Creating and Testing Simulation Software

Christina M. Heinich[1]
Texas State University, San Marcos, Texas, 78666

The goal of this project is to learn about the software development process, specifically the process to test and fix components of the software. The paper will cover the techniques of testing code, and the benefits of using one style of testing over another. It will also discuss the overall software design and development lifecycle, and how code testing plays an integral role in it. Coding is notorious for always needing to be debugged due to coding errors or faulty program design. Writing tests either before or during program creation that cover all aspects of the code provide a relatively easy way to locate and fix errors, which will in turn decrease the necessity to fix a program after it is released for common use.

The backdrop for this paper is the Spaceport Command and Control System (SCCS) Simulation Computer Software Configuration Item (CSCI), a project whose goal is to simulate a launch using simulated models of the ground systems and the connections between them and the control room. The simulations will be used for training and to ensure that all possible outcomes and complications are prepared for before the actual launch day. The code being tested is the Programmable Logic Controller Interface (PLCIF) code, the component responsible for transferring the information from the models to the model Programmable Logic Controllers (PLCs), basic computers that are used for very simple tasks.

I. Introduction

Designing software is complicated. You have to understand the kinds of data you are going to use, where they come from, and what your target audience expects. Most importantly, you need to understand how to get that data to the target audience in a way that they not only understand but can safely interact with.

The simulation software project[2] is no different. The goal of the project is to completely simulate a Space Launch System (SLS) launch so that the people manning the launch control room can practice on the simulations and prepare for any and all possibilities before the real launch date. The software team at Kennedy Space Center is responsible for simulating the ground systems (such as cooling and fuel tanks) as well as the connections between the ground systems and the launch control room. [3] Other NASA centers are responsible for creating the simulations for the SLS rocket and the Orion capsule.

The software can essentially be split into three main parts: the models themselves (which are created using a modeling program called Simulink which converts graphical models into C code), the graphical interface that people in the launch control room can use to begin the simulation and interact with, and the framework that makes sure the models and the interface can talk to each other. To expound on that, the framework needs to read data from the models, transfer it to the launch room in a manner similar to the way described in Footnote 3, and transfer

---

[1] Student in the Computer Science and Japanese Departments in Texas State University

[2] official name being the Spaceport Command and Control System (SCCS) Simulation Computer Software Configuration Item (CSCI)

[3] These connections can be summarized into a system of servers, cables, and PLCs, very basic lcomputing machines that take in input, do a simple operation on the data, then output the data through a cable. In our case they are used to get information such as pressure and temperature from various parts of the ground systems.

commands from the launch room to the simulations. The path the data takes is complicated, and it is essential there be no errors. The code needs to be tested to detect and remove all bugs, which is where I come in.

My task for this internship was to write a test for a specific file of the code, the ModelPLC code, and make sure every part of the code worked. Unexpected input should be properly handled and there should be no unexpected errors. I was to fix any errors found in the testing process.

There were a few challenges in doing this. The first one, and the most challenging, was to understand the code to be tested. The second step was to understand how to create and use a unit test. The final one was fixing the errors found.

II. Understanding the Model PLC Code

The ModelPLC code is responsible for transferring data from the models to the model PLCs[4]. The code was written a year ago and came with very little documentation. All programming languages come with a commenting feature that allows a programmer to comment on the code and explain parts that might not be clear to another person reading the code. Commenting is essential for readability, which is very important if somebody else needs to use your code in their own program. The ModelPLC code, however, had none of that. My solution: document it myself.

A. Learning the Big Picture

Before I even started, I had to know the big picture. As it turns out, the big picture is enormous, so let's start at the top, in the Launch Control Systems (LCS).

The program begins with what is called a graphical user interface (GUI). The GUI displays the models, any warnings, and shows options that the user can use to control the simulation such as start, stop, and override. This part of the program is done in Java, a programming language that was designed to be at a higher level than C++[5]. Almost all modern day programs use a GUI (I am using Microsoft Word's GUI right now to type this paper) because, while a purely command line based program will be much more efficient and quicker to use, most users do not know how to use a command line, and a GUI is a much easier (and prettier) way of displaying and interacting with data.

The GUI is just a set of screens and buttons; it does not actually execute any of the processes. What it does do is act as an interface between the user and the SIM Gateway. And the Gateway acts as the interface between the display and the actual program itself. The Gateway can send and receive numerous different messages, but at start up the user will send the message "start the simulation."

The Gateway sends this command to Trick, a piece of Commercial off the Shelf (COTS) software that allows us to run the models through steps; every time a certain measurement of time passes, Trick takes a "step" and updates any changes in the models. [6] The sim group created a lot of components for Trick to sort of "wrap around" and call when needed. The first thing it does is call the Control Interface (CIF), which gets the commands from the Gateway, in this case "start". All of the models in the sim are also called, initialized and connected to the PLCIF. The PLCIF is the final part in the Trick packet, and when it initializes it starts up the EtherNet/IP Scanner Developers Kit (ESDK) interface (another COTS program that transfers information from the PLCIF to the model PLCs) and makes sure all the parts of the models that need to be looking in a certain place for information are looking there and nowhere else. Trick will then take a step, which will update the models with their very first values. The values will

---

[4] This is a bit confusing so let me explain it in detail: to simulate PLCs we use a program called Softlogix, which emulates the PLCS and a basic set of cards to put into the PLCs. In our own software, we have a component called the PLCIF, which acts as the interface between the models and the PLCs. The modelPLC code, despite its name, is part of the PLCIF.

[5] This means, in a nutshell, that Java hides more of the "behind the scenes" processes so when you program it is somewhat closer to what a human would understand. It takes away a lot of the tools of C and C++ that, while very powerful, are also very dangerous if used incorrectly. To quote my mentor, Jason Kapusta, higher level languages make it harder for "you to shoot yourself in the foot."

[6] This was actually something new to me. In modern software development, it is very rare to create software entirely from scratch; because there are so many libraries and programs already out there, there is a good chance one of them will include something you need. As opposed to creating an entire program with original code, a large portion of the development process is finding software that can be used for the project and making it work with other COTS or already built code.

be sent through the PLCIF up to Softlogix, another COTS component that acts as the model PLC. Softlogix then performs whatever operations (if any) are required on the data before sending them to the Gateway, through the LCS network, where the information finally gets to the display GUI for the user to see.

Trick will continue to update the models and send data along the framework path until the CIF gets the command to stop. The user can also override the model at any time. This command gets sent to the CIF like all other commands, but instead of getting to the models the override gets sent straight to the PLCIF, which treats the override value just like any other model variable and sends it off to Softlogix for processing.

There are lots of different moving parts to each of these components, but the basic summary of the software is enough to help with understanding the ModelPLC code; in order to properly test the code, I need to understand what kind of input it expects and what kind of output other components expect of it.

B. Learning to Read the Code

In order to document the code, I had to actually understand how to read it. This was extremely difficult and took the entirety of the unit testing process. For starters, the level of C coding was a few notches higher than I expected. The code used quite a few coding tools and techniques that I had either never seen before or only knew the basics of. In addition, I had assumed the C language was very similar to C++, the computer language I am most comfortable in, which turned out to not be the case.[7] Through trying to understand the ModelPLC code I ended up learning many new tools for any future programming assignment. Here are a few of the concepts to give you an idea of what I learned:

1. Casting: in which a programmer takes a variable of a certain type and casts it to a different type. The variable will be treated like the new data type until the end of the command. All data is stored in memory as a certain data type. The type determines how the computer treats that part of memory. For example, numbers can be stored in many different ways, depending on the kind of number. (Is it a whole number, or does it have a decimal? Does the number require a lot of accuracy, or can it stand to lose some precision? All of these are different considerations in variable declaration.) In C/C++, an "int" represents a basic integer. It usually uses four bytes of memory and a non-negative number is represented with a relatively straight conversion from a decimal number to a binary one. On the other hand, a "double" uses eight bytes and a very different kind of binary representation. Characters can also be represented as "chars", (usually) one byte representations in ASCII code. Every variable is given a label or name in declaration (for example an integer can be declared as int numOfWords). So if I cast a one byte char as a four byte int, that variable will have not only have access to three more bytes of memory but the operations on that variable will treat it as a number, not a character. It is a strange concept that at first glance does not seem useful, but the ModelPLC code uses it as a clever way to pre-allocate memory before it knows what data type will be used.
2. Threads: a way to implement the important concept of parallel programming. Parallel programing is a style of programming that aims to make software faster and more efficient by utilizing the multiple cores of most modern day processors. A very common way to implement this is with threads. Threads are, at the most basic, a line of commands that can be executed *at the same time as another line of commands.*[8] To help to understand, suppose we have a block of 100 commands. Without threading, that block will take 100 clock cycles to execute all of the lines of code. Runtime speed is completely dependent on the speed of the clock in the processor. Now let's say that block is threaded so that the block is split equally onto two threads. Those two threads can execute simultaneously. Instead of 100 clock cycles, the program finishes in 50. Obviously, real life situations are more complicated. Most programs cannot be split equally for numerous reasons. But very often threaded software will run much faster by utilizing the hardware already available. In the SIM project, we use threads to run each model separately as well as running the connections between the different components.
3. Makefiles: a file containing a script that specifies how a program is built and executed. All programs go through a process to change from a file of code to a legitimate program that can be interacted with. First, a programmer needs to write the code and save it as a certain file, depending on the language being used. (For example, a normal C++ file is saved as a .cpp file.) Next the file is processed by a

[7] A small programming history lesson is required. C is a popular programming language used largely back in the 70s until C++ was built from it and standardized. C is still used today, but because C++ is basically C with added features and streamlines, C++ is generally favored over C. An interesting thing about C++ is, because it was built from C, almost all C coding can be used in a C++ program. The opposite, of course, is not true.

[8] Providing of course the computer has a dual core or better processor.

compiler designed specifically for the language. The compiler takes the code and converts it into assembly code, code that is closer to the format a computer can understand. The assembler then takes the assembly code and turns it into binary. The computer speaks binary, so now the computer can tell the hardware what to do when the program is finally ran. All the user needs to do is run the created executable (either through terminal commands or pressing an icon) to get the computer to start the program. This process works well for one file, but most programs are, for many reasons, in multiple files. While it is completely possible to link all of those files together by hand, it is much easier to create a makefile. A makefile will define the compilation and linking process so you do not have to do it again. A makefile can also be designed to keep code from getting re-compiled; if a file of code was not changed, there is no reason to re-compile it because the machine code is already there. In addition, it is easy to customize makefiles to compile a certain way, depending on an extra commands you send. For example, the simulation software uses "make clean" to clean up any leftover files created by past compilations, and "make tests" is designed to run the test files built to exercise the code. (I used this feature multiple times an hour for my testing.) Makefiles are optional, but they are excellent time savers on the part of the programmer.

C. Organization and Commenting

After learning how to understand the code and how it relates to the rest of the project, I could actually understand what the code was doing.

The ModelPLC code is implemented as a library of functions[9] that are called by the Trick simulation as well as various other parts of the code, such as the models or the CIF. Some of the functions initialized the ESDK or the packets of information or assemblies that get passed between the ModelPLC and Softlogix. There is a function that loads a certain kind of file with a list of all the tags (or variables) a specific model has so the PLCIF knows what to look for when it needs to get information. Two functions are dedicated to registering certain tags for output (meaning the measurements from a given model that is to be sent to Softlogix) or input (meaning the commands from Softlogix or other models to be put in a given model). There is also a set of functions designed for specific kinds of tags that aren't supported by Softlogix and thus need to be processed in the ModelPLC instead.

None of these functions came with much documentation, if any. I spent about a week's worth of work deciphering the code, figuring out what each function was supposed to do and how it was doing it, and adding my own detailed documentation for each function. This was a task originally suggested by my mentor as a good idea for future readers, but to be very honest my documentation was more for me. By documenting everything, I not only became very familiar with the code, but the descriptions I added for each function served as a very useful guide in case I needed to go back and remind myself how to use a function.

Documenting here was a little more involved than what I expected. The SIM team uses a program called Doxygen, which looks for comments written in a certain way and processes them to be displayed in an easy to read web page that shows information about the entire project. For example, parameters to functions can be commented with a preceding @param, which tells Doxygen the next line of comment is describing one of the parameters of a function and should be used as such. There are a lot of these little tricks such as Valgrind and Jenkins that are used in the review process. The tools make it easy to see how the project is doing and what still needs to be fixed or added. This will be described more in detail later in the paper.

Suffice to say, for now, that documenting the code before I started was a huge help.

```
/*****************************************************************************
 FUNCTION:     buildBMSString
 DESCRIPTION:  Builds a Battery Management System (BMS) command string with
               the information from the exeh tags listed in the BMS file.
               Requires a completely registered exeh file as well as a
               complete BMS file; all missing information will default to 0.
               dest needs to point to an array at least 495 characters long.
 INPUTS:       @param bmsNumber: which command we will be executing
               @param dest: pointer to where we will be storing the string
 RETURNS:      pointer to the completed BMS string
*****************************************************************************/
char* buildBMSString(int bmsNumber, char *dest)
```

Figure 1. Documentation for buildBMSString. *All functions' descriptions were placed before the function itself. I used the same format to document each function. The description was where I put any extra instructions needed to use the function.*

[9] Functions are a large part of programming. They can be described as small blocks of code that can take in parameters, perform operations with, without, or on those parameters, and return some value. Each function has a name and can be called anywhere that has access to that particular function. They are technically optional. However, when a set of commands is used on multiple occasions, it is best to put them in a function for easier access, readability, and bug fixing (you only have to fix the bug in one place as opposed to say twenty).

III. Writing the Test

Once I understood the code and completely documented it, I could begin testing the code. This requires a bit of work. It is not possible to just compile and run the ModelPLC code by itself; it is just a library of functions. A function needs to be called in order to be used. Normally this would happen in the main function (the default function that gets called by the operating system as soon as the program starts) but a library does not have a main.

There are a few ways to do this. One way is to use what is called an integrated test. An integrated test looks at all of the components of a project and tests how they work together. The script runs the entirety of the program and feeds it both good and bad input to see if it breaks down. Integrated tests are good for checking the overall health of a project but they are not good at finding exactly where any individual errors popped up, or even catching them in the first place. The SIM project uses a few integrated tests to get a rough idea of how healthy the project is, but the project relies on unit tests to get the specific information.

A. What Is a Unit Test and How to Make One

A unit test looks at a particular component of the project and tests its units to make sure they can handle whatever input is given. The ultimate goal is to get every single line of code exercised by the test. A unit is very loosely defined and largely up to the test designer. It could be anything from a function to a single line of code to a set of processes. For the ModelPLC code, it made more sense to test sets of processes.

In order to create the unit test, I used a tool called UnitTest++, a tool used to help programmers create unit tests in C++.

Creating tests using UnitTest++, once I learned how, was relatively easy. Before I even started I created a Fixture, which served as my initializer and de-constructer. Almost all of my tests required certain processes to happen before calling any other function and certain parts of memory to be freed upon completion. Fixture would run those defined processes in the startup of any function using Fixture and on closing free up any memory used or reset any values that were potentially changed. [10] As long as I named a given function TEST_FIXTURE and passed a single Fixture object into the function, the test would perform all of those things for me. It was a massive time saver and helped me keep my functions from accidentally depending on each other, which doesn't work out well. I could also test without Fixture (for example, if I want to specifically test one of the functions used by Fixture) by simply naming the function TEST.

```
TEST_FIXTURE(Fixture, ReadWriteBadIO)
{
    floatPacketPtr fmPtr = (floatPacketPtr)measurementAssembly[0];
    boolPacketPtr bmPtr = (boolPacketPtr)measurementAssembly[1];
    intPacketPtr imPtr = (intPacketPtr)measurementAssembly[2];

    //First, build a tag table
    buildTagTables(0, "data/goodKids");

    //Try to read/write with bad child numbers
    writeTagOutputs(-1);
    writeTagOutputs(100);
    readTagInputs(-1);
    readTagInputs(100);

    //Now try to read/write before it's registered
    writeTagOutputs(0);
    readTagInputs(0);

    //Check to make sure nothing was written
    CHECK_EQUAL(0, fmPtr->floats[1]);
    CHECK_EQUAL(0, imPtr->ints[0]);
    CHECK_EQUAL(0, bmPtr->bools[0].bit1);

    //Now very carefully set two tags (command and measurement) to be of bad type
    tagTable[0]->type = 600;
    tagTable[0]->next->type = 600;

    //Try again
    writeTagOutputs(0);
    readTagInputs(0);

    //Once again, check to make sure nothing was written
    CHECK_EQUAL(0, fmPtr->floats[1]);
    CHECK_EQUAL(0, imPtr->ints[0]);
    CHECK_EQUAL(0, bmPtr->bools[0].bit1);
}
```

Figure 2. Sample of a test function. *This is a small test function I wrote to test how certain functions dealt with bad input. In my comments I explained what I was trying to do and what I expected.*

Each one of my functions tests a unit of processes. For example, I have a test called GetAndReadCommandAssemblies, which tests whether the ModelPLC can get assemblies from SoftLogix and properly read the commands when given good input. The test uses multiple functions to do this. Most of the

[10] Freeing memory is just good design practice, but needs some explanation. As mentioned before, variables are stored in a certain place in memory. There are a few different places in memory to put variables, but only two are relevant for the moment: the heap, the stack. The stack is temporary, and any variable put on the stack is gone when the function that calls it is done. The heap is not temporary; and variable put here will stay allocated unless it is explicitly freed by the programmer. This is bad and causes memory leaks, which will be explained in Fotenote 11. Freeing memory on the heap means to de-allocate the memory so that someone else can use it.

functions used have their own test functions that treat them like mini-processes, but that particular function makes sure that the functions can work together. There is also an inverse function that tests the same process to see how it handles bad input (such as bad file names or bad info from the files). In the tests, I can use functions provided by UnitTest++ that check to make sure certain values are what I expect them to be. For example, if I have a variable called myName and I expect myName to have my name stored in it, I can write this:

```
CHECK_EQUAL("Christina Heinich",    myName);
```

This line of code will check to see if Christina Heinich is actually the value of myName. If it is, that means whatever process was supposed to set myName to Christina Heinich worked. If there is something else in myName, UnitTest++ tells me which check failed, what function it is in, and what the actual value was as opposed to the expected one. UnitTest++ also keeps a counter of all the checks that have failed as well as how many functions. These features help quickly find errors and the process that caused them.

B. Problems in Testing

The UnitTest++ tool is very useful, but there were still numerous problems. One of the largest problems was that the ModelPLC code is not modular. Modular code is code whose functions are self-contained, or in other words, do not rely on other functions or outside variables to do their job. Modular functions are clearly defined, use only variables created in the function to do its job, and clean up any side effects it may have caused.

Modular code is relatively easy to debug as the error can be easily pinpointed to a specific function. Writing a unit test alongside your code is a good way to keep a program modular as you tend to write functions to work as specific units in order to make testing simple.

This did not happen for the ModelPLC code. Almost all functions work on the side effects of other functions, and the very few that can start independently create side effects that other functions expect. Heap memory is used to keep track of the tag information passing through the code, and all functions rely heavily on global variables, variables that are declared outside of any function and can be read or changed by anyone. Both of these kinds of memory are dangerous and can easily lead to memory leaks and clobbering.[11]

Global variable clobbering made testing very problematic. Because many of the global variables are used by multiple functions, I often ran into problems where I kept getting unexpected results when testing a function that uses a specific global variable after a test of a completely different function that just happens to use the same variable. The new processes were picking up the values left over by the old tests when what I expected was a clean slate. I fixed the problem by adding to a function called freeTagTables. freeTagTables de-allocates memory from the tags. To fix my clobbering problem, I changed the function to also reset any global variables back to zero.

Memory leaks were less of a problem, as freeTagTables did a good job deleting any heap memory used by the tags. There was still an issue with the threads, which came from the thread processes not ending even after the main program had done so. That issue would normally be solved by telling the main program to wait for the threads to terminate before terminating itself, but because of the program's lack of modularity, we cannot do that. The error is still there as of the writing of this paper, but it will be fixed by the time my internship is done.

The heap memory and global variable usage did cause a lot of problems, but on the flip side many of the usages were unavoidable due to how the PLCIF interacts with other nearby components. And global variables are perfectly acceptable to use as global constants that are defined once and never changed. However, in acadamia, professors always talk about how we should never use global variables outside of constants. I finally understand why.

C. Errors Found and Fixed

Most of the changes I made to the ModelPLC code were extra checks on input and a few additions to reduce the number of dependencies certain functions have on others. While the code already had a few checks to make sure certain input files actually exist or certain numbers passed in were within a certain range, the code was still very vulnerable to bad data. I added quite a few new checks to guarantee that a function would not crash because of bad input as well as making the code skip over any invalid information while reading in a file.

---

[11] Memory leak happens when previously allocated memory can no longer be accessed by any realistic means. This happens a lot with heap memory when either the function that allocated it failed to de-allocated it before ending, leaving a box of memory with no label or pointer to it. Leaked memory cannot be de-allocated without the program having access to it, which could lead to problems if we run out of heap memory but still want to add more variable to it.

Clobbering happens when a variable is changed by one function in a way that another function does not expect.

The largest dependency I removed was discovered only because I did not know what a function (buildBMSString) was supposed to do, so I called it by itself just to see what would happen. As it turned out, the function relied on memory allocated by a function called readBMSFile, which was expected[12], and a completely different function, buildEwebTagTables, which reads in a file of Eweb tags names and creates a table. That was less expected. buildBMSString, as I found out after the program crashed, also uses the data from the Eweb tag table for the string. Because buildEwebTagTables does not provide access to the data of the tags, just the tag names, the tag table needs to be registered to get access to the data, which is an entirely different function. If buildBMSString cannot find all the tags it is looking for, it crashes. I wasn't able to untangle the stack of dependencies, but what I was able to do was add a few lines of code that checked to make sure there was actually an Eweb tag; if there wasn't, use the value 0 in its place.

D. Code Reviewing

Once I fixed all the bugs I could find, it was finally time to upload my new test and all my changes to the ModelPLC code and put it up for code review. The code review consisted of my new code being displayed on a certain web page that the reviewers (in this case my mentor and a few of the other people on the framework team) could see and make comments on. Any issues they bring up I go to my code and fix. That process gets repeated until both parties are satisfied with the product, and it gets promoted up into the main project.

There is another part of the process that was quite interesting, though. As mentioned before, the Sim group uses integrated tests to see the overall health of the project. The integrated test, firstly, just runs a mock simulation to make sure the entire thing doesn't crash, but there is more to it, too. The group uses a tool called Jenkins. Jenkins takes the information from the tests and creates a webpage to easily view the progress of the project. Through Jenkins you can see if any warnings or errors appeared, as well as the amount of code that has been covered by the unit tests. You can view the page created by Doxygen, as well as Valgrind results. Valgrind is another tool we use to catch memory leaks. It gives you an idea of where the memory leak is coming from as well as the kind of memory leak it is. There are a few other buttons and gadgets, but the ones mentioned give a very good idea of the state of the project and what needs to be done to progress.

In the case of my code review, the results were encouraging. Jenkins told us that I had covered 85 percent of the ModelPlC code, which is pretty good for the first draft. My mentor gave a ton of feedback, pointing out which of my tests were done well, and which of them could be a little more thorough or a little more streamlined. He pointed out a few places where I could add some more documentation or where my design did not follow the framework team's coding standards. Jenkins also told me where I needed to reformat my comments so Doxygen could read them. The memory leak from the threads was discovered by Valgrind, so I had to try and fix that.

For the second code review, I fixed the Doxygen warnings as well as the errors pointed out to me by the review. I also added more coverage so my coverage count went up to about 96 percent. The fix I tried for the threading error unfortunately did not work, but I came up with a different solution that hopefully fixed the problem. We will know for sure for the next code review.

Overall, the feedback I got was positive, and the rest was constructive and helped improve my test for the better.

IV. Conclusion

I end my internship providing the project with a completed unit test and a much cleaner, healthier PLCIF, but the amount of things I learned along the way was astounding. I am a better C++/C coder than I was before, and I have a better understanding of Java just from reading the GUI code. I completely understand now why global variables should not be used unless absolutely necessary and why documentation is extremely important. On a higher level, I now know what a unit test is, why it is important, and how to make one. I learned just how huge a software project can be, and just how much of software development relies on COTS. I learned about the kinds of tools like that developers use to keep their code together. I learned what a PLC is and how to use a makefile. I even learned more on how to use the Linux operating system, which will be very useful in both my personal and professional life.

On an even higher level, I learned about what NASA's mission is now that the shuttle program is over. I learned about the planned SLS rocket and the Orion capsule, as well as a much more detailed look at NASA's history. From everything I learned, I can safely say that, contrary to popular belief, NASA is not dying. It is preparing a multitude of other fantastic programs that will prove to be just as important as the ones before it.

---

[12] Just by looking at the function names it is not too hard to figure out that readBMSFile opens a Battery Management System (BMS) file and buildBMSString builds a string of characters with the information from the file.

Acknowledgments

The author thanks her string of mentors, Jason Kapusta, Lien Moore, and Cheryle Mako, for showing incredible patience and consistently finding new corners of her brain to stuff more information in. The author would also like to thank OSSI and NE-C1 for the fantastic opportunity, and sincerely hopes to come back and work in the exact desk she is sitting in for a future internship.

References

[1] Grant, Kevin. United States. NASA. "Simulation CSCI Software Design Description". 2013.[cited 24 Jul 2013].

[2] D. J Dunn, "Programmable Logic Controllers," *FreeStudy*. 24 Jul 2013. URL: http://www.freestudy.co.uk/plc/outcome1.pdf.

[3]S., Amaya, "History of C++," *cplusplus.com*[online C/C++ documentation website and community], URL: http://www.cplusplus.com/info/history/ [cited 24 Jul 2013].

[4]"Dynamic Memory," *cplusplus.co m*[online C/C++ documentation website and community], URL: http://www.cplusplus.com/doc/tutorial/dynamic/ [cited 24 Jul 2013].

[5]UnitTest++, Open Source Software, Ver. 1.4. Llopis, Noel and Nicholson, Charles, 2008.

[6]Doxygen, Open Source Software, Ver. 1.8.4 van Heesh, Dimitri, 1997.

[7]Valgrind, Open Source Software, Ver. 3.8.1. 2012.

[8]Jenkins, Open Source Software, Ver. 3.4.7. Sun Microsystems Inc., 2004.

[9]Softlogix, Software Package, Ver. 19.0. Rockwell Automation, 2011.

[10]Trick, Software Package. Johnson Space Center, Houston, TX.