EE475 Lab #3 Fall 2003

Placement of Code and Data in HC12 Memory

In this lab you will investigate the way in which the Cosmic compiler and linker interact to place your compiled code and data in the memory of the HC12 evaluation board.

Preliminaries

1. Make a temporary local folder for your work: `c:\EEClasses\EE475\tempxxx` .
2. To save time, copy your Lab #1 project into your temp directory. If you don't have the Lab #1 files anymore, obtain them again from the class web site.
3. Launch the Cosmic CPU12 program. You will need to create a new project as you did in Lab #1. If you still have a copy of the `lab1.prj` project, open it, then use `Save As…` to make a new `lab3.prj` project file for this week without having to re-enter all the parameters. You will need to remove the `lab1.c` file from the project file list. Be sure to keep a copy of your unaltered Lab #1 files for future reference.
4. We will use the new CME-12BC32 HC12 Development Boards this week and for the rest of the course. <u>These boards have the D-Bug12 monitor routines located at a different address than before</u>. You will need to edit `Dbug12.h` to change the base address of the monitor. In `Dbug12.h` find the line:
   ```
   #define DBug12FNP ((UserFNP)0xfe00)
   ```
   and change the address to:
   ```
   #define DBug12FNP ((UserFNP)0xf680)
   ```
5. Rename the `lab1.lkf` file to `lab3.lkf` . Then use CPU12 to open the `.lkf` (linker directive) file. Edit the object file name from `lab1.o` to `lab3.o` . Also, change the program start address specified in the `.lkf` file to be 0x1000 instead of 0x4000.

<u>Note</u>: You will want to keep a copy of your lab work on a network drive or a floppy disk since you will want to use this code as the basis for new lab assignments in the future.

Exercise #1:

Use CPU12 to create a new file called `lab3.c` , and write a minimal "do almost nothing" C program:

```
# include "DBug12.h"

void main(void)
{
      _asm("swi");
}
```

Add the file `lab3.c` to the list of files in the project Files icon. Then build (compile and link) the program.

BEFORE you download the S records to the board using Hyperterm, use the D-Bug12 monitor to set the EVB RAM (0x1000-0x3fff) to be all zeros. The command at the monitor prompt in Hyperterm is: '`bf 1000 3fff 0`' (bf means 'block fill' and the syntax is BF <StartAddress> <EndAddress> [<data>]).

Load the S records from the compiler to the eval board using the cut-paste technique and Hyperterm: select and copy the S records from the window, type '`load`'at the monitor prompt in Hyperterm, then select the "paste to host" menu command.

Run the program by typing '`g 1000`' (remember that the program start address was changed to 0x1000 in the linker directives file!). The program should simply cause a software interrupt (User Breakpoint Encountered) due to the '`swi`' instruction.

In CPU12 find the linker output window `lab3.h12` (or `.map` file) containing the *segments* and *symbols* list, etc. Note that your linked program contains 5 segments: .text, .const, .data, .bss, and .debug. The first three are specified in the `lab3.lkf` file, and contain your compiled program code (.text segment), program constants (.const), and static program data (.data).

For this minimal C program note that the program occupies only a few dozen bytes, and there are no constants and no static data.

→ Copy the "segments:" portion of the linker window and include it in your lab report.

The D-Bug12 monitor allows you to view and modify memory on the EVB. Using the monitor '`ASM`' command, take a look at the assembly language instructions created by the compiler which were loaded to `0x1000` in the HC12 memory.

- Can you find the SWI instruction generated by your C code?
- Can you find where the program ends and the block-filled zeros start?
- Does the size of the program match the number of bytes indicated in the linker output file?

→ Copy the assembly instructions for the entire program from the Hyperterm window and include it in your lab report. Give a summary of what each HC12 assembly instruction does.

Exercise #2:

Use CPU12 to edit the `lab3.lkf` file so that the code start address is `0x1500` instead of `0x1000`. Rebuild the C program and download it to the EVB (do NOT zero out the memory this time).

Using the monitor '`ASM`' command, observe the memory beginning at address `0x1500`. Is the program there? Is the program also still located at `0x1000` from the original download? Type '`g 1500`' to run the program in its newly downloaded location, and '`g 1000`' to run the originally loaded program.

→ Show the instructor that two copies of your program are located in memory at the `0x1000` and `0x1500` addresses.

Exercise #3:

Edit your C program to include a print statement before the `_asm("swi");` line:

```
DBug12FNP->printf("Hello, World!\n\r");
```

Build the program and observe the linker output file. Note that the .const segment now contains some bytes, since the constant string `Hello, World!\n\r` has been located there by the linker, along with its required null terminator.

Download and run the program. Using the address information from the linker output file, locate the `Hello, World` string in memory using the '`MD`' (memory display) monitor command. Is the string stored in the right place? Are all the characters stored correctly? Is the amount of storage correct?

→ Show the instructor that the string is located in memory at the location specified by the linker output file.

Finally, try changing the text of the `printf()` string and verify your understanding of the memory size and placement.

Exercise #4:

Next you will investigate what happens when you declare an array in various ways: automatic, automatic initialized, static, and global. Make 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 and write down the size of each segment from the linker output file.

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

```
char buf[40]={"test"};
```

   Rebuild and note the size of each segment from the linker output file.

3. Again edit your C program to change the declaration to *static*:

```
static char buf[40]={"test"};
```

   Rebuild and note the size of each segment.

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

```
# include "DBug12.h"

char buf[40];

void main(void)
{
      …
```

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 the static .data segment and no initialization code is needed.
4. Finally, if the buffer is declared *global*, the linker places it in the global memory segment.

→ Show the instructor your table of the memory segment sizes for the four cases listed above. Also include this information in your report.

Lab Report

The lab report is to be written up in the Memo format. Be sure to put the *lab number* in the Memo header along with your name and date. For each exercise, answer the given questions and demonstrate your understanding of the exercise. Include **commented** file excerpts and the instructor verification sheet to get credit for the lab.

This lab report is due the beginning of the lab period in one week.