Interrupt and Timer ISRs

Student's name & ID (1): _____

Partner's name & ID (2): _____

Your Section number & TA's name _____

Notes:

You must work on this assignment with your partner.

Hand in a printer copy of your software listings for the team.
Hand in a neat copy of your circuit schematics for the team.

These will be returned to you so that they may be used for reference.

---------- do not write below this line ----------

	POINTS (1)	POINTS (2)	TA init.
Grade for performance verification (50% max.)			
Grade for answers to TA's questions (20% max.)			
Grade for documentation and appearance (30% max.)			
			TOTAL

Grader's signature: _____

Date: _____

Interrupt and Timer ISRs

GOAL

By doing this lab assignment, you will learn:

1. To use IRQ and on-chip Timer interrupts on the M68HC12.
2. To add interrupt service routines to a C program using the Introl C cross-compiler.

PREPARATION

- Read Sections 8.1 to 8.8 from *Software and Hardware Engineering* by Cady & Sibigtroth. (Read Sections 6.9-10 and 7.7-9 from *Microcomputer Engineering* by Gene H. Miller.)
- Read Sections 10.1 to 10.8 from *Software and Hardware Engineering* by Cady & Sibigtroth.
- Read Section 7 in *CPU12 Reference Manual (M68HC12).*
- References: *Software and Hardware Engineering*, pp. 192 & 194

Software Design

- Do a top-down design of your program. *Provide a flowchart or equivalent.*
- Write C programs for each module/subroutine. Recall that interrupt service routines are *not* subroutines. *Provide planning documentation for both flowchart and tests. Specify test inputs and expected output or other indictors of correct operation.*
- After all the submodules are operating correctly, they may be integrated to form a higher level module. Integrate your modules to get an assembler program that is free from syntax errors (i.e., it should assemble without error messages and produce a .S19 file).

PROGRAMMING TASKS FOR THE MC6812

1. Write a simple C program that responds to an IRQ interrupt (Grounding J8 pin 44 IRQ). The interrupt handler should set a global variable that can be seen by the main routine. The main routine should handle any I/O, i.e. output to the terminal, related to the generated interrupt.
2. Write two simple C programs that respond to timer interrupts to display elapsed time in tenths of a second using two different operational modes of the programmable MC6812 timers. One of the modes may approximate 0.1 sec by rounding but the other must be exact to the accuracy of the processor's clock frequency. Responses through I/O to the console may be off by milliseconds due to serial line delays, but the internal interrupts should be generated every 0.1 seconds. Be prepared to explain the differences in operation between the two programs.

3. Combine the C programs from 1 and the exact 0.1 sec timer in 2 to test reaction time. Flash the screen after a random delay and measure the time it takes for the subject to react and ground the IRQ pin. (Use a pull-up resistor to avoid false interrupts.) Compute the average of the number of trials or a moving average of the last 5 trials and display it on the screen. Provide a means to reset or clear the program or display, the details of how it's done are left up to you. Optional features you may choose to add could penalize the operator for reacting *before* the signal from the program.

Good programmer's tip: Design the program top-down. Then write the routines bottom-up. Write them one at a time and thoroughly test each one before integrating them. This way you will have isolated any errors to the routine that you are currently writing. Good programmers follow this method.

INTRODUCTION TO INTERRUPTS - H/W BIT SETTINGS

An interrupt in a microprocessor is a request for service by a process (a background process) that is running outside the direct control of the program. That is, the process is not a subroutine or other subprogram that is called by the program at a particular point in its execution. For example, a keyboard input is not controlled by the user's program but by someone pressing on the keys. In real-time applications, interrupts are often used to perform periodic functions such as to sample data or to update the error input to a servo loop. Other functions are to measure the elapsed time from a reference starting time or to measure the time interval between external events. In this laboratory exercise, only time-based interrupts are used.

For a (background) timer process to request service, two preconditions must be present: the global interrupt mask bit I in the Condition Code Register, CCR, must be cleared (I = 0) and the local interrupt enable bit must be set.

Global Interrupt Mask

All the time-based interrupts are controller by the global interrupt mask bit I in the Condition Code Register. When I = 1, it masks, i.e., prevents, interrupts. So, for a time-based interrupt to happen, I must equal 0. This is done by executing the `CLI`, CLear I mask, instruction to enable interrupts. Note, the instruction `SEI`, Set I mask, disables interrupts.

Local Interrupt Enable Bit

Each time-based interrupt has its own interrupt enable bit. For example, the Free Running Timer has the TOI, Timer Overflow Interrupt, bit (bit 7 in the TMSK2 register at $008D). The Timer Overflow Interrupt is enabled by writing a 1 to the TOI bit.

Interrupt Request Flag

For an interrupt request to be made, the interrupting process must set its flag. When the Free Running Timer rolls over from $FFFF to $0000, a Timer Overflow occurs. This is indicated by setting the TOF, Timer Overflow Flag, bit (bit 7 in the TFLG2 register at $008F). If both preconditions are met, an interrupt request is made to the microprocessor.

M68HC12 Interrupt Request Response

The M68HC12 looks for interrupt requests after an instruction is completed and before the next instruction is fetched from memory. When it receives an interrupt request, the first action is to save the status of all the working registers on the stack. That is, 9 bytes are pushed on the stack in the following order: Program Counter Lo-byte (PCL), PCH, Index Register Y Lo-byte (IYL), IYH, IXL, IXH, ACCA, ACCB, CCR. Second, the I bit is set to prevent nested interrupts. Third, the highest priority interrupt request is identified and the 2-byte Interrupt Vector Address in the table located between $FFCE and $FFFF is put into the Program Counter. This is the address of the first instruction of the Interrupt Service Routine for that interrupt.

For an IRQ interrupt, its Vector Address is located at $FFF2, $FFF3. The first instruction of the Interrupt Service Routine for the IRQ Interrupt is at the Vector Address. In the D-Bug12 monitor ROM, $FFF2, $FFF3 contains the zero page address for interrupt vector 25, the number assigned to the IRQ interrupt (see pp. 192 - 194 in Cady & Sibigtroth and the last pages of the Introl C Cross Compiler manual). The name of your ISR is defined in your program by the Introl compiler through the function DB12->SetUserVector(int_number, ISR_name). To assign the name of your routine IRQInt to the IRQ interrupt (number 25), you must execute at the beginning of your program the statements:

```
DB12->SetUserVector(IRQ, IRQInt);
```

Before interrupts are enabled by the program, the user must create an ISR with the name IRQInt and program the instructions to be executed by the interrupt service routines for an IRQ interrupt.

Interrupt Service Routine - assembly language details

The first thing an interrupt service routine usually does is to check that the associated interrupt request flag is set. Since the IRQ interrupt is shared with the port D Key Wakeups, the processor must determine what to do. Bit 6 of the IRQ Interrupt Control Register (INTCR, $001E) must be 1 for interrupts to be enabled. If port D Key Wakeups were also used in your program, that register would need to be checked to make sure it didn't generated the interrupt. Since Key Wakeups are not being used here, it is not necessary to look elsewhere. The next thing that is done is to clear the interrupt flag so an immediate false interrupt is not seen when returning from the interrupt service routine. If external circuits were built to catch the IRQ interrupts, instructions may need to be executed that interact with the hardware to clear the request. Notice that the `RTI` instruction pulls 9 bytes off the stack. This way the CCR is

restored to the pre-interrupt condition and interrupts are again enabled. Since the Program Counter now contains the address of the next instruction in the program following the instruction that was completed when the interrupt request was serviced, the program continues as though it was not interrupted except for artifacts produced by the interrupt service routine.

INTRODUCTION TO TIMERS

The M68HC12 has three programmable timer systems: a Free-Running Counter, a Real-Time Clock, and a Pulse Accumulator Timer.

Free-Running Counter

The Free-Running Counter starts running when power is first applied to the M68HC12. It runs continuously as long as the power is on. That is why it is called "free-running." The value of the counter is in the double register TCNT ($0084, $0085). TCNT is incremented with every tick of the counter. The counter has a programmable Prescaler that allows the clock ticks to be scaled relative to the M-clock. The Prescaler bits PR2, PR1, PR0 (bits 2, 1 and 0 in TMSK2) can be changed any time. The Free-Running Counter always runs at the M-clock frequency (typically either 4 or 8 MHz) divided by 1, 2, 4, 8, 16, or 32, depending on the prescaler value.

The TOI is enabled by writing a 1 to TOI (bit 7 in TMSK2). With the statement:

```
DB12->SetUserVector(TimerOvf, TimerOvfInt);
```

the zero page ISR name for the Free-Running Timer Overflow Interrupt is TimerOvfInt(). The TOF is cleared by writing a 1 to TOF (bit 7 in TFLG2).

Related to the Free-Running Counter are the eight Output Compare functions. On each M-clock cycle, the 16-bit value of TCNT is compared to the contents of the 16-bit TCx registers, where x = 0, …, 7. When a match is found, the CxF in TFLG1 is set. If CxI in TMSK1 is enabled, a local interrupt request is made. If the global I mask bit in the CCR is clear, the interrupt request is handled by the microprocessor.

The output compare function can be used to obtain precise periodic interrupts. For example, if interrupts spaced 1 ms apart are desired, add the current 16-bit value of the TCNT to 2000_{10} (= $07D0) and store the sum in the double register TCx. Each time the interrupt occurs, add $07D0 to the old value in TCx so the next interrupt occurs 2000 ticks later. When the sum rolls over (overflows) there is no problem since that is exactly what TCNT does! You can choose other intervals up to 32.77 ms. The smaller the interval, the better the precision when the interrupts are counted to measure 1 s, 1 min, or 1 hr intervals.

Interrupt Service Routine - assembly language details

Again the interrupt service routine first checks that the associated interrupt request flag is set. *All timer flags are cleared by writing a 1 to the flag bit.* (Writing a 0 to a flag bit that is set does not affect the bit.) Normally this is done using a `BRCLR 0,X mm rr` instruction. For the Timer Overflow Flag, this is `BRCLR 0,X BIT7 RTN_ISR_TOV` where X contains $008F. The next thing that is done is to clear the interrupt flag so an immediate false interrupt is no seen when returning from the interrupt service routine. For all timer interrupts, the flag is cleared by writing a 1 to the flag bit position in its flag register. This is usually done by using a `BCLR 0,X mm` instruction. For the Timer Overflow Flag, this is `BCLR 0,X $7F` where X contains $008F. Finally the `RTI` instruction pulls 9 bytes off the stack and the CCR is restored to the pre-interrupt condition and interrupts are again enabled. Since Program Counter contains the address of the next instruction in the program following the instruction that was completed when the interrupt request was serviced, the program continues as though it was not interrupted except for artifacts produced by the interrupt service routine.

Real-Time Clock

The Real-Time Clock is used to generate periodic interrupts. Real time interrupts can be set to one of four rates that are derived from the M-clock. The RTI Rate Control bits RTR2, RTR1, RTR0 (bits 2, 1 and 0 in RTICTL) may be written at any time. The RTI rate may be changed within a program as necessary.

Real-time interrupts are governed by the I bit in the CCR. The local RTI enable bit is RTIE (bit 7 in RTICTL). The RTI flag is RTIF (bit 7 in RTIFLG). It is set at the end of each periodic interval as determined by the RTI Rate Control bits.

With the statement:

```
DB12->SetUserVector(RTI, RTIInt);
```

The zero page ISR name for the Real-Time Interrupt is RTIInt(). The RTIF is cleared by writing a 1 to RTIF (bit 7 in RTIFLG).

Pulse Accumulator Timer

The Pulse Accumulator can be used as a timer. This mode is called the Gated Time Accumulation mode. The PA timer increments every 8 μs (64/M-clock frequency). This corresponds to an overflow every 0.524288 ms (8 μs·65536). The PA timer is gated, i.e., turned on and off, by the logic level on the input pin PA7. The Pulse Accumulator Count register (PACNT at $00A2:$00A3) is an 16-bit register. Note that it is both readable and *writable* so that PACNT can be initialized to values other than $00.

	7	6	5	4	3	2	1	0	
$00A0		PAEN	PAMOD	PEDGE	CLK1	CLK0	PAOVI	PAI	PACTL
RESET =	0	0	0	0	0	0	0	0	

Setting PAEN enables the Pulse Accumulator system. Setting PAMOD selects the gated time accumulation mode. To enable timer counting when PT7 is high and to set the PAIF flag when PT7 falls, set PEDGE = 0. To enable the timer counting when PT7 is low and to set the PAIF flag when PT7 rises, set PEDGE = 1. As shown in Figure 10-5 in the text, PT7 gates the $\frac{\text{M - CLOCK}}{64}$ pulses to the counter in either case. PEDGE is used to pick whether a high or low value on PT7 enables counting, and the end of the pulse will always set the PAIF flag.

The Pulse Accumulator is governed by the I bit in the CCR. The local PA timer enable bit is PAOVI (bit 1 in PACTL). The PA timer overflow flag is PAOVF (bit 1 in PAFLG). It is set each time PACNT rolls over from $FFFF to $0000, i.e., overflows. The PAOVF is cleared by writing a 1 to PAOVF (bit 1 in PAFLG).

```
// IntrptEx.c
//
// Motorola 68HC12 IRQ Interrupt Example Program
// RPK
// January 13, 1999   Modified: September 4, 2001
//
// This program uses the IRQ to interrupt the main routine.
// The IRQ pin - PE1 (# 44 on J8) will generate an interrupt
// when it is pulled low (grounded). Each time it makes a
// high to low transition, the interrupt service routine
// IRQInt() is called and a message is printed. If PE1 is
// held low, the routine will not be called again until the
// pin goes high and is then pull low.
//
// This code was written and tested using Introl C 4.0.

// #includes

#include <hc812a4.h>
#include <introl.h>
#include <dbug12.h>

// interrupt service routine declaration
__mod2__ void IRQInt();

void __main ()
{
       int choice;
       long i;

// The IRQ is used to interrupt this module
// Set up interrupt vectors
       DB12->SetUserVector(IRQ, IRQInt);

       DB12->printf("Use the EVB RESET button to exit. \n\n\r");
       _H12INTCR=0xC0;            // 1100 0000 => IRQ falling edge triggered & enabled

       choice=0;
       while(1)
       {
// The strange overlap (overwriting the first output of choice) seems to be necessary
// to get around a bug in printf. The first parameter value is always wrong!
              DB12->printf("%i\rWaiting for an IRQ interrupt. %i \n\r",choice,choice);
              DB12->printf("\n\r");

              choice=choice+1;
              for (i=0;i<65536;i++)      // waste some time
              {
                     ;
              }

       }
}


// IRQ interrupt routine
// NOTE: this is an example of what NOT to do in an interrupt handler. No I/O should
//   be done in interrupt handlers since they are slow and the handler must be quick
__mod2__ void IRQInt()
{
       DB12->printf("   *** IRQI triggered ***\n\r"); // display message when triggered
}
```

Table 8-1 Interrupt vector assignments MC68HC812A4

Vector Address	Interrupt Source	Local Enable Bit	See Register	See Chapter
$FF80:FFCD	Reserved	-		
$FFCE:FFCF	Key Wakeup H	KWIEH[7:0]	KWIEH	8
$FFD0:FFD1	Key Wakeup J	KWIEJ[7:0]	KWIEJ	8
$FFD2:FFD3	A/D Converter	ASCIE	ATDCTL	12
$FFD4:FFD5	SCI-1 Serial System	See Chapter 11.		11
$FFD6:FFD7	SCI-0 Serial System	See Chapter 11.		11
$FFD8:FFD9	SPI Serial Transfer Complete	SPIE	SP0CR1	11
$FFDA:FFDB	Pulse Accumulator Input Edge	PAI	PACTL	10
$FFDC:FFDD	Pulse Accumulator Overflow	PAOVI	PACTL	10
$FFDE:FFDF	Timer Overflow	TOI	TMSK2	10
$FFE0:FFE1	Timer Channel 7	C7I	TMSK1	10
$FFE2:FFE3	Timer Channel 6	C6I	TMSK1	10
$FFE4:FFE5	Timer Channel 5	C5I	TMSK1	10
$FFE6:FFE7	Timer Channel 4	C4I	TMSK1	10
$FFE8:FFE9	Timer Channel 3	C3I	TMSK1	10
$FFEA:FFEB	Timer Channel 2	C2I	TMSK1	10
$FFEC:FFED	Timer Channel 1	C1I	TMSK1	10
$FFEE:FFEF	Timer Channel 0	C0I	TMSK1	10
$FFF0:FFF1	Real Time Interrupt	RTIE	RTICTL	10
$FFF2:FFF3	$\overline{\text{IRQ}}$ pin or Key Wakeup D	IRQEN,KWIED[7:0]	INTCR	8
$FFF4:FFF5	$\overline{\text{XIRQ}}$ pin	X	CCR	8
$FFF6:FFF7	SWI	None		8
$FFF8:FFF9	Unimplemented Opcode Trap	None		8
$FFFA:FFFB	COP Failure (Reset)	None		8
$FFFC:FFFD	Clock Monitor Fail (Reset)	None		8
$FFFE:FFFF	$\overline{\text{RESET}}$ Γ	None		8

Table 8-2 Interrupt vector assignments MC68HC912B32

Vector Address	Interrupt Source	Local Enable Bit	See Register	See Chapter
$FF8O:FFCF	Reserved	-		
$FFD0:FFD1	BDLC	IE	BCR1	15
$FFD2:FFD3	A/D Converter	ASCIE	ATDCTL	12
$FFD4:FFD5	Reserved	-		
$FFD6:FFD7	SCI-0 Serial System	See Chapter 11.		11
$FFD8:FFD9	SPI Serial Transfer Complete	SPIE	SP0CR1	11
$FFDA:FFDB	Pulse Accumulator Input Edge	PAI	PACTL	10
$FFDC:FFDD	Pulse Accumulator Overflow	PAOVI	PACTL	10
$FFDE:FFDF	Timer Overflow	TOI	TMSK2	10
$FFE0:FFE1	Timer Channel 7	C7I	TMSK1	10
$FFE2:FFE3	Timer Channel 6	C6I	TMSK1	10
$FFE4:FFE5	Timer Channel 5	C5I	TMSK1	10
$FFE6:FFE7	Timer Channel 4	C4I	TMSK1	10
$FFE8:FFE9	Timer Channel 3	C3I	TMSK1	10
$FFEA:FFEB	Timer Channel 2	C2I	TMSK1	10
$FFEC:FFED	Timer Channel 1	C1I	TMSK1	10
$FFEE:FFEF	Timer Channel 0	C0I	TMSK1	10
$FFF0:FFF1	Real Time Interrupt	RTIE	RTICTL	10
$FFF2:FFF3	$\overline{\text{IRQ}}$ pin	IRQEN	INTCR	8
$FFF4:FFF5	$\overline{\text{XIRQ}}$ pin	X	CCR	8
$FFF6:FFF7	SWI	None		8
$FFF8:FFF9	Unimplemented Opcode Trap	None		8
$FFFA:FFFB	COP Failure (Reset)	None		8
$FFFC:FFFD	Clock Monitor Fail (Reset)	None		8
$FFFE:FFFF	$\overline{\text{RESET}}$	None		8

Table 8-3 D-Bug12 Monitor interrupts

Number10	MC68HC812A4 Interrupt D-Bug12 Version 1.xxx	MC68HC912B32 Interrupt D-Bug12 Version 2.xxx
7	Port H Key Wake Up	Reserved
8	Port J Key Wake Up	BDLC
9	Analog-to-Digital Converter	Analog-to-Digital Converter
10	Serial Communications Interface 1 (SCI1)	Reserved
11	Serial Communications Interface 0 (SCI0)	Serial Communications Interface 0 (SCI0)
12	Serial Peripheral Interface 0 (SPI0)	Serial Peripheral Interface 0 (SPI0)
23	Timer Channel 0	Timer Channel 0
22	Timer Channel 1	Timer Channel 1
21	Timer Channel 2	Timer Channel 2
20	Timer Channel 3	Timer Channel 3
19	Timer Channel 4	Timer Channel 4
18	Timer Channel 5	Timer Channel 5
17	Timer Channel 6	Timer Channel 6
16	Timer Channel 7	Timer Channel 7
14	Pulse Accumulator Overflow	Pulse Accumulator Overflow
13	Pulse Accumulator Input Edge	Pulse Accumulator Input Edge
15	Timer Overflow	Timer Overflow
24	Real Time Interrupt	Real Time Interrupt
25	IRQ and Key wakeup D	IRQ
26	XIRQ	XIRQ
27	Software Interrupt (SWI)	Software Interrupt (SWI)
28	Unimplemented Opcode Trap	Unimplemented Opcode Trap
-1	Return the starting address of the RAM vector table	Return the starting address of the RAM vector table