Software Production Hints: Coding / Debugging

Written by Ian Troxel, TA

The production of software remains a necessary facilitator for delving into the focus of this course -- a high level overview of design concepts and implementations of microprocessors. All lab assignments, including the hardware expansion lab, require knowledge of software production and debugging techniques. In fact, each lab starting with lab 2 will require the production of software on your part. Due to the fact that most electrical engineers have not taken a rigorous software engineering course by the time they graduate, a sufficient level of coding proficiency is generally lacking. Poor code production will lead to hours wasted on incorrect solution trials or endless debugging. As a point of fact, the use of proper coding techniques will reduce the time you spend preparing labs by ***at least a factor of two***. The following should serve as a quick reference guide as to how to write code that is efficient, accurate and most importantly maintainable (including debugging).

Good Code

The focus of software coding is to create software that does the job correctly and efficiently, but is at the same time intelligible to a programmer. Software is the human-machine interface.

Steps / Goals of Good Coding

1) Conceptualize the Problem
   a. Spend time brainstorming in order to gain a full perspective of the problem at hand before you begin to code.
   b. Write out a flow of how the program should progress including variables needed, functions called and interface requirements.
   c. This should be considered in a top-down approach
2) Modularity
   a. Build the smallest functional blocks that were outlined in the above process first. Prove correctness for each through rigorous testing **BEFORE** moving onto the next component.
   b. Integrate each lower level component into a higher-level component **ONE AT A TIME** and test the correctness of each before adding another.
   c. The focus of integration is to be wary of interfacing between components because the correctness of each stand-alone component should have been proven before integration.
   d. This should be performed in a bottom-up approach
3) Comments and Documentation
   a. Comment all lines of the program describing the function of each.
   b. Provide interface information for all subroutines and main programs.
   c. Use meaningful variable / subroutine names.

Example of a subroutine description:

```
**********************
* Programmer Name:
* Date Created:
* Date Modified:
* Version:
**********************
**********************
* Subroutine Name:
* Input Variables Required:
* Variables Modified:
* Subroutines Called:
**********************
```

4) Maintainability
   a. Write code that will facilitate reuse in the future.
   b. All of the above steps allow for this.

Debugging Tips

The focus of debugging is to **isolate the problem**.

Common Sources of errors (arranged from most to least likely)

- Bad Coding
- Poor Interfacing between subroutines
- Compilation Error
- System / Hardware Error

When faced with code that seems to "just sit there" when it is executed, staring at a blank screen will generally not solve the problem. You must take a proactive role in discovering and hopefully eliminating the problem.

Debugging options

- Simulation (may not reflect the true nature of the system)
- Trace (limited use, especially with jump subroutine and interrupt driven code)
- Breakpoints (limited in number and not good for testing interrupt driven code)
- Tags (probably the best solution to the problem of distributed errors)

Tags are lines of code (generally print statements) that provide some signal to a human operator to show that the sequence of a program has executed a certain line of code. They should be placed at key locations in the code (beginning of branches, within *if* statements, etc.) in order to narrow the focus of debugging.

For example, here are the places in a given sequence of pseudo-code that would be best for tags.

```
A = B + 1;
<software tag: print("arrived at 1");
    If (A < C ) {
        <software tag: print("arrived in first

        A = B +1;
    } else {
        <software tag: print("first else taken");
        A = B +D);
    }

<software tag: print("exited first if");
A = B * 2;
C = B + A;
<software tag: print("got to subroutine sort");
Jump Subroutine Sort
<software tag: print("returned from subroutine
```

The print statements let you know that the execution of your program has reached each of the lines of code before a given print statement. If a print statement is not generated, it is likely that the problem has occurred after the last print statement that was observed. In order to further debug the code, it would be wise to also output critical variables in the vicinity of the problem once you have narrowed down the search.

Other Debugging Hints (When debugging remember to do the following:)

- Initialize variables
- Look at branch criteria
- Look at subroutine interface variables
- Think about stack utilization
- Make sure the proper interrupt subroutines / mask are used / disabled