C Language Programming, Interrupts and Timer Hardware

In this sequence of three labs, you will learn how to write simple C language programs for the MC9S12 microcontroller, and how to use interrupts and timers.

WEEK 1

Introduction and Objectives

The C programming language is used extensively in programming microprocessors. In this lab you will write some simple C programs which do the things you did in assembly language in Lab 2. For example, the following C program displays a counting pattern on the LEDs connected to Port B:

```
/* A C program to display an incrementing pattern on          *
 * Dragon12 Plus LEDs                                         */
#include "hcs12.h"          /* get MC9S12DP256 port defs      */
#define D_1MS (24000/13)    /* Inner loop is 13 cycles        */
                            /* 24,000 cycles is 1 ms          */
                            /* for 24 MHz clock               */
#define TRUE 1              /* A standard C define            */

void delay(unsigned int ms);

int main()
{
    DDRP = DDRP | 0x0F;       /* Make 4 LSB of Port P outputs */
    PTP = PTP | 0x0F;         /* Turn off seven-seg LEDs      */
    DDRJ = DDRJ | 0x02;       /* Make Bit 1 of Port J output  */
    PTJ = PTJ & ~0x02;        /* Enable individual LEDs       */
    DDRB = 0xFF;              /* Activate LED control lines   */
    while (TRUE) {            /* Loop forever                 */
        PORTB = PORTB + 1;    /* Increment Port B             */
        delay(100);           /* Wait a bit                   */
    }
}

/* Function to delay ms milliseconds                          */
void delay (unsigned int ms)
{
    volatile unsigned int i;  /* Need volatile so compiler    */
                              /* does not optimize            */
    while (ms > 0) {
        i = D_1MS;
        while (i > 0) {
            i = i - 1;
        }
        ms = ms - 1;
    }
}
```

Figure 1. A C program to increment LEDs connected to Port B.

PreLab

For the prelab write the program for Part 4 of this lab.

The Lab

1. The procedure for compiling a program using the Gnu C compiler and the EGNU IDE is discussed in detail in Section 5.10 of the text. Start the EGNU IDE, and select File, then New source file. Type in the above C program, then select File, then Save unit as, and give it the name `inc.c`, and save it in your EE 308 directory. Select File, then New project, give the project an appropriate name (such as `Lab3_Part1.prj`), and save it in your EE 308 directory. (Note: The project name must be different than the C program base name. If the C program name is `inc.c`, the project name cannot be inc.prj.) When Project options pops up, click on the down arrow below Hardware Profiles, and select *Dragon12*. Click on Edit Profile, and make sure the addresses and lengths of the memory locations are reasonable. Make sure `inc.c` is part of your project. Also, make sure the file `hcs12.h` is in your directory. Compile and link your project by selecting Build, then Make. You should now have a file "`*.s19`" and a "`*.dmp`". The "`*.s19`" can be run on the MC9S12.

2. The `Lab3_Part1.dmp` contains the assembly language listing generated by the C compiler. Look at the file and try to understand what it does. Note that there may be some things which do not make sense to you. At the very least, find the assembly language code which increments Port B.

3. The file `Lab3_Part1.dmp` also shows the addresses of the start of the functions in the program, as well as the addresses of any global variables. (Since the `inc.c` program does not use any global variables, none will appear in the `Lab3_Part1.dmp` file. The local variables used in `inc.c` are allocated on the stack when they are needed.)

Note that the function and variable names are denoted by the symbol `<_var name>`.

Note also that there is a function `<_exit>`. Find the address of this function.

Look at the file `Lab3_Part1.s19`. This contains the op codes that will be loaded into the HC12. Reverse assemble the `<_exit>` function. What does it do? Load the file inc.s19 into your MC9S12 and run it. Verify that the LEDs increments.

4. Using the program `inc.c` as a model, write a C program to implement the program from Lab 2, week 3. Compile and run your program. Have an instructor verify that it works.

5. Look at the output of the GNU IDE and determine how many bytes the program takes (the length of the .text segment). Compare this to the length of last week's program written in assembly.

6. Put your program in the EEPROM at address 0x0400. Remember, when you put code into EEPROM you need to do some setup which DBug12 normally does for you. You need to convert the assembly-language code (which multiplies the clock by 6) from Lab 2 into C code, and add it as the first lines of you program. There is no C statement to implement the assembly-language instruction "`sei`". You can use the asm() function to insert this (or any other assembly language instruction) into you program:

```
asm(" sei;");
```

You will want the array which stores the turn signal patterns into the EEPROM (so the array will not disappear when you turn off power). You will want variables which will change as the program is executed to be placed in RAM. You can tell the compiler to put constant data (such as an array of patterns to be

display on LEDs) immediately following the code (so the data will be loaded into EEPROM) by defining the data as type const. An example of setting up an array of type const is:

```
const char table[] = {0xaa,0xbb,0xcc};
```

Finally you need to tell the compiler to put the program into EEPROM. You can do this in one of two ways. In the directory with your project there should be a file called **memory.x**. You can edit memory.x to look like this:

```
OUTPUT_FORMAT("elf32-m68hc12", "elf32-m68hc12","elf32-m68hc12")
OUTPUT_ARCH(m68hc12)
ENTRY(_start)
SEARCH_DIR(C:\usr\lib\gcc-lib\m6811-elf\3.3.5-m68hc1x-20050515\m68hc12\mshort)
MEMORY
{
        ioports (!x) : org = 0x0000, l = 0x400
        eeprom (!i) : org = 0x0f00, l = 0x0100
        data (rwx) : org = 0x1000, l = 0x1000
        text (rx) : org = 0x0400, l = 0x0b00
}
PROVIDE (_stack = 0x3c00);
```

This will tell the compiler to put text at address 0x400, which is in the EEPROM. Compile your program, load it into the 9S12, and verify that it runs correctly out of EEPROM.

Another way to do this is to go to the Options menu, select Project Options, and make sure the Hardware Profile is set to Dragon12. Then select Edit Profile, and set eeprom to start at f00 with length 100, set text to start at 400 with length b00, set data to start at 1000 with length 2c00, and set the stack to 3c00. Click OK to accept these settings. Then go to Build and select Create `memory.x`. (Note: you will have to change `memory.x` back to its old values when you want to reload programs into RAM.)

USING THE MC9S12 TIMER OVERFLOW INTERRUPT AND REAL TIME INTERRUPT

WEEK 2

Introduction and Objectives

Enabling an interrupt on the MC9S12 allows your program to respond to an external event without continually checking to see if that event has occurred. Once the event occurs, the MC9S12 interrupt subsystem will transfer control of your program to an interrupt service routine (ISR) to handle the event, and then return control to your original code sequence. In this week's lab you will write assembly and C language program which enable and use interrupts.

The interrupts on the MC9S12 which are easiest to use are the Timer Overflow Interrupt and the Real Time Interrupt. These interrupts allow you to interrupt the microcontroller after a specified amount of time has passed.

PreLab

For the prelab, write the programs for Sections f and g. Also, calculate the time asked for in Part e.

The Lab

a. Connect your MC9S12 to your computer. At the DBug 12 prompt, display the contents of the TCNT register. Do this several times. How do the values compare?

b. Use DBug12 to modify the TSCR1 register to enable the counter. Repeat Part 1.

c. Use DBug12 to modify the TSCR1 register to disable the counter. Repeat Part 1.

d. Start with the following do nothing program:

```
#include    "hcs12.inc"
prog:   equ     $2000
stack:  equ     $2000

        org     prog
        lds     #stack   ; Need stack when using interrupts

loop:   wai
        bra     loop
```

Add code to make PORTB an output port, to enable the eight individual LEDs, and disable the seven-segment LEDs. Then add a Timer Overflow Interrupt to increment the four lower bits for PORTB. Set the timer overflow rate to be 175 ms. You should increment the four lower bits of PORTB in the interrupt service routine, and leave the four upper bits of PORTB unchanged. Verify that the lower four bits of PORTB function as an up-counter.

Note: To use interrupts in an assembly language program you will need to include the file hcs12.inc, which contains the addresses of the interrupt vectors. Here is a part of the hcs12.inc file showing the address of the Timer Overflow interrupt vector:

UserTimerOvf equ $3E5E

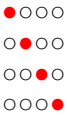
Suppose you want to use the Timer Overflow interrupt in your program. You would need to write a Timer Overflow ISR, with a label (say, tof_isr) at the first instruction of the ISR. To set the Timer Overflow interrupt vector in your program, you would include the following instruction in your program, before the instruction which enables interrupts:

movw #tof_isr,UserTimerOvf

e. Calculate how long it should take for the lower bits of PTH to count from 0x0 to 0xF and roll over to 0x0. Use a watch to measure the time. How do the two times agree?

f. Add a Real Time Interrupt to your assembly language program. Set up the RTI to generate an interrupt every 65.536 ms. In the RTI interrupt service routine, implement a rotating bit on the four upper bits of PTH, while leaving the four lower bits of PTH unchanged. Verify that the bit takes the correct amount of time to rotate through the four upper bits.

A rotating bit will look like this:



g. Implement the above program in C. (What you need to do to use interrupts in C is discussed in the textbook and lecture notes, and described briefly in Section k below.)

h. Change your C program so that you do not reset the Timer Overflow Flag in the Timer Overflow ISR. Does your up-counter work? Does your rotating bit work? Why?

i. Restore your original C program from Part g. Now change your C program so that you do not reset the Real Time Interrupt Flag in the Real Time Interrupt ISR. Does your up-counter work?
Does your rotating bit work? Why?

j. Restore your original C program from Part g. Change your C program so that you do not set the address for the Timer Overflow Interrupt. Run your program. What happens now? Why?

k. To add interrupt vector in C you will need a file listing the addresses of the interrupt service routines. Note that the different versions of the HCS12 have different interrupt vectors, and hence use different vector files. The file vectors12.c (available on the EE 308 homepage) has the appropriate vectors for the MC9S12DP256 chip. Here is part of vectors12.h:

```
#define VECTOR_BASE  0x3E00
#define INTERRUPT      __attribute__((interrupt))
#define _VEC16(off) *(volatile unsigned short *)(VECTOR_BASE + off*2)
```

```
#define UserTimerOvf    _VEC16(47)
#define UserRTI         _VEC16(56)
```

This tells the compiler that, for example, UserTimerOvf is a pointer to a short (16bit) number at address 0x3E5E (0x3E00 + 2*47).

To compile your program with interrupts, include the vectors12.h file in your program. Then, before enabling interrupts, set the interrupt vectors for those interrupts you are using. For example, if you are using the Timer Overflow Interrupt, and the name of the Timer Overflow ISR is toi_isr(), put the following line in your program:

UserTimerOvf = (unsigned short *)&toi_isr;

HCS12 TIMER INPUT CAPTURE AND OUTPUT COMPARE

WEEK 3
Introduction and Objectives

Last week you wrote programs using the MC9S12 Timer Overflow Interrupt and Real Time Interrupt. This week you will work with the timer Input Capture and Timer Output Compare functions. You will use two Output Compare functions to generate two patterns on Port T pins, and you will use the Input Capture function to determine the time of an edge on one pin of Port T.

1. Start with the following program, which is just a do-nothing infinite loop:

```
#include "hcs12.h"
#include "DBug12.h"
#include "vectors12.h"
#define  TRUE 1

main( )
{
      DDRA = 0xff;      /* make all bits of PORTA output */
      PORTA = 0x00;
      while (TRUE)
      {
            asm(" wai" );
       }
}
```

2. Add to your program a global variable called `value`. Add the following Real Time Interrupt ISR, and add code to the main part of the program to generate a real time interrupt every 2 ms. In your main loop, set `value` to a known value (e.g., 0x1234), and verify that this is correctly displayed on the seven segment LEDs.

```
void INTERRUPT RTI_isr(void)
{
    static unsigned char digit=0;
    const char c2seven_seg[] = { 0x3F, 0x06, 0x5B, 0x4F, 0x66, 0x6D,
                                 0x7D, 0x07, 0x7F, 0x6F, 0x77, 0x7c,
                                 0x58, 0x5e, 0x79, 0x71};
    switch (digit) {
        case 0: PTP = 0x0E;
                PORTB = c2seven_seg[(value>>12)&0x0F];
                break;
        case 1: PTP = 0x0D;
                PORTB = c2seven_seg[(value>>8)&0x0F];
                break;
        case 2: PTP = 0x0B;
                PORTB = c2seven_seg[(value>>4)&0x0F];
                break;
        case 3: PTP = 0x07;
                PORTB = c2seven_seg[(value)&0x0F];
                break;
    }
    if (++digit >= 4) digit = 0;
    CRGFLG = BIT7;
}
```

3. In order to measure the time of an edge from a switch, you must use a debounced switch. Build a debounced switch on the small protoboard on the Dragon12 Plus. Connect your debounced switch to input Capture 2 (PORTT, Bit 2), as shown below:

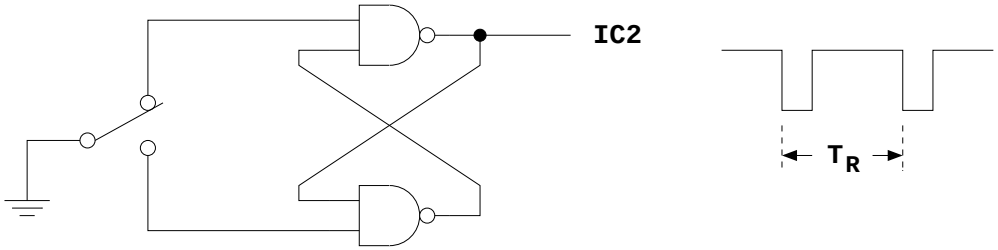


Figure 1. Circuit to measure speed of button pushing

The right part of Figure 1 is what the signal to IC2 will look like if you push the pushbutton twice.

4. Add to your program code to measure the number of timer ticks TR between the two falling edges of the signal in Figure 1. Set the prescaler so you unambiguously measure time difference of at least 250 ms.

Use the *printf()* function to print out the result – the number of timer ticks between pushes of the button. (Do not try to calculate the actual time as a floating point number.)

You should write your program as an infinite loop so that after pressing the button twice your program will print out the result, then go wait for the next two passes.

5. Test your program on your EVBU. See how fast you can push the switch twice. Record several values in your lab notebook, and convert the times to seconds.

6. Add an Output Compare function on Bit 3 of PORTT to generate a 1 kHz square wave. Use a logic probe to verify that Bit 3 of PORTT is toggling.

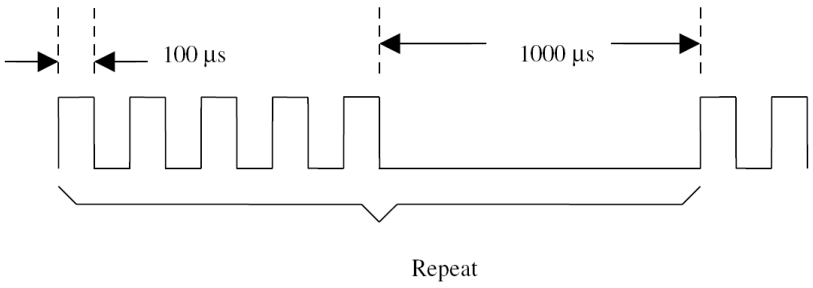


Figure 2. Signal for Part 6.

7. Add an Output Compare function on Bit 4 of PORTT to generate the following signal: The signal consists of five pulses which are high for 100 μs and low for 100 μs, followed by a 1000 μs low signal. This signal then repeats.

8. Connect your output compare signal to a logic analyzer. Verify that the square wave has a 1 kHz frequency and a 50% duty cycle, and that the other signal looks like the signal of Figure 2.