EE475 Lab #3 Fall 2004

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) the dummy program.

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 `FLASH` and `RAM` labels are defined and used, although what is identified as Flash EEPROM is actually external RAM: this makes the program loading and debugging easier for our tests.

   → From the `simulator_linker.prm` file, record the FLASH 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.

   → From the `simulator.map` file, record the `.text`, `.startData`, `.init`, `.stack`, and `.copy` section sizes and include this information in your memo report.

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 fill out the first column of the table 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[0]='\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[0]='\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)
{…
```

→ 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.

→ Does the contents of you table match these expectations? Be sure to explain in your memo report.

Exercise #2:

Replace the `main.c` file in your project with the `main.c` from Lab #2 (sequential LED flash program). Also add the `dbug12.h` file to the project.

Make the program, fix any errors, and launch the debugger. Set the debugger to use the D-bug12 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 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).

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

```
int vcnt;

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

We want to install this interrupt service routine so that it gets executed when the user presses the IRQ button on the I/O board. The button press generates the UserIRQ signal.

So, to put the address of this interrupt function where you want it, use the `SetUserVector` monitor function, and the `Address` type cast defined in `dbug12.h`
.

```
DBug12FNP->SetUserVector(UserIRQ,(Address) your_function_name);
```

Keep in mind that after your program sets the interrupt vector to point to your function, you will need to enable (unmask) interrupts on the HC12 using the statement `EnableInterrupts;` in your `main()` program. JUST BE SURE NOT TO ENABLE INTERRUPTS BEFORE YOU STORE THE ISR ADDRESS!! You want the vector to be loaded before it might get used.

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 `UserIRQ` address is located in the RAM vector table - What address is this? Also, show that the contents of that vector is the address of `your_function_name()`. Give your answers in hexadecimal.**

Exercise #3:

(**3a**) You probably noticed that the interrupt routine gets called repeatedly if you hold down the IRQ button. This is because the default behavior for the IRQ line is *level sensitive*: the interrupt is asserted whenever the IRQ pin is pulled low.

The HC12 can be programmed to be negative-edge sensitive for the IRQ line instead of level sensitive. Look at the HC12 documentation and/or Prof. Cady's HC12 textbook to find out how to program the chip for *edge-triggered* IRQ operation. Modify the initialization section of your program to enable the edge sensitive behavior.

→ **Demonstrate the edge-sensitive behavior of the program for the instructor: only one increment of the global variable should occur with each press.**

(**3b**) Finally, modify the LED flashing loop in your main() routine so that it exits once the IRQ button has been pressed three times: break out of the loop if `vcnt` is greater than or equal to 3.

NOTE that you should put a statement after the loop to prevent your main() program from exiting. The line

```
_asm("swi");
```

is a good choice, since it causes a break that turns control back to the debugger.

→ What happens if you *do* exit from the `main()` routine??

Instructor Verification Sheet
Lab #3   Fall 2004

Student Name: ___________________________________

SECTION-ALLOCATION SECTION segment sizes in bytes:

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

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