Lab Experiments #2: Introduction to Interrupts and the Metrowerks CodeWarrior IDE

CSS490B
Embedded Systems
Last update 3/26/03

Lab 2
Lab Time
Approximately
¾
20 Hrs.

Resources

Lab1A – Connect and Run M5206eLITE ColdFire
Lab1B – Timer and Seven Segment display

Introduction

This lab will introduce you to several new, and from the perspective of embedded systems design, very important, topics. These are:

A. Using the Metrowerks IDE environment to design and debug an embedded application,
B. How an embedded debug core works with the IDE to control the processor during development.
C. How to write programs that use interrupts to control the system's response to real-world events

Finally, you'll use these concepts to create a program that counts 0 – F on the seven-segment display, incrementing every second – using interrupts, rather than a polling loop. You'll notice that the time estimate required for this lab is quite long. This is not intended to scare you off. Rather, it recognizes that some reading and experimentation will be necessary before you're capable of completing the experiment.

Deliverables

For this lab you will turn in these things:

- Your program source files (including .h files)
- Comment each line of code in the timer_interrupt.c code

Equipment Checklist

Hardware

1. MCF5206Elite Evaluation Board
2. Standard D-type, 9-pin, M to F serial connection cable

3. PC
4. ColdFire P & E Wiggler
5. DB-type, 25-pin, M to F parallel connection cable

Software
1. Metrowerks C compiler, assembler, and linker.
2. DBug software (preinstalled on the board)
3. Terminal Emulator (HyperTerminal on Windows)

Documentation
4. Cold Fire Microprocessor Family Programmer's Reference Manual
5. MCF5206e User's Manual
6. Motorola 68000 Programmer's Reference Manual

Online Reference:
PDF files in:
C:\program files\metrowerks\codewarrior\documentation\codewarrior\pdf

1. Command-Line_Tools_Ref.pdf
2. C_Compilers_Reference.pdf
3. MSL_C_Reference.pdf
4. Targeting_Embedded_68K.pdf

¡
It's OK to Interrupt

This is one case where interrupting is good. Interrupts allow software to prioritize task and ensure the important things get done first.

Note about this lab

This lab assumes you have a working, licensed copy of Metrowerks CodeWarrior installed on your lab machine. Check that the CodeWarrior IDE is installed. Click on Start > Programs > CodeWarrior for 68k… > CodeWarrior IDE. If the application is not there, stop now and talk to the professor about getting it installed.

This lab requires some reading that should be done before attempting the programming assignment. Obviously, you cannot thoroughly read all of the documentation and online reference material in time to complete this lab, however you should be familiar with terms and how they relate. To begin, you should already have a good understanding of timers and seven segment display. You will need to learn about timer interrupts and the Metrowerks IDE.

Readings:

Metrowerks IDE
Read Chapters 1 through 3 (pages 9 – 44) in Inside Code Warrior Targeting 68k/Codlfire Embedded Systems Version 2.1.

Timer

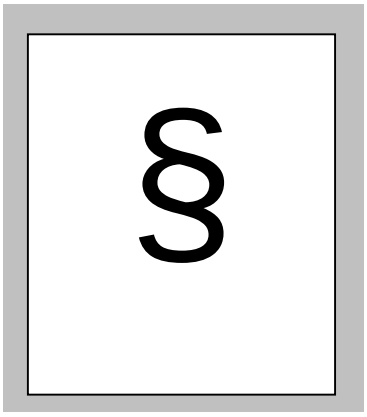
The timer will let your program know when it is time to display the next hexadecimal digit. Review timers in section 14 of the MCF5206e User's Manual

Interrupts

- § MCF5206e User's Manual Section 3.2.4: *Supervisor Programming Model*, an explanation of the status register and VBR.
- § Section 3.3 *Exception Processing Overview*, an explanation of interrupts and exceptions. This is not light reading! Table 3-1, ***Exception Vector Assignments*, is essential, you should bookmark it!**
- § Table 8.2, *Memory Map of SIM Registers*, is another good table. The SIM is the System Integration Module, the subject of Section 8.
- § Section 8.3.2.3, *Interrupt Control Register*, covers the location and bit formatting of the ICRs.
- § Section 8.3.2.4, *Interrupt Mask Register*, is an appropriately named section which described the Interrupt Mask Register.
- § Section 8.3.2.5, *Interrupt Pending Register*, which is very helpful for debugging your interrupts.
- § Appendix A, *MCF5206e Memory Map Summary*. This table contains many of the other memory tables in unabridged form.
- § The other source to have is the ColdFire Microprocessor Family Programmer's Reference Manual. Most of the relevant material mentioned here is in Section 5, *Supervisor (Privileged) Instructions*.

Lab hints

Appendix A contains some hints and suggestions from the former students who did this lab before you.



Stop! Read the suggested material before continuing.

A. Learn the Metrowerks CodeWarrior IDE

Metrowerks IDE

> **i**
> **Stop Wiggling**
>
> The device used to connect to the IDE to the board is called a wiggler, although its correct name is ***Background Debug Mode (BDM)*** probe. It takes control of the board by connecting the host computer to the dedicated debug core hardware inside of the processor. This is why HyperTerminal appears locked when it is running when the wiggler is connected. Running under BDM control is not like running under a debugger. It is closer to a master computer taking complete control over a slave computer.

The Metrowerks IDE is a powerful and complex tool. It is often called an "industrial strength" software development tool suite because it is used by software developers in industry to create embedded applications. Fortunately, it is also intuitive and relatively easy to use without much instruction. Before you start using the timers, you will familiarize yourself with the IDE by creating a “Hello World” application.

Connect the board

To connect the IDE to the board is simple. Just connect the board as you did in the previous labs with a serial cable and HyperTerminal. Then connect a parallel cable to LPT1 on your PC. Connect the other end to the wiggler, and attach the wiggler to the board. You will use 5206e stationary in the IDE that will set up the necessary configuration for you.

Follow the instructions in Chapter 2 of ***Inside Code Warrior Targeting 68k/Codlfire Embedded Systems Version 2.1*** in order to create the “Hello World” application. After you have run the example come back to this point. After you complete the program, you should be able to:

- Connect the board.
- Make the IDE output to HyperTerminal.
- Build an application in the IDE.
- Load, run, and debug and application using the IDE.

> **i**
> **Gotcha**
>
> CodeWarrior does not place the MBAR register at the same address as does the debug kernel on the evaluation board. In other words, MBAR is not located at 0x10000000. You'll need to poke around to find it.

Although this sounds like a lot, you will find that after a few practice runs you can get the board up and running under the IDE in only a few keystrokes.

About Interrupts

Interrupts are a necessity when creating software with real time constraints. Software interrupts are analogous to real-life interruptions. Imagine you are watching an engaging video on interrupts. The phone rings (interrupt level 2) so you pause the DVD (save your context) and answer the phone (acknowledge the interrupt). It turns out it is your friend who wants to discuss interrupts. Just as the conversation gets interesting, another caller beeps through on call waiting (interrupt level 3). You ask your friend to wait (save your context) and answer the other call (acknowledge the interrupt). The caller is the professor to congratulate you on your high score on your interrupt lab. You thank the instructor (service the interrupt) and switch back to your friend (restore context). You then finish your conversation (service the interrupt) and hang up. You sit back down and play the DVD again (restore context). 2 minutes later the phone rings again. You look at the caller ID display and determine it must be someone selling aluminum siding (interrupt level 0) and ignore the interruption.

ColdFire Interrupts Overview

This explains the general steps required to handle interrupts on the Motorola ColdFire 5206e processor. You should be familiar with 68K/ColdFire assembly language and the general concepts of interrupts. The process of setting up an interrupt may be divided into five steps. The five steps below are organized roughly in the order that they would appear in your application.

1. Moving the **Vector Base Register**, a preliminary system initialization step to get the vector table into RAM.
2. Setting the **Status Register's** interrupt priority mask to enable interrupts.
3. Initializing an **Interrupt Control Register** to set the level and priority of an interrupt source.
4. Permitting an interrupt source to generate interrupts by using the **Interrupt Mask Register**.
5. Placing your interrupt handler's address in the **vector table** where the processor can find it.

Additional sections are available on enabling and disabling interrupts, troubleshooting, and other sources that may be helpful to you.

Interrupt Levels and Priorities

Because there is a possibility of one interrupt occurring during the processing of another, all interrupts should be differentiated by their importance. More important interrupts may interrupt the handling of less important interrupts, but not the reverse. Motorola describes the differing importance of interrupts using the terms 'level' and 'priority'. The 'level' of an interrupt is a number between 0

and 7, 7 being the highest importance. Within each level, there are four priorities, between 0 and 3. For example, an interrupt of level 4, priority 2 is of higher importance than one of level 4, priority 1. An interrupt of level 5, priority 0 would be of higher importance than both.

The following examples have assembly code to implement portions of the timer interrupt. The code you will be given later will be in C. Hopefully, by having two ways to look at the same problem, you won't get stuck as easily.

Step 1: Vector Base Register

What you should know: The Vector Base Register (VBR) is the internal register that holds the location of the vector table. The vector table contains addresses of the initial stack pointer and program counter at reset, error handlers for when your code does bad things, and interrupt handlers for responding to hardware and software events. We will revisit the vector table in Step 5 below.

What you have to do: The following code shows how to set up the vbr, however it is already set up in the IDE. Look at the ***vectors.s*** file in the IDE for the definition of the VBR and the setting of it ***in mcf5206e_lo.s***.

To manually place your handler's address in the vector table, you must move the vector table into RAM. On the 5206eLITE Evaluation Board, a copy of the vector table has conveniently been made for you at 0x30000000 in RAM. However, the VBR does not point to this table, but to the original table in ROM. Set the VBR to the copy in RAM, using the MOVEC instruction as follows:

```
move.l          #$30000000,A0
movec           A0,vbr
```

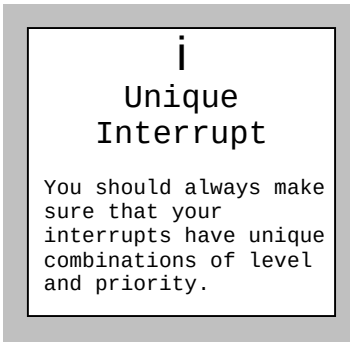
(The vbr is already set for you in the IDE)

Where you can read more

- MCF5206e User's Guide, Section 3.2.4 *Supervisor Programming Model*, an explanation of the status register and VBR.
- MCF5206e User's Guide, Section 3.3 *Exception Processing Overview*, an explanation of interrupts and exceptions. Table 3-1, *Exception Vector Assignments*, is essential, you should bookmark it.
- ColdFire Microprocessor Family Programmer's Reference Manual, page 5-6 explains the MOVEC instruction.

Step 2: Status Register

What you should know: The Status Register contains information about the current status of the processor (as the name suggests). For the purposes of interrupts, we are concerned with only three of the sixteen bits in the Status Register.



The Interrupt Priority Mask occupies bits 8, 9, and 10 (zero-based) in the Status Register. These three bits combined form a number between 0 and 7, which is used to mask out interrupts.
What this means is that if the bits are set to 101, or five, that interrupts of levels five and below will not be executed. If the bits are set to 000, no interrupts are masked. All interrupts are masked with 111.

What you have to do: Before you can handle an interrupt, you must guarantee that the Interrupt Priority Mask has been reduced below the level you have given to your interrupt. Keep in mind that the rest of the Status Register should remain the same while you perform this operation. The following code clears all of the Interrupt Priority Mask bits in the Status Register:

```
move.w      sr,D0
bclr        #8,D0
bclr        #9,D0
bclr        #10,D0
move.w      D0,sr
```

Where you can read more: MCF5206e User's Guide, Section 3.2.4.1 *Status Register* explains the contents of the Status Register and the Interrupt Priority Mask bits.

Step 3: Interrupt Control Register

What you should know: The Interrupt Control Registers (ICRs) are used to assign levels and priorities to interrupt sources. Each ICR is an 8-bit register, which looks something like this:

7	6	5	4	3	2	1	0
AVEC	-	-	IL2	IL1	IL0	IP1	IP0

AVEC is a single on/off bit that determines if the interrupt is an "Autovector" or not. Timer and UART interrupts should have this bit set. TRAP instruction handlers and error handlers should not.

IL2-IL0 form the interrupt level. These three bits combine to make a number between 0 and 7, with 7 being the highest priority and 0 meaning that no interrupt should be generated.

IP1-IP0 form the interrupt priority, which is a sub-distinction for interrupts of the same level. Priority 3 is the highest, priority 0 the lowest.

All ICRs may be specified as an offset from the Module Base Address Register (MBAR). The following table illustrates a subset of the ICRs that you are most likely to use:

Interrupt Source	ICR	Location
Software Watchdog Timer	ICR8	MBAR+$01B
Timer 1	ICR9	MBAR+$01C
Timer 2	ICR10	MBAR+$01D
MBUS	ICR11	MBAR+$01E
UART 1	ICR12	MBAR+$01F
UART 2	ICR 13	MBAR+$020
DMA 0	ICR 14	MBAR+$021
DMA 1	ICR 15	MBAR+$022

What you have to do: You must initialize the appropriate Interrupt Control Register for your interrupt source. You must select a Level and Priority for your interrupt, and you must specify if it is an Autovector or not.

The following example will initialize the ICR for Timer 1, ICR9. Timer interrupts are Autovectors, and we will set it to Level 4, Priority 1.

```
MBAR      EQU       $10000000
ICR9      EQU       MBAR+$01C
movea.l   #ICR9,A1
move.b    #$90,(A1)  ;Level4 Priority1 Autovector
```

Where you can read more: MCF5206e User's Guide, Table 8.2, *Memory Map of SIM Registers*, is another good table. The SIM is the System Integration Module, the subject of Section 8. MCF5206e User's Guide, Section 8.3.2.3, *Interrupt Control Register*, covers the location and bit formatting of the ICRs.

Step 4: Interrupt Mask Register

What you should know: The Interrupt Mask Register allows you to enable and disable interrupts individually by source, rather than just by level with the Status Register. For example, you might want to disable all timer interrupts while leaving the rest of the system intact. For that matter, you might want to disable interrupts from only Timer 1, but let Timer 2 continue to interrupt. The Interrupt Mask Register, or IMR, is located at MBAR+$36. The IMR consists of 16 bits, each bit corresponding to an individual interrupt source. More specifically, each bit corresponds to an Interrupt Control Register. The IMR bits are as follows:

i

Who was that masked interrupt?

When the bits are set, the interrupts are disabled (masked). When the bits are cleared, interrupts are enabled.

Interrupt Source	Interrupt Mask Register Bit Location
External Interrupt Request 1 External Interrupt Priority Level 1	1
External Interrupt Priority Level 2	2
External Interrupt Priority Level 3	3
External Interrupt Request 4 External Interrupt Priority Level 4	4
External Interrupt Priority Level 5	5
External Interrupt Priority Level 6	6
External Interrupt Request 7 External Interrupt Priority Level 7	7
Software Watchdog Timer	8
Timer 1	9
Timer 2	10
MBUS	11
UART 1	12
UART 2	13
DMA 0	14
DMA 1	15

What you have to do: To enable your interrupt to run, you will have to clear the appropriate bit in the IMR. You should preserve the state of the other bits in the IMR. The following code sample clears bit #9, enabling interrupts for Timer 1:

```
IMR             EQU         MBAR+$36

movea.l         #IMR,A1
move.w          (A1),D2
bclr            #9,D2
move.w          D2,(A1)
```

Where you can read more: MCF5206e User's Guide, Section 8.3.2.4, *Interrupt Mask Register*.

Step 5: Handler Address

This section describes how to register an interrupt handling function.

What you should know: The goal of your interrupt is to have the processor call your function when some event occurs. The function being called is a handler for the interrupt. Before your handler will be called, you must register it with the processor by placing the address of the first handler instruction in the Vector Table. The Vector Table is defined by the Vector Base Register. Each of your

interrupts will have an entry in the Vector Table for the handler. The vector table looks something like this:

The

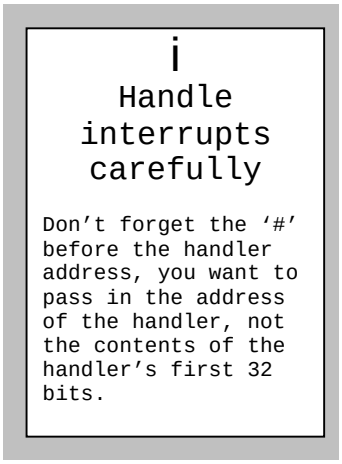
Vector Number	Vector Offset (Hex)	Assignment
0	$000	Initial stack pointer
1	$004	Initial program counter
2	$008	Access error
3	$00C	Address error
4	$010	Illegal instruction
5-7	$014-01C	Reserved
8	$020	Privilege violation
9	$024	Trace
10	$028	Unimplemented line-a opcode
11	$02C	Unimplemented line-f opcode
12	$030	Debug interrupt
13	$034	Reserved
14	$038	Format error
15	$03C	Unitialized interrupt
16-23	$040-$05C	Reserved
24	$060	Spurious interrupt
25-31	$064-$07C	Level 1-7 autovectored interrupts
32-47	$080-$0BC	Trap #0-15 instructions
48-63	$0C0-$0FC	Reserved
64-255	$100-$3FC	User-defined interrupts

Vector Offset column specifies an offset from the Vector Base Register that holds the memory address of the handler for that interrupt. For example, the address of the error-handler for Illegal Instructions is stored at VBR+$010. When an illegal instruction is encountered, the processor will get this address from the vector table, then execute the code starting at that address. Autovector interrupts, are those used by the Timers and UARTs, are differentiated by their level. An autovectored timer interrupt of level 4 will have its handler at VBR+$070, as would a UART interrupt of the same level. The moral of this story is that you need to differentiate your autovectored interrupts by level.

What you have to do: You must locate the appropriate entry in the vector table for your interrupt and place your handler's address in that location.
The following code sample shows how to set the handler for an autovectored interrupt of level 4 (such as a timer interrupt).

```
HANDLER_ENTRY    EQU     VBR+$070

movea.l          #HANDLER_ENTRY,A1
move.l           #TimerHandler,(A1)
```




```
* Timer Handler Subroutine
TimerHandler:
    .
    .
    .
```

Where you can read more: MCF5206e User's Guide, Section 3.3 *Exception Processing Overview*, a dense explanation of exceptions. The complete vector table on page 3-7 is worth looking at.

Handler Considerations

An interrupt handler must be well behaved. You should at least consider all of the following things in your handler:

1. Saving registers. Similar to other subroutines, you should be conscientious of preserving the state of data in the registers. Unlike other subroutines, your handler may have been called before the interrupted routine could perform clean up. Save the registers before you modify them. Be sure to restore the old register contents at the end of your handler.

2. Mask interrupts. You may wish to mask other interrupts at the start of your handler. This should be performed with the status register's interrupt priority mask. Be sure to restore the status register to the previous interrupt priority mask before you exit your interrupt handler.

3. Acknowledge the interrupt. Some interrupt sources, such as the timer, require that you reset a status flag or perform some other acknowledgement. You should look for this in the documentation of the interrupt source.

4. Return from Exception. Interrupts do not return in the same fashion as a typical subroutine. At the assembly level, this is reflected in the RTE (Return from Exception) instruction being used rather than the RTS (Return from Subroutine). If you are writing C or C++ code, you will have to check your compiler's documentation for how to mark a function as an interrupt handler. Using the HP 68K Compiler, this is performed by placing #pragma INTERRUPT on the line immediately preceding your function definition.

5. Performance. Interrupt handlers should be as brief as possible.

Enabling and Disabling Interrupts

There are three ways of controlling the type of interrupts that occur: changing the interrupt sources, the Interrupt Mask Register, and the Status Register.

Interrupt Source: Many interrupts are activated and deactivated explicitly, such as the timers. If the timers are not initialized with an interrupt enabling flag, no interrupts will occur. Similarly, resetting the timers will disable timer interrupts.

Interrupt Mask Register: The Interrupt Mask Register (IMR) allows interrupts to be blocked by source. The IMR contains a single bit for each interrupt source, which is masked if the bit is set. The IMR is described on page 8-11 of the MCF5206e User's Manual.

Status Register: The status register contains three bits which form the interrupt priority mask. The interrupt priority mask blocks interrupts by priority level rather than by source. See the MCF5206e User's Manual, Section 3.2.4.1 for more information about the status register. Changing the contents of the status register requires supervisor privileges, see the MOVE to SR instruction on page 5-9 of the ColdFire Microprocessor Family Programmer's Reference Manual.

Troubleshooting Interrupts

So you tried to handle an interrupt but it doesn't work. This section covers some ideas about what to do next. Basically use the IDE to check the values of the appropriate registers.

Isolate the Problem: Try working with a smaller subset of your code. Write a test program that sits in loop waiting for an interrupt to occur. Use the debug capabilities of the IDE. Try setting a breakpoint at the entry point to your Interrupt handler routine. If the breakpoint occurs, your interrupt happened.

Verify the Interrupt Source: Find a way to ensure that your interrupt source has been initialized properly and is generating the interrupt request. For example, the timers will set the Output Reference Event bit in the Timer Event Register when an interrupt has been signaled. Where possible, you should test out your source initialization by looking at the internal registers while debugging. You can check the status of the timers, UARTs, chip selects, and other items by debugging.

Verify the Interrupt Control Register: Check to see that the Interrupt Control Register you are using does in fact contain the value you assigned to it.

Verify the Interrupt Pending Register: The ColdFire has an internal register where the current status of interrupts is tracked, the Interrupt Pending Register (IPR). You should ensure that the appropriate bit for your interrupt has been set

in the IPR. The IPR is described in detail in section 8.3.2.5 of the MCF5206e User's Manual.

Verify the Interrupt Priority Mask: You can see the interrupt priority mask with the other status register bits while debugging.

Verify the Interrupt Mask Register: The IMR is part of the System Integration Module, you can verify it by debugging. Make sure the appropriate bit is cleared for your interrupt.

Verify the Interrupt Handler: Display the vector table and ensure that your handler is correctly set. You will need to look at your listing files to find the correct address for the handler. Then calculate the address by using the VBR+offset formula. For example, with the VBR at $30000000, a level 4 timer interrupt handler would by at #30000070.

Now put it all together: Writing your program

This is the coding section. This lab is functionally identical to lab 2, however it will be interrupt-driven instead of polled. You will be given the functions to implement the timer_interrupt files (see the end of this lab), your assignment will be to fill in the necessary code and create seven_segment.h, seven_segment.c, and main.c.

Begin by creating these 5 files:
1. seven_segment.h
2. seven_segment.c
3. timer_interrupt.h
4. timer_interrupt.c
5. main.c

seven_segment files

seven_segment.h: Add the #define statements from your lab1 that involve the seven segment display to the seven_segment.h file.

seven_segment.c: Create the following functions in the .c file

- **`void SevSegInitialize(void);`**
- **`void SevSegWriteHexDigit(int);`**
- **`void SevSegClearDisplay(void);`**
- **`void SevSegWriteDot(void); /* This is optional */`**

interrupt_timer files

interrupt_timer.h: Copy the interrupt_timer.h file found at the end of this lab into your timer_interrupt.h file.

Interrupt_timer.c: Copy the interrupt_timer.c file found at the end of this lab into your timer_interrupt.c file.

main: You now need to put all the pieces together and create main.c. Create two functions: **main()** and **void writeDigit().**

main() must:
1. Initialize the seven segment display,
2. Clear the seven segment display,
3. Register the **`writeDigit`** function to be called by the interrupt. use the following code: **`registerTimerInterrupt((unsigned long) writeDigit);`**
4. Start the timer, and run in an endless loop.

`writeDigit():` Copy the code for **writeDigit()** at the end of this lab into your **writeDigit()**

With the code from your previous lab and the code at the end of this one, you have almost all of the code to implement this lab. Put together all of the pieces and get the interrupts working. **You must comment each line of code in the interrupt_timer.c code to explain what it is doing and why.**

Look in these files for help on how the IDE works behind the scenes:

- map.h
- mcf5206e.h
- mcf5206e_lo.s
- sysinit.c
- vectors.s

Your solution should display 0 – F in one second intervals.

Deliverables

For this lab you will turn in these things:

- Your program source files (include .h files)
- Comment each line of code in the timer_interrupt.c code to explain what it is doing and why.
- A brief critique of this lab. What can we do to improve it. Note typos, inconsistencies and lack of clarity.

interrupt_timer.h

```
#ifndef __INTERRUPT_TIMER_H___
#define __INTERRUPT_TIMER_H___

extern void start_time(void);
extern void stop_time(void);
extern void registerTimerInterrupt(unsigned long function);

#endif
```

interrupt_timer.c

```
#include "interrupt_timer.h"
#include "seven_segment.h"


typedef unsigned short WORD;
```

```
#define MBAR  0x20000000      // This must agree with what
is set up
                         // in the P&E Wiggler init file

/* Defines for ColdFire 5206E Timer 1 */
#define TTMR1 (MBAR + 0x00000100)
#define TTRR1 (MBAR + 0x00000104)
#define TTCR1 (MBAR + 0x00000108)
#define TTCN1 (MBAR + 0x0000010c)
#define TTER1 (MBAR + 0x00000111)

#define TTMR2 (MBAR + 0x00000120)
#define TTRR2 (MBAR + 0x00000124)
#define TTCR2 (MBAR + 0x00000128)
#define TTCN2 (MBAR + 0x0000012c)
#define TTER2 (MBAR + 0x00000131)

/*
TMR1 is defined as
[15:8] = 0x00 / 0xFF     divide clock by 0 / 256
[7:6] = 00               disable interupt
[5] = 0                  output = active-low pulse
[4] = 0 / 1              disable / enable ref. interrupt
[3] = 0 / 1              free run mode disabled / enabled
[2:1] = 10               master clock/16
[0] = 1 / 0              timer1 enabled / disabled

0xFFEC = 65516
*/

inline asm void setupisr()
{
     move.b    #0x9E,D0
     move.b    D0,0x2000001C
     move.w    #0x2000,SR
}

void registerTimerInterrupt(unsigned long function) {
     unsigned long *autovector;
     setupisr();

     autovector = (unsigned long *)0x7c;
     *autovector = function;
}

void start_time()
{
```

```
     volatile WORD *memptr;


     memptr = (WORD *)TTMR1;
     *memptr = (WORD)0x0000;

     memptr = (WORD *)TTCN1;
     *memptr = (WORD)0x0000;

     memptr = (WORD *)TTRR1;
     *memptr = (WORD)0x337F;

     memptr = (WORD *)TTMR1;
     *memptr = (WORD)0xFF1D;
}

void stop_time()
{
     volatile WORD *memptr;

     memptr = (WORD *)TTMR1;
     *memptr = (WORD)0x0000;
}
```

writeDigit()

```
__declspec(interrupt) void writeDigit()
{
     static int i = 0;
     if (i == HEX) {i = 0;}
    SevSegWriteHexDigit(i++);
    stop_time();
    start_time();
}
```

Appendix A

Hints and suggestions

- Note: You cannot initialize variables in the header file. We found this out the hard way after several frustrating days of not being able to compile our code.
- The most valuable part of the lab for us was commenting the timer_interrupt.c file. We had done so much cutting and pasting of our code that we had lost track of what was given to us and what we had coded ourselves. However, after we commented the code everything became very clear. I suggest that you do this as one of your first steps after you create the files.
- It took us a while to discover that in order to RESET the CodeFire development board we had to disconnect the P&E debugger/wiggler before the RESET button is pushed.
- When setting up the Metrowerks IDE it is important to make sure that you set up the stationery for the ***M5206e Eval Lite*** board and the C programming language. We first used the wrong stationery, which automatically set-up CSAR1.
- CSAR1 has to be set-up for the lab to run correctly.
- Note that the CodeWarrior's register display shows an incorrect value for MBAR (0x20000001). We set MBAR to 0x20000000.
- CSAR3 must be set to $4000.
- One good way to check to see if you have the pre-scalar divide ratio set correctly is to time the display changing numbers against your watch.
- Need to figure how to set-up the Vector Base Register. The lab itself doesn't describe it very completely.
- Watch out! The codeWarrior environment sets-up CSAR3 for its own needs.
- We spent a lot of time learning the Metrowerks IDE. This is an unavoidable evil.
- We lost some time until we realized that we have to set the timer #1 address register and also figuring out where VBAR was originally mapped.
- The following comments were made by one of the teams. I have not been able to verify the comments yet, so I didn't want to make any corrections until I had some more time to investigate. I've included them for your information, but they might be misleading.
    1. On page 6, under **"Step 1: Vector…."**, it mentions copying the vector for you to memory location 0x30000000. After testing this out, we found this not to be true. Also, you should note that the IDE reports VBR at 0x7C, but it really is 0x0 and ignores the lower 20 bits. Also, the code example right there is misleading and should be taken out since you don't have to change VBR.

2. On page 8, the code example is not clear and may cause confusion as to whether it should go to main() or possibly one of the files already in the project that contain assembly. You should also include a description of each of these files and show where the symbol table can be found.
3. On page 16 please specify what the P&E wiggler init file is!