Laboratory 8

Programming in C

Due Date: 11th Tuesday

Introduction

The lab gives an overview of programming the HCS12 using C. The program we will use is Freescale's CodeWarrior. The lab explores a very simple existing project and generates errors that you may experience using boards in future project-based courses. Naturally, you'll have your Micros I folder close by as a reference, and this lab may help you get a new programming environment up and running.

Assignment

- Download the file lab09.zip from Blackboard and unzip it on a thumb drive.
- Start CodeWarrior using the desktop icon and open the file lab09.mcp.
- Once the project is open, click on the "Make" icon. There is one on the main menu bar, and there is a second icon in the tab window beside "HCS12 Serial Monitor.
- Normally, an evaluation board would be configured to talk specifically to CodeWarrior, and we would talk to the board through a built-in interface. Our Dragon12+ boards will only communicate with MiniIDE. Open MiniIDE and download the file lab09\bin\lab09.abs.s19.
- Run the program in MiniIDE beginning at $2000 and experiment with the program.
  Question 1: What does the program do?

  The next section will explore the source files and communicating data between them.
- In the lab09.mcp tab, find the Sources folder and open up all three files.
- Look at the main_asm.asm file. This provies almost, but not quite, the same functions that we've been using in lab this term.
  Question 2: What labels is the assembly file making public?

In large programs, those with many thousands of lines of code, one programmer may need to make a large number of labels available to other files. Instead of typing or cutting and pasting all of these references into each file that needs access, these variables can be listed in a separate file called an **include file**. An include file generally does not contain any executable code (although it could).

- Look at main_asm.h. This include file is written for C programs to access the global labels.

  Question 3: What two function prototypes are defined?

  Question 4: A C file must use a special modifier to refer to a global label instead of an internal one. According to the include file, what C modifier corresponds to assembly's XREF directive?

  Question 5: Comment out line 10 in the include file and click "Make". What error message is generated?

  Question 6: Uncomment line 10. Comment out line 8 in the include file and click "Make". What error message is generated?
- Answer the following questions based on the main.c file.

  Question 7: What function prototypes are in the main.c file?

  Question 8: What functions are in the main.c file. Include all types.

  Question 9: Why don't all the functions require function prototypes?
- The next issue in a C environment is controlling the placement of compiled code and variables. Open the file HCS12_Serial_Monitor.prm in the Prm folder. Note that this file can been manually culled to show only data relevant to our program.
- Look at the SEGMENTS section, which assigns labels (left) to types of memory and their address ranges (right side). This particular .prm file ensured compatibility with the Dragon12+ lab setup.

  Question 10: According to the types of memory shown and what you know about the Dragon12+ board, which segments are defined with incorrect types?
- Although this will give a hint at the answer, CodeWarrior complains if "code" is placed into RAM (READ_WRITE) areas and variables placed in ROM (READ_ONLY) areas. Therefore, we need at least one segment of each type. Note that it is largely lines 7 and 14 in this .prm file which determine where our program really starts in memory.
- Note that this is where the vector addresses are set in C. Change the RTI vector to the actual address of $FFF0. Make the project, and download lab09.abs.s19 to the Dragon12+.

  Question 11: What happens when you try to use the actual vector address?
- Recall from earlier this term that the listing file (.lst) is generated by the assembler or compiler, which means that the information in the listing file can't have information for

things that are resolved in the linking process. Open the file main.lst. Look through the file to see how lines of C code were converted into assembly code.

- Find assembly line for “enableInts();”.

  Question 12: What address is used as the destination of the JSR instruction? Is this reasonable?

  Question 13: How many lines of assembly does the instruction “digit(dig[i],digitx[i]);” correspond to?

  Question 14: Overall, how does the number of lines of C code correspond to the number of lines of assembly code?

- Open main_asm.lst and find the line labeled 8629 in the “Abs.” column. The line should contain the “JSR toLED” instruction.

  Question 15: What address appears in the listing file as the destination of the JSR instruction? (almost a trick question)

  This should prove that the object files which correspond to these listing files are not ready to be executed. They must share addresses to fill in these placeholders. The linker not only shares these addresses. It also determines where many things will be placed, which is why there are no ORG statements.

- Open the file lab09.map. This is an output of the linker program which tells the user where everything ended up.

  Question 16: Using this map file, give the addresses that correspond to the following lablels:

  RTI_ISR
  PTH_ISR
  digitx
  digit
  toLED
  enableInts
  _Startup

What to Demonstrate/Submit

- Typed answers to questions.