since devices will signal the CPU via interrupt request signals when they require attention. Interrupt-driven I/O is used heavily in embedded system applications, with interrupt signals coming from external devices, data acquisition hardware, timers, etc. For this introductory lab, we will simulate the operation of two external devices by using switches to issue interrupt requests. The application program will comprise a main task, which runs continuously, and two interrupt service routines, which are executed when interrupt requests are detected.

HCS12 INTERRUPTS (Review of ELEC 2220 Material)

External Interrupt Pins and Interrupt Masks

The **HCS12** CPU has a number of built-in functions that can issue interrupt request signals, including timers, analog to digital converter, communication modules, etc. In addition to these internally-generated signals, the **HCS12** has two input pins that can be used by external devices to signal interrupt requests: ***IRQ#*** (Interrupt Request on pin PE1) and ***XIRQ#*** (Non-Maskable Interrupt – on pin PE0); refer to the microcontroller pin assignments in the Lab 2 writeup.

IRQ# is an active-low signal that can be configured, via bit IRQE (bit 7) of the IRQ Control Register, INTCR (shown in Figure 1), as either a level-sensitive or edge-sensitive input. If configured as level-sensitive (IRQE=0), the CPU is interrupted if and only if the line is low at the time the CPU samples it (the CPU samples all interrupt request signals at the conclusion of each executed instruction). Therefore, ***IRQ#*** must (1) be driven low to request an interrupt, (2) remain low until the CPU acknowledges the interrupt request, and then (3) return high to prevent triggering additional interrupts. Multiple interrupt sources may be connected to ***IRQ#***, so that if two devices request interrupts simultaneously, one device may continue to hold the line low while the other is being serviced, and thus interrupt the CPU when it finishes with the first device. If ***IRQ#*** is configured as edge-sensitive (IRQE=1), then a falling edge (high-to-low transition) on the pin sets an internal flip flop, which remains set until the CPU responds. Therefore, one need not be concerned about the CPU missing the request. Since the internal flip-flop is edge triggered, the device must return ***IRQ#*** to its high state and then produce another falling edge to trigger another interrupt. Note that bit 6 (IRQEN) of the INTCR register must be 1 to enable the ***IRQ#*** pin to be used. This is the default value coming out of reset, and should not be reset to 0 if you plan to use ***IRQ#***.



Interrupt control register (INTCR)

	7	6	5	4	3	2	1	0
	IRQE	IRQEN	DLY	0	0	0	0	0
RESET Values:	0	1	1	0	0	0	0	0

- IRQE ($\overline{\text{IRQ}}$ edge-sensitivity)
  - 1 => $\overline{\text{IRQ}}$ responds only to falling edges
  - 0 => $\overline{\text{IRQ}}$ responds only to low levels
- IRQEN ($\overline{\text{IRQ}}$ enable)
  - 1 => $\overline{\text{IRQ}}$ pin connected to interrupt logic
  - 0 => $\overline{\text{IRQ}}$ pin disconnected from interrupt logic
- DLY : oscillator startup delay on exiting “stop mode”
  - 1 => delay
  - 0 => no delay

Figure 1. HCS12 Interrupt Control Register INTCR

The ***IRQ#*** pin, as is the case for most other interrupt sources in the **HCS12**, is “maskable” via the “Interrupt Mask” (I flag), which is bit 4 of the CPU’s condition code register (CCR), shown in Figure 2. All maskable interrupt signals, including ***IRQ#***, are logically ANDed with the inverted I flag, as shown in Figure 2. When I=1, the AND gate output is forced to 0, thus “masking” the ***IRQ#*** value, effectively disabling the interrupt request. When I=0, the ***IRQ#*** value appears at the AND gate output, effectively enabling/unmasking the interrupt request. The I flag is always 1 when the CPU powers on or resets, preventing devices from interrupting the program before variables, devices, etc. can be initialized. The I flag must be reset to 0 before the CPU can begin processing interrupt requests. In assembly language, the instruction *CLI* (clear interrupt flag), which assembles to *ANDCC #%11101111*, clears the I flag (I=0) to enable interrupts, and *SEI* (set interrupt flag), which assembles to *ORCC #%00010000*, sets the I flag (I=1) to disable interrupts. To set/clear the I flag in C programs, include the machine-dependent definitions header file “hidef.h” in your program, which defines the C statements ***EnableInterrupts*** and ***DisableInterrupts***.

S	X	H	I	N	Z	V	C

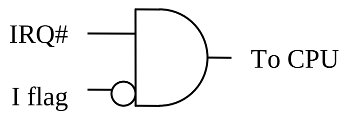


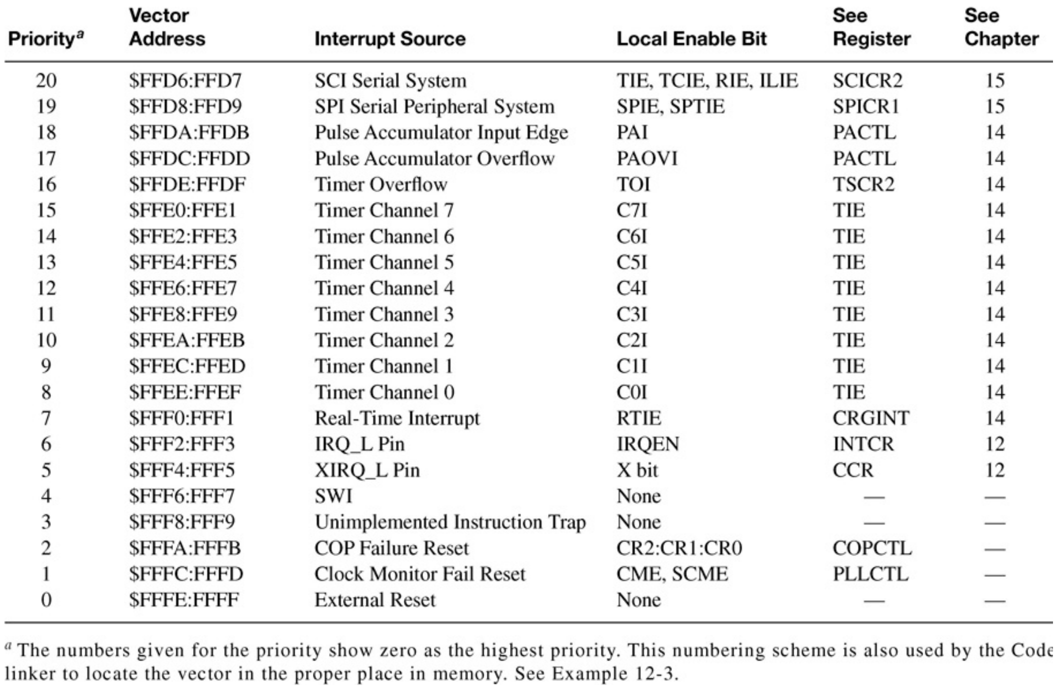
Figure 2. Condition Code Register (CCR)

The X flag of the CCR enables the non-maskable interrupt pin ***XIRQ#***. Like the I flag, X is initially set to 1 when the CPU comes out of reset. When the software is ready to enable ***XIRQ#*** interrupt requests, X must be cleared to 0 with the instruction *ANDCC*

#%10111111. There is no corresponding C language function, so an "in-line assembly" instruction must be used: ***__asm ANDCC #0xBF***. Note that ***__asm*** has <u>two</u> underlines. Unlike the I flag, once X has been cleared, it cannot be set to 1 again. Therefore, once ***XIRQ#*** interrupts have been enabled, they cannot be disabled, effectively making ***XIRQ#*** "non-maskable". The software interrupt instruction (SWI) and the unimplemented instruction trap are the other two interrupt sources masked by the X flag. External reset, clock monitor failure reset, and CPU failure reset are the three non-maskable interrupts that are not masked by any flags, and therefore cannot be disabled.

<u>Interrupt Vectors and Priorities</u>

The CPU responds to an interrupt request by suspending its current task and executing a program called an "interrupt service routine", which is unique for each device in the system. The starting address of an interrupt service routine is known as its "interrupt vector". Each HCS12 interrupt vector must be stored in an interrupt vector table (IVT) at a designated location in memory, with the IVT ending at address $FFFF. A partial IVT is shown in Table 1. The vector for each internal and external interrupt source must be stored by the assembler/compiler in the IVT at the location designated in the CPU reference manual, from which Table 1 was extracted. For your convenience, a C source file ***isr_vectors.c***, described under "Software Design" below, can be downloaded from the course web page and included in your project to set up this table. Vectors are stored in the table in decreasing order of interrupt priority, with RESET being the highest priority interrupt (priority 0). Thus, if **IRQ#** and **XIRQ#** activate concurrently, **XIRQ#** would be serviced first, having the higher priority.



Priority[a]	Vector Address	Interrupt Source	Local Enable Bit	See Register	See Chapter
20	$FFD6:FFD7	SCI Serial System	TIE, TCIE, RIE, ILIE	SCICR2	15
19	$FFD8:FFD9	SPI Serial Peripheral System	SPIE, SPTIE	SPICR1	15
18	$FFDA:FFDB	Pulse Accumulator Input Edge	PAI	PACTL	14
17	$FFDC:FFDD	Pulse Accumulator Overflow	PAOVI	PACTL	14
16	$FFDE:FFDF	Timer Overflow	TOI	TSCR2	14
15	$FFE0:FFE1	Timer Channel 7	C7I	TIE	14
14	$FFE2:FFE3	Timer Channel 6	C6I	TIE	14
13	$FFE4:FFE5	Timer Channel 5	C5I	TIE	14
12	$FFE6:FFE7	Timer Channel 4	C4I	TIE	14
11	$FFE8:FFE9	Timer Channel 3	C3I	TIE	14
10	$FFEA:FFEB	Timer Channel 2	C2I	TIE	14
9	$FFEC:FFED	Timer Channel 1	C1I	TIE	14
8	$FFEE:FFEF	Timer Channel 0	C0I	TIE	14
7	$FFF0:FFF1	Real-Time Interrupt	RTIE	CRGINT	14
6	$FFF2:FFF3	IRQ_L Pin	IRQEN	INTCR	12
5	$FFF4:FFF5	XIRQ_L Pin	X bit	CCR	12
4	$FFF6:FFF7	SWI	None	—	—
3	$FFF8:FFF9	Unimplemented Instruction Trap	None	—	—
2	$FFFA:FFFB	COP Failure Reset	CR2:CR1:CR0	COPCTL	—
1	$FFFC:FFFD	Clock Monitor Fail Reset	CME, SCME	PLLCTL	—
0	$FFFE:FFFF	External Reset	None	—	—

[a] The numbers given for the priority show zero as the highest priority. This numbering scheme is also used by the Code linker to locate the vector in the proper place in memory. See Example 12-3.

Table 1. Partial interrupt vector table *(Cady text, Chap. 12, pg 296)*

INTERRUPT SUPPORT IN C PROGRAMS

In designing a C program, three key elements must be provided to process interrupt requests from a device: an interrupt service routine must be written, the starting address of the service routine must be placed in the interrupt vector table, and CPU interrupts must be enabled by clearing the appropriate mask in the condition code register. In addition, if the IRQ# pin is to be used in other than the default operating mode, the interrupt control register, INTCR, must be configured. Other built-in functions of the microcontroller may have additional interrupt setup requirements.

The sample program below illustrates a few conventions in CodeWarrior for setting up a C program to process interrupt requests. An interrupt service routine looks like any other function, except that the keyword ***interrupt*** precedes the function return value type (*void* in the example). There are no special requirements for the function name; ideally the name should reflect the nature of the function performed by that interrupt routine.

```
#include <hidef.h>         /* common defines and macros */
#include <mc9s12c32.h>     /* derivative information */

/* global variables definitions here */

/* Define the interrupt service routine */
interrupt void IRQ_ISR(void) {
      ... instructions for the interrupt service
}

/* Define the main program */
void main(void) {
  ... initialize variables/devices/registers

  INTCR_IRQEN = 1;    /* enable IRQ# interrupts */
  INTCR_IRQE = 1;     /* IRQ# interrupts edge-triggered */
  EnableInterrupts;   /* clear I mask to enable interrupts */
  __asm ANDCC #0xBF   /* clear X mast to enable XIRQ# */

  ... instructions for the main program
}
```

Interrupts are typically set up and enabled in the main program, as shown in the example. Note that interrupts are generally enabled only after all variables and devices have been initialized and the program is ready to begin responding to interrupt requests.

To set up the interrupt vector table, you should include in your project the source file ***isr_vectors.c***, shown (partially) on the next page. At the top of this file are declarations of any interrupt routines defined in other files (such as main.c). This information is needed by the linker. The designation "near" tells the linker that the addresses of these routines are within the 64K memory address space. Next is defined a special interrupt service routine named "UnimplementedISR". The purpose of this is to provide a "clean response" to any interrupt request for which an interrupt routine has not otherwise been defined, rather than allowing the program to simply "crash" if that interrupt signal activates.

```
/**** isr_vectors.c ******/
extern void near _Startup(void);       /* Startup routine */
extern void near IRQ_ISR(void);        /* My ISR from main.c */

#pragma CODE_SEG __NEAR_SEG NON_BANKED /* Interrupt section for
this module. Placement will be in NON_BANKED area. */
__interrupt void UnimplementedISR(void)
{
   /* Unimplemented ISRs trap.*/
   asm BGND;
}

typedef void (*near tIsrFunc)(void);
const tIsrFunc _vect[] @0xFF80 = {     /* Interrupt table */
        UnimplementedISR,                 /* vector 63 */
        UnimplementedISR,                 /* vector 62 */
        UnimplementedISR,                 /* vector 61 */
        UnimplementedISR,                 /* vector 60 */
        UnimplementedISR,                 /* vector 59 */
        UnimplementedISR,                 /* vector 58 */
            ...
        UnimplementedISR,                 /* vector 08 */
        UnimplementedISR,                 /* vector 07 */
        IRQ_ISR,                          /* vector 06 */
        UnimplementedISR,                 /* vector 05 */
        UnimplementedISR,                 /* vector 04 */
        UnimplementedISR,                 /* vector 03 */
        UnimplementedISR,                 /* vector 02 */
        UnimplementedISR,                 /* vector 01 */
   //     _Startup                        /* Reset vector 00 */
   };
```

The remainder of ***isr_vectors.c*** sets up the interrupt vector table, beginning at memory address 0xFF80 and ending at 0xFFFF. Each entry in the vector table will initially be the vector to UnimplementedISR, so that if an unexpected interrupt occurs, it will be trapped and reported to you by the BDM debugger. You must replace "UnimplementedISR" with the names of your interrupt service routines in the appropriate vector positions in this table. The vector numbers listed in the comment fields correspond to the interrupt priority numbers in Table 1 above, and in the HCS12 reference manual. For example, XIRQ# and IRQ# are assigned vectors 5 and 6, respectively.

Note that "__Startup" (the reset vector) has been commented out in the above example, since that vector is separately defined in the startup code that is included in every new project.

LAB EXPERIMENT

For this lab, the "main program" is to continuously display counting sequences on two 4-bit (4 LED) displays, with the first sequence changing at twice the rate of the second. The first sequence will always be increasing (0 to 9 and repeat). The second sequence will initially be increasing. If an interrupt occurs on pin XIRQ#, the sequence is to change from increasing to decreasing, and if an interrupt occurs on pin IRQ#, the sequence is to change from decreasing to increasing. Switches will be used to activate the interrupt requests.

PRE-LAB ASSIGNMENT

Reading

Review Chapter 12 of the Cady text. Section 12.1 provides a general introduction to interrupts; sections 12.2 and 12.3 describe HCS12 interrupts; sections 12.4 and 12.5 discuss HCS12 interrupt vectors and interrupt priorities; section 12.7 discusses external interrupt sources (IRQ#, XIRQ#), and section 12.10 discusses interrupt service routines.

Hardware Design

This, and all future labs, will utilize the "DragonFly12-Plus" 40-pin DIP module and the EEBOARD platform. Refer to the previous lab documents for pin-out and schematic.

The I/O port connections to the "peripheral devices" should be made as listed in Table 2. Parallel input/output ports T and AD are to be used to display separate counting patterns on two sets of 4 LEDs (in the *Waveforms* Static I/O window). Two push-button switches, S1 and S2 (also in the *Waveforms* Static I/O window), are to be connected to the XIRQ# and IRQ# interrupt pins, respectively, which are available via Port E. You may select which ever EEBOARD Digital I/O lines and *Waveforms* Static I/O bits you wish.

Parallel Port Pins	Connected Devices
Port T, bits 0-3	1st 4-bit LED display
Port AD, bits 0-3	2nd 4-bit LED display
Port E, bit 0	Switch S1 - XIRQ#
Port E, bit 1	Switch S2 – IRQ#

Table 2. Parallel input/output port connections.

To demonstrate the operation of the program, the logic analyzer and oscilloscope are to be used to capture and show counting patterns as interrupts occur.

Software Design

Review the earlier labs to recall how to initialize and access I/O ports in C, and review the above information and pre-lab reading about interrupt processing. Remember to thoroughly comment your program.

This program is to comprise a main routine and two interrupt service routines.

1. The main program should execute in a continuous loop, displaying separate counting sequences on two 7-segment display digits. The first count is to be increasing (0 to 9 and repeat) on the four LEDs connected to Port T. This count should change approximately once per half second, and continue throughout the duration of the program.
2. The second count is to be displayed on the four LEDs connected to Port AD, and is to change approximately once per second. The second count is to initially be increasing.
3. If the XIRQ# interrupt pin is activated (via switch S1), the count sequence should change from increasing to decreasing (or continue to be decreasing if that is the current sequence.)
4. If the IRQ# interrupt pin is activated (via switch S2), the count sequence should change to increasing (or continue to be increasing if that is the current sequence.)

One possible way to implement this is to have the interrupt service routines set or reset a global variable that determines the direction the second counter, with all counting handled by the main program or a separate counting function. The time delay can either be implemented within the main program, or called as a separate function.

LAB PROCEDURE

1. If not already done, mount the DragonFly12-Plus module on your breadboard.
2. Connect the power supply pins (+5v and ground) on the module to the corresponding pins on the EEBOARD.
3. To the EEBOARD Digital I/O block, connect the Port T pins for one 4-LED display, and Port AD pins for the second 4-LED display.
4. Using EEBOARD Digital I/O signals, connect “switch” S1 to XIRQ# (pin PE0) and “switch” S2 to IRQ# (pin PE1). These switches should be initially in the logic 1 (inactive) state.
5. After double-checking connections, turn on the EEBOARD power supply via the *Waveforms* Voltage window and the power switch on the board.
6. Enter, compile and download your program. Execute your program, verifying that the correct counts are displayed on the two digits, and that the counting sequence is correctly altered by the two interrupts.

7. Set up the logic analyzer to capture and display the counting sequences on the two 4-LED displays, plus the states of the two interrupt signals. (Creating two 4-bit "buses" in the logic analyzer window would facilitate watching the counting sequences.) *Note that you can use the same Digital I/O connections for both logic analyzer and static I/O windows.* Try to use the interrupt signals to "trigger" the logic analyzer so that you can verify that the counting sequence changes correctly as each interrupt activates.

8. Demonstrate your working programs to the lab instructor.

LAB REPORT INFORMATION

1. Briefly describe your hardware design. Attach a schematic diagram.
2. Attach a printout of your C program, including well thought through comments.
3. Discuss your results. Attach a logic analyzer screen capture showing the count changing direction in response to an interrupt signal.