Z502 PROJECT

STUDENT MANUAL

Release 4.50 - Revised 03/2018

1.0 Introduction

The project is divided into two project phases. Each phase has a working OS implementation due. In addition, there are sub-projects due to make sure you get started with each project in the right way. All code must be well documented, and should be accompanied by a report following the guidelines described later.

2.0 Project Structure

You can get the code source files for the project from the Project Home Page.

You can also find documentation on the project. I especially recommend when starting out that you read the StartHere document.

There's documentation about everything – but from a practical point of view, you may want to read the code for sample.c – that's where you can see the code in use.

Compilers

I've built this code with Eclipse on Windows, with gcc installed on Windows, with a standard gcc on Linux, and with gcc on MacOS. It worked for these four environments.

3.0 Putting It All Together

It is highly recommended that you examine carefully the file StartHere. This really tells you everything about how to get started.

4.0 Project Deliverables

Excruciating detail on my expectations for these deliverables is provided below. Copies of these documents must be submitted at the end of both Project 1 and Project 2.

In general, there's a VERY LARGE amount of work to do. If you are a compulsive, then you need to understand right now, that you may NEVER finish the entire project. The project is purposely left open-ended so that you can go to whatever limit you are willing/able to accomplish; but that doesn't mean that you should spend every waking hour from now to the end of the semester on this project.

In all documents, strive to be concise. The purpose of the documents is to indicate the decisions you made and how effective they were. Your goal should be to communicate that information clearly and succinctly. Assume that I am the audience; as such, there is no need to provide detail on general operating system topics - for instance, you can assume that your reader understands the various aspects of schedulers. In essence, this is not a project you must sell to management and colleagues; thus prettiness is NOT as important as succinctness.

Architectural Document

This document should consist of an *architectural and policy overview*. You may add any additional information you think is useful. The real goal here is "how does everything hang together?" But please, if all of this Architectural Section is more than 5 - 10 pages long, something is wrong

Your architectural document should include EACH of the following subsections:

1. A list of WHAT is included in your design. Which of the many features you could have implemented did you actually do.

2. High Level Design:

   A DIAGRAM (worth several thousand words) showing your various routines and how they hook together. This could well be a structure chart or flow chart; if confused about what this means, then ask. This is meant to be a high level design; I don't need to see a low level design. Please note that you are required to conduct sound architectural design. Monolithic implementation, or one which reflects no or little organization, is guaranteed to affect the evaluation of your project negatively. (In other words, Software Engineering is good stuff.)

3. Justification of the High Level Design you drew in the section above; WHY did you do it the way you did?

4. What anomalies or bugs did you discover - ways your code interacted with the hardware that you couldn't explain. Please put these items on a "pull-out" sheet as I'll keep them for future reference.

Source Code

I will get a copy of your code simply because part of the evaluation process - part of determining that you have completed the tests - is to put your code into a directory, build it, and run it.

I expect documentation of your code. This documentation has two parts:

1. At the beginning of each function is a general paragraph describing what that function is supposed to do. It lists what comes in and what goes out.

2. Within the code itself, there's additional explanation of the code.

The SVC, interrupt, and fault handler routines are a key to your design. These routines will be examined in detail.

Output Considerations

Here is a MAJOR issue; ignore this at your peril. One of your very tough jobs will be to present output that represents what you've done. It's a very narrow path you must follow: present too little data, and it's impossible to see the extent of your work; present too much, and no one will be able follow what your code is doing. Select output quantity based on the table below.

In general, there are four levels of output; some will be appropriate for some tests, but not for others. NOTE: These descriptions apply to the TEST OUTPUT Tables given below.

- The output generated by the code in test.c. This is usually minimal in quantity and serves only as a general marker of what is going on.
- Print requests that occur in the svc, fault, and interrupt code. Stubs for these already exist in the base.c code supplied to you. Expand these to best meet your needs. They are designed to print out the first few occurrences of their execution.
- The output generated by the routines supporting "SchedulerPrinter" in StatePrinter.c. These are useful for indicating what is going on inside the scheduler at any given time. You must interface to this routine, and in so doing, determine what and when you want to display scheduler information.
- The output generated by the routines supporting "MemoryPrinter" in StatePrinter.c. These are useful for indicating what is going on inside the memory at any given time. You must interface to this routine, and in so doing, determine what and when you want to display memory information.

I do NOT want to see output that your OS has generated. It’s great to have debugging aids in your code that provide you with lots of information. But you MUST be able to turn off this additional printing when you submit your project.

Here are Tables indicating the type of output that should be provided with each test.

Table Student-1: Test Output Requirements for Project Phase I

Test name	Output Generated in test.c	SVC/Fault/ Interrupt	Scheduler Printer	Memory Printer
test0	Full	Full	None	None
test1	Full	Full	None	None
test2	Full	Full	None	None
test3	Full	Initial	Full	None
test4	Full	Initial	Full	None
test5	Full	Initial	Full	None
test6	Full	Initial	Full	None
test7	Full	Initial	Full	None
test8	Full	Initial	Full	None
test9	Full	Initial	Full	None
test10	Full	Initial	Full	None
test11	Full	Initial	Limited	None
test12	Full	Initial	Limited	None

Table Student-2: Test Output Requirements For Project Phase II

Test name	Output Generated in test.c	SVC/Fault/ Interrupt	Scheduler Printer	Memory Printer
test21	Full	Full	None	None
test22	Full	Full	None	None
test23	Full	Initial	Full	None
test24	Full	Initial	Limited	None
test25	Full	Initial	Limited	None
test26	Full	Initial	Limited	None
test27	Full	Initial	Limited	None
test28	Full	Initial	Limited	None
Test41	Full	Full	None	Full
Test42	Full	Full	None	Full
Test43	Full	Initial	None	Full
Test44	Full	Initial	None	Limited
Test45	Full	Initial	None	Limited
Test46	Full	Initial	None	Limited
Test47	Full	Initial	None	Limited
Test48	Full	Initial	None	Limited

In the Tables, "**Full**" means the complete output offered by that Option – do not cut off any printing. "**Initial**" means give me the first 10 occurrences of the printout and then do NOT give me the remaining thousands that might occur. "**Limited**" means I would like the first 50 occurrences. "**None**" means just that - don't give me any of that kind of output.

Remember, points are gained by PROVING your implementation works - this output is the best way of gaining that proof.

The Results of Running That Test

You will prove that your program works by running it for me. As you prepare to do this, think of the output from MY point of view. Am I going to be able to understand what you've done based on what you print out. I'll do my best to understand it of course, but you can make life a lot easier by thinking this out beforehand.

5.0 Reality

What can you trust about this project? Numerous classes have gone before you and have suffered through many anomalies and bugs. But perfection is illusive. (And I'm continuously changing the code.)

I've written the z502.c code. I also wrote a hardware diagnostic which runs all the hardware code. There is 100% code coverage for the disk and memory facilities. To the best of my knowledge, I've completely exercised the fault and interrupt mechanism. I built an operating system myself so that I could run all the tests.

This assures that the hardware is able to successfully do context switching both for memory requests and for other types of system calls. I am convinced there are still bugs in the hardware, even though I wrote more diagnostic code than I did hardware code. However, several classes have used the hardware without uncovering any new bugs. There are, of course, always problems with newly written code.

Evaluation

Throughout the project you will be faced with choices of "beautiful, efficient" algorithms that you have designed versus "easy, trivial(ha, ha)" algorithms that you are sure you can implement. I offer the following advice for your general design: *Choose the most complex design that you are **CERTAIN** you can implement*. The emphasis is on getting each phase of the project working. A

very simple design that works will score substantially higher than a wonderfully efficient design that does not. On the other hand, the grading does reflect the complexity of the design implemented. In this regard, MAKE SURE I know, by TELLING ME in your write-up, what features you have implemented.

You will be marked independently on each of the two projects. Your mark for the second project is independent of the first, except that numerous features for Project 1 show up in Project 2. There's no way I can possibly read all your code; I'm faced with either superficially looking over your entire project, or reading pieces in some detail. I'm likely to choose some of each (the algorithm keeps changing.) You could be lucky - the piece I pick could be one on which you did a stupendous job; then again, you could be less fortunate.

The ultimate purpose of this, believe it or not, is to have fun. You will have a great sense of satisfaction in completing this piece of work. Good luck to you.

NOTE: It is often difficult to submit your code. Zipping or RAR'ing your files is a great way to proceed. Many modern mailers will search inside these file and stop delivery if there's an exe in the zip file. Please do NOT include an executable in your submission. Should you really want to send me an executable, create a free account on megaupload.com and send me the link of where you put your data