Embedded systems – Exercise #0

Exercise purpose

Get acquainted with the development environment: compiler, linker and debugger.

Some Compiler flags you should know

-S (capital) → produces assembly code
-O<number> (capital, not zero) → Optimization level. –O7 is for full speed.
-m → produces map file
-o (small) → links and creates the executable (as demonstrated in lab).Your executable is "a.out"
-g → add debug information (as demonstrated in lab)

Elf analysis with elfdump

To dump the section headers (their metadata info), use: elfdump -s <ELF filename>
To dump the disassembly of the code section(s), use: elfdump -T <ELF filename> . If you want to see the source code intermixed into the assembly, add the –S (capital) flag.

Exercise description

1. Download the program source code (ex0.c).

2. Compile ex0.c with full speed optimization to assembly (ex0.s) and then link (ex0.out) and generate a memory-map listing file (ex0.map)

3. Compile ex0.c with source level debug information to assembly (ex0.dbg.s) and then link (ex0.dbg.out) and generate a memory-map listing file (ex0.dbg.map)

4. Using the debugger, do the following:

    a. Trace the code and find out what the “modify” function does.

    b. Use the memory window to watch the “data” array

    c. Set a breakpoint in the “replace” function and see how the array changes every time execution stops.

    d. Use a data watchpoint to follow the flow of the program instead of a code breakpoint.

    e. Display the call stack when in the “replace” function.

    f. Change the values in the array between steps.

g. Count how many instructions the program executed (with full speed optimization and without) (use the Profiling windows).

5. ELF file analysis.

Notes:

- IMPORTANT: Since the analysis examples are easier in GCC , we will be using GCC in this question rather than mcc. This is a great opportunity to get acquainted with the widely-standard GCC. The usage is almost the same:
    - Compile: gcc (-g) <source file name> -o <output executable>
    - The "-c" options allows producing an object.
    - Dump: objdump instead of elfdump. For section headers dump use –h. For code sections disassembly use –d. for intermixing source into the disassembly use –S.
- As you recall from the lab, gcc automatically links the "crt0" library with each program you build. In order to answer the following questions, we want to avoid this because we will see sections originated in the library in each dump. Add the compilation flag -nostdlib to instruct the linker to avoid linking crt0. You should also define "_start" as seen in the lab, otherwise you get a warning (BTW if you try omitting it in mcc you get an error).
- When referring to "sections" in this question we mean one of the 4 section types discussed in class(.text, .data,.rodata, .bss).

a. Create a C file which after compilation and linkage will yield an ELF file with only .text section (again, we don't count debugging info sections, etc.). Name it only_text.c

b. Create a C file which after compilation and linkage will yield an ELF file with only .text and .rodata sectios. Name it text_rodata.c

c. Create a C file which after compilation and linkage will yield an ELF file with only .text , .rodata and .data sections. Name it text_rodata_data.c

d. Create a C file which after compilation and linkage will yield an ELF file with all the sections we discussed in class. Name it all_sections.c

e. Make sure you fully understand the purpose and characteristics of each section type. Explain it in the README as described below.

6. **Bonus (Max 5 points)** – By examining ex0.dbg.s, disassembly dumps and using the "registers" view in the debugger (those are only suggestions, you don't have to use all of them, and you can think of other ways), try to understand how function calls are being implemented in the ABI (Application Binary Interface) or at least what can be understood from the program at hand.
    a. What register implements the “out” variable?
    b. What register(s) implements the “in” variable(s) as far as you can see?

7. **Submit the following files:**
    a. ex0.c
    b. A 'makefile' (with that name) that will generate the following files:
        i. ex0.s, ex0.out, ex0.map
        ii. ex0.dbg.s, ex0.dbg.out
        iii. Executables for question 5 , named: only_text.out, text_rodata.out, text_rodata_data.out, all_sections.out
    c. ‘README’ containing:
        i. Your names and IDs. Remember you should submit in pairs, and only one of you should submit with both names and IDs in the file.
        ii. answers to the following questions:
            1. What does the function “modify” do?
            2. Describe in your words what happens when <u>your</u> code finishes running.
            3. What is the size of your functions (replace, modify, main) in memory when the program runs? What is the total size of the program? What is the cause of the difference? Use the memory-map listing file you generated before.
            4. Answers to the items in question 5. Submit the source files and make sure they are produced correctly by make. In the README add the section header dumps of your executables.
            5. Bonus. What registers implement the “in” and “out” variables as far as you can see?

A friendly reminder: check the very tar you submit. Put it in a clean directory, extract, compile, and check that everything is in order. Submitting a bad tar causes trouble and therefore results in points penalty. On the other hand it's very easy to check against it.
Please do.