Software Developer CO-OP at Curtiss-Wright Defense Solutions

***Prepared by**: Mahyar Gorji*

***Student No**.: 8135109*

Faculty of Engineering, B.A.Sc. In Software Engineering

***Course Code**: SEG3901*

***Submitted**: 07/05/2018*

Letter of Transmittal

Mahyar Gorji
4168 Owl Valley Dr
Ottawa, ON K1V 1L9
May 7th, 2018.

Daniel Amyot
800 King Edward Ave.
Ottawa, ON K1N 6N5

Dear Professor Amyot,

I have completed my first CO-OP work term after the academic term 2B. The title of this report is "Software Developer CO-OP at Curtiss-Wright Defense Solutions" and is a type 1 non-confidential report. This report has left certain specific details out when referring to project names and specifically worded software requirements for Curtiss-Wright Defense Solutions systems.

This report covers my time at Curtiss-Wright as a software developer CO-OP assigned to the newly formed, independent test team. My supervisors, Gilles Duguay and Alfredo Avila gave me tasks to complete as a member of the test team testing early access hardware and software board support packages for bugs and requirement compliance, as well as testing automation implementations.

This report was written solely by me, and has not received any previous academic credit at any academic institution, including the University of Ottawa.

Thank you,

Mahyar Gorji
8135109

[illegible]

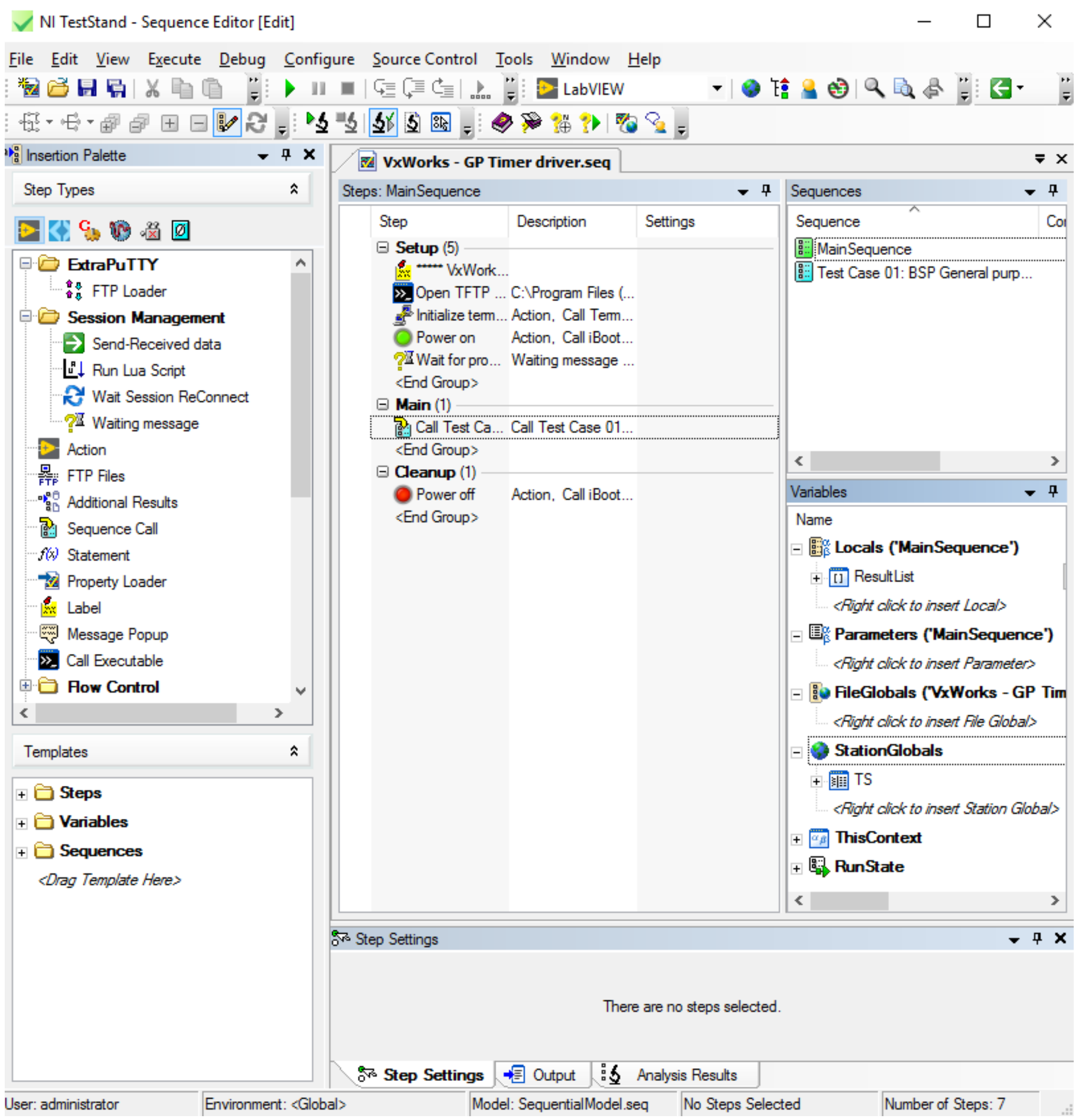
Abstract:

Curtiss-Wright Defense Solutions is an embedded computing technology company that delivers industry leading technology to its clients. Many large names employ Curtiss-Wright Defense Solutions, such as the United States Military, and commercial and private aircraft companies, with all of its products based on open-architecture standards.

During my time here, I wrote expected test case results for two types of single board computer architecture, and automated test sequences as to improve the efficiency of the test team. Being a part of the test team, we decided that the format was outdated, and should be reformatted and redesigned, to indicate the I was also exposed to a plethora of information regarding requirements, forms of testing, and software lifecycles, through a series of readings, meetings, and informative interactions with my colleagues.

Contents

1. Figures:



Step	Description	Settings
Setup (5)		
***** VxWork...		
Open TFTP ...	C:\Program Files (...	
Initialize term...	Action, Call Term...	
Power on	Action, Call iBoot...	
Wait for pro...	Waiting message ...	
<End Group>		
Main (1)		
Call Test Ca...	Call Test Case 01...	
<End Group>		
Cleanup (1)		
Power off	Action, Call iBoot...	
<End Group>		

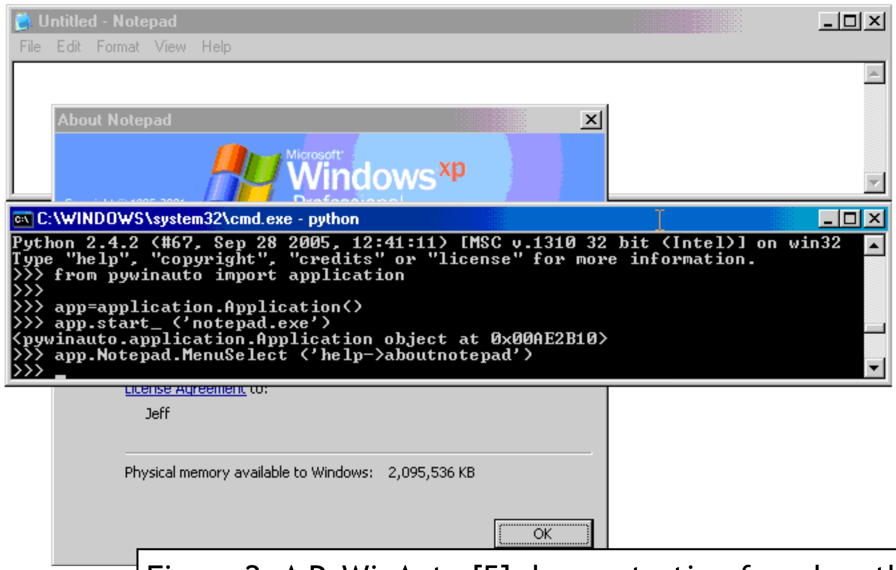
Sequence	Co
MainSequence	
Test Case 01: BSP General purp...	

Name
- Locals ('MainSequence')
  - ResultList
  - <Right click to insert Local>
- Parameters ('MainSequence')
  - <Right click to insert Parameter>
- FileGlobals ('VxWorks - GP Tim
  - <Right click to insert File Global>
- StationGlobals
  - TS
  - <Right click to insert Station Global>
- ThisContext
- RunState

There are no steps selected.

User: administrator | Environment: <Global> | Model: SequentialModel.seq | No Steps Selected | Number of Steps: 7

Figure 1: A typical TestStand sequence I designed. Opens the main communication terminal, powers the SBC on, runs tests, and powers off the SBC.


Untitled - Notepad
File Edit Format View Help
About Notepad
Microsoft Windows xp
C:\WINDOWS\system32\cmd.exe - python
License Agreement to:
Jeff
Physical memory available to Windows: 2,095,536 KB
OK

```
Python 2.4.2 (#67, Sep 28 2005, 12:41:11) [MSC v.1310 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> from pywinauto import application
>>>
>>> app=application.Application()
>>> app.start_ ('notepad.exe')
<pywinauto.application.Application object at 0x00AE2B10>
>>> app.Notepad.MenuSelect ('help->aboutnotepad')
>>> _
```

Figure 2: A PyWinAuto [5] demonstration found on their GitHub page. This example shows that notepad is being controlled via its menu bar and opens the 'about' section under the 'help' menu.

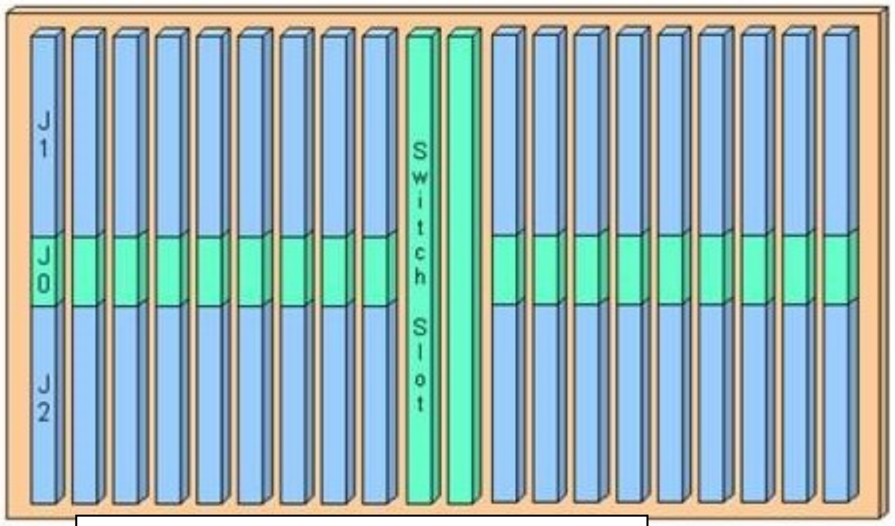


Figure 3: VME Backplane visual example

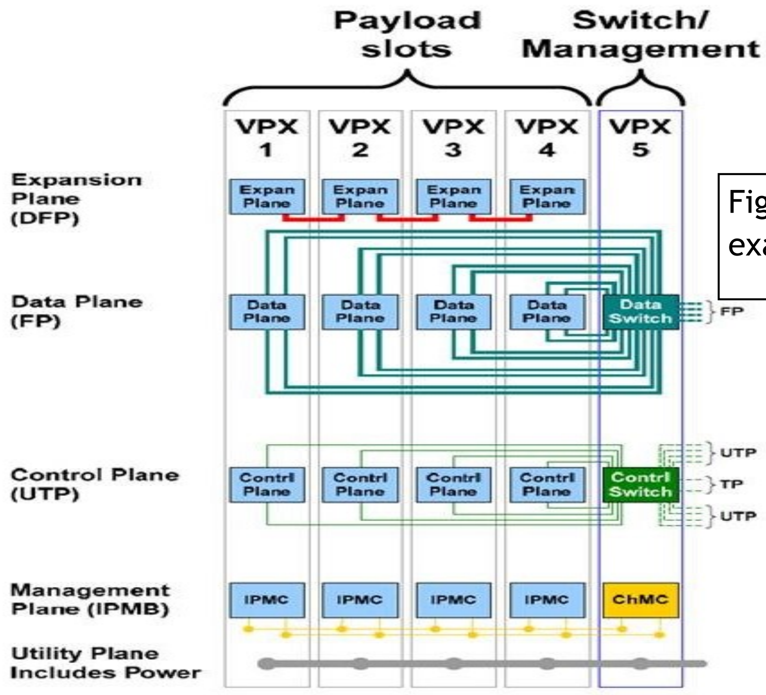
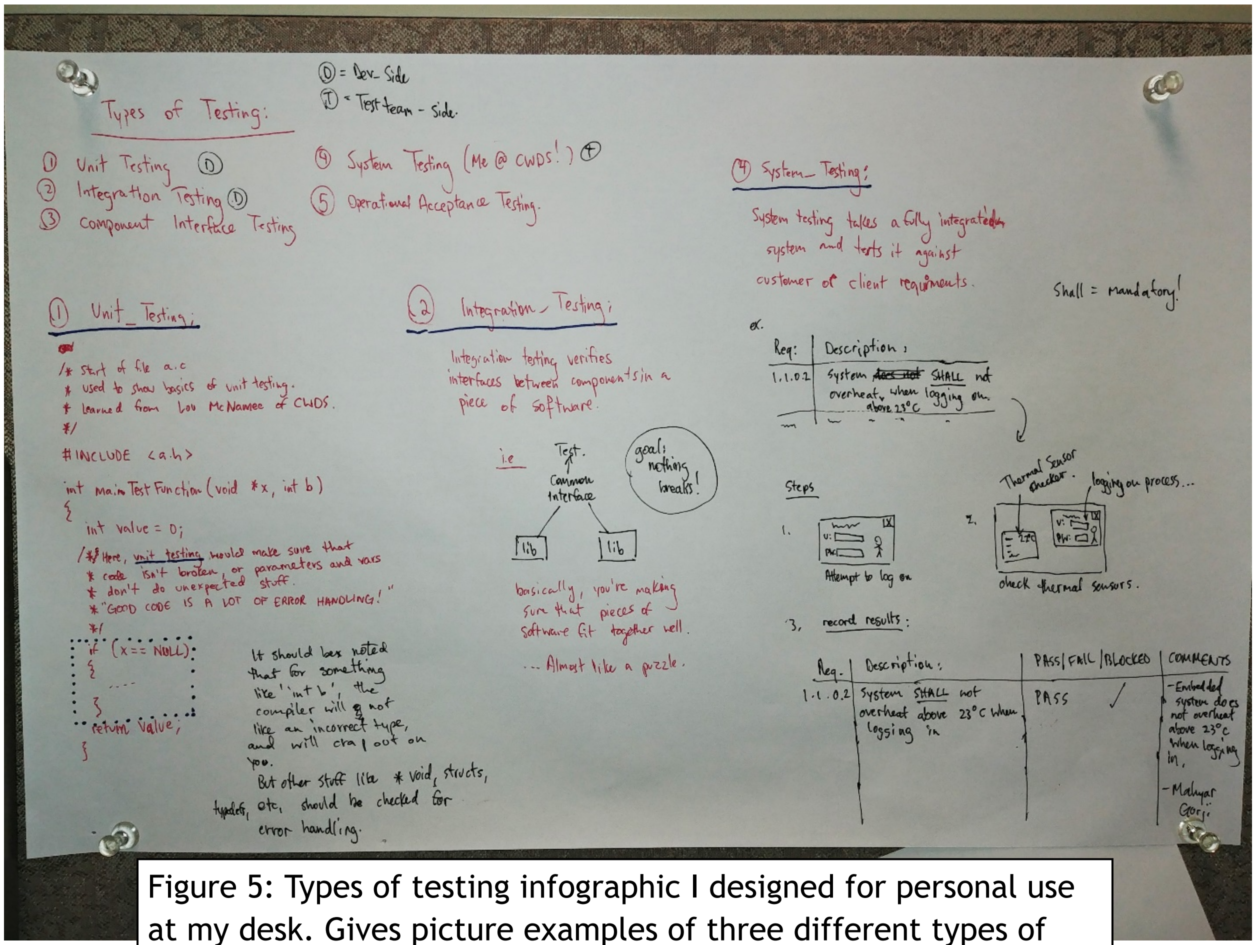


Figure 4: VPX Backplane visual example



Ⓓ = Dev_ Side
Ⓣ = Test team - Side.

Types of Testing:

① Unit Testing Ⓓ
② Integration Testing Ⓓ
③ Component Interface Testing
④ System Testing (Me @ CWDS!) Ⓣ
⑤ Operational Acceptance Testing.

① Unit_Testing;

```
/* Start of file a.c
 * used to show basics of unit testing.
 * Learned from Lou McNamee of CWDS.
 */

#INCLUDE <a.h>

int mainTestFunction(void *x, int b)
{
    int value = 0;
   /* Here, unit testing would make sure that
    * code isn't broken, or parameters and vars
    * don't do unexpected stuff.
    * "GOOD CODE IS A LOT OF ERROR HANDLING!"
    */
    if (x == NULL):
    {
       ....
    }
    return value;
}
```

It should be noted that for something like 'int b', the compiler will not like an incorrect type, and will crap out on you.
But other stuff like * void, structs, typedefs, etc, should be checked for error handling.

② Integration_Testing;

Integration testing verifies interfaces between components in a piece of software.

i.e

basically, you're making sure that pieces of software fit together well.

... Almost like a puzzle.

④ System_Testing;

System testing takes a fully integrated system and tests it against customer or client requirements.

Shall = mandatory!

ex.

Req:	Description ;
1.1.0.2	System ~~does not~~ SHALL not overheat above 23°C when logging on.

Steps

1. Attempt to log on

2. check thermal sensors.

3. record results:

Req.	Description ;	PASS/FAIL/BLOCKED	COMMENTS
1.1.0.2	System SHALL not overheat above 23°C when logging in	PASS ✓	- Embedded system does not overheat above 23°C when logging in. - Mahyar Gorji

Figure 5: Types of testing infographic I designed for personal use at my desk. Gives picture examples of three different types of testing commonly found at Curtiss Wright Defense Solutions.

2. Introduction

2.1. A Brief Context of Curtiss-Wright Defense Solutions (CWDS)

Curtiss-Wright Defense Solutions is a commercial segment of Curtiss-Wright Corporation, and is responsible for supplying industry-leading products designed for rugged deployment. The products are typically found in applications such as "aerospace, defense, and industrial applications [1]"

Clients such as The United States Military repeatedly employs CWDS for its reputable defense products. In all frontiers of exploration and combat (in the air, on the ground, on the sea, and in space), the company provides technology such as single board computers (SBCs), network switches, signal processors, control processors, propulsion systems, flight instrumentation, and more, with a very client-centric mantra.

Having facilities across North America and Europe, the company has earned global recognition with a diverse product line and technological expertise, whilst maintaining the proud legacy of its founders, Glenn Curtiss, and the Wright Brothers.

2.2. A Summary of My Time at Curtiss-Wright:

During my time at Curtiss-Wright Defense Solutions, I tested multiple VPX and VME board support packages (BSPs), with the intention of creating expected results for later releases of those same cards. These BSPs were the layers of software that allowed the boot loader and operating system to run on their single board computers (SBCs). I completed this task and gained a sense of familiarity and completion, as well as documentation for future tests and test team staff such as to lower the learning curve on embedded computing.

I also assisted in automation of test cases through National Instruments' TestStand. This was an option explored to further tester efficiency, provide more accurate results (as humans are often not as accurate as a calibrated machine), and reduce further test cost.

3. Projects undertaken:

3.1. Testing and gathering expected results according to requirements:

As a Software Engineering CO-OP, I was tasked with providing support to a newly formed testing team, which was comprised of Alfredo, a senior test engineer, and myself. Through Software Test Plans (STPs), the team expected me to review and test software releases, whether that product was a software release on an existing SBC or a combination of pre-released hardware and software.

It is during the first few days of my hiring that Alfredo informed me that we were bringing reform to standard test procedures that CWDS developers had implemented before. One such change was the implementation of testing software releases with software requirements in mind, as opposed to testing for functionality only.

To explain testing for requirements, the understanding of the substance of a requirement is necessary. Software requirements “are descriptions of features and functionalities of the target system. Requirements convey the expectations of users from the software product [2]”. Clients may produce requirements as either a recommendation or a demand, using different wording to do so, while simultaneously providing the appropriate level of importance for the requirement.

A demand requirement (known as a “shall” requirement) lists functions of the software that are of vital importance to be included in the project, and cannot be subject to neglect. An example of such a requirement is as follows, pertaining to the hypothetical example of producing a water bottle:
*The water bottle **SHALL** be constructed with red BPA-free plastic.*
The usage of the word ‘shall’ denotes that, were a blue water bottle constructed with trace amounts of BPA in its plastic, such a product would be unacceptable and will cause a noncompliance with the requirements document.

A recommended requirement (known as a ‘should’ requirement) “denotes a guideline or recommendation whenever noncompliance with the specification is permissible [3]”. In contrast to the above ‘shall’ requirement, this is the equivalent ‘should’ requirement:
*The water bottle **SHOULD** be constructed with a red BPA-free plastic.*
The usage of the word should acts as a recommendation to the bottle’s construction material, and should be followed, but provides the ability for the manufacturer to make a judgement call, should there not be enough red plastic, for example. If the water bottle was a blue colored BPA-free plastic, the manufacturer has made a reasonable production decision that ensures compliance with the requirements document.

With this new understanding of requirements, Alfredo tasked me with familiarizing myself with the test cases and looking over the requirements that he had been adding into the STPs. He rationalized it as both a learning experience and an in-case situation where I could add in missing requirements from the requirements document. In familiarizing myself with these procedures, I understood how the test cases were manipulated to provide a clear understanding of whether the requirement had been met, rather than simply verify its functionality only.

Shortly after gaining familiarity with these procedures that Alfredo had been implementing at CWDS, I was tasked with the creation of expected results for the test procedures. Before Alfredo began implementing these changes, the STPs were a set of instructions that told the tester what to do, but had a steeper learning curve associated to it. Alfredo noted that these tests should and will have expected results to notify a tester that they have passed the test case and met the associated requirement, whereas previously, the pass/fail criterion was ambiguous. This, combined with the resulting near-elimination of the learning curve, motivated me to provide as much documentation as possible to this cause.

To add in these results, I began testing a release of a VPX SBC (see subsection 3.3) that was not applicable to my official progress in the test round, but instead gave me results that I could expect to see out of the official early access unit (EAU). Having run through every test procedure to achieve this goal, I began developing a more thorough understanding of the product.

3.2. Automated Testing:

Throughout testing, the question of efficiency was always present. Even if a member of the test team had enough experience to complete the test procedures and create documentation quickly, eventually that speed of completion would plateau. It was here that Alfredo and I explored the idea of automating our test environments, such that we could finish tests as quickly as the hardware limitations could allow it, instead of being held back by human limitations.

It was at this moment in my work term that Alfredo and I branched out to explore the possibilities, whilst also continuing work on the test procedures in anticipation of the EAU SBC. I explored possibilities using scripting languages, such as a combination of Python and Tera Term Language (TTL). This option required different third-party Python packages, such as "PySerial [4]"from Python's package index, and implored the idea of reading serial output from the SBC into a Python script, where the script would take this input and act accordingly (ex. Send a command in response, shut down the card, call another TTL file on the computer, etcetera). Ultimately, this proved to be achievable, but simultaneously messy and far from maintainable, as this required editing of all of the involved scripts when moving from one SBC to another, one computer to another, or a combination of the two.

The other option I explored was the manipulation of the user interface (UI) that we used to communicate with the SBC. This was also achievable through Python through an external package called "PyWinAuto [5]", along with its other dependencies. The premise was to automate the selection of the menu bar beneath the application icon to launch other TTL macros from Tera Term (*see figure 2)*. Unfortunately, I ran into an issue where Tera Term UI did not support the automated selection of its menu bar through PyWinAuto, thus preventing me from taking this option any further.

While I was continuing to search for an answer using Python, Alfredo began experimenting with "TestStand [6]", by National Instruments. Alfredo had had prior experience with the software and, as a staple of the automated testing world; he believed that this software could fit our needs. TestStand also had plugins for other software and had a function that allowed the user to set global variables to their work station only, such that the code in TestStand could be distributed within the company with very little editing involved.

After a week of exploring different possibilities to automate our test procedures, Alfredo and I met and discussed our findings. With the knowledge that my options were both unmaintainable, and/or not possible, we decided to move ahead with TestStand. We quickly ran into an issue where, for each individual license of TestStand, the cost ran into the thousands per annum. Knowing that such a high cost was an obstacle to the company obtaining licenses for us, we instead downloaded the trial versions and attempted to automate as many STPs as we could in the short forty-five day trial, with the intention of impressing the product development managers and potentially obtaining full licenses.

This decision had the most favorable outcome, in the end, as the automated sequences proved to be, with some time, relatively easy to create and mold to the shape of a software test plan (*See figure 1*). These tests would execute commands with milliseconds of delay such that the commands do not interrupt each other, and could call arguments in any script it opened, even allowing us to turn the SBC on and off using an internal network boot device.

3.3. SBCs:

The SBCs at CWDS use two common backplane architectures, the VME, and the VPX.

The VME architecture offered a common framework to companies looking to create "interoperable computing systems [7]", where typical components of this system include "boards such as processors, IO boards, etc., as well as enclosures, backplanes, power supplies, and other subcomponents [7]". VME is a system based on a parallel bus, where digital signals are connected together in a linear series. Boards, using this backplane may communicate openly and freely with one another along the bus (*see figure 3*). This may cause problems as a bussed backplane can cause delays and is inherently slower than VPX architecture.

While the VME was a bus-based architecture, the VPX is an architecture based on switch fabrics. Signals in a VPX architecture do not travel linearly, like the VME, and allows "data rates to go much higher and are inherently much more reliable [7]". On a VPX backplane, all boards are able to communicate to a switch board (typically in the last slot on a VPX chassis, *see figure 4*). The data planes and control planes are connected to their respective switches on a switch board and all function simultaneously thanks to this architecture. Boards using this backplane architecture, however, cannot communicate directly to one another unless it is through the expansion plane (typically done with PCIe), but even then, can still only communicate to its direct neighbor.

VPX cards can come in either 3U (smaller) or 6U (larger) sizes, whereas VME cards are generally 6U size.

I found myself working with both VPX and VME SBCs throughout the course of my work term and found it much simpler working with the VME boards, as I had much less trouble finding available hardware to test with, which led to a higher sense of productivity. With many cards with the VPX backplane architecture, I found myself struggling to find a complete set that consisted of the board, appropriately sized chassis (backplane could support 6U), and rear-transmission module (RTM) required to function.

To circumvent this setback, when I had advanced knowledge of needing to work with a VPX board, I would hold onto the parts I needed as they became available before work on that type of card begun. As a direct result, I would have all the pieces I needed to improve my productivity after being assigned a VPX-6U board and task.

4. What I have learned on the job:

4.1. Writing a well-written requirement

When development and testing are based on requirements, the requirement must be well written and complete. Alfredo took time to educate me on how to write a complete requirement, using the S.M.A.R.T acronym: specific, measurable, attainable, realizable, and traceable.

When looking through the requirements documents for the projects I had a hand in, Alfredo and I took notes on any requirements that did not meet all five components of a well written requirement. One requirement in particular was a mandatory inclusion of the existence of ‘a pulse’ on the oscilloscope during normal function. However, this requirement did not tell us the amplitude of the pulse, which was noncompliant with the ‘specific’ and ‘measurable’ principles.

As a result, Alfredo and I mentioned this specific requirement in a software meeting, which sparked a discussion on requirements of requirements. This discussion was beneficial to my learning, in that I understood better what the real world requirements for requirements are, versus the theory (ex. S.M.A.R.T). Some colleagues argued that the requirement mentioned above has implied pass/fail criteria, which Alfredo disagreed with.

Lou, a Software Engineering Manager argued Alfredo’s point, referring to how a requirement is the beginning of any development, and how the requirement (if not written properly) can cause ‘technical debt’.

Technical debt, according to Techopedia, “is a concept in programming that reflects the extra development work that arises when code that is easy to implement in the short run is used instead of applying the best overall solution [8]”. Here, ‘code’ is interchangeable with ‘requirements’, in that if the proper amount of time is not allocated to reviewing and writing well-written requirements, more work will be created later in the development life-cycle, which will inevitably cost more money.

I feel that, by experiencing the technical debt created by some requirements I have read on the job, as well as the guidelines as to how one must write requirements, I am better prepared as a Software Engineering student to write quality software.

4.2. Forms of Testing

While my term at Curtiss-Wright Defense Solutions assigned me to 'testing', it was not classified what type of testing I would be doing daily. I began, however, to learn and understand the different types and approaches to testing that existed within CWDS.

From my understanding, CWDS takes the time to incorporate three types of testing into their environment: Unit Testing, Integration Testing, and System Testing.

Unit testing is done to ensure that code and is not broken, or parameters and variables do not behave unexpectedly. For instance, on a variable, this is achieved by type-checks, NULL-checks, pointer-checks, mathematical-order-checks (ex. Order of operations, and enclosing brackets to isolate a given part of an equation), etcetera. Unit testing is also beneficial, in that some errors will not cause a compiler error, and thus may be harder to catch until something important breaks.

The developers at CWDS create several unit tests and provide error handling for a large majority of their code. They stress that well written code must adhere to standards and guidelines, but must also consist of a large amount of error handling.

Integration testing ensures that interfaces between components in a piece of software work together well. It might consist of a common interface calling different libraries to verify that inter-library interactions will not break the software. Developers at CWDS will often do integration testing while they unit test.

Lastly, system testing is the act of taking a fully integrated system to test it against customer requirements. From my experiences at CWDS, the test team is responsible for an unbiased system testing. The pass/fail criteria strictly adheres to the requirements here, as this testing is typically done last. Failures will not necessarily result in a system error and a crash, but will cause a non-compliance with the requirements that must be fixed.

Alternatives to leaving system testing for the test team would be to let developers system test as well as unit test and integration test, but it is a poor alternative because developers tend to be (sometimes even minimally) biased towards their own code. This may cause a lack of depth when testing against customer requirements, and will slow down development of other projects in the meantime. As a result, the test team is able to provide a more in-depth scan of any issues or bugs related to the functionality, and/or requirements.

This relates to both my academic career and my professional career. Academically, I will be attending the uOttawa class "Software Quality Assurance" at the end of my term here at Curtiss-Wright. I believe that, due to the practical experience, and knowledge I have obtained regarding requirements testing, and about different forms of testing and validation, I will have an easier time understanding the in-class concepts.

As for my professional career, I have noted I can develop code with respect to requirements and different forms of tests. This way, I will have code that is easier to read and understand. This is achieved through developing and adhering to my own or coding standards. That is, to be commenting on code, creating well-named variables, and providing expected results as to what my code shall achieve, and strive to meet those expectations.

4.3. Software Lifecycle

At a monthly software engineering meeting conducted at the CWDS training room, Lou lectured us about software life cycles, as well as the different methodologies and potential pitfalls behind different types of development.

Commonly noted examples were of the waterfall method, where requirements 'trickle-down' into design, implementation, verification, and finally maintenance. Lou noted that, while waterfall was great for activities with hard dependencies, it is a method that most software developers would prefer to avoid. This is due to multiple reasons: defects cost exponentially more the further down the waterfall they travel without being caught, there are no inherent risk mechanisms, and it is more management-driven than developer-driven.

My colleagues at the meeting noted that the developers of the company align more with the iterative lifecycle strategy, where planning, design, testing, deployment, and evaluation is cyclical in nature. Critiques of the lifecycle strategy were noted to be that you have better learning opportunities from your mistakes, development is broken into repeating cycles, and it does not share the same magnitude of defect cost as traditional waterfall.

While other strategies were discussed at that meeting, the common theme of a lifecycle is to be wary of cost-of-defects, task size, and project focus (whether it is customer/developer, or management-based). I have found myself, in an educational environment, using a 'watered-down' variant of the iterative method, where I will code according to classroom requirements, design, test, and cycle until ready for submission (deployment). In past encounters with assignments, I have found myself regretting my choice of lifecycle development, but after witnessing it done in a professional setting here at Curtiss-Wright, I understand the importance of cyclical development and how learning from your mistakes is the key take-away of developing in that fashion.

5. In Conclusion:

5.1. Critiques:

5.1.1. For The Company:

The CO-OP experience at CWDS was a very welcome one. Perhaps the most appealing part of working for the company was that my colleagues (both full-time and other CO-OPs) took me seriously, and treated me with equal importance to other workers.

It was also evident that Curtiss-Wright Defense Solutions values the education I receive while working. Every time I walked into another cubicle or office, I left with the knowledge I needed to complete my assignment, and more. I would often leave with multiple pages of hand-written notes detailing the discussion I had with my colleagues, as well as any new knowledge imparted onto me.

Another positive experience I had was during the CO-OP lunch, organized by the CWDS Human Resources department. Coming into this work term late, I attempted to build my internal network as fast as possible, but the CO-OP lunch cemented these connections and I found it easier afterwards to strike up conversations with all of them.

5.1.2. For The CO-OP Office:

The CO-OP office was exceptional in all areas: the organization of interviews, the speed at which they replied to my emails and phone calls, and their ability to plan in the case of an emergency. Also, the mock interviews were very close to the interviews I experienced with companies during the interview phase, and even here at Curtiss-Wright. The amount of work put in by the personal development specialists in the office shows in how streamlined the process is for their student applicants. In all domains, the office was professional and sympathetic, and, without a doubt, had my best interests at heart.

5.1.3. For Software Engineering at uOttawa:

I genuinely regret not attending a work term before a select few classes I enrolled in at the University of Ottawa. The hands-on experience in such a demanding field such as Software Engineering really allows the student to prosper and mix a combination of practical skills with theory. A suggestion I could provide would be to offer more classes such as ‘Introduction to Product Development and Management’ (GNG2101), where a student is assessed on their fundamentals of engineering, ability to combine a hands-on experience with their theory from other classes, and inter-personal skills. Working at Curtiss-Wright, I realize now how important these skills are to becoming a quality engineer, and I believe that the University owes it to their students to provide more experiences such as that one throughout their career.

5.2. Overall

The learning experience at Curtiss-Wright Defense Solutions was positive, and allowed me to develop an appreciation for the world of testing and automation. Program functionality is important, but striking a balance between program functionality and client requirements is even more so. I can differentiate between the VPX and VME backplane architectures, as well as their core characteristics, and I now have generous testing knowledge I can apply to my own code in a classroom and a professional setting, so that I can be a software engineer of higher quality.

Works Cited [IEEE]:

1. Curtisswrightds.com. (2018). *About Curtiss-Wright | Highly Engineered COTS Modules*. [online] Available at: https://www.curtisswrightds.com/company/ [Accessed 13 Apr. 2018].

2. Curtisswrightds.com. (2018). *About Curtiss-Wright | Highly Engineered COTS Modules*. [online] Available at: https://www.curtisswrightds.com/company/ [Accessed 13 Apr. 2018].

3. "Definition: should," Webster's Online Dictionary, http://www.websters-online-dictionary.org/definitions/should?cx=partner-pub-0939450753529744%3Av0qd01-tdlq&cof=FORID%3A9&ie=UTF-8&q=should&sa=Search#922.

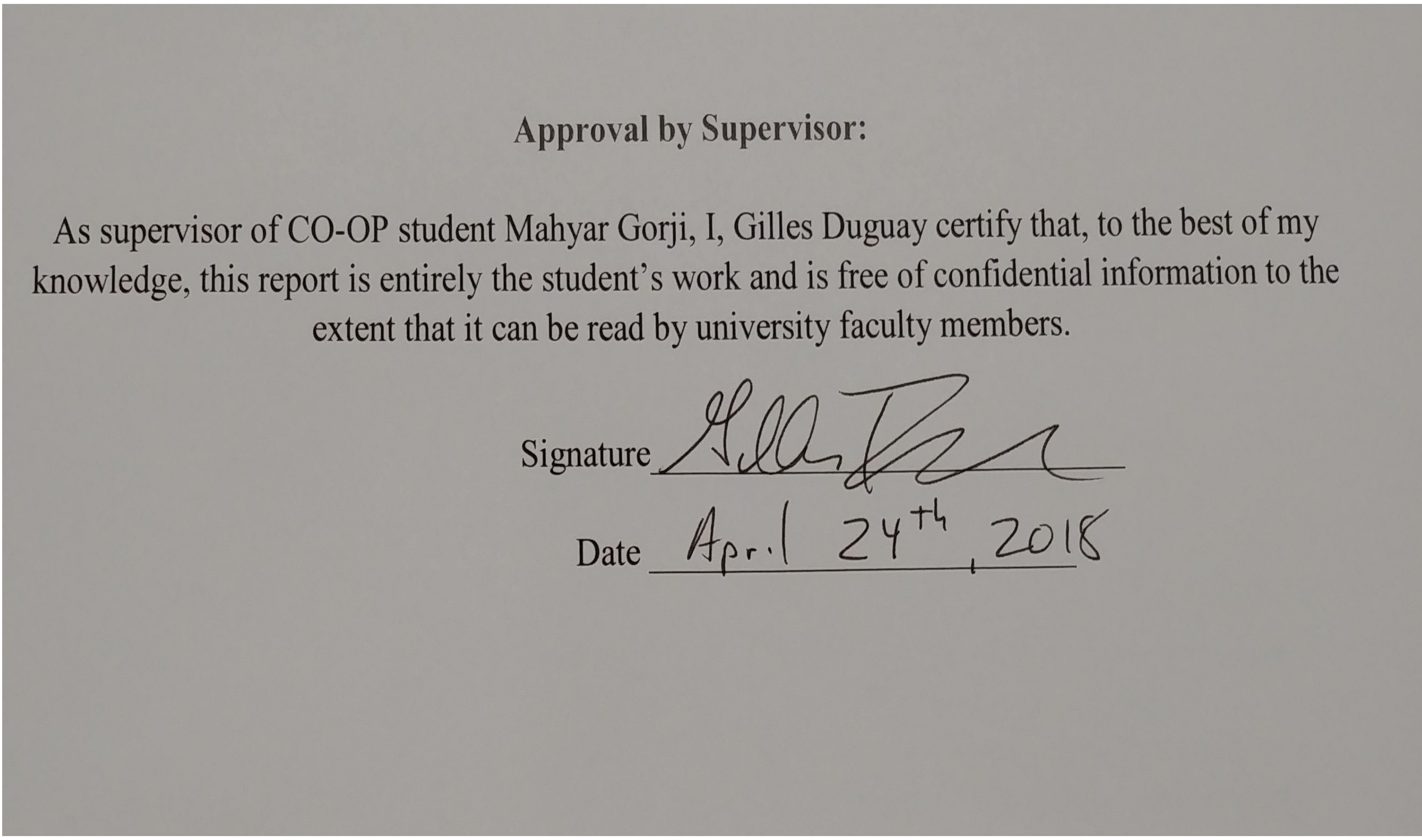
4. "pySerial," *GitHub*. [Online]. Available: https://github.com/pyserial/. [Accessed: 13-Apr-2018].

5. M. M. Mahon and V. Ryabov, "Pywinauto," *pywinauto*. [Online]. Available: http://pywinauto.github.io/. [Accessed: 13-Apr-2018].

6. "TestStand," *TestStand - National Instruments*. [Online]. Available: http://www.ni.com/teststand/. [Accessed: 13-Apr-2018].

7. J. Moll, "What's the Difference Between VME and VPX?," *Electronic Design*, 05-Apr-2016. [Online]. Available: http://www.electronicdesign.com/boards/what-s-difference-between-vme-and-vpx. [Accessed: 13-Apr-2018].

8. "What is Technical Debt? - Definition from Techopedia," *Techopedia.com*. [Online]. Available: https://www.techopedia.com/definition/27913/technical-debt. [Accessed: 16-Apr-2018].



Approval by Supervisor:

As supervisor of CO-OP student Mahyar Gorji, I, Gilles Duguay certify that, to the best of my knowledge, this report is entirely the student's work and is free of confidential information to the extent that it can be read by university faculty members.

Signature [illegible]

Date April 24th, 2018