EE475 Lab #3 Fall 2005

Memory Placement and Interrupts

In this lab you will investigate the way in which the CodeWarrior compiler and linker interact to place your compiled code and data in the memory of the HC12 evaluation board. You will also install an interrupt function that is called when the IRQ button is pressed.

Preliminaries

1. Make a temporary local folder for your work: `c:\EEClasses\EE475\tempxxx`.
2. Launch CodeWarrior and create a new project using the New Project Wizard (see Lab #2 if you don't recall the procedures).
3. Make (compile and link) a dummy `main.c` program (empty `main()` routine).
4. In CodeWarrior, open the linker <u>input</u> file, `simulator_linker.prm`. The `.prm` file tells the linker the regions of memory that are available. The `ROM` and `RAM` labels are defined and used to locate the program's code and data.

   → From the `simulator_linker.prm` file, record the `ROM` and `RAM` address ranges and include this information in your memo report.
5. Now open the linker <u>output</u> map file, `simulator.map`. The `.map` file is generated by the linker, and lists the results of the linking process, i.e., where the code and data segments were placed in the HC12 memory. For this minimal C program note that the program occupies only a few dozen bytes, and there are no constants and no static data.

Exercise #1:

Now see what happens to the actual memory allocation when you declare an array in various ways: *automatic*, *static*, *automatic initialized*, *static initialized*, and *global*. Do this by making the following modifications to your C program:

1. Edit your C program by adding the statements within the `main()` block:

   ```
   char buf[40];
   buf[0]='\0';
   ```

   This creates an *automatic* storage class array and simply sticks a null in it.

   → Build the program, open the `simulator.map` file, find the `SECTION-ALLOCATION SECTION` and <u>fill out the first column of the table</u> on the check-off sheet, indicating the size of each of the specified segments. If the segment is not present, just leave that box blank. Ignore the `.abs_section` lines.

2. Now edit your C program to change the declaration to be *initialized*:

```
char buf[40]={"test"};
buf[39]='\0';
```

→ Rebuild and note the size of each segment from the linker output file in column 2 of the table.

3. Again edit your C program to change the declaration to *static*, without initializing:

```
static char buf[40];
buf[0]='\0';
```

   → Rebuild and note the size of each segment in column 3.

4. Once again edit your program to use a *static initialized* array:

```
static char buf[40]={"test"};
buf[39]='\0';
```

   → Rebuild and note the size of each segment in column 4 of the table.

5. Finally, make the `buf` array declaration *global* by moving it up outside of the `main()` function, i.e.,

```
char buf[40];

void main(void)
{…
buf[39]='\0';
…
```

   → Rebuild and record the size of each segment in column 5.

→ Using the `.map` file information, demonstrate for the instructor that you can locate each section and view the memory contents using the debugger.

Note that the memory requirements and linker behavior differ depending on the class of storage used.

1. If the buffer is *automatic* and *uninitialized*, it will only appear on the stack and no memory is allocated explicitly in the program.

2. If the buffer is *automatic* but must be *initialized*, the code image now must include the initialization string AND some additional instructions that will copy the initialization string into the buffer (on the stack) before it is used.
3. If the storage class is *static*, the buffer is placed in a static data segment.
4. Finally, if the buffer is declared *global*, the linker places it in a global memory segment.

→ Be sure to explain your observations in your memo report. Does the contents of you table match these expectations? Which segments contain code (machine instructions) and which contain data? Verify your results and explain.

Exercise #2:

Replace the `main.c` file in your project with the `main.c` from Lab #2 (LED flash program).

Make the program, fix any errors, and launch the debugger. Set the debugger to use the P&E target interface, and verify that the code runs properly (LEDs flash) on the HC12 evaluation board.

Once everything is running properly, go back and modify your C program to include an *interrupt* function, as follows.

To do this you will have to define an interrupt service routine (ISR) by using the type qualifier `interrupt`. This is done as follows:

```
interrupt num void your_function_name(void)
{
      … your ISR code …
}
```

The `interrupt` qualifier is important: it tells the compiler to generate an RTI (return from interrupt) at the end of the function, rather than an RTS (return from subroutine).

The *`num`* indicates which interrupt vector number will be loaded with the address of your interrupt handler.

Write the interrupt function so that it just increments a global variable. Something like:

```
int vcnt;

interrupt num void your_function_name(void)
{
      vcnt++;
}
```

We want to install this interrupt service routine so that it gets executed when the user presses switch 1 (SW1) on the processor daughterboard. The button press generates the XIRQ signal, which is *level sensitive*. You need to determine which vector number (0, 1, 2, etc.) is associated with the XIRQ signal on the HC12. The vector number replaces `num` in the interrupt function declaration.

Keep in mind that after your program sets the interrupt vector to point to your function, you will need to enable (unmask) the XIRQ interrupt signal using the X bit in the condition code register: `_asm("andcc #BF")`.

Once enabled, XIRQ is unmaskable, but for other interrupts on the HC12 you need to do an interrupt enable using the statement `EnableInterrupts;` in your `main()` program.

Make and load your program, then use the debugger to observe the `vcnt` variable before and after you press the IRQ button on the I/O board. Is the variable incremented?

→ **Show the instructor where the `XIRQ` vector address is located in the interrupt vector table. Also, show that the contents of that vector is the address of `your_function_name()`. Give your answers in hexadecimal.**

Exercise #3:

(**3a**) Investigate what happens if you enable XIRQ but you do not install the interrupt service routine.

→ In other words, where does the program jump to if no vector address has been installed?

(**3b**) Finally, modify the LED flashing loop in your `main()` routine so that it stops looping (and thus stops flashing) once the XIRQ button has been pressed.

NOTE that you should put a statement after the loop to prevent your main() program from exiting. An infinite loop is a good example.

→ What happens if you *do* exit from the `main()` routine??
→ Is there an instruction that you can put in your program to "halt" execution and return control to the debugger—just as if you had pressed the red stop button?

Instructor Verification Sheet
Lab #3   Fall 2005

Student Name: ________________________________

SECTION-ALLOCATION SECTION segment sizes in bytes and start address in hex:

Section Name (.map file)		1 auto	2 auto+init	3 static	4 static+init	5 global
`.text`	*Size (bytes):*					
	Start Address:					
`.bss`	*Size (bytes):*					
	Start Address:					
`.data`	*Size (bytes):*					
	Start Address:					
`.startData`	*Size (bytes):*					
	Start Address:					
`.init`	*Size (bytes):*					
	Start Address:					
`.common`	*Size (bytes):*					
	Start Address:					
`.stack`	*Size (bytes):*					
	Start Address:					
`.copy`	*Size (bytes):*					
	Start Address:					

	Instructor Initials	**Date**
#1 Locate and view segments in memory according to `.map` file.		
#2 Demonstrate functioning interrupt service routine and vector table entry.		